SQL Anywhere® サーバ
SQL の使用法

バージョン 16.0

2013 年 2 月

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

バージョン 16.0
2013 年 2 月

© 2013 SAP AG or an SAP affiliate company.All rights reserved.

次の条件に従うかぎり、このマニュアルの全部または一部を使用、印刷、再生、配布することができます。1) マニュアルの全部または一部にかかわらず、すべてのコピーにこの情報またはマニュアル内のその他の版権と商標の表示を含めること。2) マニュアルに変更を加えないこと。3) SAP 以外の人間がマニュアルの著者または情報源であるかのように示す行為をしないこと。ここに記載された情報は事前の通知なしに変更されることがあります。

SAP AG およびディストリビュータが販売しているソフトウェア製品には、他のソフトウェアベンダー独自のソフトウェアコンポーネントが含まれているものがあります。国内製品の仕様は変わることがあります。

これらの資料は SAP AG および関連会社 (SAP グループ) が情報のみを目的として提供するものであり、いかなる種類の表明または保証も行うものではなく、SAP グループはこの資料に関する誤りまたは脱落について責任を負わないものとします。SAP グループの製品およびサービスに関する保証は、かかる製品およびサービスに付属している明確な保証文書がある場合、そこで明記されている保証に限定されます。ここに記載されているいかなる内容も、追加保証を構成するものとして解釈されるものではありません。

ここに記載された SAP および他の SAP 製品とサービス、ならびに対応するロゴは、ドイツおよび他の国における SAP AG の商標または登録商標です。http://www.sap.com/corporate-en/legal/copyright/index.epx#trademark を参照してください。

目次

はじめに ix

テーブル、ビュー、インデックス 1

データベースオブジェクトの名前とプレフィクス 1
システムオブジェクトのリストの表示 (Sybase Central の場合) 3
システムオブジェクトのリストの表示 (SQL の場合) 3
テーブル 4
テンポラリテーブル 10
計算カラム 13
プライマリキー 17
外部キー 20
インデックス 26
ビュー 39
マテリアライズドビュー 53

ストアドプロシージャ、トリガ、バッチ、ユーザ定義関数 77

プロシージャ、トリガ、およびユーザ定義関数の利点 77
プロシージャ 78
ユーザ定義関数 89
トリガ 94
バッチ 107
プロシージャ、トリガ、ユーザ定義関数の構造 110
制御文 112
結果セット 115
プロシージャ、トリガ、ユーザ定義関数、バッチのカーソル 121
エラーと警告の処理 125
プロシージャ、トリガ、ユーザ定義関数、バッチで使用される EXECUTE IMMEDIATE 135
プロシージャ、トリガ、ユーザ定義関数でのトランザクションとセーブポイント 138
プロシージャ、トリガ、ユーザ定義関数、バッチを作成するときのヒント 138
プロシージャ、トリガ、イベント、バッチで使用できる文 140

プロシージャ、ファンクション、トリガ、イベント、またはビューの内容を隠す 141

パフォーマンスの向上、診断、モニタリング 145

パフォーマンスのモニタリングと診断のツール 145
パフォーマンス向上のヒント 211
アプリケーションプロファイリングのチュートリアル 255

クエリとデータ修正 281

クエリ 281
全文検索 367
クエリ結果の要約、グループ化、ソート 468
ジョイン：複数テーブルからのデータ検索 488
共通テーブル式 532
OLAP のサポート 547
サブクエリの使用 594
データ操作文 617

SQL のダイアレクトと互換性 633

SQL Flagger を使用した SQL 準拠のテスト 633
他の SQL ソフトウェアとは異なる SQL Anywhere の機能 635
Watcom SQL 639
Transact-SQL との互換性 640
Adaptive Server Enterprise のアーキテクチャ 642
Transact-SQL と互換性のあるデータベース 647
互換性のある SQL 文 654
Transact-SQL のプロシージャ言語 660
ストアドプロシージャの自動変換 662
Transact-SQL プロシージャから返される結果セット 663
Transact-SQL プロシージャの中の変数 663
Transact-SQL プロシージャでのエラー処理 664

データベースにおける XML 667

リレーショナルデータベースにおける XML 文書の格納 667

XML としてエクスポートされるリレーショナルデータ 668
XML 文書をリレーショナルデータとしてインポートする方法 669
XML としてのクエリ結果 676
結果を表示するための Interactive SQL の使用 694
クエリ結果を XML として取得するための SQL/XML の使用 695

データベース内の JSON 703

クエリ結果を JSON として取り出すための FOR JSON 句の使用 703
FOR JSON RAW 703
FOR JSON AUTO 704
FOR JSON EXPLICIT 705

データのインポートとエクスポート 709

バルクオペレーションのパフォーマンスの側面 709
バルクオペレーションのデータリカバリの問題 710
データインポート 711
データエクスポート 730
クライアントコンピュータ上のデータへのアクセス 747
データベースの再構築 749
データベース抽出 760
SQL Anywhere へのデータベース移行 761
SQL スクリプトファイル 766
Adaptive Server Enterprise の互換性 770

リモートデータアクセス 771

リモートテーブルのマッピング 772
リモートサーバ 772
ディレクトリアクセスサーバ 781
外部ログイン 789
プロキシテーブル 790
ネイティブ文とリモートサーバ 798
リモートプロシージャコール (RPC) 799
トランザクションの管理とリモートデータ 802

内部オペレーション 803
リモートデータアクセスのトラブルシューティング 807
リモートデータアクセスのサーバクラス 808

データ整合性 833

データが有効でなくなる状況 833
整合性制約 833
データベースの内容が変更される状況 834
データの整合性を維持するためのツール 834
整合性制約を実装するための SQL 文 835
カラムデフォルト 836
テーブルとカラム制約 843
ドメイン 848
エンティティ整合性と参照整合性 850
システムテーブルの整合性ルール 859

トランザクションと独立性レベル 861

トランザクション 861
同時実行性 863
トランザクション内のセーブポイント 864
独立性レベルと一貫性 864
トランザクションのブロックとデッドロック 881
ロックの仕組み 886
独立性レベル選択のガイドライン 902
独立性レベルのチュートリアル 906
プライマリキーの生成と同時実行性 927
データ定義文と同時実行性 932
まとめ 932

LDAP ユーザ認証 935

LDAP ユーザ認証環境の作成 (SQL の場合) 936
LDAP ユーザ認証環境の作成 (Sybase Central の場合) 939

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

SQL Anywhere のデバッガ 947

Java デバッガの稼働条件 947
チュートリアル : デバッガの使用開始 948
ブレークポイント 953
変数 957
接続デバッグ 959

索引 961

はじめに

このマニュアルでは、データベースへのオブジェクトの追加方法、データのインポート、エクスポートおよび変更方法、データの検索方法、ストアドプロシージャとトリガの構築方法について説明します。

テーブル、ビュー、インデックス

データベースオブジェクトを作成、変更、削除する SQL 文は、「データ定義言語」 (DDL) と呼ばれます。データベースオブジェクトの定義は、データベーススキーマを形成します。スキーマは、データベースの論理的なフレームワークです。

参照

- 「ストアドプロシージャ、トリガ、バッチ、ユーザ定義関数」77 ページ
- 「データ整合性」833 ページ
- 「データベースの作成」『SQL Anywhere サーバ データベース管理』
- 「Sybase Central」『SQL Anywhere サーバ データベース管理』
- 「Interactive SQL」『SQL Anywhere サーバ データベース管理』

データベースオブジェクトの名前とプレフィクス

データベースオブジェクトの名前は識別子である必要があります。このマニュアルの例では、サンプルデータベースのデータベースオブジェクトは、通常その識別子のみを使用して参照されます。次に例を示します。

```
SELECT * FROM Employees;
```

テーブル、プロシージャ、ビューにはすべて所有者が存在します。GROUPO ユーザは、サンプルデータベースのサンプルテーブルを所有しています。場合によっては、オブジェクト名に所有者のユーザ ID をプレフィクスとして付ける必要があります。次に例を示します。

```
SELECT * FROM GROUPO.Employees;
```

Employees テーブルの参照は「修飾」されています。単にオブジェクト名を示すだけでよい場合もあります。この項では、どのような場合にテーブル、ビュー、プロシージャに所有者名をプレフィクスとして付ける必要があるかを説明します。

データベースオブジェクトを参照するときプレフィクスが必要ないのは、次の場合です。

- 自分がデータベースオブジェクトの所有者である場合。
- 自分が付与したロールの誰かがデータベースオブジェクトを所有している場合。

例

Acme 社のコーポレートデータベースの次の例を検討します。ユーザ ID Asmin は、データベースに対するすべての管理権限とともに作成されます。他の 2 つのユーザ ID である Joe と Sally は、販売部で働く従業員用に作成されます。

```
CREATE USER Admin IDENTIFIED BY secret;
GRANT ROLE SYS_AUTH_DBA_ROLE TO Admin;
CREATE USER Sally IDENTIFIED BY xxxxx;
CREATE USER Joe IDENTIFIED BY xxxxx;
```

Admin ユーザは、データベースにテーブルを作成して、Acme ロールに所有者を割り当てます。

```
CREATE ROLE Acme;
CREATE TABLE Acme.Customers ( ... );
CREATE TABLE Acme.Products ( ... );
CREATE TABLE Acme.Orders ( ... );
CREATE TABLE Acme.Invoices ( ... );
CREATE TABLE Acme.Employees ( ... );
CREATE TABLE Acme.Salaries ( ... );
```

会社の全員がすべての情報にアクセスできるようにはしません。販売部で働く Joe と Sally には、Customers、Products、Orders テーブルが必要ですが、その他のテーブルは不要です。これを実行するには、SalesForce ロールを作成して、制限されたテーブルのセットへのアクセスに必要な権限と、この 2 人の従業員にこのロールを割り当てます。

```
CREATE ROLE SalesForce;
GRANT ALL ON Acme.Customers TO SalesForce;
GRANT ALL ON Acme.Orders TO SalesForce;
GRANT SELECT ON Acme.Products TO SalesForce;
GRANT ROLE SalesForce TO Sally;
GRANT ROLE SalesForce TO Joe;
```

Joe と Sally はこれらのテーブルを使用するのに必要な権限を持っていますが、テーブルの所有者が Acme であるため、テーブル参照の権限も必要です。

```
SELECT * FROM Acme.Customers;
```

この状況を変更するには、Acme ロールを Sales ロールに付与します。

```
GRANT ROLE Acme TO SalesForce;
```

Sales ロールが付与されている Joe と Sally に Acme ロールが間接的に付与され、識別子なしでテーブルを参照できるようになりました。SELECT 文は、次のように簡略化できます。

```
SELECT * FROM Customers;
```

注意
Acme ユーザ定義ロールは、オブジェクトレベルの権限を参照しません。このロールでは、ユーザは所有者の修飾なしでロールが所有しているオブジェクトを参照できます。Joe と Sally は Acme ロールを持っているため、それ以外の権限は持っていません。Acme ロールには、特別な権限は明示的に付与されていません。Admin ユーザは、テーブルを作成して SYS_AUTH_DBA_ROLE システムロールを持っているため、Salaries のようなテーブルを表示する暗黙的な権限を持っています。したがって、Joe と Sally が次のいずれかの文を実行するとエラーになります。

```
SELECT * FROM Acme.Salaries;
SELECT * FROM Salaries;
```

どちらの場合も、Joe と Sally は Salaries テーブルを参照するのに必要な権限を持っていません。

参照

- 「グループ」『SQL Anywhere サーバ データベース管理』

システムオブジェクトのリストの表示 (Sybase Central の場合)

Sybase Central を使用して、システムテーブル、システムビュー、格納されたプロシージャ、ドメインなどシステムオブジェクトに関する情報を表示します。

前提条件

この作業を実行するための前提条件はありません。

内容と備考

この作業を行うのは、データベース内のシステムオブジェクトのリストとその定義を見るとき、または定義を使用して似たような他のオブジェクトを作成するときです。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. データベースを選択し、**[ファイル]** » **[所有者フィルタの設定]** をクリックします。
3. **[SYS]** と **[dbo]** を選択して、**[OK]** をクリックします。

結果

Sybase Central のシステムオブジェクトのリストが表示されます。

参照

- 「SYSOBJECT システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSTAB システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSUSER システムビュー」『SQL Anywhere サーバ SQL リファレンス』

システムオブジェクトのリストの表示 (SQL の場合)

SYSOBJECT システムビューのクエリを行って、システムテーブル、システムビュー、格納されたプロシージャ、ドメインなどシステムオブジェクトに関する情報を表示します。

前提条件

この作業を実行するための前提条件はありません。

内容と備考

この作業を行うのは、データベース内のシステムオブジェクトのリストとその定義を見るとき、または定義を使用して似たような他のオブジェクトを作成するときです。

♦ タスク

1. Interactive SQL では、データベースに接続します。
2. SELECT 文を実行し、SYSOBJECT システムビューに対してオブジェクトのリストを問い合わせます。

結果

Interactive SQL のシステムオブジェクトのリストが表示されます。

例

次の SELECT 文は、SYSOBJECT システムビューに対してクエリを行い、SYS と dbo が所有するすべてのテーブルとビューのリストを返します。SYSTAB システムビューに対してジョインを行ってオブジェクト名を返し、SYSUSER システムビューに対してジョインを行って所有者名を返します。

```
SELECT b.table_name "Object Name",
     c.user_name "Owner",
     b.object_id "ID",
     a.object_type "Type",
     a.status "Status"
  FROM ( SYSOBJECT a JOIN SYSTAB b
        ON a.object_id = b.object_id )
  JOIN SYSUSER c
WHERE c.user_name = 'SYS'
   OR c.user_name = 'dbo'
ORDER BY c.user_name, b.table_name;
```

参照

- 「SYSOBJECT システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSTAB システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSUSER システムビュー」『SQL Anywhere サーバ SQL リファレンス』

テーブル

データベースを初めて作成した場合、そのデータベースにあるテーブルはシステムテーブルだけです。システムテーブルにはデータベースのスキーマが保管されます。

必要に応じてデータベーススキーマを簡単に再作成するには、SQL スクリプトファイルを作成してデータベースのテーブルを定義します。SQL スクリプトファイルには、CREATETABLE 文と ALTERTABLE 文を含めてください。

参照

- 「ユーザ拡張ロール」『SQL Anywhere サーバ データベース管理』
- 「Interactive SQL での結果セット」『SQL Anywhere サーバ データベース管理』
- 「データベースオブジェクトの名前とプレフィクス」1 ページ

テーブルの作成

Sybase Central を使用して、データベースにテーブルを作成できます。

前提条件

ユーザ本人が所有するテーブルを作成するには、CREATE TABLE システム権限が必要です。他のユーザが所有するテーブルを作成するには、CREATE ANY TABLE または CREATE ANY OBJECT のシステム権限が必要です。

ユーザ本人が所有するプロキシテーブルを作成するには、CREATE PROXY TABLE システム権限が必要です。他のユーザが所有するプロキシテーブルを作成するには、CREATE ANY TABLE または CREATE ANY OBJECT のシステム権限が必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で **[テーブル]** を右クリックし、**[新規]** » **[テーブル]** をクリックします。
3. **テーブル作成ウィザード**の指示に従います。
4. 右ウィンドウ枠で **[カラム]** タブをクリックして、テーブルに新しいカラムを作成します。
5. **[ファイル]** » **[保存]** をクリックします。

結果

新しいテーブルが、データベースに作成されます。

次の手順

データをテーブルに入力またはロードします。

参照

- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「INSERT を使用したデータの追加」619 ページ
- 「INSERT 文を使用したデータのインポート」719 ページ
- 「INSERT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「LOAD TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

テーブル変更

カラムを追加し、さまざまなカラム属性を変更し、またはカラムを削除して、テーブルの構造またはカラム定義を変更できます。

テーブルの変更とビューの依存性

テーブルを変更する前に、sa_dependent_views システムプロシージャを使用して、そのテーブルに依存するビューがあるかどうかを判断できます。

従属ビューでテーブルのスキーマを変更するときは、ビューの種類によって、追加の手順がある場合があります。

- **従属した通常のビュー**　テーブルのスキーマを変更すると、データベース内でテーブルの定義が更新されます。従属した通常のビューが存在する場合、データベースサーバはテーブル変更を実行後にそれらを自動的に再コンパイルします。テーブルのスキーマを変更した後で、データベースサーバが従属した通常のビューを再コンパイルできない場合、変更によってビュー定義が無効になったことが原因と考えられます。この場合は、ビュー定義を訂正する必要があります。

- **従属したマテリアライズドビュー**　従属したマテリアライズドビューが存在する場合は、テーブルの変更前にそれらのビューを無効にし、テーブルの変更後にもう一度有効にする必要があります。テーブルのスキーマを変更した後で、従属した非マテリアライズドビューをもう一度有効にできない場合は、変更によってマテリアライズドビュー定義が無効になったことが原因と考えられます。この場合は、マテリアライズドビューを削除してから、有効な定義を使用してもう一度作成する必要があります。または、基となるテーブルに適切な変更を加えてから、マテリアライズドビューをもう一度有効にしてみてください。

テーブルの所有者の変更

ALTER TABLE 文または Sybase Central を使用して、テーブルの所有者を変更できます。テーブルの所有者を変更するときに、テーブル内の既存の外部キーとそれらが参照しているテーブルを保存するかどうかを指定できます。すべての外部キーを削除するとテーブルが分離されますが、必要な場合はセキュリティが強化されます。明示的に付与された既存の権限を保存するかどうかも指定できます。セキュリティ上の目的で、テーブルへのアクセスを許可している明示的に付与されたすべての権限を削除することができます。そのテーブルの所有者の権限が暗黙的に与えられたものである場合、その権限が新しい所有者に付与され、前の所有者から削除されます。

参照

- 「通常のビューの変更」48 ページ
- 「マテリアライズドビューの作成」61 ページ
- 「ビューの依存性」41 ページ
- 「sa_dependent_views システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

テーブルの変更

Sybase Central を使用して、データベースのテーブルを変更できます。たとえば、カラムの追加または削除、テーブルの所有者の変更を行うことができます。

前提条件

所有者であるか、または次のいずれかの権限を持っていることが必要です。

- そのテーブルに対する ALTER 権限に加え、COMMENT ANY OBJECT、CREATE ANY OBJECT、または CREATE ANY TABLE のシステム権限
- ALTER ANY TABLE システム権限
- ALTER ANY OBJECT システム権限
- ALTER ANY OBJECT OWNER 権限 (テーブルの所有者を変更する場合) と ALTER ANY OBJECT システム権限の 1 つ、ALTER ANY TABLE システム権限またはテーブルに対する ALTER 権限

従属マテリアライズドビューがある場合はテーブルの変更は失敗します。あらかじめ従属マテリアライズドビューを無効にしておく必要があります。sa_dependent_views システムプロシージャを使用して、依存しているマテリアライズドビューがあるかどうかを決定します。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 次のオプションのうちの 1 つを選択してください。

オプション	アクション
カラムの変更	a. 変更するテーブルをダブルクリックします。 b. 右ウィンドウ枠で **[カラム]** タブをクリックして、テーブルのカラムを変更します。 c. **[ファイル]** » **[保存]** をクリックします。
テーブルの所有者の変更	●テーブルを右クリックして、**[プロパティ]** » **[今すぐ所有者を変更]** をクリックして、テーブルの所有者を変更します。

結果

データベースで、テーブルの定義が更新されます。

次の手順

テーブルを変更するため、マテリアライズドビューを無効にする場合、各マテリアライズドビューを再び有効にし、初期化してください。

参照

- 「sa_dependent_views システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「マテリアライズドビューの有効化または無効化」64 ページ
- 「データ整合性」833 ページ
- 「ビューの依存性」41 ページ
- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「依存性とスキーマ変更」42 ページ

テーブルの削除

Sybase Central を使用して、たとえば、必要なくなったときに、データベースからテーブルを削除できます。

前提条件

所有者であるか、または DROP ANY TABLE と DROP ANY OBJECT のいずれかのシステム権限を持っている必要があります。

パブリケーションでアーティクルとして使用されているテーブルは削除できません。Sybase Central でこれを実行しようとすると、エラーが発生します。また、従属ビューのあるテーブルを削除する場合は、追加の手順がある場合があります。

従属マテリアライズドビューがある場合はテーブルの削除は失敗します。あらかじめ従属マテリアライズドビューを無効にしておく必要があります。sa_dependent_views システムプロシージャを使用して、依存しているマテリアライズドビューがあるかどうかを決定します。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[テーブル]** をダブルクリックします。
3. テーブルを右クリックして、**[削除]** をクリックします。
4. **[はい]** をクリックします。

結果

テーブルを削除すると、その定義がデータベースから削除されます。従属した通常のビューが存在する場合、データベースサーバはテーブル変更を実行後にそれらを再コンパイルしてもう一度有効にしようとします。失敗した場合は、テーブルの削除によってビューの定義が無効になったことが原因と考えられます。この場合は、ビュー定義を訂正する必要があります。

従属したマテリアライズドビューが存在する場合、その定義は有効でなくなっているため、以降の再表示は失敗します。この場合は、マテリアライズドビューを削除してから、有効な定義を使用してもう一度作成する必要があります。

テーブルのインデックスはすべて、削除されます。

テーブルを削除すると、COMMIT 文が実行されます。したがって、最後に COMMIT または ROLLBACK を実行した後の変更はすべて確定されます。

次の手順

従属する通常ビューまたはマテリアライズドビューを削除します。またはその定義を変更して、削除されたテーブルの参照を削除します。

参照

- 「sa_dependent_views システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「マテリアライズドビューの有効化または無効化」64 ページ
- 「ビューの依存性」41 ページ
- 「通常のビューの変更」48 ページ
- 「依存性とスキーマ変更」42 ページ
- 「DROP TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

テーブルでのデータの表示 (Sybase Central の場合)

Sybase Central を使用して、テーブル内のデータをブラウズできます。

前提条件

そのテーブルに対する SELECT 権限または SELECT ANY TABLE システム権限が必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[テーブル]** をダブルクリックします。
3. 右ウィンドウ枠で、**[データ]** タブをクリックします。

結果

テーブルのデータが、**[データ]** タブに表示されます。

次の手順

データは **[データ]** タブで編集できます。

Interactive SQL で作業している場合は、次のような文を実行します。*table-name* は、表示するデータです。

```
SELECT * FROM table-name;
```

参照

- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「通常のビュー内のデータのブラウズ」53 ページ

テーブルでのデータの表示 (SQL の場合)

Interactive SQL を使用して、テーブルのデータを表示できます。

前提条件

そのテーブルに対する SELECT 権限または SELECT ANY TABLE システム権限が必要です。

♦ タスク

● *table-name* が、表示するデータを含んだテーブルである場合、次のような文を実行します。

```
SELECT * FROM table-name;
```

結果

テーブルのデータが、**[結果]** ウィンドウ枠に表示されます。

次の手順

[結果] ウィンドウ枠のデータを編集できます。

参照

- 「Interactive SQL」『SQL Anywhere サーバ データベース管理』
- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』

テンポラリテーブル

テンポラリテーブルはテンポラリファイルに格納されます。他の DB 領域のページと同様に、テンポラリファイルのページはキャッシュできます。テンポラリテーブルに対する操作はトランザクションログに書き込まれません。テンポラリテーブルには「ローカルテンポラリ」テーブルと「グローバルテンポラリ」テーブルの 2 種類があります。

● **ローカルテンポラリテーブル**　ローカルテンポラリテーブルは、接続の間だけ、または複合文内で定義されている場合はその複合文が使われている間だけしか存在しません。

  同じスコープ内にある 2 つのローカルテンポラリテーブルは、同じ名前にはできません。ベーステーブルと同じ名前のテンポラリテーブルを作成すると、ローカルテンポラリテーブルのスコープが終了した時点で、そのベーステーブルは、その接続内でのみ参照できるようになります。接続では、既存のテンポラリテーブルと同じ名前のベーステーブルは作成できません。

  ローカルテンポラリテーブルにインデックスを作成するときに、auto_commit_on_create_local_temp_index オプションを OFF に設定すると、テーブルにインデックスを作成する前にコミットはされません。

● **グローバルテンポラリテーブル**　グローバルテンポラリテーブルは、DROP TABLE 文を使用して明示的に削除しないかぎり、データベース内に残ります。同じアプリケーションまたは

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

異なるアプリケーションからの複数の接続は、グローバルテンポラリテーブルを同時に使用できます。グローバルテンポラリテーブルの特性は次のとおりです。

○テーブルの定義はカタログに記録され、テーブルが明示的に削除されるまで保持される。

○テーブルでの挿入、更新、削除は、トランザクションログに記録されない。

○テーブルのカラム統計は、データベースサーバによってメモリ内に保持される。

グローバルテンポラリテーブルには「「非共有」」と「「共有」」の 2 種類があります。. 通常、グローバルテンポラリテーブルは非共有です。つまり、各接続はテーブル内で各自のローしか認識しません。接続が終了すると、その接続のローはテーブルから削除されます。

グローバルテンポラリテーブルが共有されると、テーブルのすべてのデータがすべての接続で共有されます。共有されたグローバルテンポラリテーブルを作成するには、テーブルの作成時に SHARE BY ALL 句を指定します。共有されたグローバルテンポラリテーブルには、グローバルテンポラリテーブルの一般的な特性だけでなく、次の特性が適用されます。

○明示的に削除されるまで、またはデータベースが停止するまで、テーブルのコンテンツは持続する。

○データベースの起動時、テーブルは空である。

○テーブルでのローのロック処理動作は、ベーステーブルの場合と同じである。

- **非トランザクション指向のテンポラリテーブル**　テンポラリテーブルを非トランザクション指向として宣言するには、CREATE TABLE 文の NOT TRANSACTIONAL 句を使用します。状況によっては、NOT TRANSACTIONAL 句を使用するとパフォーマンスが向上します。これは、トランザクション単位でないテンポラリテーブルでの操作では、ロールバックログにエントリが作成されないためです。たとえば、テンポラリテーブルを使用するプロシージャが COMMIT や ROLLBACK の介入を受けずに繰り返し呼び出される場合や、テーブルに多くのローが含まれる場合は、NOT TRANSACTIONAL が有用です。非トランザクション指向テンポラリテーブルへの変更は、COMMIT または ROLLBACK の影響を受けません。

参照

- 「トランザクションと独立性レベル」861 ページ
- 「ロックの仕組み」886 ページ
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DECLARE LOCAL TEMPORARY TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

グローバルテンポラリテーブルの作成

Sybase Central を使用してグローバルテンポラリテーブルを作成できます。ローのロックを心配せずにデータを処理し、トランザクションと再実行ログで必要ないアクティビティを減らす場合、この作業を実行してグローバルテンポラリテーブルを作成します。

前提条件

ユーザ本人が所有するテーブルを作成するには、CREATE TABLE システム権限が必要です。他のユーザが所有するテーブルを作成するには、CREATE ANY TABLE または CREATE ANY OBJECT のシステム権限が必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[テーブル]** を右クリックし、**[新規]** » **[グローバル テンポラリ テーブル]** をクリックします。
3. **グローバルテンポラリテーブル作成ウィザード**の指示に従います。
4. 右ウィンドウ枠で **[カラム]** タブをクリックして、テーブルを設定します。
5. **[ファイル]** » **[保存]** をクリックします。

結果

グローバルテンポラリテーブルが作成されます。グローバルテンポラリテーブル定義は、指定された時点で削除されるまでデータベースに格納され、他の接続に使用できます。

参照

- 「テンポラリテーブル」10 ページ
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DECLARE LOCAL TEMPORARY TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

プロシージャ内でのテンポラリテーブルの参照

プロシージャ間でテンポラリテーブルを共有すると、テーブル定義が矛盾している場合に問題が発生する場合があります。たとえば、procA と procB という 2 つのプロシージャがあり、その両方がテンポラリテーブル temp_table を定義して、sharedProc という名前の別のプロシージャを呼び出すとします。procA と procB のどちらもまだ呼び出されていないため、テンポラリテーブルはまだ存在しません。

ここで、procA の temp_table の定義が procB の定義と少し異なるとします。両方とも同じカラム名と型を使用していますが、カラムの順序が異なります。

procA を呼び出すと、予期した結果が返されます。一方で、procB を呼び出すと、異なる結果が返されます。

これは、procA が呼び出されたときに temp_table が作成され、その後で sharedProc が呼び出されたためです。sharedProc が呼び出されると、内部の SELECT 文が解析および検証され、その後、別の SELECT 文が実行されたときに使用できるように、解析された文の表現がキャッシュされます。キャッシュされたバージョンは、procA のテーブル定義のカラムの順序を反映しています。

procB を呼び出すと temp_table が再作成されますが、カラムの順序が異なります。procB が sharedProc を呼び出すと、データベースサーバは SELECT 文のキャッシュされた表現を使用します。そのため、結果が異なります。

次のいずれかを実行すると、このような状況の発生を防ぐことができます。

●このような方法で使用されるテンポラリテーブルは、一致するように定義する

●代わりにグローバルテンポラリテーブルを使用する

計算カラム

計算カラムとは、同一ロー内にある「従属カラム」と呼ばれる他のカラムの値を参照する式を値としているカラムのことです。計算カラムは、1 つまたは複数の従属カラムの値を含む複雑な式をインデックス化するような場合に特に便利です。データベースサーバは、計算カラムの COMPUTE 式と一致する式が認識できる場合は、常に計算カラムを使用します。これには SELECT リストや述部が含まれます。ただし、クエリ式に CURRENT TIMESTAMP などの特別値が含まれる場合は、この一致は起こりません。

TIMESTAMP WITH TIME ZONE カラムは、計算カラムとして使用しないでください。time_zone_adjustment オプションの値は、ロケーションと日付に基づいて、接続ごとに異なる値となる場合があるため、これらの値が計算されると、不正な結果や予期しない動作が発生することがあります。

クエリの最適化中、SQL Anywhere オプティマイザは、複雑な式を含む述部を、単純に計算カラムの定義を参照する述部へ自動的に変換しようとします。たとえば、クエリに製品出荷に関する一覧情報で構成されたテーブルを要求したとします。

```
CREATE TABLE Shipments(
  ShipmentID INTEGER NOT NULL PRIMARY KEY,
  ShipmentDate TIMESTAMP,
  ProductCode CHAR(20) NOT NULL,
  Quantity INTEGER NOT NULL,
  TotalPrice DECIMAL(10,2) NOT NULL
);
```

特に、クエリは平均コストが 2 ～ 4 ドルである製品出荷を返します。クエリは、次のように記述できます。

```
SELECT *
  FROM Shipments
  WHERE ( TotalPrice / Quantity ) BETWEEN 2.00 AND 4.00;
```

しかし、上記のクエリで、WHERE 句の述部は単一ベースのカラムを参照しないため、検索引数が使用できません。

Shipments テーブルのサイズが比較的大きい場合、インデックス検索の方が逐次スキャンより適している場合があります。インデックス検索を向上させるには、次のように Shipments テーブルに AverageCost という名前の計算カラムを作成してから、そのカラムにインデックスを作成します。

```
ALTER TABLE Shipments
  ADD AverageCost DECIMAL(21,13)
```

```
  COMPUTE( TotalPrice / Quantity );
CREATE INDEX IDX_average_cost
  ON Shipments( AverageCost ASC );
```

計算カラムのタイプを選択することは重要です。クエリ内の式のデータ型が計算カラムのデータ型と正確に一致した場合、SQL Anywhere のオプティマイザは複雑な式のみを計算カラムに置き換えます。式のタイプを判別するために、SQL 文内の式のタイプを返す EXPRTYPE 組み込み関数を使用できます。

```
SELECT EXPRTYPE(
 'SELECT ( TotalPrice/Quantity ) AS X FROM Shipments', 1 )
  FROM DUMMY;
```

Shipments テーブルに対して、上記のクエリは decimal(21,13) を返します。最適化中に、SQL Anywhere オプティマイザは上記のクエリを次のように書き換えます。

```
SELECT *
  FROM Shipments
  WHERE AverageCost
  BETWEEN 2.00 AND 4.00;
```

この場合 WHERE 句内の述部は検索引数可能なものとなり、オプティマイザは、新しい IDX_average_cost インデックスを使用して、クエリのアクセスプラン用のインデックススキャンを選択できます。

参照

- 「特別値」『SQL Anywhere サーバ SQL リファレンス』
- 「クエリの述部」282 ページ

計算カラムの変更

計算カラムで使用された式を変更または削除します。

前提条件

そのテーブルの所有者であるか、または次のいずれかの権限を持っていることが必要です。

- そのテーブルに対する ALTER 権限に加え、COMMENT ANY OBJECT、CREATE ANY OBJECT、または CREATE ANY TABLE のシステム権限
- ALTER ANY TABLE システム権限
- ALTER ANY OBJECT システム権限

♦ タスク

1. データベースに接続します。
2. 次のような ALTER TABLE 文を実行して、計算カラムに使用する式を変更します。

```
ALTER TABLE table-name
ALTER column-name
SET COMPUTE ( new-expression );
```

3. カラムを通常のカラム(未計算)に変換するには、次のような ALTER TABLE 文を実行します。

```
ALTER TABLE
table-name
ALTER column-name
DROP COMPUTE;
```

結果

カラムの計算を変更する場合、この文を実行するとカラムは再計算されます。

計算済みのカラムを通常のカラム(未計算)に変更する場合、文を実行してもカラム内の既存の値は変更されないので、自動的に更新されません。

例

テーブル alter_compute_test を作成してデータを移植し、次の文を実行してテーブルのクエリを選択します。

```
CREATE TABLE alter_compute_test (
  c1 INT,
  c2 INT
) ;
INSERT INTO alter_compute_test (c1) VALUES(100);
SELECT * FROM alter_compute_test ;
```

カラム c2 は NULL 値を返すことに注意してください。カラム c2 を変更して計算カラムとして、カラムにデータを移植して、alter_compute_test テーブルで別の SELECT 文を実行します。

```
ALTER TABLE alter_compute_test
  ALTER c2
  SET COMPUTE ( DAYS ( '2001-01-01' , CURRENT DATE ) )
INSERT INTO alter_compute_test (c1) VALUES(200) ;
SELECT * FROM alter_compute_test ;
```

カラム c2 に、2001 年 1 月 1 日以降の日数が入りました。次に、カラム c2 を計算カラムでないカラムに変更します。

```
ALTER TABLE alter_compute_test
ALTER c2
DROP COMPUTE ;
```

参照

- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- COMPUTE 句『SQL Anywhere サーバ SQL リファレンス』
- 「計算カラムの再計算」16 ページ

計算カラムへの挿入と更新

計算カラムへの挿入や更新における考慮事項は次のとおりです。

- **直接挿入と直接更新**　INSERT または UPDATE 文では計算カラムの値を指定できますが、値は無視されます。サーバは COMPUTE の指定に基づいて計算カラムの値を計算し、その計算カラム値を INSERT または UPDATE 文の指定された値の場所に使用します。

- **カラムの依存性**　たとえば、NULL 値を NULL 値以外の値に変更するなど、計算カラムの定義で参照されるカラムの値を設定するときにトリガを使用しないことを強くおすすめします。これは、計算カラムの値が意図した計算内容を反映していない可能性があるためです。

- **カラム名をリストする**　計算カラムのあるテーブルに対する INSERT 文では、常にカラム名を明示的に指定してください。

- **トリガ**　計算カラムにトリガを定義すると、そのカラムに影響するすべての INSERT 文または UPDATE 文はトリガを起動します。

LOAD TABLE 文では、計算カラムの**オプション**計算が可能です。ロード操作中に計算を行わないようにすると、複雑なアンロード／再ロード手順を速くすることができます。これは、COMPUTE 式が CURRENT TIMESTAMP などの非確定値を参照する場合でも、計算カラムの値が一定である必要がある場合に便利です。

値を変更すると計算カラムの値がカラム定義と一致しなくなる場合があるため、トリガの従属カラムの値は変更しないでください。

計算カラム x が NOT NULL と宣言されたカラム y に依存する場合、y に NULL を設定しようとすると、トリガが起動される前にエラーが発生し、拒否されます。

計算カラムの再計算

計算カラムの値は、ローが挿入され更新されると自動的にデータベースサーバによって維持されます。ほとんどのアプリケーションでは、計算カラム値を直接更新したり挿入したりする必要はありません。

計算カラムは、次の状況で再計算されます。

- いずれかのカラムが削除、追加、または名前が変更された場合

- テーブルは、任意のカラムのデータ型を修正する ALTER TABLE 文または COMPUTE 句によって変更されます。

- ローが挿入された場合

- ローが更新された場合

計算カラムは、次の状況では再計算**されません**。

- テーブルの名前が変更された場合

- 計算カラムが問い合わせされている。

- 計算カラムは (サブクエリまたはユーザ定義関数を使用した) 他のローの値に依存しており、しかもこれらのローは変更されます。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

プライマリキー

リレーショナルデータベース内の各テーブルには、「プライマリキー」が必要です。プライマリキーとは 1 つのカラム、またはカラムのセットで、各ローをユニークに識別します。1 つのテーブル内で、2 つのローが同じプライマリキーの値を持つことはできません。また、プライマリキーのカラムを NULL 値にすることはできません。

ベーステーブルとグローバルテンポラリテーブルのみがプライマリキーを持つことができます。宣言されたテンポラリテーブルでは、NOT NULL カラムのセットにユニークインデックスを作成し、プライマリキーのセマンティックを模倣できます。

プライマリキーや、一意性制約があるカラムには FLOAT や DOUBLE などの概数値データ型を使用しないことをおすすめします。概数値データ型は、算術演算後の丸め誤差がでます。

CLUSTERED 句を使用して、プライマリキーインデックスをクラスタ化するかどうかを指定することもできます。

マルチカラムプライマリキーのカラムの順序
プライマリキーカラムの順序は、CREATE TABLE (または ALTER TABLE) 文のプライマリキー宣言で指定したカラム順序によって決まります。個別のカラムのソート順 (昇順または降順) を指定することもできます。これらのソート順指定は、プライマリキーインデックスを作成するときにデータベースサーバによって使用されます。

プライマリキー内のカラムの順序は、参照整合性制約のカラムの順序を指定しません。外部キー宣言で、異なるカラム順序、異なるソート順序を指定できます。

例

SQL Anywhere サンプルデータベースでは、Employees テーブルに従業員の個人情報が格納されています。このテーブルにはプライマリキーカラム EmployeeID があり、各従業員に割り当てられたユニークな ID 番号が入っています。ID 番号が入っている単一のカラムにプライマリキーを割り当てるのが一般的な方法です。名前やその他の識別子は、必ずしもユニークではないため、ID 番号の方が適しています。

SQL Anywhere サンプルデータベースの SalesOrderItems テーブルのプライマリキーは、より複雑です。このテーブルには、会社からの注文に含まれる個々の項目に関する情報が入っており、次のカラムで構成されています。

- **ID**　その注文項目が含まれている注文を識別する注文番号
- **LineID**　注文内での注文項目を識別する行番号
- **ProductID**　受注した製品を識別する製品 ID
- **Quantity**　受注した項目の数量
- **ShipDate**　受注品が出荷された日付

特定の注文項目はその注文項目が含まれている注文番号と、注文内での行番号によって識別されます。この 2 つの番号は、ID カラムと LineID カラムに格納されています。注文項目が同一の ID

値を共有する場合 (1 回の注文に複数の注文項目がある場合) と、注文項目が同一の LineID 番号を共有する場合 (各注文における最初の注文項目はすべて LineID が 1) があります。どの注文項目も、これら 2 つの値を共有することはないので、プライマリキーは 2 つのカラムで構成されています。

参照

- 「クラスタドインデックス」30 ページ
- 「プライマリキーによるエンティティ整合性の確保」851 ページ

プライマリキーの管理 (Sybase Central の場合)

Sybase Central を使用してプライマリキーを管理して、テーブルのクエリパフォーマンスを向上させます。

前提条件

そのテーブルの所有者であるか、または次のいずれかの権限を持っていることが必要です。

- ALTER ANY OBJECT システム権限
- ALTER ANY INDEX と ALTER ANY TABLE のシステム権限
- そのテーブルに対する ALTER 権限と REFERENCES に加え、COMMENT ANY OBJECT、CREATE ANY OBJECT、または CREATE ANY TABLE のシステム権限

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[テーブル]** をダブルクリックします。
3. テーブルを右クリックし、次のいずれかのオプションを選択します。

オプション	アクション
プライマリキーの作成または変更	**[プライマリキーの設定]** をクリックして、**[プライマリキー設定ウィザード]** の指示に従います。
プライマリキーを削除します。	テーブルの **[カラム]** ウィンドウ枠で、**[PKey]** カラムからチェックマークをクリアして、**[保存]** をクリックします。

結果

プライマリキーが追加、変更、または削除されます。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

参照

- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「プライマリキーの管理 (SQL の場合)」19 ページ
- 「プライマリキーによるエンティティ整合性の確保」851 ページ

プライマリキーの管理 (SQL の場合)

SQL を使用してプライマリキーを管理して、テーブルのクエリパフォーマンスを向上させます。

前提条件

そのテーブルの所有者であるか、または次のいずれかの権限を持っていることが必要です。

- そのテーブルに対する ALTER 権限
- ALTER ANY TABLE システム権限
- ALTER ANY OBJECT システム権限

プライマリキーのカラムに NULL 値を含むことはできません。

♦ タスク

- データベースに接続します。

オプション	アクション
プライマリキーを作成します。	ALTER TABLE *table-name* ADD PRIMARY KEY *(column-name)* 文を実行します。
プライマリキーを削除します。	ALTER TABLE *table-name* DROP PRIMARY KEY 文を実行します。
プライマリキーを変更します。	テーブルの新しいプライマリキーを作成する前に、既存のプライマリキーを削除します。

結果

プライマリキーが追加、削除、または変更されます。

例

次の文は、Skills という名前のテーブルを作成し、SkillID カラムをプライマリキーとして割り当てます。

```
CREATE TABLE Skills (
  SkillID INTEGER NOT NULL,
  SkillName CHAR( 20 ) NOT NULL,
  SkillType CHAR( 20 ) NOT NULL,
```

```
  PRIMARY KEY( SkillID )
);
```

プライマリキー値は、テーブル内のローごとにユニークである必要があります。この例では、特定の SkillID を持つローを複数設定できません。テーブルの各ローは、そのプライマリキーによってユニークに識別されます。

プライマリキーに SkillID カラムと SkillName カラムを組み合わせて使用するようにプライマリキーを変更する場合は、作成したプライマリキーを削除してから、新しいプライマリキーを追加します。

```
ALTER TABLE Skills DELETE PRIMARY KEY;
ALTER TABLE Skills ADD PRIMARY KEY ( SkillID, SkillName );
```

参照

- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「プライマリキーの管理 (Sybase Central の場合)」18 ページ
- 「プライマリキーによるエンティティ整合性の確保」851 ページ

外部キー

外部キーは、カラムまたはカラムのセットで構成されます。キー値が一致するプライマリテーブル内のローの参照を表します。外部キーはベーステーブルのみで使用できます。テンポラリテーブル、グローバルテンポラリテーブル、ビュー、またはマテリアライズドビューでは使用できません。外部キーは「参照整合性制約」と呼ばれることもあります。外部キーを含むベーステーブルは「参照元テーブル」と呼ばれ、プライマリキーを含むテーブルは「参照先テーブル」と呼ばれます。

外部キーが NULL 入力可能な場合、参照先テーブル内に一致するプライマリキー値がない外部ローが存在する場合があるため、関係はオプションです。これは、プライマリキーと UNIQUE 制約のカラムはどちらも NULL にできないためです。外部キーカラムが NOT NULL と宣言された場合、関係は必須であり、参照元テーブルの各ローには、参照先テーブルにプライマリキーとして存在する外部キー値が含まれている必要があります。

外部キーと孤立したロー

参照整合性を維持するには、不一致の NULL でない外部キー値がデータベースに含まれないようにします。参照整合性に違反する外部ローは、参照先テーブルのプライマリキー値と一致しないため、「オーファン」と呼ばれます。オーファンは次の操作で作成されます。

- 参照元テーブルでのローの挿入または更新の際に、参照先テーブルのプライマリキー値と一致しない NULL でない値が外部キーカラムにある。

- プライマリテーブルでのローの更新または削除により、参照元テーブルの少なくとも 1 つのローで、一致するプライマリキー値がなくなる。

SQL Anywhere では、孤立したローを作成しないようにして、参照整合性違反を防ぎます。

複合外部キー

SQL Anywhere では、複数カラムのプライマリキーと外部キーもサポートされており、「複合キー」と呼ばれます。複合外部キーでも NULL 値は不一致を示しますが、オーファンの識別方法は MATCH 句での参照整合性制約の定義方法によって異なります。

外部キーインデックスとソートの程度の順序

外部キーを作成すると、キーのインデックスが自動的に作成されます。外部キーカラムの順序はプライマリキーカラムの順序を反映している必要はなく、プライマリキーインデックスのソートの順序が外部キーインデックスのソートの順序と一致している必要もありません。外部キーインデックスでのインデックス付けされた各カラムのソート –(昇順または降順)– はカスタマイズでき、外部キーインデックスのソートの順序が、アプリケーションの特定の SQL クエリに必要な、そのクエリの文の ORDER BY 句で指定されたソートの順序に一致するようにできます。各カラムのソートの順序は、外部キー制約を設定するときに指定できます。

例 1

SQL Anywhere サンプルデータベースには、従業員 (employee) 情報が入ったテーブルが 1 つと、部署 (department) 情報の入ったテーブルが 1 つあります。Departments テーブルには、次のカラムがあります。

- **DepartmentID**　部署の ID 番号。これがテーブルのプライマリキーになります。
- **DepartmentName**　部署の名前
- **DepartmentHeadID**　部長の従業員 ID

特定の従業員の所属部署名を探せるように、その従業員の部署名を Employees テーブルに入力しておく必要はありません。その代わり Employees テーブルには、Departments テーブルの DepartmentID 値の 1 つと一致する値の入った DepartmentID カラムがあります。

Employees テーブルの DepartmentID カラムは、Departments テーブルに対する外部キーです。外部キーは、対応するプライマリキーを持つテーブル内の特定のローを参照します。

そのため、Employees テーブル (関係付けの外部キーを持つ) を「外部テーブル」または「参照元テーブル」と呼びます。Departments テーブル (参照先のプライマリキーを持つ) は、「プライマリテーブル」または「参照先テーブル」と呼びます。

例 2

次の文を実行して、複合プライマリキーを作成します。

```
CREATE TABLE pt(
    pk1 INT NOT NULL,
    pk2 INT NOT NULL,
    str VARCHAR(10),
    PRIMARY KEY ( pk1, pk2 ));
```

次の文は、プライマリキーとカラムの順序が異なる外部キーと、外部キーカラムに対して異なるソートの程度を作成します。これは、外部キーインデックスを作成する場合に使用されます。

```
CREATE TABLE ft1(
    fpk INT PRIMARY KEY,
```

```
    ref1 INT,
    ref2 INT );

ALTER TABLE ft1 ADD FOREIGN KEY ( ref2 ASC, ref1 DESC)
    REFERENCES pt ( pk2, pk1 ) MATCH SIMPLE;
```

次の文を実行して、プライマリキーとカラムの順序が同じだが、外部キーインデックスに対してソートの程度が異なる外部キーを作成します。また、この例では、MATCH FULL 句を使用して、両方のカラムが NULL だった場合に孤立したローになることを指定します。UNIQUE 句は、NULL ではないカラムの **pt** テーブルと **ft2** テーブルの間に 1 対 1 の関係を適用します。

```
CREATE TABLE ft2(
    fpk INT PRIMARY KEY,
    ref1 INT,
    ref2 INT );

ALTER TABLE ft2 ADD FOREIGN KEY ( ref1, ref2 DESC )
    REFERENCES pt ( pk1, pk2 ) MATCH UNIQUE FULL;
```

参照

- FOREIGN KEY 句、CREATE TABLE 文『SQL Anywhere サーバ SQL リファレンス』
- 「参照整合性」851 ページ

外部キーの作成 (Sybase Central の場合)

テーブル間に外部キー関係を作成できます。すると、外部キーの関係は制約として機能します。データベースサーバは、子テーブルに挿入した新しいローに対して、外部キーカラムに挿入している値がプライマリテーブルのプライマリキーの値と一致するかどうかを確認します。

前提条件

そのテーブルに対する SELECT 権限または SELECT ANY TABLE システム権限が必要です。

さらに、そのテーブルの所有者であるか、または次のいずれかの権限を持っていることが必要です。

- そのテーブルに対する ALTER 権限に加え、COMMENT ANY OBJECT、CREATE ANY OBJECT、または CREATE ANY TABLE のシステム権限
- ALTER ANY TABLE システム権限
- ALTER ANY OBJECT システム権限

内容と備考

外部テーブルの作成時に外部キーを作成する必要はありません。外部キーは自動的に作成されます。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[テーブル]** をダブルクリックします。

3. 作成するテーブルまたは外部キーを選択します。
4. 右ウィンドウ枠で、**[制約]** タブをクリックします。
5. 外部キーを作成します。
   a. **[ファイル]** » **[新規]** » **[外部キー]** をクリックします。
   b. **外部キー作成ウィザード**の指示に従います。

結果

Sybase Central では、テーブルの外部キーは、テーブルが選択されている場合、右ウィンドウ枠内の **[制約]** タブに表示されます。テーブルが更新され、その外部キー定義がテーブルに含まれます。

次の手順

ウィザードを使用して外部キーを作成するときに、その外部キーのプロパティを設定できます。外部キーを作成した後でプロパティを表示するには、**[制約]** タブで外部キーを選択し、**[ファイル]** » **[プロパティ]** をクリックします。

[参照元制約] タブでテーブルを選択してから **[ファイル]** » **[プロパティ]** をクリックすると、参照元外部キーのプロパティを表示できます。

所定のテーブルを参照するテーブルのリストを表示するには、**[テーブル]** でテーブルを選択し、右ウィンドウ枠で **[参照元制約]** タブをクリックします。

参照

- 「外部キーの作成 (Sybase Central の場合)」22 ページ
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

外部キーの作成 (SQL の場合)

Interactive SQL では、CREATE TABLE 文と ALTER TABLE 文を使用して、外部キーを作成、修正できます。これらの文によって、カラムの制約や検査など、多くのテーブル属性を設定できます。

前提条件

テーブルの所有者に基づいて外部キーを作成するのに必要な権限は次のとおりです。

- **参照されるテーブル (プライマリキー) と参照するテーブル (外部キー) を所有**　権限は必要ありません。
- **参照されるテーブルではなく、参照するテーブルを所有**　そのテーブルに対する REFERENCES 権限を持っているか、または CREATE ANY INDEX と CREATE ANY OBJECT のどちらか一方のシステム権限を持っていることが必要です。

- **参照するテーブルではなく、参照されるテーブルを所有**
  - ALTER ANY OBJECT または ALTER ANY TABLE のシステム権限が必要です。
  - あるいは、そのテーブルに対する ALTER 権限に加え、COMMENT ANY OBJECT、CREATE ANY OBJECT、または CREATE ANY TABLE のシステム権限が必要です。
  - さらに、そのテーブルに対する SELECT 権限または SELECT ANY TABLE システム権限も必要です。
- **どちらのテーブルも所有していない**
  - そのテーブルに対する REFERENCES 権限を持っているか、または CREATE ANY INDEX と CREATE ANY OBJECT のどちらか一方のシステム権限を持っていることが必要です。
  - ALTER ANY OBJECT または ALTER ANY TABLE のシステム権限が必要です。
  - あるいは、そのテーブルに対する ALTER 権限に加え、COMMENT ANY OBJECT、CREATE ANY OBJECT、または CREATE ANY TABLE のシステム権限が必要です。
  - さらに、そのテーブルに対する SELECT 権限または SELECT ANY TABLE システム権限も必要です。

そのテーブルに対する SELECT 権限または SELECT ANY TABLE システム権限が必要です。

さらに、そのテーブルの所有者であるか、または次のいずれかの権限を持っていることが必要です。

- そのテーブルに対する ALTER 権限に加え、COMMENT ANY OBJECT、CREATE ANY OBJECT、または CREATE ANY TABLE のシステム権限
- ALTER ANY TABLE システム権限
- ALTER ANY OBJECT システム権限

内容と備考

外部テーブルの作成時に外部キーを作成する必要はありません。外部キーは自動的に作成されます。

♦ タスク

1. データベースに接続します。
2. 次のような ALTER TABLE 文を実行します。

```
ALTER TABLE table-name ADD FOREIGN KEY foreign-key-name
( column-name ASC ) REFERENCES table-name ( column-name )
```

結果

テーブルが更新され、その外部キー定義がテーブルに含まれます。

例

次の例では、スキルの一覧を格納する Skills というテーブルを作成し、次に Skills テーブルに対して外部キーの関係を持つ EmployeeSkills というテーブルを作成します。EmployeeSkills.SkillID は、Skills テーブルのプライマリキーカラム (Id) と外部キーの関係があります。

```
CREATE TABLE Skills (
  Id INTEGER PRIMARY KEY,
  SkillName CHAR(40),
  Description CHAR(100)
);
CREATE TABLE EmployeeSkills (
  EmployeeID INTEGER NOT NULL,
  SkillID INTEGER NOT NULL,
  SkillLevel INTEGER NOT NULL,
  PRIMARY KEY( EmployeeID ),
  FOREIGN KEY (SkillID) REFERENCES Skills ( Id )
);
```

テーブルを作成した後で、ALTER TABLE 文を使用して外部キーを追加することもできます。次の例では、前の例で作成したテーブルに似たテーブルを作成します。ただし、テーブルの作成後に外部キーを追加します。

```
CREATE TABLE Skills2 (
  ID INTEGER PRIMARY KEY,
  SkillName CHAR(40),
  Description CHAR(100)
);
CREATE TABLE EmployeeSkills2 (
  EmployeeID INTEGER NOT NULL,
  SkillID INTEGER NOT NULL,
  SkillLevel INTEGER NOT NULL,
  PRIMARY KEY( EmployeeID ),
);
ALTER TABLE EmployeeSkills2
  ADD FOREIGN KEY SkillFK ( SkillID )
  REFERENCES Skills2 ( ID );
```

外部キーを作成するとき、そのプロパティを指定できます。たとえば、次の文は例 2 と同じ外部キーを作成しますが、この外部キーは、データの更新または削除に対する制限がある NOT NULL として定義されています。

```
ALTER TABLE Skills2
ADD NOT NULL FOREIGN KEY SkillFK ( SkillID )
REFERENCES Skills2 ( ID )
ON UPDATE RESTRICT
ON DELETE RESTRICT;
```

外部キーカラム名とプライマリキーのカラム名とは、1 対 1 で対応する 2 つのリスト位置に従ってペアになります。外部キーの定義時にプライマリテーブルのカラム名が指定されていない場合、プライマリキーカラムが使用されます。たとえば、次のようにして 2 つのテーブルを作成するとします。

```
CREATE TABLE Table1( a INT, b INT, c INT, PRIMARY KEY ( a, b ) );
CREATE TABLE Table2( x INT, y INT, z INT, PRIMARY KEY ( x, y ) );
```

次に、外部キー fk1 を作成し、2 つのテーブル間でのカラムのペア方法を厳密に指定します。

```
ALTER TABLE Table2 ADD FOREIGN KEY fk1( x,y ) REFERENCES Table1( a, b );
```

次の文を使用して、外部テーブルカラムのみを指定することで、2 つ目の外部キー fk2 を作成します。データベースサーバは、これらの 2 つのカラムを、プライマリテーブル上のプライマリキーにある最初の 2 つのカラムと自動的にペアにします。

```
ALTER TABLE Table2 ADD FOREIGN KEY fk2( x, y ) REFERENCES Table1;
```

次の文を使用して、プライマリテーブルと外部テーブルのいずれに対してもカラムを指定せずに外部キーを作成します。

```
ALTER TABLE Table2 ADD FOREIGN KEY fk3 REFERENCES Table1;
```

参照元カラムを指定していないため、データベースサーバは、プライマリテーブル (Table1) のカラムと同じ名前を持つカラムを外部テーブル (Table2) で検索します。同じ名前のカラムが存在する場合、データベースサーバはデータ型が一致することを確認し、それらのカラムを使用して外部キーを作成します。カラムが存在しない場合は、Table2 に作成されます。この例では、Table2 には a や b というカラムが存在**しない**ため、Table1.a および Table1.b と同じデータ型で作成されます。このように自動的に作成されたカラムは、外部テーブルのプライマリーキーの一部になることはできません。

参照

- 「外部キーの作成 (Sybase Central の場合)」22 ページ
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

インデックス

「インデックス」は、テーブルの 1 カラムまたは複数のカラムについてローに順序を付けます。電話帳のように、インデックスは最初に姓でソートし、次に同じ姓の人を名前でソートします。この順序は、特定の姓を持つ人の電話番号をすばやく検索できますが、特定の住所から電話番号を検索する場合には意味がありません。同様に、データベースインデックスは、特定のカラムを 1 つまたは複数検索する場合にのみ役立ちます。

インデックスの有用性は、テーブルのサイズが大きくなるにつれて増大します。住所から電話番号を検索する速度は電話帳の厚さに比例しますが、姓で検索する場合は電話帳のサイズにあまり関係ないのと同じです。K. Kaminski を検索する場合、厚い電話帳と薄い電話帳ではほとんど時間は変わりません。

オプティマイザでは、自動的にインデックスを使用して、データベースの文のパフォーマンスを改善できる場合は改善します。ローが削除、更新、挿入された場合は、自動的にインデックスの更新が行われます。クエリの作成時にインデックスヒントを使用してインデックスを明示的に参照できますが、その必要はありません。

インデックスの作成にはいくつか欠点があります。特に、カラム内のデータが変更された場合はインデックスをテーブル自体とともに管理する必要があるため、挿入、更新、削除のパフォーマンスがインデックスによって影響される場合があります。このため、不要なインデックスは削除してください。インデックスコンサルタントを使用して、不要なインデックスを識別します。

作成するインデックスの決定

データベースに適切なインデックスセットを選択することは、パフォーマンスを最適化する上で重要です。適切なセットを識別することは、労力を要する作業でもあります。

インデックスの作成が必要かどうかは、単純な計算式では判断できません。インデックス検索の利点と、そのインデックスの管理に伴うオーバヘッドのトレードオフを考慮してください。次の要素を考慮して、インデックスの作成が必要かどうかを判断できます。

- **キーとユニークなカラム**　SQL Anywhere は、プライマリキー、外部キー、ユニークなカラムのインデックスを自動的に作成します。これらのカラムについては、インデックスを追加して作成しないでください。ただし、複合キーの場合は、追加インデックスで強化できることがあります。

- **検索頻度**　特定のカラムが頻繁に検索される場合は、そのカラムのインデックスを作成するとパフォーマンスを向上できます。検索頻度の低いカラムにインデックスを作成しても意味がありません。

- **テーブルのサイズ**　多数のローを持つ比較的大きなテーブルのインデックスを作成すると、比較的小さなテーブルのインデックスの場合よりも多くの利点が得られます。たとえば、ローが 20 しかないテーブルの場合、逐次スキャンにはインデックスルックアップと同程度の時間しかかからないため、インデックスを作成しても利点は得られません。

- **更新回数**　インデックスは、テーブルに対してローが挿入または削除されたり、インデックスカラムが更新されたりするたびに、更新されます。カラムにインデックスがあると、挿入、更新、削除のパフォーマンスが低下します。更新頻度の高いデータベースのインデックスは、読み込み専用データベースの場合より少なくなるようにしてください。

- **領域の注意事項**　インデックスはデータベース内の領域を占有します。データベースサイズが重要な場合は、インデックスの作成を控えてください。

- **データ分散**　インデックスルックアップから返される値が多すぎると、逐次スキャンよりも高コストになります。SQL Anywhere は、この条件を認識するとインデックスを使用しません。たとえば、SQL Anywhere サンプルデータベース内の Employees.Sex のように、値が 2 つしかないカラムについては、インデックスを使用しません。このため、重複を排除した (distinct) 値が少数しかないカラムのインデックスは作成しないでください。

インデックスを作成するときは、カラムを指定する順序は、インデックスでカラムが出現する順序になります。インデックス定義でカラム名を重複参照することはできません。

注意
インデックスコンサルタントは、適切なインデックス選択を支援するためのツールです。インデックスコンサルタントは、単一のクエリまたは一連の操作を分析して、データベースに追加するインデックスを推奨します。また、使用されていないインデックスを通知します。

テンポラリテーブルのインデックス

インデックスは、ローカルとグローバルのテンポラリテーブル上で作成できます。テンポラリテーブルが大きく、ソートされた順序またはジョインで数回アクセスされることが予想される場合は、インデックスを作成します。そのような予想がない状況では、クエリを処理するパフォーマンスの改善よりも、インデックスを作成し削除するコストの方が上回ってしまいます。

参照

- 「複合インデックス」28 ページ
- 「インデックス」26 ページ
- 「インデックスを適切に選択するとパフォーマンスに大きな違いが生じる」223 ページ
- 「クエリに対するインデックスコンサルタントの推奨内容の確認」157 ページ
- 「ヒント：インデックスの有効な使用」223 ページ
- 「クエリに対するインデックスコンサルタントの推奨内容の確認」157 ページ
- WITH (index-hint) 句、FROM 句『SQL Anywhere サーバ SQL リファレンス』

複合インデックス

インデックスには 1 つまたは複数のカラムを含めることができます。複数のカラムに対するインデックスは、「複合インデックス」と呼ばれます。たとえば、次の文では 2 カラムの複合インデックスが作成されます。

```
CREATE INDEX name
ON Employees ( Surname, GivenName );
```

複合インデックスは、最初のカラムだけでは高い選択性が得られない場合に役立ちます。たとえば、Surname と GivenName に対する複合インデックスは、従業員の姓が同じ場合に便利です。各従業員はユニークな ID を持っており、カラム Surname は追加の選択性を提供しないため、EmployeeID と Surname の複合インデックスは役に立ちません。

インデックスにカラムを追加すると検索対象を限定できますが、2 カラムのインデックスを使用することと 2 つの別個のインデックスを使用することは異なります。複合インデックスは、電話帳でまず姓が並べられ、次に同じ姓の中で名前順に並べられるのとよく似た構造を持っています。電話帳は姓を知っていれば役に立ちますし、姓と名前の両方を知っていればなお役に立ちます。しかし、名前だけを知っていても役には立ちません。

カラムの順序

複合インデックスを作成する場合は、カラムの順序を慎重に検討してください。複合インデックスは、インデックスのすべてのカラムまたは最初のカラムだけを検索する場合に役立ちます。2 番目以降のカラムだけを検索する場合には役立ちません。

1 つのカラムだけを何度も検索する場合は、そのカラムを複合インデックスの最初のカラムにしてください。2 カラムインデックスの両方のカラムを個別に検索する場合は、第 2 のカラムだけで構成される 2 番目のインデックスを作成することを検討します。

たとえば、2 つのカラムに複合インデックスを作成するとします。1 つのカラムには従業員の名前が格納され、もう 1 つには従業員の姓が格納されます。名前、姓の順に格納するインデックスを作成できます。または、姓、名前の順にインデックスを付けることもできます。この 2 つのインデックスは両方のカラムの情報を編成するものですが、その機能は異なります。

```
CREATE INDEX IX_GivenName_Surname
  ON Employees ( GivenName, Surname );
CREATE INDEX IX_Surname_GivenName
  ON Employees ( Surname, GivenName );
```

次に、名前 John を検索するとします。使用できる唯一のインデックスは、インデックスの最初のカラムに名前を格納しているインデックスです。姓、名前の順に編成されているインデックス

は使用できません。これは、John という名前の人がインデックスのどこに現れるかわからないためです。

名前だけ、または姓だけで人を検索する場合、これらのインデックスの両方を作成することを検討してください。

または、それぞれが 1 つのカラムだけを含むインデックスを 2 つ作成する方法もあります。ただし、SQL Anywhere は、1 つのクエリを処理するとき、1 つのテーブルにアクセスするために 1 つのインデックスしか使用しません。両方の名前がわかっていても、SQL Anywhere は正しい姓を持つローを検索するため、追加のローを読み込む必要があります。

前述の例のように、CREATE INDEX 文を使用してインデックスを作成した場合、カラムは文で指定した順序で表示されます。

複合インデックスと ORDER BY

デフォルトでは、インデックスのカラムは昇順でソートされますが、オプションで、CREATE INDEX 文で DESC を指定すると降順でソートできます。

ORDER BY 句にインデックスに含まれるカラムだけが指定されているかぎり、SQL Anywhere は、そのインデックスを使用して ORDER BY クエリを最適化するように選択できます。また、インデックスのカラムは、ORDER BY 句と正確に同じ、または正反対の順序になります。1 カラムインデックスの場合、順序付けは常に最適化できますが、複合インデックスには多少の考慮が必要です。次の表に、2 カラムインデックスで可能な操作を示します。

インデックスカラム	最適化可能な ORDER BY クエリ	最適化できない ORDER BY クエリ
ASC、ASC	ASC、ASC または DESC、DESC	ASC、DESC または DESC、ASC
ASC、DESC	ASC、DESC または DESC、ASC	ASC、ASC または DESC、DESC
DESC、ASC	DESC、ASC または ASC、DESC	ASC、ASC または DESC、DESC
DESC、DESC	DESC、DESC または ASC、ASC	ASC、DESC または DESC、ASC

3 つ以上のカラムを持つインデックスには、上記と同じ規則が適用されます。たとえば、次のインデックスがあるとします。

```
CREATE INDEX idx_example
ON table1 ( col1 ASC, col2 DESC, col3 ASC );
```

この場合は、次に示すクエリを最適化できます。

```
SELECT col1, col2, col3 FROM table1
ORDER BY col1 ASC, col2 DESC, col3 ASC;
```

```
SELECT col1, col2, col3 FROM example
ORDER BY col1 DESC, col2 ASC, col3 DESC;
```

ORDER BY 句に ASC と DESC の他のパターンを持つクエリの最適化には、このインデックスは使用されません。たとえば、次の文は最適化されません。

```
SELECT col1, col2, col3 FROM table1
ORDER BY col1 ASC, col2 ASC, col3 ASC;
```

クラスタドインデックス

インデックスの「クラスタ化」を宣言することで、大規模なインデックススキャンをさらに改善することができます。クラスタドインデックスを使用すると、連続したインデックスエントリにおける 2 つのローがデータベース内の同じページに現れる確率が高くなります。この戦略では、テーブルページをバッファプールに読み込む回数が減り、パフォーマンスがさらに向上します。

クラスタ化プロパティを持つインデックスが存在すると、データベースサーバはテーブルのローをクラスタドインデックスに出現する場合とほぼ同じ順序で格納しようとします。ただし、データベースサーバはキーの順序を保持しようとしますが、クラスタ化は概算であり、完全なクラスタは保証されません。このため、データベースサーバはテーブルを順次スキャンできず、クラスタドインデックスキーのシーケンスですべてのローが取得されるわけではありません。テーブルのローがソートされた順序で返されるようにするには、インデックスを使用してローにアクセスするアクセスプランか、物理ソートを実行するアクセスプランが必要です。

オプティマイザはクラスタ化プロパティを持つインデックスを利用します。これは、一致または隣接するインデックスキー値を持つテーブルローについて、オプティマイザが物理的な隣接性の予測を考慮に入れてインデックス取得コストの予測を修正することによって行われます。

多くのローが挿入または更新されていくため、テーブルのクラスタ化の程度は時間とともに低下することがあります。データベースサーバは、ISYSPHYSIDX システムテーブルのクラスタドインデックスごとにクラスタ化の程度を自動的に追跡します。テーブルのローで非クラスタ化が大幅に進行したことをデータベースサーバが検出すると、オプティマイザは予測したインデックス取得コストを調整します。

テーブルのいずれかのインデックスをクラスタ化することを決定する際は、予測されるクエリの負荷を考慮する必要があります。通常は実験が必要になります。一般的に、指定されたクエリに次のような状態が起こる場合には、データベースサーバはクラスタドインデックスを使用してパフォーマンスを向上させることができます。

- クエリの応答に必要なテーブルページの多くが、メモリ内にまだ存在しない。テーブルページがすでにメモリ内に存在する場合、サーバはこれらのページを読み込む必要がないため、クラスタリングは影響しません。

- 非自明な数のローが返されると予想されるインデックス検索を実行し、クエリが応答できる。たとえば、通常、クラスタリングは単純なプライマリキーの検索には影響しません。

- インデックス専用取得の実行とは対照的に、データベースサーバは実際にテーブルページを読み込む必要がある。

クラスタドインデックスの宣言

インデックスのクラスタ化プロパティは、SQL 文を使用していつでも追加または削除できます。あらゆるプライマリキーインデックス、外部キーインデックス、一意性制約インデックス、セカンダリインデックスは、CLUSTERED プロパティを使用して宣言できます。ただし、宣言できるクラスタドインデックスはテーブルあたり多くても 1 つです。これは、次のいずれかの文で行います。

- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER DATABASE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DECLARE LOCAL TEMPORARY TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

複数の文を組み合わせて使用すると、クラスタ化の効果の維持やリストアができます。

- UNLOAD TABLE 文を使用すると、クラスタドインデックスキーの順序でテーブルをアンロードできます。

- LOAD TABLE 文は、クラスタドインデックスキーの順序でローをテーブルに挿入します。

- INSERT 文は、新しいローを挿入するときに、クラスタドインデックスキーの順序が隣接するローと同じテーブルページに挿入しようとします。

- REORGANIZE TABLE 文はクラスタドインデックスに従ってローを再編成することによって、テーブルのクラスタ化をリストアします。クラスタ化を指定していないテーブルで REORGANIZE TABLE 文を使用すると、プライマリキーを使用してテーブルが並べ替えられます。

インデックス作成ウィザードを使用して、Sybase Central でクラスタドインデックスを作成することもできます。メッセージが表示されたら、**[クラスタドインデックスを作成する]** をクリックします。

参照

- 「UNLOAD 文」『SQL Anywhere サーバ SQL リファレンス』
- 「LOAD TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「INSERT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REORGANIZE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「インデックスの作成」31 ページ

インデックスの作成

ベーステーブルのインデックス、テンポラリテーブル、マテリアライズドビューを作成します。計算カラムを使用して、組み込み関数のインデックスを作成することもできます。

前提条件

テーブルのインデックスを作成するには、テーブルの所有者であるか、次のいずれかの権限を持っている必要があります。

- CREATE ANY INDEX システム権限
- CREATE ANY OBJECT システム権限
- そのテーブルに対する REFERENCES 権限に加え、COMMENT ANY OBJECT システム権限、ALTER ANY INDEX システム権限、または ALTER ANY OBJECT システム権限

マテリアライズドビューのインデックスを作成するには、マテリアライズドビューの所有者であるか、次のいずれかの権限を持っている必要があります。

- CREATE ANY INDEX システム権限
- CREATE ANY OBJECT システム権限

通常のインデックスは作成できません。無効になっているマテリアライズドビューのインデックスは作成できません。

内容と備考

インデックスを作成するときは、カラムを指定する順序は、インデックスでカラムが出現する順序になります。インデックス定義でカラム名を重複参照することはできません。データベースに適切なインデックスの選択を手助けをするインデックスコンサルタントも使用できます。

auto_commit_on_create_local_temp_index オプションを ON に設定した場合、ローカルテンポラリテーブルでインデックスを作成すると、オートコミットが実行されます。このオプションのデフォルト設定は Off です。

関数 (暗黙的な計算カラム) でインデックスを作成すると、チェックポイントが発生します。

カラムの統計情報が更新されます (統計情報が存在しない場合は作成されます)。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で **[インデックス]** を右クリックし、**[新規]** » **[インデックス]** をクリックします。
3. **インデックス作成ウィザード**の指示に従います。

結果

新しいインデックスがテーブルの **[インデックス]** タブと **[インデックス]** に表示されます。新しいインデックスは、クエリで使用可能です。

参照

- 「CREATE INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「パフォーマンスのモニタリングと診断のツール」 145 ページ
- 「auto_commit_on_create_local_temp_index オプション」『SQL Anywhere サーバ データベース管理』

インデックスの検証

インデックスで参照されているすべてのローが、実際にテーブルに存在するかどうか検証できます。外部キーインデックスの場合は、対応するローがプライマリテーブルにあることも確認します。

前提条件

インデックスの所有者であるか、VALIDATE ANY OBJECT システム権限を持っている必要があります。

接続によるデータベースの変更がない場合のみ、検証を実行します。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[インデックス]** をダブルクリックします。
3. インデックスを右クリックして、**[検証]** をクリックします。
4. **[OK]** をクリックします。

結果

インデックスで参照されているすべてのローが、実際にテーブルに存在するかどうか検証するため、チェックが実行されます。外部キーインデックスの場合は、対応するローがプライマリテーブルにあることをチェックします。氏

参照

- 「VALIDATE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「検証ユーティリティ (dbvalid)」『SQL Anywhere サーバ データベース管理』

インデックスの再構築

テーブルまたはマテリアライズドビューでの広範囲の挿入と削除によりフラグメント化されたインデックスを再構築します。

前提条件

テーブルのインデックスを再構築するには、テーブルの所有者であるか、次のいずれかの権限を持っている必要があります。

- そのテーブルに対する REFERENCES 権限
- ALTER ANY INDEX システム権限
- ALTER ANY OBJECT システム権限

マテリアライズドビューのインデックスを再構築するには、マテリアライズドビューの所有者であるか、次のいずれかの権限を持っている必要があります。

- ALTER ANY INDEX システム権限
- ALTER ANY OBJECT システム権限

内容と備考

インデックスを再構築するときは、物理インデックスを再構築します。物理インデックスを使用するすべての論理インデックスは、再構築操作により恩恵を受けます。論理インデックスで再構築を実行する必要はありません。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[インデックス]** をダブルクリックします。
3. インデックスを右クリックして、**[再構築]** をクリックします。
4. **[OK]** をクリックします。

結果

断片化は削除され、インデックスが再構築されます。

参照

- 「高度：論理インデックスと物理インデックス」36 ページ
- 「REORGANIZE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「インデックスの断片化とスキューの削減」245 ページ
- 「アプリケーションプロファイリングウィザードの使用」149 ページ
- 「sa_index_density システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

インデックスの削除

インデックスが必要なくなった場合、またはプライマリキーまたは外部キーの一部であるカラムの定義を変更する必要がある場合、インデックスを削除します。

前提条件

テーブルのインデックスを削除するには、テーブルの所有者であるか、次のいずれかの権限を持っている必要があります。

- そのテーブルに対する REFERENCES 権限
- DROP ANY INDEX システム権限
- DROP ANY OBJECT システム権限

外部キー、プライマリキー、または一意性制約のインデックスを削除するには、テーブルの所有者であるか、次のいずれかの権限を持っている必要があります。

- そのテーブルに対する ALTER 権限
- ALTER ANY TABLE システム権限
- ALTER ANY OBJECT システム権限

マテリアライズドビューのインデックスを削除するには、マテリアライズドビューの所有者であるか、次のいずれかの権限を持っている必要があります。

- DROP ANY INDEX システム権限
- DROP ANY OBJECT システム権限

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[インデックス]** をダブルクリックします。
3. インデックスを右クリックして、**[削除]** をクリックします。
4. **[はい]** をクリックします。

結果

インデックスはデータベースから削除されます。

次の手順

プライマリキーまたは外部キーの一部であるカラムの定義を削除または変更するため、インデックスを削除しなければならない場合、新しいインデックスを追加する必要があります。

参照

- 「DROP INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「インデックスの作成」31 ページ

高度：カタログ内のインデックス情報

ISYSIDX システムテーブルは、プライマリキーインデックスや外部キーインデックスを含む、データベース内にあるすべてのインデックスのリストを提供します。インデックスの補足情報

は、ISYSPHYSIDX、ISYSIDXCOL、ISYSFKEY の各システムテーブルにあります。Sybase Central または Interactive SQL を使用すると、これらのテーブルのシステムビューをブラウズして、それに含まれるデータを確認できます。

次に、インデックス情報がシステムテーブルに格納される仕組みの概要について説明します。

- **ISYSIDX システムテーブル**　インデックスを追跡するための中央テーブルです。ISYSIDX システムテーブルの各ローは、データベース内の論理インデックス (PKEY、FKEY、一意性制約、セカンダリインデックス) を定義します。
- **ISYSPHYSIDX システムテーブル**　ISYSPHYSIDX システムテーブルの各ローは、データベース内の物理インデックスを定義します。
- **ISYSIDXCOL システムテーブル**　SYSIDX システムビューの各ローがデータベース内のインデックス 1 つを示すように、SYSIDXCOL システムビューの各ローは、SYSIDX システムビューで記述されているインデックスのカラム 1 つを示します。
- **ISYSFKEY システムテーブル**　データベース内の各外部キーは、ISYSFKEY システムテーブル内のロー 1 つと、ISYSIDX システムテーブル内のロー 1 つによって定義されます。

参照

- 「SYSIDX システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSPHYSIDX システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSIDXCOL システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSFKEY システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「高度：論理インデックスと物理インデックス」36 ページ

高度：論理インデックスと物理インデックス

SQL Anywhere は物理インデックスと論理インデックスを使用します。物理インデックスは、インデックスがディスクに保存されるときの実際のインデックス構造です。論理インデックスは、物理インデックスへの参照です。プライマリキー、セカンダリキー、外部キー、一意性制約を作成するときに、データベースサーバは、制約の論理インデックスを作成することで参照整合性を確保します。次に、データベースサーバは制約を満たすインデックスがすでに存在するかどうかを確認します。条件を満たす物理インデックスがすでに存在する場合、データベースサーバはその物理インデックスへの論理インデックスを指します。そのような物理インデックスが存在しない場合、データベースサーバは新しい物理インデックスを作成してから、その物理インデックスへの論理インデックスを指します。

物理インデックスが論理インデックスの要件を満たすには、カラムとカラムの順序、および各カラムのデータの順序 (昇順や降順) が同一である必要があります。

データベース内のすべての論理インデックスと物理インデックスの情報は、それぞれ ISYSIDX システムテーブルと ISYSPHYSIDX システムテーブルに記録されます。論理インデックスを作成すると、そのインデックス定義を保持するためにエントリが ISYSIDX システムテーブルに作成されます。論理インデックスを満たすために使用される物理インデックスへの参照は、ISYSIDX.phys_id カラムに記録されます。物理インデックスは ISYSPHYSIDX システムテーブルに定義されます。

複数の論理インデックスは単一の物理インデックスを指すことができるため、論理インデックスを使用するということは、データベースサーバは重複した物理インデックスを作成して管理する必要がないということを意味します。

論理インデックスを削除すると、その定義が ISYSIDX システムテーブルから削除されます。特定の物理インデックスを参照するだけの論理インデックスの場合は、その物理インデックスと、ISYSPHYIDX システムテーブル内で対応するエントリも削除されます。

リモートテーブルに対して物理インデックスは作成されません。テンポラリテーブルの場合、物理インデックスは作成されますが、ISYSPHYSIDX に記録されず、使用後に廃棄されます。また、テンポラリテーブルの物理インデックスは共有されません。

参照

- 「SYSIDX システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSPHYSIDX システムビュー」『SQL Anywhere サーバ SQL リファレンス』

物理インデックスを共有する論理インデックスの特定

インデックスを削除すると、論理インデックスが削除されますが、その参照先の物理インデックスも常に削除されるとはかぎりません。別の論理インデックスが同じ物理インデックスを参照している場合、その物理インデックスは削除されません。インデックスを削除することでディスク領域が解放されることを期待していたり、物理的に再作成するためにインデックスを削除しようとしたりする場合は特に注意する必要があります。

テーブルのインデックスが他のインデックスと物理インデックスを共有しているか判断するには、Sybase Central でテーブルを選択し、**[インデックス]** タブをクリックします。インデックスの [物理 ID] 値が、リスト内の別のインデックスに対しても示されているかどうかに注意してください。[物理 ID] の値が一致するということは、それらのインデックスが同じ物理インデックスを共有しているということです。物理インデックスを再作成する場合は、ALTER INDEX...REBUILD 文を使用できます。また、すべてのインデックスを削除してから再作成する方法もあります。

物理インデックスが共有されているテーブルの特定

次のようなクエリを実行することで、物理インデックスが共有されているすべてのテーブルのリストをいつでも取得できます。

```
SELECT tab.table_name, idx.table_id, phys.phys_index_id, COUNT(*)
 FROM SYSIDX idx JOIN SYSTAB tab ON (idx.table_id = tab.table_id)
 JOIN SYSPHYSIDX phys ON ( idx.phys_index_id  = phys.phys_index_id
   AND idx.table_id = phys.table_id )
 GROUP BY tab.table_name, idx.table_id, phys.phys_index_id
   HAVING COUNT(*) > 1
ORDER BY tab.table_name;
```

このクエリの結果セットの例を次に示します。

table_name	table_id	phys_index_id	COUNT()
ISYSCHECK	57	0	2

table_name	table_id	phys_index_id	COUNT()
ISYSCOLSTAT	50	0	2
ISYSFKEY	6	0	2
ISYSSOURCE	58	0	2
MAINLIST	94	0	3
MAINLIST	94	1	2

各テーブルのローの数は、テーブルの共有物理インデックスの数を示します。この例では、すべてのテーブルに共有物理インデックスが 1 つありますが、架空のテーブル MAINLIST には 2 つあります。phys_index_id 値は、共有されている物理インデックスを表し、COUNT カラムの値は、物理インデックスを共有している論理インデックスの数を表します。

当該テーブルで物理インデックスを共有しているインデックスを確認するには、Sybase Central を使用する方法もあります。このことを行うには、左ウィンドウ枠でテーブルを選択し、右ウィンドウ枠で **[インデックス]** タブをクリックし、次に [物理 ID] カラムの値が同じであるローが複数あるかどうかを確認します。[物理 ID] の値が同じインデックスは、同じ物理インデックスを共有しています。

参照

- 「インデックスの再構築」33 ページ
- 「ALTER INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSIDX システムビュー」『SQL Anywhere サーバ SQL リファレンス』

高度：インデックスの選択性とインデックスファンアウト

「インデックスの選択性」とは、追加データを読み込まないで必要なインデックスエントリを検索するインデックスの機能です。

選択性が低い場合は、インデックスが参照するテーブルページから追加情報を取り出します。このような取り出しは「完全比較」と呼ばれ、インデックスのパフォーマンスを低下させます。

FullCompare プロパティは、発生した完全比較の数を追跡します。また、この統計は Sybase Central パフォーマンスモニタまたは Windows パフォーマンスモニタを使用してモニタできます。

注意
Windows パフォーマンスモニタは、Windows Mobile では使用できません。

さらに、完全比較の数は、統計情報付きのグラフィカルなプランの形式で提供されます。

インデックスは、ツリーのようにいくつかのレベルで編成されています。インデックスの最初のページはルートページと呼ばれ、その次の下位レベルの 1 つ以上のページへと分岐して、さらに

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

それが分岐して、インデックスの最下位レベルに達します。最下位レベルのインデックスページは、リーフページと呼ばれます。*n* レベルを持つインデックスの場合、特定のローを探すにはインデックスページを *n* 回読み込む必要があり、実際のローを含むデータページを 1 回読み込む必要があります。通常、使用頻度の高いインデックスページはキャッシュに格納される傾向があるため、ディスクからの読み込みは *n* 回よりも少なくて済みます。

「インデックスファンアウト」は、1 ページに格納されるインデックスエントリの数です。ファンアウトが大きなインデックスは、ファンアウトが小さなインデックスよりレベル数が少なくなります。したがって、通常はインデックスファンアウトが大きいほど、インデックスのパフォーマンスが向上します。データベースに適切なページサイズを選択すると、インデックスファンアウトを向上できます。

インデックスレベル数を確認するには、sa_index_levels システムプロシージャを使用します。

参照

- 「プランに使用される一般的な統計」358 ページ
- 「ヒント：適切なページサイズの使用」250 ページ
- 「sa_index_levels システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

高度 : SQL Anywhere がインデックスを使用するその他の方法

SQL Anywhere はインデックスを使用してその他のパフォーマンスを高めています。インデックスを使用すると、SQL Anywhere はカラムの一意性を強化し、ロックするローとページの数を減らし、述部の選択性を適切に推定できます。

- **カラム一意性の強化**　インデックスがないと、SQL Anywhere は、値が挿入されるたびにテーブル全体をスキャンし、その値がユニークであることを確認する必要があります。このため、SQL Anywhere は一意性制約付きで各カラムのインデックスを自動的に作成します。
- **ロックの減少**　インデックスによって、挿入、更新、削除中にロックされるローとページの数が減少します。これは、インデックスがテーブルに適用する順序付けによるものです。
- **選択性の推定**　インデックスは順序付けされているため、オプティマイザはインデックスの上位レベルをスキャンすることで、特定のクエリを満たす値のパーセンテージを推定できます。このアクションは、部分インデックススキャンと呼ばれます。

参照

- 「ロックの仕組み」886 ページ

ビュー

ビューとは、ビュー定義の結果セットによって定義される計算テーブルです。ビュー定義は SQL クエリとして表現されます。ビューを使用して、データベースのユーザに正確な情報を、制御できるフォーマットで表示できます。SQL Anywhere では、「通常のビュー」と「マテリアライズドビュー」の 2 種類のビューをサポートします。

データベース内の各ビューの定義は、ISYSVIEW システムテーブルに格納されます。

参照

- 「SYSVIEW システムビュー」『SQL Anywhere サーバ SQL リファレンス』

ビューの表記規則

「「通常のビュー」」という用語は、ビューを参照するたびに再計算されるビューの説明に使用され、結果セットはディスクに保存されません。これは最も一般的に使用される種類のビューです。マニュアルのほとんどの部分は、通常のビューを指しています。

「マテリアライズドビュー」という用語は、結果セットが事前に計算され、ベーステーブルの内容と同様にディスク上に実体化されるビューの説明に使用されます。

マニュアル内での「ビュー」 (そのもの) の意味は、文脈に基づきます。通常のビューとマテリアライズドビューの共通の事項に関するセクションで使用される場合は、通常のビューとマテリアライズドビューの両方を指します。用語がマテリアライズドビューのマニュアルで使用されている場合はマテリアライズドビューを指し、通常のビューのマニュアルの場合は通常のビューを指します。

通常のビュー、マテリアライズドビュー、ベーステーブルの機能

次の表では、通常のビュー、マテリアライズドビュー、テーブルの機能の比較を示します。

許可内容	通常のビュー	マテリアライズドビュー	テーブル
アクセス権限	○	○	○
SELECT	○	○	○
UPDATE	一部	×	○
INSERT	一部	×	○
DELETE	一部	×	○
従属ビュー	○	○	○
インデックス	×	○	○
整合性制約	×	×	○
キー	×	×	○

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

ビューを使用する利点

ビューを使用すると、データベース中のデータへのアクセスを調整できます。アクセスの調整にはいくつか目的があります。

- **効率的なリソース使用**　通常のビューでは、データを格納するために追加の記憶領域は必要ありません。呼び出しごとに再計算されます。マテリアライズドビューはディスク領域を必要としますが、呼び出しごとに再計算する必要はありません。データベースのサイズが大きく、同じテーブルに対して頻繁に繰り返し発生するジョイン要求をデータベースサーバが処理するような環境では、マテリアライズドビューを使用すると、応答時間が向上します。

- **セキュリティの向上**　関連データへのアクセスだけが許可されます。

- **利便性の向上**　ベーステーブルよりも見やすい形でユーザとアプリケーション開発者にデータを提供します。

- **一貫性の向上**　よく使われるクエリの定義をデータベース内で集中管理します。

ビューの依存性

ビュー定義は、カラム、テーブル、他のビューなど、その他のオブジェクトを参照することがあります。ビューが別のオブジェクトを参照している場合、そのビューは「参照元のオブジェクト」、ビューが参照しているオブジェクトは「参照先のオブジェクト」と呼ばれます。また、参照元のオブジェクトは、参照先のオブジェクトに「依存して」います。

あるビューの参照先オブジェクトのセットには、そのビューが直接的または間接的に参照するすべてのオブジェクトが含まれます。たとえば、ビューはあるテーブルを参照する別のビューを参照することにより、間接的にそのテーブルを参照できます。

次のテーブルとビューのセットを考えてみます。

```
CREATE TABLE t1 ( c1 INT, c2 INT );
CREATE TABLE t2( c3 INT, c4 INT );
CREATE VIEW v1 AS SELECT * FROM t1;
CREATE VIEW v2 AS SELECT c3 FROM t2;
CREATE VIEW v3 AS SELECT c1, c3 FROM v1, v2;
```

次のビューの依存性は、上記の定義から特定できます。

- ビュー v1 は t1 の各カラムと t1 自体に依存しています。

- ビュー v2 は、t2.c3 と t2 自体に依存しています。

- ビュー v3 はカラム t1.c1 と t2.c3、テーブル t1 と t2、ビュー v1 と v2 に依存しています。

データベースサーバは指定されたビューが参照するカラム、テーブル、ビューを追跡します。データベースサーバではこの依存性情報を使用して、参照先オブジェクトのスキーマ変更が使用できない状態で参照元ビューに残らないようにします。

依存性とスキーマ変更

テーブルやビューに定義したスキーマを変更しようとする場合、従属ビューに対して変更による影響があるかどうかをデータベースサーバが検討する必要があります。スキーマ変更操作の例を次に示します。

- テーブル、ビュー、マテリアライズドビュー、またはカラムの削除
- テーブル、ビュー、マテリアライズドビュー、またはカラムの名前変更
- カラムの追加、削除、または変更
- カラムのデータ型、サイズ、または NULL 入力属性の変更
- ビュー、またはテーブルのビューの依存性の無効化

スキーマ変更操作中に発生するイベント

1. データベースサーバは、変更するテーブルやビューに直接的または間接的に依存するビューのリストを生成します。DISABLED ステータスのビューは無視されます。

   いずれかの従属ビューがマテリアライズドビューである場合、要求は失敗し、エラーが返され、残りのイベントは発生しません。従属したマテリアライズドビューを明示的に無効にしてから、スキーマ変更操作を続行してください。

2. データベースサーバは、変更するオブジェクトとすべての従属した通常のビューに対して、排他スキーマロックを取得します。

3. データベースサーバはすべての従属した通常のビューのステータスを INVALID に設定します。

4. データベースサーバはスキーマ変更操作を実行します。操作が失敗すると、ロックは解放され、従属した通常のビューのステータスは VALID にリセットされます。さらにエラーが返され、残りの手順は発生しません。

5. データベースサーバは従属した通常のビューを再コンパイルし、成功した場合は各ビューのステータスを VALID に設定します。いずれかの通常のビューでコンパイルが失敗した場合も、そのビューのステータスは INVALID のままです。INVALID である通常のビューに対する以後の要求により、データベースサーバはビューを再コンパイルしようとします。それらの試行に失敗した場合は、INVALID ビューまたはそのビューが依存するオブジェクトを変更する必要があります。

通常のビュー：依存性とスキーマ変更

- 通常のビューは、マテリアライズドビューを含め、テーブルやビューを参照できる。
- テーブルまたはビューのスキーマを変更すると、データベースはすべての参照元の通常ビューを自動的に再コンパイルしようとする。
- ビューまたはテーブルを無効にするか削除すると、すべての従属した通常のビューが自動的に無効になる。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

- ALTER TABLE 文の DISABLE VIEW DEPENDENCIES 句を使用して、従属した通常のビューを無効にできる。

マテリアライズドビュー：依存性とスキーマ変更

- マテリアライズドビューは、ベーステーブルだけを参照する。
- 有効なマテリアライズドビューが参照している場合は、ベーステーブルに対するスキーマ変更は許可されない。テーブルに外部キーを追加することはできる (ALTER TABLE、ADD FOREIGN KEY など)。
- テーブルを削除する前に、すべての従属マテリアライズドビューを無効にするか削除する必要がある。
- ALTER TABLE 文の DISABLE VIEW DEPENDENCIES 句は、マテリアライズドビューには影響しない。マテリアライズドビューを無効にするには、ALTER MATERIALIZED VIEW...DISABLE 文を使用する必要があります。
- マテリアライズドビューを無効にした場合は、ALTER MATERIALIZED VIEW...ENABLE 文を実行するなどして、再度明示的に有効にする必要がある。

参照

- 「マテリアライズドビューの有効化または無効化」64 ページ

依存性情報の取得 (SQL の場合)

データベースのテーブルまたはビューの場合、そのオブジェクトに依存するオブジェクトのリストを取得できます。テーブルまたはビューを変更し、影響を受ける他のオブジェクトを把握する必要がある場合、これは便利です。

前提条件

この作業の実行に権限は必要なく、PUBLIC でカタログにアクセスできることが想定されています。

内容と備考

SYSDEPENDENCY システムビューは依存性情報を格納します。SYSDEPENDENCY システムビューの各ローは、2 つのデータベースオブジェクト間の依存性を示します。直接依存性とは、あるオブジェクトがその定義内で別のオブジェクトを直接参照していることです。データベースサーバは直接依存性情報を使用して、間接依存性を判断します。たとえば、ビュー A はビュー B を参照し、ビュー B はテーブル C を参照しているとします。この場合、ビュー A はビュー B に直接依存していて、テーブル C に間接依存しています。

♦ タスク

1. データベースに接続します。
2. sa_dependent_views システムプロシージャを呼び出す文を実行します。

結果

従属ビューの ID リストが返されます。

例

次の例では、sa_dependent_views システムプロシージャを SELECT 文で使用して、SalesOrders テーブルに依存するビューの名前リストを取得します。このプロシージャは、ViewSalesOrders ビューを返します。

```
SELECT t.table_name FROM SYSTAB t,
sa_dependent_views( 'SalesOrders' ) v
WHERE t.table_id = v.dep_view_id;
```

参照

- 「SYSDEPENDENCY システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_dependent_views システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

通常のビュー

データをブラウズすると、クエリは 1 つ以上のデータベースオブジェクトで動作し、結果セットを作成します。ベーステーブルと同様に、クエリの結果セットにもカラムとローがあります。ビューを作成するクエリには名前が与えられ、システムテーブルに定義が格納されます。

通常のビューの作成時に、データベースサーバはビュー定義をデータベースに格納します。ただしビューのデータは格納されません。ビュー定義は、そのビュー定義が参照される場合で、かつそのビューの使用中のみ実行されます。ビューの作成時は、データベースに重複したデータを格納する必要がありません。

各部署の従業員数をリストすることが頻繁にあるとします。このリストは次の文で取得できます。

```
SELECT DepartmentID, COUNT(*)
FROM Employees
GROUP BY DepartmentID;
```

通常のビューの SELECT 文に対する制限

通常のビューとして使用できる SELECT 文には制限があります。特に、SELECT クエリ中では ORDER BY 句を使用できません。リレーショナルテーブルでは、ローやカラムの並び順には意味がありませんが、ORDER BY 句を使用すると、ビューのローの順序が規定されるからです。GROUP BY 句、サブクエリ、ジョインは、ビューの定義で使用できます。

ビューを作成するには、必要とする正確な結果が必要なフォーマットで得られるまで SELECT クエリを編集します。SELECT 文が作成できたら、先頭に次の句を追加するとビューが完成します。

```
CREATE VIEW view-name AS query;
```

通常のビューを更新する文

ビューを定義するクエリ指定が更新可能な場合、UPDATE 文、INSERT 文、DELETE 文を使用してビューを更新できます。ビューを定義するクエリ指定に次のいずれかが含まれている場合、そのビューは派生の関係で**更新不可能**となります。

- UNION、EXCEPT、または INTERSECT
- DISTINCT 句
- GROUP BY 句
- WINDOW 句
- FIRST、TOP、または LIMIT 句
- 集合関数
- FROM 句に複数のテーブル (ansi_update_constraints オプションが Strict または Cursors に設定されている場合)。
- ORDER BY 句 (ansi_update_constraints オプションが Strict または Cursors に設定されている場合)。
- すべての SELECT リスト項目がベーステーブルのカラムではない。

WITH CHECK OPTION 句

ビューを作成する場合、WITH CHECK OPTION 句は、ビューを介したベーステーブルに対する挿入時または更新時に変更されるデータの制御に役立ちます。次の例で説明します。

WITH CHECK OPTION 句を使用して次の文を実行し、SalesEmployees ビューを作成します。

```
CREATE VIEW SalesEmployees AS
  SELECT EmployeeID, GivenName, Surname, DepartmentID
  FROM Employees
  WHERE DepartmentID = 200
  WITH CHECK OPTION;
```

選択すると、このビューの内容が次のように表示されます。

```
SELECT * FROM SalesEmployees;
```

EmployeeID	GivenName	Surname	DepartmentID
129	Philip	Chin	200
195	Marc	Dill	200
299	Rollin	Overbey	200
467	James	Klobucher	200

EmployeeID	GivenName	Surname	DepartmentID
...	...	...	...

次に、Philip Chin の DepartmentID を 400 に更新しようとします。

```
UPDATE SalesEmployees
SET DepartmentID = 400
WHERE EmployeeID = 129;
```

WITH CHECK OPTION が指定されたため、データベースサーバは、この更新によってビュー定義 (この場合は WHERE 句の式) で違反が発生するかどうかを評価します。DepartmentID は 200 でなければならないため、この文は失敗し、データベースサーバが「ベーステーブル 'Employees' の挿入／更新に対して WITH CHECK OPTION が違反しています。」というエラーを返します。

ビュー定義で WITH CHECK OPTION を指定しなかった場合は更新操作が実行され、Employees テーブルが新しい値で修正されてしまい、これ以降 Philip Chin がビューから消えてしまいます。

SalesEmployees ビューを参照するビュー (たとえば View2) が作成された場合は、View2 自体に WITH CHECK OPTION 句がなくても、SalesEmployees ビューの WITH CHECK OPTION の基準に合わなければ、View2 に対する更新や挿入は拒否されます。

参照

- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「クエリ結果の要約、グループ化、ソート」468 ページ
- 「マテリアライズドビュー」53 ページ
- 「ansi_update_constraints オプション」『SQL Anywhere サーバ データベース管理』

通常のビューのステータス

通常のビューには、ステータスが関連付けられています。ステータスは、データベースサーバが使用するビューの利用可能性を反映しています。すべてのビューのステータスを表示するには、Sybase Central の左ウィンドウ枠で **[ビュー]** をクリックし、右ウィンドウ枠で **[ステータス]** カラムの値を検査します。また、単一のビューのステータスを表示するには、Sybase Central でビューを右クリックし、**[プロパティ]** をクリックして **[ステータス]** の値を検査します。

通常のビューのステータスの種類について、次に説明します。

- **VALID**　ビューは有効で、その定義と一貫性があることが保証されています。データベースサーバは、追加の作業なくこのビューを利用できます。有効にされたビューのステータスは VALID です。

  SYSOBJECT システムビューで、値 1 はステータス VALID を表します。

- **INVALID**　INVALID ステータスは、参照先オブジェクトのスキーマ変更後に発生し、変更したためにビューを有効にしようとして失敗したことを表します。たとえば、ビュー v1 がテーブル t 内のカラム c1 を参照するとします。t を変更して c1 を削除する場合、カラムを削除する ALTER 操作の一環としてデータベースサーバがビューを再コンパイルすると、v1 のステータスは INVALID に設定されます。このとき、v1 は c1 が t に追加された場合だけ再コン

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

パイルできます。再コンパイルしない場合、v1 は c1 を参照しないように変更されます。ビューが参照するテーブルやビューを削除した場合も、そのビューは INVALID になります。

INVALID ビューは DISABLED ビューと異なり、クエリなどで INVALID ビューが参照されるたびに、データベースサーバはそのビューを再コンパイルしようとします。コンパイルに成功すると、クエリが処理されます。ビューを明示的に有効にしないかぎり、ステータスは INVALID のままです。失敗した場合は、エラーが返されます。

データベースサーバは、INVALID ビューを内部的に有効にすると、パフォーマンス警告を発行します。

SYSOBJECT システムビューで、値 2 はステータス INVALID を表します。

- **DISABLED**　無効にされたビューは、データベースサーバがクエリに応答するために使用できません。無効にされたビューを使用しようとするクエリは、エラーを返します。

  通常のビューは、次の場合にこのステータスになります。

  ○ビューを明示的に無効にした場合。ALTER VIEW...DISABLE 文を実行した場合など。

  ○そのビューが依存するビュー (マテリアライズドビューまたは非マテリアライズドビュー) を無効にした場合。

  ○テーブルのビューの依存性を無効にした場合。ALTER TABLE...DISABLE VIEW DEPENDENCIES 文を実行した場合など。

  SYSOBJECT システムビューで、値 4 はステータス DISABLED を表します。

参照

- 「SYSOBJECT システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「通常のビューの無効化または有効化 (SQL の場合)」51 ページ

通常のビューの作成

1 つ以上のソースからデータを結合するビューを作成します。ビューによりパフォーマンスが向上し、ユーザが問い合わせできるデータを管理できます。

前提条件

ユーザ本人が所有するビューを作成するには、CREATE VIEW システム権限が必要です。他のユーザが所有するビューを作成するには、CREATE ANY VIEW または CREATE ANY OBJECT のシステム権限が必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で **[ビュー]** を右クリックし、**[新規]** » **[ビュー]** をクリックします。

3. **ビュー作成ウィザード**の指示に従います。

4. 右ウィンドウ枠で、**[SQL]** タブをクリックし、定義のコードを編集します。変更を保存するには、**[ファイル]** » **[保存]** をクリックします。

結果

作成したビューの定義がデータベースに追加されます。クエリがビューを参照するたびに、定義を使用してビューにデータを取り込み、結果を返します。

次の手順

ビューに問い合わせて結果を検証し、正しいデータが返されたか確認します。

参照

- 「CREATE VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「auto_commit_on_create_local_temp_index オプション」『SQL Anywhere サーバ データベース管理』

通常のビューの変更

データベースの定義を編集することで、通常のビューを編集します。たとえば、ビューに追加のテーブルのデータを含める場合、ビュー定義を更新して、テーブルデータをビュー定義の既存のデータソースにジョインさせます。

前提条件

そのビューの所有者であるか、または次のいずれかの権限を持っていることが必要です。

- ALTER ANY VIEW システム権限
- ALTER ANY OBJECT システム権限

内容と備考

ビュー定義が古い場合(基本となるデータのスキーマ変更によりコンパイルしない)、カラムを追加または削除したり、設定に関連した変更を実行したりしなければならない場合があります。

既存のビューの名前を変更することはできません。代わりに、新しい名前を付けて新しくビューを作成し、以前の定義をそこにコピーしてから、元のビューを削除します。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。

2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。

3. ビューを選択します。

4. 右ウィンドウ枠で、**[SQL]** タブをクリックし、定義のコードを編集します。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

> **ヒント**
> 複数のビューを編集する場合は、各ビューを右ウィンドウ枠の **[SQL]** タブで編集するより、各ビューに対して別のウィンドウを開く方が便利な場合があります。別のウィンドウを開くには、ビューを選択し、**[ファイル]** » **[新しいウィンドウで編集]** をクリックします。

5. **[ファイル]** » **[保存]** をクリックします。

結果

データベース内のビューの定義が更新されます。

次の手順

ビューに問い合わせて結果を検証し、正しいデータが返されたか確認します。

通常のビューを変更する場合、ビューに他のビューの依存性が存在するときは、変更後に追加の操作が必要なことがあります。たとえば、ビューの変更後、データベースサーバはそのビューを自動的に再コンパイルして、データベースサーバが使用できるように有効にします。従属した通常のビューが存在する場合、データベースサーバはそれらも無効にしてからもう一度有効にします。有効にできない場合、ステータスは INVALID になるため、通常のビューの定義と従属した通常のビューの定義が一貫性を保つようにする必要があります。通常のビューに従属ビューが存在するかどうかを判断するには、sa_dependent_views システムプロシージャを使用します。

参照

- 「ビューの依存性」41 ページ
- 「sa_dependent_views システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「通常のビューの削除」49 ページ

通常のビューの削除

ビューが必要なくなったら、削除します。ビューの名前を変更する場合も、ビューを削除(および再作成)してください。

前提条件

所有者であるか、または DROP ANY VIEW と DROP ANY OBJECT のいずれかのシステム権限を持っている必要があります。

ビューを削除する前に、ビューを参照するすべての INSTEAD OF トリガを削除してください。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. ビューを右クリックして、**[削除]** をクリックします。

4. **[はい]** をクリックします。

結果

通常のビューの定義が、データベースから削除されます。

次の手順

従属ビューを持つ通常のビューを削除すると、削除操作の一環としてその従属ビューは INVALID になります。従属ビューは、変更されるか、元の削除済みビューが再作成されるまで、使用できません。

通常のビューに従属ビューが存在するかどうかを判断するには、sa_dependent_views システムプロシージャを使用します。

参照

- 「sa_dependent_views システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「通常のビューの変更」48 ページ
- 「DROP VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ビューの依存性」41 ページ
- 「トリガの削除」101 ページ
- 「DROP TRIGGER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「INSTEAD OF トリガ」105 ページ

通常のビューの無効化または有効化 (Sybase Central の場合)

通常のビューを有効または無効にすることで、データベースサーバがそのビューを使用できるかどうかを制御できます。

前提条件

所有者であるか、または次のいずれかの権限を持っていることが必要です。

- ALTER ANY VIEW システム権限
- ALTER ANY OBJECT システム権限

通常のビューを「有効にする」には、基本となるテーブルに対する SELECT 権限または SELECT ANY TABLE システム権限も必要です。

通常のビューを有効にする前に、そのビューが参照する無効なビューを再度有効にする必要があります。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. 通常のビューを無効にするには、ビューを右クリックし、**[無効にする]** をクリックします。

4. 通常のビューを有効にするには、ビューを右クリックし、**[再コンパイルして有効にする]** をクリックします。

結果

通常のビューを無効にした場合、データベースサーバはデータベース内のビューの定義を保持しますが、クエリを満たすときにそのビューを使用できません。

無効なビューをクエリが明示的に参照すると、そのクエリは失敗し、エラーが返されます。

次の手順

このため、ビューを再度有効にしたら、無効にした時点でそのビューに依存していた他のすべてのビューを、もう一度有効にする必要があります。ビューを無効にする前に sa_dependent_views システムプロシージャを使用することで、従属ビューのリストを特定できます。

通常のビューを有効にすると、データベースサーバはデータベース内に格納されたビューの定義を使用してそのビューを再コンパイルします。コンパイルが成功すると、ビューのステータスが VALID に変更されます。再コンパイルに失敗した場合、1 つ以上の参照先オブジェクトでスキーマが変更された可能性があります。その場合は、ビュー定義と参照先のオブジェクトのどちらかを変更し、相互に整合させてから、ビューを有効にします。

ビューが無効になったら、データベースサーバがそのビューを使用できるように、明示的にもう一度有効にする必要があります。

参照

- 「sa_dependent_views システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSDEPENDENCY システムビュー」『SQL Anywhere サーバ SQL リファレンス』

通常のビューの無効化または有効化 (SQL の場合)

通常のビューを有効または無効にすることで、データベースサーバがそのビューを使用できるかどうかを制御できます。

前提条件

所有者であるか、または次のいずれかの権限を持っていることが必要です。

- ALTER ANY VIEW システム権限
- ALTER ANY OBJECT システム権限

通常のビューを「有効にする」には、基本となるテーブルに対する SELECT 権限または SELECT ANY TABLE システム権限も必要です。

通常のビューを有効にする前に、そのビューが参照する無効なビューを再度有効にする必要があります。

内容と備考

ビューを無効にすると、そのビューを直接的または間接的に参照する他のビューも自動的に無効になります。このため、ビューを再度有効にしたら、無効にした時点でそのビューに依存していた他のすべてのビューを、もう一度有効にする必要があります。ビューを無効にする前に sa_dependent_views システムプロシージャを使用することで、従属ビューのリストを特定できます。

♦ タスク

1. データベースに接続します。
2. 通常のビューを無効にするには、ALTER VIEW...DISABLE 文を実行します。
3. 通常のビューを有効にするには、ALTER VIEW...ENABLE 文を実行します。

結果

通常のビューを無効にした場合、データベースサーバはデータベース内のビューの定義を保持しますが、クエリを満たすときにそのビューを使用できません。

無効なビューをクエリが明示的に参照すると、そのクエリは失敗し、エラーが返されます。

次の手順

このため、ビューを再度有効にしたら、無効にした時点でそのビューに依存していた他のすべてのビューを、もう一度有効にする必要があります。ビューを無効にする前に sa_dependent_views システムプロシージャを使用することで、従属ビューのリストを特定できます。

通常のビューを有効にすると、データベースサーバはデータベース内に格納されたビューの定義を使用してそのビューを再コンパイルします。コンパイルが成功すると、ビューのステータスが VALID に変更されます。再コンパイルに失敗した場合、1 つ以上の参照先オブジェクトでスキーマが変更された可能性があります。その場合は、ビュー定義と参照先のオブジェクトのどちらかを変更し、相互に整合させてから、ビューを有効にします。

ビューが無効になったら、データベースサーバがそのビューを使用できるように、明示的にもう一度有効にする必要があります。

例

次の例では、GROUPO が所有する通常のビュー ViewSalesOrders が無効になります。

```
ALTER VIEW GROUPO.ViewSalesOrders DISABLE;
```

次の例では、GROUPO が所有する通常のビュー ViewSalesOrders がもう一度有効になります。

```
ALTER VIEW GROUPO.ViewSalesOrders ENABLE;
```

参照

- 「sa_dependent_views システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSDEPENDENCY システムビュー」『SQL Anywhere サーバ SQL リファレンス』

通常のビュー内のデータのブラウズ

通常のビュー内のデータをブラウズできます。通常のビューは、ビューの定義としてデータベースに格納されます。ビューのデータが最新になるように、問い合わせのとき、ビューにデータが取り込まれます。

前提条件

通常のビューは定義済みで、有効になったビューである必要があります。

所有者であるか、または次のいずれかの権限を持っていることが必要です。

- そのビューに対する SELECT 権限
- SELECT ANY TABLE システム権限

内容と備考

この作業は、表示する通常のビューを要求する Sybase Central で開始し、通常のビューのデータが表示される Interactive SQL で完了します。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をクリックします。
3. ビューを選択して、**[ファイル]** » **[Interactive SQL によるデータ表示]** をクリックします。

結果

InteractiveSQL が起動して、**[結果]** ウィンドウ枠の **[結果]** タブにビューの内容が表示されます。

参照

- 「通常のビュー」44 ページ
- 「通常のビューの無効化または有効化 (Sybase Central の場合)」50 ページ
- 「クエリ」281 ページ
- 「Interactive SQL」『SQL Anywhere サーバ データベース管理』
- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』

マテリアライズドビュー

「マテリアライズドビュー」とは、ベーステーブルとよく似ていて、参照先のベーステーブルから結果セットが事前に計算されてディスクに格納されるビューです。概念としては、マテリアラ

イズドビューはビューでもあり (カタログに格納されたクエリ指定がある)、テーブルでもあります (永続的な実体化したローがある)。したがって、テーブルで実行する多くの操作は、マテリアライズドビューでも実行できます。たとえば、マテリアライズドビューに対して、インデックスを構築できます。

マテリアライズドビューを作成すると、データベースサーバが定義を検証して正しく準拠していることを確認します。すべてのカラムとテーブルの参照はデータベースサーバによって完全に修飾されるため、ビューにアクセスするすべてのユーザが同一の定義を確認できます。マテリアライズドビューの作成に成功したら、データを設定します。データの設定は、ビューの「初期化」とも呼ばれます。

マテリアライズドビューは、Sybase Central の **[ビュー]** フォルダにリストされます。

参照

● 「ヒント：マテリアライズドビューを使ったクエリのパフォーマンス改善」225 ページ

マテリアライズドビューを使用したパフォーマンスの向上

適切な条件でマテリアライズドビューを使用すると、ジョインなどのコストの高い操作を事前に計算し、その結果をディスクに保管されるビューの形式で格納することによって、パフォーマンスを大幅に向上させることができます。オプティマイザでは、クエリを満たす最も効率的な方法を決定するときに、クエリ内でマテリアライズドビューが参照されていなくても、マテリアライズドビューを検討します。

アプリケーションの設計では、負荷の高い集約操作やジョイン操作を含むクエリのように、コストの高いクエリやクエリでコストの高い部分を頻繁に実行する場合は、実体化ビュー (Materialized View) を定義することを検討してください。マテリアライズドビューは、次のような環境でのパフォーマンスを向上することを目的に設計されています。

●データベースのサイズが大きい。

●頻繁なクエリにより、大量のデータに対する集約操作やジョイン操作が繰り返し実行される。

●基本となるデータへの変更は頻度が比較的少ない。

●最新のデータにアクセスすることは重大な要件ではない。

マテリアライズドビューを使用する前に、次の要件、設定、制限を検討してください。

- **ディスク領域の要件**　マテリアライズドビューにはベーステーブルからのデータの複製が含まれるため、作成するマテリアライズドビューのサイズ分の領域をデータベースのディスク上に追加で割り付ける必要があることがあります。得られる利点がマテリアライズドビューの使用コストと釣り合うように、追加領域の要件は慎重に検討する必要があります。

- **保守コストとデータの最新性の要件**　マテリアライズドビューのデータは、基本となるテーブルのデータが変更されたときに再表示する必要があります。次のような競合要因を考慮の上、マテリアライズドビューを再表示しなければならない頻度を判断する必要があります。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

- ○ **基本となるデータの変更頻度**　データに対して頻繁な変更や大規模な変更が行われると、手動ビューが古くなります。データの最新性が重要な場合は、即時ビューの使用を検討します。

- ○ **再表示のコスト**　各マテリアライズドビューの基本となるクエリの複雑さや関係するデータの量に応じて、再表示に必要な計算のコストが非常に高くなることがあります。そのためマテリアライズドビューが頻繁に再表示されると、データベースサーバが耐えられないほどの負荷がかかる可能性があります。さらに、マテリアライズドビューは再表示操作中は使用できません。

- ○ **アプリケーションのデータ最新性の要件**　データベースサーバが古いマテリアライズドビューを使用すると、アプリケーションに対して古いデータを提示することになります。古いデータは、基本となるテーブル内のデータの現在の状態を表していません。古さの程度は、マテリアライズドビューが再表示される頻度によって決まります。高いパフォーマンスを実現するために、許容できる古さの程度を判断するようにアプリケーションを設計する必要があります。

- ○ **データの一貫性の要件**　マテリアライズドビューを再表示するときは、マテリアライズドビューを再表示しなければならない一貫性を判断する必要があります。

- **最適化の使用**　クエリの実行時にオプティマイザがマテリアライズドビューを検討することを検証してください。特定のクエリで使用されるマテリアライズドビューのリストは、Interactive SQL でクエリのグラフィカルなプランの **[高度な詳細]** ウィンドウで確認できます。

  また、Sybase Central でアプリケーションプロファイリングモードを使用して、オプティマイザで列挙されたアクセスプランを確認することで、クエリの列挙フェーズでマテリアライズドビューが検討されたかどうかを判断できます。トレーシングはオンにする必要があります。また、オプティマイザによって列挙されるアクセスプランを確認できるように、OPTIMIZATION_LOGGING トレーシングタイプを含めるように設定してください。

- **データ変更操作**　マテリアライズドビューは読み込み専用であるため、データ変更操作 (INSERT、LOAD、DELETE、UPDATE など) を適用できません。

- **キー、制約、トリガ、アーティクル**　マテリアライズドビューのインデックスは作成できますが、キー、制約、トリガ、またはアーティクルを作成することはできません。

参照

- 「高度：マテリアライズドビューに対するデータの古さの制御の設定」75 ページ
- 「ヒント：マテリアライズドビューを使ったクエリのパフォーマンス改善」225 ページ
- 「オプティマイザによるマテリアライズドビューの使用の有効化または無効化」69 ページ
- 「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「高度：クエリ実行プラン」328 ページ
- 「アプリケーションプロファイリング」148 ページ
- 「再表示タイプの手動または即時への設定」56 ページ

マテリアライズドビューとビューの依存性

マテリアライズドビューを有効または無効にすることで、データベースサーバがそのビューを使用できるかどうかを制御できます。無効になったマテリアライズドビューは、最適化時にオプティマイザによって検討されません。クエリが無効なマテリアライズドビューを明示的に参照している場合、そのクエリは失敗し、エラーが返されます。マテリアライズドビューを無効にすると、データベースサーバはそのビューのデータを削除しますが、定義はデータベース内に保持します。マテリアライズドビューをもう一度有効にすると、初期化されていない状態になるため、データを設定するためにはそのビューを再表示する必要があります。

マテリアライズドビューに依存する通常のビューは、マテリアライズドビューが無効になると、データベースサーバによって自動的に無効になります。その結果、マテリアライズドビューをもう一度有効にする場合は、すべての従属ビューをもう一度有効にする必要があります。このため、マテリアライズドビューを無効にする前に、ビューの依存性のリストを取得する場合があります。この操作は、sa_dependent_views システムプロシージャを使用して行います。このプロシージャは ISYSDEPENDENCY システムテーブルを検査して、従属ビューが存在する場合はそのリストを返します。

無効にされたオブジェクトに対する権限を付与できます。無効化されたオブジェクトに対する権限はデータベースに保存され、オブジェクトが有効になると使用できるようになります。

参照

- 「マテリアライズドビューの有効化または無効化」64 ページ
- 「依存性とスキーマ変更」42 ページ
- 「依存性情報の取得 (SQL の場合)」43 ページ
- 「sa_dependent_views システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

再表示タイプの手動または即時への設定

マテリアライズドビューには、手動と即時の 2 つの「再表示タイプ」があります。

- **手動ビュー**　手動のマテリアライズドビューまたは「手動ビュー」とは、再表示タイプが MANUAL REFRESH と定義されているマテリアライズドビューです。手動ビューは、たとえば REFRESH MATERIALIZED VIEW 文または sa_refresh_materialized_views システムプロシージャを使用して再表示を明示的に要求するまで再表示されないため、手動ビューのデータが古くなる場合があります。デフォルトでは、マテリアライズドビューを作成すると手動ビューになります。

  基本となるいずれかのテーブルが変更されると、マテリアライズドビューのデータが変更による影響を受けなくても、手動ビューは古くなったと見なされます。a_materialized_view_info システムプロシージャが返す DataStatus の値を調べて、手動ビューが古くなったかどうかを判断することができます。S が返されると、手動ビューが古いことを表します。

- **即時ビュー**　即時のマテリアライズドビューまたは「即時ビュー」とは、再表示タイプが IMMEDIATE REFRESH と定義されているマテリアライズドビューです。即時ビューのデータは、基本となるテーブルへの変更がビューのデータに影響を与える場合に、自動的に再表示されます。基本となるテーブルへの変更がビューのデータに影響しない場合、ビューは再表示されません。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

また、即時ビューが再表示されるときは、古いローのみを変更する必要があります。この点が手動ビューの再表示とは異なります。手動ビューの再表示では、再表示するためにすべてのデータが削除され、再作成されます。

手動ビューを即時ビューに、または即時ビューを手動ビューに変更できます。ただし、手動ビューから即時ビューへの変更処理では、少し多くの手順があります。

マテリアライズドビューの再表示タイプを変更すると、特に手動ビューから即時ビューに変更する場合は、ビューのステータスとプロパティに影響を与える場合があります。

参照

- 「古さと手動マテリアライズドビュー」57 ページ
- 「高度：マテリアライズドビューの再表示タイプの変更」71 ページ
- 「高度：マテリアライズドビューのステータスとプロパティ」72 ページ
- 「sa_materialized_view_info システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

古さと手動マテリアライズドビュー

基本となるベーステーブルで変更が発生すると、手動で再表示されるマテリアライズドビューは古くなります。マテリアライズドビューに対して設定された古さのしきい値をデータが超えた場合、オプティマイザはそのビューを、クエリを満たすための候補と見なしません。手動ビューを再表示するということは、データベースサーバがそのビューのクエリ定義を再実行し、ビューのデータをそのクエリの新しい結果セットで置き換えることを意味します。再表示を行うと、ビューのデータが基本となるデータと一致します。手動ビューのデータの古さについて許容可能な程度を検討し、再表示の方式を考案してください。その際には、再表示操作中はビューでクエリを処理できないため、再表示が完了するまでにかかる時間を考慮する必要があります。

また、イベントを使用してビューを再表示する方式を設定することもできます。たとえば、イベントを作成して一定の間隔で再表示することができます。

即時マテリアライズドビューは、トランケートされた後などの未初期化状態 (データがない) の場合を除き、再表示する必要はありません。

materialized_view_optimization データベースオプションを使用して、古さのしきい値を設定できます。オプティマイザは、クエリの処理時にこのしきい値を超えたマテリアライズドビューを使用しません。

マテリアライズドビューが含まれるデータベースのアップグレード
データベースサーバをアップグレードした後、またはアップグレード後のデータベースサーバで使用できるようにデータベースを再構築またはアップグレードした後は、マテリアライズドビューを再表示することをおすすめします。

参照

- 「高度：マテリアライズドビューに対するデータの古さの制御の設定」75 ページ
- 「マテリアライズドビューの手動再表示」63 ページ
- 「materialized_view_optimization オプション」『SQL Anywhere サーバ データベース管理』

マテリアライズドビューの制限

マテリアライズドビューを作成、初期化、再表示、ビューマッチングする際の制限

- マテリアライズドビューを作成するときは、マテリアライズドビューの定義でカラム名を明示的に定義する必要があります。カラム定義の一部として SELECT * 構成要素を含めることはできません。

- TIMESTAMP WITH TIME ZONE として定義されているカラムを、マテリアライズドビューに含めないでください。time_zone_adjustment オプションの値は、ロケーションと日付に基づいて、接続ごとに異なる値となる場合があるため、不正な結果や予期しない動作が発生することがあります。

- マテリアライズドビューを作成するときは、マテリアライズドビューの定義に次の項目を含めることはできません。

  - 他のビュー (マテリアライズドビューまたは通常のビュー) に対する参照
  - リモートテーブルまたはテンポラリテーブルに対する参照
  - CURRENT USER などの変数。すべての式は確定的でなければなりません
  - ストアドプロシージャ、ユーザ定義関数、または外部関数の呼び出し
  - Transact-SQL 外部ジョイン
  - FOR XML 句

  グループ化されたプロジェクト選択ジョインのクエリブロックには select リストに COUNT(*) を含める必要があり、SUM と COUNT 集合関数のみを使用できます。

- 次のデータベースオプションでは、マテリアライズドビューを作成するときに特定の設定が必要です。そのように設定しない場合は、エラーが返されます。これらのデータベースオプションの値は、オプティマイザが使用するビューにも必要です。

  - ansinull=On
  - conversion_error=On
  - divide_by_zero_error=On
  - sort_collation=Internal
  - string_rtruncation=On

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

- 次のデータベースオプションは、マテリアライズドビューを作成するときにマテリアライズドビューごとに格納されます。マテリアライズドビューが最適化で使用されるには、接続の現在のオプションの値がビューに対して格納された値と一致する必要があります。

  - date_format
  - date_order
  - default_timestamp_increment
  - first_day_of_week
  - nearest_century
  - precision
  - scale
  - time_format
  - timestamp_format
  - timestamp_with_time_zone_format
  - default_timestamp_increment
  - uuid_has_hyphens

- ビューが再表示されるときは、上記の項目に示されたすべてのオプションの接続設定が無視されます。代わりに、データベースオプションの設定 (格納されたビューの設定と一致する必要がある) が使用されます。

マテリアライズドビュー定義内の ORDER BY 句は影響しない

マテリアライズドビューは、ローが特定の順序で格納されない点がベーステーブルに似ています。データベースサーバは、データの計算時に最も効率の良い方法でローを並べます。そのため、マテリアライズドビュー定義で ORDER BY 句を指定しても、ビューが実体化されるときのローの順序に影響はありません。また、ビューマッチングの実行時に、ビューの定義内の ORDER BY 句はオプティマイザによって無視されます。

マテリアライズドビューを手動から即時に変更する際の制限

手動ビューを即時ビューに変更するときには、次の制限が検査されます。ビューがいずれかの制限に違反している場合は、エラーが返されます。

> **注意**
> sa_materialized_view_can_be_immediate システムプロシージャを使用して、手動ビューを即時ビューにする条件がそろっているかどうかを調べることができます。

- ビューは未初期化状態である必要がある。

- ビューに外部ジョインがない場合は、ビューの NULL 入力不可のカラムにユニークインデックスが作成されている。ビューに外部ジョインがある場合は、ビューの NULL 入力不可のカラムにユニークインデックスを作成するか、NULL 入力可のカラムにユニークインデックスを WITH NULLS NOT DISTINCT と宣言して作成してください。

- ビューの定義がグループ化されたクエリである場合、ユニークインデックスのカラムは、集合関数ではない SELECT リスト項目に対応する必要がある。

- ビューの定義に次のものを含むことはできない。
  - GROUPING SETS 句
  - CUBE 句
  - ROLLUP 句
  - DISTINCT 句
  - ロー制限句
  - 非決定的な式
  - セルフジョインと再帰ジョイン
  - LATERAL、CROSS APPLY、または APPLY 句
- ビューの定義は、単一のプロジェクト選択ジョインまたはグループ化されたプロジェクト選択ジョインのブロックである必要があり、グループ化されたプロジェクト選択ジョインのクエリブロックには HAVING 句を含めることはできない。
- グループ化されたプロジェクト選択ジョインのクエリブロックには、SELECT リストに COUNT(*) を含める必要があり、SUM と COUNT 集合関数のみを使用できます。
- SELECT リストの集合関数は、複雑な式の中では参照できない。たとえば、SUM(expression) + 1 は SELECT リストで使用できません。
- SELECT リストに SUM(expression) 集合関数が含まれており、*expression* が NULL 入力可能な式である場合は、SELECT リストに COUNT(expression) 集合関数を含めます。
- ビュー定義に外部ジョイン (LEFT OUTER JOIN、RIGHT OUTER JOIN、FULL OUTER JOIN) が含まれる場合は、ビュー定義が以下の追加条件を満たすようにしてください。

  1. テーブル T が OUTER JOIN の ON 条件で保護側として参照されている場合は、T にはプライマリキーがあり、かつ、そのプライマリキーのカラムがビューの SELECT リストに含まれている必要があります。たとえば、SELECT T1.pk, R1.X FROM T1, T2 LEFT OUTER JOIN (R1 KEY JOIN R2) ON T1.Y = R.Y として定義された即時マテリアライズドビュー V において、ON 句で保護テーブル T1 が参照され、かつ、このテーブルのプライマリキーのカラム T1.pk が即時マテリアライズドビュー V の SELECT リストに含まれています。

  2. 外部ジョインの各 NULL 入力側については、NULL 入力不可のカラムのいずれかが即時マテリアライズドビューの SELECT リストに含まれているベーステーブルが、少なくとも 1 つ存在する必要があります。たとえば、SELECT T1.pk, R1.X FROM T1, T2 LEFT OUTER JOIN (R1 KEY JOIN R2) ON T1.Y = R1.Y と定義された即時マテリアライズドビュー V の場合、左外部ジョインの NULL 入力側はテーブル式 (R1 KEY JOIN R2) です。カラム R1.X はテーブル R1 の NULL 入力不可のカラムであり、V の SELECT リストに含まれています。

  3. ビューが GROUP 化されたビューであり、以前の条件が保持されていない場合は、外部ジョインの各 NULL 入力側に、ベーステーブル T (NULL 入力不可のカラムの 1 つである T.C が、即時マテリアライズドビューの SELECT リストの集合関数 COUNT(T.C) で使用されている) を少なくとも 1 つ指定してください。たとえば、SELECT T1.pk, COUNT(R1.X) FROM T1, T2 LEFT OUTER JOIN (R1 KEY JOIN R2) ON T1.Y = R1.Y GROUP BY T1.pk と定義された即時マテリアライズドビュー V の場合、左外部ジョインの NULL 入力側はテーブル式 (R1 KEY JOIN R2) です。集合関数 COUNT(R1.X) は V の SELECT リストに含まれており、R1.X はテーブル R1 の NULL 入力不可のカラムです。

4. 外部ジョインを持つビューの述部が、以下の条件を満たすようにしてください。

   ○LEFT OUTER JOIN、RIGHT OUTER JOIN、FULL OUTER JOIN の ON 句述部によって、保護テーブル式と NULL 入力テーブル式の両方が参照されていること。たとえば、T LEFT OUTER JOIN R ON R.X = 1 では、述部 R.X=1 によって NULL 入力側 R のみが参照されているため、この条件が満たされません。

   ○ネストした外部ジョインによって生成された NULL 入力されるローが、すべての述部で拒否されること。つまり、述部で、ネストした外部ジョインによって NULL 入力されるテーブル式が参照されている場合は、その外部ジョインによって生成された、NULL を持つすべてのローが拒否されるようにします。

   たとえば、ビュー V1 SELECT T1.pk, R1.X FROM T1, T2 LEFT OUTER JOIN (R1 KEY JOIN R2) ON (T1.Y = R1.Y) WHERE R1.Z = 10 の述部の R1.Z=10 では、T2 LEFT OUTER JOIN (R1 KEY JOIN R2) によって NULL 入力される可能性のあるテーブル R1 が参照されるため、NULL 入力されるすべてのローが拒否されるようにしてください。カラム R1.Z が NULL の場合、述部が UNKNOWN と評価されるため、このことが必要となります。

   ただし、ビュー V2 SELECT T1.pk, R1.X FROM T1, T2 LEFT OUTER JOIN (R1 KEY JOIN R2) ON (T1.Y = R1.Y) WHERE R1.Z IS NULL はこのプロパティを持っていません。述部 R1.Z IS NULL は NULL 入力側 R1 を参照していますが、テーブル R1 が NULL 入力される (つまり、R1.Z カラムが NULL) 場合は、TRUE と評価されます。NULL 入力されるローを拒否する方法は、NULL を許容しないプロパティと比較して制限が少なくなります。たとえば、述部 R.X IS NOT DISTINCT FROM T.X and rowid(T) IS NOT NULL は、T.X が NULL の場合に TRUE と評価されるため、テーブル T の NULL が許容されます。ただし、述部はベーステーブル T で NULL 入力されるすべてのローを拒否します。

参照

- 「sa_materialized_view_can_be_immediate システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「高度：マテリアライズドビューのステータスとプロパティ」72 ページ
- 「インデックスの作成」31 ページ
- 「マテリアライズドビューの作成」61 ページ
- TimeZoneAdjustment 接続プロパティ『SQL Anywhere サーバ データベース管理』
- 「CREATE MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「外部ジョイン」501 ページ

マテリアライズドビューの作成

マテリアライズドビューを作成して、頻繁に実行されるために大量のデータで繰り返し集約やジョイン操作が発生するクエリから、データを保存します。マテリアライズドビューを使用すると、ディスクに保管されるビュー形式のコストの高い操作を事前に計算することで、パフォーマンスを向上させることができます。

前提条件

自分が所有するマテリアライズドビューを作成する場合は、基礎となるすべてのテーブルに対する SELECT 権限とともに、CREATE MATERIALIZED VIEW システム権限を持っていることが必要です。

他者が所有するマテリアライズドビューを作成する場合は、基礎となるすべてのテーブルに対する SELECT 権限とともに、CREATE ANY MATERIALIZED VIEW または CREATE ANY OBJECT のシステム権限を持っていることが必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で **[ビュー]** を右クリックし、**[新規]** » **[マテリアライズドビュー]** をクリックします。
3. **マテリアライズドビュー作成ウィザード**の指示に従います。

結果

初期化されていないマテリアライズドビューが、データベースに作成されます。まだデータはありません。

次の手順

マテリアライズドビューを初期化して、データを取り込んでから使用する必要があります。

参照

- 「マテリアライズドビュー」53 ページ
- 「SQL Anywhere サンプルデータベース」『SQL Anywhere 16 紹介』
- 「マテリアライズドビューの制限」58 ページ
- 「マテリアライズドビューの削除」67 ページ
- 「マテリアライズドビューの初期化」62 ページ
- 「CREATE MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』

マテリアライズドビューの初期化

マテリアライズドビューを初期化してデータを取り込み、データベースサーバから使用できるようにします。マテリアライズドビューを初期化するには、マテリアライズドビューの再表示と同じ手順に従います。

前提条件

そのマテリアライズドビューの所有者であるか、そのマテリアライズドビューに対する INSERT 権限を持っているか、または INSERT ANY TABLE 権限を持っていることが必要です。

マテリアライズドビューを作成、初期化、または再表示する前に、マテリアライズドビューの制限をすべて満たしていることを確認してください。

内容と備考

データベースの sa_refresh_materialized_views システムプロシージャを使用することで、すべての初期化されていないマテリアライズドビューを 1 回で初期化できます。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. マテリアライズドビューを右クリックして、**[データの再表示]** をクリックします。
4. 独立性レベルを選択して、**[OK]**をクリックします。

結果

マテリアライズドビューにデータが取り込まれ、データベースサーバから使用できるようになります。これで、マテリアライズドビューに問い合わせできます。

次の手順

マテリアライズドビューに問い合わせして、希望のデータを戻すか確認します。

初期化(再表示)に失敗した場合、マテリアライズドビューは未初期化状態に戻ります。初期化に失敗した場合、マテリアライズドビューの定義を確認して、指定された基本テーブルとカラムが有効で、データベースで使用できるオブジェクトであることを確認します。

参照

- 「マテリアライズドビュー」53 ページ
- 「マテリアライズドビューの削除」67 ページ
- 「マテリアライズドビューの制限」58 ページ
- 「マテリアライズドビューの有効化または無効化」64 ページ
- 「CREATE MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_refresh_materialized_views システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

マテリアライズドビューの手動再表示

自動的に再表示するよう設定されていないマテリアライズドビューは、手動で再表示してデータを更新する必要があります。

前提条件

そのマテリアライズドビューの所有者であるか、そのマテリアライズドビューに対する INSERT 権限を持っているか、または INSERT ANY TABLE システム権限を持っていることが必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. マテリアライズドビューを右クリックして、**[データの再表示]** をクリックします。
4. 独立性レベルを選択して、**[OK]**をクリックします。

結果

マテリアライズドビューのデータは再表示され、基本となるオブジェクトの最新データを表示します。

次の手順

マテリアライズドビューに問い合わせして、希望のデータを戻すか確認します。

再表示に失敗した場合は、マテリアライズドビューは未初期化状態に戻ります。これが発生した場合、マテリアライズドビューの定義を確認して、指定された基本テーブルとカラムが有効で、データベースで使用できるオブジェクトであることを確認します。

参照

- 「マテリアライズドビュー」53 ページ
- 「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「マテリアライズドビューの削除」67 ページ
- 「高度：マテリアライズドビューの再表示タイプの変更」71 ページ
- 「スケジュールとイベントの使用によるタスクの自動化」『SQL Anywhere サーバ データベース管理』
- 「materialized_view_optimization オプション」『SQL Anywhere サーバ データベース管理』
- 「sa_refresh_materialized_views システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

マテリアライズドビューの有効化または無効化

マテリアライズドビューを有効または無効にすることで、問い合わせに使用できるかどうか管理できます。

前提条件

そのマテリアライズドビューの所有者であるか、または次のいずれかのシステム権限を持っていることが必要です。

- ALTER ANY MATERIALIZED VIEW
- ALTER ANY OBJECT

マテリアライズドビューを「有効にする」には、基本となるテーブルに対する SELECT 権限または SELECT ANY TABLE システム権限も必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。

オプション	アクション
マテリアライズドビューの有効化	a. ビューを右クリックして、**[再コンパイルして有効にする]** をクリックします。 b. (オプション) ビューを右クリックし、**[データの再表示]** をクリックしてビューにデータを取り込みます。有効後にビューに対して最初に実行されるクエリによってビューにデータが取り込まれるので、この手順はオプションになります。
マテリアライズドビューの無効化	ビューを右クリックして、**[無効にする]** をクリックします。

結果

マテリアライズドビューを有効にすると、データベースサーバから使用できるので、問い合わせできます。

マテリアライズドビューを無効にすると、データとインデックスは削除されます。ビューが即時ビューである場合、手動ビューに変わります。無効であるマテリアライズドビューへの問い合わせは失敗し、エラーが返されます。

次の手順

ビューを再び有効にしたらビューのインデックスを再構築し、ビューを無効にしたときに即時ビューであった場合は、即時ビューに戻す必要があります。

参照

- 「マテリアライズドビュー」53 ページ
- 「高度：マテリアライズドビューの再表示タイプの変更」71 ページ
- 「sa_dependent_views システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「ビューの依存性」41 ページ
- 「SYSDEPENDENCY システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』

マテリアライズドビュー定義を隠す

マテリアライズドビュー定義をユーザから隠して、データベースに保存されているビュー定義を難読化することができます。この設定は、元に戻せません。

前提条件

そのマテリアライズドビューの所有者であるか、または次のいずれかのシステム権限を持っていることが必要です。

- ALTER ANY MATERIALIZED VIEW
- ALTER ANY OBJECT

内容と備考

マテリアライズドビューが隠されると、デバッガを使用したデバッグでも、プロシージャプロファイリングによっても、ビュー定義は表示されません。このビューはアンロードして、他のデータベースに再ロードできます。

マテリアライズドビューを非表示にすると元に戻せず、SQL を使用するだけで非表示にできます。

♦ タスク

1. データベースに接続します。
2. ALTER MATERIALIZED VIEW...SET HIDDEN 文を実行します。

結果

オートコミットが実行されます。

カタログをブラウズするとき、ビューは表示されません。ただし、ビューは直接参照でき、クエリ処理中に使用できることは変わりません。

例

次の文は、マテリアライズドビュー EmployeeConfid3 を作成し、再表示して、ビュー定義を難読化します。

```
CREATE MATERIALIZED VIEW EmployeeConfid3 AS
  SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary, ManagerID,
    Departments.DepartmentName, Departments.DepartmentHeadID
  FROM Employees, Departments
  WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid3;
ALTER MATERIALIZED VIEW EmployeeConfid3 SET HIDDEN;
```

警告
次の例を実行するとき、作成したマテリアライズドビューを削除する必要があります。このようにしない場合は、他の例を試すとき、基本となるテーブル Employees および Departments に対するスキーマ変更を実行できなくなります。

参照

- 「マテリアライズドビュー」53 ページ
- 「ALTER MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「マテリアライズドビューの削除」67 ページ
- 「DROP STATEMENT 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』

マテリアライズドビューの削除

データベースからマテリアライズド/ビューを削除できます。マテリアライズドビューが必要なくなったとき、または基本となる参照オブジェクトにスキーマ変更を実施して、マテリアライズドビュー定義が有効でなくなったとき、この作業を実行します。

前提条件

所有者であるか、または DROP ANY MATERIALIZED VIEW と DROP ANY OBJECT のいずれかのシステム権限を持っている必要があります。

マテリアライズドビューを削除する前に、すべての従属ビューを削除または無効にする必要があります。マテリアライズドビューに従属ビューが存在するかどうかを判断するには、sa_dependent_views システムプロシージャを使用します。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. マテリアライズドビューを右クリックして、**[削除]** をクリックします。
4. **[はい]** をクリックします。

結果

マテリアライズドビューはデータベースから削除されます。

次の手順

通常のビューがマテリアライズドビューに依存している場合、これらを有効にすることはできません。ビュー定義を変更するか、ビューを削除する必要があります。

参照

- 「マテリアライズドビュー」53 ページ
- 「ビューの依存性」41 ページ
- 「DROP MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_dependent_views システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

マテリアライズドビューの暗号化または復号化

セキュリティを高めるために、マテリアライズドビューを暗号化できます。たとえば、基本となるテーブルで暗号化されていたデータがマテリアライズドビューに含まれる場合、そのマテリアライズドビューも暗号化する状況も考えられます。

前提条件

所有者であるか、または CREATE ANY MATERIALIZED VIEW と DROP ANY MATERIALIZED VIEW の両方のシステム権限を持っているか、または CREATE ANY OBJECT と DROP ANY OBJECT の両方のシステム権限を持っていることが必要です。

マテリアライズドビューを暗号化するには、データベースでテーブルの暗号化をあらかじめ有効にしておく必要があります。

内容と備考

データベースの作成時に指定した暗号化アルゴリズムとキーを使用して、マテリアライズドビューを暗号化します。テーブル暗号化が有効であるかどうかなど、暗号化設定がデータベースで有効であることを確認するには、次のように DB_PROPERTY 関数を使用して Encryption データベースプロパティの値を取得します。

```
SELECT DB_PROPERTY( 'Encryption' );
```

テーブルの暗号化と同様に、マテリアライズドビューを暗号化するとパフォーマンスに影響がある可能性があります。データベースサーバがビューから取得したデータを復号化する必要があるためです。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. マテリアライズドビューを右クリックして、**[プロパティ]** をクリックします。
4. **[その他]** タブをクリックします。

5. 必要に応じて、**[マテリアライズド ビューのデータは暗号化済み]** チェックボックスをオンにする、またはクリアします。

6. **[OK]** をクリックします。

結果

マテリアライズドビューのデータが暗号化されます。

参照

- 「テーブルの暗号化」『SQL Anywhere サーバ データベース管理』
- 「ALTER MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DB_PROPERTY 関数 [システム]」『SQL Anywhere サーバ SQL リファレンス』

オプティマイザによるマテリアライズドビューの使用の有効化または無効化

クエリがマテリアライズドビューを参照しない場合でも、クエリを満たすとパフォーマンスが向上する場合、オプティマイザは、クエリを満たすためビューを使用するよう決定できます。クエリを満たすため、オプティマイザによるマテリアライズドビューの使用を有効または無効にできます。

前提条件

所有者であるか、または ALTER ANY MATERIALIZED VIEW と ALTER ANY OBJECT のいずれかのシステム権限を持っている必要があります。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。

2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。

3. マテリアライズドビューを右クリックして、**[プロパティ]** をクリックします。

4. 必要に応じて、**[一般]** タブをクリックして、**[最適化に使用]** をオンにする、またはクリアします。

5. **[OK]** をクリックします。

結果

オプティマイザがマテリアライズドビューを使用できる場合、クエリでビューが明示的に参照されないとしても、オプティマイザは、クエリを満たすための最良プランを計算するときにビューを考慮します。オプティマイザがマテリアライズドビューを使用できない場合、オプティマイザはビューを考慮しません。

次の手順

クエリ実行プランを見て、オプティマイザがビューを使用するかどうか判断するため、ビューの基本となるオブジェクトに問い合わせることがあります。ただし、ビューの可用性によって、オプティマイザがビューを使用しない可能性があります。オプティマイザの選択は、パフォーマンスに基づいています。

参照

- 「ALTER MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「マテリアライズドビューを使用したパフォーマンスの向上」54 ページ
- 「高度：クエリ実行プラン」328 ページ

高度：カタログ内のマテリアライズドビュー情報の表示

すべてのマテリアライズドビューとそのステータスのリストを表示し、各マテリアライズドビューを作成したときに実行されていたデータベースオプションを確認することもできます。

前提条件

マテリアライズドビューを非表示にすることはできません。

内容と備考

依存性の情報は、SYSDEPENDENCY システムビューで探すこともできます。

♦ タスク

1. データベースに接続します。
2. すべてのマテリアライズドビューとそのステータスのリストを表示するには、次の文を実行します。

```
SELECT * FROM sa_materialized_view_info();
```

3. マテリアライズドビューが作成されたときに、各ビューに対して実行されていたデータベースオプションを確認するには、次の文を実行します。

```
SELECT b.object_id, b.table_name, a.option_id, c.option_name, a.option_value
FROM SYSMVOPTION a, SYSTAB b, SYSMVOPTIONNAME c
WHERE a.view_object_id=b.object_id
AND b.table_type=2;
```

4. 所定のマテリアライズドビューに依存している通常のビューのリストを要求するには、次の文を実行します。

```
CALL sa_dependent_views( 'materialized-view-name' );
```

結果

要求されたマテリアライズドビュー情報が返されます。

参照

- 「sa_materialized_view_info システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「高度：マテリアライズドビューのステータスとプロパティ」 72 ページ
- 「sa_dependent_views システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSDEPENDENCY システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSMVOPTION システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSMVOPTIONNAME システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSTAB システムビュー」『SQL Anywhere サーバ SQL リファレンス』

高度：マテリアライズドビューの再表示タイプの変更

マテリアライズドビューを作成すると、その再表示タイプは手動になりますが、即時に変更することはできません。再び、即時ビューを手動ビューに戻すこともできます。

前提条件

所有者であるか、または CREATE ANY MATERIALIZED VIEW と DROP ANY MATERIALIZED VIEW の両方のシステム権限を持っているか、または CREATE ANY OBJECT と DROP ANY OBJECT の両方のシステム権限を持っていることが必要です。必要な権限がない場合、マテリアライズドビューを即時ビューに変更する (ALTER MATERIALIZED VIEW...IMMEDIATE REFRESH) には、ビューとそのすべての参照先テーブルを所有している必要があります。

手動から即時に変更するには、ビューが未初期化状態である (データを含まない) 必要があります。ビューが作成された直後でまだ再表示されていない場合は、未初期化状態になっています。マテリアライズドビューにデータがある場合、即時に変更する前に TRUNCATE 文を実行して未初期化状態に戻す必要があります。また、マテリアライズドビューにはユニークインデックスも必要で、即時ビューに必要な制限に従う必要があります。

再表示タイプの変更以外の手順を追加しないで、即時ビューをいつでも手動に変更できます。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. マテリアライズドビューを右クリックして、**[プロパティ]** をクリックします。
4. **[再表示タイプ]** フィールドで、次のいずれかのオプションを選択します。

オプション	アクション
手動ビューから即時ビューへの変更	即時
即時ビューから手動ビューへの変更	手動

5. **[OK]** をクリックします。

結果

マテリアライズドビューの再表示タイプが変更されます。基本となるオブジェクトのデータが変更されると、即時ビューが更新されます。手動ビューを再表示すると、手動ビューが更新されます。

次の手順

手動から即時へ変更したら、ビューを初期化(再表示)してデータを取り込む必要があります。

参照

- 「再表示タイプの手動または即時への設定」56 ページ
- 「マテリアライズドビューの初期化」62 ページ
- 「sa_materialized_view_can_be_immediate システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「マテリアライズドビューを手動から即時に変更する際の制限」59 ページ
- 「ALTER MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「TRUNCATE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「インデックスの作成」31 ページ

高度：マテリアライズドビューのステータスとプロパティ

マテリアライズドビューは、ステータスとプロパティの組み合わせによって特徴付けられます。マテリアライズドビューのステータスは、データベースサーバが使用するビューの利用可能性を反映しています。マテリアライズドビューのプロパティは、ビュー内のデータのステータスを反映しています。

既存のマテリアライズドビューのステータスとプロパティを判断するのに一番良い方法は、sa_materialized_view_info システムプロシージャを使用することです。

また、Sybase Central の **[ビュー]** フォルダを選択して個々のビュー詳細を調べるか、SYSTAB および SYSVIEW システムビューを問い合わせることにより、マテリアライズドビューの情報を表示できます。

参照

- 「sa_materialized_view_info システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSTAB システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSVIEW システムビュー」『SQL Anywhere サーバ SQL リファレンス』

マテリアライズドビューのステータス

マテリアライズドビューには、次の 2 種類のステータスがあります。

- **有効**　マテリアライズドビューは、コンパイルが正常に実行され、データベースサーバから使用できます。有効にされたマテリアライズドビューにはデータがない場合もあります。たとえば、有効にされたマテリアライズドビューからデータをトランケートすると、有効に変わりますが、初期化されていません。マテリアライズドビューを初期化することはできます

が、マテリアライズドビューの定義を満たす基本となるテーブルにデータがない場合は、空になります。これは、初期化されていないためにデータがないマテリアライズドビューと同じではありません。

- **無効**　マテリアライズビューは、ALTER MATERIALIZED VIEW...DISABLE 文の使用などによって明示的に無効化されました。マテリアライズドビューを無効にすると、そのビューのデータとインデックスは削除されます。また、即時ビューを無効にすると、手動ビューに変わります。

ビューが有効であるか、無効であるかを判断するには、ビューのステータスプロパティを返す sa_materialized_view_info システムプロシージャを使用します。

参照

- 「ALTER MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_materialized_view_info システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「マテリアライズドビューの有効化または無効化」64 ページ
- 「マテリアライズドビューのプロパティ」73 ページ

マテリアライズドビューのプロパティ

マテリアライズドビューのプロパティは、ビューを使用するかどうかを評価するときにオプティマイザによって使用されます。次のリストでは、sa_materialized_view_info システムプロシージャが返すマテリアライズドビューのプロパティについて説明します。

- **Status**　ビューが有効であるか、無効であるかを示します。
- **DataStatus**　ビュー内のデータのステータスを反映します。たとえば、ビューが初期化されているかどうか、ビューが古いかどうかなどを示します。マテリアライズドビューが最後に再表示されてから基本となるテーブルのデータが変更されると、手動ビューは古くなります。即時ビューが古くなることはありません。
- **ViewLastRefreshed**　ビューが最後に再表示された時刻を示します。
- **DateLastModified**　ビューが古い場合に、基本となるテーブルのデータが最後に修正された時刻を示します。
- **AvailForOptimization**　オプティマイザがビューを使用できるかどうかを反映します。
- **RefreshType**　ビューが手動ビューであるか、即時ビューであるかを示します。

各プロパティで使用できる値については、sa_materialized_view_info システムプロシージャを使用してください。

手動ビューを即時ビューに変換できるかどうかを示すプロパティはありませんが、sa_materialized_view_can_be_immediate システムプロシージャを使用してこれを判断できます。

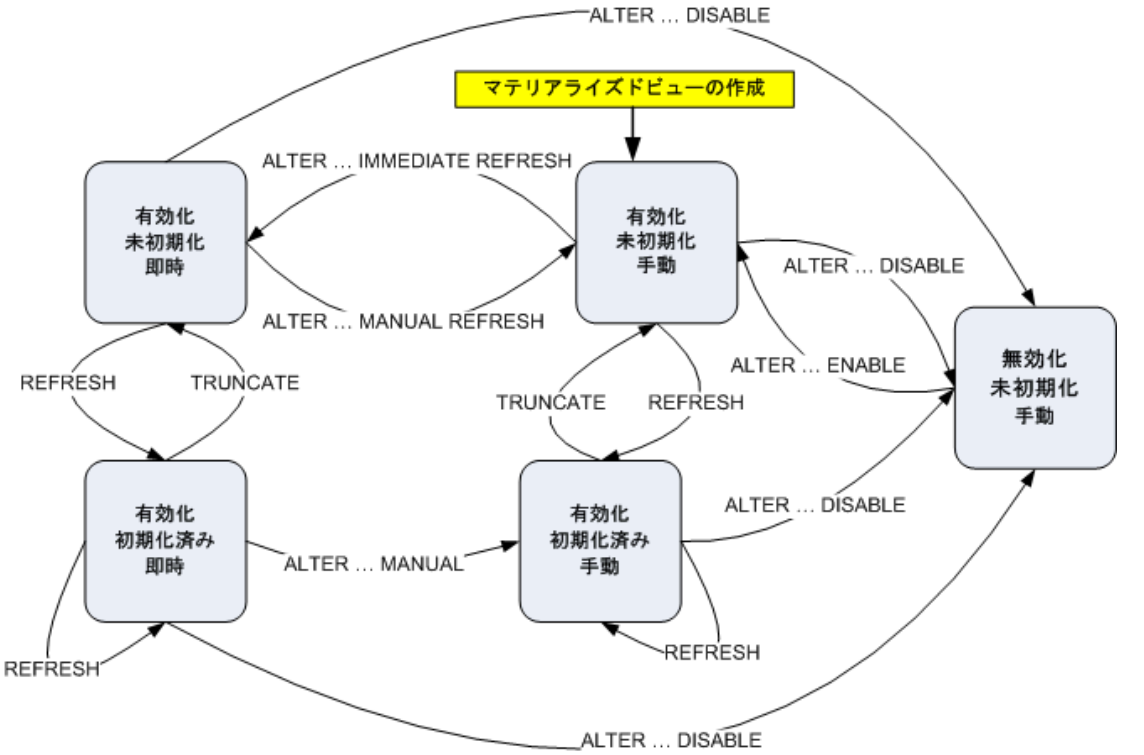
参照

- 「sa_materialized_view_info システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_materialized_view_can_be_immediate システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「マテリアライズドビューのステータス」72 ページ

マテリアライズドビューの変更、再表示、トランケート実行時のステータスとプロパティの変更

マテリアライズドビューで変更、再表示、トランケートなどの操作を実行すると、ビューのステータスとプロパティに影響を与えます。次の図に、これらのタスクがマテリアライズドビューのステータスと一部のプロパティに与える影響を示します。

この図では、灰色の四角がマテリアライズドビューであり、即時ビューは IMMEDIATE という用語、手動ビューは MANUAL という用語で区別されています。灰色のボックスの間を接続する ALTER という用語は、ALTER MATERIALIZED VIEW の省略です。マテリアライズドビューのステータスを変更するために SQL 文が示されていますが、Sybase Central を使用してこれらの操作を実行することもできます。



- マテリアライズドビューを作成すると、そのビューは有効な手動ビューになり、未初期化状態である (データは含まれない)。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

- 未初期化状態のビューを再表示すると、初期化された状態になる (データが入力される)。
- 手動ビューから即時ビューに変更するにはいくつかの手順が必要である。また、即時ビューには追加の制限がある。
- マテリアライズドビューを無効にすると、次のことが行われる。
  - データが削除される
  - ビューが未初期化状態に戻る
  - インデックスが削除される
  - 即時ビューが手動に戻る

参照

- 「ALTER MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「再表示タイプの手動または即時への設定」56 ページ
- 「高度：マテリアライズドビューの再表示タイプの変更」71 ページ
- 「マテリアライズドビューを手動から即時に変更する際の制限」59 ページ
- 「マテリアライズドビューのプロパティ」73 ページ
- 「マテリアライズドビューのステータス」72 ページ

高度：マテリアライズドビューに対するデータの古さの制御の設定

マテリアライズドビュー内のデータは、そのビューが参照するテーブルのデータが変更されることによって古くなります。マテリアライズドビューがオプティマイザによって検討されない場合、古さに原因がある可能性があります。マテリアライズドビューに対する古さのしきい値は、materialized_view_optimization データベースオプションを使用して調整できます。

ビューを再表示するイベントまたはトリガに対して指定した間隔を調整することもできます。

クエリがマテリアライズドビューを明示的に参照している場合、そのビューのデータの古さに関係なく、ビューがクエリの処理に使用されます。さらに、materialized_view_optimization データベースオプションの設定を上書きし、マテリアライズドビューを強制的に使用する場合に、SELECT、UPDATE、INSERT などの文の OPTION 句を使用できます。

スナップショットアイソレーションが使用されている場合、トランザクションのスナップショットの開始後にマテリアライズドビューが再表示されると、オプティマイザはそのマテリアライズドビューを使用しません。

参照

- 「オプティマイザによって検討されたマテリアライズドビューの判断」228 ページ
- 「マテリアライズドビューとビューマッチング」226 ページ
- 「materialized_view_optimization オプション」『SQL Anywhere サーバ データベース管理』

ストアドプロシージャ、トリガ、バッチ、ユーザ定義関数

プロシージャとトリガは、すべてのアプリケーションで使えるように SQL 文をデータベースに格納します。プロシージャとトリガは、SQL 文の繰り返し (LOOP 文) と条件付き実行 (IF 文と CASE 文) を含むことができます。バッチは、データベースサーバにグループとして送られる SQL コマンドのセットです。制御文など、プロシージャとトリガで使用できる機能の多くは、バッチでも使用できます。

ソース制御
ソースコードに対する変更と、ソース (ストアドプロシージャを含む) から作成されデータベースに展開されるオブジェクトに対する変更を追跡する場合は、ソース制御を使用します。

プロシージャは CALL 文で呼び出され、パラメータを使って値を受け取り、呼び出しを行った環境に結果の値を返します。SELECT 文の FROM 句にプロシージャ名を含めると、プロシージャの結果セットを操作できます。

プロシージャは、呼び出し元に結果セットを返すか、他のプロシージャを呼び出すか、トリガを起動します。たとえば、ユーザ定義関数はストアドプロシージャの一種であり、呼び出しを行った環境に 1 つの値を返します。ユーザ定義関数は、渡されたパラメータを変更しないで、クエリや他の SQL 文に使用可能な関数のスコープを拡張します。

トリガは、特定のデータベーステーブルに関連付けられています。トリガは、関連するテーブルのローが挿入、更新、削除されるたびに自動的に起動します。トリガでプロシージャを呼び出したり、他のトリガを起動したりはできますが、トリガにパラメータを指定したり、CALL 文で呼び出したりはできません。

SQL Anywhere のデバッガ

ストアドプロシージャとトリガは、SQL Anywhere のデバッガを使用してデバッグできます。

ストアドプロシージャをプロファイリングして、Sybase Central でパフォーマンスを分析できます。

参照

- 「システムプロシージャを使用したプロシージャプロファイリング」188 ページ
- 「SQL Anywhere のデバッガ」947 ページ

プロシージャ、トリガ、およびユーザ定義関数の利点

プロシージャとトリガによりデータベースのセキュリティ、効率、標準化を高めることができます。

プロシージャとトリガの定義はデータベース内にあり、データベースアプリケーションから分離されています。これには、いくつかの利点があります。

標準化

プロシージャとトリガを使用すると、複数のアプリケーションプログラムで実行するアクションを標準化できます。アクションをコーディングし、将来利用するためにデータベースに格納します。アプリケーションはプロシージャを呼び出すか、トリガを起動するだけで、何度でもそのアクションを実行できます。すべての変更が 1 か所で行われるため、アクションの実装が変更された場合、アクションを使用するすべてのアプリケーションが自動的に新機能を取得します。

効率化

ネットワークデータベースサーバ環境で使用されるプロシージャとトリガは、ネットワーク通信を使用しないでデータベースのデータにアクセスできます。つまり、クライアント上のアプリケーションに実装する場合と比較して、ネットワークのパフォーマンスを低下させることなく高速に実行されます。

プロシージャとトリガを作成すると、自動的に構文チェックを行った後に、システムテーブルに格納されます。アプリケーションが初めてプロシージャを呼び出すか、トリガを起動するときには、システムテーブルからコンパイルされて仮想メモリにロードされ、実行されます。最初に実行された後もプロシージャまたはトリガのコピーがメモリに保持されるため、同じプロシージャまたはトリガの実行を繰り返す場合、すぐに実行できます。また、複数のアプリケーションが同時にプロシージャまたはトリガを使用することも、1 つのアプリケーションが再帰的に使用することもできます。

セキュリティ

「セキュリティ：プロシージャとトリガ」『SQL Anywhere サーバ データベース管理』を参照してください。

参照

- 「セキュリティ：ビューとプロシージャを使用して、ユーザがアクセスできるデータを制限」『SQL Anywhere サーバ データベース管理』

プロシージャ

プロシージャとファンクションを設定し所有者または呼び出し側の権限で実行する

プロシージャまたはファンクションを指定する場合、プロシージャまたはファンクションをその「所有者」の権限で実行するか、またはそれを呼び出したユーザまたはプロシージャ (「呼び出し側」) の権限で実行するかを指定できます。呼び出し側の特定は常に明白であるとはかぎりません。ユーザがプロシージャを呼び出すこともできますが、プロシージャが別のプロシージャを呼び出すこともできます。このような場合、「ログインユーザ」 (トップレベルのプロシージャを最初に呼び出したユーザ) と、最初のプロシージャによって呼び出されたプロシージャの所有者の場合もある「有効なユーザ」とは区別されます。プロシージャが呼び出し側の権限で実行される場合、有効なユーザの権限が適用されます。

プロシージャまたはファンクションを作成する場合、CREATE PROCEDURE 文または CREATE FUNCTION 文の SQL SECURITY 句により、プロシージャまたはファンクションを実行するときに適用される権限が、修飾されていないオブジェクトの所有者とともに設定されます。この句で

は、INVOKER または DEFINER が選択されます。ただし、ユーザは別のユーザが所有するプロシージャまたはファンクションを作成することができます。この場合は、実際は定義者ではなく所有者の権限となります。

プロシージャまたはファンクションを作成するときに、すべてのオブジェクト名 (テーブル、プロシージャなど) を該当する所有者で修飾する場合は注意が必要です。プロシージャ内のオブジェクトが所有者に対して修飾されていない場合、そのオブジェクトが所有者として実行されるか呼び出し側として実行されるかにより、所有者が異なります。たとえば、user1 が次のプロシージャを作成するとします。

```
CREATE PROCEDURE user1.myProcedure()
  RESULT( columnA INT )
   「SQL SECURITY INVOKER」
  BEGIN
    SELECT columnA FROM table1;
  END;
```

別のユーザである user2 がこのプロシージャの実行を試行したときに user2.table1 が存在しない場合は、エラーが返されます。user1.table1 ではなく user2.table1 が存在する場合は、それが使用されます。

呼び出し側の権限を使用してプロシージャまたはファンクションを実行する場合、呼び出し側はプロシージャ、ファンクションまたはシステムプロシージャが動作するデータベースオブジェクトに必要な権限に加えて、そのプロシージャの EXECUTE 権限を持っている必要があります。

プロシージャまたはファンクションを呼び出し側として実行するのか定義者として実行するのかわからない場合は、その SQL 定義で SQL SECURITY 句を参照します。

データベースで権限付き操作を実行するプロシージャまたはファンクションに必要な権限を決定する場合は、sp_proc_priv システムプロシージャを使用します。「sp_proc_priv システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』を参照してください。

参照

- 「sp_proc_priv システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE DATABASE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「初期化ユーティリティ (dbinit)」『SQL Anywhere サーバ データベース管理』
- 「アップグレードユーティリティ (dbupgrad)」『SQL Anywhere サーバ データベース管理』
- 「ALTER DATABASE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE FUNCTION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「アップグレードユーティリティ (dbupgrad)」『SQL Anywhere サーバ データベース管理』

16.0 より前のバージョンのシステムプロシージャを呼び出し側または定義者として実行する

テーブルの変更のような、データベースで権限付きタスクを実行する 16.0 より前のバージョンのソフトウェアに存在するシステムプロシージャには、呼び出し側または定義者 (所有者) のどちらの権限でも実行できるものがあります。データベースを作成または初期化する場合、これらの特別なシステムプロシージャを、その所有者 (定義者) の権限で実行するか、または呼び出し側の権限で実行するかを指定できます。

データベースがこれらのシステムプロシージャを定義者モードで実行するように設定されている場合は、すでに必要な権限を持っている定義者 (通常は dbo ロールまたは SYS ロール) の権限でプロシージャが実行されるため、追加の権限は必要ありません。

呼び出し側モードで実行するようにデータベースが設定されている場合、呼び出し側はプロシージャのマニュアルに記載されている権限を持っている必要があります。呼び出し側には、プロシージャに対する EXECUTE 権限も必要ですが、これは、PUBLIC のメンバーであることにより継承されます。

注意
ユーザ定義のプロシージャのデフォルトの権限は、呼び出し側／定義者モードの影響を受けることはありません。つまり、ユーザ定義のプロシージャの定義で呼び出し側または定義者が指定されていない場合、プロシージャは定義者の権限で実行されます。

次の方法の 1 つを使用して、データベースでこれらのシステムプロシージャの作成時またはアップグレード時の実行方法を制御できます。

- **CREATE DATABASE...SYSTEM PROCEDURE AS DEFINER 文**　CREATE DATABASE...SYSTEM PROCEDURE AS DEFINER OFF の指定は、データベースサーバが呼び出し側の権限を適用することを意味します。これは、新しいデータベースのデフォルトの動作です。

  CREATE DATABASE...SYSTEM PROCEDURE AS DEFINER ON の指定は、データベースサーバが定義者 (所有者) の権限を適用することを意味します。これは、16.0 より前のバージョンのデータベースのデフォルトの動作です。

- **ALTER DATABASE UPGRADE...SYSTEM PROCEDURE AS DEFINER 文**　この句は、CREATE DATABASE 文と同じ動作となります。句が指定されていない場合は、アップグレードされたデータベースの既存の動作が維持されます。たとえば、16.0 より前のバージョンのデータベースをアップグレードした場合、定義者の権限による実行がデフォルトとなります。

- **-pd オプション、初期化ユーティリティ (dbinit)**　データベースの作成時に -pd オプションを指定すると、データベースサーバは、これらのシステムプロシージャの実行時に定義者の権限を適用します。-pd オプションを指定しない場合は、呼び出し側の権限の適用がデフォルトの動作となります。

- **-pd オプション、アップグレードユーティリティ (dbupgrad)**　データベースのアップグレード時に -pd Y オプションを指定すると、データベースサーバは、これらのシステムプロシージャの実行時に定義者の権限を適用します。

  -pd N オプションを指定すると、これらのシステムプロシージャの実行時に、データベースサーバは呼び出し側の権限を適用します。

  このオプションが指定されていない場合は、アップグレードされたデータベースの既存の動作が維持されます。

注意
PUBLIC システムロールには、すべてのシステムプロシージャに対する EXECUTE 権限が付与されます。新しく作成されたユーザには、デフォルトで PUBLIC ロールが付与されます。したがって、ユーザはシステムの EXECUTE 権限をすでに持っていることになります。

ユーザ定義関数とプロシージャのデフォルトは、呼び出し側と定義者の決定の影響を受けません。つまり、これらのシステムプロシージャを呼び出し側として実行するよう選択した場合でも、ユーザ定義のプロシージャは定義者のままです。

呼び出し側／定義者設定の影響を受けるプロシージャのリスト

次は、呼び出し側／定義者設定の影響を受けるシステムプロシージャのリストです。これらは、権限付き操作を実行していたデータベース上で 16.0 より前のバージョンの SQL Anywhere のシステムプロシージャです。これらのプロシージャを定義者として実行するようデータベースが設定されている場合、ユーザは、実行する必要のある各プロシージャで EXECUTE 権限のみが必要となります。データベースが INVOKER として実行するよう設定されている場合、ユーザは各

プロシージャで EXECUTE 権限を必要としませんが、代わりに各プロシージャを正常に実行するために個別の権限が必要となります。

- sa_audit_string
- sa_clean_database
- sa_column_stats
- sa_conn_activity
- sa_conn_compression_info
- sa_conn_info
- sa_conn_list
- sa_conn_options
- sa_conn_properties
- sa_db_info
- sa_db_list
- sa_db_properties
- sa_disable_auditing_type
- sa_disk_free_space
- sa_enable_auditing_type
- sa_external_library_unload
- sa_flush_cache
- sa_flush_statistics
- sa_get_histogram
- sa_get_request_profile
- sa_get_request_times
- sa_get_table_definition
- sa_index_density
- sa_index_levels
- sa_install_feature
- sa_java_loaded_classes
- sa_load_cost_model
- sa_make_object
- sa_materialized_view_can_be_immediate
- sa_procedure_profile
- sa_procedure_profile_summary
- sa_recompile_views
- sa_refresh_materialized_views
- sa_refresh_text_indexes
- sa_remove_tracing_data
- sa_reset_identity
- sa_save_trace_data
- sa_send_udp
- sa_server_option
- sa_set_tracing_level
- sa_table_fragmentation
- sa_table_page_usage
- sa_table_stats
- sa_text_index_vocab_nchar
- sa_unload_cost_model

- sa_user_defined_counter_add
- sa_user_defined_counter_set
- sa_validate
- sa_verify_password
- sp_copy_directory
- sp_copy_file
- sp_create_directory
- sp_delete_directory
- sp_delete_file
- sp_forward_to_remote_server
- sp_get_last_synchronize_result
- sp_list_directory
- sp_move_directory
- sp_move_file
- sp_remote_columns
- sp_remote_exported_keys
- sp_remote_imported_keys
- sp_remote_primary_keys
- sp_remote_procedures
- sp_remote_tables
- st_geometry_predefined_srs
- st_geometry_predefined_uom
- xp_cmdshell
- xp_getenv
- xp_read_file
- xp_sendmail
- xp_startmail
- xp_startsmtp
- xp_stopmail
- xp_stopsmtp
- xp_write_file

呼び出し側／定義者設定に関係なく呼び出し側の権限で実行されるプロシージャのリスト

権限付き操作を実行する 16.0 より前のバージョンのシステムプロシージャの小さなサブセットでは、**呼び出し側／定義者設定に関係なく**、タスクを実行するため呼び出し側は追加の権限を持っていることが必要です。これらのプロシージャで必要な追加の権限のリストを表示するには、各プロシージャのマニュアルを参照してください。

- sa_locks
- sa_report_deadlocks
- sa_snapshots
- sa_transactions
- sa_performance_statistics
- sa_performance_diagnostics
- sa_describe_shapefile
- sa_text_index_stats
- sa_get_user

参照

- 「sp_proc_priv システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE DATABASE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「初期化ユーティリティ (dbinit)」『SQL Anywhere サーバ データベース管理』
- 「アップグレードユーティリティ (dbupgrad)」『SQL Anywhere サーバ データベース管理』
- 「ALTER DATABASE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE FUNCTION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「アップグレードユーティリティ (dbupgrad)」『SQL Anywhere サーバ データベース管理』

データベース (SQL) が使用するセキュリティモデルの決定

データベースの作成時またはアップグレード時に指定されたセキュリティモデル設定 (呼び出し側と定義者) を、Capabilities データベースプロパティのクエリによって取得します。

前提条件

この作業を実行するための前提条件はありません。

内容と備考

デフォルトでは、新しいデータベースは INVOKER モデルのみを使用して権限付きシステムプロシージャを実行します。これは、権限付き操作を実行する 16.0 より前のバージョンのシステムプロシージャが、ユーザがプロシージャを呼び出した権限で実行されることを意味します。この設定は、データベースの作成時またはアップグレード時に変更できます。この方法を使用して、指定したセキュリティモデル設定 (呼び出し側と定義者) を決定できます。

♦ タスク

- Interactive SQL で、データベースにログインし、次の SQL 文を実行します。

```
SELECT IF ((HEXTOINT(SUBSTRING(DB_PROPERTY('Capabilities'),
1,LENGTH(DB_PROPERTY('Capabilities'))-20)) & 8) = 8)
THEN 1
ELSE 0
END IF
```

結果

A 1 は、権限付き操作を実行する 16.0 より前のバージョンのシステムプロシージャが、呼び出し側モデルを使用して実行されたことを示します。A 0 は、プロシージャが定義者 (所有者) の権限で実行されたことを示します。

次の手順

なし。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

参照

- 「呼び出し側／定義者設定の影響を受けるプロシージャのリスト」81 ページ
- 「16.0 より前のバージョンのシステムプロシージャを呼び出し側または定義者として実行する」79 ページ

プロシージャの作成 (Sybase Central の場合)

Sybase Central では、**プロシージャ作成ウィザード**に、プロシージャテンプレートを使用するオプションもあります。

前提条件

ユーザ本人が所有するプロシージャを作成するには、CREATE PROCEDURE システム権限が必要です。他のユーザが所有するプロシージャを作成するには、CREATE ANY PROCEDURE または CREATE ANY OBJECT の権限が必要です。

外部プロシージャを作成するには、CREATE EXTERNAL REFERENCE システム権限も必要です。

テンポラリプロシージャの作成には権限は必要ありません。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックします。
3. **[ファイル]** » **[新規]** » **[プロシージャ]** をクリックします。
4. **プロシージャ作成ウィザード**の指示に従います。
5. 右ウィンドウ枠で、**[SQL]** タブをクリックして、プロシージャコードを終了します。

結果

新しいプロシージャは、**[プロシージャとファンクション]** に表示されます。このプロシージャはアプリケーションで使用できます。

参照

- 「CREATE PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「複合文」113 ページ
- 「リモートプロシージャの作成 (Sybase Central の場合)」800 ページ
- 「リモートサーバ」772 ページ
- 「CALL 文」『SQL Anywhere サーバ SQL リファレンス』
- 「名前付きパラメータ」『SQL Anywhere サーバ SQL リファレンス』

プロシージャの変更 (Sybase Central の場合)

既存のプロシージャを変更できます。

前提条件

そのプロシージャの所有者であるか、または次のいずれかの権限を持っていることが必要です。

- ALTER ANY PROCEDURE システム権限
- ALTER ANY OBJECT システム権限

内容と備考

Sybase Central では、既存のプロシージャの名前を直接変更することはできません。新しい名前のプロシージャを新しく作成し、以前のコードをそこへコピーしてから、元のプロシージャを削除します。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックします。
3. プロシージャを選択します。
4. 次のいずれかの方法を使用して、プロシージャを編集します。

   - 右ウィンドウ枠で、**[SQL]** タブをクリックします。
   - プロシージャを右クリックして、**[新しいウィンドウで編集]** をクリックします。

     ヒント
     プロシージャごとに別のウィンドウを開いて、プロシージャ間でコードをコピーできます。

   - プロシージャのコメントを追加または編集するには、プロシージャを右クリックして、**[プロパティ]** をクリックします。

     データベースドキュメントウィザードを使用して、SQL Anywhere データベースをドキュメント化する場合、これらのコメントを出力に含めることができます。

結果

プロシージャのコードが変更されます。

参照

- 「データベースのドキュメント化」『SQL Anywhere サーバ データベース管理』
- 「ALTER PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「プロシージャの作成 (Sybase Central の場合)」85 ページ
- 「ストアドプロシージャの変換」662 ページ

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

プロシージャの呼び出し (SQL の場合)

CALL 文はプロシージャを呼び出します。アプリケーションプログラムまたは他のプロシージャやトリガからプロシージャを呼び出すことができます。

前提条件

そのプロシージャの所有者であるか、そのプロシージャに対する EXECUTE 権限を持っているか、または EXECUTE ANY PROCEDURE システム権限を持っていることが必要です。

プロシージャの EXECUTE 権限を付与されたすべてのユーザは、テーブルの権限がなくてもプロシージャを呼び出すことができます。

♦ タスク

● 次の文を実行してプロシージャを呼び出し、次の値を挿入します。

```
CALL procedure-name( values );
```

この呼び出しの後で、値が追加されたことを確認することもできます。

> **注意**
> クエリで呼び出すことで結果セットを返すプロシージャを呼び出すことができます。プロシージャの結果セットに対してクエリを実行し、WHERE 句やその他の SELECT 機能を適用して、結果セットを制限できます。

結果

プロシージャが呼び出され、実行されます。

例

次に、NewDepartment プロシージャを呼び出して、部署 Eastern Sales を追加する例を示します。

```
CALL NewDepartment( 210, 'Eastern Sales', 902 );
```

この呼び出しの後で、実際に新しく部署が追加されたことを確認するために、Departments テーブルを確認できます。

プロシージャの EXECUTE 権限を付与されたすべてのユーザは、Department テーブルの権限がなくても NewDepartment プロシージャを呼び出すことができます。

参照

● 「名前付きパラメータ」『SQL Anywhere サーバ SQL リファレンス』
● 「ユーザのセキュリティ (ロールと権限)」『SQL Anywhere サーバ データベース管理』
● 「CALL 文」『SQL Anywhere サーバ SQL リファレンス』
● 「GRANT EXECUTE 文」『SQL Anywhere サーバ SQL リファレンス』

プロシージャのコピー (Sybase Central の場合)

Sybase Central を使用して、データベース間または同じデータベース内でプロシージャをコピーできます。同じデータベース内でプロシージャをコピーする場合は、プロシージャの名前を変更するか、コピーするプロシージャで別の所有者を選択する必要があります。

前提条件

プロシージャをコピーして自分自身を所有者として割り当てる場合は、プロシージャのコピー先のデータベースで、CREATE PROCEDURE システム権限を持っていることが必要です。プロシージャをコピーして別のユーザを所有者として割り当てる場合は、プロシージャのコピー先のデータベースで、CREATE ANY PROCEDURE または CREATE ANY OBJECT のシステム権限を持っていることが必要です。

♦ タスク

1. Sybase Central の **[SQL Anywhere 16]** プラグインを使用すると、コピーするプロシージャを含むデータベースに接続できます。
2. プロシージャをコピーするデータベースに接続します。
3. 最初のデータベースの左ウィンドウ枠でコピーするプロシージャを選択して、それを 2 番目のデータベースの **[プロシージャとファンクション]** にドラッグします。

結果

新しいプロシージャが作成されて、元のプロシージャのコードがコピーされます。新しいプロシージャにコピーされるのは、プロシージャのコードだけです。権限など、その他のプロシージャのプロパティはコピーされません。

プロシージャの削除 (Sybase Central の場合)

Sybase Central を使用して、たとえば、必要なくなったときに、データベースからプロシージャを削除できます。

前提条件

そのプロシージャの所有者であるか、または次のいずれかのシステム権限を持っていることが必要です。

- DROP ANY PROCEDURE
- DROP ANY OBJECT

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックします。

3. プロシージャを右クリックして、**[削除]** をクリックします。

4. **[はい]** をクリックします。

結果

プロシージャがデータベースから削除されます。

次の手順

従属データベースオブジェクトの場合は、その定義を変更して削除されたプロシージャへの参照を削除する必要があります。

参照

- 「DROP PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』

ユーザ定義関数

ユーザ定義関数はプロシージャの集まりで、呼び出しを行った環境に単一の値を返します。

> **注意**
> SQL Anywhere はユーザ定義関数がスレッドセーフであるかどうかについて想定しません。これはアプリケーション開発者の責任になります。

CREATE FUNCTION の構文は、CREATE PROCEDURE 文の構文と若干異なります。

- IN、OUT、INOUT などのキーワードは必要ありません。すべてのパラメータは IN パラメータです。
- 返されるデータ型を指定するために RETURNS 句が必要です。
- 返される値を指定するために RETURN 文が必要です。
- 名前付きパラメータはサポートされていません。

ユーザ定義関数の作成

Sybase Central からユーザ定義関数を作成できます。ユーザ定義関数はプロシージャの集まりで、呼び出しを行った環境に単一の値を返します。

前提条件

ユーザ本人が所有する関数を作成するには、CREATE PROCEDURE システム権限が必要です。他のユーザが所有する関数を作成するには、CREATE ANY PROCEDURE または CREATE ANY OBJECT のシステム権限が必要です。

外部関数を作成するには、CREATE EXTERNAL REFERENCE システム権限が必要です。

テンポラリ関数の作成には権限は必要ありません。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で **[プロシージャとファンクション]** を右クリックし、**[新規]** » **[ファンクション]** をクリックします。
3. **ファンクション作成ウィザード**の指示に従います。
4. 右ウィンドウ枠で、**[SQL]** タブをクリックして、ファンクションコードを終了します。

結果

新しいファンクションは、**[プロシージャとファンクション]** に表示されます。

参照

- 「CREATE FUNCTION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE FUNCTION 文 [Web サービス]」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE FUNCTION 文 [外部呼び出し]」『SQL Anywhere サーバ SQL リファレンス』

ユーザ定義関数の呼び出し

ユーザ定義関数は、集合関数以外の組み込み関数が使われていればどこでも使用できます。

前提条件

その関数に対する EXECUTE 権限が必要です。

♦ タスク

1. Interactive SQL で、データベースに接続します。
2. ユーザ定義関数を使用して、EXECUTE 文を実行します。

結果

関数が呼び出され、実行されます。

例

例 1 : ユーザ定義関数の呼び出し　次の関数は、firstname 文字列と lastname 文字列を連結します。

```
CREATE FUNCTION fullname(
  firstname CHAR(30),
  lastname CHAR(30) )
RETURNS CHAR(61)
BEGIN
  DECLARE name CHAR(61);
  SET name = firstname || ' ' || lastname;
```

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

```
  RETURN (name);
END;
```

Interactive SQL で次の文を実行すると、姓と名前の入った 2 つのカラムから氏名が返されます。

```
SELECT FullName( GivenName, Surname )
 AS "Full Name"
 FROM Employees;
```

Full Name
Fran Whitney
Matthew Cobb
Philip Chin
...

Interactive SQL で次の文を実行すると、FullName ユーザ定義関数が使用され、指定された姓と名前から氏名が返されます。

```
SELECT FullName('Jane', 'Smith')
 AS "Full Name";
```

Full Name
Jane Smith

例 2：ローカル変数の宣言　次にローカル変数の宣言の例としてユーザ定義関数を示します。

> **注意**
> この関数は説明には役立ちますが、多数のローを含む SELECT に使用する場合は、性能が低くなることがあります。たとえば、テーブルに 100000 のローがあり、その中の 10000 のローを返すようなクエリの SELECT リストで関数を使用した場合、関数は 10000 回呼び出されます。同じクエリの WHERE 句に関数を使用した場合は、100000 回呼び出されます。

Customers テーブルには、カナダとアメリカの顧客が含まれます。ユーザ定義関数 Nationality は、Country カラムの入力データに基づいて 3 文字の国コードを生成します。

```
CREATE FUNCTION Nationality( CustomerID INT )
RETURNS CHAR( 3 )
BEGIN
   DECLARE nation_string CHAR(3);
   DECLARE nation country_t;
   SELECT DISTINCT Country INTO nation
   FROM Customers
   WHERE ID = CustomerID;
   IF nation = 'Canada' THEN
         SET nation_string = 'CDN';
   ELSE IF nation = 'USA' OR nation = ' ' THEN
         SET nation_string = 'USA';
      ELSE
         SET nation_string = 'OTH';
```

```
        END IF;
    END IF;
RETURN ( nation_string );
END;
```

この例では国名を入れる変数 nation_string を宣言し、SET 文を使用して値を nation_string に入れます。次に nation_string の値を、この関数を呼び出した環境に返します。

次に示すクエリは、Customers テーブルに含まれるカナダの顧客をすべてリストします。

```
SELECT *
FROM Customers
WHERE Nationality( ID ) = 'CDN';
```

参照

- 「ユーザ定義関数」89 ページ
- 「CREATE FUNCTION 文」『SQL Anywhere サーバ SQL リファレンス』

ユーザ定義関数の削除 (SQL の場合)

ユーザ定義関数は、明示的に削除されるまでデータベースに残ります。

前提条件

ユーザ定義関数の所有者であるか、または次のいずれかのシステム権限を持っていることが必要です。

- DROP ANY PROCEDURE
- DROP ANY OBJECT

♦ タスク

1. データベースに接続します。
2. 次のような DROP FUNCTION 文を実行します。

```
DROP FUNCTION function-name;
```

結果

ユーザ定義関数が削除されます。

例

次に、FullName 関数をデータベースから削除する文を示します。

```
DROP FUNCTION FullName;
```

参照

- 「DROP FUNCTION 文」『SQL Anywhere サーバ SQL リファレンス』

ユーザ定義関数実行機能の付与 (SQL の場合)

EXECUTE オブジェクトレベル権限を付与して、ユーザ定義関数を実行する機能を付与します。

前提条件

ユーザ定義関数の所有者であるか、またはその関数の EXECUTE 権限に対する管理権限を持っていることが必要です。

ユーザ定義関数の所有権はその関数の作成者に属しており、そのユーザが関数を実行する場合には権限は必要ありません。

内容と備考

関数を作成して、他のユーザがその関数を実行できるようにします。

♦ タスク

1. データベースに接続します。
2. 次のような GRANT EXECUTE 文を実行します。

```
GRANT EXECUTE ON function-name TO user-id;
```

結果

被付与者がプロシージャを実行できるようになりました。

次の手順

なし。

例

たとえば、Nationality 関数の作成者が別のユーザに Nationality の使用許可を与える文は、次のようになります。

```
GRANT EXECUTE ON Nationality TO BobS;
```

参照

- 「GRANT EXECUTE 文」『SQL Anywhere サーバ SQL リファレンス』

ユーザ定義関数に関する詳細情報

SQL Anywhere では、すべてのユーザ定義関数は、NOT DETERMINISTIC と宣言されないかぎり「べき等」として扱われます。べき等関数は、同じパラメータに対して一貫した結果を返し、副次効果はありません。同じパラメータを持つ冪等関数が連続して 2 回呼び出されている場合は、どちらの呼び出しでも同じ結果が返され、クエリのセマンティックに不要な副次効果は生じません。

非決定的関数と決定性関数の詳細については、「関数のキャッシュ」327 ページを参照してください。

トリガ

トリガとは、データを修正する文が実行されると自動的に実行されるストアドプロシージャの特別な形式です。トリガは、参照整合性や他の宣言制約では不十分な場合に使います。

検査項目を細かく設定して複雑な参照整合性を設定したり、既存のデータは制約の範囲から外れても許可するが新しいデータはチェックしたりする場合があります。トリガはこのようなときに使用すると便利です。また、データベースにアクセスするアプリケーションとは個別に、データベーステーブルのアクティビティのログを取るときにもトリガを使います。

注意
その後のトリガが起動しない LOAD TABLE、TRUNCATE、WRITETEXT の 3 つの特別な文があります。

トリガを実行する権限

トリガは、関連するテーブルまたはビューの所有者の権限によって実行されます。そのトリガを起動したユーザの ID ではありません。トリガはユーザが直接変更できないテーブルのローを変更できます。

トリガが起動しないようにするには、-gf サーバオプションを指定するか、または fire_triggers オプションを設定します。「-gf データベースサーバオプション」『SQL Anywhere サーバ データベース管理』または「fire_triggers オプション」『SQL Anywhere サーバ データベース管理』を参照してください。

- 「-gf データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「fire_triggers オプション」『SQL Anywhere サーバ データベース管理』

トリガのタイプ

SQL Anywhere では、次のトリガのタイプがサポートされています。

- **BEFORE トリガ**　BEFORE トリガは、トリガ元アクションが実行される前に実行されます。BEFORE トリガはテーブルに定義できますが、ビューには定義できません。

- **AFTER トリガ**　AFTER トリガは、トリガ元アクションが完了した後に実行されます。AFTER トリガはテーブルに定義できますが、ビューには定義できません。

- **INSTEAD OF トリガ**　INSTEAD OF トリガは、トリガ元アクションの代わりに実行される条件付きのトリガです。INSTEAD OF トリガはテーブルとビューに定義できます (マテリアライズドビューを除く)。

トリガイベント

トリガを起動するイベントのリストを次に示します。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

動作	説明
INSERT	トリガの関連するテーブルに新しいローが挿入されたときに、トリガが起動される。
DELETE	トリガの関連するテーブル内のローが削除されたときに、トリガが起動される。
UPDATE	トリガの関連するテーブル内のローが更新されたときに、トリガが起動される。
UPDATE OF *column-list*	トリガの関連するテーブル内のローが、**column-list** 中のカラムが変更されるなどして更新されたときに、トリガが起動される。

処理が必要なイベントごとにトリガを個別に作成できます。または、共有するアクションや、イベントに応じたアクションが複数ある場合は、すべてのイベントに対して 1 つのトリガを作成し、IF 文を使用して実行するアクションを区別できます。

参照

- 「データ整合性」833 ページ
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「TRUNCATE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「WRITETEXT 文 [T-SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「LOAD TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「INSTEAD OF トリガ」105 ページ
- 「CREATE TRIGGER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「アトミックな複合文」114 ページ

トリガのタイミング

トリガのレベルには、「ローレベル」と「文レベル」があります。

- ローレベルトリガは、変更されるローごとに一回実行されます。ローレベルトリガは、ローの変更前または変更後に実行されます。

  対象ローの新しいイメージと古いイメージのカラム値は、変数によってトリガから使用可能になります。

- 文レベルトリガは、トリガする文全体の処理が完了した後に実行されます。トリガする文の対象ローは、ローの新しいイメージと古いイメージを表すテンポラリテーブルによってトリガから使用可能になります。SQL Anywhere では、文レベル BEFORE トリガはサポートされていません。

トリガ実行のタイミングは柔軟に設定できるので、実行に応じてカスケード更新または削除の実行が決まるような、参照整合性に依存するトリガに対して有効です。

トリガの実行中にエラーが発生すると、トリガを起動した操作そのものがエラーになります。INSERT、UPDATE、DELETE はアトミックオペレーションです。これらがエラーになると、ト

リガの結果とトリガが起動したプロシージャを含め、その文のすべての結果がキャンセルされます。

テーブルに対するトリガの作成 (Sybase Central の場合)

[トリガ作成ウィザード] を使用して、テーブル内にトリガを作成します。

前提条件

CREATE ANY TRIGGER または CREATE ANY OBJECT のシステム権限が必要です。さらに、そのトリガが作成されるテーブルの所有者であるか、または次のいずれかの権限を持っていることも必要です。

- そのテーブルに対する ALTER 権限
- ALTER ANY TABLE システム権限
- ALTER ANY OBJECT システム権限

♦ タスク

1. **[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で **[トリガ]** を右クリックし、**[新規]** » **[トリガ]** をクリックします。
3. **[トリガ作成ウィザード]** の指示に従います。
4. コードを完了するには、右ウィンドウ枠で、**[SQL]** タブをクリックします。

結果

新しいトリガが作成されます。

参照

- 「CREATE TRIGGER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「データベース接続」『SQL Anywhere サーバ データベース管理』
- 「複合文」113 ページ

テーブル内でのトリガの作成 (SQL の場合)

CREATE TRIGGER 文を使用して、テーブル内にトリガを作成します。トリガの本文は複合文、つまり BEGIN 文と END 文に挟まれ、セミコロンで区切られた SQL 文のセットから構成されています。

前提条件

CREATE ANY TRIGGER または CREATE ANY OBJECT のシステム権限が必要です。さらに、そのトリガが作成されるテーブルの所有者であるか、または次のいずれかの権限を持っていることも必要です。

- そのテーブルに対する ALTER 権限
- ALTER ANY TABLE システム権限
- ALTER ANY OBJECT システム権限

内容と備考

COMMIT と ROLLBACK 文、いくつかの ROLLBACK TO SAVEPOINT 文をトリガ内に使用することはできません。

♦ タスク

1. データベースに接続します。
2. CREATE TRIGGER 文を実行します。

結果

新しいトリガが作成されます。

例

例 1：ローレベルの INSERT トリガ　次にローレベルの INSERT トリガの例を示します。新しい従業員の生年月日が正しく入力されたかどうかをチェックします。

```
CREATE TRIGGER check_birth_date
  AFTER INSERT ON Employees
REFERENCING NEW AS new_employee
FOR EACH ROW
BEGIN
    DECLARE err_user_error EXCEPTION
  FOR SQLSTATE '99999';
  IF new_employee.BirthDate > 'June 6, 2001' THEN
        SIGNAL err_user_error;
  END IF;
END;
```

注意
SQL Anywhere のサンプルデータベースに check_birth_date というトリガがすでにある可能性があります。このトリガがある場合に上記の SQL 文を実行しようとすると、トリガの定義が既存のトリガと矛盾していることを示すエラーが表示されます。

このトリガは、Employees テーブルに新しいローが追加されると起動されます。2001 年 6 月 6 日以降の生年月日に対応する新しいローを検知し、エラーにします。

フレーズ REFERENCING NEW AS new_employee は、トリガコード中の文がエイリアス new_employee を使用して、新しいローのデータを参照できるようにします。

エラーが発生すると、トリガ元の文、およびトリガによる前の変更内容がすべて取り消されます。

Employees テーブルに複数のローを追加する INSERT 文の場合は、新しいローごとに check_birth_date トリガが起動されます。どれか 1 つのローでトリガが失敗すると、INSERT 文のすべての結果がロールバックされます。

ローを追加した後でなく、追加する前にトリガが起動されるようにするには、例文の 2 行目を次のように変更します。

```
BEFORE INSERT ON Employees
```

REFERENCING NEW 句は追加されるローの値を参照します。この句はトリガが起動されるタイミング (BEFORE と AFTER) には影響されません。

トリガではなく、宣言参照整合性や検査制約を使用して整合性を確保する方が簡単な場合があります。たとえば、上記の例でカラム検査制約を使用すると、さらに効率が良く、簡潔になります。

```
CHECK (@col <= 'June 6, 2001')
```

例 2：ローレベルの DELETE トリガの例　次に示す CREATE TRIGGER 文は、ローレベルの DELETE トリガを定義します。

```
CREATE TRIGGER mytrigger
BEFORE DELETE ON Employees
REFERENCING OLD AS oldtable
FOR EACH ROW
BEGIN
  ...
END;
```

REFERENCING OLD 句は、トリガが起動されるタイミング (BEFORE または AFTER) に影響されず、エイリアス oldtable を使用して、削除されるローの値を削除トリガコードが参照できるようにします。

例 3：文レベルの UPDATE トリガの例　文レベルの UPDATE トリガを作成する CREATE TRIGGER 文の例を次に示します。

```
CREATE TRIGGER mytrigger AFTER UPDATE ON Employees
REFERENCING NEW AS table_after_update
        OLD AS table_before_update
FOR EACH STATEMENT
BEGIN
  ...
END;
```

REFERENCING NEW 句と REFERENCING OLD 句は、UPDATE トリガのコマンド文が更新の前と後の両方の値を参照できるようにします。テーブルエイリアス table_after_update は、新しいローのカラムを参照し、テーブルエイリアス table_before_update は古いローのカラムを参照します。

REFERENCING NEW 句と REFERENCING OLD 句は、文レベルとローレベルのトリガで少し異なる意味を持ちます。文レベルではテーブルが対象になりますが、ローレベルでは変更されるローが対象になります。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

参照

- 「CREATE TRIGGER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「データベース接続」『SQL Anywhere サーバ データベース管理』
- 「COMMIT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ROLLBACK TO SAVEPOINT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「複合文」113 ページ

トリガの実行

トリガは指定したテーブルに INSERT、UPDATE、DELETE が行われたときに、自動的に実行されます。ローレベルトリガは、ローが影響を受けるごとに起動され、文レベルトリガは、文全体が一度に起動されます。

INSERT、UPDATE、DELETE がトリガを起動すると、トリガのタイプ (BEFORE または AFTER) によって、次の順序で操作が行われます。

1. BEFORE トリガが起動します。
2. 追加などの操作そのものが実行されます。
3. 参照動作を行います。
4. AFTER トリガが起動します。

注意
CREATE TRIGGER 文を使用してトリガを作成するときにトリガのタイプを指定しなかった場合は、デフォルトで AFTER になります。

手順の途中で、プロシージャまたはトリガの内部で処理されないエラーが発生すると、それより以前の手順は取り消され、それ以降の手順は実行されません。トリガを起動した操作そのものも失敗となります。

参照

- 「CREATE TRIGGER 文」『SQL Anywhere サーバ SQL リファレンス』

トリガの変更

Sybase Central を使用してトリガを変更し、トリガのコードを編集またはコメントを追加します。

前提条件

コメントを追加または編集するには、次のいずれかのシステム権限を持っている必要があります。

- COMMENT ANY OBJECT
- ALTER ANY TRIGGER
- ALTER ANY OBJECT
- CREATE ANY TRIGGER
- CREATE ANY OBJECT

コードを編集するには、ALTER ANY OBJECT システム権限または ALTER ANY TRIGGER システム権限および次の条件の 1 つを持っている必要があります。

- 基礎となるテーブルの所有者である。
- ALTER ANY TABLE システム権限
- 基礎となるテーブルに対する ALTER 権限

内容と備考

Sybase Central では、既存のトリガの名前を直接変更することはできません。代わりに、新しい名前を付けて新しくトリガを作成し、このトリガに以前のコードをコピーしてから、元のトリガを削除します。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[トリガ]** をダブルクリックします。
3. トリガを選択します。
4. 次のいずれかの方法を使用して、トリガを変更します。

オプション	アクション
コードの編集	トリガを右クリックして、**[新しいウィンドウで編集]** をクリックするか、右ウィンドウ枠の **[SQL]** タブでコードを編集できます。 **ヒント** プロシージャごとに別のウィンドウを開いて、トリガ間でコードをコピーできます。

オプション	アクション
コメントの追加	トリガのコメントを追加または編集するには、トリガを右クリックして、**[プロパティ]** をクリックします。 **データベースドキュメントウィザード**を使用して、SQL Anywhere データベースをドキュメント化する場合、これらのコメントを出力に含めることができます。

結果

トリガのコードが変更されます。

参照

- 「データベース接続」『SQL Anywhere サーバ データベース管理』
- 「データベースのドキュメント化」『SQL Anywhere サーバ データベース管理』
- 「ストアドプロシージャの変換」 662 ページ
- 「ALTER TRIGGER 文」『SQL Anywhere サーバ SQL リファレンス』

トリガの削除

トリガが不要になった場合は、Sybase Central を使用してデータベースからトリガを削除します。

前提条件

そのトリガの所有者であるか、または次のいずれかのシステム権限を持っていることが必要です。

- DROP ANY TRIGGER
- DROP ANY OBJECT

♦ タスク

1. **[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[トリガ]** をダブルクリックします。
3. トリガを選択して、**[編集]** » **[削除]** をクリックします。
4. **[はい]** をクリックします。

結果

トリガがデータベースから削除されます。

次の手順

従属データベースオブジェクトの場合は、その定義を変更して削除されたトリガへの参照を削除する必要があります。

参照

- 「データベース接続」『SQL Anywhere サーバ データベース管理』
- 「DROP TRIGGER 文」『SQL Anywhere サーバ SQL リファレンス』

例：トリガ操作の一時的な無効化

ユーザがカラムデータに対して (トリガを起動する) アクションを実行するときに、トリガの操作が一時的に無効になるようにトリガを設定できます。その場合でも、事前定義の接続変数を含むプロシージャを使用して、トリガを起動してその操作を実行できます。ユーザは、トリガが起動されてもトリガ操作を実行しないで、カラムに対して INSERT、ALTER、または DELETE を実行できます。

> **注意**
> ローレベルのトリガを使用している場合は、WHEN 句を使用して、トリガをいつ起動するかを指定します。

例：1 つのトリガの操作の一時的な無効化

この例では、接続変数が存在するかどうかに基づいてトリガの操作を無効にします。

1. 接続変数の状態を確認してトリガ論理が有効かどうかを判断する AFTER INSERT トリガを作成します。変数が存在しない場合、トリガの操作は有効です。

```
CREATE TRIGGER myTrig AFTER INSERT
REFERENCING NEW AS new-name
FOR EACH STATEMENT
BEGIN
  DECLARE @execute_trigger integer;
  IF varexists('enable_trigger_logic') = 1 THEN
    SET @execute_trigger = enable_trigger_logic;
  ELSE
    SET @execute_trigger = 1;
  END IF;
  IF @execute_trigger = 1 THEN
    ... -your-trigger-logic
  END IF;
END;
```

2. 次のコードを文に追加し、手順 1 で作成したトリガを呼び出します。文では、接続変数を使用してトリガをいつ無効にするかを制御します。文は無効にするコードを囲んでいる必要があります。

```
...
 IF varexists('enable_trigger_logic') = 0 THEN
     CREATE VARIABLE enable_trigger_logic INT;
 END IF;
 SET enable_trigger_logic = 0;
  ... 「execute-your-code-that-you-do-not-want-triggers-to-run」
```

```
SET enable_trigger_logic = 1;
... now-your-trigger-logic-will-do-its-work
```

例：複数のトリガの操作を一時的に無効にする

この例では、例 1 の接続変数の方法を使用して、複数のトリガの操作を制御します。複数のトリガを有効および無効にするために呼び出すことができる 2 つのプロシージャを作成します。トリガ操作が有効かどうかを確認するために使用することができる関数も作成します。

1. トリガ操作を無効にするために呼び出すことができるプロシージャを作成します。動作は接続変数の値に基づいています。

```
CREATE PROCEDURE sp_disable_triggers()
BEGIN
    IF VAREXISTS ('enable_trigger_logic') = 0 THEN
      CREATE VARIABLE enable_trigger_logic INT;
    END IF;
    SET enable_trigger_logic = 0;
END;
```

2. トリガ操作を有効にするために呼び出すことができるプロシージャを作成します。動作は接続変数の値に基づいています。

```
CREATE PROCEDURE sp_enable_triggers()
BEGIN
     IF VAREXISTS ('enable_trigger_logic') = 0 THEN
       CREATE VARIABLE enable_trigger_logic INT;
     END IF;
    SET enable_trigger_logic = 1;
END;
```

3. トリガ操作が有効かどうかを判別するために呼び出すことができる関数を作成します。

```
CREATE FUNCTION f_are_triggers_enabled()
RETURNS INT
BEGIN
    IF VAREXISTS ('enable_trigger_logic') = 1 THEN
      RETURN enable_trigger_logic;
    ELSE
      RETURN 1;
    END IF;
END;
```

4. 操作を制御するトリガに IF 句を追加します。

```
IF f_are_triggers_enabled() = 1 THEN
   ... your-trigger-logic
END IF;
```

5. トリガ操作を有効にするために手順 2 で作成したプロシージャを呼び出します。

```
CALL sp_enable_triggers();
... execute-code-where-trigger-logic-runs
```

6. トリガ操作を無効にするために手順 1 で作成したプロシージャを呼び出します。

```
CALL sp_disable_triggers();
... execute-your-code-where-trigger-logic-is-disabled
```

参照

- 「CREATE TRIGGER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「-gf データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- fire_triggers 接続プロパティ『SQL Anywhere サーバ データベース管理』
- 「トリガ」94 ページ

トリガを実行する権限

ユーザはトリガを実行できないため、トリガ実行の権限を付与することはできません。SQL Anywhere は、データベースでのアクションに対応してトリガを起動します。トリガが実行される場合は、トリガに関連する権限があり、その動作を実行する権利を定義します。

トリガは、トリガが定義されているテーブルの所有者の権限を使用して実行します。トリガを起動する原因となったユーザの権限や、トリガを作成したユーザの権限ではありません。

トリガがテーブルを参照するときは、そのテーブルの所有者名を特に指定しないで、テーブル作成者のロールメンバーシップを使います。たとえば、user_1.Table_A にあるトリガが Table_B を参照し、Table_B の所有者の名前を指定しないとします。この場合、Table_B が user_1 によって作成されたか、user_1 が Table_B の所有者であるロールの (直接または間接的に) メンバーでなければなりません。どちらの条件も満たされない場合は、トリガを起動すると、データベースサーバがテーブルが見つからないことを示すメッセージを返します。

また、user_1 はトリガに指定された操作を実行するための権限を持っていなければなりません。

参照

- 「ユーザのセキュリティ (ロールと権限)」『SQL Anywhere サーバ データベース管理』

トリガに関する詳細情報

トリガの理解しにくい一面として、複数のトリガが同じトリガ元アクションの影響を受ける場合に、これらのトリガが実行される順序があります。競合するトリガが実行されるかどうか、また実行される場合の順序は、トリガのタイプ (BEFORE、INSTEAD OF、または AFTER) とトリガのスコープ (ローレベルまたは文レベル) の 2 点で決まります。

UPDATE 文は、複数のテーブルのカラム値を変更できます。トリガ起動の順序は各テーブルで同じですが、テーブルが更新される順序は保証されません。

ローレベルのトリガの場合、BEFORE トリガが実行されてから INSTEAD OF トリガが実行され、その後に AFTER トリガが実行されます。特定のローのローレベルのトリガがすべて実行されてから、後続のローのトリガが実行されます。

文レベルのトリガの場合、INSTEAD OF トリガが実行されてから AFTER トリガが実行されます。文レベルの BEFORE トリガはサポートされていません。

文レベルとローレベルの AFTER トリガが競合する場合は、ローレベルの AFTER トリガがすべて完了してから文レベルのトリガが実行されます。

文レベルとローレベルの INSTEAD OF トリガが競合する場合、ローレベルのトリガは実行されません。

AFTER STATEMENT トリガに対して作成される OLD および NEW テンポラリテーブルは、基本となるベーステーブルと同じスキーマを持ち、カラム名とデータ型は同じです。ただし、これらのテーブルにはプライマリキー、外部キー、またはインデックスはありません。OLD および NEW テンポラリテーブル内のローの順序は保証されず、ベーステーブルのローが元々更新された順序と一致しない可能性があります。

INSTEAD OF トリガ

INSTEAD OF トリガは、トリガが実行されるとトリガ元アクションはスキップされ、代わりに指定されたアクションが実行される点で BEFORE トリガや AFTER トリガと異なります。

INSTEAD OF トリガに固有の機能や制限を次に示します。

- INSTEAD OF トリガは、特定のテーブルのトリガイベントごとに 1 つだけ指定できます。
- INSTEAD OF トリガはテーブルまたはビューに定義できます。ただし、INSTEAD OF トリガはマテリアライズドビューには定義できません。マテリアライズドビューには INSERT 文、DELETE 文、UPDATE 文などの DML 操作を実行できないからです。
- INSTEAD OF トリガを定義するときは、ORDER 句または WHEN 句は指定できません。
- INSTEAD OF トリガは、UPDATE OF *column-list* トリガイベントには定義できません。
- INSTEAD OF トリガが再帰を実行するかどうかは、トリガのターゲットがベーステーブルであるか、ビューであるかで異なります。ビューの場合は再帰が発生しますが、ベーステーブルの場合は発生しません。たとえば、INSTEAD OF トリガによって、トリガが定義されているベーステーブルに対して DML 操作が実行されても、これらの操作でトリガは実行されません (BEFORE トリガや AFTER トリガを含む)。ターゲットがビューの場合は、ビューに対して実行された操作ですべてのトリガが実行されます。
- テーブルに INSTEAD OF トリガが定義されている場合、ON EXISTING 句を含む INSERT 文をテーブルに対して実行できません。実行しようとすると、SQLE_INSTEAD_TRIGGER エラーが返されます。
- WITH CHECK OPTION を指定して定義されているか、WITH CHECK OPTION を指定して定義されている別のビューにネストされており、INSTEAD OF INSERT トリガが定義されているビューには、INSERT 文を実行できません。UPDATE 文と DELETE 文も同様です。実行しようとすると、SQLE_CHECK_TRIGGER_CONFLICT エラーが返されます。
- 位置付け更新、位置付け削除、PUT 文、またはワイド挿入操作の結果として INSTEAD OF トリガが実行されると、SQLE_INSTEAD_TRIGGER_POSITIONED エラーが返されます。

INSTEAD OF トリガを使用した更新不可のビューの更新

INSTEAD OF トリガを使用すると、派生の関係で更新不可能なビューに INSERT 文、UPDATE 文、または DELETE 文を実行できます。トリガの本文で、対応する INSERT、UPDATE、DELETE 文を実行する意味を定義します。たとえば、次のビューを作成するとします。

```
CREATE VIEW V1 ( Surname, GivenName, State )
  AS SELECT DISTINCT Surname, GivenName, State
      FROM Contacts;
```

DISTINCT キーワードによって、派生の関係で V1 が更新不可能になるので、V1 のローは削除できません。つまり、データベースサーバでは、V1 からローを削除する意味を明確に特定できません。ただし、V1 への削除操作を実装する INSTEAD OF DELETE トリガを定義することはできます。たとえば、次のトリガでは、指定した Surname、GivenName、State のローが Contacts から削除されるときに、そのすべてのローが削除されます。

```
CREATE TRIGGER V1_Delete
  INSTEAD OF DELETE ON V1
  REFERENCING OLD AS old_row
  FOR EACH ROW
BEGIN
    DELETE FROM Contacts
      WHERE Surname = old_row.Surname
        AND GivenName = old_row.GivenName
        AND State = old_row.State
END;
```

V1_Delete トリガを定義すると、V1 からローを削除できます。さらに、V1 で INSERT 文と UPDATE 文を実行させる他の INSTEAD OF トリガを定義することもできます。

INSTEAD OF DELETE トリガが定義されたビューが別のビューにネストされている場合は、DELETE に対する更新可能性を確認するためにベーステーブルのように処理されます。INSERT 操作と UPDATE 操作も同様です。前の例の続きで、別のビューを作成します。

```
CREATE VIEW V2 ( Surname, GivenName ) AS
  SELECT Surname, GivenName from V1;
```

V1_Delete トリガがなければ、V2 からローを削除することはできません。これは、派生の関係で V1 は更新不可能なので、V2 も更新不可能になるからです。しかし、V1 に INSTEAD OF DELETE トリガを定義すれば、V2 からローを削除できます。V2 からローが削除されるたびに V1 からローが削除され、V1_Delete トリガが実行されます。

INSTEAD OF トリガをネストされているビューに定義する場合は、これらのトリガが実行されるときに、意図しない影響がある可能性があるため、注意してください。意図する動作を明示的にするには、ネストされているビューを参照するすべてのビューに INSTEAD OF トリガを定義します。

次のトリガを V2 に定義し、DELETE 文の意図した動作を実行できます。

```
CREATE TRIGGER V2_Delete
  INSTEAD OF DELETE ON V2
  REFERENCING OLD AS old_row
  FOR EACH ROW
BEGIN
    DELETE FROM Contacts
      WHERE Surname = old_row.Surname
        AND GivenName = old_row.GivenName
END;
```

V2_Delete トリガによって、V1 に対する INSTEAD OF DELETE トリガが削除または変更されても、V2 に対する削除操作の動作は変わりません。

参照

- 「CREATE TRIGGER 文」『SQL Anywhere サーバ SQL リファレンス』

バッチ

バッチは一緒に送信されてグループとして次々に実行される一連の SQL 文です。プロシージャ (CASE、IF、LOOP など) で使われる制御文はバッチでも使えます。バッチが BEGIN/END で囲まれた複合文で構成される場合、ホスト変数、変数のローカル宣言、カーソル、テンポラリテーブル、例外を含めることもできます。ホスト変数参照は、次の制限付きでバッチ内で使用できます。

- ホスト変数を参照できるのはバッチ内の 1 文だけです。
- ホスト変数を使用する文の前に、結果セットを返す文を入れることはできません。

バッチの使用を明確に示すために、BEGIN/END を使うことをおすすめします。

バッチ内の文はセミコロンで区切ることができます。その場合、バッチは Watcom-SQL 構文に準拠しています。文を区切るためにセミコロンを使用しない複数文のバッチは、Transact-SQL 構文に準拠します。バッチの構文によって、バッチ内で使用できる文とバッチ内でのエラーの処理方法が決まります。

多くの点で、バッチはストアドプロシージャに似ていますが、いくつかの違いがあります。

- バッチには名前がありません。
- バッチにはパラメータを使用できません。
- バッチは永続的にデータベースに保存されません。
- バッチは異なる接続で共有できません。

簡単なバッチは、デリミタのない SQL 文のセットで、次の行に go という単語が続きます。次の例では、Eastern Sales という部署を作成し、Massachusetts のすべての営業担当者を Eastern Sales に転送します。これは Transact-SQL バッチの例です。

```
INSERT
INTO Departments ( DepartmentID, DepartmentName )
VALUES ( 220, 'Eastern Sales' )

UPDATE Employees
SET DepartmentID = 220
WHERE DepartmentID = 200
AND State = 'MA'

COMMIT
go
```

go という単語は Interactive SQL によって認識され、前の文は 1 つのバッチとしてサーバに送信されます。

次の例は見た目は似ていますが、Interactive SQL での処理はまったく異なります。この例では、Transact-SQL 構文を使用しません。各文はセミコロンで区切られています。Interactive SQL はセ

ミコロンで区切られた各文を個別にサーバに送信します。この場合は、バッチとしては処理されません。

```
INSERT
INTO Departments ( DepartmentID, DepartmentName )
VALUES ( 220, 'Eastern Sales' );

UPDATE Employees
SET DepartmentID = 220
WHERE DepartmentID = 200
AND State = 'MA';

COMMIT;
```

Interactive SQL でバッチとして処理するには、BEGIN ... END を使用して、複合文に変更します。次の構文は、前の例を修正したものです。複合文に含まれる 3 つの文は、バッチとしてサーバに送信されます。

```
BEGIN
  INSERT
  INTO Departments ( DepartmentID, DepartmentName )
  VALUES ( 220, 'Eastern Sales' );

  UPDATE Employees
  SET DepartmentID = 220
  WHERE DepartmentID = 200
  AND State = 'MA';

  COMMIT;
END
```

この例の場合、サーバがバッチと個別のどちらで文を実行しても結果は同じになります。ただし、結果が異なる場合もあります。次の例を考えます。

```
DECLARE @CurrentID INTEGER;
SET @CurrentID = 207;
SELECT Surname FROM Employees
  WHERE EmployeeID=@CurrentID;
```

Interactive SQL を使用してこの例を実行すると、データベースサーバが変数が見つからないことを示すエラーを返します。このエラーは、Interactive SQL が 3 つの文を個別にサーバに送信するために発生します。これらの文はバッチとしては実行されません。このようなエラーに対処するには、複合文を使用して Interactive SQL が強制的に 3 つの文をバッチとしてサーバに送信するようにします。次の例では、複合文を使用しています。

```
BEGIN
  DECLARE @CurrentID INTEGER;
  SET @CurrentID = 207;
  SELECT Surname FROM Employees
    WHERE EmployeeID=@CurrentID;
END
```

一連の文を BEGIN と END で囲んだ場合、Interactive SQL は強制的に文をバッチとして処理します。

IF 文は複合文の一例です。Interactive SQL は、次の文を 1 つのバッチとしてサーバに送信します。

```
IF EXISTS(   SELECT *
             FROM SYSTAB
             WHERE table_name='Employees' )
THEN
  SELECT   Surname AS LastName,
           GivenName AS FirstName
  FROM Employees;
  SELECT Surname, GivenName
  FROM Customers;
  SELECT Surname, GivenName
  FROM Contacts;
ELSE
  MESSAGE 'The Employees table does not exist'
  TO CLIENT;
END IF
```

別の方法で SQL 文を作成して実行した場合、この例は適用されません。たとえば、ODBC を使用するアプリケーションでは、セミコロンで区切られた文をバッチとして作成および実行できます。

Interactive SQL の文とサーバ向けの SQL 文が混在している場合は、注意が必要です。次の例は、Interactive SQL の文と SQL 文を混合する場合の問題を示します。この例では、Interactive SQL の OUTPUT 文が複合文に組み込まれているので、その他のすべての文と一緒にバッチとしてサーバに送信され、構文エラーが発生します。

```
IF EXISTS(   SELECT *
             FROM SYSTAB
             WHERE table_name='Employees' )
THEN
  SELECT   Surname AS LastName,
           GivenName AS FirstName
  FROM Employees;
  SELECT Surname, GivenName
  FROM Customers;
  SELECT Surname, GivenName
  FROM Contacts;
  OUTPUT TO 'c:¥¥temp¥¥query.txt';
ELSE
  MESSAGE 'The Employees table does not exist'
  TO CLIENT;
END IF
```

正しい OUTPUT 文の例は、次のとおりです。

```
IF EXISTS(   SELECT *
             FROM SYSTAB
             WHERE table_name='Employees' )
THEN
  SELECT   Surname AS LastName,
           GivenName AS FirstName
  FROM Employees;
  SELECT Surname, GivenName
  FROM Customers;
  SELECT Surname, GivenName
  FROM Contacts;
ELSE
  MESSAGE 'The Employees table does not exist'
  TO CLIENT;
END IF;
OUTPUT TO 'c:¥¥temp¥¥query.txt';
```

参照

- 「Transact-SQL のバッチ」661 ページ
- 「SQL 文の実行 (Interactive SQL)」『SQL Anywhere サーバ データベース管理』

プロシージャ、トリガ、ユーザ定義関数の構造

プロシージャまたはトリガの本文は複合文で構成されています。複合文は、SQL 文のセットを BEGIN と END で囲んだものです。各文はセミコロンで区切ります。

参照

- 「複合文」113 ページ

プロシージャのパラメータ宣言

プロシージャパラメータは、CREATE PROCEDURE 文にリストとして表示されます。パラメータ名は、カラム名など他のデータベース識別子に対するルールに従って付けてください。パラメータは有効なデータ型で、キーワード IN、OUT、INOUT のいずれかのプレフィクスが付いています。デフォルトでは、パラメータは INOUT パラメータです。これらのキーワードには、次のような意味があります。

- **IN**　引数はプロシージャに値を提供する式です。
- **OUT**　引数はプロシージャから値を与えられる変数です。
- **INOUT**　引数はプロシージャに値を提供する変数で、プロシージャから新しい値を与えられることもあります。

CREATE PROCEDURE 文中のプロシージャパラメータにはデフォルト値を設定できます。デフォルト値は定数で、NULL でもかまいません。たとえば、次に示すプロシージャは、IN パラメータのデフォルトとして NULL を指定しています。これは意味のないクエリを実行するのを避けるためです。

```
CREATE PROCEDURE CustomerProducts(
    IN customer_ID
            INTEGER DEFAULT NULL )
RESULT ( product_ID INTEGER,
      quantity_ordered INTEGER )
BEGIN
  IF customer_ID IS NULL THEN
    RETURN;
  ELSE
    SELECT    Products.ID,
            sum( SalesOrderItems.Quantity )
      FROM   Products,
              SalesOrderItems,
            SalesOrders
    WHERE SalesOrders.CustomerID = customer_ID
    AND SalesOrders.ID = SalesOrderItems.ID
    AND SalesOrderItems.ProductID = Products.ID
    GROUP BY Products.ID;
  END IF;
END;
```

 Copyright © 2013, SAP AG or an SAP affiliate company.

次に示す文は DEFAULT NULL を割り当て、プロシージャはクエリを実行しないで戻ります。

```
CALL CustomerProducts();
```

参照

- 「SQL データ型」『SQL Anywhere サーバ SQL リファレンス』

プロシージャにパラメータを渡す方法

ストアドプロシージャパラメータのデフォルト値は、CALL 文の 2 通りの形式のどちらでも使用できます。

CREATE PROCEDURE 文の引数リストの末尾にオプションのパラメータがある場合、これらは CALL 文で省略できます。次に示すのは、INOUT パラメータを 3 つ持つプロシージャの例です。

```
CREATE PROCEDURE SampleProcedure(
    INOUT var1 INT DEFAULT 1,
            INOUT var2 int DEFAULT 2,
            INOUT var3 int DEFAULT 3 )
...
```

この例では、プロシージャを呼び出す環境で、プロシージャに渡す数値を格納するための変数を 3 つ設定してあるものと想定しています。

```
CREATE VARIABLE V1 INT;
CREATE VARIABLE V2 INT;
CREATE VARIABLE V3 INT;
```

次に示すように、SampleProcedure プロシージャは最初のパラメータを指定するだけで呼び出せます。この場合、**var2** と **var3** にはデフォルト値が使用されます。

```
CALL SampleProcedure( V1 );
```

また、次のように、プロシージャは最初のパラメータには DEFAULT 値を使用して、2 番目のパラメータのみを指定することによっても呼び出せます。

```
CALL SampleProcedure( DEFAULT, V2 );
```

オプションの引数を使ってプロシージャを呼び出すよりも柔軟な方法は、パラメータに名前を付けて渡すという方法です。このとき、SampleProcedure プロシージャは次のように呼び出すことができます。

```
CALL SampleProcedure( var1 = V1, var3 = V3 );
```

または次のようになります。

```
CALL SampleProcedure( var3 = V3, var1 = V1 );
```

関数にパラメータを渡す方法

ユーザ定義関数は CALL 文で呼び出すのではなく、組み込み関数と同じように使用できます。たとえば、次の文は FullName 関数を使用して従業員の名前を取り出します。

注意

- デフォルトパラメータは呼び出し関数でも使用できます。ただしパラメータは、名前を付けて関数に渡すことはできません。
- パラメータは参照ではなく、値で渡されます。関数がそのパラメータの値を変更しても、その変更は関数を呼び出した環境には戻されません。
- ユーザ定義関数では出力パラメータは使用できません。
- ユーザ定義関数は結果セットを返すことはできません。

例：全従業員の名前のリスト

Interactive SQL で次のクエリを実行します。

```
SELECT FullName( GivenName, Surname ) AS Name
  FROM Employees;
```

次の結果が表示されます。

Name
Fran Whitney
Matthew Cobb
Philip Chin
Julie Jordan
...

参照

- 「ユーザ定義関数の作成」 89 ページ

制御文

プロシージャ、トリガ、またはユーザ定義関数の本文、またはバッチの中には、論理フローや意思決定のための制御文が複数あります。使用可能な制御文は、次のとおりです。

制御文	構文
複合文	BEGIN [ATOMIC] Statement-list END

制御文	構文

条件実行：IF	IF condition THEN
	  Statement-list
	ELSEIF condition THEN
	  Statement-list
	ELSE
	  Statement-list
	END IF

条件実行：CASE	CASE expression
	WHEN value THEN
	  Statement-list
	WHEN value THEN
	  Statement-list
	ELSE
	  Statement-list
	END CASE

繰り返し：WHILE、LOOP	WHILE condition LOOP Statement-list END LOOP

繰り返し：FOR カーソルループ	FOR loop-name
	  AS cursor-name CURSOR FOR
	  select-statement
	DO
	  Statement-list
	END FOR

中断：LEAVE	LEAVE label
CALL	CALL procname(arg, ...)

参照

- 「BEGIN 文」『SQL Anywhere サーバ SQL リファレンス』
- 「IF 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CASE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「LOOP 文」『SQL Anywhere サーバ SQL リファレンス』
- 「FOR 文」『SQL Anywhere サーバ SQL リファレンス』
- 「LEAVE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CALL 文」『SQL Anywhere サーバ SQL リファレンス』

複合文

複合文はキーワード BEGIN で始まり、キーワード END で終わります。プロシージャまたはトリガの本文は「複合文」です。また、バッチでも使うことができます。複合文はネストでき、他の制御文とともにプロシージャ、トリガ、またはバッチの実行フローを定義します。

複合文は、SQL 文のセットをまとめて 1 つの単位として扱えるようにします。複合文の中の SQL 文はセミコロンで区切ります。

複合文での宣言

複合文中のローカル宣言は、キーワード BEGIN のすぐ後に続きます。このローカル宣言は複合文中にのみ存在します。複合文に次のものを宣言できます。

- 変数
- カーソル
- テンポラリテーブル
- 例外処理 (エラー識別子)

ローカル宣言は、複合文またはその中でネストされる複合文の中のどの文からでも参照できます。ローカル宣言は、複合文中から呼び出された他のプロシージャからは見えません。

アトミックな複合文

「アトミック」な文とは、完全に実行されるか、まったく実行されない文のことです。たとえば、何千ものローを更新する UPDATE 文では、たくさんのローを更新した後にエラーが発生することがあります。文が完了しないと、変更されたすべてのローが元の状態に戻ります。したがって、UPDATE 文はアトミックです。

複合文でないすべての SQL 文はアトミックです。BEGIN キーワードの後にキーワード ATOMIC を追加して、複合文をアトミックにすることができます。

```
BEGIN ATOMIC
  UPDATE Employees
  SET ManagerID = 501
  WHERE EmployeeID = 467;
   UPDATE Employees
  SET BirthDate = 'bad_data';
END
```

この例の 2 つの UPDATE 文は、アトミックな複合文の一部です。これら 2 つの文は、1 つの文として更新を完了するか、両方ともエラーになります。最初の UPDATE 文はエラーなしで完了するとします。次の UPDATE 文は BirthDate カラムに割り当てた値を日付に変換できないため、エラーになります。

このアトミックな複合文はエラーになり、UPDATE 文の結果は両方とも取り消されます。現在実行中のトランザクションがコミットされても、この複合文中の文は両方ともその効果をもたらしません。

アトミックな複合文が成功すると、現在実行中のトランザクションがコミットされた場合のみ、複合文中で実行された変更は有効になります。アトミックな複合文が成功しても、その文で発生したトランザクションがロールバックされた場合は、アトミックな複合文もロールバックされます。アトミックな複合文の開始時に、セーブポイントが設定されます。文でエラーが発生すると、そのセーブポイントにロールバックされます。

アトミックな複合文がオートコミット (非連鎖) モードで実行されると、文の実行が完了するまでコミットモードが手動 (連鎖) に変更されます。手動モードでは、アトミックな複合文内で

DML 文を実行しても、即座に COMMIT や ROLLBACK は実行されません。アトミックな複合文が正常に完了すると、COMMIT 文が実行されます。正常に完了しない場合は ROLLBACK 文が実行されます。

COMMIT 文と ROLLBACK 文、および一部の ROLLBACK TO SAVEPOINT 文は、アトミックな複合文内で使用できません。

参照

- 「オートコミットの動作を制御する方法」『SQL Anywhere サーバ プログラミング』
- 「オートコミットまたは手動コミットモード」『SQL Anywhere サーバ プログラミング』
- 「プロシージャ、トリガ、ユーザ定義関数でのトランザクションとセーブポイント」138 ページ
- 「例外処理とアトミックな複合文」130 ページ

結果セット

プロシージャは結果を 1 つまたは複数のローとして返します。1 つのローのデータから構成される結果は、引数としてプロシージャに返すことができます。複数のローのデータから構成される結果は、結果セットとして返されます。また、プロシージャは RETURN 文の中で 1 つの値を返すこともできます。

参照

- 「プロシージャ」78 ページ

RETURN 文を使って値を返す

RETURN 文は、呼び出しを行った環境に 1 つの整数値を返した後、すぐにプロシージャを終了します。

前提条件

この作業を実行するための前提条件はありません。

♦ タスク

1. 次の文を実行します。

```
RETURN expression
```

2. 式の値が、呼び出しを行った環境に返されます。返ってきた値を変数に保存するには、CALL 文の拡張機能を使います。

```
CREATE VARIABLE returnval INTEGER;
returnval = CALL variable/procedure-name? myproc();
```

結果

値が返され、変数として保存されます。

参照

- 「RETURN 文」『SQL Anywhere サーバ SQL リファレンス』

結果をプロシージャのパラメータとして返す方法

プロシージャは、プロシージャのパラメータで呼び出しを行った環境に結果を返すことができます。次の文を使用して、プロシージャ内でパラメータと変数に値を割り当てることができます。

●SET 文

次に示すプロシージャは、SET 文を使用して OUT パラメータに値を割り当てて返します。次の文を実行するには、CREATE PROCEDURE システム権限を持っていることが必要です。

```
CREATE PROCEDURE greater(
  IN a INT,
  IN b INT,
  OUT c INT )
BEGIN
  IF a > b THEN
    SET c = a;
  ELSE
    SET c = b;
  END IF ;
END;
```

●INTO 句を持つ SELECT 文

「シングルロークエリ」がデータベースから取り出すローの数は多くても 1 つだけです。このタイプのクエリは SELECT 文に INTO 句を組み合わせて作成します。INTO 句は SELECT リストの後に続き、FROM 句より前に指定します。SELECT リストの各項目の値を受け取るための変数のリストが含まれます。変数は、SELECT リストの項目数と同じ数だけ用意します。

SELECT 文が実行されると、データベースサーバは SELECT 文の結果を取り出して、変数に入れます。クエリの結果、複数のローが取り出されると、データベースサーバはエラーを返します。複数のローを返すクエリにはカーソルを使用します。

クエリの結果、選択されたローが存在しない場合、変数は更新されず、警告が返されます。

SELECT 文を実行するには、オブジェクトに対する適切な SELECT 権限を持っている必要があります。

例 1：プロシージャを作成し、SELECT...INTO 文を使用してその結果を選択します。

1. Interactive SQL を起動して、SQL Anywhere サンプルデータベースに接続します。CREATE PROCEDURE システム権限、および Employee テーブルに対する SELECT 権限または SELECT ANY TABLE システム権限のいずれかを持っていることが必要です。

2. **[SQL 文]** ウィンドウ枠で、次の文を実行し、従業員の平均給与を OUT パラメータとして返すプロシージャ (AverageSalary) を作成します。

```
CREATE PROCEDURE AverageSalary( OUT average_salary NUMERIC(20,3) )
BEGIN
  SELECT AVG( Salary )
  INTO average_salary
  FROM GROUPO.Employees;
END;
```

3. プロシージャの結果を格納する変数を作成します。この場合、出力変数は小数点以下 3 桁の数値となります。

```
CREATE VARIABLE Average NUMERIC(20,3);
```

4. 作成した変数を使ってプロシージャを呼び出します。

```
CALL AverageSalary( Average );
```

5. プロシージャが正しく作成され、実行された場合、Interactive SQL の **[メッセージ]** タブにエラーは表示されません。

6. 次の文を実行して変数の値を検査します。

```
SELECT Average;
```

7. 出力変数 Average の値を見ます。**[結果]** ウィンドウ枠の **[結果]** タブに、この変数の値 49988.623 が表示されます。これが従業員の給与の平均値です。

例 2：シングルロー SELECT 文の結果を返します。

1. Interactive SQL を起動して、SQL Anywhere サンプルデータベースに接続します。CREATE PROCEDURE システム権限、および Customers テーブルに対する SELECT 権限または SELECT ANY TABLE システム権限のいずれかを持っていることが必要です。

2. 指定した顧客によって行われた発注の数を返すには、次の文を実行します。

```
CREATE PROCEDURE OrderCount(
  IN customer_ID INT,
  OUT Orders INT )
BEGIN
  SELECT COUNT(SalesOrders.ID)
    INTO Orders
  FROM GROUPO.Customers
    KEY LEFT OUTER JOIN SalesOrders
  WHERE Customers.ID = customer_ID;
END;
```

3. このプロシージャは、Interactive SQL で次の文を使ってテストします。次の文は ID が 102 の顧客からの注文の回数を返します。

```
CREATE VARIABLE orders INT;
CALL OrderCount ( 102, orders );
SELECT orders;
```

例 2 の注意

○customer_ID パラメータは IN パラメータとして宣言されます。このパラメータは顧客の ID をプロシージャに渡します。

○Orders パラメータは OUT パラメータとして宣言されます。これは変数 orders の値を呼び出し元の環境に返します。

○変数 Orders はプロシージャの引数リストで宣言されているので、DECLARE 文は必要ありません。

○SELECT 文は 1 つのローを返して、変数 Orders に入れます。

参照

- 「プロシージャからの結果セットで返される情報」118 ページ
- 「データベース接続」『SQL Anywhere サーバ データベース管理』

プロシージャからの結果セットで返される情報

プロシージャは、個別のパラメータとして呼び出しを行った環境に結果を返すだけでなく、結果セットとして情報を返すこともできます。通常、結果セットになるのはクエリの結果です。

RESULT 句の変数の数は、SELECT 文のリスト内の変数の数に一致しなければなりません。データ型が一致しない場合は、可能であれば自動的にデータ型の変換が行われます。

RESULT 句は CREATE PROCEDURE 文の一部であり、文デリミタは付きません。

SELECT 文のリスト内の変数の名前は、RESULT 句の変数の名前と一致する必要はありません。

ビューでプロシージャの結果セットを変更するには、基本となるテーブルに対する適切な権限がユーザに必要です。

ストアドプロシージャまたはユーザ定義関数が結果を返す場合、出力パラメータや戻り値もサポートされません。

デフォルトでは Interactive SQL は最初の結果セットのみを返します。Interactive SQL でプロシージャが結果の複数のローを返せるようにするには、**[オプション]** ウィンドウの **[結果]** タブで **[複数の結果セットを表示]** オプションを設定します。

例 1

次に示すプロシージャは、注文した顧客のリストと、注文の合計額を返します。

Interactive SQL で次の文を実行します。

```
CREATE PROCEDURE ListCustomerValue()
RESULT ( "Company" CHAR(36), "Value" INT )
BEGIN
  SELECT CompanyName,
    CAST( SUM(  SalesOrderItems.Quantity *
                Products.UnitPrice )
                AS INTEGER ) AS value
  FROM Customers
    INNER JOIN SalesOrders
    INNER JOIN SalesOrderItems
    INNER JOIN Products
  GROUP BY CompanyName
  ORDER BY value DESC;
END;
```

CALL ListCustomerValue ( ); を実行すると、次の結果セットが返されます。

Company	Value
The Hat Company	5016
The Igloo	3564
The Ultimate	3348
North Land Trading	3144
Molly's	2808
...	...

例 2

次に示すプロシージャは、ある部署の従業員 1 人 1 人の給与をセットにして返します。Interactive SQL で次の文を実行します。

```
CREATE PROCEDURE SalaryList( IN department_id INT )
RESULT ( "Employee ID" INT, Salary NUMERIC(20,3) )
BEGIN
  SELECT EmployeeID, Salary
   FROM Employees
   WHERE Employees.DepartmentID = department_id;
END;
```

RESULT 句での名前はクエリの結果と対応付けられ、表示される結果のカラムの見出しに使われます。

研究開発部 (部署 ID 100) の従業員の給与を表示するには、次の文を実行します。

```
CALL SalaryList( 100 );
```

[結果] ウィンドウ枠に次の結果セットが表示されます。

Employee ID	Salary
102	45700.000
105	62000.000
160	57490.000
243	72995.000
...	...

複数の結果セットを返す

Interactive SQL を使用して、プロシージャから複数の結果セットを返すことができます。

前提条件

この作業を実行するための前提条件はありません。

内容と備考

デフォルトでは、Interactive SQL では、複数の結果セットは表示されません。

♦ タスク

1. Interactive SQL で、データベースに接続します。
2. **[ツール]** » **[オプション]** をクリックします。
3. **[SQL Anywhere]** をクリックします。
4. **[結果]** タブで、**[すべての結果セットを表示]** をクリックします。
5. **[OK]** をクリックします。

結果

このオプションを有効にすると、Interactive SQL には複数の結果セットが表示されます。設定内容はすぐに反映され、無効にされるまで後続のセッションすべてについて有効になります。

次の手順

プロシージャ定義に含まれる RESULT 句を使う場合、結果セットはそれに合わせなければなりません。つまり、結果セットは SELECT 文のリストと同じ数の項目を持ち、データ型は RESULT 句にリストされたデータ型に自動的に変換されます。

RESULT 句を省略した場合は、プロシージャは返されるカラムの数や型がさまざまに異なる結果セットを返すことができます。

例

次の例は、すべての従業員、顧客、連絡先の名前をリストするプロシージャです。

```
CREATE PROCEDURE ListPeople()
RESULT ( Surname CHAR(36), GivenName CHAR(36) )
BEGIN
  SELECT Surname, GivenName
  FROM Employees;
  SELECT Surname, GivenName
  FROM Customers;
  SELECT Surname, GivenName
  FROM Contacts;
END;
```

参照

- 「プロシージャの変数結果セット」121 ページ
- 「isql_show_multiple_result_sets オプション [Interactive SQL]」『SQL Anywhere サーバ データベース管理』

プロシージャの変数結果セット

RESULT 句は、プロシージャでは省略可能です。RESULT 句を省略すると、実行方法に応じて、さまざまなカラム数またはカラム型の、異なる結果セットを返すプロシージャを記述できます。

変数結果セット機能を使用しない場合は、性能を高めるために RESULT 句を使用してください。

たとえば、次のプロシージャは、変数として Y を入力した場合は 2 カラムを、それ以外の場合は 1 カラムを返します。

```
CREATE PROCEDURE Names( IN formal char(1) )
BEGIN
  IF formal = 'y' THEN
    SELECT Surname, GivenName
    FROM Employees
  ELSE
    SELECT GivenName
    FROM Employees
  END IF
END;
```

クライアントアプリケーションで使用しているインタフェースによっては、プロシージャでの変数結果セットの使用に制限があります。

- **Embedded SQL**　正しい形式の結果セットを取得するには、結果セットのカーソルが開かれてからローが返されるまでの間に、プロシージャコールを記述 (DESCRIBE) します。

  RESULT 句を使用しないでプロシージャを作成し、そのプロシージャが変数結果セットを返す場合には、プロシージャを参照する SELECT 文の DESCRIBE が失敗することがあります。DESCRIBE の失敗を防ぐには、SELECT 文の FROM 句に WITH 句を含めることをおすすめします。または、DESCRIBE 文で WITH VARIABLE RESULT 句を使用できます。WITH VARIABLE RESUL 句は、各 OPEN 文の後にプロシージャ呼び出しを記述すべきかどうかを判断するのに使用できます。

- **ODBC**　変数結果セットプロシージャは ODBC アプリケーションで使用できます。SQL Anywhere ODBC ドライバは、変数結果セットを正しく記述します。

- **Open Client アプリケーション**　Open Client アプリケーションは、変数結果セットプロシージャを使用できます。SQL Anywhere は、変数結果セットを正しく記述します。

参照

- 「DESCRIBE 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』

プロシージャ、トリガ、ユーザ定義関数、バッチのカーソル

カーソルは、結果セットに複数のローがあるクエリまたはストアドプロシージャからローを 1 つずつ取り出します。カーソルは、クエリまたはプロシージャに対するハンドルまたは識別子で、結果セットの中の現在の位置を示します。

カーソル管理

カーソル管理はファイル管理に似ています。カーソル管理については、次の手順に従います。

1. DECLARE 文を使って、SELECT 文またはプロシージャにカーソルを宣言します。
2. OPEN 文を使って、カーソルを開きます。
3. FETCH 文を使って、カーソルから結果をローごとに取り出します。
4. ローが見つかりませんという警告が、結果セットの最後に表示されます。
5. CLOSE 文を使ってカーソルを閉じます。

デフォルトでは、カーソルはトランザクションの最後に、COMMIT または ROLLBACK によって自動的に閉じられます。WITH HOLD 句を使って開いたカーソルは、明示的に閉じるまで、トランザクション内で開いた状態になります。

参照

- 「カーソル位置」『SQL Anywhere サーバ プログラミング』

SELECT 文のカーソル

次に、SELECT 文でカーソルを使用するプロシージャの例を示します。この例では、ListCustomerValue プロシージャに使用するのと同じクエリをベースとして、ストアドプロシージャ言語の特徴を示します。

```
CREATE PROCEDURE TopCustomerValue(
    OUT TopCompany CHAR(36),
    OUT TopValue INT )
BEGIN
  -- 1. Declare the "row not found" exception
  DECLARE err_notfound
    EXCEPTION FOR SQLSTATE '02000';
  -- 2.  Declare variables to hold
  --    each company name and its value
  DECLARE ThisName CHAR(36);
  DECLARE ThisValue INT;
  -- 3.  Declare the cursor ThisCompany
  --     for the query
  DECLARE ThisCompany CURSOR FOR
  SELECT CompanyName,
      CAST( sum( SalesOrderItems.Quantity *
          Products.UnitPrice ) AS INTEGER )
      AS value
  FROM Customers
    INNER JOIN SalesOrders
    INNER JOIN SalesOrderItems
    INNER JOIN Products
  GROUP BY CompanyName;
  -- 4. Initialize the values of TopValue
  SET TopValue = 0;
  -- 5. Open the cursor
  OPEN ThisCompany;
  -- 6. Loop over the rows of the query
```

 Copyright © 2013, SAP AG or an SAP affiliate company.

```
  CompanyLoop:
  LOOP
    FETCH NEXT ThisCompany
      INTO ThisName, ThisValue;
    IF SQLSTATE = err_notfound THEN
      LEAVE CompanyLoop;
    END IF;
    IF ThisValue > TopValue THEN
      SET TopCompany = ThisName;
      SET TopValue = ThisValue;
    END IF;
  END LOOP CompanyLoop;
  -- 7. Close the cursor
  CLOSE ThisCompany;
END;
```

注意

この TopCustomerValue プロシージャには、次の特徴があります。

- 例外が宣言されます。この例外は、プロシージャの後でクエリの結果のループが完了するときに通知されます。
- クエリの各ローの結果を入れる 2 つのローカル変数 ThisName と ThisValue が宣言されます。
- カーソル ThisCompany が宣言されます。SELECT 文は会社名とその会社からの注文の合計額のリストを作成します。
- ループで使うため、TopValue の初期値は 0 に設定されています。
- ThisCompany カーソルが開きます。
- LOOP 文はクエリの各ローをループして、各会社の名前を変数 ThisName と ThisValue に入れます。ThisValue が現在の最大値よりも大きければ、TopCompany と TopValue は ThisName と ThisValue の値にリセットされます。
- プロシージャの最後にカーソルは閉じられます。
- SELECT 文に ORDER BY 値 DESC 句を追加して、ループを使わずにこのプロシージャを作成することもできます。その場合、カーソルの最初のローのみをフェッチする必要があります。

この TopCompanyValue プロシージャでの LOOP 文は標準的な形式で、最後のローを処理して終了します。FOR ループを使うと、このプロシージャはさらに簡潔になります。FOR 文は 1 つの文に、上記のプロシージャのいくつかの要素を組み込みます。

```
CREATE PROCEDURE TopCustomerValue2(
    OUT TopCompany CHAR(36),
    OUT TopValue INT )
BEGIN
  -- 1. Initialize the TopValue variable
  SET TopValue = 0;
  -- 2. Do the For Loop
  FOR CompanyFor AS ThisCompany
    CURSOR FOR
    SELECT CompanyName AS ThisName,
      CAST( sum( SalesOrderItems.Quantity *
          Products.UnitPrice ) AS INTEGER )
```

```
        AS ThisValue
      FROM Customers
        INNER JOIN SalesOrders
        INNER JOIN SalesOrderItems
        INNER JOIN Products
      GROUP BY ThisName
    DO
      IF ThisValue > TopValue THEN
        SET TopCompany = ThisName;
        SET TopValue = ThisValue;
      END IF;
    END FOR;
  END;
```

参照

- 「プロシージャからの結果セットで返される情報」118 ページ
- 「エラーと警告の処理」125 ページ
- 「ローが見つかりません。」『エラーメッセージ』

プロシージャ、トリガ、ユーザ定義関数、バッチ内の位置付け更新

次に、SELECT 文で更新可能なカーソルを使用するプロシージャの例を示します。次の例では、ストアドプロシージャ言語を使用して、ローに対して位置付け更新を実行する方法を示しています。

```
CREATE PROCEDURE UpdateSalary(
  IN employeeIdent INT,
  IN salaryIncrease NUMERIC(10,3) )
BEGIN
-- Procedure to increase (or decrease) an employee's salary
  DECLARE err_notfound
      EXCEPTION FOR SQLSTATE '02000';
  DECLARE oldSalary NUMERIC(20,3);
  DECLARE employeeCursor
    CURSOR FOR SELECT Salary from Employees
              WHERE EmployeeID = employeeIdent
    FOR UPDATE;
  OPEN employeeCursor;
  FETCH employeeCursor INTO oldSalary FOR UPDATE;
  IF SQLSTATE = err_notfound THEN
    MESSAGE 'No such employee' TO CLIENT;
  ELSE
    UPDATE Employees SET Salary = oldSalary + salaryIncrease
      WHERE CURRENT OF employeeCursor;
  END IF;
  CLOSE employeeCursor;
END;
```

上のストアドプロシージャを呼び出すには、次の文を入力してください。

```
CALL UpdateSalary( 105, 220.00 );
```

エラーと警告の処理

アプリケーションが SQL 文を実行した後、「ステータスコード」をチェックできます。ステータスコード (リターンコード) は文が正しく実行されたかどうかを表示して、エラーの場合はその理由を提示します。プロシージャを呼び出す CALL 文にも同じメカニズムが使われます。

エラーのレポートには、SQLCODE か SQLSTATE のどちらかのステータス表示を使用します。

SQL 文が実行されると、SQLCODE と SQLSTATE と呼ばれる特別なプロシージャ変数に値が入ります。この特別値は、文の実行中に変わった状況が発生したかどうかを示します。SQLCODE と SQLSTATE の値は、IF 文を SQL 文の後に置いてチェックできます。その結果によって適切な動作が行われます。

たとえば、SQLSTATE 変数はローが正しくフェッチされたかどうかを示すのに使用できます。TopCustomerValue プロシージャには、SELECT 文中のすべてのローが処理されたかどうかを検知するために SQLSTATE テストが使われています。

参照

- エラーメッセージ

デフォルトのエラー処理

この項では、プロシージャ内にエラー処理を指定しなかった場合に、SQL Anywhere がエラーを処理する方法を説明します。

さまざまな動作に例外ハンドラを使用できます。

警告の処理はエラーの処理とは少し異なります。

エラーを処理するには、特に指定しないかぎり次の 2 通りの方法があります。

- **デフォルトのエラー処理**　プロシージャかトリガがエラーを起こしたときに、呼び出しを行った環境にエラーコードが返されます。
- **ON EXCEPTION RESUME**　CREATE PROCEDURE 文に ON EXCEPTION RESUME 句が含まれていれば、プロシージャはエラーを起こした箇所の次の文から実行を再開します。

  ON EXCEPTION RESUME を使用するプロシージャの正確な動作は、on_tsql_error オプション設定によって指定します。

デフォルトのエラー処理

通常、プロシージャまたはトリガの SQL 文がエラーを起こすと、そのプロシージャまたはトリガは実行を停止し、SQLCODE と SQLSTATE に適切な値が入った状態でアプリケーションに制御が戻されます。これは最初の文がエラーを起こしたときも同じです。また、トリガの場合、トリガを起動した操作も取り消され、エラーがアプリケーションに返されます。

次の例のプロシージャは、アプリケーションからプロシージャ OuterProc を呼び出し、OuterProc が InnerProc を呼び出して、そこでエラーが発生した場合の処理を示します。

```
CREATE PROCEDURE OuterProc()
BEGIN
  MESSAGE 'Hello from OuterProc.' TO CLIENT;
  CALL InnerProc();
  MESSAGE 'SQLSTATE set to ',
    SQLSTATE,' in OuterProc.' TO CLIENT
END;
CREATE PROCEDURE InnerProc()
  BEGIN
    DECLARE column_not_found
      EXCEPTION FOR SQLSTATE '52003';
    MESSAGE 'Hello from InnerProc.' TO CLIENT;
    SIGNAL column_not_found;
  MESSAGE 'SQLSTATE set to ',
    SQLSTATE, ' in InnerProc.' TO CLIENT;
END;

CALL OuterProc();
```

Interactive SQL の [メッセージ] タブに次のメッセージが表示されます。

```
Hello from OuterProc.
Hello from InnerProc.
```

InnerProc 内の DECLARE 文は、サーバが認識しているエラー条件に関連して事前に定義された SQLSTATE 値のうち、1 つの記号名を宣言します。

MESSAGE 文は Interactive SQL の [メッセージ] タブにメッセージを送ります。

SIGNAL 文は InnerProc プロシージャ内から、エラーであることを外部に知らせる役割を持ちます。

InnerProc の SIGNAL 文の後の文は実行されず、InnerProc はすぐに呼び出しを行った環境 (この場合はプロシージャ OuterProc) に制御を戻します。OuterProc の CALL 文の後に続く文は実行されません。エラーは呼び出しを行った環境に戻され、処理されます。たとえば、Interactive SQL はエラーメッセージをメッセージウィンドウに表示してエラーの処理を行います。

TRACEBACK 関数はエラーが起きたときに実行していた文をリストします。Interactive SQL から TRACEBACK を使うには、次の文を入力します。

```
SELECT TRACEBACK();
```

参照

- 「例外ハンドラ」 129 ページ
- 「デフォルトの警告処理」 128 ページ
- 「on_tsql_error オプション」『SQL Anywhere サーバ データベース管理』

ON EXCEPTION RESUME を使ったエラー処理

ON EXCEPTION RESUME 文が CREATE PROCEDURE 文に含まれていた場合、エラーが起きると、次の文が検査されます。その文がエラーを処理する場合、プロシージャの実行が続行され、エラーが発生した文の次の文を実行します。エラーが発生したとき、呼び出しを行った環境に制御を戻しません。

on_tsql_error オプション設定を使用して、ON EXCEPTION RESUME を使用するプロシージャの動作を変更できます。

エラー処理文には、次のようなものがあります。

- IF
- SELECT @variable =
- CASE
- LOOP
- LEAVE
- CONTINUE
- CALL
- EXECUTE
- SIGNAL
- RESIGNAL
- DECLARE
- SET VARIABLE

次に示すプロシージャは、アプリケーションからプロシージャ OuterProc を呼び出し、OuterProc が InnerProc を呼び出して、そこでエラーが発生した場合の処理を示します。例文は、この項の最初で使用したプロシージャを基にしています。

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE OuterProc()
ON EXCEPTION RESUME
BEGIN
  DECLARE res CHAR(5);
  MESSAGE 'Hello from OuterProc.' TO CLIENT;
  CALL InnerProc();
  SET res=SQLSTATE;
  IF res='52003' THEN
    MESSAGE 'SQLSTATE set to ',
      res, ' in OuterProc.' TO CLIENT;
  END IF
END;

CREATE PROCEDURE InnerProc()
ON EXCEPTION RESUME
BEGIN
  DECLARE column_not_found
    EXCEPTION FOR SQLSTATE '52003';
  MESSAGE 'Hello from InnerProc.' TO CLIENT;
  SIGNAL column_not_found;
  MESSAGE 'SQLSTATE set to ',
  SQLSTATE, ' in InnerProc.' TO CLIENT;
END;

CALL OuterProc();
```

Interactive SQL の [メッセージ] タブに、次のメッセージが表示されます。

```
Hello from OuterProc.
Hello from InnerProc.
SQLSTATE set to 52003 in OuterProc.
```

実行パスを次に示します。

1. OuterProc は InnerProc を実行して呼び出します。
2. InnerProc では、SIGNAL 文がエラーを通知します。
3. MESSAGE 文はエラー処理文ではないので、制御は OuterProc に返され、メッセージは表示されません。
4. OuterProc では、エラーに続く文が SQLSTATE の値を **res** という変数に割り当てます。これはエラー処理文なので、実行は継続され、OuterProc メッセージが表示されます。

参照

- 「on_tsql_error オプション」『SQL Anywhere サーバ データベース管理』

デフォルトの警告処理

エラーと警告の処理方法は異なります。デフォルトのエラー処理は、SQLSTATE と SQLCODE に値を入れてエラー発生時の呼び出しを行った環境に制御を戻しますが、警告処理のデフォルトは、SQLSTATE と SQLCODE に値を入れてプロシージャの実行を続けます。

次に示す例は、デフォルトの警告処理を示します。

この場合、ローが見つからないことを示す条件が SIGNAL 文によって生成されます。これはエラーではなく警告です。

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE OuterProc()
BEGIN
  MESSAGE 'Hello from OuterProc.' TO CLIENT;
  CALL InnerProc();
  MESSAGE 'SQLSTATE set to ',
    SQLSTATE,' in OuterProc.' TO CLIENT;
END;
CREATE PROCEDURE InnerProc()
BEGIN
  DECLARE row_not_found
    EXCEPTION FOR SQLSTATE '02000';
  MESSAGE 'Hello from InnerProc.' TO CLIENT;
  SIGNAL row_not_found;
  MESSAGE 'SQLSTATE set to ',
  SQLSTATE, ' in InnerProc.' TO CLIENT;
END;

CALL OuterProc();
```

Interactive SQL の [メッセージ] タブに、次のメッセージが表示されます。

```
Hello from OuterProc.
Hello from InnerProc.
SQLSTATE set to 02000 in InnerProc.
SQLSTATE set to 00000 in OuterProc.
```

両方のプロシージャとも、警告によって SQLSTATE に値 (02000) が設定された後も実行を続けました。

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

InnerProc で 2 番目の MESSAGE 文を実行すると、警告がリセットされます。SQL 文は、SQLSTATE を 00000、SQLCODE を 0 にリセットします。プロシージャがエラー状態を保存する必要がある場合、エラーまたは警告の原因となった文の実行直後に値を割り当てる必要があります。

参照

- 「デフォルトのエラー処理」125 ページ
- 「ローが見つかりません。」『エラーメッセージ』

例外ハンドラ

エラーは呼び出しを行った環境へ戻すよりも、プロシージャまたはトリガの内部で捕捉して処理した方が良い場合があります。これは「例外ハンドラ」を使用して行います。

例外ハンドラは、複合文の EXCEPTION 部分で定義します。

複合文でエラーが起きた場合、例外ハンドラが実行されます。警告では、例外ハンドラは実行されません。ネストされた複合文の中でエラーが起きた場合、また、複合文の中から起動されたプロシージャやトリガの中でエラーが起きた場合は、例外処理コードが実行されます。

中断エラー SQL_INTERRUPT、SQLSTATE 57014 の例外ハンドラには、ROLLBACK や ROLLBACK TO SAVEPOINT などの中断のできない文だけを含めます。例外ハンドラに、接続の中断時に呼び出される中断可能な文を含めると、データベースサーバは最初の中断可能な文で例外ハンドラを停止し、中断エラーを返します。

例外ハンドラでは、文が失敗した原因を判断するために、SQLSTATE または SQLCODE 特別値を使用できます。代わりに、ERRORMSG 関数を引数なしで使用すると、SQLSTATE に関連付けられたエラー条件が返されます。この情報を指定できるのは、各 WHEN 句の最初の文のみであり、文は複合文にはできません。

次に示す例では、「デフォルトのエラー処理」125 ページで使用したプロシージャを基にして、例外ハンドラの処理を示します。

この例では、InnerProc プロシージャで見つからないカラムに関するエラーを処理するために、コードが追加されています。

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE OuterProc()
BEGIN
  MESSAGE 'Hello from OuterProc.' TO CLIENT;
  CALL InnerProc();
  MESSAGE 'SQLSTATE set to ',
    SQLSTATE,' in OuterProc.' TO CLIENT
END;
CREATE PROCEDURE InnerProc()
BEGIN
  DECLARE column_not_found
   EXCEPTION FOR SQLSTATE '52003';
  MESSAGE 'Hello from InnerProc.' TO CLIENT;
 SIGNAL column_not_found;
```

```
  MESSAGE 'Line following SIGNAL.' TO CLIENT;
  EXCEPTION
    WHEN column_not_found THEN
      MESSAGE 'Column not found handling.' TO CLIENT;
    WHEN OTHERS THEN
      RESIGNAL ;
END;

CALL OuterProc();
```

Interactive SQL の **[メッセージ]** タブに、次のメッセージが表示されます。

```
Hello from OuterProc.
Hello from InnerProc.
Column not found handling.
SQLSTATE set to 00000 in OuterProc.
```

EXCEPTION 句は例外ハンドラを宣言します。これ以降の文はエラーが起きないかぎり実行されません。WHEN 句は例外名 (DECLARE 文で宣言) と、その例外が起こったときに実行する文を定義します。WHEN OTHERS THEN 句はその前の WHEN 句以外で例外が起こったときに実行する文を定義します。

上の例では、RESIGNAL は例外を上位レベルの例外ハンドラに渡します。WHEN OTHERS THEN が例外ハンドラ中に定義されていない場合は、RESIGNAL がデフォルト処理になります。

追加の注意事項

- InnerProc の SIGNAL 文に続く行ではなく、EXCEPTION ハンドラが実行されます。
- 見つからないカラムに関するエラーが発生したため、エラー処理のための MESSAGE 文が実行され、SQLSTATE は 0 にリセットされます (エラーがないことを示します)。
- 例外処理コードが実行された後、制御は OuterProc に戻され、OuterProc はエラーがなかったかのように前へ進みます。
- ON EXCEPTION RESUME は指定した例外処理とは一緒に使えません。ON EXCEPTION RESUME が含まれていると、例外処理コードは実行されません。
- エラーに対するエラー処理コードが RESIGNAL 文である場合、制御は OuterProc プロシージャに戻され、SQLSTATE の値は 52003 に設定されたままになります。これは、InnerProc にはエラー処理コードがないのと同じです。OuterProc にはこれ以外のエラー処理コードはないため、プロシージャはエラーになります。

例外処理とアトミックな複合文

アトミック複合内でエラーが発生し、その文にエラー処理用の例外ハンドラが実装されている場合は、複合文はアクティブな例外なしで完了し、例外より前の変更は取り消されません。例外ハンドラが発生したエラーを処理しない場合、または別のエラー (RESIGNAL によるエラーを含む) を発生させた場合は、そのアトミックな複合文で行われた変更は取り消されます。

参照

- 「複合文」113 ページ
- 「カラム '%1' が見つかりません。」『エラーメッセージ』
- 「SQLCODE 特別値」『SQL Anywhere サーバ SQL リファレンス』
- 「SQLSTATE 特別値」『SQL Anywhere サーバ SQL リファレンス』
- 「ERRORMSG 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「RESIGNAL 文 [SP]」『SQL Anywhere サーバ SQL リファレンス』
- 「TRY 文」『SQL Anywhere サーバ SQL リファレンス』

ネストされた複合文と例外処理

エラーを引き起こした文に続くコードは、プロシージャ定義に ON EXCEPTION RESUME 句が含まれる場合のみ、実行されます。

ネストされた複合文を使用すると、エラーの後にどの文が実行され、どの文が実行されないのかを制御できます。

次の例は、ネストされた複合文をどのように使用してフローを制御するかを示します。

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE InnerProc()
BEGIN
  BEGIN
    DECLARE column_not_found
    EXCEPTION FOR SQLSTATE VALUE '52003';
    MESSAGE 'Hello from InnerProc' TO CLIENT;
    SIGNAL column_not_found;
    MESSAGE 'Line following SIGNAL' TO CLIENT
    EXCEPTION
      WHEN column_not_found THEN
        MESSAGE 'Column not found handling' TO
        CLIENT;
      WHEN OTHERS THEN
        RESIGNAL;
  END;
  MESSAGE 'Outer compound statement' TO CLIENT;
END;

CALL InnerProc();
```

Interactive SQL の **[メッセージ]** タブに、次のメッセージが表示されます。

```
Hello from InnerProc
Column not found handling
Outer compound statement
```

エラーを引き起こした SIGNAL 文が検出されると、制御は複合文の例外ハンドラに渡されて、「**カラムが見つかりません。**」というメッセージが出力されます。次に制御は外部複合文に渡され、「Outer compound statement」メッセージが出力されます。

内部複合文でカラムが見つかりません。(SQLSTATE) 以外のエラーが検出されると、例外ハンドラは RESIGNAL 文を実行します。RESIGNAL 文は、呼び出しを行った環境に制御を直接戻します。外部複合文の残りの文は実行されません。

例

この例は、RESIGNAL による sa_error_stack_trace システムプロシージャの出力を示しています。

```
CREATE PROCEDURE DBA.error_reporting_procedure()
BEGIN
      SELECT *
      FROM sa_error_stack_trace();
END;

CREATE PROCEDURE DBA.proc1()
BEGIN TRY
   BEGIN TRY
      DECLARE v INTEGER = 0;
      SET v = 1 / v;
   END TRY
   BEGIN CATCH
      CALL DBA.proc2();
   END CATCH
END TRY
BEGIN CATCH
   CALL DBA.error_reporting_procedure();
END CATCH;

CREATE PROCEDURE DBA.proc2()
BEGIN
   CALL DBA.proc3();
END;

CREATE PROCEDURE DBA.proc3()
BEGIN
   RESIGNAL;
END;
```

CALL proc1() を使用して上記のプロシージャが呼び出されると、次の結果セットが作成されます。

StackLevel	UserName	ProcName	LineNumber	IsResignal
1	DBA	proc1	8	0
2	DBA	proc2	3	0
3	DBA	proc3	3	1
4	DBA	proc1	5	0

この例は、RESIGNAL と BEGIN 文による sa_error_stack_trace システムプロシージャの出力を示しています。

```
CREATE PROCEDURE DBA.error_reporting_procedure()
BEGIN
      SELECT *
      FROM sa_error_stack_trace();
END;

CREATE PROCEDURE DBA.proc1()
BEGIN
   BEGIN
```

```
        DECLARE v INTEGER = 0;
        SET v = 1 / v;
    EXCEPTION WHEN OTHERS THEN
        CALL DBA.proc2();
    END
END
EXCEPTION WHEN OTHERS THEN
    CALL DBA.error_reporting_procedure();
END;

CREATE PROCEDURE DBA.proc2()
BEGIN
    CALL DBA.proc3();
END;

CREATE PROCEDURE DBA.proc3()
BEGIN
    RESIGNAL;
END;
```

CALL proc1() を使用して上記のプロシージャが呼び出されると、次の結果セットが作成されます。

StackLevel	UserName	ProcName	LineNumber	IsResignal
1	DBA	proc1	8	0
2	DBA	proc2	3	0
3	DBA	proc3	3	1
4	DBA	proc1	5	0

参照

- 「TRY 文」『SQL Anywhere サーバ SQL リファレンス』
- 「BEGIN 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ERROR_LINE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「ERROR_MESSAGE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「ERROR_PROCEDURE 関数 [関数タイプ]」『SQL Anywhere サーバ SQL リファレンス』
- 「ERROR_SQLCODE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「ERROR_SQLSTATE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「ERROR_STACK_TRACE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「STACK_TRACE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_error_stack_trace システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_stack_trace システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「ネストされた複合文と例外処理」131 ページ

例：例外ハンドラによって呼び出し可能なエラーログプロシージャの作成

例外ハンドラでのアプリケーション間で統一されたエラーログに使用できる、エラーログプロシージャを定義できます。

1. 次のテーブルを作成して、エラーログプロシージャが実行されるたびにエラー情報のログを作成します。

```
CREATE TABLE IF NOT EXISTS error_info_table (
  idx INTEGER,
  ln UNSIGNED INTEGER,
  code INTEGER,
  state CHAR(5),
  err_msg CHAR(256),
  name CHAR(257),
  err_stack LONG VARCHAR,
  traceback LONG VARCHAR
);

CREATE TABLE IF NOT EXISTS error_stack_trace_table (
  idx UNSIGNED SMALLINT NOT NULL,
  stack_level UNSIGNED SMALLINT NOT NULL,
  user_name VARCHAR(128),
  proc_name VARCHAR(128),
  line_number UNSIGNED INTEGER NOT NULL,
  is_resignal BIT NOT NULL, PRIMARY KEY (idx, stack_level)
);
```

2. error_info_table と error_stack_trace_table にエラー情報のログを作成して、データベースサーバメッセージウィンドウにメッセージを書き込む、次のプロシージャを作成します。

```
CREATE OR REPLACE PROCEDURE error_report_proc ( IN location_indicator INTEGER )
NO RESULT SET
BEGIN
  INSERT INTO error_info_table VALUES (
    location_indicator,
    ERROR_LINE(),
    ERROR_SQLCODE(),
    ERROR_SQLSTATE(),
    ERROR_MESSAGE(),
    ERROR_PROCEDURE(),
    ERROR_STACK_TRACE(),
    TRACEBACK()
  );
  INSERT INTO error_stack_trace_table
    SELECT location_indicator, *
    FROM sa_error_stack_trace() ;
  MESSAGE 'The error message is '|| ERROR_MESSAGE() ||' and the stack trace is '||
ERROR_STACK_TRACE()
  TYPE WARNING TO CONSOLE ;
END;
```

3. 次のようなプロシージャを作成して、例外ハンドラからエラーログプロシージャを起動します。

```
CREATE OR REPLACE PROCEDURE MyProc()
BEGIN
  DECLARE column_not_found
```

```
    EXCEPTION FOR SQLSTATE '52003';
  MESSAGE 'Hello from MyProc.' TO CLIENT;
  SIGNAL column_not_found;
  MESSAGE 'Line following SIGNAL.' TO CLIENT;
EXCEPTION
WHEN column_not_found THEN
  MESSAGE 'Column not found handling.' TO CLIENT;
  CALL error_report_proc();
END ;
```

参照

- 「例外ハンドラ」 129 ページ
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「MESSAGE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「TRY 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ERROR_LINE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「ERROR_SQLCODE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「ERROR_SQLSTATE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「ERROR_MESSAGE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「ERROR_PROCEDURE 関数 [関数タイプ]」『SQL Anywhere サーバ SQL リファレンス』
- 「ERROR_STACK_TRACE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』

プロシージャ、トリガ、ユーザ定義関数、バッチで使用される EXECUTE IMMEDIATE

EXECUTE IMMEDIATE 文を使うと、文字列 (引用符で囲む) と変数を使って文を組み立てることができます。次に示すのは、テーブルを作成する EXECUTE IMMEDIATE 文を含むプロシージャの例です。

```
CREATE PROCEDURE CreateTableProcedure(
    IN tablename CHAR(128) )
BEGIN
  EXECUTE IMMEDIATE 'CREATE TABLE '
  || tablename
  || '( column1 INT PRIMARY KEY )'
END;
```

EXECUTE IMMEDIATE 文は、結果セットを返すクエリで使用できます。文が結果セットを返すように指定するために、EXECUTE IMMEDIATE 文で WITH RESULT SET ON 句を使用します。デフォルトの動作では、文は結果セットを返しません。WITH RESULT SET ON または WITH RESULT SET OFF を指定することは、プロシージャが作成されるときだけではなく、プロシージャが実行されるときの動作についても影響をもたらすことになります。

次のプロシージャを考えてみます。

```
CREATE OR REPLACE PROCEDURE test_result_clause()
BEGIN
   EXECUTE IMMEDIATE WITH RESULT SET OFF 'SELECT 1';
END;
```

プロシージャ定義に RESULT SET 句が含まれていないと、データベースサーバ側ではプロシージャによって結果セットが生成されるのかどうかを判断しようとします。ここでは、EXECUTE IMMEDIATE 文では、結果セットを生成しないように設定されています。そのため、データベースサーバ側では、プロシージャには結果セットのカラムがないと判断されるため、このプロシージャの SYSPROCPARM システムビューにはカラムが表示されません。このプロシージャの呼び出しに DESCRIBE を指定しても、結果カラムは返されません。カーソルを開いたり、文を実行するかどうかを決定するために、Embedded SQL アプリケーションによってこの情報が使用された場合には、文が実行されて、エラーが返されます。

2 番目の例では、上記のプロシージャの修正バージョンを考えてみます。

```
CREATE OR REPLACE PROCEDURE test_result_clause()
BEGIN
   EXECUTE IMMEDIATE WITH RESULT SET ON 'SELECT 1';
END;
```

ここでは、WITH RESULT SET ON 句が指定されているため、SYSPROCPARM システムビューにはこのプロシージャのローが存在しています。プロシージャが EXECUTE IMMEDIATE を使用しているため、データベースサーバ側では、実際に結果セットがどのようになるかついては把握できません。ただし、結果セットが存在することは予測できるため、データベースサーバは SYSPROCPARM に、expression という名前の SMALLINT 型のダミーの結果セットのカラムを定義することによって、結果セットに対応していることを示します。作成される結果セット用ダミーカラムが **1 つ** だけです。サーバ側では、EXECUTE IMMEDIATE 文が使用されているときに、各結果セットのカラム数や型については判断できません。そのため、さらに少し修正したバージョンの例を考えてみます。

```
CREATE OR REPLACE PROCEDURE test_result_clause()
BEGIN
   EXECUTE IMMEDIATE WITH RESULT SET ON 'SELECT 1, 2, 3';
END;
```

ここでは、SELECT 文は 3 つのカラムの結果セットを返しますが、サーバの SYSPROCPARM システムビューには、1 つのローがあるだけです。そこで、次のクエリを見てください。

```
SELECT * FROM test_result_clause();
```

このクエリは、SQLCODE -866 エラーになります。これは、実行時の結果セットの定義が SYSPROCPARM に事前に用意された定義と一致しないためです。

上記のクエリを実行するには、次に示すように、結果セットのカラムの名前と型を明示的に指定します。

```
SELECT * FROM test_result_clause() WITH (x INTEGER, y INTEGER, z INTEGER);
```

WITH RESULT SET ON が指定されている場合は、実行時に、データベースサーバは EXECUTE IMMEDIATE 文を処理し、結果セットを返します。ただし、WITH RESULT SET OFF が指定されているか、または句が省略されている場合でも、データベースサーバは解析された文字列引数の最初の文の型については、**同じように**認識することができます。その文が SELECT 文であれば、結果セットを返します。それでは、前述した 2 番目の例を見てください。

```
CREATE OR REPLACE PROCEDURE test_result_clause()
BEGIN
   EXECUTE IMMEDIATE WITH RESULT SET OFF 'SELECT 1';
END;
```

Interactive SQL からこのプロシージャを呼び出すとエラーにはなりません。ここで、1 つの SELECT 文ではなく、バッチが含まれたプロシージャに変更するとします。

```
CREATE OR REPLACE PROCEDURE test_result_clause()
BEGIN
   EXECUTE IMMEDIATE WITH RESULT SET OFF
   'begin declare v int; set v=1; select v; end';
END;
```

この場合には、test_result_clause プロシージャを呼び出すと、エラー (SQLCODE -946, SQLSTATE 09W03) が発生します。

次の最後の例では、プロシージャ内で SELECT 文を EXECUTE IMMEDIATE 文の引数として構築して、プロシージャが結果セットを返すようにする方法を示しています。

```
CREATE PROCEDURE DynamicResult(
  IN Columns LONG VARCHAR,
  IN TableName CHAR(128),
  IN Restriction LONG VARCHAR DEFAULT NULL )
BEGIN
   DECLARE Command LONG VARCHAR;
   SET Command = 'SELECT ' || Columns || ' FROM ' || TableName;
   IF ISNULL( Restriction,'') <> '' THEN
       SET Command = Command || ' WHERE ' || Restriction;
   END IF;
   EXECUTE IMMEDIATE WITH RESULT SET ON Command;
END;
```

このプロシージャは次のように呼び出されます。

```
CALL DynamicResult(
  'table_id,table_name',
  'SYSTAB',
  'table_id <= 10');
```

これは次のような結果になります。

table_id	table_name
1	ISYSTAB
2	ISYSTABCOL
3	ISYSIDX
...	...

上記の CALL では、プロシージャで EXECUTE IMMEDIATE が使用されていても、正しく結果セットが返されます。ODBC などの一部のサーバ API では、PREPARE-DESCRIBE-EXECUTE-OR-OPEN を利用して要求が結合されており、結果セットを返すかどうかによって、文を実行するか、または開くかが選択されます。文が開かれる必要がある場合には、API またはアプリケーションは DESCRIBE CURSOR を続けて発行することによって、プロシージャが作成されたときに構築された SYSPROCPARM システムビューの内容に依存するのではなく、実際の結果セットを把握できるようにします。この方法は、DBISQL でも DBISQLC でも使用されています。この場合、上記のプロシージャの CALL はエラーにならずに実行されます。ただし、文の DESCRIBE

の結果に依存するアプリケーションインタフェースの場合は、どの文でも処理できるというわけではありません。

アトミックな複合文中では、COMMIT を行う EXECUTE IMMEDIATE 文は使えません。COMMIT 文はこのコンテキストでは許可されていません。

参照

- 「EXECUTE IMMEDIATE 文 [SP]」『SQL Anywhere サーバ SQL リファレンス』
- 「%1 から返された結果セットのスキーマは予期されているものと異なります。」『エラーメッセージ』
- 「'%1' では結果セットは許可されません。」『エラーメッセージ』

プロシージャ、トリガ、ユーザ定義関数でのトランザクションとセーブポイント

プロシージャまたはトリガ内の SQL 文は現在のトランザクションの一部です。1 つのトランザクション内で複数のプロシージャを呼び出すことや、1 つのプロシージャ内に複数のトランザクションを持つことができます。

アトミックな文中では COMMIT と ROLLBACK は許可されません。トリガはアトミックな文である INSERT、UPDATE、DELETE によって起動さます。COMMIT と ROLLBACK はトリガまたはトリガから呼び出されたプロシージャ内では許可されません。

プロシージャまたはトリガではセーブポイントを使用できますが、ROLLBACK TO SAVEPOINT 文はアトミックオペレーションが開始される以前のセーブポイントを参照することはできません。また、アトミックオペレーション内のすべてのセーブポイントは、その操作が終了したときに解除されます。

参照

- 「トランザクションと独立性レベル」861 ページ
- 「アトミックな複合文」114 ページ
- 「トランザクション内のセーブポイント」864 ページ

プロシージャ、トリガ、ユーザ定義関数、バッチを作成するときのヒント

この項では、プロシージャ、トリガ、ユーザ定義関数、バッチを作成するときのヒントについて説明します。

SQL 文デリミタを変更する必要性のチェック

プロシージャを書き込むときに、文デリミタを変更する必要はありません。他のブラウズツールを使う場合には、文デリミタをセミコロンから他の文字に変更する必要がある場合があります。

プロシージャ内の各文はセミコロンで終わります。ブラウズするアプリケーションが CREATE PROCEDURE 文自体を解析するには、文デリミタにセミコロン以外の文字を使用する必要があります。

文デリミタを変更する必要があるアプリケーションを使用する場合は、文デリミタとして 2 つのセミコロン (;;) を使うか、複数の文字を使ったデリミタが許可されないシステムであれば疑問符 (?) が適切です。

プロシージャの中で文を区切る

プロシージャ内の各文はセミコロンで終わります。最後の文はセミコロンがなくてもかまいませんが、各文の後ろにはセミコロンを付けることを習慣にしてください。

CREATE PROCEDURE 文は本体である複合文と、RESULT 指定の両方を含みます。キーワード BEGIN または END の後と、RESULT 句の後には、セミコロンは必要ありません。

プロシージャ内のテーブル名

プロシージャがテーブルを参照する場合は、テーブル名に必ず所有者 (作成者) の名前をプレフィクスとして付けてください。

プロシージャがテーブルを参照するときは、プロシージャ作成者のロールメンバーシップを使い、所有者の名前は指定しません。たとえば、user_1 が作成したプロシージャが Table_B を参照し、Table_B の所有者の名前を指定しないとします。この場合、Table_B が user_1 によって作成されたか、user_1 が Table_B の所有者であるロールの (直接的または間接的な) メンバーでなければなりません。どちらの条件も満たされない場合は、プロシージャが呼び出されると、「**テーブルが見つかりません。**」というメッセージが表示されます。

テーブルの指定に相関名を使うと、テーブルの長い、完全に修飾された名前を入力しないですみます。相関名の詳細については、「FROM 句」『SQL Anywhere サーバ SQL リファレンス』を参照してください。

プロシージャの中で日付と時刻を指定する

プロシージャは、日付と時刻を文字列としてデータベースに送ります。この文字列は date_order データベースオプションの現在の設定に従って変換されます。異なる接続はこのオプションを異なる値に設定することもあるので、文字列が間違った日付に変換されたり、まったく変換できなかったりすることがあります。

プロシージャで日付文字列を使用するときには、あいまいでない *yyyy-mm-dd* か *yyyy/mm/dd* の日付形式を使用します。date_order データベースオプションの設定に関係なく、サーバはこれらの文字列を日付として正しく解釈します。

プロシージャの入力引数が正しく渡されていることを検証する

入力引数を検証する 1 つの方法は、MESSAGE 文を使って Interactive SQL の **[メッセージ]** タブにパラメータ値を表示することです。たとえば、次に示すプロシージャは、入力パラメータの **var** の値を表示します。

```
CREATE PROCEDURE message_test( IN var char(40) )
BEGIN
  MESSAGE var TO CLIENT;
END;
```

デバッガを使用して、プロシージャの入力引数が正常に渡されたことを確認することもできます。

参照

- 「command_delimiter オプション [Interactive SQL]」『SQL Anywhere サーバ データベース管理』
- 「日付と時刻データ型」『SQL Anywhere サーバ SQL リファレンス』
- 「レッスン 2：バグの診断」951 ページ

プロシージャ、トリガ、イベント、バッチで使用できる文

バッチにはほとんどの SQL 文を使用できますが、次の文は使用できません。

- ALTER DATABASE (構文 3 および 4)
- CONNECT
- CREATE DATABASE
- CREATE DECRYPTED FILE
- CREATE ENCRYPTED FILE
- DISCONNECT
- DROP CONNECTION
- DROP DATABASE
- FORWARD TO
- INPUT、OUTPUT などの Interactive SQL 文
- PREPARE TO COMMIT
- STOP SERVER

COMMIT、ROLLBACK、SAVEPOINT 文はプロシージャ、トリガ、バッチで使用できますが、若干の制限があります。

参照

- 「SQL 文」『SQL Anywhere サーバ SQL リファレンス』
- 「プロシージャ、トリガ、ユーザ定義関数でのトランザクションとセーブポイント」138 ページ

バッチで使用される SELECT 文

バッチには 1 つまたは複数の SELECT 文を含めることができます。次に例を示します。

```
IF EXISTS( SELECT *
           FROM SYSTAB
           WHERE table_name='Employees' )
THEN
  SELECT   Surname AS LastName,
           GivenName AS FirstName
  FROM Employees;
  SELECT Surname, GivenName
  FROM Customers;
  SELECT Surname, GivenName
```

```
  FROM Contacts;
END IF;
```

結果セットのエイリアスは、最初の SELECT 文でのみ必要です。サーバはバッチ中の最初の SELECT 文を結果セットの記述に使用するからです。

各クエリの後には、次の結果セットを取り出すための RESUME 文が必要です。

プロシージャ、ファンクション、トリガ、イベント、またはビューの内容を隠す

プロシージャ、ファンクション、トリガ、およびビュー内に含まれるロジックを公開しないで、アプリケーションとデータベースを配布する場合は、ALTER PROCEDURE、ALTER FUNCTION、ALTER TRIGGER、ALTER EVENT、ALTER VIEW 文の SET HIDDEN 句を使用して、これらのオブジェクトの内容を隠すことができます。

前提条件

オブジェクトの所有者であるか、ALTER ANY OBJECT システム権限を持っているか、次の権限の 1 つを持っていることが必要です。

プロシージャとファンクション　ALTER ANY PROCEDURE システム権限

ビュー　ALTER ANY VIEW システム権限

イベント　MANAGE ANY EVENT システム権限

トリガ

- ALTER ANY TRIGGER システム権限
- 基礎となるテーブルに対する ALTER 権限と CREATE ANY OBJECT システム権限
- ビューに対するトリガについては、ALTER ANY TRIGGER と ALTER ANY VIEW のシステム権限を持っていることが必要です。

内容と備考

SET HIDDEN 句は、関連オブジェクトを使用可能な状態に保ちながら、その内容を難読化して読み取れないようにします。また、アンロードして、別のデータベースに再ロードすることもできます。

修正を元に戻すことはできません。修正すると、オブジェクトの元のテキストが削除されます。オブジェクトの元のソースをデータベースの外部に保存しておく必要があります。

デバッガによるデバッグでは、プロシージャ定義が表示されず、プロシージャプロファイリングにもソースが表示されません。

注意
preserve_source_format データベースオプションを On に設定すると、データベースサーバは、プロシージャ、ビュー、トリガ、イベントの CREATE 文と ALTER 文によってフォーマットされたソースを保存し、それを適切なシステムビューのソースカラムに配置します。この場合、オブジェクト定義とソース定義の両方が隠されます。

ただし、preserve_source_format データベースオプションを On に設定しても、SET HIDDEN 句でオブジェクトの元のソース定義を削除できます。

♦ タスク

- SET HIDDEN 句を持つ適切な ALTER 文を使用します。

オプション	アクション
個別のオブジェクトを隠す	1 つのプロシージャ、ファンクション、トリガ、イベント、またはビューを隠すには、SET HIDDEN 句を持つ適切な ALTER 文を実行します。
特定のタイプのすべてのオブジェクトを隠す	すべてのプロシージャ、ファンクション、トリガ、イベント、またはビューを隠すには、SET HIDDEN 句を持つ適切な ALTER 文をループで実行します。

結果

オートコミットが実行されます。オブジェクト定義は表示されません。ただし、オブジェクトは直接参照でき、クエリ処理中に使用できることは変わりません。

例

すべてのプロシージャを隠すには、次のループを実行します。

```
BEGIN
   FOR hide_lp as hide_cr cursor FOR
      SELECT proc_name, user_name
      FROM SYS.SYSPROCEDURE p, SYS.SYSUSER u
      WHERE p.creator = u.user_id
      AND p.creator NOT IN (0,1,3)
   DO
      MESSAGE 'altering ' || proc_name;
      EXECUTE IMMEDIATE 'ALTER PROCEDURE "' ||
         user_name || '"."' || proc_name
         || '" SET HIDDEN'
   END FOR
END;
```

参照

- 「ALTER FUNCTION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TRIGGER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER EVENT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「preserve_source_format オプション」『SQL Anywhere サーバ データベース管理』

パフォーマンスの向上、診断、モニタリング

この項では、パフォーマンスの向上、パフォーマンスに関する問題の診断、パフォーマンスのモニタリングの方法について説明します。

パフォーマンスのモニタリングと診断のツール

この項では、SQL Anywhere 分析ツールを使用したデータベースパフォーマンスの分析および調整について説明します。

SQL Anywhere には、運用データベースのパフォーマンス上の問題を検出するための診断ツールがいくつかあります。これらのツールの一部は「診断トレーシング」インフラストラクチャに依存します。このインフラストラクチャは、診断データを取得、格納するテーブル、ファイルなどのコンポーネントから構成されるシステムです。診断トレーシングのデータを使用して、「アプリケーションプロファイリング」などの診断やモニタリングのタスクを行うことができます。

SQL Anywhere のパフォーマンスデータを分析するには、次のような方法があります。

ツール	詳細
診断トレーシングツール	● **データベーストレーシングウィザード**　このウィザードでは、収集するパフォーマンスデータのタイプをカスタマイズできます。このウィザードにより、特定のユーザやアクティビティのパフォーマンスをモニタリングできます。 ● **インデックスコンサルタント**　この機能では、データベース内のインデックスが分析され、改善のための推奨内容が表示されます。このツールは、アプリケーションプロファイリングモードで使用したり、Interactive SQL で単一クエリ用のスタンドアロンツールとして使用できます。 ● **アプリケーションプロファイリング**　アプリケーションプロファイリングによって生成されたデータを使用して、アプリケーションとデータベースとのやりとりを理解し、パフォーマンス上の問題を特定および解消できます。プロファイリング情報を生成する方法は 2 つあります。 ○ **アプリケーションプロファイリングウィザード**　Sybase Central のアプリケーションプロファイリングモードからこのウィザードを使用すると、パフォーマンスを完全に自動的に確認できます。ウィザードの終了時に、パフォーマンスを向上させるための推奨内容が表示されます。このツールは開発環境向けに推奨され、クエリパフォーマンスの追跡に最適化されています。 ○ **プロシージャプロファイリング**　この機能を使用すると、プロシージャ、ユーザ定義関数、イベント、システムトリガ、トリガの実行所要時間を確認できます。プロシージャプロファイリングは、Sybase Central の**アプリケーションプロファイリング**モードの機能として使用できます。 システムプロシージャを使用してプロシージャプロファイリングを実装することもできます。

ツール	詳細
その他のツール	● **要求ロギング**　要求ロギングは、アプリケーションから受け取った要求と、アプリケーションに送られた応答のログを個別にテキストファイルに記録します。データベースサーバがアプリケーションに何を要求されているかを特定したい場合に最も役立ちます。特定のアプリケーションのパフォーマンスを分析するときに、データベースサーバとクライアントのどちらに原因があるか不明な場合は、要求ロギングから始めることもおすすめします。要求ロギングは、問題の原因となっている可能性の高い、データベースサーバに対する特定の要求を特定するためにも使用できます。要求ログは、診断トレーシングとイベントトレーシングによってもたらされる情報のサブセットを提供します。 ● **イベントのトレース**　イベントのトレースは、運用環境で推奨されており、これを使用すると、ログに記録される情報をきめ細かく制御できます。データベースサーバとアプリケーションの両方についての、ユーザ定義とシステム定義の両方のトレースイベントを記録でき、トレースイベントをカスタマイズして、パフォーマンスの問題を特定することができます。 ● **実行プラン**　この機能では、実行プランを調べて、文に関連するデータベース内の情報にアクセスできます。Interactive SQL または SQL 関数を使用すると、実行プランを表示できます。実行プランは、さまざまなフォーマットで取り出すことができます。また、プランを保存することもできます。

注意
このマニュアルでは、「アプリケーションプロファイリング」と「診断トレーシング」の各用語を同じ意味で使うことがあります。診断トレーシングは、詳細なアプリケーションプロファイリングです。

注意
Sybase Central を使用してバージョン 9 のデータベースサーバに接続できます。ただし、Sybase Central のウィンドウのレイアウトが、アプリケーションプロファイリングモードを含まないバージョン 9 のレイアウトに戻ります。Sybase Central のインデックスコンサルタントの場所や使用方法に関する詳細については、バージョン 9 のマニュアルを参照してください。

参照

- 「アプリケーションプロファイリング」148 ページ
- 「診断トレーシング」162 ページ
- 「イベントのトレース」『SQL Anywhere サーバ データベース管理』
- 「要求トレース分析の実行」182 ページ
- 「インデックスコンサルタント」156 ページ
- 「アプリケーションプロファイリングモードでのプロシージャプロファイリング」150 ページ
- 「システムプロシージャを使用したプロシージャプロファイリング」188 ページ
- 「高度：クエリ実行プラン」328 ページ

アプリケーションプロファイリング

アプリケーションプロファイリングによって生成されたデータを使用して、アプリケーションとデータベースとのやりとりを理解し、パフォーマンス上の問題を特定および解消できます。プロファイリング情報を生成する方法は 2 つあります。**アプリケーションプロファイリングウィザード**を使用して自動的に生成する方法と、Sybase Central のアプリケーションプロファイリングモードのツールや機能を使用して生成する方法です。

アプリケーションプロファイリングウィザードは、Windows Mobile ではサポートされていません。ただし、**データベーストレーシングウィザード**はサポートされています。Windows Mobile デバイスからトレーシングデータベースを自動的に作成することはできません。また、Windows Mobile デバイス上のローカルデータベースにトレースすることはできません。Windows Mobile デバイスからトレースし、デスクトップコンピュータ上のデータベースサーバで実行している Windows Mobile データベースのコピーに格納する必要があります。

- **自動アプリケーションプロファイリング**　Sybase Central の**アプリケーションプロファイリングウィザード**を使用すると、一般的なパフォーマンス上の問題を特定できます。アプリケーションプロファイリングウィザードにより、プロファイリングするアクティビティのタイプを定義できます。完了するとデータベースのパフォーマンスを改善するための推奨内容が表示されます。**アプリケーションプロファイリングウィザード**にはインデックスコンサルタントも統合されています。インデックスコンサルタントでは、このデータを使用して、インデックスの改善のための推奨内容が検討されます。

  この方法は、データベースへの接続が少ない環境や、詳細なプロファイリングを必要としない場合に最適です。

- **診断トレーシングを使用した詳細なアプリケーションプロファイリング**　**データベーストレーシングウィザード**を使用すると、トレーシングセッション中に返されたデータや、そのデータが格納される場所をカスタマイズできます。また、コマンドラインを使用して、カスタマイズされたトレーシングデータを格納したり返すことができます。プロファイリングするア

クティビティを制御し、対象とする特定の問題を指定できます。たとえば、データベースサーバで実行される特定の文、使用されるクエリプラン、デッドロック、互いにブロックし合う接続、パフォーマンス統計値などを対象にできます。

この方法は、データベースの負荷が高い環境や、問題を診断するために詳細なプロファイリングが必要な環境に適しています。トレーシングセッションをカスタマイズすることで、トレーシングのスコープを特定のアクティビティに制限できます。また、トレーシングデータをリモートデータベースに格納できます。これらの操作で、プロファイリング対象のデータベースの負荷を下げることができます。

参照

- 「診断トレーシング」162 ページ

アプリケーションプロファイリングウィザードの使用

Sybase Central の**アプリケーションプロファイリングウィザード**を使用して、プロファイリングアプリケーションの診断トレーシングセッションを実行します。このウィザードは、アプリケーションとデータベースとのやりとりに関するデータを収集し、収集されたデータへのアクセスを可能にし、インデックスに関する推奨内容がある場合には表示します。

前提条件

DIAGNOSTICS システムロールに加え、MANAGE PROFILING、SERVER OPERATOR、SELECT ANY TABLE のシステム権限が必要です。

内容と備考

アプリケーションプロファイリングウィザードを使用して、Windows Mobile で実行しているデータベースのトレーシングセッションは作成できません。**データベーストレーシングウィザード**を使用する必要があります。

Sybase Central の**アプリケーションプロファイリングウィザード**を使用すると、ウィザードで分析ファイルに指定する名前と同じ名前でトレーシングデータベースが自動的に作成されます。

アプリケーションプロファイリングウィザードから返されるインデックスの推奨内容の詳細については、「インデックスコンサルタントの推奨内容」158 ページを参照してください。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[モード]** » **[アプリケーショ プロファイリング]** をクリックします。

   アプリケーションプロファイリングウィザードが表示されない場合は、**[アプリケーションプロファイリング]** » **[アプリケーションプロファイリングウィザードを開く]** をクリックします。
3. **アプリケーションプロファイリングウィザード**の指示に従います。

ウィザードは次のことを行います。

- 診断トレーシング情報を格納するローカルデータベースを作成する。
- ネットワークサーバを起動する。
- トレーシングセッションを開始する。
- プロファイリング対象のアプリケーションの実行を要求するプロンプトを表示する。

結果

ウィザードにより結果が表示され、トレーシングセッション中に収集されたデータを確認できます。

参照

- 「アプリケーションプロファイリングウィザードの使用」149 ページ
- 「トレーシングセッションのデータ」163 ページ
- 「プロシージャプロファイリングの結果を解釈する方法」154 ページ
- 「診断トレーシングセッションの作成 (Sybase Central の場合)」176 ページ

アプリケーションプロファイリングモードでのプロシージャプロファイリング

この項では、Sybase Central のアプリケーションプロファイリングモードを使用して、プロシージャプロファイリングを実行する方法について説明します。プロシージャプロファイリングの結果にアクセスするには、この方法を推奨します。ただし、SQL 文を使用してプロシージャプロファイリングを実行することもできます。

プロシージャプロファイリングは、プロシージャ、ユーザ定義関数、イベント、システムトリガ、トリガの実行所要時間を示します。プロファイリング中にこれらのオブジェクトが実行されたら、行ごとの実行時間も表示できます。プロシージャプロファイリングの結果の情報を使用すると、どのオブジェクトを微調整すればデータベース内のパフォーマンスを向上できるかを判断できます。

プロシージャプロファイリングでは、要求ロギングでコストが高いと判断されたストアドプロシージャ、関数、イベント、トリガなどの特定のデータベースプロシージャの分析もできます。また、トリガ、イベント、ネストされたストアドプロシージャコールなどの隠れたコストの高いプロシージャを発見するためにも役立ちます。さらに、プロシージャ本体内の問題となりそうな箇所をピンポイントで見つけるためにも役立ちます。

プロシージャプロファイリングの結果は、データベースサーバによってメモリに格納されます。プロファイリング情報は累積されます。その精度は、1 ミリ秒です。

参照

- 「システムプロシージャを使用したプロシージャプロファイリング」188 ページ

プロシージャプロファイリングの有効化

プロシージャプロファイリングを有効にすると、この機能を無効にするか、データベースサーバが停止されるまで、データベースサーバはプロファイリング情報を収集できます。

前提条件

パーソナルサーバ (dbeng) ではなく、ネットワークサーバ (dbsrv) が稼動している必要があります。

DIAGNOSTICS システムロールに加え、MANAGE PROFILING、SERVER OPERATOR、SELECT ANY TABLE のシステム権限が必要です。

内容と備考

注意
データベースサーバが停止すると、プロファイリング情報はすべて削除されます。プロファイリング情報をエクスポートするには、sa_procedure_profile システムプロシージャを使用します。

SQL 文を使用して、データベースサーバが保持するプロファイリング情報を問い合わせることはできません。プロファイリング情報は、メモリ内データベースサーバのデータ構造に保管されます。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠でデータベースを選択します。
3. **[モード]** » **[アプリケーションプロファイリング]** をクリックします。

   アプリケーションプロファイリングウィザードが表示されない場合は、**[アプリケーションプロファイリング]** » **[アプリケーションプロファイリングウィザードを開く]** をクリックします。
4. アプリケーションプロファイリングウィザードの指示に従います。

   [プロファイリングオプション] ページで **[ストアドプロシージャ、ファンクション、トリガ、またはイベントの実行時間]** をクリックします。

   別のモードに切り替えると、プロシージャプロファイリング情報の収集を停止するかどうかを確認するメッセージが表示されます。**[いいえ]** をクリックすると、プロファイリングを続けながら他のモードで作業ができます。

結果

プロシージャプロファイリングは有効になり、データベースサーバはプロシージャプロファイリング情報を収集します。

参照

- 「プロシージャプロファイリングのリセット」152 ページ
- 「プロシージャプロファイリングの無効化」153 ページ
- 「sa_procedure_profile システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「プロシージャプロファイリングの結果」154 ページ

プロシージャプロファイリングのリセット

プロシージャプロファイリングをリセットすると、プロシージャ、ファンクション、イベント、トリガに関する既存のプロファイリング情報をクリアできます。プロシージャプロファイリングをリセットするために無効にする必要はありません。

前提条件

パーソナルサーバ (dbeng) ではなく、ネットワークサーバ (dbsrv) が稼動している必要があります。

DIAGNOSTICS システムロールに加え、MANAGE PROFILING、SERVER OPERATOR、SELECT ANY TABLE のシステム権限が必要です。

内容と備考

注意
プロシージャプロファイリングが有効になっている場合、リセットしてもプロファイリングは停止しません。また、プロファイリングが無効になっている場合、リセットしてもプロファイリングは開始しません。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠でデータベースを選択します。
3. **[モード]** » **[アプリケーションプロファイリング]** をクリックします。

   アプリケーションプロファイリングウィザードが表示された場合は、**[キャンセル]** をクリックします。
4. 次のうちの 1 つを選択してください。

オプション	アクション
プロシージャプロファイリングが有効になっている場合	**[アプリケーションプロファイリングの詳細]** ウィンドウ枠で、データベースをクリックし、**[選択されたデータベースにおけるプロファイリング設定の表示]** をクリックします。

オプション	アクション
プロシージャプロファイリングが有効になっていない場合	左ウィンドウ枠でデータベースを右クリックして **[プロパティ]** をクリックします。

5. **[プロファイリング設定]** タブをクリックします。
6. **[すぐにリセット]** をクリックします。
7. **[OK]** をクリックします。

結果

プロシージャプロファイリングはリセットされ、既存のプロファイル情報はクリアされます。

参照

- 「プロシージャプロファイリングの有効化」151 ページ
- 「プロシージャプロファイリングの無効化」153 ページ
- 「プロシージャプロファイリングの結果」154 ページ

プロシージャプロファイリングの無効化

プロシージャ、トリガ、ファンクションのプロファイリング情報を取得し終わったら、プロシージャプロファイリングを無効にします。分析が完了している場合は、これまでに収集したプロファイリング情報を削除できます。

前提条件

DIAGNOSTICS システムロールに加え、MANAGE PROFILING、SERVER OPERATOR、SELECT ANY TABLE のシステム権限が必要です。

内容と備考

プロファイリングデータを削除しなかった場合は、プロシージャプロファイリングを無効にした後も Sybase Central の**アプリケーションプロファイリング**モードでデータを表示できます。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠でデータベースを選択します。
3. **[モード]** » **[アプリケーションプロファイリング]** をクリックします。

   アプリケーションプロファイリングウィザードが表示された場合は、**[キャンセル]** をクリックします。
4. (省略可) これまでに収集したプロファイリング情報を削除します。

a. **[アプリケーションプロファイリングの詳細]** ウィンドウ枠で、データベースを選択し、**[選択されたデータベースにおけるプロファイリング設定の表示]** をクリックします。
b. **[プロファイリング設定]** タブをクリックします。
c. **[すぐにクリア]** をクリックします。
d. **[OK]** をクリックします。

5. **[アプリケーションプロファイリングの詳細]** ウィンドウ枠で、**[選択されたデータベースにおけるプロファイリング情報の収集の停止]** をクリックします。

結果

指定されている場合、プロシージャプロファイリングは無効になり、プロファイリング情報がクリアされます。

参照

- 「プロシージャプロファイリングの有効化」151 ページ
- 「プロシージャプロファイリングのリセット」152 ページ
- 「プロシージャプロファイリングの結果」154 ページ

プロシージャプロファイリングの結果

プロシージャプロファイリングの結果を解釈する方法

[プロファイリング結果] タブには、プロシージャプロファイリングを開始してからデータベース内で実行されたすべてのオブジェクトのプロファイリング情報の概要がタイプ別に表示されます。表示される情報は次のとおりです。

カラム	説明
[名前]	オブジェクトの名前。
[所有者]	オブジェクトの所有者。
[テーブル] または [テーブル名]	トリガが属しているテーブル (このカラムはデータベースの [プロファイル] タブにのみ表示される)。
[イベント]	オブジェクトタイプ (プロシージャなど)。
[タイプ]	システムトリガのタイプ。Update または Delete のいずれかです。
[実行]	各オブジェクトが呼び出された回数。
[ミリ秒]	各オブジェクトの合計実行時間。

これらのカラムとその内容は、オブジェクトのタイプによって異なります。

[プロファイリング結果] タブで、プロシージャなどの特定のオブジェクトをダブルクリックすると、そのオブジェクトに固有の詳細な情報が表示されます。表示される情報は次のとおりです。

カラム	説明
[実行]	オブジェクト内のコード行が実行された回数。
[ミリ秒]	行の実行に要した合計時間。
[%]	合計時間に対する、行の実行に要した時間の割合 (パーセント)。
[行]	オブジェクト内の行番号。
[ソース]	実行されたコード。

コード内の他の行に比べて実行時間が長い行は、より効率のいい別の方法で同じ機能を実行できるかどうかを分析してください。プロシージャプロファイリング情報にアクセスするには、データベースに接続され、プロファイリングを有効にしていて、MANAGE PROFILING システム権限を持っている必要があります。

参照

- 「プロシージャプロファイリングの有効化」151 ページ

プロシージャプロファイリングの結果の分析

データベース内のストアドプロシージャ、ユーザ定義のファンクション、トリガ、システムトリガ、イベントのプロファイリング結果を表示できます。

前提条件

DIAGNOSTICS システムロールに加え、MANAGE PROFILING、SERVER OPERATOR、SELECT ANY TABLE のシステム権限が必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. プロシージャプロファイリングの有効化
   a. **[モード]** » **[アプリケーションプロファイリング]** をクリックします。

      アプリケーションプロファイリングウィザードが表示されない場合は、**[アプリケーションプロファイリング]** » **[アプリケーションプロファイリングウィザードを開く]** をクリックします。
   b. アプリケーションプロファイリングウィザードの指示に従います。

      [プロファイリングオプション] ページで **[ストアドプロシージャ、ファンクション、トリガ、またはイベントの実行時間]** をクリックします。

別のモードに切り替えると、プロシージャプロファイリング情報の収集を停止するかどうかを確認するメッセージが表示されます。**[いいえ]** をクリックすると、プロファイリングを続けながら他のモードで作業ができます。

3. 左ウィンドウ枠で、**[トリガ]**、**[システム トリガ]**、**[プロシージャとファンクション]**、または **[イベント]** のいずれかをダブルクリックします。

4. 右ウィンドウ枠で、**[プロファイリング結果]** タブをクリックします。

   プロシージャプロファイリングを有効にしてから実行された、選択したタイプのオブジェクトのリストが表示されます。

   必要なオブジェクトが実行されていないために見つからない場合があります。また、実行されたが、結果が再表示されていない可能性があります。[F5] キーを押して、リストを再表示します。

   予想よりも多くのオブジェクトが表示される場合もあります。1 つのオブジェクトから別のオブジェクトが呼び出される場合は、ユーザが明示的に呼び出す数よりも多くの項目が表示されます。

5. **[プロファイリング結果]** タブで特定のオブジェクトをダブルクリックすると、そのオブジェクトの詳細なプロファイリング結果が表示されます。

結果

右ウィンドウ枠の詳細は、そのオブジェクトの詳細なプロファイリング情報に置き換わります。

インデックスコンサルタント

適切なインデックスセットを選択すると、データベースのパフォーマンスを向上させることができます。SQL Anywhere インデックスコンサルタントを使用すると、データベースに最適なインデックスセットが推奨されるため、インデックスを選択する際に役立ちます。

インデックスコンサルタントは、Interactive SQL を使用して単一のクエリに対して実行するか、Sybase Central のアプリケーションプロファイリングモードを使用してデータベースに対して実行できます。データベースを分析する際、インデックスコンサルタントはトレーシングセッションを使用してデータを収集し、推奨内容を表示します。どのインデックスを使用すると実行プランが改善されるかを判断するために、インデックスコンサルタントでは、これらのインデックスを使用してクエリ実行コストを見積もります。また、複数カラムのインデックスおよび単一カラムのインデックスを評価し、クラスタドインデックスまたは非クラスタドインデックスの影響を調べます。

インデックスコンサルタントは、候補インデックスを生成し、パフォーマンスに対するそれらの効果を調べることにより、データベースまたは単一のクエリを分析します。異なる候補インデックスの効果を調べるために、インデックスコンサルタントは、インデックスセットごとにクエリの最適化を繰り返します。実際にクエリは実行しません。

インデックスコンサルタントの実行中に発生したエラーの報告を分析するには、"DBO"."ix_consultant_log" テーブルの sql_code カラムと log_message カラムを参照します。

参照

- 「インデックス」26 ページ
- 「インデックス」26 ページ
- 「アプリケーションプロファイリング」148 ページ
- 「アプリケーションプロファイリング」148 ページ
- 「インデックスコンサルタントの推奨内容」158 ページ

クエリに対するインデックスコンサルタントの推奨内容の確認

Interactive SQL のクエリに対するインデックスコンサルタントの推奨内容にアクセスできます。

前提条件

DIAGNOSTICS システムロール、または次に示すすべてのシステム権限が必要です。

- SELECT ANY TABLE
- INSERT ANY TABLE
- DELETE ANY TABLE
- UPDATE ANY TABLE

♦ タスク

1. Interactive SQL で、データベースに接続します。
2. **[SQL Statements]** ウィンドウ枠で、SELECT 文、UPDATE 文、または DELETE 文を入力します。
3. **[ツール]** » **[インデックス コンサルタント]** をクリックし、指示に従います。

結果

インデックスコンサルタントの推奨内容は、インデックスコンサルタントウィザードの **[概要]** ウィンドウ枠に表示されます。

データベースに対するインデックスコンサルタントの推奨内容の確認

データベース全体に対するインデックスコンサルタントの推奨内容を確認するには、Sybase Central のアプリケーションプロファイリングモードを使用します。

前提条件

DIAGNOSTICS システムロール、または次に示すすべてのシステム権限が必要です。

- SELECT ANY TABLE
- INSERT ANY TABLE
- DELETE ANY TABLE
- UPDATE ANY TABLE

さらに、MANAGE PROFILING、SERVER OPERATOR、SELECT ANY TABLE システム権限も必要です。

内容と備考

インデックスコンサルタントが推奨内容を決定するためには、プロファイリングデータが必要になります。次の手順は、データを収集し、**アプリケーションプロファイリングウィザード**で収集したデータを使用して推奨内容を表示させるための簡単な方法です。ただし、アプリケーションプロファイリングデータがすでにある場合 (**データベーストレーシングウィザード**を使用して、すでにデータベースのプロファイリングを実行した場合など) は、作成したトレーシングデータベースに対してインデックスコンサルタントを実行することもできます。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[モード]** » **[アプリケーションプロファイリング]** をクリックします。

   アプリケーションプロファイリングウィザードが表示されない場合は、**[アプリケーションプロファイリング]** » **[アプリケーションプロファイリングウィザードを開く]** をクリックし、完了するまでウィザードの指示に従います。
3. **アプリケーションプロファイリングウィザード**の指示に従います。
4. Sybase Central で、**[アプリケーションプロファイリング]** » **[トレーシング データベースでのインデックス コンサルタントの実行]** をクリックします。
5. **インデックスコンサルタントウィザード**の指示に従います。

結果

インデックスコンサルタントの推奨内容が表示されます。

インデックスコンサルタントの推奨内容

トレーシングセッションを分析する前に、推奨内容のタイプを確認するメッセージが表示されます。

- **クラスタドインデックスの推奨**　このオプションを選択すると、インデックスコンサルタントは、クラスタドインデックスと非クラスタドインデックスの効果を分析します。

  一部の負荷に対しては、クラスタドインデックスを適切に選択すると、非クラスタドインデックスよりも大幅にパフォーマンスが改善されます。ただし、このためには REORGANIZE TABLE 文を使用してテーブルを再編成する必要があります。さらに、クラスタドインデックスの効果を考慮すると分析に時間がかかります。

- **既存のセカンダリインデックスの維持**　インデックスコンサルタントは、データベース内の既存のセカンダリインデックスセットを維持または無視して分析を実行できます。セカンダリインデックスは、一意性制約、プライマリキー、または外部キー以外のインデックスです。

参照整合性制約を確保するためのインデックスは、アクセスプランの選択時に常に考慮されます。

分析は、次の手順で行われます。

- **候補インデックスの生成**　各トレーシングセッションに対して、インデックスコンサルタントは、候補インデックスのセットを生成します。大きなテーブルに対して実際のインデックスを作成することは、時間のかかるオペレーションとなる場合があります。そこでインデックスコンサルタントは、候補を仮想インデックスとして作成します。仮想インデックスは、実際にクエリを実行するときには使用できません。しかしオプティマイザは、実際にインデックスが利用できるかのように仮想インデックスを使用して実行プランのコストを推定できます。仮想インデックスにより、インデックスコンサルタントは、実際にインデックスを作成し管理するコストなしに、「もしこうしたインデックスが存在したらどうなるか」という分析ができます。仮想インデックスは、最大 4 カラムまでを扱えます。

- **候補インデックスの利益とコストのテスト**　インデックスコンサルタントは、オプティマイザに、いくつかの候補インデックスの組み合わせを使用した場合としない場合の、トレーシングデータベース内のクエリの実行コストを推定するように要求します。

- **推奨内容の生成**　インデックスコンサルタントは、クエリコストの結果をまとめ、提供する総利益によってインデックスをソートします。インデックスコンサルタントは、SQL スクリプトを提供します。そのスクリプトを実行して推奨内容を実装することも、保存して独自に確認、分析することもできます。

参照

- 「クラスタドインデックス」30 ページ

インデックスコンサルタントの結果

インデックスコンサルタントは、指定された分析の結果を一連のタブで示します。分析の結果は、保存して後で確認できます。

[概要] タブ

[概要] タブには、分析の概要が示されます。これには、クエリ数、推奨インデックス数、推奨インデックスに必要なページ数、推奨インデックスによってもたらされる予測利益などの情報が含まれます。利益値は、内部的なコスト単位に基づいて測定されます。

[推奨インデックス] タブ

[推奨インデックス] タブには、各推奨インデックスに関するデータが含まれます。表示される情報は次のとおりです。

- **[クラスタド]**　各テーブルは、1 つのクラスタドインデックスを持つことができます。クラスタドインデックスは、非クラスタドインデックスと比べて大きな利益をもたらすことがあります。

- **[ページ]**　インデックスを作成することを選択した場合に、インデックスを保持するために必要な推定データベースページ数。

- **[相対利益]**　指定されたインデックスを作成した場合の、総合的な推定利益を示す、1 から 10 までの数字。数字が大きいほど、利益が大きいことを示します。

  相対利益は、内部アルゴリズムを使用して、[コスト利益の合計] カラムとは別に計算されます。相対利益の推定に含まれるいくつかの要素は、コスト利益の合計には含まれません。たとえば、あるインデックスの存在が、別のインデックスに関連する利益に大きく影響することがあります。この場合、相対利益では、各インデックスの影響を個別に推定しようとします。

- **[総利益]**　インデックスに関連して減るコストで、トレーシングセッション内のすべての操作について合計され、内部的なコスト単位 (コストモデル) に基づいて測定されます。

- **[更新コスト]**　インデックスを追加すると、記憶領域が余分に必要となり、データの修正時に余分な作業が必要になるという点で、コストが増えます。[更新コスト] カラムには、インデックスに関連して追加される推定保守コストが示されます。このコストは、内部的なコスト単位に基づいて測定されます。

- **[コスト利益の合計]**　インデックスに関連する総利益から更新コストを引いたものです。

[要求] タブ

[要求] タブには、トレーシングセッション内の個別の要求に対する推奨内容の効果の内訳が示されます。これには、推奨インデックスを適用する前と後の推定コストや、クエリが使用した仮想インデックスなどの情報が含まれます。ボタンを使用すると、要求に対して最適と判断された実行プランを表示できます。

[更新] タブ

[更新] タブには、推奨内容の効果の内訳が示されます。

[未使用のインデックス] タブ

[未使用インデックス] タブには、データベースに存在し、トレーシングセッション内のどの要求の実行にも使用されなかったインデックスがリストされます。セカンダリインデックスのみがリストされます。すなわち、プライマリキー、外部キー、一意性制約のインデックスはリストされません。

[ログ] タブ

[ログ] タブには、この分析について完了したアクティビティがリストされます。

参照

- 「ヒント：適切なページサイズの使用」250 ページ
- 「インデックスコンサルタントの結果の実装」161 ページ
- 「オプティマイザの仕組み」321 ページ
- 「クラスタドインデックス」30 ページ
- 「インデックス」26 ページ
- 「インデックス」26 ページ
- 「アプリケーションプロファイリング」148 ページ

インデックスコンサルタントの結果の実装

インデックスコンサルタントに用意されている SQL スクリプトを実行することで、その結果を実装することもできますが、結果を評価してから実装することが必要な場合もあります。たとえば、分析中に提案されたインデックスの名前を変更する場合などです。

結果を評価する際には、次の点を検討してください。

- **提案されたインデックスは、期待どおりか。** データベース内のデータと、データベースに対して実行されるクエリをよく理解している場合は、提案されたインデックスの有用性を自分自身の知識に照らして確認するとよい場合があります。提案されたインデックスがめったに実行されない単一のクエリにのみ影響する場合や、小さなテーブルに対するインデックスで全体への影響があまりない場合もあります。インデックスコンサルタントが削除するように提案したインデックスが、トレーシングセッションに含まれていなかった他のタスクに使用される場合もあります。

- **提案されたインデックスの効果に密接な相関関係はあるか。** インデックスの推奨では、各インデックスの相対利益を別々に評価しようとします。しかし、2 つのインデックスは、両方とも存在する場合のみ使用されます (クエリは両方存在する場合のみ両方を使用し、どちらかが欠けていれば両方とも使用しない)。**[要求]** タブで、提案されたインデックスがどのように使用されているかクエリプランを検査できます。

- **クラスタドインデックスを作成するときにテーブルを再編成できるか。** クラスタドインデックスを最大限に生かすには、インデックスが作成されるテーブルを REORGANIZE TABLE 文を使用して再編成する必要があります。インデックスコンサルタントが多数のクラスタドインデックスを推奨する場合は、最大の利益を得るために、データベースをアンロードし、再ロードする必要がある場合があります。テーブルのアンロードと再ロードには時間がかかる可能性があり、大量のディスク領域リソースが必要になることがあります。推奨内容を実装するために必要な時間とリソースがあることを確認した方がよいでしょう。

- **分析中のサーバと接続の状態は、運用中の現実的な状態を反映しているか。** 分析の結果は、どのデータがキャッシュにあるかなど、データベースサーバの状態に依存します。また、一部のデータベースオプションの設定などの、接続の状態にも依存します。分析では仮想インデックスのみが作成され、実際に要求は実行されないため、分析中のデータベースサーバの状態は、基本的に静的です (ただし、他の接続によってもたらされた変更を除きます)。分析時の状態がデータベースの典型的なオペレーションでなかった場合は、違う状況で分析を再度実行した方がよい場合があります。

参照

- 「インデックスコンサルタントの推奨内容」158 ページ
- 「SQL スクリプトファイル」766 ページ
- 「インデックス」26 ページ
- 「インデックス」26 ページ
- 「REORGANIZE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「アプリケーションプロファイリング」148 ページ

診断トレーシング

「診断トレーシング」は、詳細なアプリケーションプロファイリングです。データベースサーバによって生成された診断トレーシングのデータには、データベースサーバで処理された文のタイムスタンプと接続 ID が含まれる場合があります。クエリについては、独立性レベル、フェッチされたローの数、カーソルタイプ、クエリ実行プランも診断レーシングのデータに含まれます。INSERT 文、UPDATE 文、DELETE 文については、影響を受けるロー数も含まれます。診断トレーシングを使用して、ロックとデッドロックに関する情報を記録し、パフォーマンス統計値を取得することもできます。

診断トレーシング中に収集されたデータを使用して、次の問題点の特定やトラブルシューティングなどの詳細なアプリケーションプロファイリングアクティビティを実行できます。

- 特定のパフォーマンスの問題
- 実行時間が異常に長い文
- 不正なオプションの設定
- オプティマイザで最適なプランが選択されない状況
- リソース (CPU、メモリ、ディスク I/O) の競合
- アプリケーションの論理の問題

トレーシングデータは、インデックスコンサルタントなどのツールでも、パフォーマンスを向上させるために推奨されるデータベースまたはアプリケーションの具体的な変更方法を判断するために使用されます。

トレーシング アーキテクチャは信頼性とスケーラビリティに優れています。要求ロギングで記録されるすべての情報だけでなく、個別要件に合った分析のための詳細情報も記録できます。

注意
アプリケーションプロファイリングでは、イベントは取得されませんが、イベント内の実行時間が長い文は取得されます。イベントを取得するには、プロシージャ内にイベントコードを埋め込み、イベントからプロシージャを呼び出します。プロシージャを実行する前に、詳細レベルの低いカスタムレベルを指定し、文の取得に条件を設定しないで、トレーシングを有効にします。トレーシングセッションが完了したら、アプリケーションプロファイリングを無効にします。

参照

- 「アプリケーションプロファイリング」148 ページ
- 「要求トレース分析の実行」182 ページ

トレーシングセッションのデータ

診断トレーシングのデータは、「トレーシングセッション」中に収集されます。トレーシングセッションデータを取得するには、次の 3 つの方法があります。

- Sybase Central の**データベーストレーシングウィザード**を使用する。
- **アプリケーションプロファイリングウィザード**の自動処理の一環として透過的に行う。
- ATTACH TRACING 文と DETACH TRACING 文を使用する。

トレーシングセッションの実行中は、SQL Anywhere によって指定されたデータベースの診断情報が生成されます。生成されるトレーシングデータの量は、トレーシングの設定によって異なります。

プロファイリング対象のデータベースは、「運用データベース」、ソースデータベース、またはプロファイリング対象のデータベースと呼ばれます。トレーシングデータが格納されるデータベースは、「トレーシングデータベース」と呼ばれます。運用データベースとトレーシングデータベースは同じデータベースでもかまいません。ただし、運用データベースのサイズが増大するのを防ぐために、トレーシングデータは別のデータベースに格納することをおすすめします。データベースファイルのサイズは一度増大してしまうと、縮小することはできません。また、特に運用データベースが大きく、頻繁に使用される場合は、トレーシングデータの格納および管理にかかるオーバヘッドを別のデータベースで処理した方が、運用データベースのパフォーマンスが向上します。

トレーシングデータが格納されるトレーシングデータベース内のテーブルは、「診断トレーシングテーブル」と呼ばれます。これらのテーブルの所有者は dbo です。

注意
アプリケーションプロファイリングウィザードは、Windows Mobile ではサポートされていません。ただし、**データベーストレーシングウィザード**はサポートされています。また、Windows Mobile デバイスからトレースし、デスクトップコンピュータ上のデータベースサーバで実行している Windows Mobile データベースのコピーに格納する必要があります。Windows Mobile デバイスからトレーシングデータベースを自動的に作成することはできません。また、Windows Mobile デバイス上のローカルデータベースにトレースすることはできません。

トレーシングセッション中に作成されるファイル

トレーシングセッションで作成され、使用されるファイルは、**アプリケーションプロファイリングウィザード**を使用するか、**データベーストレーシングウィザード**を使用するかによって異なります。

アプリケーションプロファイリングウィザードを実行すると、ウィザードによって通知せずにバックグラウンドでトレーシングセッションが取得され、診断テーブルを格納するトレーシングデータベースが作成されます。この外部データベースはウィザードで指定する名前とロケーションで作成され、拡張子が *.adb* です。ウィザードでは、トレーシングデータベースと同じディレクトリに名前が同じで拡張子が *.alg* の分析ログファイルも作成されます。この分析ログファイルには、ウィザードによって行われた分析の結果が含まれ、テキストエディタでいつでも開くことができます。

アプリケーションプロファイリングウィザードによって生成されたデータが不要になったら、セッションに関連するトレーシングデータベースと分析ログファイルを削除できます。

データベーストレーシングウィザードを使用してトレーシングセッションを作成すると、トレーシングデータを内部の運用データベースに保存するか、外部データベース (*tracingData.db* など) に個別に保存するかを確認するメッセージが表示されます。外部トレーシングデータベースを作成することをおすすめします。

> **注意**
> トレーシング情報は、データベースのアンロードまたは再ロード操作の一環としてアンロードされません。トレーシング情報を別のデータベースに転送する場合は、sa_diagnostic_* テーブルの内容を手動でコピーしてください。ただし、この方法はおすすめしません。

参照

- 「外部トレーシングデータベースの作成 (Sybase Central の場合)」183 ページ
- 「診断トレーシングの設定」164 ページ
- 「診断トレーシングテーブル」『SQL Anywhere サーバ SQL リファレンス』

診断トレーシングの設定

Sybase Central の**アプリケーションプロファイリングウィザード**では、事前に設定されたトレーシングの設定を変更できません。ただし、**データベーストレーシングウィザード**を使用すると、トレーシングアクティビティのほとんどの側面を設定できます。次のいずれかの方法により、診断トレーシングを設定します。

- Sybase Central の**データベーストレーシングウィザード**を使用する。この方法では有効なトレーシングの設定がすべて表示されるので、この方法をおすすめします。
- システムプロシージャを使用して診断トレーシングテーブルに格納されている設定を変更する。

トレーシングの設定は、sa_diagnostic_tracing_level システムテーブルに格納されます。

SendingTracingTo と ReceivingTracingFrom の各データベースプロパティは、それぞれトレーシングデータベースと運用データベースを指定します。

参照

- 「診断トレーシングのタイプ」168 ページ
- 「sa_diagnostic_tracing_level テーブル」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_set_tracing_level システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_save_trace_data システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「データベースプロパティの一覧」『SQL Anywhere サーバ データベース管理』
- 「診断トレーシングの設定」175 ページ

診断トレーシングのレベル

次のリストは、**データベーストレーシングウィザード**で指定される診断トレーシングレベルを示したものです。

パフォーマンスへの影響は、トレーシングデータが別のデータベースサーバ上にある (推奨) トレーシングデータベースに送信されるという前提で推定しています。

- **レベル 0**　このレベルでは、トレーシングセッションが実行されますが、トレーシングデータがトレーシングテーブルに送信されません。

- **レベル 1**　パフォーマンスカウンタと実行された文のサンプリングが (5 秒に 1 回) 収集されます。このレベルでは、次の診断トレーシングタイプがあります。

  ○volatile_statistics、1 秒ごとのサンプリング

  ○nonvolatile_statistics、60 秒ごとのサンプリング

  このレベルは、パフォーマンスにほとんど影響がありません。

- **レベル 2**　このレベルでは、パフォーマンスカウンタと実行された文のサンプリングが (5 秒に 1 回) 収集され、すべての実行された文が記録されます。このレベルでは、次の診断トレーシングタイプがあります。

  ○volatile_statistics、1 秒ごとのサンプリング

  ○nonvolatile_statistics、60 秒ごとのサンプリング

  ○statements

  ○plans、5 秒ごとのサンプリング

  このレベルは、パフォーマンスに影響があります。20% 以下のオーバヘッドが発生します。

- **レベル 3**　このレベルでは、レベル 2 と同じ情報が記録されますが、プランのサンプリング頻度が高く (2 秒に 1 回)、ブロックとデッドロックの情報がより詳細です。このレベルには、次の診断トレーシングタイプがあります。

  ○volatile_statistics、1 秒ごとのサンプリング

  ○nonvolatile_statistics、60 秒ごとのサンプリング

  ○statements

  ○blocking

  ○deadlock

  ○statements_with_variables

  ○plans、2 秒ごとのサンプリング

  このレベルはパフォーマンスへの影響が最大で、20% を超えるオーバヘッドが発生します。

診断トレーシングのスコープ

次は、診断トレーシングの「スコープ」のリストです。スコープの値を使用すると、データベース内のアクティビティの特定の発生元だけをトレースできます。たとえば、特定の接続からの要求をトレースするようにスコープを設定できます。スコープの値は、dbo.sa_diagnostic_tracing_level 診断テーブルの scope カラムに格納されます。また、対応する引数 (通常はオブジェクト名やユーザ名などの識別子) が identifier カラムに格納されていることがあります。スコープカラムの値は、**データベーストレーシングウィザード**で指定される設定を反映しています。

scope カラムの値	説明
DATABASE	データベース内で発生し、指定するレベルと条件に対応するすべてのイベントのトレーシングデータを記録します。高コストのクエリのソースを判断するために、バックグラウンドでのデータベースの長期のモニタリングや、短期の診断に使用します。 DATABASE を指定するときに指定する識別子はありません。
ORIGIN	データベースの外部または内部から発生するクエリのトレーシングデータを記録します。 スコープ ORIGIN を指定するときは、External または Internal のいずれかの識別子を指定できます。External は、データベースサーバ外で発生し、指定するレベルと条件に対応するクエリの文のテキストと関連する詳細情報のログを取るように指定します。Internal は、データベースサーバ内で発生し、指定するレベルと条件に対応するクエリに関する同じ情報のログを取るように指定します。
USER	指定したユーザと、指定したユーザが作成した接続から発行されたクエリだけのトレーシングデータを記録します。特定のユーザに関連する問題のあるクエリを診断するために使用します。 このスコープの識別子は、トレースを行う対象ユーザのユーザ ID です。

scope カラムの値	説明
CONNECTION_NAME または CONNECTION_NUMBER	現在の接続で実行された文のトレーシングデータだけを記録します。これらのスコープは、ユーザに複数の接続があり、その 1 つで高コストの文が実行されている場合に使用します。 このスコープの識別子は、それぞれ接続の名前または接続番号です。
FUNCTION、PROCEDURE、EVENT、TRIGGER、または TABLE	指定するオブジェクトを使用する文のトレーシングデータが記録されます。オブジェクトが他のオブジェクトを参照する場合、参照先オブジェクトのデータもすべて記録されます。たとえば、イベントをトリガする関数を使用するプロシージャのトレースを行っている場合、3 つのオブジェクトで、指定するレベルと条件に対応する文のログが取られます。特定のオブジェクトのコストが高い場合や、オブジェクトを参照する文が完了するまでの時間が異常に長い場合に使用します。 TABLE スコープは、テーブル、マテリアライズドビュー、非マテリアライズドビューに使用します。 このスコープの識別子は、オブジェクトの完全に修飾された名前です。

参照

- 「診断トレーシングのタイプ」168 ページ
- 「診断トレーシング条件」172 ページ

カスタマイズされた診断トレーシングレベル

診断トレーシングの設定は、複数のレベルに分類されていますが、これらのレベル内で設定をさらにカスタマイズすることもできます。各レベルで収集される情報のタイプは、「診断トレーシングタイプ」と呼ばれます。指定できるレベルと、それぞれに含まれる診断トレーシングタイプについてこの後で説明します。

診断トレーシングの設定をカスタマイズすると、診断トレーシングセッション内の不要なトレーシングデータの量を減らすことができます。たとえば、ユーザ AliceB のアプリケーションの実行は遅いが、他のユーザは同じ問題が発生していないとします。このとき必要なのは、AliceB のクエリがどのように実行されているかです。したがって、AliceB がアプリケーションで実行しているすべてのクエリとその他の文、および実行時間が長いクエリのクエリプランのリストを収集する必要があります。そのためには、診断トレーシングレベルを 3 に設定し、1 ～ 2 日のトレー

シングデータを生成できます。ただし、このレベルは、他のユーザのパフォーマンスに大きく影響するので、AliceB のアクティビティだけをトレースします。そのためには、診断トレーシングレベルを 3 に設定し、診断トレーシングのスコープを USER にカスタマイズし、ユーザ名として AliceB を指定します。診断トレーシングセッションを数時間実行し、結果を確認します。

診断トレーシングの設定のカスタマイズには、**データベーストレーシングウィザード**を使用することをおすすめします。

sa_set_tracing_level システムプロシージャを使用することもできますが、この方法では、カスタマイズできる設定が少なくなります。

トレーシングセッションの実行中は診断トレーシングの設定を変更しないことをおすすめします。データの解釈が困難になるからです。

参照

- 「診断トレーシングのタイプ」168 ページ
- 「診断トレーシングの設定」175 ページ
- 「sa_set_tracing_level システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「トレーシングセッション中の診断トレーシングの設定の変更」176 ページ

診断トレーシングのタイプ

次の表は、診断トレーシングで選択できるトレーシング「タイプ」を示します。各診断トレーシングタイプには、次に示すように、対応する条件が必要です。また、dbo.sa_diagnostic_tracing_level 診断テーブルの trace_type カラムに格納されます。対応する診断トレーシング条件が trace_condition カラムに格納されていることがあります。

trace_type カラムの値は、**データベーストレーシングウィザード**で指定される設定を反映しています。

trace_type カラムの値	説明
VOLATILE_STATISTICS	頻繁に変化するデータベースとサーバの統計値のサンプルを収集します。 スコープと条件：この診断トレーシングタイプは、DATABASE スコープが必要で、SAMPLE_EVERY 条件をデータ収集の間隔として使用します。

trace_type カラムの値	説明
NONVOLATILE_STATISTICS	頻繁に変化しないデータベースとサーバの統計値のサンプルを収集します。不揮発性の統計値は、揮発性の統計値よりも頻繁に収集できません。不揮発性の統計値を収集するには、揮発性の統計値を収集する必要があります。不揮発性の統計値のサンプリング間隔は、揮発性の統計値に指定する間隔の倍数である必要があります。 スコープと条件：この診断トレーシングタイプは、DATABASE スコープが必要で、SAMPLE_EVERY 条件をデータ収集の間隔として使用します。
CONNECTION_STATISTICS	接続の統計値のサンプルを収集します。スコープがデータベースの場合は、データベースへのすべての接続の統計値が収集されます。スコープがユーザの場合は、指定するユーザのすべての接続の統計値が収集されます。スコープが CONNECTION_NAME または CONNECTION_NUMBER の場合、指定する接続の統計値だけが収集されます。CONNECTION_STATISTICS を収集するには、揮発性の統計値を収集する必要があります。サンプリング間隔は、VOLATILE_STATISTICS に指定する間隔の倍数である必要があります。 スコープと条件：この診断トレーシングタイプは、DATABASE、USER、CONNECTION_NUMBER、CONNECTION_NAME スコープで使用できます。また、SAMPLE_EVERY 条件をデータ収集の間隔として使用します。

trace_type カラムの値	説明
BLOCKING	指定するスコープと条件に従ってブロックに関する情報を収集します。スコープが CONNECTION_NAME または CONNECTION_NUMBER の場合、接続が別の接続をブロックしたとき、または別の接続によってブロックされたときにブロックを記録できます。 スコープと条件：この診断トレーシングタイプはすべてのスコープで使用でき、NONE、NULL、SAMPLE_EVERY の収集条件のいずれかを使用できます。
PLANS	条件とスコープに従ってクエリ実行プランを収集します。 スコープと条件：この診断トレーシングタイプはすべてのスコープで使用でき、NONE、NULL、SAMPLE_EVERY、ABSOLUTE_COST の収集条件のいずれかを使用できます。
PLANS_WITH_STATISTICS	実行の統計値があるプランを収集します。プランはカーソルのクローズ時間に記録されます。RELATIVE_COST_DIFFERENCE 条件を指定した場合、出力内の統計値の一部は推測された統計値である可能性があります。 スコープと条件：この診断トレーシングタイプはすべてのスコープで使用でき、収集条件のいずれかを受け入れます。
STATEMENTS	指定するスコープと条件の SQL 文を収集します。内部変数は、各プロシージャが初めて実行されるときに収集されます。この診断トレーシングタイプは、STATEMENTS_WITH_VARIABLES、PLANS、PLANS_WITH_STATISTICS、OPTIMIZATION_LOGGING、OPTIMIZATION_LOGGING_WITH_PLANS のいずれかの診断トレーシングタイプを指定した場合に自動的に追加されます。 スコープと条件：この診断トレーシングタイプはすべてのスコープで使用でき、収集条件のいずれかを使用できます。

trace_type カラムの値	説明
STATEMENTS_WITH_VARIABLES	SQL 文と文に付加された変数を収集します。各変数 (内部変数またはホスト変数) に割り当てられた値も収集されます。 スコープと条件：この診断トレーシングタイプはすべてのスコープで使用でき、収集条件のいずれかを使用できます。
OPTIMIZATION_LOGGING	各クエリの実行に対してオプティマイザで考慮されたジョイン方式に関するデータを収集します。各方式の実行コストに関する情報や、構造のツリーを再構成するために必要な基本情報が収集されます。クエリに適用された書き換えに関する情報も収集されます。スコープが DATABASE、CONNECTION_NAME、CONNECTION_NUMBER、ORIGIN、または USER 以外の場合、最初に記録される文のテキストが、クエリの最初のテキストと異なる場合があります。これは、現在の文に最適化ロギングを適用するかどうかを判断する前に書き換えが適用される場合があるからです。この診断トレーシングタイプは、OPTIMIZATION_LOGGING_WITH_PLANS 診断トレーシングタイプを指定すると自動的に追加されます。 この診断トレーシングタイプはすべてのスコープに対応し、条件は指定できません。

trace_type カラムの値	説明
OPTIMIZATION_LOGGING_WITH_PLANS	オプティマイザで考慮されたジョイン方式に関するデータを収集します。各方式の実行コストに関する情報や、ジョイン方式のツリー構造を表す完全な XML プランが収集されます。クエリに適用された書き換えに関する情報も収集されます。スコープが DATABASE、CONNECTION_NAME、CONNECTION_NUMBER、ORIGIN、または USER 以外の場合、最初に記録される文のテキストが、クエリの最初のテキストと異なる場合があります。これは、現在の文に最適化ロギングを適用するかどうかを判断する前に書き換えが適用される場合があるからです。OPTIMIZATION_LOGGING トレーシングタイプは、OPTIMIZATION_LOGGING_WITH_PLANS トレーシングタイプを指定すると自動的に追加されます。 この診断トレーシングタイプはすべてのスコープに対応し、条件は指定できません。

参照

- 「診断トレーシングのスコープ」166 ページ
- 「診断トレーシング条件」172 ページ

診断トレーシング条件

次の表は、設定可能な診断トレーシング「条件」を示します。条件は、特定の診断トレーシングタイプのトレーシングデータが記録されるために満たす必要がある条件です。次の表に示すように、ほとんどの条件には値が必要です。条件は、dbo.sa_diagnostic_tracing_level 診断テーブルの trace_condition カラムに格納されます。対応する値 (ミリ秒単位の時間) が value カラムに格納されていることがあります。条件カラムの値は、**データベーストレーシングウィザード**で指定される設定を反映しています。

trace_condition カラムの値	説明
NONE または NULL	レベルとスコープの条件を満たすトレーシングデータをすべて記録します。この条件を高コストの診断トレーシングレベル (たとえばプラン) と同時に長時間使用することはおすすめしません。

trace_condition カラムの値	説明
SAMPLE_EVERY	最後のイベントが記録されてから指定された間隔以上の時間が経過した場合に、レベルとスコープの要件を満たすトレーシングデータを記録します。 値：この条件には、時間をミリ秒単位で表した正の整数を指定します。
ABSOLUTE_COST	実行コストが指定する値以上である文を記録します。 値：この条件には、ミリ秒単位のコスト値を指定します。
RELATIVE_COST_DIFFERENCE	予想実行時間と実際の実行時間の差が、指定する値以上である文を記録します。 値：この条件には、パーセント単位のコスト値を指定します。たとえば、予想よりも 2 倍以上低速の文をログに取るには、値 200 を指定します。

参照

- 「診断トレーシングのスコープ」166 ページ
- 「診断トレーシングのタイプ」168 ページ

現在の診断トレーシングの設定の確認 (Sybase Central の場合)

Sybase Central の**データベーストレーシングウィザード**を使用して、現在の診断トレーシングの設定を表示できます。

前提条件

トレーシングセッションを開始するには、トレーシングデータベースと運用データベースが稼働しているデータベースサーバで TCP/IP が実行されている必要があります。

DIAGNOSTICS システムロールに加え、MANAGE PROFILING、SERVER OPERATOR、SELECT ANY TABLE のシステム権限が必要です。

内容と備考

診断トレーシング設定は、トレーシングセッションが実行中であるかどうかに関係なく取得できます。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[モード]** » **[アプリケーションプロファイリング]** をクリックします。

   アプリケーションプロファイリングウィザードが表示された場合は、**[キャンセル]** をクリックします。
3. 左ウィンドウ枠で、データベースを右クリックし、**[トレーシング]** をクリックして、**データベーストレーシングウィザード**の指示に従ってください。

結果

[トレーシング レベルの編集] リストで、診断トレーシングに現在指定されている設定が表示されます。

参照

- 「sa_diagnostic_tracing_level テーブル」『SQL Anywhere サーバ SQL リファレンス』

現在の診断トレーシングの設定の確認 (SQL の場合)

sa_diagnostic_tracing_level テーブルへのクエリで、現在、有効になっている診断トレーシングの設定を取得できます。

前提条件

DIAGNOSTICS システムロールと MANAGE PROFILING システム権限が必要です。

♦ タスク

1. データベースに接続します。
2. sa_diagnostic_tracing_level テーブルに対して、enabled カラムが 1 のローを問い合わせます。

結果

データベースサーバから、現在使用中の診断トレーシングの設定が返されます。enabled カラムが 1 の場合、設定が有効であることを示します。

例

次の文は、sa_diagnostic_tracing_level 診断テーブルに問い合わせ、現在の診断トレーシングの設定を取得する方法を示します。

```
SELECT * FROM sa_diagnostic_tracing_level WHERE enabled = 1;
```

次の表は、クエリの結果セットの例を示します。

id	scope	identifier	trace_type	trace_condition	value	enabled
1	database	(NULL)	volatile_statistics	sample_every	1,000	1
2	database	(NULL)	nonvolatile_statistics	sample_every	60.000	1
3	database	(NULL)	connection_statistics	(NULL)	60,000	1
4	database	(NULL)	blocking	(NULL)	(NULL)	1
5	database	(NULL)	deadlock	(NULL)	(NULL)	1
6	database	(NULL)	plans_with_statistics	sample_every	2,000	1

参照

- 「sa_diagnostic_tracing_level テーブル」『SQL Anywhere サーバ SQL リファレンス』

診断トレーシングの設定

診断トレーシングの設定は、運用データベースに固有です。Sybase Central の**データベーストレーシングウィザード**を使用すると、トレーシングセッションを作成するときに診断トレーシングの設定を変更できます。

データベーストレーシングウィザードで設定する診断トレーシングの設定は、**アプリケーションプロファイリングウィザード**の設定または動作に影響しません。**アプリケーションプロファイリングウィザード**の設定は事前に設定され、変更できません。

sa_set_tracing_level システムプロシージャを使用して診断トレーシングレベルを変更することもできます。このプロシージャでは、トレーシングセッションは開始されず、またトレーシングセッションが実行中だった場合には失敗します。また、Sybase Central の場合と異なり、スコープ、条件、値など、設定できる項目が限定されてしまいます。

例

次の文は、sa_set_tracing_level システムプロシージャを使用して診断トレーシングレベルを 1 に設定します。

```
CALL sa_set_tracing_level( 1 );
```

既存の設定は、診断トレーシングレベル 1 に関連付けられているデフォルトの設定で上書きされます。

参照

- 「sa_set_tracing_level システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「診断トレーシングのレベル」164 ページ
- 「診断トレーシングセッションの作成 (Sybase Central の場合)」176 ページ

トレーシングセッション中の診断トレーシングの設定の変更

Sybase Central では、トレーシングセッションが進行中の間、新しいトレーシングレベルを追加したり、既存のトレーシングレベルを削除することができます。

前提条件

トレーシングセッションを開始するには、トレーシングデータベースと運用データベースが稼働しているデータベースサーバで TCP/IP が実行されている必要があります。

DIAGNOSTICS システムロールに加え、MANAGE PROFILING、SERVER OPERATOR、SELECT ANY TABLE のシステム権限が必要です。

内容と備考

> **注意**
> トレーシングセッションの実行中は診断トレーシングの設定を変更しないことをおすすめします。データの解釈が困難になるからです。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠でデータベースを右クリックし、**[トレーシング]** » **[トレーシング レベルの変更]** をクリックします。
3. 新しいトレーシングレベルを追加するか、既存のトレーシングレベルを削除します。
4. **[OK]** をクリックします。

結果

トレーシング設定が変更されます。

参照

- 「カスタマイズされた診断トレーシングレベル」167 ページ

診断トレーシングセッションの作成 (Sybase Central の場合)

Sybase Central では、診断トレーシングセッションを作成して、データベースからトレーシングデータを収集できます。

前提条件

トレーシングセッションを開始するには、トレーシングデータベースと運用データベースが稼働しているデータベースサーバで TCP/IP が実行されている必要があります。

DIAGNOSTICS システムロールに加え、MANAGE PROFILING、SERVER OPERATOR、SELECT ANY TABLE のシステム権限が必要です。

内容と備考

診断トレーシングセッションを開始するときに、実行するトレーシングのタイプも設定し、トレーシングデータの格納場所を指定します。トレーシングセッションは、明示的に停止を要求するまで続行されます。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. データベースを右クリックして、**[トレーシング]** をクリックします。
3. **[次へ]** をクリックします。
4. **[トレーシング詳細レベル]** ページで、トレーシングのレベルを選択します。
5. **[トレーシングレベルの編集]** ページで、診断トレーシングの設定をカスタマイズします。
6. **[外部データベースの作成]** ページで、次の操作を行います。
   - **[新しいトレーシング データベースを作成]** をクリックします。
   - データベースを保存するロケーションを選択します。
   - **[ユーザ名]** と **[パスワード]** フィールドに入力します。
   - **[現在のサーバでデータベースを起動]** をクリックします。
   - **[データベースの作成]** をクリックします。
7. **[トレースの開始]** ページで、次の操作を行います。
   - **[外部データベースにトレーシング データを保存]** をクリックします。
   - **[ユーザ名]** と **[パスワード]** フィールドに入力します。運用データベースに接続するために使用したユーザ名とパスワードを指定します。
   - **[その他の接続パラメータ]** フィールドに、部分的な接続文字列として、データベースサーバとデータベース名を入力します。たとえば Server=Server47;DBN=TracingDB のように指定します。

> **注意**
> 外部データベースの場合、接続文字列でサポートされるのは、DBN、DBF、Server、DBKEY、HOST、LINKS (CommLinks) だけです。

- **[格納するトレーシングデータの量を制限するかどうかを指定してください。]** リストで、オプションを選択します。

8. **[完了]** をクリックします。

9. 診断トレーシングデータの収集を終了したら、データベースを右クリックし、**[トレーシング]** » **[トレーシングを停止して保存]** をクリックします。

結果

診断トレーシングセッションが開始し、完了し、データが保存されます。

参照

- 「アプリケーションプロファイリング」 148 ページ
- 「TCP/IP プロトコル」『SQL Anywhere サーバ データベース管理』
- 「ATTACH TRACING 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DETACH TRACING 文」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_set_tracing_level システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

診断トレーシングセッションの作成 (SQL の場合)

Interactive SQL で ATTACH TRACING 文を実行して、トレーシングセッションを開始できます。

前提条件

DIAGNOSTICS システムロールと MANAGE PROFILING システム権限が必要です。

内容と備考

トレーシングセッションを開始することをトレーシングの追加ともいいます。同様に、トレーシングセッションを停止することをトレーシングの分離ともいいます。トレーシングを開始、停止する SQL 文は、それぞれ ATTACH TRACING と DETACH TRACING です。

♦ タスク

1. データベースに接続します。

2. sa_set_tracing_level システムプロシージャを使用してトレーシングレベルを設定します。次に例を示します。

```
CALL sa_set_tracing_level( 1 );
```

3. ATTACH TRACING 文を実行してトレースを開始します。

4. DETACH TRACING 文を実行してトレースを停止します。

結果

トレーシングセッションが作成され、完了します。

次の手順

診断トレーシングデータは、Sybase Central のアプリケーションプロファイリングモードで表示されます。

例

この例は、現在のデータベースの診断トレーシングを開始し、トレーシングデータを別個のデータベースに格納し、格納するデータ量を 2 時間に制限する方法を示します。この例は、パスワード sql と正しい権限を持っている DBA というユーザ ID が存在することを前提にしています。

```
ATTACH TRACING TO 'UID=DBA;PWD=sql;Server=server47;DBN=tracing;Host=myhost'
LIMIT HISTORY 2 HOURS;
```

この例は、現在のデータベースの診断トレーシングを開始し、トレーシングデータをローカルデータベースに格納し、格納するデータ量を 2 MB に制限する方法を示します。

```
ATTACH TRACING TO LOCAL DATABASE LIMIT SIZE 2 MB;
```

この例は、診断トレーシングを停止し、トレーシングセッション中に取得された診断データを保存する方法を示します。

```
DETACH TRACING WITH SAVE;
```

この例は、診断トレーシングを停止し、診断データを保存しない方法を示します。

```
DETACH TRACING WITHOUT SAVE;
```

参照

- 「sa_save_trace_data システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「アプリケーションプロファイリング」 148 ページ
- 「TCP/IP プロトコル」『SQL Anywhere サーバ データベース管理』
- 「ATTACH TRACING 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DETACH TRACING 文」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_set_tracing_level システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

診断トレーシング情報の分析

診断トレーシングのデータは、データベースサーバで発生し、診断トレーシングレベルとトレーシングセッションの設定に対応するすべてのアクティビティの記録です。データを分析するときは、どのような設定だったかを考慮する必要があります。たとえば、予想していた文がトレーシングセッションになかった場合、文が実行されなかった可能性がありますが、高コストの文をトレースするように指定した条件を満たさなかった可能性もあります。

さまざまな理由より、データベースサーバで実行されているアクティビティを詳細に調べることができます。パフォーマンスの問題のトラブルシューティング、リソースの使用量の予測による今後の負荷の計画、アプリケーションの論理のデバッグなどです。

注意
アプリケーションプロファイリングでは、イベントは取得されませんが、イベント内の実行時間が長い文は取得されます。イベントを取得するには、プロシージャ内にイベントコードを埋め込み、イベントからプロシージャを呼び出します。プロシージャを実行する前に、詳細レベルの低いカスタムレベルを指定し、文の取得に条件を設定しないで、トレーシングを有効にします。トレーシングセッションが完了したら、アプリケーションプロファイリングを無効にします。

参照

- 「アプリケーションプロファイリングのチュートリアル」255 ページ

パフォーマンスの問題のトラブルシューティング

アプリケーションプロファイリング機能を使用すると、パフォーマンス上の問題を引き起こしている原因を次の中から特定できます。

- アプリケーションによる処理時間が長い
- 不適切なクエリプラン
- CPU やディスク I/O などの共有ハードウェアリソースの競合
- データベースオブジェクトの競合
- 不適切なデータベース設計

低いデータベースパフォーマンスに関するトラブルシューティングを行う場合、一番の原因がアプリケーションにあるのか、データベースサーバにあるのかをまず判断します。クライアントアプリケーションがどれだけの処理時間を消費しているかを確認するには、アプリケーションプロファイリングツールの **[詳細]** タブで、1 つの接続で結果をフィルタします。その接続からの各要求に間隔がある場合は、速度が下がった一番の原因はアプリケーションクライアントにあります。

データベースサーバがパフォーマンスに影響している場合は、その原因を特定する必要があります。

注意
アプリケーションプロファイリングでは、イベントは取得されませんが、イベント内の実行時間が長い文は取得されます。イベントを取得するには、プロシージャ内にイベントコードを埋め込み、イベントからプロシージャを呼び出します。プロシージャを実行する前に、詳細レベルの低いカスタムレベルを指定し、文の取得に条件を設定しないで、トレーシングを有効にします。トレーシングセッションが完了したら、アプリケーションプロファイリングを無効にします。

参照

- 「アプリケーションプロファイリングのチュートリアル」255 ページ

ハードウェアリソースが制限要因である場合

通常、データベースの負荷が増えると、CPU サイクル、メモリ容量、ディスク I/O 帯域幅によってパフォーマンスが制限されます。非効率的なアプリケーションやデータベースサーバが原因となっている可能性があります。効率の悪い箇所が見つからなかった場合は、ハードウェアリソースを追加する必要がある場合があります。

リソースを追加してもスケーラビリティの問題が解決されず、コンピュータのパフォーマンスが改善されない場合があります。たとえば、データベースサーバが割り当てられた CPU を完全に使用していいる場合は、CPU リソースの追加が必要なことを示している可能性があります。ただし、データベースサーバが使用できる CPU を倍増しても、実行できる処理量が 2 倍になるとはかぎりません。

[アプリケーションプロファイリングの詳細] 領域の **[統計情報]** タブを使用して、ハードウェアリソースがパフォーマンスを制限している要因であるかどうかを判断できます。

- **CPU が制限要因であるかどうかの判断**　CPU が制限要因であるかどうかを判断するには、ProcessCPU 統計を確認します。この統計がグラフに表示されていない場合は、**[統計情報の追加]** ボタンをクリックし、**[ProcessCPU]** をクリックします。グラフで、データベースサーバに割り当てられている CPU あたり 1 秒に 1 ポイント近く ProcessCPU が増加している場合は、CPU が制限要因です。たとえば、2 つの CPU があるデータベースサーバで、プロセス CPU のカウンタが 10 秒間で 2220 から 2237 に増加した場合、この 10 秒間の CPU の使用量は (2237-2220) / 10s * 100 % = 170% となり、各 CPU は容量の 170% / 2 = 85% で動作していることがわかります。

- **メモリが制限要因であるかどうかの判断**　メモリ (バッファプールサイズ) が制限要因になっているかどうかを判断するには、CacheHits と CacheReads データベース統計を確認します。これらの統計がグラフに表示されていない場合は、**[統計情報の追加]** ボタンをクリックし、**[CacheHits]** と **[CacheReads]** をクリックします。CacheHits が CacheReads の 10% 未満である場合は、バッファプールが小さすぎます。この割合が 10 ～ 70% の場合は、バッファプールが小さすぎる可能性があります。データベースサーバのキャッシュサイズを大きくしてみてください。割合が 70% を超える場合は、キャッシュサイズは適切である可能性があります。この方式は、データベースサーバが安定した状態で実行中のとき、つまり起動直後ではなく通常の負荷を処理しているときにのみ該当します。

- **I/O 帯域幅が制限要因であるかどうかの判断**　I/O 帯域幅が制限要因であるかどうかを判断するには、CurrIO データベース統計を確認します。この統計がグラフに表示されていない場合は、**[統計情報の追加]** ボタンをクリックし、**[CurrIO]** をクリックします。この統計値が高く保たれている箇所を探します。グラフの水平部分が長いほど、影響が大きくなります。グラフが、データベースサーバで使用されている物理ディスク数 + 3 以上の値を保っている場合は、ディスクシステムがデータベースサーバのアクティビティのレベルに対処できていない可能性があります。

参照

- 「パフォーマンスモニタの統計値」197 ページ
- 「アプリケーションプロファイリングのチュートリアル」255 ページ
- 「パフォーマンスの問題のトラブルシューティング」180 ページ

アプリケーション論理のデバッグ用のツール

アプリケーションのコードやストアドプロシージャ、トリガ、ファンクション、またはイベントにエラーがある場合は、データベースサーバで実行された、不正なコードに関連する文をすべて確認することをおすすめします。SQL を動的に生成するアプリケーションでは、データベースサーバに渡される実際のテキストを確認して、アプリケーションで SQL のテキストが作成される方法のエラーを検出できます。このようなエラーがあると、クエリの実行に失敗するか、予想とは異なる結果がクエリから返される可能性があります。たとえば、開発時にアプリケーションで SQL 構文エラーが発生したと報告されるが、失敗したクエリの SQL テキストを報告する機能がアプリケーションにないとします。アプリケーションの実行時のトレースがあった場合、構文 (またはその他の) エラーを返した文を検索し、アプリケーションで生成された正確なテキストを確認できます。

プロシージャやトリガなどの内部データベースオブジェクトには、Sybase Central のデバッガを使用できます。ただし、データベースサーバで、特定のプロシージャによって実行される文をすべてトレースするようにして、アプリケーションプロファイリングツールを使用してこれらの文を確認した方が効果的である場合もあります。たとえば、特定のストアドプロシージャが、呼び出し 1000 回のうち 1 回、不正な結果を返すが、失敗する条件がわからないとします。この場合、デバッガでプロシージャのコードを 1000 回実行しないで、このプロシージャの診断トレーシングをオンにしてアプリケーションを実行できます。次に、データベースサーバで実行された一連の文を確認し、プロシージャの不正な実行に対応する一連の文を見つけ、プロシージャが失敗した理由または予想外の動作をする条件を特定できます。プロシージャが予想外の動作をする条件がわかれば、プロシージャにブレークポイントを設定し、デバッガで詳細に調べることができます。

参照

- 「SQL Anywhere のデバッガ」947 ページ

要求トレース分析の実行

特定のアプリケーションまたは要求に問題がある場合、要求トレース分析を行って問題を特定できます。要求トレース分析を行うには、**データベーストレーシングウィザード**を設定して、問題があるユーザ、接続、または要求の診断データだけが収集されるようにします。

前提条件

トレーシングセッションを開始するには、トレーシングデータベースと運用データベースが稼働しているデータベースサーバで TCP/IP が実行されている必要があります。

DIAGNOSTICS システムロールに加え、MANAGE PROFILING、SERVER OPERATOR、SELECT ANY TABLE のシステム権限が必要です。

内容と備考

アプリケーション/プロファイリング/モードのさまざまなデータ表示ツールを使用すると、競合やボトルネックの可能性を特定できます。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[モード]** » **[アプリケーションプロファイリング]** をクリックします。

   アプリケーションプロファイリングウィザードが表示された場合は、**[キャンセル]** をクリックします。
3. データベースを右クリックして **[トレーシング]** をクリックするか、**[トレーシング]** » **[トレーシングの設定と開始]** をクリックします。
4. データベーストレーシングウィザードの指示に従います。
5. トレーシングデータの収集を終了したら、データベースを右クリックし、**[トレーシング]** » **[トレーシングを停止して保存]** をクリックします。
6. **[アプリケーションプロファイリングの詳細]** ウィンドウ枠で、**[分析ファイルを開くかトレーシングデータベースに接続します。]** をクリックします。
7. **[トレーシング データベース内]** をクリックし、**[開く]** をクリックします。
8. **[ユーザ名]** と **[パスワード]** フィールドに入力し、**[OK]** をクリックします。
9. **[アプリケーションプロファイリングの詳細]** ウィンドウ枠で、**[ロギング セッション ID]** リストの最後のエントリをクリックします。
10. **[アプリケーションプロファイリングの詳細]** ウィンドウ枠の最下部にある **[データベーストレーシングデータ]** タブをクリックします。

結果

タブを選択すると、分析のために収集されたデータをさまざまなビューで表示できます。たとえば、**[概要]** タブには、トレーシングセッション中にデータベースに対して実行されたすべての要求が表示されます。これには、要求の実行回数、実行の継続時間、要求を実行したユーザなどが含まれます。リストが長く、特定の要求を探している場合は、**[概要]** タブの **[フィルタリング]** タイトルバーをクリックし、**[SQL 文に含まれる内容]** フィールドに文字列を入力します。

次の手順

特定の要求に関する詳細を表示するには、要求を右クリックし、**[選択された概要の SQL 文に対する詳細 SQL 文を表示]** をクリックします。**[詳細]** タブが表示されます。要求を含むローを右クリックすると、その他の SQL 文、接続、ブロックの詳細など、さらに表示する情報を選択できます。

外部トレーシングデータベースの作成 (Sybase Central の場合)

トレーシングセッションを作成するときは、トレーシングデータをプロファイリング対象のデータベース内に格納するかどうかを選択できます。アプリケーションをテストする場合や、データ

ベースへの接続数が少ない場合は、プロファイリング対象のデータベースへの格納が適しています。

前提条件

トレーシングセッションを開始するには、トレーシングデータベースと運用データベースが稼働しているデータベースサーバで TCP/IP が実行されている必要があります。

DIAGNOSTICS システムロールに加え、MANAGE PROFILING、SERVER OPERATOR、SELECT ANY TABLE のシステム権限が必要です。

内容と備考

注意
データベースで一定時間に 10 以上の接続が処理される場合は、パフォーマンスへの影響を抑えるために、トレーシングデータを外部トレーシングデータベースに格納することをおすすめします。

トレーシングデータベースに、後で実行するトレーシングセッションのデータを格納できます。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[モード]** » **[アプリケーションプロファイリング]** をクリックします。

   アプリケーションプロファイリングウィザードが表示された場合は、**[キャンセル]** をクリックします。
3. 左ウィンドウ枠でデータベースを右クリックして **[トレーシング]** をクリックします。
4. **データベーストレーシングウィザード**の **[外部データベースの作成]** ページで、**[新しいトレーシング データベースを作成]** をクリックし、指示に従います。

結果

分析データを格納する外部データベースが作成され、トレーシング接続が確立します。

参照

- 「診断トレーシングセッションの作成 (Sybase Central の場合)」176 ページ
- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』

外部トレーシングデータベースの作成 (コマンドラインの場合)

アンロードユーティリティ (dbunload) を使用すると、トレーシングセッションのないトレーシングデータベースを手動で作成できます。

前提条件

DIAGNOSTICS システムロールと MANAGE PROFILING システム権限が必要です。

♦ タスク

1. データベースに接続します。
2. dbunload コマンドを実行して、スキーマを運用データベースから新しいトレーシング/データベースにアンロードします。

   例：

   ```
   dbunload -c "UID=DBA;PWD=sql;Server=demo;DBN=demo" -an tracing.db -n -k -kd
   ```

   この例は、-an オプションで指定した名前 (*tracing.db*) で新しいデータベースを作成します。-n オプションによって、プロファイリング対象のデータベース (この例では SQL Anywhere サンプルデータベースの *demo.db*) から新しいトレーシングデータベースにスキーマがアンロードされます。-k オプションによって、アプリケーションプロファイリングツールでトレーシングデータの分析に使用される情報がトレーシングデータベースに格納されます。-kd オプションによって、1 つの DB 領域ファイルのすべての DB 領域が配置されます。
3. トレーシングデータベースを別のコンピュータに保存する場合は、新しいロケーションにコピーします。

結果

分析データを格納する外部データベースが作成され、トレーシングセッションは作成されません。

参照

- 「診断トレーシングセッションの作成 (Sybase Central の場合)」176 ページ
- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』

その他の診断ツールと方法

アプリケーションプロファイリングと診断トレーシングのほかにも、SQL Anywhere データベースの現在のパフォーマンスを分析し、モニタリングするのに役立つさまざまな診断ツールと方法が用意されています。

要求ロギング

要求ロギングは、アプリケーションから受け取った要求と、アプリケーションに送られた応答のログを個別に記録します。データベースサーバがアプリケーションに何を要求されているかを特定したい場合に最も役立ちます。

特定のアプリケーションのパフォーマンスを分析するときに、データベースサーバとクライアントのどちらに原因があるか不明な場合は、要求ロギングから始めることもおすすめします。要求

ロギングは、問題の原因となっている可能性の高い、データベースサーバに対する特定の要求を特定するためにも使用できます。

注意
要求ロギング機能で提供されるすべての機能とデータは、診断トレーシングを使用した場合も利用できます。診断トレーシングでは、それ以外の機能やデータも提供されます。

ログに取られる情報には、タイムスタンプ、接続 ID、要求タイプなどがあります。クエリについては、独立性レベル、フェッチされたローの数、カーソルタイプもあります。INSERT 文、UPDATE 文、DELETE 文については、対象のロー数と実行されたトリガ数もあります。

注意
要求ログには、難読化された機密情報を持つすべての文が含まれます。機密情報が難読化されていない唯一のケースは、文に解析エラーがある場合です。

-zr サーバオプションを使用すると、データベースサーバの起動時に要求ロギングをオンにできます。-zo サーバオプションを使用すると、出力を要求ログファイルにリダイレクトして、さらに分析を進めることができます。-zn オプションと -zs オプションを使用すると、保存する要求ログファイルの数と要求ログファイルの最大サイズを指定できます。

これらのオプションの詳細については、次の項を参照してください。

- 「-zr データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-zo データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-zn データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-zs データベースサーバオプション」『SQL Anywhere サーバ データベース管理』

注意
サーバオプションは、Sybase Central の診断トレーシングに影響しません。ファイルベースの要求ロギングは、Sybase Central の診断トレーシング機能とは完全に別個のものです。Sybase Central の診断トレーシング機能では、データベース内の dbo 所有の診断テーブルを使用して要求ログ情報が格納されます。

sa_get_request_times システムプロシージャは、要求ログを読み込み、ログの文とその実行時間をグローバルテンポラリテーブル (satmp_request_time) に格納します。INSERT、UPDATE、DELETE の各文の場合は、記録される時間は、文が実行された時間です。クエリの場合、記録された時間は、PREPARE から DROP (DESCRIBE/OPEN/FETCH/CLOSE) までの合計所要時間です。したがって、オープンカーソルには注意する必要があります。

改善候補の文について satmp_request_time を分析します。低コストでも頻繁に実行される文が、パフォーマンスの問題を引き起こしている可能性があります。

sa_get_request_profile を使用して sa_get_request_times を呼び出し、satmp_request_time を satmp_request_profile という別のグローバルテンポラリテーブルにまとめることができます。また、このプロシージャは、文をグループ化し、呼び出しの回数、実行時間などの情報を提供します。

> **警告**
> Perl スクリプト *tracetime.pl* を使用してログを分析する場合は、要求ログがキャプチャされる間、max_client_statements_cached オプションを 0 に設定してクライアント文のキャシュを無効にしておく必要があります。

参照

- 「診断トレーシング」162 ページ
- 「sa_get_request_times システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_get_request_profile システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「max_client_statements_cached オプション」『SQL Anywhere サーバ データベース管理』
- 「sa_server_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

例

要求ログへの出力は、sa_server_option システムプロシージャを使用してフィルタして、特定の接続やデータベースからの要求のみを含めるようにできます。これにより、アクティブな接続の多いデータベースサーバや複数のデータベースのあるデータベースサーバをモニタリングするときのログサイズを縮小できます。

- **接続に基づくフィルタ**　次の構文を使用します。

  ```
  CALL sa_server_option( 'RequestFilterConn' , connection-id );
  ```

  connection-id は CALL sa_conn_info( ) を実行して取得できます。

- **データベースに基づくフィルタ**　次の構文を使用します。

  ```
  CALL sa_server_option( 'RequestFilterDB' , database-id );
  ```

  database-id は、データベースに接続している場合に SELECT CONNECTION_PROPERTY('DBNumber') を実行して入手できます。フィルタは、明示的にリセットするまで、またはデータベースサーバが停止されるまで有効です。

- **フィルタのリセット**　次の 2 つの文のいずれかを使用して、接続またはデータベースごとにフィルタをリセットします。

  ```
  CALL sa_server_option( 'RequestFilterConn' , -1 );
  ```

  ```
  CALL sa_server_option( 'RequestFilterDB' , -1 );
  ```

- **要求ログへのホスト変数の出力**　ホスト変数の値を要求ログに含めるには、次の操作を行います。

  ○-zr サーバオプションに値 **hostvars** を指定します。

  ○次の文を実行します。

  ```
  CALL sa_server_option( 'RequestLogging' , 'hostvars' );
  ```

  要求ログ分析プロシージャ sa_get_request_times は、ログ中のホスト変数を認識し、それらをグローバルテンポラリテーブル satmp_request_hostvar に追加します。

システムプロシージャを使用したプロシージャプロファイリング

プロシージャプロファイリングでは、すべての接続によるストアドプロシージャ、ユーザ定義関数、イベント、システムトリガ、トリガの使用状況に関する有用な情報が収集されます。プロシージャプロファイリングはシステムプロシージャ呼び出しを使用して実行できます。Sybase Central のアプリケーションプロファイリングモードのプロシージャプロファイリング機能を使用することをおすすめします。

参照

- 「アプリケーションプロファイリングモードでのプロシージャプロファイリング」150 ページ

プロシージャプロファイリングの有効化 (SQL の場合)

sa_server_option システムプロシージャを使用して、プロシージャプロファイリングを有効にすることができます。

前提条件

DIAGNOSTICS システムロールと MANAGE PROFILING システム権限が必要です。

♦ タスク

1. データベースに接続します。
2. sa_server_option システムプロシージャを呼び出して、ProcedureProfiling オプションを ON に設定します。

   たとえば、次の CALL 文を実行します。

   ```
   CALL sa_server_option( 'ProcedureProfiling' , 'ON' );
   ```

結果

プロシージャプロファイリングが有効になります。

参照

- 「sa_server_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

プロシージャプロファイリングのユーザ別のフィルタ (SQL の場合)

必要に応じて、他の接続によるデータベースの使用を妨害することなく、特定のユーザが使用しているプロシージャを確認できます。この機能は、接続がすでに存在する場合、または複数のユーザが同じユーザ ID で接続する場合に便利です。

前提条件

DIAGNOSTICS システムロールと MANAGE PROFILING システム権限が必要です。

♦ タスク

1. データベースに接続します。
2. 次のように sa_server_option システムプロシージャを呼び出します。

```
CALL sa_server_option( 'ProfileFilterUser' , 'userid' );
```

結果

userid の値は、モニタされるユーザの名前です。

参照

- 「sa_server_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

プロシージャプロファイリングのリセット (SQL の場合)

プロファイリングをリセットすると、データベースは古い情報をクリアし、すぐにプロシージャ、ファンクション、イベント、トリガに関する新しい情報の収集を開始します。sa_server_option システムプロシージャを使用して、Interactive SQL からプロシージャプロファイリングをリセットできます。

前提条件

DIAGNOSTICS システムロールと MANAGE PROFILING システム権限が必要です。

プロシージャプロファイリングが有効になっている必要があります。

♦ タスク

- sa_server_option システムプロシージャを呼び出して、ProcedureProfiling オプションを RESET に設定します。

  たとえば、次の　CALL 文を実行します。

```
CALL sa_server_option( 'ProcedureProfiling' , 'RESET' );
```

結果

プロシージャプロファイリングがリセットされます。

参照

- 「sa_server_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

プロシージャプロファイリングの無効化 (SQL の場合)

sa_server_option システムプロシージャを使用して、Interactive SQL からプロシージャプロファイリングを無効にし、既存のデータをクリアできます。

前提条件

DIAGNOSTICS システムロールと MANAGE PROFILING システム権限が必要です。

内容と備考

プロファイリング情報の確認後に、プロファイリングを無効にするか、クリアできます。プロファイリングを無効にすると、データベースはプロファイリング情報の収集を停止します。ただし、その時点までに収集された情報は、Sybase Central の **[プロファイル]** タブに残ります。プロファイリングをクリアすると、データベースはプロファイリングをオフにして、Sybase Central の **[プロファイル]** タブからプロファイリングデータをすべてクリアします。

♦ タスク

- sa_server_option システムプロシージャを呼び出して、ProcedureProfiling オプションを OFF または CLEAR に設定します。

  たとえば、次の CALL 文を実行してプロファイリングをオフにします。

  ```
  CALL sa_server_option( 'ProcedureProfiling' , 'OFF' );
  ```

  または次の CALL 文を実行してプロファイリングをクリアします。

  ```
  CALL sa_server_option( 'ProcedureProfiling' , 'CLEAR' );
  ```

結果

オプション CLEAR を使用した場合、プロシージャプロファイリングは無効になり、プロファイリングデータはクリアされます。

参照

- 「sa_server_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

システムプロシージャはプロファイリング情報を取得可能

システムプロシージャを使用して、ストアドプロシージャ、ファンクション、イベント、システムトリガ、トリガのプロシージャプロファイリング情報を表示できます。また、プロシージャプロファイリングが有効になっている必要があります。

sa_procedure_profile システムプロシージャでは、各オブジェクト内の行の実行時間などの詳細なプロファイリング情報が表示されます。結果セット内の各行は、オブジェクト内の実行可能なコード行を表します。

sa_procedure_profile_summary システムプロシージャでは、各オブジェクトの合計実行時間が表示され、実行された全オブジェクトの概要を示します。結果セット内の各行は、1 つのオブジェクトの実行の詳細を表します。

これらのシステムプロシージャの結果では、表示されるオブジェクト数が、呼び出したオブジェクトよりも多い場合があります。これは、1 つのオブジェクトから別のオブジェクトが呼び出さ

れる場合があるからです。たとえば、トリガからストアドプロシージャが呼び出され、そのストアドプロシージャから別のストアドプロシージャが呼び出される場合があります。

参照

- 「プロシージャプロファイリングの有効化 (SQL の場合)」 188 ページ
- 「sa_procedure_profile_summary システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_procedure_profile システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_server_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

タイミングユーティリティ

Fetchtst、Instest、Trantest など、いくつかのパフォーマンステストユーティリティが、*%SQLANYSAMP16%¥SQLAnywhere* にあります。

fetchtst ユーティリティは、任意のクエリについてフェッチ速度を測定します。instest ユーティリティは、ローをテーブルに挿入するための所要時間を決定します。trantest ユーティリティは、特定のデータベース設計とトランザクションセットが与えられている特定のサーバ設定で処理できる負荷を測定します。

これらのツールを使用すると、統計情報付のグラフィカルなプランよりもさらに正確に測定でき、特定のサーバとデータベース構成について、達成可能な最善のパフォーマンス (たとえばスループット) の目安がわかります。

ツールの完全なマニュアルについては、ユーティリティと同じフォルダの *readme.txt* ファイルを参照してください。

データベースのパフォーマンスのモニタ

SQL Anywhere には、データベースのパフォーマンスをモニタするのに使用する統計値のセットがあります。これらの統計値には数多くの方法でアクセスできます。

- **SQL 関数**　これらの関数を使用すると、アプリケーションから SQL Anywhere データベースの統計値に直接アクセスできます。
- **Sybase Central パフォーマンスモニタ**　このグラフィカルツールでは、データベースに問い合わせて、パフォーマンスモニタでグラフ表示するように設定した統計値だけが表示されます。
- **Windows パフォーマンスモニタ**　Windows オペレーティングシステムに付属するモニタツールです。
- **パフォーマンス統計ユーティリティ (dbstats)**　データベースサーバ、データベース、UNIX 上で実行されているデータベースサーバの接続の統計値をモニタリングできます。
- **SQL Anywhere コンソールユーティリティ (dbconsole)**　データベースサーバ接続の管理機能とモニタリング機能を提供します。

これらの方法は、リアルタイムでモニタリングする場合に役立ちます。しかし、診断トレーシングの一環として統計値を取得して、後で分析するために保存することもできます。

参照

- 「統計のモニタに使用される SQL 関数」192 ページ
- 「統計をモニタリングするための Sybase Central パフォーマンスモニタ機能」193 ページ
- 「Windows パフォーマンスモニタ」195 ページ
- 「パフォーマンス統計値ユーティリティ (dbstats) (UNIX)」『SQL Anywhere サーバ データベース管理』
- 「SQL Anywhere コンソールユーティリティ (dbconsole)」『SQL Anywhere サーバ データベース管理』
- 「診断トレーシング」162 ページ
- 「パフォーマンスモニタの統計値」197 ページ
- 「データベースのモニタリング」『SQL Anywhere サーバ データベース管理』

統計のモニタに使用される SQL 関数

SQL Anywhere は、接続ごと、データベースごと、またはサーバワイドに情報にアクセスできるシステム関数のセットを提供します。アクセスできる情報は、データベースサーバ名のような静的な情報からディスクとメモリの使用状況のようなパフォーマンス関連の詳細な統計にまで及びます。

システム情報を取り出す関数

次の関数は、システム情報を取り出します。

- **PROPERTY 関数**　この関数は、サーバワイドに指定されたプロパティの値を返します。
- **DB_PROPERTY 関数と DB_EXTENDED_PROPERTY 関数**　これらの関数は、指定されている場合にはそのデータベースの、デフォルトでは現在のデータベースの指定されたプロパティ値を返します。
- **CONNECTION_PROPERTY 関数と CONNECTION_EXTENDED_PROPERTY 関数**　これらの関数は、指定されている場合はその接続の、デフォルトでは現在の接続の指定されたプロパティ値を返します。

取り出したいプロパティの名前のみ、引数として指定します。関数は、現在のサーバ、接続、またはデータベースの値を返します。

クエリの効率を改善する

よりよいパフォーマンスを得るには、データベースのアクティビティをモニタするクライアントアプリケーションは PROPERTY_NUMBER 関数を使用して、名前が付けられたプロパティを識別すべきです。その後そのプロパティ番号を使用して統計を繰り返し取り出すようにします。

こうして取得したプロパティ名は、さまざまなデータベースの統計値を得るのに使うことができます。トランザクションログページへの書き込み回数や、チェックポイント実行回数から、メモリキャッシュからインデックスリーフページを読み出した回数まで、幅広く使えます。

次の文のセットは、Interactive SQL から行うプロセスを示します。

```
CREATE VARIABLE propnum INT;
CREATE VARIABLE propval INT;
SET propnum = PROPERTY_NUMBER( 'CacheRead' );
SET propval = DB_PROPERTY( propnum );
```

例

次に示す文は、変数 server_name を現在のサーバ名に設定します。

```
SET server_name = PROPERTY( 'name' );
```

次に示すクエリは、現在の接続のユーザ ID を返します。

```
SELECT CONNECTION_PROPERTY( 'UserID' );
```

次に示すクエリは、現在のデータベースのルートファイル名を返します。

```
SELECT DB_PROPERTY( 'file' );
```

参照

- 「PROPERTY_NUMBER 関数 [システム]」『SQL Anywhere サーバ SQL リファレンス』
- 「PROPERTY 関数 [システム]」『SQL Anywhere サーバ SQL リファレンス』
- 「DB_PROPERTY 関数 [システム]」『SQL Anywhere サーバ SQL リファレンス』
- 「DB_EXTENDED_PROPERTY 関数 [システム]」『SQL Anywhere サーバ SQL リファレンス』
- 「CONNECTION_PROPERTY 関数 [システム]」『SQL Anywhere サーバ SQL リファレンス』
- 「CONNECTION_EXTENDED_PROPERTY 関数 [文字列]」『SQL Anywhere サーバ SQL リファレンス』
- 「システム関数」『SQL Anywhere サーバ SQL リファレンス』
- 「統計をモニタリングするための Sybase Central パフォーマンスモニタ機能」193 ページ

統計をモニタリングするための Sybase Central パフォーマンスモニタ機能

Sybase Central パフォーマンスモニタは、ディスクアクセスやメモリアクセスなど、データベースサーバの動作に関する詳細な情報を追跡するのに役立ちます。Sybase Central パフォーマンスモニタでは、接続できる任意の SQL Anywhere データベースサーバの統計値をグラフ表示できます。

次に、Sybase Central パフォーマンスモニタの機能を示します。

- リアルタイム更新 (調整可能な間隔で)
- 色分けされたサイズ変更可能な凡例
- 設定可能な表示プロパティ

Sybase Central パフォーマンスモニタでは、データベースに問い合わせて統計値が収集されます。このため、キャッシュ読み込み／秒などの統計値に影響する可能性があります。代わりに Windows パフォーマンスモニタを使用すると、統計値がモニタの影響を受けません。

複数バージョンの SQL Anywhere を同時に実行している場合は、複数バージョンのパフォーマンスモニタも同時に実行できます。

参照

- 「Windows パフォーマンスモニタ」 195 ページ
- 「パフォーマンスモニタの統計値」 197 ページ

Sybase Central パフォーマンスモニタを開く

Sybase Central パフォーマンスモニタは、Sybase Central の右ウィンドウ枠で **[パフォーマンスモニタ]** タブを選択すると表示されます。このグラフには、表示するように設定した統計値だけが表示されます。

前提条件

この作業を実行するための前提条件は、ありません。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠でサーバを選択します。
3. 右ウィンドウ枠で、**[パフォーマンスモニタ]** タブをクリックします。

結果

パフォーマンスモニタ が Sybase Central で開きます。

参照

- 「統計値の追加と削除」 194 ページ
- 「Windows パフォーマンスモニタ」 195 ページ
- 「統計値の追加と削除」 194 ページ

統計値の追加と削除

Sybase Central を使用して、モニタした統計情報をパフォーマンスモニタに追加する、またはモニタした統計情報をパフォーマンスモニタから削除できます。

前提条件

この作業を実行するための前提条件は、ありません。

内容と備考

> **ヒント**
> 統計値の追加と削除は、統計値のプロパティウィンドウにある Sybase Central パフォーマンスモニタからもできます。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠でサーバを選択します。
3. 右ウィンドウ枠で、**[統計情報]** タブをクリックします。
4. モニタされていない統計値を右クリックし、**[パフォーマンスモニタに追加]** または **[パフォーマンスモニタから削除]** をクリックします。

結果

指定された統計情報は、Sybase Central のパフォーマンスモニタに追加されるか、パフォーマンスモニタから削除されます。

参照

- 「パフォーマンスモニタの統計値」197 ページ
- 「Sybase Central パフォーマンスモニタを開く」194 ページ
- 「Windows パフォーマンスモニタ」195 ページ

Windows パフォーマンスモニタ

Sybase Central パフォーマンスモニタの代わりに、Windows パフォーマンスモニタを使用することもできます。

Windows パフォーマンスモニタは、Sybase Central パフォーマンスモニタよりもパフォーマンスの統計値、特にネットワーク通信の統計値が多くなっています。Windows 版モニタは、データベースサーバに対してクエリを実行する代わりに、共有メモリスキームを使用するので、統計値自体には影響を与えません。

複数のバージョンの SQL Anywhere を同時に実行する場合は、複数のバージョンのパフォーマンスモニタも同時に実行できます。

Windows パフォーマンスモニタで使用されるメモリを制御するデータベースサーバを起動する際には、データベースサーバのオプション、およびパフォーマンスモニタでモニタできる接続の最大数やデータベースを指定できます。

- 「-ks データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-ksc データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-ksd データベースサーバオプション」『SQL Anywhere サーバ データベース管理』

参照

- 「パフォーマンスモニタの統計値」197 ページ

Windows パフォーマンスモニタの使用

SQL Anywhere データベース、サーバ、または接続に関連したカウンタを表示するには、Windows パフォーマンスモニタを使用します。

前提条件

SQL Anywhere データベースサーバは、稼働している必要があります。

内容と備考

他のバージョンの Windows で Windows パフォーマンスモニタを起動する方法については、使用する Windows オペレーティングシステムのマニュアルを参照してください。

♦ タスク

1. パフォーマンスモニタを起動します。
   a. Windows の **[コントロール パネル]** で **[管理ツール]** をクリックします。
   b. **[パフォーマンス]** をクリックします。
2. ツールバーのプラス記号ツール (+) をクリックします。
3. **[パフォーマンス オブジェクト]** リストから、次のいずれかを選択します。
   - **SQL Anywhere 16 接続**　単一の接続のパフォーマンスがモニタされます。この項目が表示されるためには、接続が存在する必要があります。
   - **SQL Anywhere 16 データベース**　単一のデータベースのパフォーマンスがモニタされます。
   - **SQL Anywhere 16 サーバ**　サーバワイドでパフォーマンスがモニタされます。

   [カウンタ] ボックスに閲覧できる統計値のリストが表示されます。

   [SQL Anywhere 16 接続] または **[SQL Anywhere 16 データベース]** をクリックした場合は、**[インスタンス]** ボックスに統計値を表示できる接続またはデータベースのリストが表示されます。
4. **[カウンタ]** リストから、閲覧する統計値をクリックします。
5. **[SQL Anywhere 16 接続]** または **[SQL Anywhere 16 データベース]** をクリックした場合は、**[インスタンス]** ボックスから、モニタするデータベース接続またはデータベースを選択します。
6. 選択したカウンタの詳細については、**[説明]** をクリックします。
7. カウンタを表示するには、**[追加]** をクリックします。
8. 表示するカウンタをすべて選択したら **[閉じる]** をクリックします。

結果

指定された統計情報が、Windows パフォーマンスモニタに表示されます。

パフォーマンスモニタの統計値

SQL Anywhere で提供される統計値を次に示します。

- 「キャッシュの統計値」
- 「チェックポイントとリカバリの統計値」
- 「通信の統計値」
- 「ディスク I/O の統計値」
- 「ディスク読み込みの統計値」
- 「ディスク書き込みの統計値」
- 「インデックスの統計値」
- 「メモリページの統計値」
- 「要求の統計値」
- 「ユーザ定義統計情報」
- 「その他の統計値」

処理速度は毎秒レポートされます。

キャッシュの統計値

これらの統計値により、キャッシュの使用状態を分析できます。

統計情報	スコープ	説明
キャッシュヒット／秒	接続とデータベース	ルックアップの対象となるデータベースページがキャッシュ内で検出された頻度を示します。
キャッシュ読み込み：インデックス内部／秒	接続とデータベース	インデックスの内部ノードのページがキャッシュから読み込まれる頻度を示します。
キャッシュ読み込み：インデックスリーフ／秒	接続とデータベース	インデックスリーフページがキャッシュから読み込まれる頻度を示します。
キャッシュ読み込み：テーブル／秒	接続とデータベース	テーブルページがキャッシュから読み込まれる頻度を示します。
キャッシュ読み込み：合計ページ数／秒	接続とデータベース	データベースのページをキャッシュで検索する頻度を示します。
キャッシュ読み込み：ワークテーブル	接続とデータベース	ワークテーブルページがキャッシュから読み込まれる頻度を示します。

統計情報	スコープ	説明
キャッシュ置換：合計ページ数／秒	サーバ	必要な別のページの領域を確保するためにデータベースページがキャッシュからパージされている頻度を示します。
キャッシュサイズ：現在の値	サーバ	データベースサーバキャッシュの現在のサイズ (キロバイト) を示します。
キャッシュサイズ：最大値	サーバ	データベースサーバキャッシュの許容最大サイズ (キロバイト) を示します。
キャッシュサイズ：最小値	サーバ	データベースサーバキャッシュの許容最小サイズ (キロバイト) を示します。
キャッシュサイズ：ピーク値	サーバ	データベースサーバキャッシュのピーク時のサイズ (キロバイト) を示します。

チェックポイントとリカバリの統計値

これらの統計値により、データベースがアイドル状態のときに行われたチェックポイントとリカバリ動作を分析できます。

統計情報	スコープ	説明
チェックポイントフラッシュ／秒	データベース	チェックポイントの間に隣接ページを書き出す頻度を示します。
チェックポイントの緊急度	データベース	チェックポイントの緊急度 (パーセント) を示します。
チェックポイント／秒	データベース	チェックポイントを実行する頻度を示します。
チェックポイントログ：ビットマップサイズ	データベース	チェックポイントログのビットマップのサイズを示します。

統計情報	スコープ	説明
チェックポイントログ：ディスクへのコミット／秒	データベース	チェックポイントログの commit_to_disk 操作の実行頻度を示します。
チェックポイントログ：ログサイズ	データベース	チェックポイントログのページのサイズを示します。
チェックポイントログ：保存ページイメージ／秒	データベース	変更前にページがチェックポイントログに保存されている頻度を示します。
チェックポイントログ：使用ページ数	データベース	チェックポイントログ内で現在使用中のページ数を示します。
チェックポイントログ：ページ再配置／秒	データベース	チェックポイントログ内のページが再配置されている頻度を示します。
チェックポイントログ：プレイメージ保存／秒	データベース	新しいデータベースページのプレイメージがチェックポイントログに追加されている頻度を示します。
チェックポイントログ：ページ書き込み／秒	データベース	ページがチェックポイントログに書き込まれている頻度を示します。
チェックポイントログ：書き込み／秒	データベース	チェックポイントログでディスク書き込みが行われている頻度を示します。1 回の書き込みに複数のページが含まれる場合があります。
チェックポイントログ：ビットマップへの書き込み／秒	データベース	チェックポイントログでビットマップページのディスク書き込みが行われている頻度を示します。
アイドルアクティブ／秒	データベース	データベースサーバのアイドルスレッドがアクティブになって、アイドル書き込み、アイドルチェックポイントなどを行う頻度を示します。

統計情報	スコープ	説明
アイドルチェックポイント時間	データベース	アイドルチェックポイントに費やされた合計時間 (秒) を示します。
アイドルチェックポイント／秒	データベース	チェックポイントがデータベースサーバのアイドルスレッドによって最後まで行われる頻度を示します。アイドルスレッドが最後のダーティページをキャッシュに書き出すたびに、アイドルチェックポイントが発生します。
アイドル書き込み／秒	データベース	データベースサーバのアイドルスレッドによってディスク書き込みが発行される頻度を示します。
リカバリ I/O 予測	データベース	データベースのリカバリに必要な I/O 操作の推定回数を示します。
リカバリの緊急度	データベース	リカバリの緊急度 (パーセント) を示します。

参照

- 「データベースサーバがチェックポイントのタイミングを決定する方法」『SQL Anywhere サーバ データベース管理』
- 「checkpoint_time オプション」『SQL Anywhere サーバ データベース管理』
- 「-gc データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「recovery_time オプション」『SQL Anywhere サーバ データベース管理』
- 「チェックポイントログ」『SQL Anywhere サーバ データベース管理』

通信の統計値

これらの統計値により、クライアント／サーバ間の通信アクティビティを分析できます。

統計情報	スコープ	説明
通信：受信バイト数／秒	接続、データベース、サーバ	ネットワークデータが受信される速度 (バイト単位) を示します。

統計情報	スコープ	説明
通信：無圧縮受信バイト数／秒	接続、データベース、サーバ	圧縮が無効になっていた場合のバイトの受信速度を示します。
通信：送信バイト数／秒	接続、データベース、サーバ	ネットワークでバイトが送信される速度を示します。
通信：無圧縮送信バイト数／秒	接続、データベース、サーバ	圧縮が無効になっていた場合のバイトの送信速度を示します。
通信：空きバッファ	サーバ	空きネットワークバッファの数を示します。
通信：受信されるマルチパケット数／秒	サーバ	マルチパケットデリバリの受信速度を示します。
通信：送信されるマルチパケット数／秒	サーバ	マルチパケットデリバリの送信速度を示します。
通信：受信パケット数／秒	接続、データベース、サーバ	ネットワークパケットの受信速度を示します。
通信：無圧縮受信パケット数／秒	接続、データベース、サーバ	圧縮が無効になっていた場合のネットワークパケットの受信速度を示します。
通信：送信パケット数／秒	接続、データベース、サーバ	ネットワークパケットの送信速度を示します。
通信：無圧縮送信パケット数／秒	接続、データベース、サーバ	圧縮が無効になっていた場合のネットワークパケットの送信速度を示します。
通信：リモートプット待ち／秒	サーバ	情報の送信に使用できるバッファがないため、通信リンクが待機する必要がある頻度を示します。この統計は TCP/IP の場合にのみ収集されます。

統計情報	スコープ	説明
通信：受信要求数	接続、データベース、サーバ	クライアント／サーバ通信要求またはラウンドトリップの数を示します。[通信：受信パケット数] の統計値とは異なり、マルチパケット要求を 1 つの要求として数え、活性パケットを計数の対象から除外します。
通信：失敗した送信／秒	サーバ	基本のプロトコルがパケット送信に失敗した頻度を示します。
通信：総バッファ数	サーバ	ネットワークバッファの総数を示します。
通信：ユニークなクライアントアドレス	サーバ	データベースサーバに接続しているユニークなクライアントネットワークアドレスの数を示します。通常は接続しているクライアントマシン数であり、接続の合計数よりも少ない場合があります。

ディスク I/O の統計値

これらの統計値により、ディスクへのアクセス (読み込みと書き込み) 状況を取得し、ディスク I/O に使用されたアクティビティの負荷について全体的な情報を把握できます。

統計情報	スコープ	説明
ディスク：アクティブ I/O	データベース	データベースサーバによって発行され、まだ完了していない現在のファイル I/O 数を示します。
ディスク：アクティブ I/O の最大値	データベース	[ディスク：アクティブ I/O] の到達した最大値を示します。

ディスク読み込みの統計値

これらの統計値により、ディスクから情報を読み込むのに使用されたアクティビティの負荷とそのタイプを分析できます。

統計情報	スコープ	説明
ディスク読み込み：合計ページ数／秒	接続とデータベース	ページがファイルから読み込まれる速度を示します。
ディスク読み込み：アクティブ	データベース	データベースサーバによって発行され、まだ完了していない現在のファイル読み込み数を示します。
ディスク読み込み：インデックス内部／秒	接続とデータベース	インデックス内部ノードのページがディスクから読み込まれる頻度を示します。
ディスク読み込み：インデックスリーフ／秒	接続とデータベース	インデックスリーフページがディスクから読み込まれる頻度を示します。
ディスク読み込み：テーブル／秒	接続とデータベース	テーブルページがディスクから読み込まれる頻度を示します。
ディスク読み込み：アクティブの最大値	データベース	[ディスク読み込み：アクティブ] が到達した最大値を示します。
ディスク読み込み：ワークテーブル	接続とデータベース	ワークテーブルページがディスクから読み込まれる頻度を示します。

ディスク書き込みの統計値

これらの統計値により、ディスクへ情報を書き込むのに使用されたアクティビティの負荷とそのタイプを分析できます。

統計情報	スコープ	説明
ディスク書き込み：アクティブ	データベース	データベースサーバによって発行され、まだ完了していない現在のファイル書き込み数を示します。
ディスク書き込み：アクティブの最大値	データベース	[ディスク書き込み：アクティブ] が到達した最大値を示します。

統計情報	スコープ	説明
ディスク書き込み：コミットファイル／秒	データベース	データベースサーバが強制的に行うディスクキャッシュのフラッシュの頻度を示します。Windows プラットフォームではバッファなし (ダイレクト) の IO が使われるため、フラッシュは不要です。
ディスク書き込み：データベース拡張／秒	データベース	データベースファイルの拡張頻度 (ページ／秒) を示します。
ディスク書き込み：テンポラリ拡張／秒	データベース	テンポラリファイルの拡張頻度 (ページ／秒) を示します。
ディスク書き込み：ページ／秒	接続とデータベース	修正されたページがディスクに書き込まれる頻度を示します。
ディスク書き込み：トランザクションログ／秒	接続とデータベース	ページをトランザクションログに書き込む頻度を示します。
トランザクションロググループのコミット／秒	接続とデータベース	トランザクションログのコミットが要求されたときすでにログが書き込まれている (コミットはいつでも可能) 頻度を示します。

インデックスの統計値

これらの統計値により、インデックスの使用状態を分析できます。

統計情報	スコープ	説明
インデックス：追加／秒	接続とデータベース	インデックスにエントリが追加される頻度を示します。
インデックス：ルックアップ／秒	接続とデータベース	インデックスでエントリを検索する頻度を示します。
インデックス：完全比較／秒	接続とデータベース	インデックスのハッシュ値を超える比較が必要となる頻度を示します。

メモリ診断の統計値

これらの統計値により、データベースサーバによるメモリの使用状況を分析できます。

統計情報	スコープ	説明
キャッシュ：マルチページ割り当て	サーバ	マルチページ割り当ての数を示します。
キャッシュ：パニック	サーバ	キャッシュマネージャが割り付けるページの検索に失敗した回数を示します。
キャッシュ：スカベンジアクセス	サーバ	割り付けるページのスカベンジ中にアクセスしたページの数を示します。
キャッシュ：スカベンジ	サーバ	キャッシュマネージャが割り付けるページをスカベンジした回数を示します。
キャッシュページ：割り当て構造体	サーバ	データベースサーバのデータ構造に割り付けられたキャッシュページの数を示します。
キャッシュページ：ファイル	サーバ	データベースファイルから取得したデータを格納するキャッシュページの数を示します。
キャッシュページ：ファイルダーティ	サーバ	ダーティな (書き込みが必要な) キャッシュページの数を示します。
キャッシュページ：空き	サーバ	未使用のキャッシュページ数を示します。
キャッシュページ：固定	サーバ	現在再利用できないページ数を示します。
キャッシュ置換：合計ページ数／秒	サーバ	必要な別のページの領域を確保するためにデータベースページがキャッシュからパージされている頻度を示します。
ヒープ：カーバー	接続、データベース、サーバ	クエリ最適化などの短期的な目的に使用されたヒープの数を示します。

統計情報	スコープ	説明
ヒープ：クエリ処理	接続、データベース、サーバ	クエリ処理 (ハッシュ操作およびソート操作) に使用されるヒープの数を示します。
ヒープ：再配置可能	接続、データベース、サーバ	再配置可能なヒープの数を示します。
ヒープ：再配置可能ロック	接続、データベース、サーバ	キャッシュ内で現在ロックされている再配置可能なヒープの数を示します。
マップ物理メモリ／秒	サーバ	Address Windowing Extensions を使用して、データベースページのアドレス領域がキャッシュ内の物理メモリにマッピングされている頻度を示します。
メモリページ：カーバー	接続、データベース、サーバ	クエリ最適化などの短期的な目的に使用されたヒープページの数を示します。
メモリページ：固定カーソル	接続、データベース、サーバ	メモリ内でカーソルヒープを固定するために使用されているページ数を示します。
メモリページ：クエリ処理	接続、データベース、サーバ	クエリ処理 (ハッシュ操作およびソート操作) に使用されるキャッシュページの数を示します。
クエリメモリ：現在アクティブ	接続、データベース、サーバ	クエリメモリをアクティブに使用する現在の要求数を示します。
クエリメモリ：予測アクティブ	サーバ	安定した状態のデータベースサーバで、クエリメモリをアクティブに使用する要求の推定平均数を示します。
クエリメモリ：追加利用可能	サーバ	基本的なメモリ集約型の付与を超えて付与可能なメモリ量を示します。

統計情報	スコープ	説明
クエリメモリ：付与失敗数	接続、データベース、サーバ	要求がクエリメモリ待ちになり、取得できなかった合計回数を示します。
クエリメモリ：付与要求数	接続、データベース、サーバ	要求がクエリメモリを取得しようとした合計回数を示します。
クエリメモリ：付与待機数	接続、データベース、サーバ	要求がメモリ待ちになった合計回数を示します。
クエリメモリ：付与されたページ	接続、データベース、サーバ	要求に現在付与されているページ数を示します。
クエリメモリ：待機中の要求数	接続、データベース、サーバ	クエリメモリ待ちになっている現在の要求数を示します。

メモリページの統計値

これらの統計値により、データベースサーバが使用しているメモリ量とその目的を分析できます。

統計情報	スコープ	説明
メモリページ：ロックテーブル	データベース	ロック情報の保持に使用されているページの数を示します。
メモリページ：ロックヒープ	サーバ	キャッシュ内でロックされているヒープページの数を示します。
メモリページ：メインヒープ	サーバ	グローバルデータベースサーバのデータ構造に使用されているページ数を示します。
メモリページ：マップページ	データベース	ロックテーブル、頻度テーブル、テーブルレイアウトへのアクセスに使用されたマップページの数を示します。
メモリページ：プロシージャ定義	データベース	プロシージャに使用された再配置可能なヒープページ数を示します。

統計情報	スコープ	説明
メモリページ：再配置可能	データベース	再配置可能なヒープ (カーソル、文、プロシージャ、トリガ、ビューなど) で使用されるページの数を示します。
メモリページ：再配置／秒	データベース	再配置可能なヒープページがテンポラリファイルから読み出される頻度を示します。
メモリページ：ロールバックログ	接続とデータベース	ロールバックログのページ数を示します。
メモリページ：トリガ定義	データベース	トリガで使用される再配置可能なヒープページの数を示します。
メモリページ：ビュー定義	データベース	ビューで使用される再配置可能なヒープページの数を示します。

要求の統計値

これらの統計値により、クライアントアプリケーションの要求への応答に使用されたデータベースサーバのアクティビティを分析できます。

統計情報	スコープ	説明
カーソル	接続、データベース、サーバ	現在データベースサーバが保持している宣言されたカーソルの数を示します。
開いているカーソル	接続、データベース、サーバ	現在データベースサーバが保持しているオープンカーソルの数を示します。
ロック数	接続とデータベース	ロックの数を示します。
要求／秒	サーバ	データベースサーバが新しい要求を処理するか、既存の要求の処理を続行できる状態になる頻度を示します。
要求：アクティブ	サーバ	現在要求を処理中のデータベースサーバのスレッド数を示します。

統計情報	スコープ	説明
タスク：交換	サーバ	クエリの並列実行に現在使用されているデータベースサーバのスレッド数を示します。
要求：未スケジュール	サーバ	使用できるデータベースサーバのスレッドの解放を待ってキューイングされている要求の数を示します。
スナップショット数	接続とデータベース	アクティブなスナップショットの数を示します。
文キャッシュヒット	接続、データベース、サーバ	クライアントによってキャッシュされた文の準備が、データベースサーバで再利用されている頻度を示します。
文キャッシュミス	接続、データベース、サーバ	クライアントによってキャッシュされた文の準備をデータベースサーバで再度準備する必要がある頻度を示します。
文の準備	接続とデータベース	データベースサーバにより文の準備が処理される頻度を示します。
文	接続、データベース、サーバ	現在データベースサーバが保持している準備文の数を示します。
トランザクションのコミット	接続、データベース、サーバ	コミット要求が処理される頻度を示します。
トランザクションのロールバック	接続、データベース、サーバ	ロールバック要求が処理される頻度を示します。

ユーザ定義統計情報

これらの統計情報には、アプリケーションによって追跡される値に関連するティビティが記述されます。

統計情報	スコープ	説明
ユーザ定義変化率：カウンタ 1	接続、データベース、サーバ	ユーザ定義カウンタ値の経時的変化率を表示します。

統計情報	スコープ	説明
ユーザ定義変化率：カウンタ 2	接続、データベース、サーバ	ユーザ定義カウンタ値の経時的変化率を表示します。
ユーザ定義変化率：カウンタ 3	接続、データベース、サーバ	ユーザ定義カウンタ値の経時的変化率を表示します。
ユーザ定義変化率：カウンタ 4	接続、データベース、サーバ	ユーザ定義カウンタ値の経時的変化率を表示します。
ユーザ定義変化率：カウンタ 5	接続、データベース、サーバ	ユーザ定義カウンタ値の経時的変化率を表示します。
ユーザ定義未加工データ：カウンタ 1	接続、データベース、サーバ	現在のユーザ定義カウンタ値を表示します。
ユーザ定義未加工データ：カウンタ 2	接続、データベース、サーバ	現在のユーザ定義カウンタ値を表示します。
ユーザ定義未加工データ：カウンタ 3	接続、データベース、サーバ	現在のユーザ定義カウンタ値を表示します。
ユーザ定義未加工データ：カウンタ 4	接続、データベース、サーバ	現在のユーザ定義カウンタ値を表示します。
ユーザ定義未加工データ：カウンタ 5	接続、データベース、サーバ	現在のユーザ定義カウンタ値を表示します。

参照

- 「ユーザ定義プロパティ」『SQL Anywhere サーバ データベース管理』

その他の統計値

統計情報	スコープ	説明
使用可能 IO	サーバ	現在使用可能な I/O 制御ブロックの数を示します。
接続数	データベース	このデータベースとの接続の数を示します。
メインヒープバイト	サーバ	グローバルデータベースサーバのデータ構造に使用されているバイト数を示します。

統計情報	スコープ	説明
クエリ：プランキャッシュページ	接続とデータベース	実行プランの保存に使用されているキャッシュページの数を示します。
クエリ：メモリ不足時方式	接続とデータベース	メモリ不足状態のため、データベースサーバがその実行中に実行プランを変更した回数を示します。
クエリ：ローの実体化／秒	接続とデータベース	クエリ処理中にローがワークテーブルに書き込まれる割合を示します。
要求：GET DATA／秒	接続とデータベース	接続で GET DATA 要求が発行されている頻度を示します。
テンポラリテーブルページ	接続とデータベース	テンポラリテーブルで使用されるテンポラリファイルのページ数を示します。
バージョンストアページ	データベース	スナップショットアイソレーションが有効な場合にローバージョンストアで現在使用されているテンポラリファイルのページ数を示します。

パフォーマンス向上のヒント

サーバ関連のパフォーマンスに関連するヒント

- 「ヒント：パフォーマンス向上のためのキャッシュの使用」212 ページ
- 「ヒント：同時実行性の問題点の確認」219 ページ
- 「ヒント：オプティマイザの目標の選択」220 ページ
- 「ヒント：カラム統計の更新」221 ページ
- 「ヒント：インデックスの有効な使用」223 ページ
- 「ヒント：インデックスのパフォーマンスの向上」224 ページ
- 「ヒント：混合負荷または OLAP 負荷の最適化」224 ページ
- 「ヒント：クエリ結果の効果的なソートの使用」224 ページ
- 「ヒント：正しいカーソルタイプの指定」225 ページ
- 「ヒント：明示的な選択性推定の提供を控える」225 ページ
- 「ヒント：マテリアライズドビューを使ったクエリのパフォーマンス改善」225 ページ
- 「ヒント：テーブル検証時の WITH EXPRESS CHECK オプションの使用」229 ページ
- 「ヒント：クエリ処理におけるワークテーブルの使用 (All-rows 最適化ゴールの使用)」230 ページ

アプリケーション関連のパフォーマンスに関連するヒント

- 「ヒント：効率的な SQL クエリの構築」231 ページ
- 「ヒント：クエリのパフォーマンスのモニタ」237 ページ
- 「ヒント：コストの高いユーザ定義関数の削減」238 ページ
- 「ヒント：オートコミットモードをオフにする」238 ページ
- 「ヒント：インメモリモードの使用」238 ページ

データベース関連のパフォーマンスに関連するヒント

- 「ヒント：常にトランザクションログを使用する」239 ページ
- 「ヒント：コミットの遅延の使用」239 ページ
- 「ヒント：小さいテーブルに関する統計収集」240 ページ
- 「ヒント：断片化の削減」240 ページ
- 「ヒント：テーブル構造の正規化」246 ページ
- 「ヒント：カスケードされた参照アクションを最小限に抑える」247 ページ
- 「ヒント：制約の宣言」247 ページ
- 「ヒント：異なるファイルの異なるデバイスへの配置」247 ページ
- 「ヒント：データベースの再構築」248 ページ
- 「ヒント：キーを使ったクエリのパフォーマンス改善」249 ページ
- 「ヒント：プライマリキー幅の縮小」249 ページ
- 「ヒント：テーブル幅の縮小」250 ページ
- 「ヒント：テーブル内のカラムの順序の確認」250 ページ
- 「ヒント：コストの高いトリガを置き換える」250 ページ
- 「ヒント：適切なページサイズの使用」250 ページ
- 「ヒント：オートインクリメントを使用したプライマリキーの作成」252 ページ
- 「ヒント：適切なデータ型の使用」252 ページ
- 「ヒント：バルクオペレーション方法の使用」253 ページ
- 「ヒント：リソースガバナを使用する」253 ページ

通信関連のパフォーマンスに関連するヒント

- 「ヒント：クライアントとサーバとの間の要求数の削減」254 ページ
- 「ヒント：圧縮の慎重な使用」254 ページ
- 「ヒント：パケットサイズを変更してパフォーマンスを向上させる」255 ページ

ヒント：パフォーマンス向上のためのキャッシュの使用

キャッシュとは、メモリの領域で、データベースサーバがデータベースページを格納して繰り返し高速にアクセスするために使用します。キャッシュでアクセスできるページが増えると、データベースサーバによるディスクからのデータの読み込み (遅い操作) の回数が減ります。そのため、キャッシュサイズがパフォーマンスにおける重要な要因になることがよくあります。

SQL Anywhere では動的キャッシュサイズ決定がサポートされており、システムを全体としてモニタリングすることにより、キャッシュサイズが適切かつ自動的に調整されます。ただし、データベースの起動時にデータベースサーバのコマンドラインで -c オプションを使用して、データベースキャッシュのサイズを制御することもできます。

データベースサーバメッセージウィンドウには起動時のキャッシュサイズが表示されますが、次の文を使用して現在のキャッシュサイズを取得することもできます。

```
SELECT PROPERTY( 'CurrentCacheSize' );
```

暗号化されたデータベースには、I/O 操作を最小限に抑えるのに十分なキャッシュが必要です。暗号化や複合化は操作ごとに実行する必要があるため、暗号化されたデータベース上では、暗号化されていないデータベース上よりも I/O 操作の負荷が高くなるためです。

参照

- 「動的キャッシュサイズ決定」216 ページ
- 「-c データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-ca データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-ch データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-cl データベースサーバオプション」『SQL Anywhere サーバ データベース管理』

キャッシュとメモリガバナー

SQL Anywhere データベースサーバは、キャッシュ (バッファプール) を使用し、データベースページのイメージをメモリに一時的に格納します。これらのページは通常、テーブルページとインデックスページです。ただし、SQL Anywhere データベースには他の種類の物理ページも格納されます。これらのページに加えて、データベースサーバは他に 2 つのメモリプールのキャッシュを使用します。これらのプールの 1 つは仮想メモリです。接続、文、カーソルを表す構造など、データベースサーバのデータ構造に使用されます。2 番目のプールは、クエリメモリの仮想ストレージとして使用されるキャッシュページで構成されます。

クエリの実行が効率的に動作するには、メモリが必要です。SQL Anywhere はメモリガバナーを使用して、各文がクエリの実行に使用できるクエリメモリの量を決定します。メモリガバナーはクエリメモリプールを文に割り付け、負荷を効率的に実行できるようにします。

メモリガバナーは選択されたページ数を各文に付与し、文はメモリを大量に消費するクエリ処理にそのページを使用できます。クエリプロセッサがページを使用するまでは、引き続きクエリメモリプール内のメモリを他の目的 (テーブルのバッファリングやインデックスページなど) にも使用できます。クエリメモリを使用し、メモリを大量に消費するクエリ処理には、ハッシュ DISTINCT、ハッシュ GROUP BY、ハッシュジョインなどのあらゆるハッシュベースの演算子や、ソート演算子、ウィンドウ演算子があります。

次の設定、演算子、統計を使用して、メモリガバナーによるキャッシュの使用方法を理解 (および制御) します。

- **QueryMemMaxUseful グラフィカルプラン演算子**　文が実行を開始すると、メモリガバナーはオプティマイザの推定を使用して、文が使用できるメモリ容量を決定します。この推定は、グラフィカルなプランに QueryMemMaxUseful として表示されます。

- **QueryMemActiveMax サーバプロパティ**　メモリガバナーは、メモリを大量に消費する要求の同時実行数を制限します。この最大値は、データベースサーバを実行するコンピュータのパフォーマンス特性に応じて選択されます。また、この制限は、サーバプロパティ QueryMemActiveMax に表示されます。

- **QueryMemActiveEst パフォーマンスモニタの統計**　メモリガバナーは、メモリを大量に消費する要求の同時実行数の推定を継続します。この推定はデータベースサーバプロパティおよびパフォーマンスモニタの統計 QueryMemActiveEst に使用できます。

- **query_mem_timeout データベースオプション**　メモリを大量に消費する文の実行を開始したとき、メモリを大量に消費する要求の最大同時実行数にすでに達している場合は、いずれかの実行中の要求が割り付けられたメモリを解放するまで入力された文は待機します。query_mem_timeout データベースオプションは、入力された要求にメモリが付与されるまで待つ時間を制御します。デフォルト設定 -1 を使用すると、要求はデータベースサーバで定義された期間待機します。待機後、使用できるメモリが付与されない場合は文のアクセスプランは少ないメモリ容量で実行されます。これによって実行が遅くなる場合がありますが、プラン内のメモリを大量に消費する物理演算子に対して実行時のメモリ量が少なくなる方式を使用できる場合は、その方式が使用されます。

- **QueryMemGrantWaiting サーバプロパティおよびパフォーマンスモニタの統計**　データベースサーバプロパティおよびパフォーマンスモニタの統計 QueryMemGrantWaiting には、メモリ要求が付与されるのを待機している現在の要求数が表示されます。

- **QueryMemGrantWaited サーバプロパティおよびパフォーマンスモニタの統計**　データベースサーバプロパティおよびパフォーマンスモニタの統計 QueryMemGrantWaited には、メモリ要求が付与されるまでに要求が待機する必要があった合計回数が表示されます。

- **QueryMemNeedsGrant グラフィカルプラン演算子**　グラフィカルなプランでは、QueryMemNeedsGrant には、要求が単純な要求 (メモリの付与が不要) かメモリを大量に消費する (メモリの付与が必要) かをメモリガバナーが検討した結果が表示されます。メモリガバナーによってメモリの付与が不要と分類された要求は、ただちに実行されます。それ以外の場合、要求はクエリメモリプールの一部の使用を求めます。

- **QueryMemLikelyGrant グラフィカルプラン演算子**　グラフィカルなプランでは、「QueryMemLikelyGrant」 には、要求を実行するために付与されることが予想される推定ページ数が表示されます。

参照

- 「統計情報付きのグラフィカルなプラン」335 ページ
- QueryMemActiveMax サーバプロパティ『SQL Anywhere サーバ データベース管理』
- QueryMemPages サーバプロパティ『SQL Anywhere サーバ データベース管理』
- QueryMemPercentOfCache サーバプロパティ『SQL Anywhere サーバ データベース管理』
- 「query_mem_timeout オプション」『SQL Anywhere サーバ データベース管理』
- 「マルチプログラミングレベルのデータベースサーバ設定」『SQL Anywhere サーバ データベース管理』
- 「-ch データベースサーバオプション」『SQL Anywhere サーバ データベース管理』

キャッシュとオプティマイザ

たとえば、カーソルの中身を保持するために追加メモリを予約すると、コストがかかります。キャッシュが一杯であれば、1 ページまたは複数のページをディスクに書き込み、新しいページの領域を確保しなければなりません。後続の操作を完了するために、数ページの再読み込みが必要になることもあります。この状況で、SQL Anywhere は、追加バッファキャッシュオーバヘッドを必要とする実行プランに高いコストがかかるようにしています。このコストにより、オプティマイザはワークテーブルを使用するプランを選択しないようになっています。ただし、パフォーマンスが向上する場合には、オプティマイザは注意しながらメモリを使用します。たとえ

ば、サブクエリの結果がクエリ処理中に繰り返し必要になる場合、これらの結果をキャッシュします。

キャッシュメモリ使用の制限

キャッシュの初期サイズ、最小サイズ、最大サイズはすべて、データベースサーバのコマンドラインから制御できます。

- **初期キャッシュサイズ**　データベースサーバの初期キャッシュサイズを指定するには、データベースサーバの -c オプションを使用します。-c オプションを指定しない場合、データベースサーバによって初期キャッシュメモリの割り付けサイズが計算されます。

- **最大キャッシュサイズ**　最大キャッシュサイズを制御するには、データベースサーバの -ch オプションを指定します。デフォルトは、使用しているコンピュータの物理メモリによって異なるヒューリスティックに基づいています。Windows Mobile では、デフォルトの最大キャッシュサイズは、使用可能なプログラムメモリから 4 MB を引いた値です。他の UNIX 以外のコンピュータでは、最大キャッシュサイズは、最大キャッシュサイズとコンピュータの物理メモリの 90% のうち、いずれか低い方になります。UNIX では、デフォルトの最大キャッシュサイズは次のように計算されます。

  ○32 ビットの UNIX プラットフォームでは、物理メモリ量の合計の 90% または 1,834,880 KB のいずれか小さい方です。

  ○64 ビットの UNIX プラットフォームでは、物理メモリ量の合計の 90% または 8,589,672,320 KB のいずれか小さい方です。

- **最小キャッシュサイズ**　最小キャッシュサイズを制御するには、データベースサーバの -cl サーバオプションを指定します。Windows Mobile を除き、デフォルトでは、最小キャッシュサイズは初期キャッシュサイズと同じです。Windows Mobile では、デフォルトの最小キャッシュサイズは 600 KB です。

  -cl を指定しないで -c サーバオプションを指定した場合、最小キャッシュサイズは -c サーバオプションで指定された初期キャッシュサイズに設定されます。

  -c または -cl サーバオプションを設定しない場合、必要に応じてキャッシュを縮小できるように、最小キャッシュサイズは非常に低いハードコード定数値に設定されます。この値は、Windows では 2 MB、UNIX では 8 MB、Windows Mobile では 600 KB です。

初期キャッシュサイズまたは最小キャッシュサイズを最大キャッシュサイズの 8 分の 1 より小さい値に設定しようとすると、最大キャッシュサイズに応じて初期および最小のキャッシュサイズが自動的に増大します。

また、動的キャッシュサイズ決定を無効にするには、-ca 0 サーバオプションを使用します。

次のデータベースサーバプロパティは、データベースサーバのキャッシュに関する情報を返します。

- **CurrentCacheSize**　現在のキャッシュサイズ (キロバイト単位) を返す。

- **MinCacheSize** 許容最小キャッシュサイズ (キロバイト単位) を返す。
- **MaxCacheSize** 許容最大キャッシュサイズ (キロバイト単位) を返す。
- **PeakCacheSize** 現在のセッションでキャッシュが到達した最大値 (キロバイト単位) を返す。

参照

- キャッシュサイズ『SQL Anywhere サーバ データベース管理』
- 「データベースサーバプロパティの一覧」『SQL Anywhere サーバ データベース管理』
- 「-c データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-ca データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-ch データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-cl データベースサーバオプション」『SQL Anywhere サーバ データベース管理』

動的キャッシュサイズ決定

SQL Anywhere を使用して、データベースサーバの実行中にデータベースキャッシュを自動的にリサイズできます。完全な「動的キャッシュサイズ決定」では、多くのキャッシュを使用することでデータベースサーバのパフォーマンスが向上する場合は、使用可能なメモリがあるかぎりキャッシュサイズは大きくなり、他のアプリケーションがキャッシュメモリを必要とするときはキャッシュサイズは小さくなります。動的キャッシュサイズ決定の効果は、オペレーティングシステムや、利用可能な物理メモリの量によってもさまざまに異なります。

キャッシュの必要量は、動的キャッシュサイズ決定によって通常 1 分ごとに評価されます。ただし、新しいデータベースが起動されたときやファイル量が大幅に増加した場合、30 秒間は 5 秒ごとに評価されるよう実行間隔が狭まることがあります。最初の 30 秒間が経過すると、評価の実行頻度は 1 分間隔に戻ります。ファイル増加量が前回に評価の実行間隔を狭めたときの増加量よりも 8 分の 1 多い場合、またはファイル量がデータベース起動時より 8 分の 1 増加している場合に、これは大幅な増加であると判断されます。

動的キャッシュサイズ決定を使用する場合は、明示的にデータベースキャッシュを設定する必要はありません。

初期キャッシュサイズまたは最小キャッシュサイズを最大キャッシュサイズの 8 分の 1 より小さい値に設定しようとすると、最大キャッシュサイズに応じて初期および最小のキャッシュサイズが自動的に増大します。

参照

- 「-ca データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-ch データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-cl データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- キャッシュサイズ『SQL Anywhere サーバ データベース管理』

動的キャッシュサイズ決定 (Windows)

Windows と Windows Mobile では、データベースサーバがキャッシュとオペレーションの統計を 1 分ごとに評価し、最適なキャッシュサイズを計算します。データベースサーバは、ターゲット

のキャッシュサイズを計算します。このキャッシュは、現在使用されていない物理メモリの約 5 MB をシステムが使用できるように残し、それ以外をすべて使用します。ターゲットキャッシュサイズが、明示的または暗黙的に指定された最小キャッシュサイズより小さくなることはありません。ターゲットのキャッシュサイズは、明示的または暗黙的に指定された最大キャッシュサイズ、またはすべてのオープンデータベースとテンポラリファイル合計サイズにメインヒープのサイズを加えたものを超えることはありません。

キャッシュサイズの変動を防ぐために、データベースサーバはキャッシュサイズを段階的に増やします。すぐにターゲット値に調整するのではなく、調整するたびに現在のサイズとターゲットサイズの差の 75% ずつキャッシュサイズを修正します。

動的キャッシュサイズ決定 (UNIX)

UNIX では、データベースサーバはスワップ領域とメモリを使用してキャッシュサイズを管理します。スワップ領域はシステムワイドなリソースで、ほとんどの UNIX オペレーティングシステムにあります。この項では、メモリとスワップ領域の合計を「システムリソース」と呼びます。詳細については、使用しているオペレーティングシステムのマニュアルを参照してください。

起動時に、データベースは指定した最大キャッシュサイズをシステムリソースから割り付けます。この一部をメモリ (初期キャッシュサイズ) にロードし、残りをスワップ領域として残します。

データベースサーバが使用するシステムリソースの総量は、データベースサーバが停止するまで一定です。ただし、メモリにロードされる比率は変わります。データベースサーバは、1 分ごとにキャッシュとオペレーションの統計を評価します。データベースサーバがビジーでメモリが必要になると、キャッシュページをスワップ領域からメモリに移します。システム内の他の処理がメモリを必要とした場合、データベースサーバがキャッシュページをメモリからスワップ領域に移すことがあります。

初期キャッシュサイズ

デフォルトでは、初期キャッシュサイズは使用可能なシステムリソースに基づいたヒューリスティックを使用して割り当てられます。初期キャッシュサイズは、データベースサイズの総量の 1.1 倍より常に小さくなります。

初期キャッシュサイズが使用可能なシステムリソースの 4 分の 3 より大きい場合、データベースサーバはメモリの不足を通知するエラーで終了します。

初期キャッシュサイズは、-c オプションを使用して変更できます。

最大キャッシュサイズ

最大キャッシュサイズは、コンピュータの使用可能なシステムリソースより小さくしてください。デフォルトでは、最大キャッシュサイズはコンピュータの使用可能なシステムリソースと物理メモリ量の合計に基づいたヒューリスティックを使用して割り当てられます。キャッシュサイズは、明示的または暗黙的に指定された最大キャッシュサイズ、またはすべてのオープンデータベースとテンポラリファイル合計サイズにメインヒープのサイズを加えたものを超えることはありません。

使用可能なシステムリソースより大きい最大キャッシュサイズを指定すると、データベースサーバはメモリの不足を通知するエラーで終了します。使用可能なメモリより大きい最大キャッ

シュサイズを指定すると、データベースサーバはパフォーマンス低下の警告を出しますが、終了はしません。

データベースサーバはすべての最大キャッシュサイズをシステムリソースから割り付け、終了まで解放しません。他のアプリケーション用に領域を残しながら SQL Anywhere のパフォーマンスも低下させないように最大キャッシュサイズを選択してください。デフォルトの最大キャッシュサイズを求める式は、このバランスを考慮に入れたヒューリスティックを使用しています。値を調整する必要があるのは、デフォルト値が、使用しているシステムに適さないときのみです。

-ch サーバオプションを使用して最大キャッシュサイズを設定し、自動キャッシュ増加機能を制限できます。

最小キャッシュサイズ

-c オプションを指定した場合、最小キャッシュサイズは初期キャッシュサイズと同じです。-c オプションを指定しない場合は、UNIX 上での最小キャッシュサイズは 8 MB です。

-cl サーバオプションを使用して、最小キャッシュサイズを調整できます。

初期キャッシュサイズまたは最小キャッシュサイズを最大キャッシュサイズの 8 分の 1 より小さい値に設定しようとすると、最大キャッシュサイズに応じて初期および最小のキャッシュサイズが自動的に増大します。

参照

- 「-c データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-ch データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-cl データベースサーバオプション」『SQL Anywhere サーバ データベース管理』

キャッシュサイズをモニタリングする統計

次の統計が、Windows パフォーマンスモニタとデータベースのプロパティ関数にあります。

- **CurrentCacheSize**　キロバイト単位の現在のキャッシュサイズ
- **MinCacheSize**　キロバイト単位の最小許容キャッシュサイズ
- **MaxCacheSize**　キロバイト単位の最大許容キャッシュサイズ
- **PeakCacheSize**　キロバイト単位のピークキャッシュサイズ

参照

- CurrentCacheSize サーバプロパティ『SQL Anywhere サーバ データベース管理』
- MinCacheSize サーバプロパティ『SQL Anywhere サーバ データベース管理』
- MaxCacheSize サーバプロパティ『SQL Anywhere サーバ データベース管理』
- PeakCacheSize サーバプロパティ『SQL Anywhere サーバ データベース管理』
- 「データベースサーバプロパティの一覧」『SQL Anywhere サーバ データベース管理』
- 「データベースのパフォーマンスのモニタ」191 ページ

キャッシュウォーミング

キャッシュウォーミングは、データベースに対して実行される初期クエリの実行時間を減らすのに役立つように設計されています。これは、データベースが最後に起動したときに参照していたデータベースサーバのキャッシュとデータベースページに対するキャッシュを事前にロードすることで実行されます。キャッシュを準備することにより、データベースが起動するたびに同一または類似のクエリがデータベースに対して実行される場合にのみ、パフォーマンスを向上させることができます。ただし、データベースの起動時に実行された文が、前回キャッシュの収集をオンにしてデータベースを起動したときのものと異なる場合は、キャッシュウォーミングによってパフォーマンスが向上することはありません。

キャッシュウォーミング設定をデータベースサーバのコマンドラインで制御できます。データベースが起動してキャッシュウォーミングがオンになると、データベースページの収集とキャッシュの再ロード (準備) という 2 種類のアクティビティが実行されます。

参照されたデータベースページの収集は、-cc データベースサーバオプションで制御され、デフォルトでオンになっています。データベースページの収集がオンになると、データベースサーバは、収集ページ数が最大数に到達するまで (値はキャッシュサイズとデータベースサイズに基づく)、収集速度が最小しきい値を下回るまで、またはデータベースが停止するまで、データベース起動時に要求されたすべてのデータベースページを追跡し続けます。データベースサーバは、収集最大ページ数と収集しきい値を制御します。いったん収集が完了すると、参照されたページはデータベースに記録されるので、次回のデータベース起動時にキャッシュの準備に使用できます。

キャッシュウォーミング (再ロード) はデフォルトでオンになっていて、-cr データベースサーバオプションで制御されます。キャッシュを準備するために、データベースサーバはデータベースにすでに記録された収集ページがあるかどうかをチェックします。ある場合、データベースサーバが対応するページをキャッシュにロードします。データベースはキャッシュがページをロードしている間、引き続き要求を処理できますが、大量の I/O アクティビティがデータベースで検出された場合は準備が停止することがあります。この場合、キャッシュに再ロードされるページセットに含まれないページにアクセスするクエリのパフォーマンスの低下を防ぐために、キャッシュウォーミングは停止します。キャッシュウォーミングに関する情報をデータベースサーバメッセージウィンドウに表示する場合は、-cv オプションを指定できます。

参照

- 「-cc データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-cr データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「-cv データベースサーバオプション」『SQL Anywhere サーバ データベース管理』

ヒント：同時実行性の問題点の確認

データベースサーバがトランザクションを処理するとき、テーブルローを 1 つまたは複数ロックできます。ロックは他のトランザクションが同時にアクセスすることを防止し、データベースに格納されている情報の信頼性を維持します。また、更新中の情報を識別して、結果クエリの精度を高めます。

データベースサーバは自動的にこれらのロックを設定するので、明示的な指示は必要ありません。データベースサーバは、トランザクションが獲得したすべてのロックを、そのトランザク

ションが完了するまで保持します。ローにアクセスしているトランザクションは、ロックを保持しているといえます。ロックの種類により、他のトランザクションのそのローへのアクセスは限定されるか、まったくできなくなります。

頻繁に 1 つまたは複数のローに複数のユーザが同時にアクセスすると、パフォーマンスは低下することがあります。ロックが問題になっていると考えられる場合は、sa_locks プロシージャを使用して、データベースのロックに関する情報を入手できます。

ロックの問題が識別された場合は、関連する接続プロセスに関する情報を AppInfo 接続プロパティを使って表示できます。

参照

- 「sa_locks システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「接続プロパティの一覧」『SQL Anywhere サーバ データベース管理』

ヒント：オプティマイザの目標の選択

optimization_goal オプションを使用すると、SQL Anywhere において SQL 文を応答時間 (First-row) に対して最適化するか、リソースの総消費量 (All-rows) に対して最適化するかを制御できます。すなわち、クエリ処理の最適化の目的を、最初のローを迅速に返すことに設定するか、または完全な結果セットを返すコストを最小限に抑えることに設定できます。

このオプションを First-row に設定すると、SQL Anywhere は、クエリの結果の最初のローをフェッチするまでの時間を短縮するアクセスプランを選択します。この場合、検索にかかる合計時間は長くなることがあります。また、通常オプティマイザでは、可能であれば結果の実体化を必要とするアクセスプランは使用しないで、最初のローを返すまでの時間を短縮します。この設定では、たとえばオプティマイザは、明示的なソートの操作を必要とするアクセスプランではなく、クエリの ORDER BY 句を満たすインデックスを使用するアクセスプランを採用します。

オプティマイザが特定の文に使用する最適化ゴールは、次の規則に従って決まります。

- メインクエリブロックの FROM 句で、テーブルヒントが FASTFIRSTROW に設定されたテーブルが存在する場合、文は First-row 最適化ゴールを使用して最適化されます。

- 文の OPTION 句で optimization_goal オプションが設定されている場合、文はその設定を使用して最適化されます。

- それ以外の場合、オプティマイザは現在の optimization_goal オプションの設定を使用します。

最適化ゴールが First-row であっても、最初のローをすぐに返すことができるプランをオプティマイザが見つけられない可能性があります。たとえば、DISTINCT、GROUP BY、または ORDER BY 句が存在するために実体化が必要な文で、必要な順序を実現するためのインデックスが存在しない場合は、All-rows 目標で最適化されます。

このオプションを All-rows (デフォルト) に設定すると、SQL Anywhere のクエリは最適化され、予測される合計検索時間が最短になるアクセスプランを選択します。PowerBuilder DataWindow アプリケーションなど、処理の前に結果セット全体が必要になるアプリケーションでは、optimization_goal を All-rows に設定するのが適切です。

参照

- 「optimization_goal オプション」『SQL Anywhere サーバ データベース管理』
- 「FROM 句」『SQL Anywhere サーバ SQL リファレンス』

次の SQL 文の OPTION 句も参照してください。

- 「DELETE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「INSERT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「MERGE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「UPDATE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「UNION 文」『SQL Anywhere サーバ SQL リファレンス』

ヒント：カラム統計の更新

カラム統計は、データベースのシステムテーブル ISYSCOLSTAT に永久的に格納されます。オプティマイザのパフォーマンスを向上させ続けられるように、データベースサーバは、SELECT、INSERT、UPDATE、DELETE の各文の処理中に、自動的にカラム統計を更新します。これには、単一のローに対する文も含まれます。これは、テーブルやカラムを参照する述部に一致するローの数をモニタリングし、その数を推定されたローの数と比較し、必要に応じて既存の統計を更新することで行います。

利用可能なカラム統計の精度が高くなると、それに従ってオプティマイザがより適切に推定を計算できるため、以降のクエリのパフォーマンスが向上します。

データベースオプションを使用して、カラム統計を更新するかどうかを設定できます。update_statistics データベースオプションは、クエリの実行中にカラム統計を更新するかどうかを指定します。collect_statistics_on_dml_updates データベースオプションは、LOAD、INSERT、DELETE、UPDATE などのデータを変更する DML 文の実行中に統計を更新するかどうかを指定します。

統計が現在のカラムの値を正確に反映していないためにパフォーマンスが悪いと考えられる場合は、CREATE STATISTICS 文や DROP STATISTICS 文を実行します。CREATE STATISTICS は古い統計を削除して新しい統計を作成し、DROP STATISTICS は古い統計を削除するだけです。

CREATE INDEX 文を実行すると、インデックスの統計が自動的に作成されます。

LOAD TABLE 文を実行すると、テーブルの統計が自動的に作成されます。

参照

- 「SYSCOLSTAT システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE STATISTICS 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER STATISTICS 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DROP STATISTICS 文」『SQL Anywhere サーバ SQL リファレンス』
- 「update_statistics オプション」『SQL Anywhere サーバ データベース管理』
- 「collect_statistics_on_dml_updates オプション」『SQL Anywhere サーバ データベース管理』

統計ガバナーが統計を保守する方法

統計ガバナーでは、クエリが実行されるときにカラム統計を自動調整する以外に、オプティマイザ統計の正常性と使用状況もモニタします。統計ガバナーでは、データベース内の各統計の正常性と有用性を自動的に評価し、統計を自己モニタして自己回復できるように、必要な保守を実行します。統計の保守はバックグラウンドで実行され、データベースサーバのパフォーマンスに対する大幅な負荷は発生しません。

統計ガバナーによって、以下のタスクが実行されます。

- クエリのフィードバックから取得した推定エラーと統計の使用状況の記録
- 精度の低い統計の修正または再作成
- 効率的に保守を行えない統計の自動保守の停止
- 有用な可能性のある統計の作成
- 使用されていない統計の削除

update_statistics オプションは、指定した接続が統計ガバナーにクエリフィードバックを送信できるかどうかを制御します。このオプションが Off に設定されている場合、統計ガバナーは指定した接続からクエリフィードバックを受け取りません。ただし、統計ガバナーはその他の接続からクエリフィードバックを受け取り、統計の保守操作を実行できます。

統計ガバナーは、統計の正常性と使用状況に基づいて、いつ統計を修正または作成するかを決定します。統計を修正または作成するには、クエリの実行中に統計情報を収集するか、または統計クリーナーと呼ばれる独立したプロセスを実行します。統計ガバナーを無効にすることなく、統計クリーナーを無効にするには、sa_server_option システムプロシージャの StatisticsCleaner オプションを使用します。ただし、統計クリーナーをオフにすると、クエリの実行時にのみ統計が作成または修正されます。

サーバの負荷を減らすために、統計ガバナーは修正が困難な統計や使用されない統計の保守を停止します。短期間に何回も修正され、精度の低い推定を返す統計は削除され、30 日間保守されません。削除された統計は 30 日後に再作成され、定期的な保守が再開されます。sa_server_option システムプロシージャの DropBadStatistics オプションを使用すると、この機能を無効にできます。最近 90 日間に使用されなかった統計も削除されます。この機能を無効にするには、sa_server_option システムプロシージャの DropUnusedStatistics オプションを使用します。CREATE STATISTICS、DROP STATISTICS、または ALTER STATISTICS 文を使用すると、統計の保守をいつでも再開できます。

統計はメモリにロードされているテーブルに対してのみモニタされます。これらの統計は 30 分ごとにフラッシュされます。フラッシュする間に、統計の正常性と使用状況が検査され、統計ガバナーが統計の保守を実行します。統計のステータス情報 (正常性、使用状況、統計の更新または削除に関する情報など) はセッション間で保持されません。データベースサーバが停止されると、ステータス情報はリセットされます。

参照

- 「CREATE STATISTICS 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER STATISTICS 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DROP STATISTICS 文」『SQL Anywhere サーバ SQL リファレンス』
- 「update_statistics オプション」『SQL Anywhere サーバ データベース管理』
- 「sa_server_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

ヒント：インデックスの有効な使用

通常、SQL Anywhere はインデックスを使用して探索条件を評価できます。インデックスを使用することで、オプティマイザはデータへのアクセスをスピードアップし、ベーステーブルから読み込んで処理する情報量を減らします。たとえば、クエリに WHERE column-name=value という探索条件があり、カラムにインデックスが存在する場合、インデックススキャンを使って、探索条件を満たすテーブルのローだけを読み込むことができます。テーブルをジョインする場合にインデックスがあれば、パフォーマンスを大幅に向上させることもできます。

クエリを実行するとき、SQL Anywhere は各テーブルへのアクセス方法を選択します。データベースサーバが適切なインデックスを見つけられないときは、テーブルを順にスキャンしなければならず、時間を要します。

たとえば、大規模なデータベースから複数の従業員を検索する必要があるのに、姓か名前のどちらかしかわからないとします。インデックスがない場合、SQL Anywhere はテーブル全体をスキャンします。しかし、姓が先に入力されているインデックスと名前が先に入力されているインデックスの 2 つのインデックスが作成されていれば、SQL Anywhere はこの 2 つのインデックスを先にスキャンするので、一般的にはより速く情報を返すことができます。

インデックスを適切に選択するとパフォーマンスに大きな違いが生じる

SQL Anywhere では、インデックスによって非常に効率よく情報を検索できますが、インデックスを追加するときは注意してください。各インデックスは、ローの挿入、削除、更新時に余分な作業を生みます。SQL Anywhere は影響を受けるインデックスもすべて更新するからです。

このため、SQL Anywhere がデータにより効率的にアクセスできるときにかぎってインデックスを追加するようにしてください。特に、大きなテーブルに連続して不必要なアクセスを行うような処理をなくすときに実施します。ただし、テーブルにローを追加するときのパフォーマンスを向上させる必要があり、情報を迅速に検索する必要がない場合は、できるだけ少ないインデックスを使用します。

データベースに有効なインデックスを選択する手助けをするインデックスコンサルタントを使用することも 1 つの手段です。

クエリ最適化

オプティマイザは、可能であれば常に、インデックス専用取得でクエリを処理しようとします。インデックス専用取得では、データベースサーバでインデックスのデータだけを使用してクエリが処理されるため、テーブル内のローにアクセスする必要がありません。オプティマイザは、最高のパフォーマンスが得られるインデックスを判断し、それを使用するように自動的に選択します。ただし、クエリでインデックスヒントを使用して、オプティマイザが使用するインデックスを指定することもできます。指定したインデックスが 1 つでも使用できなかった場合はエラーが返されます。インデックスヒントによってパフォーマンスが低下する場合があるため、経験のあるユーザだけが実行するようにしてください。インデックスコンサルタントを使用して、データベースにインデックスを追加する必要があるかどうかを判別できます。

クラスタドインデックス

クラスタドインデックスを使用すると、テーブル内のローをインデックス内の順序とほぼ同じ順序で格納できます。

参照

- 「インデックス」26 ページ
- 「インデックスコンサルタント」156 ページ
- 「インデックス」26 ページ
- 「クラスタドインデックス」30 ページ
- WITH (index-hint) 句、FROM 句『SQL Anywhere サーバ SQL リファレンス』
- 「クエリの述部」282 ページ
- 「FROM 句」『SQL Anywhere サーバ SQL リファレンス』

ヒント：インデックスのパフォーマンスの向上

インデックスから予期したパフォーマンスが得られない場合は、次の措置を検討してください。

- 複合インデックスの再編成
- ページサイズの拡大

これらの措置によって、インデックスの選択性とインデックスファンアウトが向上します。

参照

- 「高度：インデックスの選択性とインデックスファンアウト」38 ページ

ヒント：混合負荷または OLAP 負荷の最適化

optimization_workload オプションを使用すると、通常はクエリと同時に更新、削除、または挿入が実行される (混合負荷) データベースに対してクエリ処理を最適化するかどうか、またはデータベース内の更新アクティビティの主な形式が、クエリ実行と同時にはめったに実行されないバッチ形式の更新かどうかを指定できます。

参照

- 「optimization_workload オプション」『SQL Anywhere サーバ データベース管理』

ヒント：クエリ結果の効果的なソートの使用

不要なデータのソート回数を減らします。予測可能な順序で返されるデータが必要でなければ、SELECT 文で ORDER BY 句を指定しないようにします。ソートを行うには、クエリを処理するために余計な時間とリソースが必要です。

参照

- 「ORDER BY 句：クエリ結果のソート」478 ページ
- 「GROUP BY 句：クエリ結果のグループへの編成」471 ページ

ヒント：正しいカーソルタイプの指定

正しいカーソルタイプを指定すると、パフォーマンスが改善されます。たとえば、カーソルが読み込み専用の場合、読み込み専用と宣言することで、最適化と実行が迅速になります。これは、検査制約など、構築する実体が少ないためです。カーソルが更新可能な場合は、クエリ書き換えの一部を省略できます。また、クエリが更新可能な場合、オプティマイザが選択した実行プランによっては、データベースサーバはキーセット駆動型アプローチを使用する必要があります。キーセットカーソルは、コストが高いことに留意してください。

参照

- 「カーソルタイプ」『SQL Anywhere サーバ プログラミング』

ヒント：明示的な選択性推定の提供を控える

場合によっては、統計が不正確になることがあります。このような状況が最も発生しやすいのは、大量のデータが追加、更新、または削除されてから実行されたクエリが少ない場合です。統計が不正確な場合、または利用できない場合、パフォーマンスに悪影響を与えます。統計の更新に SQL Anywhere が長時間要している場合、CREATE STATISTICS または DROP STATISTICS を実行して、統計を再表示してください。

SQL Anywhere では、LOAD TABLE 文の実行時、クエリの実行中、また更新 DML 文の実行時にも一部の統計が更新されます。

ただし、特異な状況では、この方法では効果的でないことがあります。条件の成功する確率がオプティマイザの推定とは異なることがあらかじめ判明している場合、ユーザ推定を明示的に探索条件として指定できます。

ユーザ定義の推定はパフォーマンスを改善することがありますが、継続的に使用される文にユーザ定義の推定を明示的に指定しないでください。データが変更されると、明示的な推定が不正確になり、オプティマイザが誤って不適切なプランを選択することがあります。

ソフトウェアによって選択されたアクセスプランが不適切であり、パフォーマンス問題を回避するために選択性推定を使用したものの、それが不正確であった場合は、user_estimates を Off に設定して、値を無視できます。

参照

- 「明示的な選択性推定」『SQL Anywhere サーバ SQL リファレンス』

ヒント：マテリアライズドビューを使ったクエリのパフォーマンス改善

負荷の高い集約操作やジョイン操作などを含むクエリのように、頻繁に実行されるコストの高いクエリでは、マテリアライズドビューを使用することを検討してください。マテリアライズドビューには、集約されジョインされたデータを格納するクエリ可能な構造体があります。マテリアライズドビューは、データベースのサイズが大きく、頻繁にクエリが行われるために大量のデータで繰り返し集約やジョイン操作が発生するような環境において、パフォーマンスが向上す

るように設計されています。たとえば、マテリアライズドビューは、データウェアハウスアプリケーションでの使用に適しています。

最適化時に送信されるクエリを部分的または全体的に満たす候補として見なされるマテリアライズドビューについて、オプティマイザはそのリストを管理します。オプティマイザがクエリの全体または部分を満たす候補となるマテリアライズドビューを検出すると、最適化の列挙フェーズ (コストを基に最適なプランを判断する) で行われる推奨の対象にそのビューを含めます。マテリアライズドビューをクエリに一致させるためにオプティマイザが使用する処理を「ビューマッチング」と呼びます。オプティマイザがマテリアライズドビューを検討するには、そのビューが一定の条件を満たす必要があります。マテリアライズドビューの使用が許可されていることをオプティマイザが判断すると、候補となるマテリアライズドビューのそれぞれが検査されます。マテリアライズドビューがクエリによって明示的に参照されないかぎり、オプティマイザによって使用される保証はありません。ただし、検討されるビューが条件を満たすようにすることはできます。

参照

- 「マテリアライズドビュー」53 ページ
- 「オプティマイザによるマテリアライズドビューの使用の有効化または無効化」69 ページ

マテリアライズドビューとビューマッチング

オプティマイザは、ビューマッチングアルゴリズムを使用して、クエリを満たすためにマテリアライズドビューを使用できるかどうかを判断します。この判断には、クエリ検査ステップとマテリアライズドビュー検査ステップが含まれます。

クエリ検査ステップ

クエリ検査では、ビューマッチングアルゴリズムによってクエリが検査されます。次のいずれかの条件に該当する場合、クエリの処理にマテリアライズドビューは「使用されません」。

- クエリで参照されるすべてのテーブルが更新可能である。

  派生の関係で更新可能である SELECT 文や、更新可能なカーソルで明示的に宣言された SELECT 文については、オプティマイザはマテリアライズドビューを検討しません。この状況は、Interactive SQL を使用していると発生することがあります。デフォルトで、Interactive SQL は SELECT 文で更新可能なカーソルを利用します。

- 文は、オプティマイザバイパスを使用する単純な DML 文であり、ヒューリスティックに最適化される。ただし、OPTION 句で FORCE OPTIMIZATION オプションを使用することにより、任意の SELECT 文でコストベースの最適化を使用できます。

- ストアドプロシージャやユーザ定義関数内に含まれるクエリの場合、クエリの実行プランはキャッシュされます。データベースサーバは、再利用できるようにこれらのクエリの実行プランをキャッシュする場合があります。このクラスのクエリの場合、クエリ実行プランは実行後にキャッシュされます。次回クエリが実行されると、プランが取得され、実行フェーズまでのすべてのフェーズがスキップされます。

マテリアライズドビュー検査ステップ

オプティマイザは、ビュー定義が次の状態である場合に、ビューマッチングアルゴリズムで検査されるマテリアライズドビューのセットにそのマテリアライズドビューを含めます。

- 含まれるクエリブロックが 1 つだけである。
- 含まれる FROM 句が 1 つだけである。
- 次の構成体や仕様のいずれも含まない。
  - GROUPING SETS
  - CUBE
  - ROLLUP
  - サブクエリ
  - 派生テーブル
  - UNION
  - EXCEPT
  - INTERSECT
  - マテリアライズドビュー
  - DISTINCT
  - TOP
  - FIRST
  - セルフジョイン
  - 再帰ジョイン
  - FULL OUTER JOIN
- (オプション) GROUP BY 句や HAVING 句 (HAVING 句に subselect やサブクエリが含まれない場合) が含まれる。

ビュー定義の基準を満たす以外に、次の条件を満たす必要があります。

- データベースサーバでマテリアライズドビューを使用できる必要がある
- 最適化でマテリアライズドビューを使用できる必要がある
- マテリアライズドビューは初期化されている (データが入力されている) 必要がある
- マテリアライズドビューを作成するための重要なオプションの値が、クエリを実行する接続のオプションと一致している必要がある
- 最後に再表示したマテリアライズドビューで、materialized_view_optimization データベースオプションに設定された古さのしきい値を超えていない

マテリアライズドビューが上記の基準を満たし、クエリの全体または一部を満たすことがわかった場合、コストベースの最適プランが検出されると、ビューマッチングアルゴリズムは最適化の列挙フェーズ用の推奨にマテリアライズドビューを含めます。ただし、それにより最終的にそのマテリアライズドビューが最終実行プランで使用されるというわけではありません。たとえば、マテリアライズドビューを使用しない別のアクセスプランの方がコストが低いと推定されると、クエリの結果を計算するのに適切であると考えられるマテリアライズドビューであっても使用されない可能性があります。

参照

- 「オプティマイザによるマテリアライズドビューの使用の有効化または無効化」69 ページ
- 「sa_materialized_view_info システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- ホワイトペーパー『**Query Processing Based on SQL Anywhere 12.0.1 Architecture**』(http://www.sybase.com/detail?id=1096047)
- 「プランのキャッシュ」325 ページ
- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「高度：クエリ実行プラン」328 ページ
- 「高度：クエリ処理のフェーズ」317 ページ
- 「アプリケーションプロファイリング」148 ページ

マテリアライズドビューの候補リストの取得

Interactive SQL から、オプティマイザによって検討されるマテリアライズドビューの候補リストを取得できます。

前提条件

この作業を実行するための前提条件はありません。

♦ タスク

1. 次の文を実行します。

   ```
   SELECT * FROM sa_materialized_view_info( ) WHERE AvailForOptimization='Y';
   ```

   オプティマイザはリストの生成時にオプション設定を考慮するため、返されるリストは要求している接続に固有です。接続に指定したオプションとマテリアライズドビューの作成時のオプションとが一致しない場合、そのマテリアライズドビューは候補と見なされません。

2. オプション設定の不一致が原因で候補と見なされないすべてのマテリアライズドビューのリストを取得するには、クエリを実行する接続から次のクエリを実行します。

   ```
   SELECT * FROM sa_materialized_view_info( ) WHERE AvailForOptimization='O';
   ```

結果

マテリアライズドビューの候補リストが、表示されます。

参照

- 「マテリアライズドビューの制限」58 ページ

オプティマイザによって検討されたマテリアライズドビューの判断

Interactive SQL では、オプティマイザがクエリを満たすために使用するマテリアライズドビューを判断できます。

前提条件

この作業を実行するための前提条件は、ありません。

内容と備考

> **注意**
> スナップショットアイソレーションが使用されている場合、オプティマイザは現在のトランザクションのスナップショットの開始後に再表示されたマテリアライズドビューを検討しません。

♦ タスク

1. Interactive SQL で、データベースに接続します。
2. **[ツール]** » **[プラン ビューアを開く]** をクリックします (または [Shift + F5] キーを押します)。
3. **[SQL 文]** ウィンドウ枠にクエリを入力します。
4. **[統計レベル]**、**[カーソルタイプ]**、**[更新のステータス]** を選択します。
5. **[プランの取得]** をクリックします。
6. **[詳細]** と **[高度な詳細]** ウィンドウ枠を見て、クエリを満たすために使用したマテリアライズドビューを判断します。

結果

クエリを満たすために使用したマテリアライズドビューが、表示されます。

参照

- 「高度：クエリ実行プラン」328 ページ
- 「アプリケーションプロファイリング」148 ページ
- 「カスタマイズされた診断トレーシングレベル」167 ページ
- 「高度：クエリ処理のフェーズ」317 ページ

ヒント：テーブル検証時の WITH EXPRESS CHECK オプションの使用

少ないキャッシュ容量で大きいデータベースを検証するのに長時間かかる場合は、2 つのオプションのいずれかを使用して所要時間を短縮できます。VALIDATE TABLE 文で WITH EXPRESS CHECK オプションを指定するか、検証ユーティリティ (dbvalid) で -fx オプションを使用すると、テーブルの検証が大幅に速くなります。

参照

- 「VALIDATE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「検証ユーティリティ (dbvalid)」『SQL Anywhere サーバ データベース管理』

ヒント：クエリ処理におけるワークテーブルの使用 (All-rows 最適化ゴールの使用)

ワークテーブルは、クエリの実行中に作成される一時的な結果セットを実体化したものです。ワークテーブルは、代替方式よりもコストが低い場合に使用します。通常、ワークテーブルを使用すると、最初のいくつかのローをフェッチする所要時間は長くなります。ただし、ワークテーブルを使用できれば、すべてのローを検索するコストは大幅に低下する場合があります。このような違いがあるため、SQL Anywhere は optimization_goal の設定に基づいてさまざまな方式を選択します。デフォルトは All-rows です。All-rows に設定されている場合、クエリの合計実行コストが減少するなら、ワークテーブルが使用されます。optimization_goal が First-row に設定されている場合、SQL Anywhere はワークテーブルを使用しないようにします。

ワークテーブルが使用されるのは、次の場合です。

- クエリに ORDER BY 句、GROUP BY 句、または DISTINCT 句があり、SQL Anywhere がローのソートにインデックスを使用しないとき。適切なインデックスが存在し、optimization_goal が First-row に設定されている場合、SQL Anywhere はワークテーブルを使用しません。ただし、optimization_goal が All-rows に設定されている場合は、ワークテーブルを作成してローをソートするよりも、インデックスを使用してクエリのローをすべてフェッチする方が高コストになることがあります。最適化ゴールが All-rows に設定されている場合、SQL Anywhere は負荷がより低い方式を選択します。GROUP BY と DISTINCT の場合、ハッシュベースのアルゴリズムではワークテーブルが使用されますが、通常はクエリからローをすべてフェッチする方が効率的です。

- ハッシュジョインアルゴリズムが選択されたとき。この場合、ワークテーブルが中間結果の格納 (入力がメモリに収まらない場合) とジョイン結果の格納に使用されます。

- sensitive 値を使用してカーソルが開かれたとき。この場合、ベーステーブルのロー識別子とプライマリキーを入れるワークテーブルが作成されます。このワークテーブルは、クエリから前方にローがフェッチされるたびに埋められます。ただし、カーソルから最後のローをフェッチすると、テーブル全体が埋められます。

- insensitive セマンティックを使用してカーソルが開かれたとき。この場合、クエリが開かれるときに、ワークテーブルにクエリの結果が設定されます。

- 複数のローに UPDATE を実行し、UPDATE の WHERE 句または更新に使用するインデックスに、更新されるカラムが表示されるとき。

- 複数のローに対する UPDATE または DELETE が、修正されるテーブルを参照する WHERE 句にサブクエリを持つとき。

- SELECT 文から INSERT を実行し、その SELECT 文が挿入テーブルを参照するとき。

- 複数のローに INSERT、UPDATE、または DELETE を実行し、その操作中に起動するように、対応するトリガがテーブルに定義されているとき。

これらの場合は、操作の影響を受けるレコードがワークテーブルに入れられます。キーセット駆動型カーソルなど、場合によっては、ワークテーブルにテンポラリインデックスが作成されます。要求されたレコードをワークテーブルに抽出する操作は、クエリの結果が出るまでかなりの

時間がかかります。上記の最初の例でソートを実行するのに使用できるインデックスを作成すると、最初のいくつかのローを検索する時間が短縮されます。ただし、ワークテーブルを使用すると、すべてのローをフェッチするための合計時間を短縮できる場合があります。これは、ハッシュとマージソートに基づくクエリアルゴリズムが許可されるためです。これらのアルゴリズムでは、シーケンシャル I/O が使用されます。これは、インデックススキャンと併用されるランダム I/O より高速です。

オプティマイザは、各クエリを分析し、ワークテーブルを使用した場合に最適なパフォーマンスが得られるかどうかを判断します。こうした最適化を利用するのに、ユーザ側の操作は必要ありません。

注意

上述の INSERT、UPDATE、DELETE は通常は 1 回かぎりの操作のため、通常、パフォーマンスは問題にはなりません。ただし、問題が発生した場合は、文を書き直して、競合とワークテーブルが作成されるのを防止できる場合があります。これは、常に可能とはかぎりません。

参照

- 「optimization_goal オプション」『SQL Anywhere サーバ データベース管理』

ヒント：効率的な SQL クエリの構築

クエリの処理パフォーマンスを向上させるために、次のヒントを使用して、より効率的なクエリを構築することを検討してください。これらのヒントは、クエリをより効率的に書き換えるために、クエリ処理中にオプティマイザによって選択される最適化を反映しています。これらの効率化をクエリに組み込むことによって、オプティマイザの作業量が減少する可能性があります。

ヒント	改善の前後	説明
不要な DISTINCT 条件の排除	改善前： SELECT DISTINCT p.ID, p.Quantity FROM Products p; 改善後： SELECT p.ID, p.Quantity FROM Products p;	Products テーブルにプライマリキー p.ID があり、結果セットの一部になっているので、最初の文の DISTINCT キーワードは不要です。

ヒント	改善の前後	説明

不要な DISTINCT 条件の排除	改善前：
	
	SELECT DISTINCT *
	FROM SalesOrders o JOIN Customers c
	  ON o.CustomerID = c.ID
	WHERE c.State = 'NY';
	
	改善後：
	
	SELECT *
	FROM SalesOrders o JOIN Customers c
	  ON o.CustomerID = c.ID
	WHERE c.State = 'NY';
		最初のクエリには両方のテーブルのプライマリキーが含まれているため、結果内の各ローは異なります。

サブクエリのネスト解除	改善前：
	
	SELECT s.*
	FROM SalesOrderItems s
	WHERE EXISTS (SELECT *
	 FROM Products p
	 WHERE s.ProductID = p.ID
	  AND p.ID = 300
	  AND p.Quantity > 20);
	
	改善後：
	
	SELECT s.*
	FROM Products p JOIN SalesOrderItems s
	  ON p.ID = s.ProductID
	WHERE p.ID = 300 AND p.Quantity > 20;
		ネストされたクエリをジョインとして書き換えると、より効率的な実行とより効果的な最適化につながる場合があります。一般的に、サブクエリのネスト解除は、FROM 句にテーブルを多くても 1 つだけ含んだ相関サブクエリに対して常に実行されます。これらの相関サブクエリは、ANY、ALL、EXISTS の各述部で使用されます。クエリのセマンティクス上、サブクエリが返すローは多くても 1 つだけであると判断できる場合は、非相関サブクエリ、または FROM 句に複数のテーブルを含んだサブクエリはフラットにされます。
		
		この例では、サブクエリは外部ブロック内の各ローに対して、多くても 1 つのローしか一致させられません。多くても 1 つのローしか一致させられないため、内部ジョインに変換できます。

ヒント	改善の前後	説明
サブクエリのネスト解除	改善前： SELECT p.* FROM Products p WHERE EXISTS (SELECT * FROM SalesOrderItems s WHERE s.ProductID = p.ID AND s.ID = 2001); 改善後： SELECT DISTINCT p.* FROM Products p JOIN SalesOrderItems s ON p.ID = s.ProductID WHERE s.ID = 2001;	改善前のクエリは、サブクエリ内に結合 EXISTS 述部があり、複数のローと一致する可能性があります。SELECT リストに DISTINCT を使って、内部ジョインに変換できます。
サブクエリのネスト解除	改善前： SELECT * FROM Products p WHERE p.ID = (SELECT s.ProductID FROM SalesOrderItems s WHERE s.ID = 2001 AND s.LineID = 1); 改善後： SELECT p.* FROM Products p, SalesOrderItems s WHERE p.ID = s.ProductID AND s.ID = 2001 AND s.LineID = 1;	サブクエリで、外部ブロックの各ローに対して多くても 1 つのローしか一致しない場合、比較しているサブクエリを削除します。
インデックスカラムを問い合わせる場合の IN 述部の使用の検討	改善前： SELECT * FROM SalesOrders WHERE SalesRepresentative = 902 OR SalesRepresentative = 195; 改善後： SELECT * FROM SalesOrders WHERE SalesRepresentative IN (195, 902);	書き換えられた形式では、IN リスト述部を検索引数可能として扱い、インデックス検索に利用できます。また、オプティマイザが IN リストをインデックスのソート順に合わせてソートし、検索効率を向上させることができます。 定数だけ、またはクエリブロック 1 回の実行中に定数である値 (外部参照など) だけを IN リストに指定してください。

ヒント	改善の前後	説明
不要なジョインの排除	改善前： SELECT s.ID, s.LineID, p.ID FROM SalesOrderItems s KEY JOIN Products p FOR READ ONLY; 改善後： SELECT s.ID, s.LineID, s.ProductID FROM SalesOrderItems s WHERE s.ProductID IS NOT NULL FOR READ ONLY;	次の場合にジョインの排除を検討してください。 ●ジョインがプライマリキーから外部キーへのジョインで、プライマリテーブルからのプライマリキーのカラムだけがクエリで参照される場合。この場合は、プライマリキーテーブルが更新可能でないと、削除されます。 ●ジョインが同じテーブルの 2 つのインスタンス間におけるプライマリキーからプライマリキーへのジョインである場合。この場合は、一方のテーブルが更新可能でないと、そのテーブルは削除されます。 ●ジョインが外部ジョインであり、NULL 入力テーブル式は、外部ジョインの保護された側の各ローについて、多くても 1 つのローしか返さず、NULL 入力テーブル式で生成される式は、外部ジョイン以降のクエリの残りの部分で必要ない場合。 この場合、ジョインはプライマリキーから外部キーへのジョインであり、プライマリキーテーブル Products は削除できます。つまり、2 番目のクエリは、SalesOrderItems テーブルの中で、Products への NULL 外部キーを持つローが結果に現れないため、セマンティック上は 1 番目のクエリと同じです。

ヒント	改善の前後	説明
不要なジョインの排除	改善前： SELECT s.ID, s.LineID FROM SalesOrderItems s LEFT OUTER JOIN Products p ON p.ID = s.ProductID WHERE s.Quantity > 5 FOR READ ONLY; 改善後： SELECT s.ID, s.LineID FROM SalesOrderItems s WHERE s.Quantity > 5 FOR READ ONLY;	最初のクエリでは、NULL 入力テーブル式が保護された側のローに対して複数のローを生成できず、LEFT OUTER JOIN を超えて Products のカラムが使用されないため、OUTER JOIN を削除できます。
大文字と小文字の不要な変換の排除	改善前： SELECT * FROM Customers WHERE UPPER(Surname) = 'SMITH'; 改善後： SELECT * FROM Customers WHERE Surname = 'SMITH';	大文字と小文字を区別しないデータベースでは、最初のクエリは書き換えることができます。そのため、オプティマイザでは Customers.Surname のインデックスを使用することを検討できます。 デフォルトでは、明示的なテキスト変換指示 (UPPER、UCASE、LOWER、LCASE の使用) が指定されないかぎり、データベースサーバで大文字と小文字を区別しない文字列比較が実行されます。大文字と小文字の不要な変換の排除により、述部を検索引数可能な述部にすることができます。検索引数可能な述部は、対応するテーブルのインデックス検索を使用できます。

ヒント	改善の前後	説明

関数のインライン化の検討	改善前：
	
	CREATE FUNCTION F1(arg1 INT, arg2 INT)
	RETURNS INT
	BEGIN
	RETURN arg1 * arg2
	END;
	SELECT F1(e.EmployeeID, 2.5)
	FROM Employees e;
	
	改善後：
	
	SELECT CAST(e.EmployeeID AS INT) * CAST(2.5 AS INT)
	FROM Employees e;
		次のいずれかの形式の場合、ユーザ定義関数をインライン化できます。
		
		●RETURN 文を 1 つ含んでいる
		
		●1 つの変数を宣言し、その変数に代入して 1 つの値を返す
		
		●1 つの変数を宣言し、その変数に SELECT INTO して 1 つの値を返す
		
		このヒントは、テンポラリ関数、再帰関数、または NOT DETERMINISTIC 句のある関数には適用されません。
		
		また、このヒントは関数がサブクエリを使用して引数として呼び出された場合、または関数がテンポラリプロシージャの内部から呼び出された場合にも適用されません。

単純なストアドプロシージャのインライン化の検討	改善前：
	
	CREATE PROCEDURE Test1(arg1 INT)
	BEGIN
	SELECT * FROM Employees WHERE EmployeeID=arg1
	END;
	SELECT * FROM Test1(200);
	
	改善後：
	
	SELECT * FROM (
	SELECT * FROM Employees
	WHERE EmployeeID=CAST(200 AS INT))
	AS Test1;
		単一の SELECT 文としてのみ定義されたストアドプロシージャは、クエリの FROM 句で呼び出すときにインライン化できます。プロシージャがインライン化される場合は、派生テーブルとして書き換えられます。このヒントは、デフォルトの引数を使用し、単一の SELECT 文以外が本文に存在するプロシージャには適用されません。

参照

- 「ユーザ定義関数」 89 ページ
- 「CREATE FUNCTION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CAST 関数 [データ型変換]」『SQL Anywhere サーバ SQL リファレンス』
- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「プロシージャ」 78 ページ
- 「CREATE PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』

ヒント：クエリのパフォーマンスのモニタ

SQL Anywhere には、クエリのパフォーマンスをテストするツールが複数用意されています。これらのツールは、この後に示す *%SQLANYSAMP16%¥SQLAnywhere* のサブディレクトリにあります。各ツールの完全なマニュアルについては、ツールと同じフォルダにある *readme.txt* ファイルを参照してください。

クエリの実行時間を測定するシステムプロシージャについては、「sa_get_request_profile システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』と「sa_get_request_times システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』を参照してください。

ツール	機能	ロケーション
fetchtst	結果セットの取り出しに必要な時間を決定します。	*%SQLANYSAMP16%¥SQLAnywhere¥PerformanceFetch*
odbcfet	結果セットの取り出しに必要な時間を決定します。このツールは fetchtst に似ていますがより限定的です。	*%SQLANYSAMP16%¥SQLAnywhere¥PerformanceFetch*
instest	ローをテーブルに挿入するための所要時間を決定します。	*%SQLANYSAMP16%¥SQLAnywhere¥PerformanceInsert*
trantest	データベース設計とトランザクションセットが与えられている特定のデータベースサーバ設定によって処理できる負荷を測定します。	*%SQLANYSAMP16%¥SQLAnywhere¥PerformanceTransaction*

参照

- 「チュートリアル：速度が遅い文の診断」 263 ページ
- 「パフォーマンスの問題のトラブルシューティング」 180 ページ

ヒント：コストの高いユーザ定義関数の削減

クエリ中で何度も実行されるコストの高いユーザ定義関数を削減すると、パフォーマンスが改善されます。

参照

- 「ユーザ定義関数」89 ページ

ヒント：オートコミットモードをオフにする

アプリケーションが「オートコミットモード」で実行されている場合、SQL Anywhere は各文を別々のトランザクションとして扱います。これは、それぞれの文の最後に COMMIT 文を付加して実行するのと同じ効果があります。

オートコミットモードで実行する代わりに、SQL 文をグループ分けして各グループが 1 つの論理タスクを実行するようにします。オートコミットを無効にする場合は、SQL 文の各論理グループの後に明示的にコミットを実行してください。また、論理トランザクションが大きい場合は、ブロッキングとデッドロックが発生する可能性があることに注意してください。

トランザクションログファイルを使用していない場合、オートコミットモードを使用するコストは高くなります。各文の終わりで、チェックポイントが強制的に実行されます。チェックポイントとは、多数の情報ページをディスクに書き込む操作です。

各アプリケーションインタフェースには、オートコミット動作を設定する独自の方法があります。Open Client、ODBC、JDBC インタフェースでは、オートコミットはデフォルトの動作です。

参照

- 「オートコミットまたは手動コミットモード」『SQL Anywhere サーバ プログラミング』

ヒント：インメモリモードの使用

最新のチェックポイント後にコミットされたトランザクションが失われても構わないアプリケーションには、インメモリモードを使用すると便利な場合があります。

大量のメモリを使用できる (通常、すべてのデータベースファイルをキャッシュ内に保持できる) システムを実行している場合、パフォーマンスの向上が求められるアプリケーションでこのモードを使用すると便利です。

インメモリモードは 2 種類から選択できます。非書き込みモードでは、コミットされたトランザクションはディスク上のデータベースファイルに書き込まれません。非書き込みモードを指定すると、複数の LOAD TABLE 文を同じまたは異なるテーブルで同時にアクティブにできます。データベースが停止した場合、または接続が切断された場合は、すべての変更が失われます。チェックポイント専用モードでは、データベースサーバはトランザクションログを使用しないため、最新のコミットされたトランザクションにリカバリすることはできません。ただし、チェックポイントログは有効になっているため、データベースを最新のチェックポイントにリカバリすることができます。

インメモリモードの設定方法およびアプリケーションに対して適切か判断する方法については、「-im データベースサーバオプション」『SQL Anywhere サーバ データベース管理』を参照してください。

注意
インメモリモードには別途ライセンスが必要です。「別途ライセンスが必要なコンポーネント」『SQL Anywhere 16 紹介』を参照してください。

ヒント：常にトランザクションログを使用する

トランザクションログを使用すると、データを保護できます。また、SQL Anywhere のパフォーマンスを大幅に向上できます。

トランザクションログなしで操作すると、SQL Anywhere は各トランザクションの終わりにチェックポイントを実行します。これにより、大量のリソースが消費されます。

トランザクションログを使用して操作すると、SQL Anywhere は変更が発生したときにその詳細を示すメモを書き込むだけです。新しいデータベースページすべてを、最も効率のいい時に一度に書き込むように選択できます。「チェックポイント」は、情報がデータベースファイルに入力され、矛盾せず、最新のものであるということを確実にします。

トランザクションログを、プライマリデータベースファイルとは別の物理デバイスに入れることで、パフォーマンスをさらに改善できます。追加されたドライブヘッドは、通常、トランザクションログの終わりを探す必要はありません。

ヒント：コミットの遅延の使用

データベースに対してコミットされる変更の頻度が高い場合、トランザクションログの書き込みの頻度が、データベース全体のパフォーマンスを左右する最大の要因になります。トランザクションログのパフォーマンスを改善する場合は、delayed_commits オプションを On に設定します。On に設定すると、データベースサーバは COMMIT のトランザクションログエントリがディスクに書き込まれるのを待たずに、直ちに COMMIT 文に応答します。Off に設定すると、アプリケーションは COMMIT がディスクに書き込まれるまで待たなければなりません。
delayed_commits オプションをオンにすると、一部が埋まっているログページの複数回の書き換えが回避され、トランザクションログの書き込みが減少します。このオプションは、接続ごとまたは接続全体に対して設定できます。delayed_commits オプションをオンにすると、トランザクションログページがディスクにフラッシュされる前にサーバがダウンした場合、コミットされた操作が失われるリスクがあります。

参照

- 「delayed_commits オプション」『SQL Anywhere サーバ データベース管理』

ヒント：小さいテーブルに関する統計収集

SQL Anywhere は、統計情報を基に各文の実行に最も効率的な方法を判別します。SQL Anywhere は、それらの統計を自動的に収集、更新し、データベースに永続的に格納します。ある文の処理中に収集された統計は、以降の文の効率的な実行方法を見いだすときに使用できます。

デフォルトでは、SQL Anywhere は 5 つ以上のローを持つすべてのテーブルの統計を作成します。5 つ未満のローを持つテーブルの統計を作成する必要がある場合は、CREATE STATISTICS 文を使用して作成します。この文は、テーブル内のローの数に関係なく、すべてのテーブルの統計を作成します。統計は、いったん作成されると、SQL Anywhere によって自動的に管理されます。

参照

- 「CREATE STATISTICS 文」『SQL Anywhere サーバ SQL リファレンス』

ヒント：断片化の削減

データベースの変更には断片化がともないます。ファイル、テーブル、またはインデックスが必要以上に断片化されていると、パフォーマンスが低下することがあります。断片化の削減は、データベースのサイズに比例して重要になります。SQL Anywhere には、ファイル、テーブル、インデックスの断片化情報を生成するストアドプロシージャが用意されています。

パフォーマンスが大幅に低下した場合は、次の方法で対処できます。

- 複数のテーブルで多数の削除／更新／挿入アクティビティを実行した場合は、テーブルやインデックスの断片化を削減するためにデータベースを再構築する。
- データベースをディスクパーティションに単独で入れ、ファイルの断片化を低減する。
- 利用できる Windows ユーティリティの 1 つを定期的に実行して、ファイルの断片化を低減する。
- データベースを再編成して、データベースの断片化を低減する。
- REORGANIZE TABLE 文を使用して、テーブル内のローをデフラグしたり、DELETE によって散在したインデックスを圧縮する。テーブルを再編成すると、テーブルとインデックスの格納に使用される合計ページ数を減らし、インデックスツリーのレベル数も減らすことができます。

ファイルの断片化の削減

断片化したデータベースファイルは、データベースサーバのパフォーマンスに影響することがあります。ディスク断片化を減らすことは、データベースのサイズが大きくなるに従って重要になります。

Windows でデータベースを起動すると、データベースサーバが各 DB 領域内でファイルの断片化数を判断します。断片化の数が 1 より大きい場合は、データベースサーバメッセージウィンドウに次のパフォーマンス警告が表示されます。**データベースファイル "mydatabase.db" は nnn のディスクフラグメントで構成されています。**

また、DBFileFragments データベースプロパティを使用しても、データベースファイルの断片化の数を取得できます。

ファイルの断片化の問題を解決するには、データベースをディスクパーティションに単独で入れ、使用可能な Windows ディスクデフラグユーティリティの 1 つを定期的に実行します。

参照

- 「データベースプロパティの一覧」『SQL Anywhere サーバ データベース管理』
- 「パフォーマンス警告：データベースファイル "%1" は %2 のディスクフラグメントで構成されています。」『SQL Anywhere サーバ データベース管理』
- DBFileFragments データベースプロパティ『SQL Anywhere サーバ データベース管理』

テーブルの断片化削減

テーブルの断片化は、ローが連続して格納されていない場合、またはローが複数のページに分割されている場合に発生します。これらのローには追加のページアクセスが必要になるため、データベースサーバのパフォーマンスを低下させます。

断片化がパフォーマンスに与える影響は、さまざまなものがあります。テーブルが極端に断片化されていても、メモリ内に収まっていて、かつアクセス方法によってページのキャッシュが許可されている場合、影響は最小限に抑えられる可能性があります。ただし、断片化されたテーブルにより、大量の I/O 処理が発生することがあります。分割されたローが頻繁にアクセスされ、かつ余計な I/O のコストがキャッシュによって削減されない場合は、パフォーマンスが大幅に低下する可能性があります。

テーブルの再編成とデータベースの再構築を行うと断片化を削減できますが、行う回数が多すぎても少なすぎても、パフォーマンスに影響を与えることがあります。次の項で説明するツールや方法を実際に使ってみて、テーブルの許容可能な断片化レベルを判断してください。

断片化を削減しても依然としてパフォーマンスがよくない場合は、統計が不正確であるなどの別の問題が考えられます。

テーブルの断片化レベルの判断

パフォーマンスを改善するためにデフラグを実行するかどうかを判断するうえで、テーブルの断片化のチェックを 1 回実行するだけでは役立ちません。データベースを再構築してからタブの断片化をチェックし、ベースラインとなる結果を取得してください。次に、長期間にわたってテーブルの断片化を定期的にチェックし、断片化における変化とパフォーマンス測定時の変化との相関を調べます。この方法によって、テーブルの断片化がパフォーマンスに影響をきたすまでの状態になる速度を確認でき、それによりテーブルのデフラグを実行する最適な頻度を決定できます。

データベーステーブルの断片化の程度に関する情報を取得するには、次のいずれかの方法を使用します。

- sa_table_fragmentation システムプロシージャを呼び出します。次に例を示します。

```
CALL sa_table_fragmentation( );
```

●SQL Anywhere プラグインの **[断片化]** タブ。**[断片化]** タブを使用すると、ベーステーブルで sa_table_fragmentation システムプロシージャを実行した結果がグラフィックで表示されます。

参照

- 「sa_table_fragmentation システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_table_fragmentation システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「REORGANIZE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』
- 「パフォーマンスのモニタリングと診断のツール」145 ページ
- 「[断片化] タブ (SQL Anywhere プラグインの場合)」242 ページ

テーブルの断片化を削減する方法

次の方法で、テーブルの断片化を制御できます。

- **PCTFREE の使用**　SQL Anywhere では、ローが少し大きくなることを見込んで各ページに余分な領域を確保します。ローが更新されたために、最初に割り付けられていた領域より大きくなると、そのローは分割され、最初のローのロケーションにはロー全体が格納されている別のページへのポインタが入れられます。たとえば、空のローを UPDATE 文で埋めたり、テーブルに新しいカラムを挿入したりすると、ローが分割される可能性があります。別個のページに格納されるローが増えるほど、追加ページへのアクセス所要時間が長くなります。

  テーブルページに今後の更新のための予約領域の割合を指定しておくと、テーブルの断片化量を減らすことができます。この PCTFREE 指定は、CREATE TABLE、ALTER TABLE、DECLARE LOCAL TEMPORARY TABLE、または LOAD TABLE で設定できます。

- **テーブルの再編成**　REORGANIZE TABLE 文を使用するか、Sybase Central の **[断片化]** タブで **[再編成]** をクリックすると、特定のテーブルの断片化を解除できます。

- **データベースの再構築**　データベースを **2 段階で再構築**すると、システムテーブルを含むすべてのテーブルの断片化が解除されます。2 段階で行うには、データをディスクにアンロードして保存してから、再ロードします。この方法で再構築すると、テーブルのローが並べ替えられ、クラスタドインデックスとプライマリキーで指定された順序になります。-ar、-an、-ac などのオプションを使用して一度に再構築を行うと、テーブルの断片化は減りません。

参照

- PCTFREE 句『SQL Anywhere サーバ SQL リファレンス』
- 「REORGANIZE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「[断片化] タブ (SQL Anywhere プラグインの場合)」242 ページ
- 「データベースの再構築」749 ページ
- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』

[断片化] タブ (SQL Anywhere プラグインの場合)

[断片化] タブは次の用途で使用できます。

●ベーステーブルおよびベーステーブル上のインデックスの断片化の表示

●テーブルおよびインデックスの再編成

DB 領域マップでのズーム

[断片化] タブの下部ウィンドウ枠で DB 領域マップを開くと、デフォルトで、ズームレベルは **[ウィンドウに合わせる]** に設定されます。ズームインするには DB 領域マップをクリックし、ズームアウトするには [Shift] キーを押しながらクリックします。DB 領域マップ内でクリックするか、[Shift] キーを押しながらクリックすると、ズームレベルの変更後、クリックしたページが中央に表示されます。

ツールバーボタンを使用して、次のレベルにズームすることもできます。

ツールバーボタン	定義
1: 1	**1 ページ : 1 ピクセル**
64KB: 1	**1 64 KB ブロック : 1 ピクセル** (1 データベース読み込み)
ウィンドウに合わせる	ウィンドウのすべての使用可能な領域を使用

オブジェクトの断片化の詳細の表示

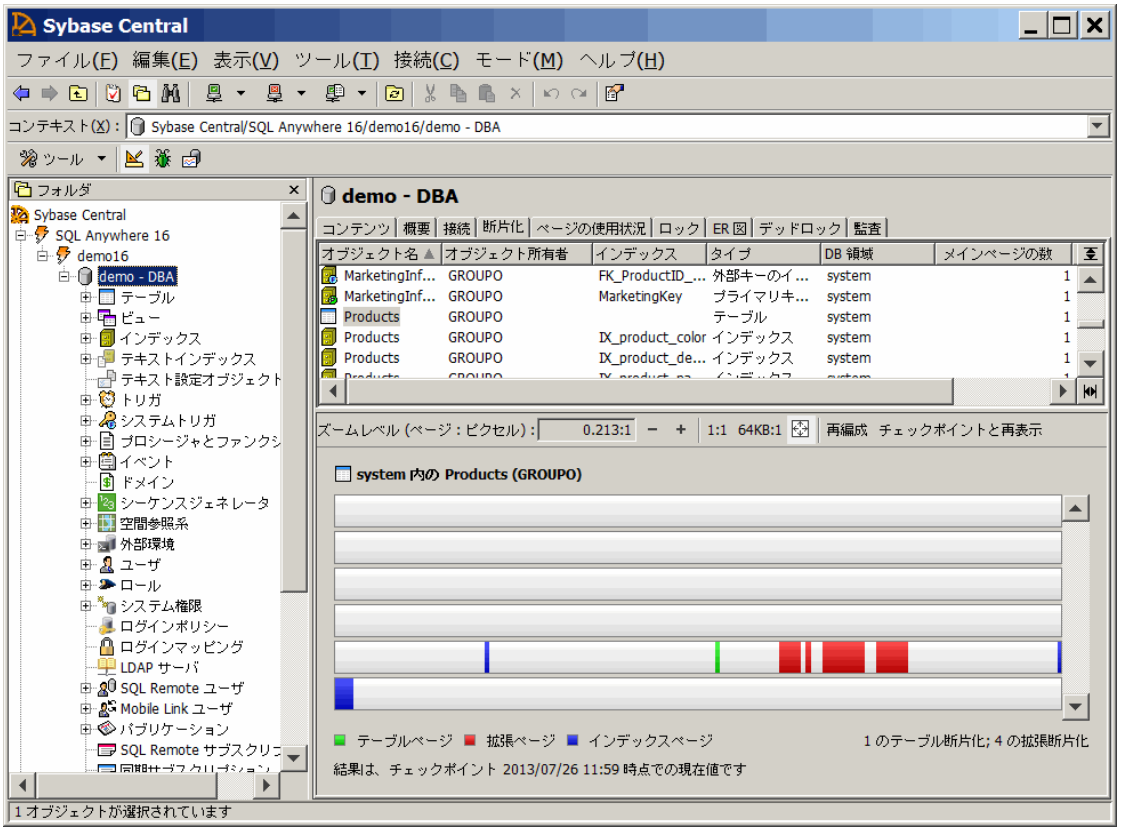
Sybase Central の **[断片化]** タブから、オブジェクトの断片化情報を表示できます。

前提条件

CHECKPOINT システム権限が必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠でデータベースを選択します。右ウィンドウ枠で、**[断片化]** タブをクリックします。
3. 上部ウィンドウ枠からオブジェクトを選択します。下部ウィンドウ枠の DB 領域マップに断片化情報が表示されます。
   - ●ベーステーブルを選択すると、下部ウィンドウ枠の DB 領域マップにテーブル、テーブルの拡張ページ、該当するインデックスページが表示されます。
   - ●インデックスを選択すると、下部ウィンドウの DB 領域マップにインデックスページが表示されます。



4. **[チェックポイントと再表示]** をクリックして、チェックポイントを実行し、最新の断片化情報を確認します。

5. ページインデックスを表示します。

   ●下部ウィンドウの DB 領域マップで、カーソルを色付きの縦線の上に置き、その位置の最初と最後のページを確認します。

   ●DB 領域マップで、[Ctrl] キーを押したままマウスポインタを色付きの縦線の上に置き、その位置のすべてのページインデックスを確認します。

結果

指定されたオブジェクトの断片化の詳細が、下部のウィンドウ枠の dbspace マップに表示されます。

ベーステーブルとインデックスの再編成

管理者は、Sybase Central の **[断片化]** タブからベーステーブルとインデックスを再編成できます。

前提条件

オブジェクトの所有者であるか、REORGANIZE ANY OBJECT システム権限を持っている必要があります。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠でデータベースを選択します。右ウィンドウ枠で、**[断片化]** タブをクリックします。
3. 上部ウィンドウ枠からオブジェクトを選択します。下部ウィンドウ枠の DB 領域マップに断片化情報が表示されます。
4. 次のいずれかの方法を選択して、オブジェクトを再編成します。
   - **[再編成]** をクリックして、選択したオブジェクトに対して REORGANIZE TABLE 文を実行します。
   - 上部ウィンドウ枠から Interactive SQL の **[SQL 文]** ウィンドウ枠にオブジェクトをコピーします。**[SQL 文]** ウィンドウ枠にオブジェクトの REORGANIZE TABLE 文が表示されます。文を実行します。

     この方法は、オブジェクトの再編成を後で行う場合、またはオブジェクトを再編集しながら Sybase Central を引き続き使用する場合に有用です。

結果

指定されたテーブルまたはインデックスが再編成されます。

参照

- 「REORGANIZE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

インデックスの断片化とスキューの削減

インデックスは特定のカラムの検索が高速になるように設計されていますが、インデックス付きテーブルに対して多数の DELETE 操作が実行されると、インデックスが断片化 (密度が下がる) およびスキューする (偏る) ことがあります。

インデックスの密度は、インデックスページの平均の満杯度を反映しています。インデックススキューは、平均密度からの一般的な偏差を反映しています。オプティマイザが選択性推定を行う場合、このスキュー量は重要です。

データベース内に許容できないレベルの断片化またはスキューを含むインデックスがあるかどうかを判断するには、**アプリケーションプロファイリングウィザード**を使用します。

sa_index_fragmentation システムプロシージャを使用して、インデックスの断片化およびスキューのレベルを確認することもできます。たとえば、次の文は sa_index_density システムプロシージャを呼び出して、Customers テーブルのインデックスを調べます。

```
CALL sa_index_density( 'Customers' );
```

TableName	TableId	IndexName	IndexID	IndexType	LeafPages	Density	Skew
Customers	718	CustomersKey	0	PKEY	1	0.127686	1.000000
Customers	718	IX_customer_name	1	NUI	1	0.789795	1.000000

SQL Anywhere はプライマリキーのインデックスを自動的に作成します。これらのインデックスでは、sa_index_density システムプロシージャの結果に、IndexID が 0 のインデックスが存在します。

リーフページ数が少ない場合は、密度とスキューの値を考慮する必要はありません。密度とスキューの値は、リーフページ数が多いときだけ重要です。リーフページ数が多い場合、密度の値が低いと断片化を表し、不均衡値が高いとインデックスが偏っていることを示します。どちらも、パフォーマンス低下の要因になります。REORGANIZE TABLE 文を実行すると、この両方の問題に対処できます。

SQL Anywhere プラグインで **[断片化]** タブを使用して、ベーステーブルに関連付けられたインデックスのインデックス断片化のレベルを確認することもできます。

参照

- 「アプリケーションプロファイリングウィザードの使用」149 ページ
- 「REORGANIZE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「[断片化] タブ (SQL Anywhere プラグインの場合)」242 ページ
- 「[断片化] タブ (SQL Anywhere プラグインの場合)」242 ページ
- 「sa_index_density システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「インデックス」26 ページ
- 「[断片化] タブ (SQL Anywhere プラグインの場合)」242 ページ

ヒント：テーブル構造の正規化

1 つまたは複数のデータベーステーブルに同じ情報の複数のコピーが含まれる場合があり (たとえば、複数のテーブルで繰り返されているカラム)、テーブルの正規化が必要な場合があります。

正規化によって、リレーショナルデータベース内の重複データは減少します。たとえば、従業員が複数の異なるオフィスに勤務しているとします。データベースを正規化するには、住所や代表電話番号などオフィスに関する情報を、各従業員用にコピーせず、個別のテーブルに入れておくことを考えます。

重複する情報量が少ない場合は、情報をコピーし、トリガなどの制約を使用して整合性を維持する方が良い場合があります。

参照

- 「データベースの作成」『SQL Anywhere サーバ データベース管理』

ヒント：カスケードされた参照アクションを最小限に抑える

カスケードされた参照アクションは、コストが高くなります。これは、トランザクションごとに複数のテーブルが更新され、パフォーマンスに影響を与えるためです。たとえば、Employees テーブルから Departments テーブルへの外部キーが ON UPDATE CASCADE を使用して定義されている場合、department ID を更新すると Employees テーブルが自動的に更新されます。カスケードされた参照アクションは便利ですが、アプリケーションの論理で実装する方がより効率的な場合があります。

参照

- 「データ整合性」833 ページ

ヒント：制約の宣言

異なるテーブルのカラム値間に暗示的な関係がある場合、テーブル間には、宣言されていないプライマリキー － 外部キー関係が存在します。関係を宣言しないとインデックスの管理に必要な時間を節約できますが、関係を宣言すると、ジョインが発生するときのクエリのパフォーマンスを改善できます。これは、コストモデルの推定精度が上がるためです。

参照

- 「テーブルとカラム制約」843 ページ

ヒント：異なるファイルの異なるデバイスへの配置

ディスクドライブは、最新のプロセッサや RAM よりオペレーション速度が非常に遅くなります。しばしば、ディスクによるページの読み込みまたは書き込みをただ待っていることが、データベースサーバの処理を低速にする原因になっています。

異なるデータベースファイルを異なる物理デバイスまたはドライブに入れると、データベースのパフォーマンスが向上する場合があります。たとえば、1 つのディスクドライブがキャッシュとデータベースページとの相互スワップを実行中の場合も、別のドライブはログファイルに書き込みができます。このようなパフォーマンスの向上を得るためには、各ドライブを独立させます。小さい論理ドライブに分割した単一のディスクでは、利点はありません。

SQL Anywhere では、「データベースファイル」、「トランザクションログファイル」、「トランザクションログミラー」、および「テンポラリファイル」 の 4 種類のファイルが使用されます。. これらのファイルは別々のドライブに置かれている必要があります。

メディア障害から保護するために、データベースファイルとトランザクションログファイルは別の物理ドライブに格納することをおすすめします。

トランザクションログミラーファイルとテンポラリファイルを別の物理ドライブに格納すると、SQL Anywhere の高速化に役立つ場合があります。トランザクションログファイルとトランザク

ションログミラーファイルが別のドライブに格納されていると、SQL Anywhere の書き込みが効率化されます。データベースサーバでテンポラリファイルが使用される必要があるとき、データベース全体のパフォーマンスはテンポラリファイルが格納されるドライブの速度に大きく依存します。テンポラリファイルを使用する操作の多くでは、データベースからも情報を取得する必要があるため、テンポラリファイルを別のドライブに格納することによって、それらの操作を同時に実行できるようになります。

1 つのデータベースは 13 個までのファイルに保管されます (メインファイル 1 つと 12 の DB 領域)。これらは複数の別のドライブに配置できます。テーブルを個別の DB 領域に配置して、一般的なジョイン操作によって異なる DB 領域から情報を呼び込めるようにします。

すべてのテーブルやインデックスをシステム DB 領域以外のロケーションに作成した場合、システム DB 領域は、チェックポイントログとシステムテーブルの保存にのみ使用されます。この設定は、パフォーマンス上の理由からチェックポイントログを他のデータベースオブジェクトとは別のドライブに保存する場合に便利です。別の DB 領域にベーステーブルを作成するには、IN DBSPACE 句を使用するようにすべての CREATE TABLE 文を変更して、代わりの DB 領域を指定するか、テーブルを作成する前に default_dbspace オプションの設定を変更します。テンポラリテーブルは、TEMPORARY DB 領域だけに作成できます。

同様の方式では、テンポラリファイルとデータベースファイルの RAID デバイスまたはストライプセットへの配置があります。これらのデバイスは論理ドライブとして動作しますが、ファイルを多くの物理デバイスに分散し、複数のヘッドを使用して情報にアクセスすることによってパフォーマンスを飛躍的に向上させます。

データベースサーバの起動時に -fc オプションを指定して、データベースサーバでファイルシステムがいっぱいになった場合のコールバック関数を実装できます。

参照

- 「データベースファイルのタイプ」『SQL Anywhere サーバ データベース管理』
- 「トランザクションログユーティリティ (dblog)」『SQL Anywhere サーバ データベース管理』
- 「トランザクションログの場所の変更 (Sybase Central の場合)」『SQL Anywhere サーバ データベース管理』
- 「トランザクションログの場所の変更 (コマンドラインの場合)」『SQL Anywhere サーバ データベース管理』
- 「ヒント：クエリ処理におけるワークテーブルの使用 (All-rows 最適化ゴールの使用)」 230 ページ
- 「バックアップとデータリカバリ」『SQL Anywhere サーバ データベース管理』
- 「SATMP 環境変数」『SQL Anywhere サーバ データベース管理』
- 「メディア障害」『SQL Anywhere サーバ データベース管理』
- 「追加の DB 領域の考慮事項」『SQL Anywhere サーバ データベース管理』
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「default_dbspace オプション」『SQL Anywhere サーバ データベース管理』
- 「-fc データベースサーバオプション」『SQL Anywhere サーバ データベース管理』

ヒント：データベースの再構築

データベースの再構築は、データベース全体をアンロードし、再ロードするプロセスです。再構築は、データベースファイルフォーマットのアップグレードとも呼ばれます。

再構築すると、データやスキーマを含むすべての情報が削除され、同一の形式ですべての情報が元に戻されます。ディスクドライブのデフラグメンテーションと同じように、空き領域が埋められてパフォーマンスが改善されます。また、再構築のタイミングで特定の設定を変更することもできます。

参照

- 「データベースの再構築」749 ページ

ヒント：キーを使ったクエリのパフォーマンス改善

プライマリキーと外部キーは、主に検証に使用されますが、データベースのパフォーマンスを改善することもできます。

例

次の例は、クエリを高速で実行するためのプライマリキーの使用方法を示します。

```
SELECT *
FROM Employees
WHERE EmployeeID = 390;
```

データベースサーバがこのクエリを実行する一番簡単な方法は、Employees テーブルの 75 のローすべてを調べ、各ローの EmployeeID 番号が 390 であるかどうかをチェックすることです。従業員が 75 人しかいなければあまり時間はかかりませんが、何千ものエントリがあるテーブルでは長時間かかってしまいます。

各プライマリキーまたは外部キーによって埋め込まれる参照整合性制約は、各キーの宣言で暗黙的に作成されるインデックスの助けを借りて、SQL Anywhere によって確保されます。EmployeeID カラムは Employees テーブルのプライマリキーです。対応するプライマリキーインデックスによって、従業員番号 390 をすばやく検索できます。この検索は、Employees テーブルのローの数が 100 の場合も、1000000 の場合もほとんど同じ時間で行うことができます。

プライマリキーと外部キーについて、インデックスが個別に自動的に作成されます。これによって、SQL Anywhere で多数のオペレーションを効率的に実行できます。

参照

- 「テーブル間の関係」『SQL Anywhere 16 紹介』

ヒント：プライマリキー幅の縮小

幅の広いプライマリキーは、2 つ以上のカラムで構成されます。プライマリキーに含まれるカラム数が多いほど、データベースサーバに負荷がかかります。プライマリキーに含まれるカラム数を減らすと、パフォーマンスが改善されます。

参照

- 「プライマリキー」17 ページ
- 「ヒント：キーを使ったクエリのパフォーマンス改善」249 ページ

ヒント：テーブル幅の縮小

結合されたカラム (または個々のローのサイズ) がデータベースページサイズを超えるため、複数のデータベースページに分割する必要があるテーブルは、ワイドテーブルと呼ばれます。ローが多数のページに分割されるほど、データベースサーバは各ローの読み込みに時間がかかります。ワイドテーブルが存在し、パフォーマンスが低下している場合は、テーブルをさらに正規化してカラム数を減らすことを検討してください。テーブルを正規化できない場合は、データベースページサイズを拡大するとパフォーマンスが改善されることがあります。特にほとんどのテーブルの幅が広い場合は有効です。

参照

- 「データベースの作成」『SQL Anywhere サーバ データベース管理』

ヒント：テーブル内のカラムの順序の確認

テーブル内のカラムの順序はパフォーマンスに影響します。ローのカラムは、作成された順に逐次方式でアクセスされます。たとえば、ローの末尾のカラムにアクセスするために、データベースサーバは、ロー内でそのカラムより前にあるカラムも処理しなければなりません。そのため、サイズの小さいカラムや頻繁にアクセスされるカラムが、アクセス頻度の低いカラムや幅の広いカラムより前に配置されるように、テーブルのカラムの順序を配列する必要があります。

幅の広いカラムとは、15 バイトより大きなサイズのカラム、LONG 型データのカラム (LONG VARCHAR など)、または XML として定義されたカラムを指します。プライマリキーカラムは、常にローの先頭に格納されます。

参照

- 「データベースの作成」『SQL Anywhere サーバ データベース管理』
- 「プライマリキー」17 ページ

ヒント：コストの高いトリガを置き換える

使用しているトリガを評価して、データベースサーバで利用できる機能で置き換えられるトリガはないかを確認します。たとえば、最終更新時間とユーザ情報でカラムを更新するトリガは、データベースサーバ内の対応する特別値で置き換えられます。また、既存のトリガにデフォルト設定を使用することでも、パフォーマンスを改善できます。

参照

- 「トリガ」94 ページ

ヒント：適切なページサイズの使用

選択したページサイズが、データベースのパフォーマンスに影響することがあります。大きいページサイズにも小さいページサイズにも、それぞれ利点と欠点があります。

SQL Anywhere は、ページをできるだけ埋めようとします。空き領域が累積するのは、新しいオブジェクトが大きすぎて既存のページの空き領域に収まらない場合だけです。したがって、ページサイズを調整しても、データベース全体のサイズに大きく影響することはありません。

ページサイズを選択するときは、パフォーマンスとその他の動作をテストすることを強くおすすめします。そして、満足できる結果を得られた最小のページサイズを選択します。同じサーバで複数のデータベースが起動される場合は、正しく適切なページサイズを選択することが重要です。

小さいページは保持できる情報量が少ないため、特に、ページの半分よりも少し大きいサイズのローを挿入する場合に領域の使用効率が低下します。ただし、ページサイズが小さいと、同じサイズのキャッシュにより多くのページを格納できるため、SQL Anywhere をより少ないリソースで実行できます。メモリが限られている小型のコンピュータでデータベースを実行する場合に便利です。また、不特定のロケーションから少量の情報を取得するためにデータベースを使う場合にも便利です。

ページサイズが大きいと、SQL Anywhere はデータベースをより効率的に読み込めます。データベースの規模が大きい場合や、逐次テーブルスキャンを実行するクエリの場合にパフォーマンスが高くなる傾向があります。通常は、ディスクの物理的な設計のために、大きなブロックを少数取り出す方が、小さなブロックを多数取り出す場合よりも効率的です。大きいページサイズには、他の利点もあります。インデックスのファンアウトを改善することによってインデックスレベル数を減らし、カラム数の多いテーブルが作成できます。大きなページサイズ選択すると、同じサイズのキャッシュに収まるページ数が少なくなるので、キャッシュサイズを大きくする必要がある場合があります。キャッシュに十分なページを格納できない場合、データベースサーバは頻繁に使用されるページをディスクにスワップし始めるため、パフォーマンスが低下します。

ページサイズが大きいと、メモリも余分に必要になります。また、大容量のデータベースサーバキャッシュを常に利用できないかぎり、大半のアプリケーションでは、極端に大きなページサイズ (16 KB または 32 KB) の使用はおすすめしません。

データベースサーバのメモリの使用状況は、ロードされたデータベース数とデータベースのページサイズに比例します。ページサイズを選択するときは、パフォーマンステスト (およびテスト全般) を実行することを強くおすすめします。その上で、満足できる結果を得られた最小のページサイズ (4 KB 以上) を選択します。同じサーバで多数のデータベースが起動される場合は、正しい (かつ適切な) ページサイズの選択が重要です。

既存のデータベースのページサイズは変更できません。その代わり、新しいデータベースを作成し、dbinit の -p オプションを使用してページサイズを指定します。たとえば、次のコマンドは 4 KB のページサイズのデータベースを作成します。

```
dbinit -p 4096 new.db
```

新しいページサイズでデータベースを作成するために、PAGE SIZE 句を指定した CREATE DATABASE 文を使用することもできます。

SQL Anywhere では、DB 領域ファイル全体での各テーブルページの位置を示すビットマップがテーブルごとに作成されます。データベースサーバでは、テーブルページが 1 ページずつ読み込まれるのではなく、このビットマップを使用して大きなブロック (64 KB) で読み込まれます。「グループ読み込み」と呼ばれるこの方法で、ディスクへの I/O 操作の合計数が減り、パフォーマン

スが向上します。ユーザは、ビットマップの作成や使用に関するデータベースサーバの条件を制御できません。

ページサイズとインデックス

ページサイズは、インデックスにも影響を与えます。各インデックスルックアップでは、インデックスのレベルごとに 1 ページを読み込み、さらにテーブルページについて 1 ページを読み込む必要があります。また、1 回のクエリには数千のインデックスルックアップが必要になることがあります。ページのサイズはファンアウトにかなりの影響を与え、またテーブルに必要なインデックスの深さに影響します。大きなファンアウトは、通常、必要なインデックスレベル数が少ないことを意味し、検索パフォーマンスを大幅に向上できます。テーブル内のローの数が多い大規模なデータベースでは、高パフォーマンスのために 1 ページを 8 KB にできます。

分散読み込み

Windows システムでは、最小のページサイズ 4 KB を使用すると、データベースサーバはディスク上で連続した広範囲のデータベースページをキャッシュ内の適切な場所に直接読み込んで、64 KB のバッファを完全に回避できます。この機能によって、パフォーマンスを大幅に向上できます。

注意
リモートコンピュータ上のファイルや UNC 名 (たとえば *¥¥mycomputer¥myshare¥mydb.db*) で指定されたファイルに対して、分散読み込みは使用されません。

参照

- 「最大ページサイズの考慮事項」『SQL Anywhere サーバ データベース管理』
- 「CREATE DATABASE 文」『SQL Anywhere サーバ SQL リファレンス』

ヒント：オートインクリメントを使用したプライマリキーの作成

プライマリキーの値はユニークである必要があります。プライマリキーにユニークな値を作成する方法は何通りもありますが、最も効率的な方法は、デフォルトカラム値にオートインクリメント (AUTOINCREMENT) を設定することです。このデフォルト設定は、ユニークな値を管理するカラムのすべてに使用できます。オートインクリメント機能を使用したプライマリキーの作成は、値がデータベースサーバによって生成されるため、他のどの方法よりも高速です。

参照

- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

ヒント：適切なデータ型の使用

データ型は、値の範囲、値に対して可能な演算、メモリでの値の格納方法など、特定のデータセットに関する情報を格納します。データに適切なデータ型を使用することで、パフォーマンスを改善できます。たとえば、数値データのみを含む値にデータ型 CHAR を割り当てないでくだ

さい。また、コストの高い数値型や文字列型でなく、効率的なデータ型をできるだけ選択してください。

参照

- 「SQL データ型」『SQL Anywhere サーバ SQL リファレンス』

ヒント：バルクオペレーション方法の使用

非常に大量の情報をデータベースにロードする場合、各タスクに対する専用ツールを使用すると便利です。

大きなファイルをロードする場合は、データのロード後にテーブルにインデックスを作成するとより効率的になります。

参照

- 「バルクオペレーションのパフォーマンスの側面」709 ページ

ヒント：リソースガバナを使用する

ユーザとロールのセットを作成すると、データベースの権限を管理できます。またデータベースのセキュリティと管理によって、個々のユーザが使用できるリソースを制限することもできます。

たとえば、他のユーザのデータベースへの接続速度を落とさないように、1 つの接続がメモリや CPU を大量に使用することを防止できます。

SQL Anywhere には、リソースの制御に使用できるデータベースオプションが備わっています。このオプションを「リソースガバナ」といいます。これらのオプションは、SET OPTION 文を使用して設定できます。

管理できるリソース

次のオプションを使用して、リソースを管理できます。

- **max_cursor_count**　接続用のカーソルの数を制限します。
- **max_statement_count**　接続用の準備文の数を制限します。
- **priority**　接続からの要求を実行する優先度レベルを設定します。
- **max_priority**　接続の最高優先度レベルを制御します。

参照

- 「max_cursor_count オプション」『SQL Anywhere サーバ データベース管理』
- 「max_statement_count オプション」『SQL Anywhere サーバ データベース管理』
- 「priority オプション」『SQL Anywhere サーバ データベース管理』
- 「max_priority オプション」『SQL Anywhere サーバ データベース管理』
- 「データベースオプション」『SQL Anywhere サーバ データベース管理』
- 「SET OPTION 文」『SQL Anywhere サーバ SQL リファレンス』

ヒント：クライアントとサーバとの間の要求数の削減

ネットワークの遅延時間が長いか、アプリケーションでカーソルを開く要求と閉じる要求を多数送信する場合は、ネットワーク接続パラメータ LazyClose と PrefetchOnOpen を使用して、クライアントとサーバ間の要求数を減らし、その結果、パフォーマンスを向上できます。

参照

- 「LazyClose (LCLOSE) 接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「PrefetchOnOpen 接続パラメータ」『SQL Anywhere サーバ データベース管理』

ヒント：圧縮の慎重な使用

接続用パケットの圧縮

1 つまたはすべての接続について圧縮機能を有効にして、パケット圧縮時の最小サイズ制限を調整すると、パフォーマンスを大幅に向上できる場合があります。

圧縮を有効にすることに効果があるかどうかを判断するために、アプリケーションを使用してネットワークのパフォーマンス分析を行ってから、運用環境で通信の圧縮を使用します。

圧縮機能を有効にすると、データパケットに格納される情報量が増大し、特定のデータセットの送信に必要なパケット数が減少します。パケット数を減らすと、データを高速で送信できます。

圧縮しきい値を指定すると、圧縮対象になるデータパケットの最小サイズを選択できます。圧縮しきい値の最適値は、使用するネットワークのタイプや速度など、さまざまな要素の影響を受けることがあります。

データベースの圧縮

圧縮レイヤによって IO 操作のコストが大幅に増加することがあるため、データベースおよびログファイルのファイルレベルまたはディスクレベルの圧縮を使用することはおすすめしません。

参照

- 「通信の圧縮設定」『SQL Anywhere サーバ データベース管理』
- 「Compress (COMP) 接続パラメータ」『SQL Anywhere サーバ データベース管理』
- 「CompressionThreshold (COMPTH) 接続パラメータ」『SQL Anywhere サーバ データベース管理』

ヒント：パケットサイズを変更してパフォーマンスを向上させる

多くの場合、パケットサイズを大きくすることで同等またはパフォーマンスの向上が期待できます。場合によっては、パフォーマンスが低下します。パケットサイズを大きくすると、クライアントとデータベースサーバが使用するメモリの量も増えます。運用環境では、調整を行う前に、アプリケーションを使用してネットワークのパフォーマンス分析を行ってください。

パケットサイズを大きくすると、データベース要求の応答時間を短縮できます。特に、クライアントとデータベースサーバの間で大量のデータを転送する要求では大変有効です。たとえば、大きな結果セット (64 KB 以上) を多数持つアプリケーション、または多数の大きな (64 KB 以上) BLOB を高速なローカルネットワーク上で転送するアプリケーションでは、パケットサイズをデフォルトよりも大きくすることで、パフォーマンスが大きく向上する場合があります。

パケットサイズを設定するには、-p dbeng16/dbsrv16 サーバオプションを使用するか、接続文字列に CommBufferSize (CBSIZE) 接続パラメータを設定します。

参照

- 「-p データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「CommBufferSize (CBSIZE) 接続パラメータ」『SQL Anywhere サーバ データベース管理』
- PacketSize 接続プロパティ『SQL Anywhere サーバ データベース管理』

アプリケーションプロファイリングのチュートリアル

アプリケーションプロファイリングのチュートリアルでは、**[アプリケーションプロファイリングウィザード]**と**[データベーストレーシングウィザード]**を使用して、デッドロック、速度の遅い文、インデックスの断片化、テーブルの断片化、低速のプロシージャなどの一般的なパフォーマンス問題を分析する方法を学習します。

> **警告**
> アプリケーションプロファイリングのチュートリアルでは、サンプルデータベース (*demo.db*) ではなく、テストデータベース *app_profiling.db* を使用します。チュートリアルを実行する場合は、サンプルデータベースは使用しないでください。

チュートリアル：アプリケーションプロファイリングのチュートリアル用のテストデータベースの作成

アプリケーションプロファイリングのチュートリアルに使用する *app_profiling.db* というテストデータベースを作成します。

前提条件

この作業を実行するための前提条件はありません。

♦ タスク

1. ディレクトリ *c:¥AppProfilingTutorial* を作成します。
2. 次のコマンドを実行して、サンプルデータベースのデータを含むテストデータベース *app_profiling.db* を作成します。

```
newdemo c:¥AppProfilingTutorial¥app_profiling.db
```

結果

app_profiling.db データベースが作成されます。

参照

- 「アプリケーションプロファイリングウィザードの使用」149 ページ
- 「診断トレーシング」162 ページ
- 「サンプルデータベースの再作成 (demo.db)」『SQL Anywhere 16 紹介』
- 「チュートリアル：サンプルデータベースへの接続」『SQL Anywhere サーバ データベース管理』

チュートリアル：デッドロックの診断

このチュートリアルのレッスンでは、**データベーストレーシングウィザード**を使用して、データベースに発生するデッドロックを参照する方法を学習します。**データベーストレーシングウィザード**を使用すると、デッドロックが発生した状態や、デッドロック発生の原因となっている接続を調査することもできます。

権限

このチュートリアルにはユーザ本人が作成したテストデータベース、*app_profiling.db* が使用されるので、このチュートリアルの実行に権限は必要がありません。デフォルトのユーザ、DBA として接続します。

レッスン 1：データベースのセットアップ

アプリケーションプロファイリングのテストデータベース *app_profiling.db* を設定して、2 つのテーブルと 2 つのプロシージャを作成することにより、デッドロックを作成します。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：デッドロックの診断」

このチュートリアルは、テストデータベース *app_profiling.db* が作成されていることを前提としています。「チュートリアル：アプリケーションプロファイリングのチュートリアル用のテストデータベースの作成」255 ページを参照してください。

内容と備考

デッドロックは、複数のトランザクションが互いにブロックされている場合に発生します。たとえば、トランザクション A がテーブル B にアクセスする必要があるが、テーブル B はトランザクション B によってロックされ、トランザクション B はテーブル A にアクセスする必要があるが、テーブル A はトランザクション A によってロックされている場合などです。これは環状ブロッキングの競合が発生している状態です。

デッドロックが発生している状況は、SQLCODE -306 と -307 が返されることで分かります。デッドロックを解決するために、SQL Anywhere はデッドロックが作成された最後の文を自動的にロールバックします。文が絶え間なくロールバックされる場合は、パフォーマンス問題が発生しています。

♦ タスク

1. Interactive SQL を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Sybase Central]** をクリックします。

2. Interactive SQL で、次のように *app_profiling.db* に接続します。
   a. **[接続]** ウィンドウで、テストデータベース *app_profiling.db* に接続するための情報を以下のフィールドに入力します。
      i. **[認証]** フィールドで **[データベース]** を選択します。
      ii. **[ユーザ ID]** フィールドに **DBA** と入力します。
      iii. **[パスワード]** フィールドに **sql** と入力します。
      iv. **[アクション]** ドロップダウンリストで、**[このコンピュータのデータベースを起動して接続]** をクリックします。
      v. **[データベースファイル]** フィールドに、**C:¥AppProfilingTutorial¥app_profiling.db** と入力します。
      vi. **[開始行]** フィールドに、**dbeng16 -x tcpip** と入力します。
   b. **[接続]** をクリックします。

3. Interactive SQL で、次の SQL 文を実行します。
   a. テーブルを 2 つ作成します。

```
CREATE TABLE "DBA"."deadlock1" (
  "id" UNSIGNED BIGINT NOT NULL DEFAULT AUTOINCREMENT,
  "val" CHAR(1) );
CREATE TABLE "DBA"."deadlock2" (
  "id" UNSIGNED BIGINT NOT NULL DEFAULT AUTOINCREMENT,
  "val" CHAR(1) );
```

   b. 各テーブルに値を挿入します。

```
INSERT INTO "deadlock1"("val") VALUES('x');
INSERT INTO "deadlock2"("val") VALUES('x');
```

   c. 次の 2 つのプロシージャを作成します。

```
CREATE PROCEDURE "DBA"."proc_deadlock1"( )
  BEGIN
```

```
    LOCK TABLE "DBA"."deadlock1" IN EXCLUSIVE MODE;
    WAITFOR DELAY '00:00:20:000';
    UPDATE deadlock2 SET val='y';
  END;
CREATE PROCEDURE "DBA"."proc_deadlock2"( )
  BEGIN
    LOCK TABLE "DBA"."deadlock2" IN EXCLUSIVE MODE;
    WAITFOR DELAY '00:00:20:000';
    UPDATE deadlock1 SET val='y';
  END;
```

d. データベースに加えた変更をコミットします。

```
COMMIT;
```

4. Interactive SQL を閉じます。

結果

デッドロックの作成に使用する 2 つのテーブルと 2 つのプロシージャが作成されます。

次の手順

「レッスン 2：デッドロックの作成とデータの取得」258 ページに進みます。

参照

- 「ローカルコンピュータ上のデータベースの起動と接続 (Sybase Central または Interactive SQL の場合)」『SQL Anywhere サーバ データベース管理』
- 「デッドロックが検出されました。」『エラーメッセージ』
- 「すべてのスレッドがブロックされています。」『エラーメッセージ』

レッスン 2：デッドロックの作成とデータの取得

データベーストレーシングウィザードを使用して診断トレーシングセッションを作成することにより、デッドロックを作成してその情報を取得します。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：デッドロックの診断」

このレッスンは、受講者がこれまでのすべてのレッスンを終了していることを前提としています。「レッスン 1：データベースのセットアップ」256 ページ を参照してください。

内容と備考

アプリケーションプロファイリングのチュートリアルでは、チュートリアルを実行しているデータベースと同じテストデータベース (*app_profiling.db*) にトレーシング情報が格納されます。ただし、負荷の高いデータベースをプロファイリングする場合は、パフォーマンスへの影響を避けるため、運用データベースとは別のデータベースにトレーシングデータを格納することを検討してください。

♦ タスク

1. Sybase Central を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Sybase Central]** をクリックします。

2. Sybase Central で、次のように *app_profiling.db* に接続します。
   a. **[接続]** » **[SQL Anywhere 16 に接続]** をクリックします。
   b. **[接続]** ウィンドウで、テストデータベース *app_profiling.db* に接続するための情報を以下のフィールドに入力します。
      i. **[認証]** フィールドで **[データベース]** を選択します。
      ii. **[ユーザ ID]** フィールドに、**DBA** と入力します。
      iii. **[パスワード]** フィールドに、**sql** と入力します。
      iv. **[アクション]** ドロップダウンリストで、**[このコンピュータのデータベースを起動して接続]** をクリックします。
      v. **[データベースファイル]** フィールドに、**C:¥AppProfilingTutorial¥app_profiling.db** と入力します。
      vi. **[開始行]** フィールドに、**dbeng16 -x tcpip** と入力します。
   c. **[接続]** をクリックします。

3. Sybase Central で、**[モード]** » **[アプリケーションプロファイリング]** をクリックします。

   アプリケーションプロファイリングウィザードが表示された場合は、**[キャンセル]** をクリックします。

4. データベーストレーシングウィザードを起動します。
   a. **[ファイル]** » **[トレーシング]** をクリックします。
   b. **[ようこそ]** ページで、**[次へ]** をクリックします。
   c. **[トレーシング詳細レベル]** ページで、**[高 (短期間の集中的なモニタリングに推奨)]** をクリックし、**[次へ]** をクリックします。
   d. **[トレーシングレベルの編集]** ページで、**[次へ]** をクリックします。
   e. **[外部データベースの作成]** ページで、**[新しいデータベースを作成せず、既存のトレーシングデータベースを使用]** を選択し、**[次へ]** をクリックします。
   f. **[トレースの開始]** ページで、**[このデータベースにトレーシングデータを保存]** をクリックします。
   g. 格納するトレーシングデータの量を制限しない場合は、**[制限なし]** をクリックし、**[完了]** をクリックします。

5. 次の手順に従って、デッドロックを発生させます。
   a. 左ウィンドウ枠で、**app_profiling - DBA** が選択されていることを確認してから、**[ファイル]** » **[Interactive SQL を開く]** をクリックします。

      Interactive SQL が起動し、*app_profiling.db* データベースに接続します。

b. 前の手順を繰り返し、2 番目の Interactive SQL ウィンドウを開きます。

c. 最初の Interactive SQL ウィンドウで、次の SQL 文を実行します。

```
CALL "DBA"."proc_deadlock1"();
```

d. 最初の Interactive SQL ウィンドウで SQL 文を実行してから 20 秒以内に、2 番目の Interactive SQL ウィンドウで次の SQL 文を実行します。

```
CALL "DBA"."proc_deadlock2"();
```

しばらくすると、デッドロックが検出されたことを示す **[ISQL エラー]** ウィンドウが表示されます。

proc_deadlock1 は deadlock2 テーブルにアクセスする必要がありますが、このテーブルが proc_deadlock2 によってロックされていることによって、このデッドロックが発生しました。同様に、proc_deadlock2 は deadlock1 テーブルにアクセスする必要がありますが、このテーブルも proc_deadlock1 によってロックされています。

e. **[OK]** をクリックします。

6. デッドロックした操作は SQL Anywhere によって停止されたため、Interactive SQL ウィンドウを閉じることができます。

7. Sybase Central の左ウィンドウ枠で **app_profiling - DBA** を選択し、**[ファイル]** » **[トレーシング]** » **[トレーシングを停止して保存]** をクリックして、トレーシングセッションを停止します。

結果

デッドロックが作成され、その情報が収集されます。

次の手順

「レッスン 3：ブロックされた接続データの確認」260 ページに進みます。

参照

- 「トランザクションのブロックとデッドロック」881 ページ
- 「カスタマイズされた診断トレーシングレベル」167 ページ
- 「デッドロック」883 ページ
- 「診断トレーシング」162 ページ

レッスン 3：ブロックされた接続データの確認

[アプリケーションプロファイリング] モードを使用すると、デッドロックに関係する接続がグラフィックで表示されます。**アプリケーションプロファイリング**モードでは、**[接続のブロック]** タブに、ブロックされた接続に関する追加情報も表示されます。

前提条件

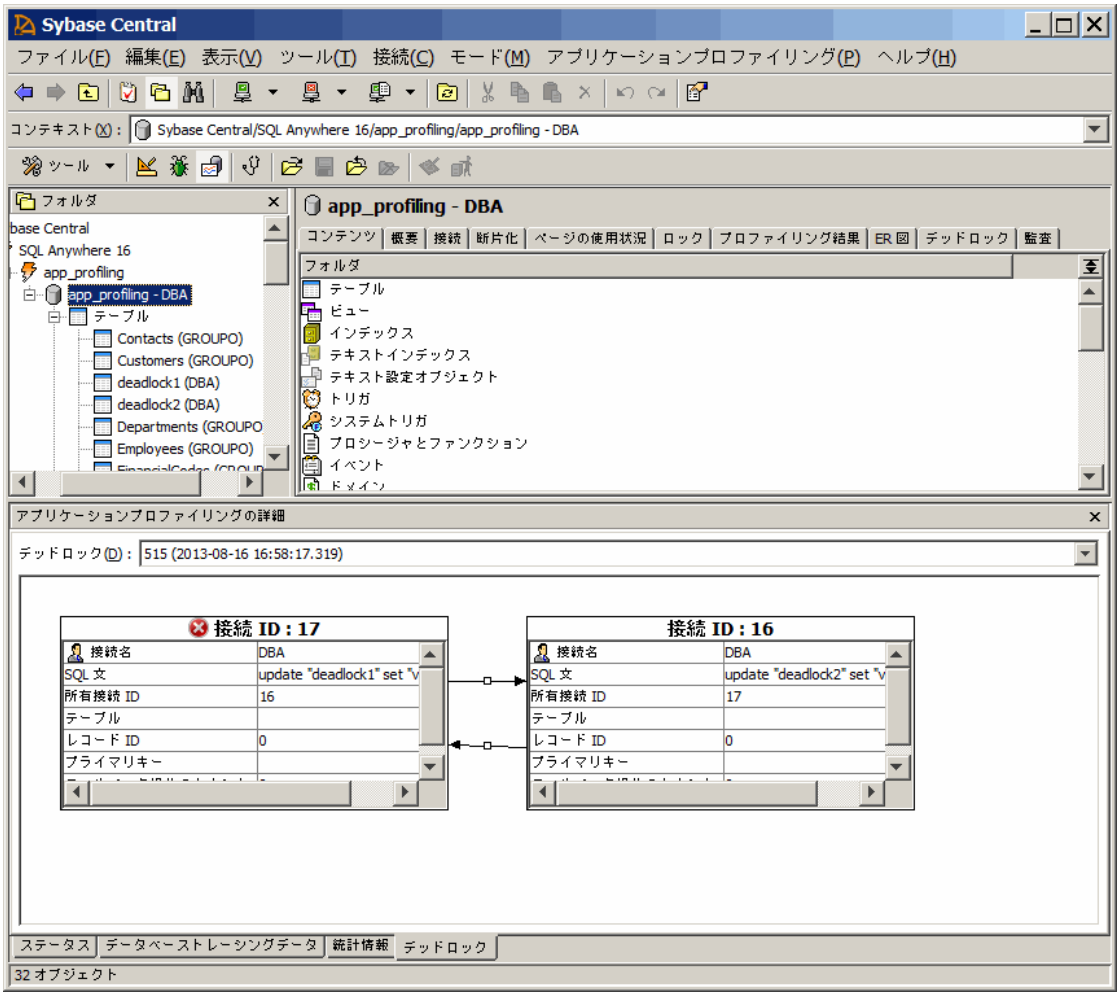
このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：デッドロックの診断」

このレッスンは、受講者がこれまでのすべてのレッスンを終了していることを前提としています。「レッスン 1：データベースのセットアップ」256 ページ を参照してください。

♦ タスク

1. 次の手順に従って、トレーシングセッション中に作成された分析ファイルを開きます。
   a. Sybase Central で、**[アプリケーションプロファイリング]** » **[分析ファイルを開くかトレーシングデータベースに接続]** をクリックします。
   b. **[トレーシングデータベース内]** をクリックします。
   c. **[開く]** をクリックします。
   d. **[ユーザ ID]** フィールドに、**DBA** と入力します。
   e. **[パスワード]** フィールドに、**sql** と入力します。
   f. **[アクション]** ドロップダウンリストで、**[このコンピュータで稼働しているデータベースに接続]** をクリックします。
   g. **[データベース名]** フィールドに、**app_profiling** と入力します。
   h. **[接続]** をクリックします。
2. 次の手順に従って、デッドロックのグラフィック表示を参照します。
   a. **[アプリケーションプロファイリングの詳細]** ウィンドウ枠で、**[ステータス]** タブをクリックし、**[ロギングセッション ID]** リストから最新の ID を選択します。

      [アプリケーションプロファイリングの詳細] ウィンドウ枠が表示されない場合は、**[表示]** » **[アプリケーションプロファイリングの詳細]** をクリックします。
   b. **[アプリケーションプロファイリングの詳細]** ウィンドウ枠の下部で、**[デッドロック]** タブをクリックします。最新のデッドロックが表示されます。別のデッドロックを参照するには、**[デッドロック]** リストをクリックします。
   c. 次の図は、UPDATE 文により、どのようにデッドロック状態が作成されたかを示しています。



デッドロックに関連する各接続は、次のフィールドが含まれるテーブルで表示されます。

- **[接続名]**　接続を開いたユーザ ID。
- **[SQL 文]**　デッドロックに関係する文。この場合、Interactive SQL の各インスタンスから実行したプロシージャで検出された UPDATE 文が原因でデッドロックが発生しました。
- **[所有接続 ID]**　現在の接続をブロックした接続の ID。
- **[レコード ID]**　現在の接続がブロックされたローの ID。
- **[ロールバック操作のカウント]**　デッドロックの結果としてロールバックされる必要がある操作回数。この場合、プロシージャには UPDATE 文しか含まれていないため、この数は 0 になります。

3. 追加のデッドロック情報 (発生頻度、継続時間など) を表示するには、次のように **[接続のブロック]** タブを使用します。

   a. **[アプリケーションプロファイリングの詳細]** ウィンドウ枠で、**[データベーストレーシングデータ]** タブをクリックします。

b. **[データベーストレーシングデータ]** の上にある **[接続のブロック]** タブをクリックします。

   [接続のブロック] タブが開き、ブロック時間、ブロックが解除された時間、各接続に対するブロックの継続時間が表示されます。

結果

デッドロックの診断についてのアプリケーションプロファイリングのチュートリアルを終了しました。

参照

- 「トランザクションのブロックとデッドロック」881 ページ
- 「カスタマイズされた診断トレーシングレベル」167 ページ
- 「デッドロック」883 ページ
- 「診断トレーシング」162 ページ

チュートリアル：速度が遅い文の診断

このチュートリアルのレッスンでは、**データベーストレーシングウィザード**を使用して文の実行時間を表示する方法や、(実行時間が長いクエリなど) 実行速度が遅い文を識別する方法について学習します。

速度が遅い文は、データベースサーバがその文の処理に長時間かかる場合に発生します。処理時間が長い場合、このことは、データベースが正しく設計されていない、インデックスの使用方法が不適切、インデックスとテーブルの断片化、キャッシュサイズが小さいなど、いくつかの問題が原因で発生する可能性があります。また、文の形式が適切でないこと、または、より効率的に結果を得ることができるショートカットが使用されていないことによって、文の実行速度が遅くなる場合もあります。

このチュートリアルでは、各文に特別な要件がある場合があるため、速度の遅い文の書き換え方法は示していません。ただし、このチュートリアルでは、代替構文を使用してクエリを書き換えたときに、実行時間を検出する場所や、実行時間を比較する方法を示します。

権限

このチュートリアルにはユーザ本人が作成したテストデータベース、*app_profiling.db* が使用されるので、このチュートリアルの実行に権限は必要がありません。デフォルトのユーザ、DBA として接続します。

参照

- 「クエリ」281 ページ
- 「ジョイン：複数テーブルからのデータ検索」488 ページ
- 「サブクエリの使用」594 ページ
- 「カスタマイズされた診断トレーシングレベル」167 ページ
- 「診断トレーシング」162 ページ
- 「ヒント：クエリのパフォーマンスのモニタ」237 ページ
- 「パフォーマンスの問題のトラブルシューティング」180 ページ

レッスン 1：診断トレーシングセッションの作成

データベーストレーシングウィザードを使用して、処理中の文のデータ (継続時間など) を取得する診断トレーシングセッションを作成します。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：速度が遅い文の診断」

このチュートリアルは、テストデータベース *app_profiling.db* が作成されていることを前提としています。「チュートリアル：アプリケーションプロファイリングのチュートリアル用のテストデータベースの作成」255 ページを参照してください。

♦ タスク

1. Sybase Central を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Sybase Central]** をクリックします。

2. Sybase Central で、次のように *app_profiling.db* に接続します。
   a. **[接続]** » **[SQL Anywhere 16 に接続]** をクリックします。
   b. **[接続]** ウィンドウで、テストデータベース *app_profiling.db* に接続するための情報を以下のフィールドに入力します。
      i. **[認証]** フィールドで **[データベース]** を選択します。
      ii. **[ユーザ ID]** フィールドに、**DBA** と入力します。
      iii. **[パスワード]** フィールドに、**sql** と入力します。
      iv. **[アクション]** ドロップダウンリストで、**[このコンピュータのデータベースを起動して接続]** をクリックします。
      v. **[データベースファイル]** フィールドに、**C:¥AppProfilingTutorial¥app_profiling.db** と入力します。
      vi. **[開始行]** フィールドに、**dbeng16 -x tcpip** と入力します。
   c. **[接続]** をクリックします。

3. Sybase Central で、**[モード]** » **[アプリケーションプロファイリング]** をクリックします。

   アプリケーションプロファイリングウィザードが表示された場合は、**[キャンセル]** をクリックします。

4. データベーストレーシングウィザードを起動します。
   a. **[ファイル]** » **[トレーシング]** をクリックします。
   b. **[ようこそ]** ページで、**[次へ]** をクリックします。
   c. **[トレーシング詳細レベル]** ページで、**[高 (短期間の集中的なモニタリングに推奨)]** をクリックし、**[次へ]** をクリックします。
   d. **[トレーシングレベルの編集]** ページで、**[次へ]** をクリックします。

e. **[外部データベースの作成]** ページで、**[新しいデータベースを作成せず、既存のトレーシングデータベースを使用]** を選択し、**[次へ]** をクリックします。

f. **[トレースの開始]** ページで、**[このデータベースにトレーシングデータを保存]** をクリックします。

g. 格納するトレーシングデータの量を制限しない場合は、**[制限なし]** をクリックし、**[完了]** をクリックします。

5. 左ウィンドウ枠で、**app_profiling - DBA** が選択されていることを確認してから、**[ファイル]** » **[Interactive SQL を開く]** をクリックします。

   Interactive SQL が起動し、*app_profiling.db* データベースに接続します。

6. Interactive SQL で、次の SQL 文を実行します。

```
SELECT SalesOrderItems.ID, LineID, ProductID, SalesOrderItems.Quantity, ShipDate
FROM SalesOrderItems, SalesOrders
WHERE SalesOrders.CustomerID = 105 AND
  SalesOrderItems.ID=SalesOrders.ID;
```

7. Interactive SQL を終了します。

8. Sybase Central の左ウィンドウ枠で **app_profiling - DBA** を選択し、**[ファイル]** » **[トレーシング]** » **[トレーシングを停止して保存]** をクリックして、トレーシングセッションを停止します。

結果

トレーシングセッション中の診断情報が収集されます。

次の手順

「レッスン 2：データベースサーバによって処理された文の確認」265 ページに進みます。

参照

- 「カスタマイズされた診断トレーシングレベル」167 ページ
- 「診断トレーシング」162 ページ

レッスン 2：データベースサーバによって処理された文の確認

Sybase Central の**[アプリケーションプロファイリング]** ウィンドウ枠にある **[概要]** タブと **[詳細]** タブを使用すると、データベースサーバが最も処理に時間を要する文を識別できます。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：速度が遅い文の診断」

このレッスンは、受講者がこれまでのすべてのレッスンを終了していることを前提としています。「レッスン 1：診断トレーシングセッションの作成」264 ページ を参照してください。

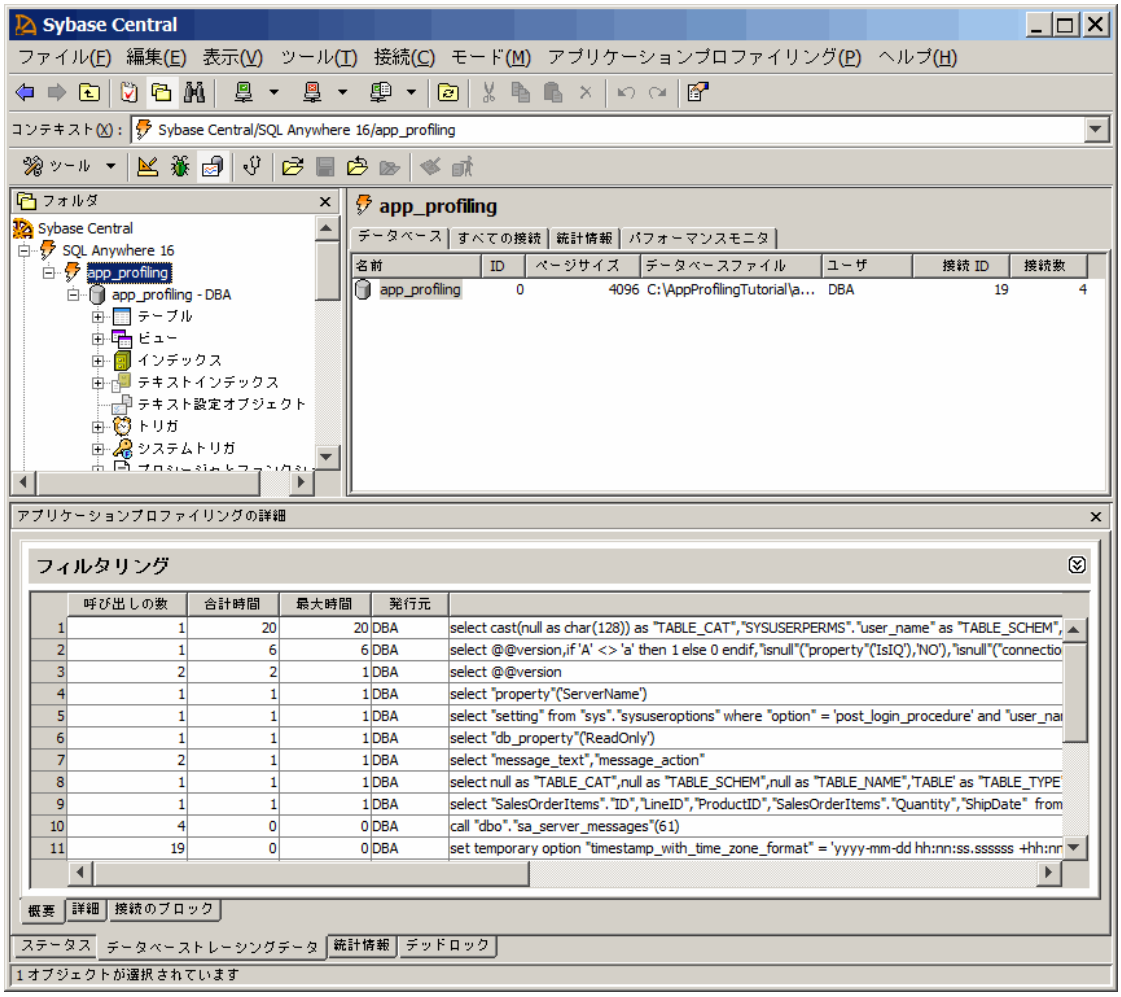
♦ **タスク**

1. Sybase Central で、次の手順に従って分析ファイルを開きます。
   a. **[アプリケーションプロファイリング]** » **[分析ファイルを開くかトレーシングデータベースに接続]** を選択します。
   b. **[トレーシングデータベース内]** をクリックし、**[開く]** をクリックします
   c. **[ユーザ ID]** フィールドに、**DBA** と入力します。
   d. **[パスワード]** フィールドに、**sql** と入力します。
   e. **[アクション]** ドロップダウンリストで、**[このコンピュータで稼働しているデータベースに接続]** をクリックします。
   f. **[データベース名]** フィールドに **app_profiling** と入力します。
   g. **[接続]** をクリックします。
   h. ウィンドウの下部に **[アプリケーションプロファイリングの詳細]** ウィンドウ枠が表示されない場合は、**[表示]** » **[アプリケーションプロファイリングの詳細]** をクリックします。
2. 次の手順に従って、トレーシングセッション中に処理された文の実行時間を調査します。
   a. **[アプリケーションプロファイリングの詳細]** ウィンドウ枠の **[ステータス]** タブで、**[ロギングセッション ID]** フィールドから最新の ID (最も大きな数値) を選択し、**[データベーストレーシングデータ]** タブをクリックします。

      [概要] タブに、そのセッション中に実行した SQL 文が表示されます。別の文も表示される場合があります。これは、実行した文によって他の文が自動的に実行されたためです (トリガなど)。

      [概要] タブでは、似ている文がグループ化され、合計呼び出し回数と合計処理時間がまとめて表示されます。SELECT、INSERT、UPDATE、DELETE の各文は、参照するテーブル、カラム、式ごとに分類されます。その他の文は 1 つにまとめられます。たとえば、CREATE TABLE 文は **[概要]** タブに 1 つのエントリとして表示されます。**[概要]** タブに文のコストが高いと表示された場合は、その文のコストが高いか、その文が頻繁に実行された可能性があります。

      [合計時間] と **[最大時間]** カラムを調べて、実行時間が長い文を見つけます。これらのカラムによって、データベースサーバによって処理される各文の実行時間についての情報が提供されます。



3. 文を実行した接続に関する情報を表示するには、**[詳細]** タブをクリックし、文を右クリックして、**[選択された文に対する接続の詳細を表示]** をクリックします。

4. SQL 文に使用された実行プランを表示するには、**[詳細]** タブをクリックし、文を右クリックして、**[選択された文に対する SQL 文の詳細を表示]** をクリックします。

   [SQL 文の詳細] ウィンドウが開き、文が使用されたコンテキストに関する詳細情報とともに、文の完全なテキストが表示されます。表示される文のテキストは、実行した元の SQL 文とは一致しない場合があります。さらに、**[SQL 文の詳細]** ウィンドウには、データベースサーバによって処理されたときに書き換えられたフォームで文が表示されます。たとえば、ビュー定義はクエリの実行時にオプティマイザによって書き換えられることが多いため、ビューに表示されるクエリは異なる可能性があります。

   [SQL 文の詳細] ウィンドウの下部にある **[クエリ情報]** タブをクリックして、実行プランを表示します。

結果

速度が遅い文の診断のチュートリアルを終了しました。

参照

- 「サブクエリの使用」594 ページ
- 「要求トレース分析の実行」182 ページ
- 「クエリ」281 ページ
- 「ジョイン：複数テーブルからのデータ検索」488 ページ
- 「サブクエリの使用」594 ページ
- 「カスタマイズされた診断トレーシングレベル」167 ページ
- 「診断トレーシング」162 ページ

チュートリアル：インデックスの断片化の診断

このチュートリアルのレッスンでは、**アプリケーションプロファイリングウィザード**または Interactive SQL を使用して、データベースに許容できないレベルのインデックスの断片化が発生しているかどうかを判別し、必要に応じてインデックスの断片化を修正する方法を説明します。

インデックスが作成されると、テーブルデータが読み込まれ、インデックスの値が論理順序に従ってインデックスページに記録されます。テーブル内のデータを変更した場合は、新しいインデックス値を既存の値の間に挿入できます。データベースサーバは、インデックス値の論理順序を維持するために、新しいインデックスページを作成し、移動された既存の値を調整する必要がある場合があります。この新しいページは、通常、その値が最初に格納されていたページとは隣接していません。このようなインデックスページの順序の崩れが累積することを、インデックスの断片化と呼びます。

インデックスの断片化が起こると、大きなローのブロックが継続的に挿入、更新、削除されるテーブルに対してよく実行されるクエリの実行時間が長くなるという現象が発生します。

権限

このチュートリアルにはユーザ本人が作成したテストデータベース、*app_profiling.db* が使用されるので、このチュートリアルの実行に権限は必要がありません。デフォルトのユーザ、DBA として接続します。

レッスン 1：アプリケーションプロファイリングウィザードを使用した、インデックスの断片化の検査と修正

アプリケーションプロファイリングウィザードを使用して、インデックスの断片化を識別および修正します。運用データベースに断片化がないか定期的にチェックしてください。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：インデックスの断片化の診断」

このチュートリアルは、テストデータベース *app_profiling.db* が作成されていることを前提としています。「チュートリアル：アプリケーションプロファイリングのチュートリアル用のテストデータベースの作成」255 ページを参照してください。

♦ タスク

1. Sybase Central を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Sybase Central]** をクリックします。

2. Sybase Central で、次のように *app_profiling.db* に接続します。
   a. **[接続]** » **[SQL Anywhere 16 に接続]** をクリックします。
   b. **[接続]** ウィンドウで、テストデータベース *app_profiling.db* に接続するための情報を以下のフィールドに入力します。
      i. **[ユーザ ID]** フィールドに、**DBA** と入力します。
      ii. **[パスワード]** フィールドに、**sql** と入力します。
      iii. **[アクション]** ドロップダウンリストで、**[このコンピュータのデータベースを起動して接続]** をクリックします。
      iv. **[データベースファイル]** フィールドに、**C:¥AppProfilingTutorial¥app_profiling.db** と入力します。
      v. **[開始行]** フィールドに、**dbeng16 -x tcpip** と入力します。
   c. **[接続]** をクリックします。

3. **[モード]** » **[アプリケーションプロファイリング]** をクリックします。

   アプリケーションプロファイリングウィザードが表示されない場合は、**[アプリケーションプロファイリング]** » **[アプリケーションプロファイリングウィザードを開く]** をクリックします。

4. 次のようにアプリケーションプロファイリングウィザードを起動します。
   a. **[ようこそ]** ページで、**[次へ]** をクリックします。
   b. **[プロファイリングオプション]** ページで、**[データベーススキーマに基づく全体的なデータベースパフォーマンス]** をクリックし、**[次へ]** をクリックします。
   c. **[分析ファイル]** ページの **[次のファイルに分析を保存]** フィールドに、*C:¥AppProfilingTutorial¥analysis* と入力します。
   d. **[完了]** をクリックします。

   [アプリケーションプロファイリングの詳細] ウィンドウ枠に推奨リストが表示されます。

5. **[断片化されたインデックス]** が表示されていたら、それをダブルクリックします。**[推奨]** ウィンドウが開き、SQL 文が表示されます。この SQL 文を実行するとインデックスの断片化を解決できます。

6. Sybase Central を閉じます。

結果

アプリケーションプロファイリングウィザードを使用して、インデックスの断片化が識別され、修正されました。

参照

- 「インデックスの再構築」33 ページ
- 「インデックスの断片化とスキューの削減」245 ページ
- 「アプリケーションプロファイリング」148 ページ
- 「診断トレーシング」162 ページ

レッスン 2 : Interactive SQL 文を使用したインデックスの断片化の識別と修正

Interactive SQL を使用して、インデックスの断片化を識別および修正します。運用データベースに断片化がないか定期的にチェックしてください。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル : インデックスの断片化の診断」

このチュートリアルは、テストデータベース *app_profiling.db* が作成されていることを前提としています。「チュートリアル : アプリケーションプロファイリングのチュートリアル用のテストデータベースの作成」255 ページを参照してください。

ほかに *app_profiling.db* への接続がないことを確認します。

♦ タスク

1. Interactive SQL を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Interactive SQL]** をクリックします。

2. **[接続]** ウィンドウで、テストデータベース *app_profiling.db* に接続するための情報を以下のフィールドに入力します。
   a. **[ユーザ ID]** フィールドに、**DBA** と入力します。
   b. **[パスワード]** フィールドに、**sql** と入力します。
   c. **[アクション]** ドロップダウンリストで、**[このコンピュータのデータベースを起動して接続]** をクリックします。
   d. **[データベースファイル]** フィールドに、**C:¥AppProfilingTutorial¥app_profiling.db** と入力します。
   e. **[開始行]** フィールドに、**dbeng16 -x tcpip** と入力します。

3. **[接続]** をクリックします。

4. Interactive SQL で、次の SQL 文を実行して、Employees テーブルのインデックス密度をテストします。

```
CALL sa_index_density( 'Employees' );
```

密度の値は 0 から 1 の範囲の値です。1 に近い値は、インデックスの断片化がほとんどないことを示しています。0.5 未満の値は、インデックスの断片化のレベルがパフォーマンスに影響を与える可能性があることを示しています。

> **注意**
> 結果の Density カラムの値が 0.5 以下であるため、断片化の問題を示すために Employees のインデックスの値が表示されます。ただし、これらの数値が必要以上に低いのは、テーブルが非常に小さいためです。

5. Interactive SQL で、次の ALTER INDEX...REBUILD 文を実行して、インデックスの密度を向上させます。

```
ALTER INDEX PRIMARY KEY ON Employees REBUILD;
```

6. Interactive SQL を閉じます。

結果

Interactive SQL を使用して、インデックスの断片化が識別され、修正されました。

参照

- 「ALTER INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_index_density システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「REORGANIZE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「インデックスの再構築」33 ページ
- 「インデックスの断片化とスキューの削減」245 ページ
- 「アプリケーションプロファイリング」148 ページ
- 「カスタマイズされた診断トレーシングレベル」167 ページ
- 「診断トレーシング」162 ページ

チュートリアル：テーブルの断片化の診断

このチュートリアルのレッスンでは、**アプリケーションプロファイリングウィザード**または Interactive SQL を使用して、データベースにテーブルの断片化が発生しているかどうかを判別し、必要に応じてテーブルの断片化を修正する方法を説明します。

テーブルデータはデータベースページに格納されています。INSERT、UPDATE、DELETE などのデータ操作をテーブルに対して実行すると、ローが連続して格納されなかったり、ローが複数のページに渡って分割される場合があります。CPU のアクティビティが高い場合でも、テーブルの断片化は、テーブルのスキャンを必要とするクエリのパフォーマンスに悪影響を与える可能性があります。

権限

このチュートリアルにはユーザ本人が作成したテストデータベース、*app_profiling.db* が使用されるので、このチュートリアルの実行に権限は必要がありません。デフォルトのユーザ、DBA として接続します。

レッスン 1：アプリケーションプロファイリングウィザードを使用したテーブルの断片化の検査と修正

アプリケーションプロファイリングウィザードを使用して、テーブルの断片化を識別および修正します。運用データベースに断片化がないか定期的にチェックしてください。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：テーブルの断片化の診断」

このチュートリアルは、テストデータベース *app_profiling.db* が作成されていることを前提としています。「チュートリアル：アプリケーションプロファイリングのチュートリアル用のテストデータベースの作成」255 ページを参照してください。

♦ タスク

1. Sybase Central を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Sybase Central]** をクリックします。

2. Sybase Central で、次のように *app_profiling.db* に接続します。
   a. **[接続]** » **[SQL Anywhere 16 に接続]** をクリックします。
   b. **[接続]** ウィンドウで、テストデータベース *app_profiling.db* に接続するための情報を以下のフィールドに入力します。
      i. **[ユーザ ID]** フィールドに、**DBA** と入力します。
      ii. **[パスワード]** フィールドに、**sql** と入力します。
      iii. **[アクション]** ドロップダウンリストで、**[このコンピュータのデータベースを起動して接続]** をクリックします。
      iv. **[データベースファイル]** フィールドに、**C:¥AppProfilingTutorial¥app_profiling.db** と入力します。
      v. **[開始行]** フィールドに、**dbeng16 -x tcpip** と入力します。
   c. **[接続]** をクリックします。

3. Sybase Central で、**[モード]** » **[アプリケーションプロファイリング]** をクリックします。

   アプリケーションプロファイリングウィザードが表示されない場合は、**[アプリケーションプロファイリング]** » **[アプリケーションプロファイリングウィザードを開く]** をクリックします。

4. 次のように**アプリケーションプロファイリングウィザード**を起動します。
   a. **[プロファイリングオプション]** ページで、**[データベーススキーマに基づく全体的なデータベースパフォーマンス]** をクリックします。
   b. **[分析ファイル]** ページの **[次のファイルに分析を保存]** フィールドに、*C:¥AppProfilingTutorial¥analysis* と入力します。ファイルがすでに存在しているため、置き換えるかどうかを確認するプロンプトが表示された場合は、**[はい]** をクリックします。
   c. **[完了]** をクリックします。

   [アプリケーションプロファイリングの詳細] ウィンドウ枠に推奨リストが表示されます。
5. **[断片化されたテーブル]** が表示されていたら、それをダブルクリックします。**[推奨]** ウィンドウが開き、SQL 文が表示されます。この SQL 文を実行するとテーブルの断片化を解決できます。
6. Sybase Central を閉じます。

結果

アプリケーションプロファイリングを使用して、テーブルの断片化が識別され、修正されました。

参照

- 「テーブルの断片化削減」241 ページ
- 「アプリケーションプロファイリング」148 ページ
- 「診断トレーシング」162 ページ

レッスン 2 : Interactive SQL 文を使用したテーブルの断片化の識別と修正

Interactive SQL を使用して、テーブルの断片化を識別および修正します。運用データベースに断片化がないか定期的にチェックしてください。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：テーブルの断片化の診断」

このチュートリアルは、テストデータベース *app_profiling.db* が作成されていることを前提としています。「チュートリアル：アプリケーションプロファイリングのチュートリアル用のテストデータベースの作成」255 ページを参照してください。

ほかに *app_profiling.db* への接続がないことを確認します。

♦ タスク

1. Interactive SQL を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Interactive SQL]** をクリックします。

2. **[接続]** ウィンドウで、テストデータベース *app_profiling.db* に接続するための情報を以下のフィールドに入力します。
   a. **[ユーザ ID]** フィールドに、**DBA** と入力します。
   b. **[パスワード]** フィールドに、**sql** と入力します。
   c. **[アクション]** ドロップダウンリストで、**[このコンピュータのデータベースを起動して接続]** をクリックします。
   d. **[データベースファイル]** フィールドに、**C:¥AppProfilingTutorial¥app_profiling.db** と入力します。
   e. **[開始行]** フィールドに、**dbeng16 -x tcpip** と入力します。
3. **[接続]** をクリックします。
4. Interactive SQL で、次の SQL 文を実行して、Employees テーブルの断片化をテストします。

   ```
   CALL sa_table_fragmentation( 'Employees' );
   ```

   segs_per_row (ローごとのセグメント数) カラムの値が 1.1 よりも多い場合は、テーブルの断片化が存在します。断片化の程度が高いほど、パフォーマンスに悪影響を与える可能性があります。
5. Interactive SQL で、次の REORGANIZE TABLE 文を実行して、テーブルの断片化を減少させます。

   ```
   REORGANIZE TABLE Employees;
   ```

6. Interactive SQL を閉じます。

結果

Interactive SQL を使用してテーブルの断片化が診断されました。

参照

- 「sa_table_fragmentation システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「REORGANIZE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「テーブルの断片化削減」241 ページ
- 「アプリケーションプロファイリング」148 ページ
- 「カスタマイズされた診断トレーシングレベル」167 ページ
- 「診断トレーシング」162 ページ

チュートリアル：プロシージャプロファイリングをベースラインとして使用

アプリケーションプロファイリングウィザードを使用して、パフォーマンスを向上する際の比較の目的で使用できるベースラインを作成します。

プロシージャプロファイリングでは、プロシージャ、ユーザ定義関数、イベント、システム、システムトリガ、トリガの実行時間の測定値が収集されます。保存された結果をベースラインとし

て設定し、プロシージャを段階的に変更して、変更後にプロシージャを実行できます。新しい結果をベースラインと比較して、変更が目的どおり機能しているかどうかを確認できます。

権限

このチュートリアルにはユーザ本人が作成したテストデータベース、*app_profiling.db* が使用されるので、このチュートリアルの実行に権限は必要がありません。デフォルトのユーザ、DBA として接続します。

参照

- 「アプリケーションプロファイリング」148 ページ
- 「プロシージャプロファイリングの結果」154 ページ
- 「アプリケーションプロファイリングモードでのプロシージャプロファイリング」150 ページ
- 「カスタマイズされた診断トレーシングレベル」167 ページ
- 「診断トレーシング」162 ページ

レッスン 1：baseline プロシージャの作成

プロシージャプロファイリングの目的で baseline プロシージャを作成します。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：プロシージャプロファイリングをベースラインとして使用」

このチュートリアルは、テストデータベース *app_profiling.db* が作成されていることを前提としています。「チュートリアル：アプリケーションプロファイリングのチュートリアル用のテストデータベースの作成」255 ページを参照してください。

♦ タスク

1. Interactive SQL を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Interactive SQL]** をクリックします。

2. **[接続]** ウィンドウで、テストデータベース *app_profiling.db* に接続するための情報を以下のフィールドに入力します。
   a. **[認証]** フィールドで **[データベース]** を選択します。
   b. **[ユーザ ID]** フィールドに、**DBA** と入力します。
   c. **[パスワード]** フィールドに、**sql** と入力します。
   d. **[アクション]** ドロップダウンリストで、**[このコンピュータのデータベースを起動して接続]** をクリックします。
   e. **[データベースファイル]** フィールドに、**C:¥AppProfilingTutorial¥app_profiling.db** と入力します。
   f. **[開始行]** フィールドに、**dbsrv16 -x tcpip** と入力します。

3. **[接続]** をクリックします。

4. Interactive SQL で、次の SQL 文を実行します。

   a. テーブルを作成します。

```
CREATE TABLE table1 (
Count INT );
```

   b. ストアドプロシージャを作成します。

```
CREATE PROCEDURE baseline( )
  BEGIN
    INSERT table1
      SELECT COUNT (*)
        FROM rowgenerator r1, rowgenerator r2,
            rowgenerator r3
      WHERE r3.row_num < 5;
  END;
```

   c. データベースに加えた変更をコミットします。

```
COMMIT;
```

5. Interactive SQL を閉じます。

結果

比較の目的で使用する baseline プロシージャが作成されました。

次の手順

「レッスン 2：baseline プロシージャに対する更新されたプロシージャの実行」276 ページに進みます。

レッスン 2：baseline プロシージャに対する更新されたプロシージャの実行

プロシージャを更新して baseline プロシージャに対して実行します。プロシージャプロファイリングのための 2 つのプロシージャに関する診断トレーシングデータを収集します。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：プロシージャプロファイリングをベースラインとして使用」

このレッスンは、受講者がこれまでのすべてのレッスンを終了していることを前提としています。「レッスン 1：baseline プロシージャの作成」275 ページを参照してください。

♦ タスク

1. Sybase Central を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Sybase Central]** をクリックします。

2. Sybase Central で、次のように *app_profiling.db* に接続します。
   a. **[接続]** » **[SQL Anywhere 16 に接続]** をクリックします。
   b. **[接続]** ウィンドウで、テストデータベース *app_profiling.db* に接続するための情報を以下のフィールドに入力します。
      i. **[認証]** フィールドで **[データベース]** を選択します。
      ii. **[ユーザ ID]** フィールドに、**DBA** と入力します。
      iii. **[パスワード]** フィールドに、**sql** と入力します。
      iv. **[アクション]** ドロップダウンリストで、**[このコンピュータのデータベースを起動して接続]** をクリックします。
      v. **[データベースファイル]** フィールドに、**C:¥AppProfilingTutorial¥app_profiling.db** と入力します。
      vi. **[開始行]** フィールドに、**dbsrv16 -x tcpip** と入力します。
   c. **[接続]** をクリックします。

3. Sybase Central で、**[モード]** » **[アプリケーションプロファイリング]** をクリックします。

   アプリケーションプロファイリングウィザードが表示されない場合は、**[アプリケーションプロファイリング]** » **[アプリケーションプロファイリングウィザードを開く]** をクリックします。

4. アプリケーションプロファイリングウィザードを起動します。
   a. **[ようこそ]** ページで、**[次へ]** をクリックします。
   b. **[プロファイリングオプション]** ページで **[ストアドプロシージャ、ファンクション、トリガ、またはイベントの実行時間]** をクリックします。
   c. **[完了]** をクリックします。

   データベースサーバでプロシージャプロファイリングが開始します。

5. Sybase Central の左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックします。

6. baseline プロシージャを右クリックし、**[Interactive SQL から実行]** をクリックします。プロシージャプロファイリングが有効になっているため、プロシージャの実行の詳細が取得されます。

7. Interactive SQL を閉じます。

8. プロファイリング結果を表示します。
   a. Sybase Central の左ウィンドウ枠で、baseline プロシージャを選択します。
   b. 右ウィンドウ枠で、**[プロファイリング結果]** タブをクリックします。結果が表示されない場合は、**[表示]** » **[フォルダの再表示]** をクリックします。

      baseline プロシージャの行ごとの実行時間が表示されます。

9. プロファイリング結果を保存します。
   a. データベースを右クリックし、**[プロパティ]** をクリックします。
   b. **[プロファイリング設定]** タブをクリックします。
   c. **[データベース内に現在あるプロファイリング情報を次のプロファイリングログファイルに保存する]** を選択し、プロファイリングログファイルのロケーションとファイル名を指定します(*C:¥AppProfilingTutorial¥baseline* など)。
   d. **[適用]** をクリックします。プロパティウィンドウは閉じないでください。

      収集したプロシージャのプロファイリング情報は、指定したプロファイリングログファイル (*.plg*) に保存されます。

10. プロファイリングログファイルをベースラインとして使用することを指定します。
    a. **[App_Profiling - DBA データベースのプロパティ]** ウィンドウの **[プロファイリング設定]** タブで、**[次のプロファイリングログファイルのプロファイリング情報を比較のベースラインとして使用する]** を選択します。
    b. 作成したプロファイリングログファイルを探して選択します。
    c. **[適用]** をクリックします。
    d. **[OK]** をクリックし、**[App_Profiling - DBA データベースのプロパティ]** ウィンドウを閉じます。

11. baseline プロシージャに変更を加えます。
    a. Sybase Central で、**[モード]** » **[設計]** をクリックします。
    b. 左ウィンドウ枠の **[プロシージャとファンクション]** で、ベースラインプロシージャを参照して選択します。
    c. 右ウィンドウ枠の **[SQL]** タブで、既存の INSERT 文を次の INSERT 文に置き換えます。

       ```
       INSERT table1
         SELECT COUNT ( * ) FROM rowgenerator r1, rowgenerator r2, rowgenerator r3
         WHERE r3.row_num < 250;
       ```

    d. **[ファイル]** » **[保存]** をクリックします。

12. **[プロシージャとファンクション]** で、baseline プロシージャを右クリックし、**[Interactive SQL から実行]** をクリックします。

13. プロシージャが完了したら、Interactive SQL を終了します。

結果

プロシージャが更新されて baseline プロシージャに対して実行され、トレーシングセッション中の 2 つのプロシージャに関するデータが収集されました。

次の手順

「レッスン 3：プロシージャプロファイリングの結果の比較」279 ページに進みます。

レッスン 3：プロシージャプロファイリングの結果の比較

プロシージャプロファイリングの結果を比較して、更新されたプロシージャの実行時間が短くなったかどうかを判断します。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：プロシージャプロファイリングをベースラインとして使用」

このレッスンは、受講者がこれまでのすべてのレッスンを終了していることを前提としています。「レッスン 1：baseline プロシージャの作成」275 ページを参照してください。

♦ タスク

1. Sybase Central で、**[モード]** » **[アプリケーションプロファイリング]** をクリックします。

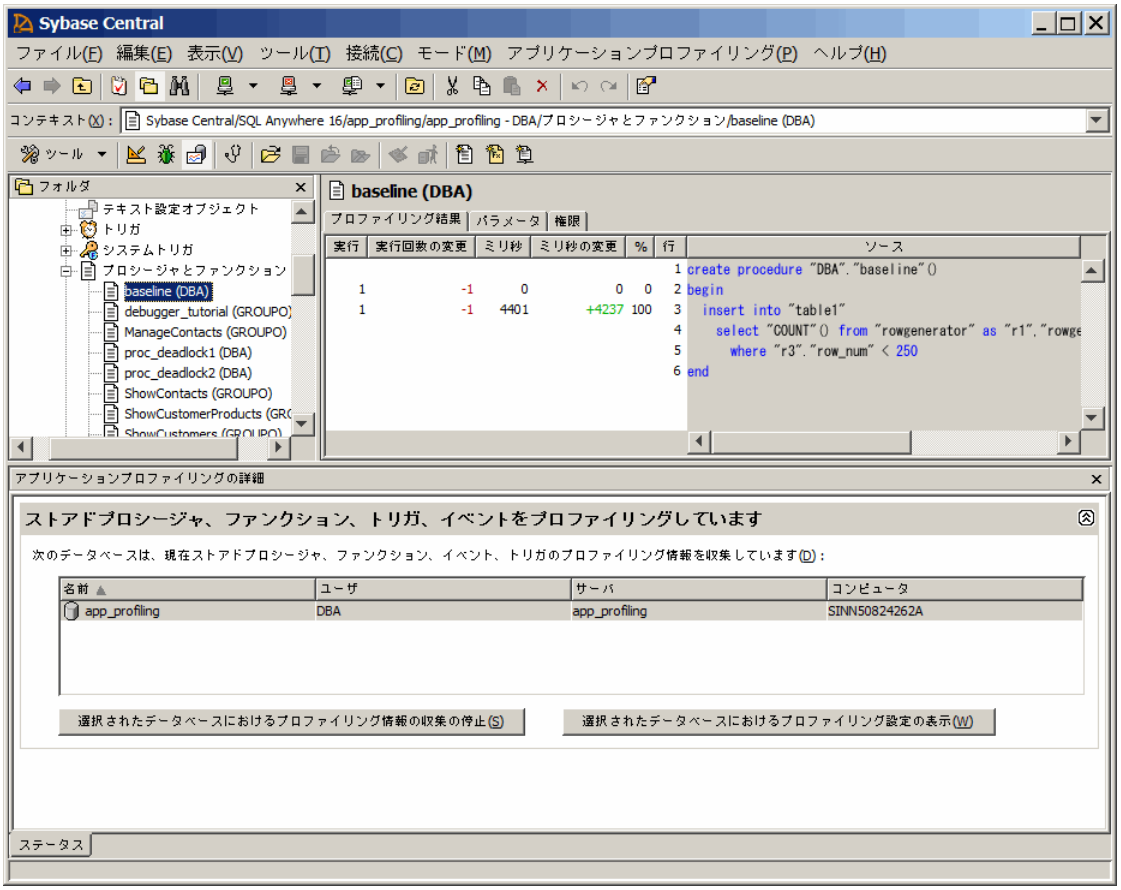
   アプリケーションプロファイリングウィザードが表示された場合は、**[キャンセル]** をクリックします。

2. Sybase Central の左ウィンドウ枠の **[プロシージャとファンクション]** で、baseline プロシージャをクリックします。

3. 右ウィンドウ枠で、**[プロファイリング結果]** タブをクリックします。

4. **[表示]** » **[フォルダの再表示]** をクリックします。

   2 つの新しいカラム、**[実行回数の変更]** と **[ミリ秒の変更]** が表示されます。



[実行回数の変更] と **[ミリ秒の変更]** カラムには、プロファイリングログファイル内の統計値と、プロシージャを最後に実行したときに取得された統計値の比較結果が表示されます。具体的には、プロシージャ内のコード行ごとに、それぞれの実行回数と実行時間を比較します。

[ミリ秒の変更] カラムは、プロシージャ内のコード行の実行時間が短くなったかどうかを示します。時間が短くなった場合は、マイナス記号が表示され、文字の色が赤になります。時間が長くなった場合は、プラス記号で示され、文字の色が緑になります。

このチュートリアルでは、**[ミリ秒の変更]** カラムの値は、プラス記号の付いた実行時間が緑の文字で表示されます。つまり、更新されたプロシージャの INSERT 文の方が、baseline プロシージャの INSERT 文よりも実行時間が長いということです。

結果

プロシージャプロファイリングをベースラインとして使用する方法のチュートリアルを終了しました。

クエリとデータ修正

この項では、データの問い合わせ方法と変更方法、ジョインの使用方法について説明します。クエリについて、数章に分けて簡単な内容から複雑な内容へと説明します。また、データの挿入、削除、更新についても説明します。また、この章では、多次元の結果を返す分析クエリの作成方法の詳細についても説明します。

クエリ

クエリはデータベースにデータを要求し、結果を受け取ります。この処理はデータ検索とも呼ばれます。SQL クエリはすべて、SELECT 文を使用して表現されます。SELECT 文は、1 つ以上のテーブルですべてのローまたはそのサブセットを検索するとき、および 1 つ以上のテーブルですべてのカラムまたはそのサブセットを検索するときに使用します。

クエリの最適化、選択性推定、コスト推定などの SQL Anywhere クエリ処理の詳細については、ホワイトペーパー『**Query Processing Based on SQL Anywhere 12.0.1 Architecture**』(http://www.sybase.com/detail?id=1096047) を参照してください。

SELECT 文と問い合わせ

SELECT 文は、クライアントアプリケーションで使用する情報をデータベースから取り出します。SELECT 文は「クエリ」とも呼ばれます。情報は、結果セットとしてクライアントアプリケーションに配信されます。クライアントは、配信された結果セットを処理できます。たとえば、Interactive SQL では、結果セットが [結果] ウィンドウ枠に表示されます。結果セットは、データベースのテーブルと同様に一連のローで構成されます。

SELECT 文には、返される結果のスコープを定義する「句」が含まれます。次の SELECT 構文では、各行が別々の句です。ここではごく一般的な句を示します。

SELECT *select-list*
[**FROM** *table-expression*]
[**WHERE** *search-condition*]
[**GROUP BY** *column-name*]
[**HAVING** *search-condition*]
[**ORDER BY** { *expression* | *integer* }]

次に、SELECT 文の各句について説明します。

- SELECT 句は、どのカラムを取得するかを指定します。この句は、SELECT 文で必須の句です。

- FROM 句は、どのテーブルからカラムを検索するかを指定します。この句は、テーブルからデータを取り出すすべてのクエリに必要です。FROM 句を持たない SELECT 文には別の意味がありますが、それについてはこの項では説明しません。

  ほとんどのクエリはテーブルを操作しますが、クエリを使用して、ビュー、他のクエリ (派生テーブル)、ストアドプロシージャの結果セットなど、カラムとローを持つ他のオブジェクトからデータを取り出せます。

●WHERE 句は、参照するテーブルのローを指定します。

●GROUP BY 句を使用するとデータを集約できます。

●HAVING 句は、集合データが収集されるローを指定します。

●ORDER BY 句は、結果セット内のローをソートします。(デフォルトでは、リレーショナルデータベースから返されるローの順序に意味はありません)。

これらの句の大半は省略可能ですが、SELECT 文に記述するときは、正しい順序で記述してください。

参照

- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「クエリ結果の要約、グループ化、ソート」468 ページ
- 「FROM 句」『SQL Anywhere サーバ SQL リファレンス』

クエリの述部

「述部」は条件式であり、論理演算子 AND や OR と組み合わせて、WHERE 句、HAVING 句、または ON 句に条件のセットを構成します。SQL では、UNKNOWN と評価される述部が FALSE として解釈されます。

インデックスを使用してテーブルからローを取り出すことができる述部を、「検索引数可能(sargable)」 であるといいます。この名前は、**search argument-able** というフレーズからとったものです。定数、他のカラム、または式とカラムとの比較を伴う述部は、検索引数可能です。

次に示す文の述部は、検索引数可能です。SQL Anywhere は、Employees テーブルのプライマリインデックスを使用して、この文を効率的に評価できます。

```
SELECT *
FROM Employees
WHERE Employees.EmployeeID = 102;
```

最適なアクセスプランでは、これは次のように表示されます。 Employees<Employees>.

反対に、次に示す述部は、検索引数可能ではありません。EmployeeID カラムはプライマリインデックスでインデックスが付けられていますが、結果にはすべてのローまたは 1 つを除くすべてのローが格納されるため、このインデックスを使用しても計算速度は上がりません。

```
SELECT *
FROM Employees
where Employees.EmployeeID <> 102;
```

最適なアクセスプランでは、これは次のように表示されます。 Employees<seq>.

同様に、名前が k という文字で終わるすべての従業員を検索する場合、インデックスは役に立ちません。この結果を計算するには、それぞれのローを個別に調べるしかありません。

関数

通常、カラム名に対する関数を持つ述部は、検索引数可能ではありません。たとえば、次に示すクエリにはインデックスは使用されません。

```
SELECT *
FROM SalesOrders
WHERE YEAR ( OrderDate ) ='2000';
```

クエリを書き換えて検索引数可能にすると、関数を使用しなくても済みます。たとえば、上記のクエリを次のように書き換えることができます。

```
SELECT *
FROM SalesOrders
WHERE OrderDate > '1999-12-31'
AND OrderDate < '2001-01-01';
```

関数値を計算カラムに格納し、このカラムのインデックスを構築すると、関数を使用するクエリが検索引数可能になります。「計算カラム」は、テーブル内の他のカラムから値を取得するカラムです。たとえば、注文の日付を保持するカラム OrderDate がある場合は、OrderDate カラムから抽出した年の値を保持する計算カラム OrderYear を作成できます。

```
ALTER TABLE SalesOrders
ADD OrderYear INTEGER
COMPUTE ( YEAR( OrderDate ) );
```

次に、カラム OrderYear のインデックスを通常どおり追加できます。

```
CREATE INDEX IDX_year
ON SalesOrders ( OrderYear );
```

次の文を実行すると、データベースサーバは、情報を保持するインデックスカラムがあることを認識し、そのインデックスを使用してクエリに応答します。

```
SELECT * FROM SalesOrders
WHERE YEAR( OrderDate ) = '2000';
```

計算カラムのドメインは、カラムが置換されるように、COMPUTE 式のドメインと同じにします。上記の例では、YEAR(OrderDate) が integer ではなく string を返した場合には、オプティマイザは式の計算カラムを置換しません。その結果、必要なローの取り出しにインデックス IDX_year を使用できなくなってしまいます。

例

次に示す各例では、属性 **x** と **y** は、単一テーブルのそれぞれのカラムです。属性 **z** は、別のテーブルに格納されています。このような属性のそれぞれにインデックスが 1 つ存在することが前提です。

検索引数可能	検索引数不可能
x = 10	**x** < > 10
x IS NULL	

検索引数可能	検索引数不可能
x IS NOT NULL	
x > 25	**x** = 4 OR **y** = 5
x = **z**	**x** = **y**
x IN (4, 5, 6)	**x** NOT IN (4, 5, 6)
x LIKE 'pat%'	**x** LIKE '%tern'
x = 20 - 2	**x** + 2 = 20
X IS NOT DISTINCT FROM *Y*+1	
X IS DISTINCT FROM *Y*+1	

述部が検索引数可能かどうか明らかでない場合があります。この場合、述部を、検索引数可能になるように書き換えることができます。各例では、u はアルファベット順で t の後にくるという事実を利用し、述部 **x** LIKE 'pat%' を **x** >= 'pat' and **x** < 'pau' に書き換えることが可能です。この形式では、属性 *x* のインデックスを使用して、一定範囲内の値を検索できます。幸い、SQL Anywhere では、この特殊な変形を自動で行います。

テーブルでのインデックス検索に使用される検索引数可能な述部は、「マッチング」述部です。WHERE 句には、多くのマッチング述部が含まれることがあります。最も適切な述部はアクセスプランによって異なります。オプティマイザは、アクセスプランの変更を検討するとき、マッチング述部の選択を再評価します。

参照

- 「計算カラム」13 ページ

SQL クエリ

このマニュアルの SELECT 文とその他の SQL 文の表記では、それぞれの句を個別の行で示し、SQL キーワードを大文字で示します。これは文を読みやすくするためであり、必須条件ではありません。SQL キーワードは大文字／小文字のいずれでも入力でき、文のどこでも改行できます。

キーワードと改行

たとえば、次の SELECT 文は、Contacts テーブルからカリフォルニア州内の連絡先の名前と姓を検索します。

```
SELECT GivenName, Surname
FROM Contacts
WHERE State = 'CA';
```

読みやすくはありませんが、この文を次のように入力しても有効です。

```
SELECT GivenName,
Surname from Contacts
WHERE State
 = 'CA';
```

文字列と識別子の大文字と小文字の区別

SQL Anywhere データベースでは、テーブル名、カラム名などの識別子は、大文字と小文字を区別しません。

文字列はデフォルトでは大文字と小文字が区別されないので、'CA'、'ca'、'cA'、'Ca' は同じです。ただし、大文字と小文字を区別するデータベースを作成する場合は、文字列中の大文字／小文字が重要になります。SQL Anywhere サンプルデータベースでは大文字と小文字を区別しません。

識別子の修飾

どのオブジェクトのことを指しているのかはっきりしない場合には、データベース識別子の名前を修飾します。たとえば、SQL Anywhere サンプルデータベースにはカラム City を含んだテーブルがいくつかあるので、City への参照をテーブル名で修飾する必要があります。より規模の大きいデータベースでは、テーブルの所有者の名前を使用してテーブルを識別する場合もあります。

```
SELECT Contacts.City
FROM Contacts
WHERE State = 'CA';
```

この項の例で示すのは単一テーブルのクエリですから、構文のモデルや例に出てくるカラム名を、カラムが属しているテーブルや所有者の名前で修飾することは通常はありません。

これらの要素は読みやすさを考慮して省略してあるものなので、修飾しても決して間違いではありません。

結果セットのローの順序

結果セットのローの順序に意味はありません。データベースからローが返される順序に保証はなく、またその順序に意味はありません。ローを特定の順序で取り出す場合は、クエリで順序を指定する必要があります。

参照

- 「データベースの作成」『SQL Anywhere サーバ データベース管理』
- 「初期化ユーティリティ (dbinit)」『SQL Anywhere サーバ データベース管理』
- 「大文字と小文字の区別」651 ページ

SELECT リスト：カラムの指定

SELECT リストは、データを問い合わせる 1 つ以上のオブジェクトから構成されます。SELECT リストは、通常はカンマで区切った一連のカラム名か、またはすべてのカラムを表すアスタリスク演算子で構成されています。より一般的には、SELECT リストには、1 つ以上の式がカンマで区切られた形で記述されます。リストの最終カラムの後や、リストにカラムが 1 つしかない場合、カンマはありません。

SELECT リストの一般的な構文は次のようになります。

SELECT *expression* [, *expression*]...

リスト中に、有効な識別子としての規則を満たしていないテーブルやカラムを記述する場合は、それらの識別子を二重引用符で囲んでください。

SELECT リストの式に記述することができるのは、* (すべてのカラム)、カラム名のリスト、文字列、カラムの見出し、算術演算子のある式です。また、集合関数も使用できます。

参照

- 「クエリ結果の要約、グループ化、ソート」468 ページ
- 「式」『SQL Anywhere サーバ SQL リファレンス』

テーブルからのすべてのカラムの選択

SELECT 文の中のアスタリスク (*) には特殊な意味があります。アスタリスクは、FROM 句で指定されたすべてのテーブルにあるすべてのカラム名を表します。1 つのテーブルのカラムをすべて参照したいときにアスタリスクを使用すると、入力する時間が節約でき、入力ミスを防ぐことができます。

SELECT * を使用すると、テーブルが作成されたときに定義された順で、カラムが返されます。

1 つのテーブルのカラムをすべて選択するための構文は次のとおりです。

```
SELECT *
FROM table-expression;
```

SELECT * は、1 つのテーブルに現在登録されているカラムをすべて検索するので、カラムの追加、削除、名前の変更など、テーブル構造の変更によって、SELECT * の結果も自動的に変わります。カラムを個別にリストする方が、結果の正確さを確保できます。

例

次の文は Departments テーブルのすべてのカラムを検索します。WHERE 句はありません。そのため、この文はテーブルのすべてのローを検索します。

```
SELECT *
FROM Departments;
```

結果は次のようになります。

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576

DepartmentID	DepartmentName	DepartmentHeadID
...	...	...

SELECT キーワードの後ろにテーブルのカラム名をすべて並べても、まったく同じ結果になります。

```
SELECT DepartmentID, DepartmentName, DepartmentHeadID
FROM Departments;
```

次のクエリに示すように、カラム名と同様、* もテーブル名で修飾できます。

```
SELECT Departments.*
FROM Departments;
```

テーブルからの特定のカラムの選択

SELECT 文によって取り出されるカラムは、SELECT キーワードに続けてカラムをリストすることで制限できます。この SELECT 文の構文は、次のとおりです。

SELECT *column-name* [, *column-name*]...
FROM *table-name*

この構文では、*column-name* と *table-name* は、問い合わせるカラムとテーブルの名前に置き換えてください。

次に例を示します。

```
SELECT Surname, GivenName
FROM Employees;
```

射影と制限

「射影」とは、テーブル内のカラムのサブセットです。「制限」 (「選択」とも言う) は、いくつかの条件に基づいたテーブル内のローのサブセットとのことです。

たとえば、次の SELECT 文では、SQL Anywhere サンプルデータベースで価格が $15 より高い製品すべての名前と価格が検索されます。

```
SELECT Name, UnitPrice
FROM Products
WHERE UnitPrice > 15;
```

このクエリでは、射影 (SELECT Name, UnitPrice) と制限 (WHERE UnitPrice > 15) を使用しています。

カラムの順番の再調整

カラム名をリストする順番で、カラムが表示される順番が決まります。次の 2 つの例は、表示におけるカラム順の指定方法を示しています。2 つとも、Departments テーブルの 5 つのロー全部から部署名と ID を検出して表示します。ただし、順番が異なります。

```
SELECT DepartmentID, DepartmentName
FROM Departments;
```

DepartmentID	DepartmentName
100	R & D
200	Sales
300	Finance
400	Marketing
...	...

```
SELECT DepartmentName, DepartmentID
FROM Departments;
```

DepartmentName	DepartmentID
R & D	100
Sales	200
Finance	300
Marketing	400
...	...

ジョイン

ジョインは、各テーブルのカラムの値を比較して、2 つ以上のテーブル内のローをリンクします。たとえば、1 ダースを超える数が出荷されたすべての注文項目について、注文項目 ID 番号と製品名を次のように選択できます。

```
SELECT SalesOrderItems.ID, Products.Name
FROM Products JOIN SalesOrderItems
WHERE SalesOrderItems.Quantity > 12;
```

Products テーブルと SalesOrderItems テーブルは、この両テーブル間の外部キー関係に基づいてジョインされます。

参照

● 「ジョイン：複数テーブルからのデータ検索」488 ページ

クエリ結果にある名前が変更されたカラム

デフォルトでは、結果セットの各カラムの見出しは SELECT リストに指定されている式の名前です。式がカラム値の場合、カラムの見出しはカラム名になります。Embedded SQL の場合、DESCRIBE 文を使用して、カーソルによって返される各式の名前を指定できます。また、他のアプリケーションインタフェースでも、インタフェース固有のメカニズムを使用して、結果セットの各カラム名に対するクエリがサポートされています。sa_describe_query システムプロシー

ジャでは、任意の SQL クエリの結果セットのカラム名を指定する、インタフェースに依存しない方法が提供されます。

クエリの SELECT リスト内の式の名前を上書きするには、次のように「エイリアス」を使用します。

SELECT *column-name* [**AS**] *alias*

エイリアスを作成すると結果が読みやすくなります。たとえば、次のように、部署リストの DepartmentName を Department に変更できます。

```
SELECT DepartmentName AS Department,
   DepartmentID AS "Identifying Number"
FROM Departments;
```

Department	Identifying Number
R & D	100
Sales	200
Finance	300
Marketing	400
...	...

使用方法

注意
次の文字は、エイリアスでは使用できません。

●" (二重引用符)

●制御文字 (0X20 未満の文字)

●円記号

●角カッコ

●逆引用符

● **エイリアスでのスペースとキーワードの使用**　上の例では、DepartmentID の "Identifying Number" エイリアスには空白が含まれているため、二重引用符で囲まれています。キーワードまたは特殊文字をエイリアスとして使用する場合も、二重引用符を使用します。たとえば、次のクエリは引用符がないと無効です。

```
SELECT DepartmentName AS Department,
   DepartmentID AS "integer"
FROM Departments;
```

- **ネームスペースの遮断**　エイリアスは、定義されている SELECT ブロックの任意の場所に使用できます。エイリアスは他の SELECT リスト式にも使用できます。この場合は、追加のエイリアスが定義されます。循環的なエイリアスの参照は許可されません。式に指定されたエイリアスが SELECT ブロックのネームスペースにある変数名またはカラム名と同じ場合は、エイリアス定義によってそのカラムまたは変数が遮断されます。次に例を示します。

```
SELECT DepartmentID AS DepartmentName
FROM Departments
WHERE DepartmentName = 'Marketing'
```

この文を実行すると、「値 'Marketing' をデータ型を numeric に変換できません。」というエラーが返されます。これは、クエリの WHERE 句の等号述部が、文字列リテラル "Marketing" と整数カラム DepartmentID を比較しようとしたものの、データ型に互換性がないためです。

> **注意**
> カラム名を参照する場合、テーブル名によってカラム名を明示的に修飾できます。たとえば、Departments.DepartmentID として、エイリアスと名前が競合しないようにします。

- **Transact-SQL との互換性**　Adaptive Server Enterprise では、SELECT リスト項目のエイリアスを識別するために、SQL/2008 の AS キーワードと等号の使用の**両方**がサポートされています。

参照

- 「Transact-SQL と互換性のあるクエリ」656 ページ
- 「sa_describe_query システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「DESCRIBE 文 [ESQL]」『SQL Anywhere サーバ SQL リファレンス』

クエリ結果の文字列

ほとんどの SELECT 文は、FROM 句のテーブルのデータだけで構成された結果を生成します。ただし、文字列を一重引用符で囲み、それを SELECT リストの他の要素とカンマで区切ると、その文字列もクエリ結果に表示されます。文字列に引用符を記述するには、その前に引用符をもう 1 つ記述します。次に例を示します。

```
SELECT 'The department''s name is' AS "Prefix",
   DepartmentName AS Department
FROM Departments;
```

Prefix	Department
The department's name is	R & D
The department's name is	Sales
The department's name is	Finance
The department's name is	Marketing
The department's name is	Shipping

SELECT リストの計算値

SELECT リストの式は、カラム名や文字列だけではなく、より複雑にできます。たとえば、SELECT リストの数値カラムのデータを使用して計算を行うことができます。

算術演算

SELECT リストで実行できる数値演算を説明するには、まず SQL Anywhere サンプルデータベース内の製品の名前、在庫数、単価をリストすることから始めます。

```
SELECT Name, Quantity, UnitPrice
FROM Products;
```

Name	Quantity	UnitPrice
Tee Shirt	28	9
Tee Shirt	54	14
Tee Shirt	75	14
Baseball Cap	112	9
...	...	...

どの製品も在庫数が 10 になったときに在庫を補充するとします。次のクエリは、各製品をあといくつ売ったら再発注するかをリストします。

```
SELECT Name, Quantity - 10
  AS "Sell before reorder"
FROM Products;
```

Name	Sell before reorder
Tee Shirt	18
Tee Shirt	44
Tee Shirt	65
Baseball Cap	102
...	...

カラムの値を組み合わせることもできます。次のクエリは、在庫中の各製品の総額をリストします。

```
SELECT Name, Quantity * UnitPrice AS "Inventory value"
FROM Products;
```

Name	Inventory value
Tee Shirt	252.00
Tee Shirt	756.00
Tee Shirt	1050.00
Baseball Cap	1008.00
...	...

1 つの式に算術演算子が 2 つ以上ある場合は、乗法、除法、剰余をまず計算し、その後で減法と加法を計算します。1 つの式のすべての算術演算子の優先度が同じ場合、計算は左から右に順番に行われます。カッコに入っている式は、他のすべての計算より優先されます。

たとえば、次の SELECT 文は、在庫中の各製品の総額を計算し、次にその値から 5 ドル引きます。

```
SELECT Name, Quantity * UnitPrice - 5
FROM Products;
```

確実に正しい結果を得るためには、可能な限りカッコを使用します。次のクエリは前述のクエリと同じ意味で、同じ結果を生成しますが、構文の精度が高くなります。

```
SELECT Name, ( Quantity * UnitPrice ) - 5
FROM Products;
```

算術演算は、演算の結果をそのデータ型で表現できないことによってオーバーフローする場合があります。オーバーフローが発生すると、値の代わりにエラーが返されます。

文字列の演算

文字列連結演算子を使用して文字列を連結できます。「||」(SQL/2008 で定義されている) または「+」(Adaptive Server Enterprise でサポート) を連結演算子として使用します。たとえば、次の文では、結果の Surname と GivenName の値を取り出して、連結しています。

```
SELECT EmployeeID, GivenName || ' ' || Surname AS Name
FROM Employees;
```

EmployeeID	Name
102	Fran Whitney
105	Matthew Cobb
129	Philip Chin
148	Julie Jordan
...	...

日付と時刻の演算

日付カラムと時刻カラムでも演算子を使用できますが、そのためには一般的に関数を使用することになります。

計算カラムについての注意事項

- **カラムのエイリアスを指定できる**　デフォルトでは、カラム名は SELECT リストにリストされる式ですが、計算カラムの場合、式では煩雑でわかりにくくなります。

- **その他の演算子を使用できる**　乗算演算子はカラムを結合するために使用できます。標準的な算術演算子、論理演算子、文字列演算子など、その他の演算子も使用できます。

  たとえば、次のクエリは、すべての顧客のフルネームをリストします。

  ```
  SELECT ID, (GivenName || ' ' || Surname ) AS "Full name"
  FROM Customers;
  ```

  || 演算子は文字列を連結しています。このクエリの場合、カラムのエイリアスにはスペースが付いているので、二重引用符で囲みます。この規則は、カラムのエイリアスだけでなく、データベース内のテーブル名やその他の識別子にも適用されます。

- **関数を使用できる**　カラムを結合するだけでなく、さまざまな組み込み関数を使用して、必要な結果を得ることができます。

  たとえば、次のクエリは製品名を大文字でリストします。

  ```
  SELECT ID, UCASE( Name )
  FROM Products;
  ```

ID	UCASE(Products.name)
300	TEE SHIRT
301	TEE SHIRT
302	TEE SHIRT
400	BASEBALL CAP
...	...

参照

- 「演算子の優先度」『SQL Anywhere サーバ SQL リファレンス』
- 「 SQL 関数」『SQL Anywhere サーバ SQL リファレンス』
- 「演算子」『SQL Anywhere サーバ SQL リファレンス』
- 「クエリ結果にある名前が変更されたカラム」288 ページ

重複したクエリ結果の削除

オプションの DISTINCT キーワードは、SELECT 文の結果から重複するローを削除します。DISTINCT を指定しないと、重複ローを含むすべてのローが表示されます。オプションとして、

SELECT リストの前に ALL を指定すると、すべてのローが表示されます。SQL のその他の実装との互換性のために、SQL Anywhere 構文では、ALL を使用して明示的にすべてのローを要求できるようになっています。ALL がデフォルトです。

たとえば、DISTINCT を指定しないで Contacts テーブルのすべての都市を検索した場合は、60 個のローが表示されます。

```
SELECT City
FROM Contacts;
```

DISTINCT を使用すれば、重複するエントリを削除できます。次のクエリは 16 個のローだけ返します。

```
SELECT DISTINCT City
FROM Contacts;
```

NULL 値は互いに区別されない

DISTINCT キーワードは、複数の NULL 値を互いに重複するものとして処理します。つまり、SELECT 文に DISTINCT が記述されていると、どんなに多くの NULL 値が検出された場合でも、結果には 1 つの NULL しか返されません。

FROM 句：テーブルの指定

テーブル、ビュー、またはストアドプロシージャのデータに関係のある SELECT 文には、FROM 句が必須です。

FROM 句には、2 つ以上のテーブルをリンクする JOIN 条件を記述できるほか、他のクエリ (派生テーブル) へのジョインを記述できます。

テーブル名の修飾

FROM 句では、テーブルとビューについては常に、次のようなフルネームでの構文を作成できます。

```
SELECT select-list
FROM owner.table-name;
```

テーブル名、ビュー名、またはプロシージャ名を修飾する必要があるのは、そのオブジェクトが現在の接続のユーザ ID と異なるユーザ ID によって所有されている場合、または所有者のユーザ ID が現在の接続のユーザ ID が所属するロール名と異なる場合だけです。

相関名の使用

テーブル名に相関名を指定すると、読みやすさが向上し、テーブル名を参照する箇所で毎回完全な名前を入力する手間が省けます。相関名は、次のように、FROM 句でテーブル名の後に入力して割り当てます。

```
SELECT d.DepartmentID, d.DepartmentName
FROM Departments d;
```

相関名が使用される場合、たとえば WHERE 句の中など、そのテーブルに対する他のすべての参照には、テーブル名ではなく、**必ず**相関名を使用してください。相関名は有効な識別子としての規則に従っている必要があります。

派生テーブルのクエリ

派生テーブルは、クエリ式の評価によって、1 つまたは複数のテーブルから直接または間接的に派生されるテーブルです。派生テーブルは、SELECT 文の FROM 句に定義します。

派生テーブルのクエリは、ビューのクエリと同様に動作します。つまり、派生テーブルの値は、派生テーブル定義の評価時に決定されます。ただし、派生テーブルは、その定義がデータベースに保存されないという点で、ビューと異なります。また、派生テーブルは実体化されず、それが定義されているクエリ外からは参照できないという点で、ベーステーブルやテンポラリテーブルと異なります。

次のクエリでは、各部署の最高給与を取得するために、派生テーブル (my_derived_table) を使用しています。派生テーブルのデータは、Employees テーブルにジョインして、その給与の従業員の姓を取得できます。

```
SELECT Surname,
  my_derived_table.maximum_salary AS Salary,
  my_derived_table.DepartmentID
FROM Employees e,
 ( SELECT MAX( Salary ) AS maximum_salary, DepartmentID
    FROM Employees
    GROUP BY DepartmentID ) my_derived_table
 WHERE e.Salary = my_derived_table.maximum_salary
 AND e.DepartmentID = my_derived_table.DepartmentID
ORDER BY Salary DESC;
```

Surname	Salary	DepartmentID
Shea	138948.00	300
Scott	96300.00	100
Kelly	87500.00	200
Evans	68940.00	400
Martinez	55500.80	500

次の例では、Products テーブルの項目をランクする派生テーブル (MyDerivedTable) を作成し、派生テーブルを問い合わせて、最も低価格の 3 つの項目を返しています。

```
SELECT TOP 3 *
     FROM ( SELECT Description,
                   Quantity,
                   UnitPrice,
                   RANK() OVER ( ORDER BY UnitPrice ASC )
              AS Rank
              FROM Products ) AS MyDerivedTable
ORDER BY Rank;
```

テーブル以外のオブジェクトのクエリ

FROM 句内の最も一般的な要素は、テーブル名です。ただし、テーブルに似た構造を持つ、すなわち、適切に定義されたローとカラムを持つ他のデータベースオブジェクトからローを問い合わ

せることも可能です。たとえば、ビューを問い合わせることも、結果セットを返すストアドプロシージャを問い合わせることもできます。

たとえば、次の文は ShowCustomerProducts というストアドプロシージャの結果セットを問い合わせます。

```
SELECT *
FROM ShowCustomerProducts( 149 );
```

参照

- 「FROM 句」『SQL Anywhere サーバ SQL リファレンス』
- 「ジョイン：複数テーブルからのデータ検索」488 ページ

DML 文に対する SELECT

SQL Anywhere は、DML 文 (INSERT、UPDATE、DELETE、または MERGE) をクエリの FROM 句内のテーブル式として使用することをサポートします。

文に *dml-derived-table* を含めると、DESCRIBE では無視されます。OPEN 時には、UPDATE 文が最初に実行され、結果がテンポラリテーブルに格納されます。テンポラリテーブルでは、文によって変更されるテーブルのカラム名が使用されます。変更された値は、REFERENCING 句の相関名を使用して参照できます。OLD または FINAL を指定することによって、クエリで参照される更新されたテーブルに、一連のユニークなカラム名を指定する必要がなくなります。*dml-derived-table* 文は更新可能なテーブルを 1 つのみ参照できます。複数のテーブルに対する更新の場合は、エラーが返されます。

たとえば、次のクエリでは、UPDATE 文に対して SELECT を使用し、以下にリストする操作を実行します。

- サンプルデータベース内のすべての製品の値段を 7% 上げるように更新する
- 影響を受ける製品とその注文のうち、2000 年 4 月 10 日から 2000 年 5 月 21 日までの間に出荷され、注文数が 36 より多いものをリストする

```
SELECT old_products.ID, old_products.name, old_products.UnitPrice AS OldPrice,
      final_products.UnitPrice AS NewPrice, SOI.ID AS OrderID, SOI.Quantity
FROM
( UPDATE Products SET UnitPrice = UnitPrice * 1.07 )
    REFERENCING ( OLD AS old_products FINAL AS final_products )
  JOIN SalesOrderItems AS SOI ON SOI.ProductID = old_products.ID
WHERE SOI.ShipDate BETWEEN '2000-04-10' AND '2000-05-21'
     AND SOI.QUANTITY > 36
ORDER BY old_products.ID;
```

次のクエリは、MERGE 文と UPDATE 文の両方を使用しています。modified_employees テーブルは、ステータスが変更された従業員のコレクションを表しています。MERGE 文では、給与が 3% 上がった従業員と modified_employees テーブルに含まれている従業員の従業員 ID と名前をマージしています。このクエリで OPTION 句に指定されているオプション設定は、UPDATE 文と MERGE 文の両方に適用されます。

```
CREATE TABLE modified_employees
( EmployeeID INTEGER PRIMARY KEY, Surname VARCHAR(40), GivenName VARCHAR(40) );
```

```
MERGE INTO modified_employees AS me
USING (SELECT modified_employees.EmployeeID,
            modified_employees.Surname,
            modified_employees.GivenName
      FROM (
        UPDATE Employees
        SET Salary = Salary * 1.03
        WHERE ManagerID = 501)
          REFERENCING (FINAL AS modified_employees) ) AS dt_e
      ON dt_e.EmployeeID = me.EmployeeID
WHEN MATCHED THEN SKIP
WHEN NOT MATCHED THEN INSERT
OPTION( optimization_level=1, isolation_level=2 );
```

クエリでの複数テーブルの使用

クエリ内で複数の *dml-derived-table* 引数を使用する場合、UPDATE 文の実行順序は保証されません。次の文は、サンプルデータベースの Products テーブルと SalesOrderItems テーブルの両方を更新し、その操作を含むジョインに基づく結果を生成します。

```
SELECT old_products.ID, old_products.name, old_products.UnitPrice AS OldPrice,
      final_products.UnitPrice AS NewPrice,
      SalesOrders.ID AS OrderID, SalesOrders.CustomerID,
      old_order_items.Quantity,
      old_order_items.ShipDate AS OldShipDate,
      final_order_items.ShipDate AS RevisedShipDate
FROM
( UPDATE Products SET UnitPrice = UnitPrice * 1.07 )
     REFERENCING ( OLD AS old_products FINAL AS final_products )
  JOIN
( UPDATE SalesOrderItems
  SET ShipDate = DATEADD( DAY, 6, ShipDate )
  WHERE ShipDate BETWEEN  '2000-04-10' AND '2000-05-21' )
     REFERENCING ( OLD AS old_order_items FINAL AS final_order_items )
       ON (old_order_items.ProductID = old_products.ID)
  JOIN SalesOrders ON ( SalesOrders.ID = old_order_items.ID )
WHERE old_order_items.Quantity > 36
ORDER BY old_products.ID;
```

結果の実体化を伴わないテーブルの使用

REFERENCING (NONE) 句を使用すると、結果を実体化しないで UPDATE 文を埋め込むことができます。この場合は、UPDATE 文の結果が空となるため、意図した結果が返されるようにクエリを記述します。空ではない結果が返されるようにするには、外部ジョインの NULL 入力側に *dml-derived-table* を指定します。次に例を示します。

```
SELECT 'completed' AS finished, ( SELECT COUNT( * ) FROM Products ) AS product_total
FROM SYS.DUMMY LEFT OUTER JOIN
   ( UPDATE Products SET UnitPrice = UnitPrice * 1.07 )
     REFERENCING ( NONE ) ON 1=1;
```

また、集合演算子 (UNION、EXCEPT、または INTERSECT) のいずれかを使用するクエリ式の一部として *dml-derived-table* を使用することによって、空ではない結果が返されるようにすることもできます。次に例を示します。

```
SELECT 'completed' AS finished, ( SELECT COUNT( * ) FROM Products ) AS product_total
FROM SYS.DUMMY
UNION ALL
SELECT 'dummy', 1 /* This query specification returns the empty set */
```

```
FROM ( UPDATE Products SET UnitPrice = UnitPrice * 1.07 )
    REFERENCING ( NONE );
```

参照

- 「FROM 句」『SQL Anywhere サーバ SQL リファレンス』
- 「データ操作文」617 ページ

WHERE 句：ローの指定

SELECT 文の WHERE 句は、どのローを検索するのかを正確に決定する探索条件を指定します。探索条件は「述部」とも呼ばれます。一般的なフォーマットは次のとおりです。

SELECT *select-list*
FROM *table-list*
WHERE *search-condition*

WHERE 句の探索条件には、次のものがあります。

- **比較演算子**　(=、<、> など) たとえば、給料が $50,000 を上回る従業員全員をリストする場合は、次のようになります。

```
SELECT Surname
  FROM Employees
  WHERE Salary > 50000;
```

- **範囲**　(BETWEEN と NOT BETWEEN) たとえば、給料が $40,000 ～ $60,000 の従業員をすべてリストする場合は、次のようになります。

```
SELECT Surname
  FROM Employees
  WHERE Salary BETWEEN 40000 AND 60000;
```

- **リスト**　(IN、NOT IN) たとえば、オンタリオ州、ケベック州、またはマニトバ州の顧客をすべてリストする場合は、次のようになります。

```
SELECT CompanyName, State
  FROM Customers
  WHERE State IN( 'ON', 'PQ', 'MB');
```

- **文字の一致**　(LIKE と NOT LIKE) たとえば、電話番号が 415 ではじまる顧客をすべてリストする場合は、次のようになります (データベースでは、電話番号は文字列として保存されています)。

```
SELECT CompanyName, Phone
  FROM Customers
  WHERE Phone LIKE '415%';
```

- **不定の値**　(IS NULL と IS NOT NULL) たとえば、マネージャがいる部署をすべてリストする場合は、次のようになります。

```
SELECT DepartmentName
  FROM Departments
  WHERE DepartmentHeadID IS NOT NULL;
```

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

- **組み合わせ**　(AND、OR) たとえば、給料が $50,000 を上回り、名前が A で始まるすべての従業員をリストする場合は、次のようになります。

```
SELECT GivenName, Surname
  FROM Employees
  WHERE Salary > 50000
  AND GivenName like 'A%';
```

参照

- 「探索条件」『SQL Anywhere サーバ SQL リファレンス』

WHERE 句での比較演算子

WHERE 句で比較演算子が使用できます。演算子は次の構文に従います。

WHERE *expression comparison-operator expression*

比較についての注意事項

- **ソート順**　文字データの比較の場合、< はソート順の前の方、> はソート順の後の方であるという意味です。ソート順は、データベースを作成するときに選択した照合順によって決まります。データベースに対して dbinfo コマンドラインユーティリティを実行すると、照合順を確認できます。

  ```
  dbinfo -c "uid=DBA;pwd=sql"
  ```

  Sybase Central からも **[データベースのプロパティ]** の [詳細情報] タブに移動して、照合順を確認することができます。

- **後続ブランク**　データベースの作成時には、比較処理のために後続ブランクを無視するかどうかを示します。

  デフォルトでは、後続ブランクを無視しないでデータベースを作成します。たとえば、'Dirk' は 'Dirk ' と同じではありません。後続ブランクが無視されるように、ブランクを埋め込んだデータベースを作成できます。

- **日付の比較**　日付を比較するときには、< はより古いことを意味し、> はより新しいことを意味します。

- **大文字と小文字の区別**　データベースの作成時に、文字列比較が大文字と小文字を区別するかどうかを指定します。

  デフォルトでは、データベースは大文字と小文字を区別しないで作成されます。たとえば、'Dirk' は 'DIRK' と同じです。大文字と小文字を区別するように、データベースを作成できます。

次は比較演算子を使用する SELECT 文です。

```
SELECT *
  FROM Products
  WHERE Quantity < 20;
SELECT E.Surname, E.GivenName
```

```
  FROM Employees E
  WHERE Surname > 'McBadden';
SELECT ID, Phone
  FROM Contacts
  WHERE State  != 'CA';
```

NOT 演算子

NOT 演算子は式を否定します。次の 2 つのクエリはどちらも、単価が $10 以下の T シャツ (Tee Shirt) と野球帽 (BaseBall Cap) をすべて検索します。ただし、否定の論理演算子 (NOT) と否定の比較演算子 (!>) では、位置が異なることに注意してください。

```
SELECT ID, Name, Quantity
  FROM Products
  WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
  AND NOT UnitPrice > 10;
SELECT ID, Name, Quantity
  FROM Products
  WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
  AND UnitPrice !> 10;
```

参照

- 「比較演算子」『SQL Anywhere サーバ SQL リファレンス』
- 「式」『SQL Anywhere サーバ SQL リファレンス』

WHERE 句での範囲

BETWEEN キーワードは包括的範囲を指定します。包括的範囲には、その範囲の間の値だけではなく、上限値と下限値も含まれます。

NOT BETWEEN を使用すると、この範囲内にないローをすべて検出できます。

例

●次のクエリは、$10 以上 $15 以下の価格の製品をすべてリストします。

```
SELECT Name, UnitPrice
  FROM Products
  WHERE UnitPrice BETWEEN 10 AND 15;
```

Name	UnitPrice
Tee Shirt	14
Tee Shirt	14
Baseball Cap	10
Shorts	15

●次のクエリは、$10 未満か、$15 を超える製品をすべてリストします。

```
SELECT Name, UnitPrice
  FROM Products
  WHERE UnitPrice NOT BETWEEN 10 AND 15;
```

Name	UnitPrice
Tee Shirt	9
Baseball Cap	9
Visor	7
Visor	7
...	...

参照

- 「BETWEEN 探索条件」『SQL Anywhere サーバ SQL リファレンス』
- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』

WHERE 句でのリスト

IN キーワードは、値のリストのいずれかに一致する値を選択するためのキーワードです。式は定数またはカラム名であり、リストは、定数のセット、またはより一般的には、サブクエリです。

たとえば、IN を使用せずに、オンタリオ州、マニトバ州、またはケベック州内の顧客すべての名前と州名をリストする場合は、次のようなクエリを入力します。

```
SELECT CompanyName, State
  FROM Customers
  WHERE State = 'ON' OR State = 'MB' OR State = 'PQ';
```

ただし、得られる結果は IN を使用したときと同じになります。IN キーワードに続く項目は、カンマで区切り、カッコで囲んでください。文字、日付、時刻の値の前後に一重引用符を入力します。次に例を示します。

```
SELECT CompanyName, State
  FROM Customers
  WHERE State IN( 'ON', 'MB', 'PQ');
```

おそらく、IN キーワードの最も重要な用途は、サブクエリとも呼ばれる、ネストされたクエリの中で使用される場合です。

WHERE 句での文字列のパターン一致

パターン一致は、文字データを識別する便利な方法です。SQL では、パターンの検索には LIKE キーワードを使用します。パターン一致では、ワイルドカード文字を使用して、文字のさまざまな組み合わせに対応します。

LIKE キーワードは、その後に入力された文字列がマッチングパターンであると指定します。LIKE は文字データとともに使用します。

LIKE の構文は、次のとおりです。

expression [**NOT**] **LIKE** *match-expression*

一致する式は、次の特殊記号を含むマッチ式と比較されます。

記号	意味
%	文字数が 0 以上の文字列に一致します。
_	文字 1 個に一致します。
[specifier]	カッコ内の指定子の形式は、次のとおりです。 ● **範囲**　範囲のフォームは *rangespec1-rangespec2* です。*rangespec1* は文字の範囲の始点を指し、ハイフンは範囲、*rangespec2* は文字範囲の終点を指します。 ● **セット**　セットには、任意の順序での不連続な値の集合が含まれます。たとえば、[a2bR] などです。 範囲 [a-f] と、セット [abcdef] と [fcbdae] は、同じ値セットを返します。
[^specifier]	指定子の前にある脱字記号 (^) は非包含を表します。[^a-f] は a ～ f の範囲ではないとの意味で、[^a2bR] は a、2、b、R ではないとの意味です。

カラムデータを、定数、変数、またはテーブルに示されているワイルドカード文字を含むその他のカラムに一致させることができます。定数を使用するときは、マッチ文字列と文字列を一重引用符で囲みます。

例

次の例はすべて、Contacts テーブルの Surname カラムで LIKE を使用しています。クエリの形式は次のとおりです。

```
SELECT Surname
  FROM Contacts
  WHERE Surname LIKE match-expression;
```

最初の例は次のように入力します。

```
SELECT Surname
  FROM Contacts
  WHERE Surname LIKE 'Mc%';
```

マッチ式	説明	検索結果
'Mc%'	Mc で始まるすべての名前を検索します。	McEvoy
'%er'	er で終わるすべての名前を検索します。	Brier, Miller, Weaver, Rayner
'%en%'	en を含むすべての名前を検索します。	Pettengill, Lencki, Cohen
'_ish'	ish で終わるすべての 4 文字の名前を検索します。	Fish
'Br[iy][ae]r'	Brier、Bryer、Briar、または Bryar を検索します。	Brier
'[M-Z]owell'	M ～ Z の範囲の 1 文字で始まり、owell で終わるすべての名前を検索します。	Powell
'M[^c]%'	M で始まり、2 番目の文字が c ではないすべての名前を検索します。	Moore, Mulley, Miller, Masalsky

ワイルドカードには LIKE が必須

ワイルドカード文字を LIKE なしで使用すると、パターンとは解釈されず、「「文字列リテラル」」と解釈されます。つまり、ワイルドカード文字そのものの値を表すことになります。次のクエリは、415% の 4 文字だけから成る電話番号を検出しようとします。415 で始まる電話番号は検出しません。

```
SELECT Phone
  FROM Contacts
  WHERE Phone = '415%';
```

日付値と時刻値での LIKE の使用

LIKE は、DATE、TIME、TIMESTAMP、TIMESTAMP WITH TIME ZONE の各フィールドに対して使用できます。ただし、LIKE 述部は文字データでのみ使用できます。日付値や時刻値で LIKE を使用すると、これらの値は、DATE、TIME、TIMESTAMP、TIMESTAMP WITH TIME ZONE の各データ型の対応するオプション設定を使用して、暗黙的に CHAR または VARCHAR にキャストされ、値がフォーマットされます。

日付／時刻	VARCHAR への CAST で使用
DATE	date_format
TIME	time_format

日付／時刻	VARCHAR への CAST で使用
TIMESTAMP	timestamp_format
TIMESTAMP WITH TIME ZONE	timestamp_with_time_zone_format

日付値と時刻値にはさまざまな日付部分があり、上記のオプションに基づいてそれぞれ異なる方法でフォーマットされるため、DATE、TIME、または TIMESTAMP の値を検索するときに LIKE を使用する場合、検索に成功するには LIKE パターンを慎重に書き込む必要があります。

たとえば、TIMESTAMP カラム arrival_time に値 9:20 と現在の日付を入力した場合、timestamp_format オプションによってコロンを使用して時間と分を区切るように値の時刻部分がフォーマットされると、次の句は TRUE と評価されます。

```
WHERE arrival_time LIKE '%09:20%'
```

LIKE とは異なり、文字列リテラルと DATE、TIME、TIMESTAMP、または TIMESTAMP WITH TIME ZONE の各値の単純な比較を含む検索条件では、比較ドメインとして日付／時刻データ型が使用されます。この場合、SQL Anywhere では、まず文字列リテラルを TIMESTAMP 値に変換してから、その値の必要な部分を使用して比較を実行します。TIME、DATE、TIMESTAMP 値の変換について、SQL Anywhere は ISO 8601 標準に従っており、追加の拡張が行われています。

たとえば、定数文字列値 9:20 は 9:20 を時刻部分および日付部分の現在の日付として使用して TIMESTAMP に変換されるため、次の句は TRUE と評価されます。

```
WHERE arrival_time = '9:20'
```

NOT LIKE の使用

LIKE とともに使用できるのと同じワイルドカード文字を、NOT LIKE にも使用できます。次のクエリのいずれかを使用すると、Contacts テーブル内で市外局番が 415 ではない電話番号をすべて検索します。

```
SELECT Phone
  FROM Contacts
  WHERE Phone NOT LIKE '415%';

SELECT Phone
  FROM Contacts
  WHERE NOT Phone LIKE '415%';
```

アンダースコアの使用

LIKE とともに使用できるもう 1 つの特殊文字は _ (アンダースコア) 文字です。この特殊文字は、厳密に 1 つの文字と対応します。たとえば、パターン 'BR_U%' は、BR で始まり 4 番めの文字が U であるすべての名前と対応します。Braun では、_ は文字 A に対応し、% は N に対応します。

参照

- 「文字列リテラル」『SQL Anywhere サーバ SQL リファレンス』
- 「LIKE 探索条件」『SQL Anywhere サーバ SQL リファレンス』

文字列と引用符

文字データや日付データを入力したり、検索したりするときは、次の例のように一重引用符で囲んでください。

```
SELECT GivenName, Surname
  FROM Contacts
  WHERE GivenName = 'John';
```

quoted_identifier データベースオプションが Off に設定されている場合は (デフォルトでは On)、次の例のように二重引用符を使用して文字や日付のデータを囲むこともできます。

```
SET OPTION quoted_identifier = 'Off';
```

quoted_identifier オプションは、Adaptive Server Enterprise との互換性のために提供されています。デフォルトでは、Adaptive Server Enterprise オプションは quoted_identifier が Off で、SQL Anywhere オプションは quoted_identifier が On です。

文字列での引用符

文字エントリ内でリテラル引用符を指定する方法は 2 つあります。1 つめの方法は、引用符を 2 回連続して使用する方法です。たとえば、一重引用符で文字列を開始し、そのエントリの一部として一重引用符を 1 つ入力する場合は、一重引用符を 2 つ使用します。

```
'I don''t understand.'
```

二重引用符の場合は次のように指定します (quoted_identifier が Off の場合)。

```
"He said, ""It is not really confusing."""
```

2 つめの方法は、quoted_identifier が Off の場合にしか使用できませんが、引用符を別の種類の引用符で囲む方法です。つまり、二重引用符が入っているエントリは一重引用符で囲み、逆に一重引用符が入っているエントリは二重引用符で囲みます。次にその例を示します。

```
'George said, "There must be a better way."'
"Isn't there a better way?"
'George asked, "Isn''t there a better way?"'
```

参照

- 「quoted_identifier オプション」『SQL Anywhere サーバ データベース管理』

不定の値：NULL

カラムに NULL がある場合は、ユーザまたはアプリケーションがそのカラムになにも入力していないことを意味します。つまり、このカラムのデータ値は、不定か、使用不可です。

NULL は 0 (数値) やブランク (文字値) と同じではありません。むしろ、NULL 値によって、数値カラムの場合の 0 や文字カラムの場合のブランクなどの意図的な入力と、入力がない場合とを区別できます。入力が行われていない場合は、数値カラムと文字カラムは両方とも NULL です。

NULL の入力

NULL は、NULL 値が許可されたカラムにのみ入力できます。カラムに NULL 値が許可されるかどうかは、テーブルの作成時に決まります。カラムに NULL 値が許可されることを前提として、NULL を挿入します。

- **デフォルト**　データが入力されず、カラムに他のデフォルト設定がない場合。
- **明示的な入力**　引用符なしで NULL と明示的に挿入できます。NULL という単語を引用符をつけて文字カラムに入力すると、NULL は NULL 値としてではなく、1 つのデータとして処理されます。

たとえば、Departments テーブルの DepartmentHeadID カラムは NULL 値を許可します。次のように、マネージャのいない部署のローを 2 種類入力できます。

```
INSERT INTO Departments (DepartmentID, DepartmentName)
  VALUES (201, 'Eastern Sales')
INSERT INTO Departments
  VALUES (202, 'Western Sales', NULL);
```

NULL 値を返す

NULL 値は他の値とまったく同じように、クライアントアプリケーションに返され、表示されます。たとえば、次の例では、Interactive SQL で NULL 値がどのように表示されるかを示しています。

```
SELECT *
FROM Departments;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	904
300	Finance	1293
400	Marketing	1576
500	Shipping	703
201	Eastern Sales	(NULL)
202	Western Sales	(NULL)

カラム値を NULL と比較する

IS NULL 探索条件を使用すると、カラム値を NULL と比較したり、その比較の結果に基づいてカラム値を選択したりするなど、特定の動作を実行できます。TRUE の値を返すカラムだけが、選択されたり、結果として指定の動作を行ったりします。FALSE や UNKNOWN を返すカラムの場合、そういうことはありません。

次の例では、UnitPrice が $15 未満または NULL のローだけを選択します。

```
SELECT Quantity, UnitPrice
  FROM Products
  WHERE UnitPrice < 15
  OR UnitPrice IS NULL;
```

NULL が指定の値や別の NULL に等しいか (または等しくないか) わからないので、NULL 比較の結果はすべて UNKNOWN になります。

true を決して返さない条件があり、そうした条件を使用するクエリは結果セットを返しません。たとえば、NULL は不定の値があるという意味であるため、次の比較は決して true とは見なされません。

```
WHERE column1 > NULL
```

WHERE 句でカラム名を 2 つ使用する場合、つまり 2 つのテーブルをジョインする場合にも、この論理は適用されます。条件 WHERE column1 = column2 を含む句は、カラムに NULL が含まれるローを返しません。

次のパターンで NULL や NOT NULL も検索できます。

```
WHERE column_name IS NULL
```

```
WHERE column_name IS NOT NULL
```

次に例を示します。

```
WHERE advance < $5000
OR advance IS NULL
```

参照

- 「NULL 特別値」『SQL Anywhere サーバ SQL リファレンス』

NULL のプロパティ

次の項目で、NULL 値のプロパティを詳しく説明します。

- **FALSE と UNKNOWN の相違**　FALSE と UNKNOWN はどちらも値を返しませんが、FALSE と UNKNOWN の間には大きな論理的相違があります。false の反対 ("not false") は true ですが、UNKNOWN の反対は何かを知っていることを意味しません。たとえば、1 = 2 は false に評価され、1 != 2 (1 は 2 に等しくない) は true に評価されます。

  ただし、比較の中に NULL がある場合は、式を否定しても、反対のローセットや反対の真の値は得られません。UNKNOWN 値は UNKNOWN のままです。

- **複数の NULL 値を 1 つの値で置き換える**　ISNULL 組み込み関数を使用して、複数の NULL 値を 1 つの特定の値と置き換えることができます。置換は表示目的のためだけに実行されます。実際のカラム値には影響しません。構文は次のとおりです。

  ISNULL(*expression*, *value*)

たとえば、次の文を使用して Departments のすべてのローを選択し、すべての NULL 値を -1 という値でカラム DepartmentHeadID に表示します。

```
SELECT DepartmentID,
       DepartmentName,
       ISNULL( DepartmentHeadID, -1 ) AS DepartmentHead
  FROM Departments;
```

- **NULL と評価される式**　オペランドのいずれかが NULL 値の場合、算術演算子またはビット処理演算子のある式は NULL と評価されます。たとえば 1 + column1 は、column1 が NULL であれば NULL と評価されます。

- **文字列と NULL の連結**　文字列と NULL を連結すると、式は文字列と評価されます。たとえば、次の文は文字列 abcdef を返します。

```
SELECT 'abc' || NULL || 'def';
```

参照

- 「算術演算子」『SQL Anywhere サーバ SQL リファレンス』
- 「ビット処理演算子」『SQL Anywhere サーバ SQL リファレンス』

条件を接続する論理演算子

論理演算子 AND、OR、NOT を使用して、WHERE 句で複数の探索条件を接続します。1 つの文で複数の論理演算子が使用されている場合、通常は、AND 演算子が OR 演算子の前に評価されます。実行順序はカッコを使用して変更できます。

AND の使用

AND 演算子は 2 つ以上の条件を結合し、それらの条件のすべてが true である場合にだけ、結果を返します。たとえば次のクエリは、連絡先の姓が Purcell で、名前が Beth のローだけを検出します。

```
SELECT *
  FROM Contacts
  WHERE GivenName = 'Beth'
    AND Surname = 'Purcell';
```

OR の使用

OR 演算子は 2 つ以上の条件を結合し、それらの条件のうち、**いずれかが** true の場合に、結果を返します。次のクエリは、GivenName カラムの中で Elizabeth の別名を含むローを検索します。

```
SELECT *
  FROM Contacts
  WHERE GivenName = 'Beth'
    OR GivenName = 'Liz';
```

NOT の使用

NOT 演算子は、その後に来る式を否定します。次のクエリは、カリフォルニア州以外の連絡先をすべてリストします。

```
SELECT *
  FROM Contacts
  WHERE NOT State = 'CA';
```

日付を比較する探索条件

等号以外の演算子を使用して、探索条件と一致するローのセットを選択できます。不等号演算子 (< と >) を使用すると、数値、日付、文字列を比較できます。

例

Interactive SQL で次のクエリを実行して、1964 年 3 月 13 日より前に生まれたすべての従業員をリストします。

```
SELECT Surname, BirthDate
  FROM Employees
  WHERE BirthDate < 'March 13, 1964'
  ORDER BY BirthDate DESC;
```

Surname	BirthDate
Ahmed	1963-12-12
Dill	1963-07-19
Rebeiro	1963-04-12
Garcia	1963-01-23
Pastor	1962-07-14
...	...

注意

- **日付への自動変換**　SQL Anywhere データベースサーバは BirthDate カラムに日付が入っていることを認識して、'March 13, 1964' の文字列を自動的に日付に変換します。

- **日付の指定方法**　日付を指定するには、さまざまな方法があります。次に例を示します。

  ```
  'March 13, 1964'
  '1964/03/13'
  '1964-03-13'
  ```

  date_order オプションデータベースオプションを設定して、クエリに含まれる日付の解釈を設定できます。

  yyyy/mm/dd または *yyyy-mm-dd* フォーマットの日付は、date_order 設定がされているかどうかに関わらず、日付として常に正確に認識されます。

- **その他の比較演算子**　SQL Anywhere では、さまざまな比較演算子をサポートしています。

参照

- 「date_order オプション」『SQL Anywhere サーバ データベース管理』
- 「比較演算子」『SQL Anywhere サーバ SQL リファレンス』

音によるローの一致

SOUNDEX 関数を使用すると、ローを発音で対応させることができます。たとえば、**Ms. Brown** のように聞こえる名前を持つ従業員に対して電話メッセージが残されているとします。次のクエリを実行して、Brown のように聞こえる名前を持つ従業員を検索することができます。

> **注意**
> SOUNDEX によって使用されるアルゴリズムは、主に英語版データベースを対象としています。

例

Interactive SQL で次のクエリを実行して、Brown のように聞こえる姓の従業員をリストします。

```
SELECT Surname, GivenName
  FROM Employees
  WHERE SOUNDEX( Surname ) = SOUNDEX( 'Brown' );
```

Surname	GivenName
Braun	Jane

参照

- 「SOUNDEX 関数 [文字列]」『SQL Anywhere サーバ SQL リファレンス』

ORDER BY 句：結果の順序付け

特に指定しないかぎり、データベースサーバは、テーブルのローを意味のない順序で返します。テーブルのローは、多くの場合、意味のある順序にした方が便利です。たとえば、製品をアルファベット順に見たいとします。

SELECT 文の末尾に ORDER BY 句を追加して、結果セットのローの順序を指定します。この SELECT 文の構文は、次のとおりです。

SELECT *column-name-1*, *column-name-2*,...
FROM *table-name*
ORDER BY *order-by-column-name*

column-name-1、*column-name-2*、*table-name* を、問い合わせるカラムとテーブルの名前に置き換えてください。*order-by-column-name* はテーブルのカラムです。テーブルのすべてのカラムを表す省略形としてアスタリスクを使用できます。

注意

- **句の順序が重要**　ORDER BY 句は、FROM 句と SELECT 句の後に指定してください。

- **昇順または降順を指定できる**　デフォルトの順序は昇順です。次のクエリのように句の末尾にキーワード DESC を追加すると、降順を指定できます。

```
SELECT ID, Quantity
  FROM Products
  ORDER BY Quantity DESC;
```

ID	Quantity
400	112
700	80
302	75
301	54
600	39
...	...

- **複数のカラム順に順序を設定できる**　次のクエリは、最初にサイズ順 (アルファベット順)、次に名前順にソートします。

```
SELECT ID, Name, Size
  FROM Products
  ORDER BY Size, Name;
```

ID	Name	Size
600	Sweatshirt	Large
601	Sweatshirt	Large
700	Shorts	Medium
301	Tee Shirt	Medium
...	...	...

- **ORDER BY カラムが SELECT リストになくてもよい**　次のクエリは、価格が結果セットになくても、製品を単価順にソートします。

```
SELECT ID, Name, Size
  FROM Products
  ORDER BY UnitPrice;
```

ID	Name	Size
500	Visor	One size fits all

ID	Name	Size
501	Visor	One size fits all
300	Tee Shirt	Small
400	Baseball Cap	One size fits all
...	...	...

- **ORDER BY 句を使用せず、クエリを複数回実行すると、異なる結果が表示される可能性がある**　この理由は、SQL Anywhere が同じ結果セットを異なる順序で返す可能性があるためです。ORDER BY 句がない場合は、SQL Anywhere が最も効率のよい順序でローを返します。これは、結果セットの提示が、最後にアクセスしたローとその他の要因によって変化する可能性があることを意味します。特定の順序でローが返されるようにする唯一の方法は ORDER BY を使用することです。

例

Interactive SQL で次の文を実行して、製品をアルファベット順にリストします。

```
SELECT ID, Name, Description
  FROM Products
  ORDER BY Name;
```

ID	Name	Description
400	Baseball Cap	Cotton Cap
401	Baseball Cap	Wool cap
700	Shorts	Cotton Shorts
600	Sweatshirt	Hooded Sweatshirt
...	...	...

参照

- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』

ORDER BY のパフォーマンスを改善するインデックス

SQL Anywhere データベースサーバで ORDER BY 句を使用してクエリを実行するときに、複数の方法が考えられる場合があります。インデックスを使用すると、データベースサーバがより効率的にテーブルを検索できるようになります。

WHERE 句と ORDER BY 句を使ったクエリ

複数の実行方法があるクエリの一例は、WHERE 句と ORDER BY 句の両方を含むクエリです。

```
SELECT *
  FROM Customers
  WHERE ID > 300
  ORDER BY CompanyName;
```

この例で、SQL Anywhere は次の 2 つの方法のどちらを採用するか決定する必要があります。

1. Customers テーブル全体を、会社名の順序で検索し、各ローの顧客の ID の値が 300 以上かどうかをチェックする。

2. ID カラムのキーを使用して、300 を超える ID を持つ会社だけを読み込む。結果を会社名順にソートする必要があります。

ID 値が 300 を超える会社がほとんどない場合は、2 番目の方法の方が優れています。スキャンするローの数が少なく、ソートにも時間がかからないからです。大半の会社の ID 値が 300 を超える場合は、ソートの必要がない最初の方法の方がはるかに優れています。

問題の解決

ID と CompanyName の 2 つのカラムでインデックスを作成すれば、上の例を解決できます。SQL Anywhere は、このインデックスを使用してテーブルからローを適切な順序で選択できます。ただし、インデックスはデータベースファイルの領域を消費し、更新にオーバヘッドがかかることは覚えておく必要があります。インデックスの作成は慎重に行ってください。

参照

- 「インデックスを適切に選択するとパフォーマンスに大きな違いが生じる」223 ページ

クエリの集合関数

一部のクエリでは、テーブル内の、個々のローではなくローグループのプロパティを反映するデータの内容を検査します。たとえば、顧客が注文した総額の平均値や、各部門で何人の従業員が仕事をしているかなどです。この種のタスクには、「集合」関数と GROUP BY 句を使用します。

関数 COUNT、MIN、MAX は「集合関数」と呼ばれます。この 3 つの関数は、それぞれ情報を要約します。その他の集合関数として、AVG、STDDEV、VARIANCE などの統計関数があります。COUNT 以外のすべての集合関数では、パラメータが必要です。

集合関数は、一連のローについて単一の値を返します。GROUP BY 句がない場合、集合関数は「スカラー集合関数」と呼ばれ、クエリの他の条件を満たすすべてのローについて、単一の値を返します。GROUP BY 句がある場合、集合関数は「ベクトル集合関数」と呼ばれ、グループごとに 1 つの値を返します。

SQL Anywhere は、OLAP 関数と呼ばれることもある分析用の追加の集合関数をサポートしています。それらの関数のいくつかは Window 関数として使用されます。これには、RANK、PERCENT_RANK、CUME_DIST、ROW_NUMBER、線形回帰分析をサポートするための関数があります。

例

- **社内の従業員数のリスト**　Interactive SQL で次のクエリを実行します。

```
SELECT COUNT( * )
  FROM Employees;
```

COUNT()
75

この結果セットは、タイトル COUNT(*) を持つ 1 つのカラムと、従業員の合計数が入った 1 つのローで構成されています。

- **社内の従業員数と最年長の従業員および最年少の従業員の生年月日のリスト**　Interactive SQL で次のクエリを実行します。

```
SELECT COUNT( * ), MIN( BirthDate ), MAX( BirthDate )
  FROM Employees;
```

COUNT()	MIN(Employees.BirthDate)	MAX(Employees.BirthDate)
75	1936-01-02	1973-01-18

参照

- 「集合関数」『SQL Anywhere サーバ SQL リファレンス』
- 「OLAP のサポート」547 ページ

グループ分けされたデータに対する集合関数の使用方法

集合関数はテーブル全体についての情報を提供するだけでなく、ローのグループについても使用できます。GROUP BY 句は、ローをグループ単位で配置し、集合関数はローグループごとに 1 つの値を返します。

空のセットの意味の違い

SQL 言語では、集合関数を使用するとき、空のセットを扱う方法が異なります。GROUP BY 句がない場合、0 より大きい入力ローの集合関数があるクエリでは、結果として単一のローが返されます。COUNT の場合、結果は値 0 となり、その他のすべての集合関数では、結果は NULL となります。ただし、クエリに GROUP BY 句があり、クエリへの入力が空の場合、クエリの結果は空となり、ローは返されません。

たとえば、次のクエリでは値 0 の単一のローが返されます。部署 103 に従業員はいません。

```
SELECT COUNT() FROM Employees WHERE DepartmentID = 103;
```

ただし、この変更したクエリでは、GROUP BY 句が存在するためにローは返されません。

```
SELECT COUNT() FROM Employees WHERE DepartmentID = 103 GROUP BY State;
```

GROUP BY 句を使用する場合の一般的なエラー

GROUP BY 句を使用する場合の一般的なエラーは、1 つのグループにまとめることができない情報を得ようとすることです。たとえば、次のクエリではエラーが発生します。

```
SELECT SalesRepresentative, Surname, COUNT( * )
  FROM SalesOrders KEY JOIN Employees
  GROUP BY SalesRepresentative;
```

エラーメッセージには、GROUP BY 句にも Surname カラムへの参照が指定される必要があることが示されています。このエラーは、指定された ID を持つ従業員に対するそれぞれの結果ローで、姓が同じであることを SQL Anywhere が検証できないことによって発生します。

このエラーを解決するには、GROUP BY 句にカラムを追加します。

```
SELECT SalesRepresentative, Surname, COUNT( * )
  FROM SalesOrders KEY JOIN Employees
  GROUP BY SalesRepresentative, Surname
  ORDER BY SalesRepresentative;
```

この方法が適切でない場合は、代わりに集合関数を使用して、値を 1 つだけ選択できます。

```
SELECT SalesRepresentative, MAX( Surname ), COUNT( * )
  FROM SalesOrders KEY JOIN Employees
  GROUP BY SalesRepresentative
  ORDER BY SalesRepresentative;
```

MAX 関数は、各グループのディテールローから最大 (アルファベット順の最後) の姓 (Surname) を選択します。最大値は 1 つしかないので、この文は有効です。この場合は、グループ内のすべてのディテールローに同じ姓が表示されます。

例

Interactive SQL で次のクエリを実行して、営業担当と各担当が受けた注文数をリストします。

```
SELECT SalesRepresentative, COUNT( * )
  FROM SalesOrders
  GROUP BY SalesRepresentative
  ORDER BY SalesRepresentative;
```

SalesRepresentative	COUNT()
129	57
195	50
299	114
467	56
...	...

GROUPBY 句は SQL Anywhere に対して、もしこれがなければ返されるすべてのローのセットを分割するように指示します。各分割、つまりグループに含まれるすべてのローは、指定のカラムに同じ値を持ちます。ユニークな値ごと、または値セットごとに、1 つだけグループがあります。この場合、各グループに含まれるすべてのローの SalesRepresentative 値が同じになります。

COUNT などの集合関数は、各グループのローに適用されます。したがって、この結果セットには各グループのローの合計数が表示されます。クエリの結果は、各営業担当者の ID 番号が入った 1 つのローで構成されています。各ローには、営業担当者 ID と、その担当者の受注の合計数が入ります。

GROUP BY を使用した場合には、結果テーブルには GROUP BY 句に指定したカラムまたはカラムセットごとにローが 1 つずつあります。

参照

- 「GROUP BY 句：クエリ結果のグループへの編成」471 ページ
- 「GROUP BY 句」『SQL Anywhere サーバ SQL リファレンス』
- 「集合関数を伴う GROUP BY」474 ページ
- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』

HAVING 句

グループ内のローを制限するには、HAVING 句を使用します。

例

Interactive SQL 句で次の文を実行して、55 件を超える注文を持っているすべての営業担当をリストします。

```
SELECT SalesRepresentative, COUNT( * ) AS orders
FROM SalesOrders KEY JOIN Employees
GROUP BY SalesRepresentative
HAVING count( * ) > 55
ORDER BY orders DESC;
```

SalesRepresentative	orders
299	114
129	57
1142	57
467	56

参照

- 「HAVING 句：データグループの選択」477 ページ
- 「HAVING 句でのサブクエリ」601 ページ
- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』

WHERE 句と HAVING 句の組み合わせ

WHERE 句または HAVING 句を使用して、同じローセットを指定できる場合があります。このような場合に、効率的な方法とそうでない方法があります。オプティマイザは、入力された各文

を常に自動的に分析し、効率的な実行方法を選択します。意図する結果を最も明確に記述する構文を使用するのが最善です。通常は、前にある句の不要なローが削除されます。

例

受注数が 55 を超え、かつ ID が 1000 よりも大きいすべての営業担当者をリストするには、次の文を入力します。

```
SELECT SalesRepresentative, COUNT( * )
FROM SalesOrders
WHERE SalesRepresentative > 1000
GROUP BY SalesRepresentative
HAVING count( * ) > 55
ORDER BY SalesRepresentative;
```

次の SQL 文も同じ結果になります。

```
SELECT SalesRepresentative, COUNT( * )
FROM SalesOrders
GROUP BY SalesRepresentative
HAVING count( * ) > 55 AND SalesRepresentative > 1000
ORDER BY SalesRepresentative;
```

SQL Anywhere は、両方の文で同じ結果セットが記述されていることを検出するため、それぞれの文が効率的に実行されます。

高度：クエリ処理のフェーズ

この項では、文の注釈フェーズから実行完了までの各フェーズについて説明します。また、オプティマイザの設計の基本となる仮定条件について説明し、次に選択性推定、コスト推定、クエリ処理のステップについて説明します。

SQL Anywhere クエリ処理の詳細については、ホワイトペーパー『**Query Processing Based on SQL Anywhere 12.0.1 Architecture**』(http://www.sybase.com/detail?id=1096047) を参照してください。

結果セットのない文 (UPDATE 文や DELETE 文など) も、クエリ処理のフェーズを経由します。

- **注釈フェーズ**　データベースサーバは、クエリを受け取ると、パーサを使用して文を解析し、クエリの代数表現 (解析ツリー) に変換します。このフェーズでは、「解析ツリー」はセマンティックと構文のチェック (クエリで参照されるオブジェクトがカタログ内に存在することの検証など)、権限のチェック、定義済み参照整合性制約を使用した KEY JOIN と NATURAL JOIN 変換、非マテリアライズドビュー展開などに使用されます。このフェーズの出力は、解析ツリーの形式で書き換えられたクエリで、元のクエリで参照されるすべてのオブジェクトに対する注釈が含まれます。

- **セマンティック変形フェーズ**　このフェーズでは、クエリに対して反復的なセマンティック変形を実行します。クエリが注釈付きの解析ツリーとして表されることに変わりはありませんが、リライト最適化 (ジョインの削除、DISTINCT の削除、述部の正規化など) がこのフェーズで適用されます。このフェーズのセマンティック変形は、解析ツリー表現にヒューリスティックに適用されるセマンティック変形規則に従って実行されます。

データベースサーバによってプランがキャッシュ済みのクエリは、このクエリ処理のフェーズをスキップします。また、単純な文もこのフェーズをスキップする場合があります。たとえば、オプティマイザのバイパスでヒューリスティックプラン選択を使用する文の多くは、セマンティック変形フェーズで処理されません。このフェーズが文に適用されるかどうかは SQL 文の複雑さによって決まります。

- **最適化フェーズ**　最適化フェーズでは、クエリの別の内部表現であるクエリ最適化構造体を使用します。クエリ最適化構造体は、解析ツリーから構築されます。

  データベースサーバによってプランがキャッシュ済みのクエリでは、このクエリ処理フェーズはスキップされます。また、単純な文もこのフェーズをスキップする場合があります。

  このフェーズは、次の 2 つのサブフェーズに分かれています。

  - **最適化前フェーズ**　最適化前フェーズは、後から列挙フェーズで必要になる情報を最適化構造体に設定します。このフェーズでは、クエリを分析し、クエリアクセスプランで使用できる関連するインデックスとマテリアライズドビューをすべて検出します。たとえばこのフェーズでは、ビューマッチングアルゴリズムにより、クエリのすべてまたは一部を満たすために使用可能なすべてのマテリアライズドビューが特定されます。また、オプティマイザは、クエリの述部分析を基にして、クエリのテーブルをジョインするために列挙フェーズで使用可能な代替のジョイン方式を構築します。このフェーズでは、最適なクエリアクセスプランに関する決定は行われません。このフェーズの目的は、列挙フェーズの準備です。

  - **列挙フェーズ**　このフェーズでオプティマイザは、最適化前フェーズで生成した構成要素を使用して、クエリの可能なアクセスプランを列挙します。検索領域は非常に大きいため、オプティマイザは生成に独自の列挙アルゴリズムを使用し、生成されたアクセスプランを削除します。プランごとにコスト推定が計算されます。コスト推定は、それまでの最適なプランと現在のプランとを比較するために使用されます。この比較時に、コストの高いプランは廃棄されます。コスト推定では、リソースの使用 (ディスクや CPU の操作など)、中間結果のロー数の予測値、最適化ゴール、キャッシュサイズなどが考慮されます。列挙プランの出力は、クエリの最適なアクセスプランです。

- **プラン構築フェーズ**　プラン構築フェーズでは、最適なアクセスプランを利用して、クエリの実行に使用するクエリ実行プランの対応する最終表現を構築します。Interactive SQL のプランビューアで、プランのグラフィカルバージョンを表示できます。グラフィカルなプランにはツリー構造があり、各ノードは特定の関係代数演算を実装する物理演算子です。たとえば [ハッシュジョイン] や [順序付けされた Group By] は、それぞれジョイン操作や GROUP BY 操作を実装する物理演算子です。

  データベースサーバによってプランがキャッシュ済みのクエリでは、このクエリ処理フェーズはスキップされます。

- **実行フェーズ**　クエリの結果は、プラン構築フェーズで構築されたクエリ実行プランを使用して計算されます。

参照

- 「クエリ処理中に実行される最適化」327 ページ
- 「クエリ処理のフェーズをスキップするための条件」319 ページ
- 「プランのキャッシュ」325 ページ
- 「グラフィカルなプラン」333 ページ
- 「オプティマイザの仕組み」321 ページ

クエリ処理のフェーズをスキップするための条件

ほぼすべての文が、すべてのクエリ処理のフェーズを経由します。ただし、例外が主に 2 つあります。「プランのキャッシュ」の恩恵を受けるクエリ (データベースサーバによってプランがキャッシュ済みのクエリ) と「バイパスクエリ」です。.

- **プランのキャッシュ**　ストアドプロシージャまたはユーザ定義関数内のクエリの場合、データベースサーバは再利用できるように実行プランをキャッシュします。このクラスのクエリの場合、クエリ実行プランは実行後にキャッシュされます。次回クエリが実行されると、プランが取得され、実行フェーズまでのすべてのフェーズがスキップされます。

- **バイパスクエリ**　バイパスクエリは、データベースサーバがオプティマイザをバイパスできると認識する特定の特性を持った、単純なクエリのサブクラスです。最適化をバイパスすることで、実行プランの構築時間を削減できます。

  クエリがバイパスクエリとして認識されると、コストベースの最適化ではなくヒューリスティックが使用されます。つまり、セマンティック変形と最適化のフェーズはスキップされ、クエリの実行プランはクエリの解析ツリー表現から直接構築されます。

単純なクエリ

単純なクエリとは、次の特性を持った単一クエリブロックの SELECT 文、INSERT 文、DELETE 文、UPDATE 文です。

- クエリブロックに、サブクエリ、UNION、INTERSECT、EXCEPT などの追加クエリブロック、および共通テーブル式が含まれていない。

- クエリブロックが 1 つのベーステーブルまたはマテリアライズドビューを参照する。

- クエリブロックには、TOP N 句、FIRST 句、ORDER BY 句、DISTINCT 句を含めることができる。

- クエリブロックには、GROUP BY 句や HAVING 句を含まない集合関数を含めることができる。

- クエリブロックに、Window 関数が含まれていない。

- クエリブロック式に、NUMBER、IDENTITY、サブクエリが含まれていない。

- ベーステーブルに定義された制約が単純な式である。

セマンティック変形フェーズ後、複雑な文は単純な文に変形される場合があります。単純な文に変形されると、クエリをオプティマイザバイパスで処理したり、SQL Anywhere サーバでクエリプランをキャッシュしたりできます。

最適化の実行と最適化の非実行の強制

プランのキャッシュまたはオプティマイザのバイパスの条件を満たすクエリは、SQL Anywhere オプティマイザでの処理を強制できます。これには、任意の SQL 文で FORCE OPTIMIZATION 句を使用します。

また、文がオプティマイザをバイパスするように強制することもできます。これには、文の FORCE NO OPTIMIZATION 句を使用します。データベースオプションの設定またはスキーマやクエリの特性などが原因で、文が複雑すぎてオプティマイザをバイパスできない場合、クエリは失敗し、エラーが返されます。

FORCE OPTIMIZATION 句と FORCE NO OPTIMIZATION 句は、次の文の OPTION 句で使用できます。

- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「UPDATE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「INSERT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DELETE 文」『SQL Anywhere サーバ SQL リファレンス』

参照

- 「プランのキャッシュ」325 ページ

高度：クエリ最適化

最適化は、クエリの適切なアクセスプランを生成するのに重要な処理です。クエリが解析されると、「クエリオプティマイザ」 (または簡略して、オプティマイザ) はクエリを分析し、できるだけ少ないリソースを使用して結果を計算するアクセスプランを決定します。最適化が実行直前に開始されます。アプリケーションでカーソルを使用している場合は、カーソルを開いたときに最適化が開始されます。

他の多くの商用データベースシステムとは異なり、SQL Anywhere では通常、各文を実行する直前に最適化を行います。SQL Anywhere は各文の最適化をそのつど実行するため、オプティマイザはホスト変数とストアドプロシージャ変数の値にアクセスできます。これにより、より良い選択性推定分析を実行できます。また、最適化をそのつど実行するため、オプティマイザは前のクエリ実行後に保存された統計を基に、選択を調整できます。

SQL Anywhere は操作を効率化するために、セマンティック上は同等で、構文上は異なる形式にユーザのクエリを書き換えます。SQL Anywhere は、さまざまな書き換え操作を実行します。アクセスプランを読めば、それが元の文のリテラルな解釈と一致していないことがよくあります。たとえば、SQL 文の効率を高めるために、オプティマイザはジョインを使ってサブクエリを可能なかぎり書き換えようとします。

SQL Anywhere クエリ処理の詳細については、ホワイトペーパー『**Query Processing Based on SQL Anywhere 12.0.1 Architecture**』(http://www.sybase.com/detail?id=1096047) を参照してください。

オプティマイザの仕組み

オプティマイザの役割は、SQL 文を実行する効率的な方法を考案することです。これを行うには、オプティマイザは、クエリの実行プランを判断する必要があります。これには、クエリで参照されるテーブルのアクセス順序、各テーブルに使用されるジョイン演算子とアクセス方式、クエリの各部分の計算でクエリで参照されないマテリアライズドビューを使用できるかどうかなどの判断が含まれます。オプティマイザは、クエリで可能なアクセスプランを生成してコストを計算するときに、ジョイン列挙フェーズ中にクエリを実行するための最適なアクセスプランを選択します。最適なアクセスプランでは、オプティマイザの推測値が最短時間と最低コストで望ましい結果セットを返します。オプティマイザは、ディスクへの必要な読み書き回数を推定して、列挙された各方式のコストを決定します。

Interactive SQL で、**[ツール]** » **[プランビューア]** をクリックして、クエリの実行に使用される最適なアクセスプランを表示できます。

最初のローを返すコストの最小化

オプティマイザは、汎用的なディスクアクセスコストモデルを使用して、データベースファイルに対するランダム検索と逐次検索の相対的なパフォーマンスの差異を認識します。ALTER DATABASE 文を使用すると、データベースを特定のハードウェア構成に対応させることができます。

デフォルトでは、クエリ処理はすべての結果セットを返すように最適化されています。optimization_goal オプションを使用してデフォルトの動作を変更すると、最初のローを迅速に返すコストを最小限に抑えることができます。このオプションを First-row に設定すると、オプティマイザは、クエリの結果の最初のローをフェッチするまでの時間を短縮するアクセスプランを選択します。この場合、検索にかかる合計時間は長くなることがあります。

セマンティック上等しい構文の使用

ほとんどの文は、SQL 言語を使用してさまざまな方法で表すことができます。これらの表現は、同じタスクを実行するという点でセマンティック上は同等ですが、構文はまったく異なる場合があります。ごくわずかな例外はありますが、オプティマイザは、各文のセマンティックだけに基づいて適切なアクセスプランを考案します。

構文の違いは重要に見えるかもしれませんが、通常はまったく影響ありません。たとえば、クエリ構文で述部、テーブル、属性の順序が違ってもアクセスプランの選択には影響しません。クエリに非マテリアライズドビューが含まれているかどうかによってオプティマイザが影響を受けることもありません。

クエリを最適化するコストの削減

オプティマイザが実現可能な最も効率の良いアクセスプランを見つけるのが理想的ですが、通常、これは現実的ではありません。複雑なクエリの場合、さまざまな可能性が存在することがあります。

オプティマイザは効率的ですが、各オプションの分析には、時間とリソースを必要とします。オプティマイザは、これから行う最適化のコストと、これまでに見つけた最高のプランの実行にかかるコストを比較します。相対的にコストの低いプランが考案されると、オプティマイザは停止し、そのプランの実行を進めます。さらに最適化を行うと、すでに見つかっているアクセスプラ

ンを実行する場合よりも多くのリソースを消費する可能性があります。optimization_level オプションの値を高く設定することで、オプティマイザが費やす作業量を指定できます。

高いコストがかかる複雑なクエリの場合、または、最適化レベルが高く設定されている場合は、オプティマイザの動作時間が長くなります。コストが非常に高いクエリの場合は、クエリの実行時間が長くなり、認識できるほどの遅延が発生することがあります。

参照

- 「グラフィカルなプラン」333 ページ
- 「高度：クエリ実行プラン」328 ページ
- 「ALTER DATABASE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「optimization_level オプション」『SQL Anywhere サーバ データベース管理』
- 「optimization_goal オプション」『SQL Anywhere サーバ データベース管理』

オプティマイザの推定と統計

オプティマイザは、データベースに格納されている「カラム統計」と「ヒューリスティック」(発見的手法) に基づいて、文の処理方式を選択します。オプティマイザが検討するアクセスプランごとに、推定された結果サイズ (ローの数) を計算する必要があります。たとえば、クエリで使用される述部の選択性推定に基づいたジョイン方式やインデックスアクセスごとに、推定された結果サイズが計算されます。推定された結果サイズは、プランで使用される演算子ごと (ジョイン方式、GROUP BY 方式、逐次スキャンなど) にディスクアクセスや CPU の推定コストの計算に使用されます。カラム統計は、述部の選択性推定を計算するためにオプティマイザが使用するプライマリデータです。そのため、アクセスプランのコストを適切に推定するためにカラム統計は重要です。

カラム統計が古くなったりなくなったりすると、不正確な統計により非効率な実行プランとなる可能性があり、パフォーマンスが低下することがあります。パフォーマンスの悪化の原因が不正確なカラム統計にあると考えられる場合は、カラム統計を再作成してください。

オプティマイザによる統計の使用方法

オプティマイザが使用するカラム統計の最も重要なコンポーネントは、「ヒストグラム」です。ヒストグラムは、単一カラムの値の分散に関する情報を格納します。SQL Anywhere では、ヒストグラムはカラムのデータ分散を表します。これは、カラムのドメインを連続する値の範囲集合 (「バケット」とも呼ばれる) に分け、値の範囲 (バケット) それぞれに収まるカラム値のあるテーブルのロー数を記憶することで表されます。

SQL Anywhere では、テーブルの多数のローにある単一のカラム値が特に注目されます。重要な単一値の選択性は、単集合のヒストグラムバケット (たとえば、カラムドメインの単一の値を包含するバケット) で管理されます。SQL Anywhere は、各ヒストグラムの単集合バケットの数を最小限に抑えようとします。その数は、テーブルのサイズによって決まりますが、通常 10 から 100 の間です。また、選択性が 1% より大きい単一値はすべて単集合バケットとして管理されます。その結果、あるカラムのヒストグラムは、そのカラムの単一値の選択性のうち上位 *N* 個を記憶します。ここで *N* の値は、テーブルのサイズと 1% より大きい単一値の選択性の数によって決まります。

いったん値の範囲の数が最小数に達すると、頻度の高い選択性が出現するごとに頻度の低い選択性を置き換えます。ヒストグラムは、選択性が 1% より大きい値が十分あると判断した場合のみ、最小数を超える単集合の値の範囲を持ちます。

ベーステーブルとは異なり、FROM 句で実行されるプロシージャコールにはカラム統計がありません。したがって、オプティマイザでは、プロシージャコールからのデータの選択性推定にデフォルトまたは推測が使用されます。プロシージャコールの実行時間と、結果セット内のローの合計数は、以前の呼び出しから収集された統計を使用して推定されます。これらの統計は、ISYSPROCEDURE システムテーブルの stats カラムに格納されます。

オプティマイザによるヒューリスティックの使用方法

採用できそうな実行プラン内の各テーブルについて、オプティマイザは結果の一部となるローの数を推定します。ローの数は、テーブルのサイズとクエリの WHERE 句または ON 句に指定された制限によって異なります。

SQL Anywhere は、カラムに対する特定のクエリ述部を満たすローの数を、そのカラムに対するヒストグラムによって推定します。そのためには、指定の述部を満たす値を含むすべての範囲にあるローの数を合算します。クエリの結果セットに部分的に含まれるヒストグラムの値の範囲には、内挿法が使用されます。

多くの場合、オプティマイザはより高度なヒューリスティックを使用します。たとえば、オプティマイザは適切な統計がない場合にかぎってデフォルトの推定値を使用します。また、オプティマイザは、インデックスとキーを使用してロー数の推定精度を上げます。単一のカラムで推定する例を次に示します。

- カラム内である値を持つロー数 : そのカラムがユニークインデックスを持つかプライマリキーである場合、ロー数は 1 つだと推定します。
- インデックス付きカラムで定数と比較したときのロー数 : インデックスを調査し、比較条件を満たすローのパーセンテージを推定します。
- 外部キーからプライマリキーへのロー数 (キージョイン) : テーブルの相対的サイズを使って推定します。たとえば、5000 ローのテーブルが 1000 ローのテーブルに対して外部キーを持つとき、オプティマイザは 1 つのプライマリキーローに対して 5 個の外部キーローがあると推定します。

参照

- 「ヒント : カラム統計の更新」221 ページ
- 「選択性推定ソース」324 ページ
- 「ESTIMATE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「ESTIMATE_SOURCE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSPROCEDURE システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_get_histogram システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「ヒストグラムユーティリティ (dbhist)」『SQL Anywhere サーバ データベース管理』

選択性推定ソース

任意の述部で、オプティマイザは選択性推定の次のいずれかのソースを使用できます。選択されたソースは、クエリのグラフィカルな長いプランに示されます。

- **Statistics**　オプティマイザは、格納されたカラム統計を使用して選択性推定を計算できます。述部で定数が使用されている場合、格納された統計情報を使用できるのは、定数の選択性が統計情報に格納されるだけ重大な数値であるときだけです。

  たとえば、述部 EmployeeID > 100 では、EmployeeID カラムの統計情報が存在する場合、カラム統計を選択性推定ソースとして使用できます。

- **Join**　オプティマイザは、参照整合性制約、一意性制約、またはジョインヒストグラムを使用して選択性推定を計算できます。ジョインヒストグラムは、T.X カラムおよび R.X カラムの使用可能な統計情報から T.X=R.X の形式の述部に対して計算されます。

- **Column-column**　選択性ソースとして使用できる参照整合性制約、一意性制約、またはジョインヒストグラムがないジョインの場合、オプティマイザは選択性ソースとして、ジョインの結果セットの推定ロー数を 2 つのテーブルの直積のロー数で割った値を使用できます。

- **Column**　オプティマイザは、カラム統計に格納されたすべての値の平均値を使用できます。

  たとえば、述部 DepartmentName = expression の選択性は、*expression* が定数でない場合、平均値を使用して計算できます。

- **Index**　オプティマイザは、インデックスを調査して選択性推定を計算できます。通常、インデックスは、カラム統計などの選択性推定の他のソースが使用できない場合に選択性推定に使用されます。

  たとえば、述部 DepartmentName = 'Sales' では、オプティマイザはカラム DepartmentName で定義されたインデックスを使用して Sales の値を持つローの数を推定できます。

- **User**　user_estimates データベースオプションが Disabled に設定されていない場合、オプティマイザはユーザが提供する選択性推定を使用できます。

- **Guess**　使用できる適切なインデックスがない場合、参照先カラムの統計情報が収集されていない場合、または述部が複雑である場合、オプティマイザは最も妥当な推測を行って選択性推定を計算できます。この場合は、各タイプの述部に組み込み規則が定義されます。

- **Computed**　たとえば、選択性を乗算するか、加算して選択性推定が計算された場合、非常に複雑な述部では、たとえば、選択性推定を 100% に設定し、選択性ソースを Computed に設定できます。

- **Always**　述部が常に true の場合、選択性ソースは 'Always' になります。たとえば、述部 1=1 は常に true です。

- **Combined**　上記の複数のソースを組み合わせて選択性推定が計算される場合、選択性ソースは 'Combined' になります。

- **Bounded**　SQL Anywhere が選択性推定に上限か下限またはその両方を設定した場合、選択性ソースは 'Bounded' になります。たとえば、限界は、推定値が 100% を超えたり、選択性が 0% より小さくなったりしないことを保証するために設定されます。

参照

- 「ESTIMATE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「ESTIMATE_SOURCE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「グラフィカルなプラン内の選択性情報」339 ページ
- 「ヒント：明示的な選択性推定の提供を控える」225 ページ
- 「sa_get_histogram システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「INDEX_ESTIMATE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「EXPERIENCE_ESTIMATE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「ヒント：明示的な選択性推定の提供を控える」225 ページ
- 「user_estimates オプション」『SQL Anywhere サーバ データベース管理』

プランのキャッシュ

通常、オプティマイザはクエリが実行されるたびに、その実行プランを選択します。実行時に最適化すると、オプティマイザは、現在のシステム状態、現在の選択性推定値、ホスト変数の値に基づく推定値に従ってプランを選択できます。頻繁に実行されるクエリの場合は、実行時に最適化することによる利点よりもクエリ最適化のコストの方が重要である場合があります。SQL Anywhere サーバは、これらの文を繰り返し最適化するコストを削減するために、次の文に関するプランのキャッシュを検討します。

- ストアドプロシージャ、ユーザ定義関数、トリガ内で実行されるすべての文。

- 最適化をバイパスするための条件を満たす SELECT 文、INSERT 文、UPDATE 文、DELETE 文。

  INSERT 文については、INSERT...VALUES 文だけがキャッシュの条件を満たします。INSERT...ON EXISTING 文は条件を満たしません。

  UPDATE 文と DELETE 文については、WHERE 句が存在し、プライマリキーを使用してローを識別する探索条件を含んでいる必要があります。プランをキャッシュする場合は、他の探索条件は指定できません。また、UPDATE 文については、変数の代入を含む SET 句がある文はキャッシュの条件を満たしません。

1 つの接続でこのいずれかの文が複数回実行されたら、オプティマイザはホスト変数の値が不明なまま文の再利用可能なプランを構築します。選択性推定やセマンティッククエリ変形にホスト変数の値を使用できないため、再利用可能なプランのコストは高くなる場合があります。再利用可能なプランの構造が、文が以前に実行されたときのプランと同じである場合、データベースサーバは、そのプランキャッシュに再利用可能なプランを追加します。最適化を避けることによるコストの削減よりも、実行のたびに最適化することによる利点が勝る場合、実行プランはキャッシュされません。

文で参照されないマテリアライズドビューを実行プランで使用し、materialized_view_optimization オプションが Stale 以外に設定されている場合、実行プランはキャッシュされず、ストアドプロシージャ、ユーザ定義関数、トリガの次回呼び出し時に文は再度最適化されます。

プランキャッシュは、アクセスプランを実行したときに使われるデータ構造の接続ごとのキャッシュです。キャッシュされたプランの再利用には、キャッシュ内のプランを調べて初期状態にリセットする処理も含まれます。通常、これはすべてのクエリ処理フェーズを経由させて文を処理するよりも、はるかに高速です。使用頻度が低く、キャッシュの使用量が増えない場合、キャッシュされたプランはディスクに格納されます。オプティマイザはクエリを定期的に最適化し直して、キャッシュされたプランが依然として効率的であるかどうかを確認します。

キャッシュできるプランの最大数は、max_plans_cached オプションで指定します。デフォルトは 20 です。プランのキャッシュを無効にするには、このオプションを 0 に設定します。

QueryCachedPlans 統計を使用して、現在キャッシュされているクエリ実行プラン数を表示できます。このプロパティを検索するために CONNECTION_PROPERTY 関数を使用すると、特定の接続についてキャッシュされているクエリ実行プラン数を表示できます。DB_PROPERTY 関数を使用すると、接続全体でキャッシュされている実行プランを数えることができます。このプロパティを QueryCachePages、QueryOptimized、QueryBypassed、QueryReused と組み合わせて使用すると、max_plans_cached オプションの最適な設定を決定するのに役立ちます。

データベースプロパティまたは接続プロパティである QueryCachePages を使用すると、実行プランをキャッシュするために使用するページ数を決定できます。これらのページはテンポラリファイル内の領域を占めますが、メモリに常駐するとはかぎりません。

クエリ実行プランは、実行時間の長いクエリについてはキャッシュされません。これは、クエリの最適化を避けることのメリットが、クエリの総コストに比べて小さいためです。さらに、データベースサーバでは、ホスト変数の値に大きく左右されるクエリの再利用可能なクエリプランの再構築は行われません。

参照

- 「クエリ処理のフェーズをスキップするための条件」319 ページ
- 「マテリアライズドビュー」53 ページ
- 「materialized_view_optimization オプション」『SQL Anywhere サーバ データベース管理』
- 「DB_PROPERTY 関数 [システム]」『SQL Anywhere サーバ SQL リファレンス』
- 「CONNECTION_PROPERTY 関数 [システム]」『SQL Anywhere サーバ SQL リファレンス』
- 「高度：クエリ処理のフェーズ」317 ページ
- 「接続プロパティの一覧」『SQL Anywhere サーバ データベース管理』
- 「max_plans_cached オプション」『SQL Anywhere サーバ データベース管理』

サブクエリと関数のキャッシュ

SQL Anywhere は、サブクエリを処理すると、結果をキャッシュします。このキャッシュは要求ごとに行われるので、キャッシュされた結果が同時に実行された要求や接続の間で共有されることはありません。同じ相関値のセットについてサブクエリの再評価が必要な場合、SQL Anywhere ではキャッシュから結果を取り出すだけで済みます。このようにして、SQL Anywhere は、何度も繰り返される冗長な計算を避けます。要求が完了すると (クエリのカーソルが閉じられると)、SQL Anywhere はキャッシュされた値を解放します。

クエリの処理が進むに従って、SQL Anywhere は、キャッシュされたサブクエリの値が再使用された頻度をモニタします。相関変数の値がめったに繰り返されない場合、SQL Anywhere は、ほとんどの値を 1 回しか計算する必要がありません。このような場合、SQL Anywhere は、一度し

か発生しない数多くのエントリをキャッシュするよりも、たまに重複する値を再計算する方が効率的であると判断します。そのため、データベースサーバは文の残りの部分についてこのサブクエリのキャッシュを中断し、外部クエリブロック内のすべてのローに関するサブクエリの再評価を開始します。

従属カラムのサイズが 255 バイトを超える場合も、SQL Anywhere はキャッシュを行いません。その場合、クエリを書き換えるか、または別のカラムをテーブルに追加して、操作をより効率的にします。

関数のキャッシュ

一部の組み込み関数とユーザ定義関数は、サブクエリの結果と同じ方法でキャッシュされます。このため、同じパラメータを使用したクエリの処理中に呼び出される高コストの関数のパフォーマンスが大幅に向上します。ただし、これは関数の呼び出し回数が予想より少なくなることを意味しています。

関数がキャッシュされるためには、2 つの条件を満たす必要があります。

●特定のパラメータセットに対して常に同じ結果を戻す

●基本となるデータに対し副次的な影響を与えない

これらの条件を満たす関数は、「決定性」関数、または「べき等」関数と呼ばれます。SQL Anywhere では、すべてのユーザ定義関数は、作成時に NOT DETERMINISTIC と宣言されないかぎり「決定性」として扱われます。つまり、データベースサーバは、同じパラメータを持つ同じ関数が連続して 2 回呼び出されている場合は、どちらの呼び出しでも同じ結果が返され、クエリの意味に不要な弊害は生じないものとみなします。

組み込み関数は、決定性関数として扱われますが、いくつか例外があります。RAND、NEWID、GET_IDENTITY 関数は非決定性関数として扱われ、その結果はキャッシュされません。

参照

● 「CREATE FUNCTION 文」『SQL Anywhere サーバ SQL リファレンス』

クエリ処理中に実行される最適化

クエリ書き換えフェーズでは、より効率性と便宜性に優れたクエリの表現方法を検討するために、SQL Anywhere はセマンティック変形を実行します。クエリはセマンティック上等しいクエリに書き換えられる場合があるため、このプランは、元のクエリとはかなり異なる場合があります。一般的な操作は次のとおりです。

●不要な DISTINCT 条件の排除

●サブクエリのネスト解除

●UNION または GROUP 化されたビューや派生テーブルでの述部のプッシュダウンの実行

●OR 述部と IN リスト述部の最適化

- LIKE 述部の最適化
- 外部ジョインの内部ジョインへの変換
- 外部ジョインと内部ジョインの削除
- 述部の推定による利用可能な条件の発見
- 大文字と小文字の不要な変換の排除
- サブクエリを EXISTS 述部として書き換え
- 検索引数可能な IN 述部の推定 (AND 述部に変換できない OR 述部からの部分インデックススキャンに使用可能)

注意
カーソルが更新可能な場合、メインクエリブロックではクエリ書き換え最適化を実行できないことがあります。カーソルを読み込み専用として宣言すると、最適化を利用できます。

クエリ書き換えフェーズで実行される書き換えの最適化のいくつかは、REWRITE 関数で返される結果で観察できます。

参照

- 「カーソルタイプ」『SQL Anywhere サーバ プログラミング』
- 「DECLARE CURSOR 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』
- 「REWRITE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』

高度：クエリ実行プラン

実行プランは、データベースサーバがデータベース内の文に関連する情報にアクセスするために使用するステップのセットです。最適化されたばかりかどうか、オプティマイザをバイパスしたかどうか、プランが以前の実行からキャッシュされたかどうかなどに関係なく、文の実行プランの保存と確認が可能です。クエリの実行プランは、元の文で使用される構文と正確に対応するとはかぎりません。クエリで明示的に指定したベーステーブルの代わりにマテリアライズドビューを使用する場合があります。ただし、実行プランに記述された操作は、セマンティック上は元のクエリと同等です。

Interactive SQL 内で、または SQL 関数を使用することによって、実行プランを表示できます。実行プランを取り出すときに、次のようなフォーマットを選択できます。

- 短いテキストプラン
- 長いテキストプラン
- グラフィカルなプラン
- グラフィカルなプラン (ルート統計あり)
- グラフィカルなプラン (全統計あり)
- Ultra Light (短い、長い、またはグラフィカル)

クエリ実行プランのテキスト表示には、短いプランと長いプランの 2 種類があります。SQL 関数を使用して、テキストプランにアクセスできます。プランには、グラフィカルなバージョンも

あります。GRAPHICAL_PLAN と EXPLANATION の各関数を使用し、特定のカーソルタイプに基づいて SQL クエリのプランを取得することもできます。

SQL Anywhere クエリ処理の詳細については、ホワイトペーパー『**Query Processing Based on SQL Anywhere 12.0.1 Architecture**』(http://www.sybase.com/detail?id=1096047) を参照してください。

参照

- 「GRAPHICAL_PLAN 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「EXPLANATION 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「高度：クエリ処理のフェーズ」317 ページ
- 「グラフィカルなプランの表示」341 ページ
- 「高度：クエリ実行プラン」328 ページ
- 「グラフィカルなプラン」333 ページ

参照

- 「短いテキストプラン」329 ページ
- 「長いテキストプラン」330 ページ
- 「グラフィカルなプラン」333 ページ

短いテキストプラン

短いテキストプランは、プランを短時間で比較する場合に便利です。短いプランでは、すべてのプランフォーマットの最低限の情報が 1 行で表示されます。

次の例では、ORDER BY 句によって結果セット全体がソートされるため、プランは Work[Sort で始まります。Customers テーブルは、プライマリキーインデックス CustomersKey によってアクセスされます。カラム Customers.ID がプライマリキーのため、インデックススキャンを使用して探索条件が満たされます。省略形 JNL は、Customers と SalesOrders の間のジョインを処理するためにオプティマイザでマージジョインが選択されたことを示します。最後に、外部キーインデックス FK_CustomerID_ID を使用して SalesOrders テーブルがアクセスされ、Customers テーブル内で CustomerID が 100 未満であるローが検索されます。

```
SELECT EXPLANATION ('SELECT GivenName, Surname, OrderDate
FROM Customers JOIN SalesOrders
WHERE CustomerID < 100
ORDER BY OrderDate');
```

```
Work[ Sort[ Customers<CustomersKey> JNL SalesOrders<FK_CustomerID_ID> ] ]
```

ジョイン方式の区切りにコロンを使用

次に示す文には、「クエリブロック」が 2 つ含まれています。1 つは SalesOrders テーブルと SalesOrderItems テーブルを参照する外部 SELECT ブロック、もう 1 つは Products テーブルから選択するサブクエリです。

```
SELECT EXPLANATION ('SELECT *
FROM SalesOrders AS o
  KEY JOIN SalesOrderItems AS I
WHERE EXISTS
  ( SELECT *
```

```
FROM Products p
WHERE p.ID = 300 )');
```

o<seq> JNL i<FK_ID_ID> : p<ProductsKey>

各クエリブロックのジョイン方式はコロンで区切られます。短いプランでは常に、メインブロックのジョイン方式が先にリストされます。その後に、他のクエリブロックのジョイン方式がリストされます。このような他のクエリブロックのジョイン方式の順序は、文におけるクエリブロックの順序やその実行順序とは一致しない場合があります。

参照

- 「実行プランのコンポーネント」342 ページ

長いテキストプラン

長いテキストプランでは、短いテキストプランより多くの情報が、スクロールしなくても簡単に印刷したり表示したりできる状態で提供されます。長いプランには文のキャッシュされたプランなどの情報が含まれており、アプリケーションプロファイリングで使用した場合は、クエリが最適化された方法や部分インデックススキャンで使用された述部に関する追加情報が含まれています。

例 1

この例では、長いテキストプランは次の文に基づいています。

```
SELECT PLAN ('SELECT GivenName, Surname, OrderDate, Region, Country
FROM Customers JOIN SalesOrders ON ( SalesOrders.CustomerID = Customers.ID )
WHERE CustomerID < 100 AND ( Region LIKE "Eastern"
    OR Country LIKE "Canada" )
ORDER BY OrderDate');
```

長いテキストプランは次のように表示されます。

```
( Plan [ Total Cost Estimate: 6.46e-005, Costed Best Plans: 1, Costed Plans: 10, Optimization Time:
0.0011462,
Estimated Cache Pages: 348 ]
  ( WorkTable
    ( Sort
      ( NestedLoopsJoin
        ( IndexScan Customers CustomersKey[ Customers.ID < 100 : 0.0001% Index | Bounded ] )
        ( IndexScan SalesOrders FK_CustomerID_ID[ Customers.ID = SalesOrders.CustomerID :
0.79365% Statistics ]
          [  ( SalesOrders.CustomerID < 100 : 0.0001% Index | Bounded )
          AND  ( ( ((Customers.Country LIKE 'Canada' : 100% Computed)
          AND (Customers.Country = 'Canada' : 5% Guess))
          OR ((SalesOrders.Region LIKE 'Eastern' : 100% Computed)
          AND (SalesOrders.Region = 'Eastern' : 5% Guess)) ) : 100% Guess )  ] )
      )
    )
  )
)
```

Plan という語はクエリブロックの開始を示します。Total Cost Estimate は、オプティマイザでプランの実行に要すると推測された時間 (ミリ秒単位) です。Costed Best Plans、Costed Plans、Optimization Time は最適化処理の統計であり、Estimated Cache Pages は文の処理に使用できる現在の推定キャッシュサイズです。

このプランは、結果がソートされ、ネストループジョインが使用されることを示します。ジョイン演算子と同じ行に、ジョイン条件とその選択性推定 (ジョイン演算子によって作成されるすべてのローについて推定) があります。IndexScan の行は、Customers と SalesOrders の各テーブルが、それぞれ CustomersKey と FK_CustomerID_ID の各インデックスを使用してアクセスされることを示します。

例 2

次の文がプロシージャ、トリガ、または関数内で使用され、文のプランがキャッシュされて 5 回再利用された場合、長いテキストプランには、文が再利用可能であり、キャッシュ後に 5 回使用されたことを示す文字列 [R: 5] が含まれます。文で使用されるパラメータ parm1 は、このプランでは不定の値を持ちます。

```
UPDATE Account SET Account.A = 10 WHERE Account.B =parm1

( Update [ Total Cost Estimate: 1e-006, Costed Best Plans: 1, Costed Plans: 2, Carver pages: 0,
Estimated Cache Pages: 46768 ] [ R: 5 ]
  ( Keyset
    ( TableScan ( Account ) ) [ Account.B = parm1 : 0.39216% Column ]
  )
  )
)
```

同じ文のプランがまだキャッシュされていない場合、長いテキストプランにはパラメータ parm1 の値 (10 など) が含まれており、プランはこのパラメータの値を使用して最適化されたことを示しています。

```
( Update [ Total Cost Estimate: 1e-006, Costed Best Plans: 1, Costed Plans: 2, Carver pages: 0,
Estimated Cache Pages: 46768 ]
  ( Keyset
    ( TableScan ( Account ) ) [ Account.B = parm1 [ 10 ] : 0.001% Statistics ]
  )
  )
)
```

参照

- 「実行プランのコンポーネント」342 ページ

短いテキストプランの表示 (SQL の場合)

短いテキストプランでは、クエリ実行プランの短いバージョンが表示されます。

前提条件

その関数が実行されるオブジェクトの所有者であるか、または、そのオブジェクトに対する SELECT、UPDATE、DELETE、INSERT のうち、いずれか適切な権限を持っている必要があります。

♦ タスク

1. データベースに接続します。
2. EXPLANATION 関数を実行します。

結果

短いテキストプランは、Interactive SQL の **[結果]** ウィンドウ枠に表示されます。

例

この例では、短いテキストプランは次の文に基づいています。

```
SELECT EXPLANATION ('SELECT GivenName, Surname, OrderDate
FROM GROUPO.Customers JOIN GROUPO.SalesOrders
WHERE CustomerID < 100
ORDER BY OrderDate');
```

短いテキストプランは次のように表示されます。

```
Work[ Sort[ Customers<CustomersKey> JNL SalesOrders<FK_CustomerID_ID> ] ]
```

ORDER BY 句によって結果セット全体がソートされるため、短いテキストプランは Work[Sort で始まります。Customers テーブルは、プライマリキーインデックス CustomersKey によってアクセスされます。カラム Customers.ID がプライマリキーのため、インデックススキャンを使用して探索条件が満たされます。省略形 JNL は、Customers と SalesOrders の間のジョインを処理するためにオプティマイザでマージジョインが選択されたことを示します。最後に、外部キーインデックス FK_CustomerID_ID を使用して SalesOrders テーブルがアクセスされ、Customers テーブル内で CustomerID が 100 未満であるローが検索されます。

参照

- 「EXPLANATION 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「実行プランのコンポーネント」342 ページ
- 「短いテキストプラン」329 ページ
- 「長いテキストプランの表示 (SQL の場合)」332 ページ

長いテキストプランの表示 (SQL の場合)

長いテキストプランでは、文のキャッシュされたプランなどの、短いテキストプランよりも詳細な情報が、印刷しやすくスクロールなしで表示できるフォーマットで提供されます。

前提条件

その関数が実行されるオブジェクトの所有者であるか、または、そのオブジェクトに対する SELECT、UPDATE、DELETE、INSERT のうち、いずれか適切な権限を持っている必要があります。

♦ タスク

1. データベースに接続します。
2. PLAN 関数を実行します。

結果

長いテキストプランは、Interactive SQL の **[結果]** ウィンドウ枠に表示されます。

例

この例では、長いテキストプランは次の文に基づいています。

```
SELECT PLAN ('SELECT GivenName, Surname, OrderDate, Region, Country
FROM GROUPO.Customers JOIN GROUPO.SalesOrders ON ( SalesOrders.CustomerID =
Customers.ID )
WHERE CustomerID < 100 AND ( Region LIKE "Eastern"
     OR Country LIKE "Canada" )
ORDER BY OrderDate');
```

長いテキストプランは次のように表示されます。

```
( Plan [ Total Cost Estimate: 6.46e-005, Costed Best Plans: 1, Costed Plans: 10, Optimization Time:
0.0011462,
Estimated Cache Pages: 348 ]
 ( WorkTable
  ( Sort
   ( NestedLoopsJoin
    ( IndexScan Customers CustomersKey[ Customers.ID < 100 : 0.0001% Index | Bounded ] )
    ( IndexScan SalesOrders FK_CustomerID_ID[ Customers.ID = SalesOrders.CustomerID :
0.79365% Statistics ]
     [  ( SalesOrders.CustomerID < 100 : 0.0001% Index | Bounded )
     AND  ( ( ((Customers.Country LIKE 'Canada' : 100% Computed)
     AND (Customers.Country = 'Canada' : 5% Guess))
     OR ((SalesOrders.Region LIKE 'Eastern' : 100% Computed)
     AND (SalesOrders.Region = 'Eastern' : 5% Guess)) ) : 100% Guess )  ] )
   )
  )
 )
)
```

Plan という語はクエリブロックの開始を示します。Total Cost Estimate は、オプティマイザでプランの実行に要すると推測された時間 (ミリ秒単位) です。Costed Best Plans、Costed Plans、Optimization Time は最適化処理の統計であり、Estimated Cache Pages は文の処理に使用できる現在の推定キャッシュサイズです。

このプランは、結果がソートされ、ネストループジョインが使用されることを示します。ジョイン演算子と同じ行に、ジョイン条件とその選択性推定 (ジョイン演算子によって作成されるすべてのローについて推定) があります。IndexScan の行は、Customers と SalesOrders の各テーブルが、それぞれ CustomersKey と FK_CustomerID_ID の各インデックスを使用してアクセスされることを示します。

参照

- 「実行プランのコンポーネント」342 ページ
- 「PLAN 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「長いテキストプラン」330 ページ
- 「短いテキストプランの表示 (SQL の場合)」331 ページ

グラフィカルなプラン

Interactive SQL のグラフィカルなプラン機能を使用すると、クエリの実行プランを **[プランビューア]** ウィンドウに表示できます。実行プランは、関係代数演算子のツリーで構成されます。ツリーのルートで最終的な結果が得られるように、ツリーの最下位から開始して、クエリへの基

本入力 (通常はテーブルのロー) を消費し、下から上にローを処理します。このツリーのノードは特定の代数演算子に対応しています。ただし、サーバで実行されるすべてのクエリ検査がノードで表示されるとはかぎりません。たとえば、サブクエリと関数のキャッシュの影響は、グラフィカルなプランには直接表示されません。

グラフィカルなプランでは、次のようにノードをさまざまな形で表示して、実行される操作の種類を示します。

- データを実体化する操作は六角形。
- インデックススキャンは台形。
- テーブルスキャンは長方形。
- その他の操作は角丸四角形。

グラフィカルなプランを使用して、特定のクエリのパフォーマンス上の問題を診断できます。たとえば、プラン内の情報に基づいて、この特定のクエリのパフォーマンスを向上させるためにテーブルにインデックスが必要かを判断できます。**[プランビューア]** で **[保存]** ボタンをクリックすると、クエリのグラフィカルなプランを保存して後で参照できます。SQL Anywhere のグラフィカルなプランは、拡張子 *.saplan* で保存されます。

パフォーマンスに問題がある可能性がある場合は、グラフィカルなプラン内で太い線と赤い枠線で示されます。次に例を示します。

- 処理対象のロー数が増加するに従って、プラン内のノード間の線が太くなります。テーブルスキャンに太い線が表示されている場合は、インデックスの作成が必要である可能性を示していることがあります。
- ノードの枠線が赤い場合は、実行プラン内の他の演算子に比べて、その演算子のコストが高かったことを示しています。

プランのノードの形とその他のグラフィカルなコンポーネントは、Interactive SQL 内でカスタマイズできます。

グラフィカルなプラン、概要付きのグラフィカルなプラン、または詳細な統計情報付きのグラフィカルなプランのいずれかを表示できます。3 つのプランはすべて、プランの中で最も高コストとして推定された部分を表示できます。統計情報付きのグラフィカルなプランの生成は、クエリの実行時にデータベースサーバがモニタしている実際のクエリ実行統計が表示されるため高コストです。統計情報付きのグラフィカルなプランでは、クエリオプティマイザがアクセスプランの作成時に使用する推定を、実行中にモニタされた実際の統計と直接比較できます。ただし、オプティマイザはクエリのコストを正確に推定できないことが多いので、推定値と実際の統計値には違いがあると想定してください。

参照

- 「グラフィカルなプランの表示」341 ページ
- 「アプリケーションプロファイリング」148 ページ
- 「高度：クエリ実行プラン」328 ページ
- 「グラフィカルなプランのカスタマイズ」『SQL Anywhere サーバ データベース管理』

統計情報付きのグラフィカルなプラン

グラフィカルなプランは、短いテキストプランや長いテキストプランよりも多くの情報を提供します。統計情報付きのグラフィカルなプランを生成する場合は負荷が高くなりますが、クエリの実行時にデータベースサーバがモニタしている実際のクエリ実行統計が表示されます。このため、オプティマイザがアクセスプランの作成時に使用する推定を、実行中にモニタされた実際の統計と直接比較できます。推定と実際の統計が大きく異なる場合は、情報不足のためにオプティマイザが正確にクエリのコストを推定できず、その結果非効率な実行プランになっていることがあります。

統計情報付きのグラフィカルなプランを生成するには、データベースサーバで文を実行する必要があります。実行時間が長い文のグラフィカルなプランを生成すると、かなり時間がかかる場合があります。UPDATE、INSERT、または DELETE 文の場合は、文の読み込み専用の部分だけが実行されます。テーブルの操作は実行されません。ただし、文にユーザ定義関数が含まれる場合は、クエリの一部としてユーザ定義関数が実行されます。ユーザ定義関数に副次的影響 (たとえば、ローの変更、テーブルの作成、コンソールへのメッセージ送信など) がある場合、統計情報付きのグラフィカルなプランを取得するときにこれらの変更が実行されます。統計情報付きのグラフィカルなプランの取得後に ROLLBACK 文を実行して、これらの副次的影響を元に戻すことができる場合があります。

参照

- 「ROLLBACK 文」『SQL Anywhere サーバ SQL リファレンス』

統計情報付きのグラフィカルなプランによるパフォーマンス分析

統計情報付きのグラフィカルなプランを使用して、データベースのパフォーマンス上の問題を識別できます。統計情報付きのグラフィカルなプランのフィールドの詳細については、「実行プランのコンポーネント」342 ページを参照してください。

クエリの実行上の問題の識別

クエリの実行に影響するデータベースオプションやその他のグローバルな設定をルート演算子ノードについて表示できます。

選択性のパフォーマンスの確認

述部の選択性 (条件式) は、条件を満たすローの割合のことです。推定される述部の選択性が提供する情報に基づいて、オプティマイザはコストの推定を行います。オプティマイザの正常な処理には、正確な選択性推定が不可欠です。たとえば、オプティマイザが述部の選択性が高い (5% の選択性など) と誤って推定しても、実際には述部の選択性がかなり低い (50% など) 場合は、パフォーマンスが低下することがあります。選択性推定は正確ではない場合がありますが、極端に大きな誤差は問題がある可能性を示しています。

クエリの重要な部分の選択性情報が不正確であると判断した場合、CREATE STATISTICS を使用してカラムの新しい統計値セットを生成できます。まれに、明示的な選択性推定を指定したい場合があります。ただし、この方法では、後で統計を更新するときに問題が発生する可能性があります。

クエリがバイパスクエリだと判断された場合、選択性統計値は表示されません。

次のような場合に不適切な選択性の兆候が見られます。

- **RowsReturned の実際値と推定値**　**RowsReturned** は結果セット内のロー数です。**RowsReturned** 統計値は、ツリーの先頭にあるルートノードのテーブル内に表示されます。推定ローカウントが実際のローカウントと大きく異なる場合は、このノードまたはサブツリーに接続された述部の選択性が正しくない可能性があります。

- **述部選択性の実際値と推定値**　「述部」というサブヘッダを検索して、述部の選択性を確認します。

  ヒストグラムが存在しないベースカラムに対する述部がある場合は、ヒストグラムを作成するために CREATE STATISTICS 文を実行すると問題を修正できる場合があります。

  選択性の誤差に問題が残る場合は、クエリテキストに述部とともにユーザ推定選択性を指定することも可能です。

- **推定ソース**　選択性推定のソースも、**[統計情報]** ウィンドウ枠の述部サブヘッダの下にリストされています。

  述部選択性推定のソースが **Guess** の場合、オプティマイザには述部のフィルタリングの特性の判断に使用できる情報がないため、ヒストグラムがないなどの問題を示す可能性があります。推定ソースが **Index** で、選択性推定が正しくない場合は、インデックスが偏っていることが問題である可能性があります。REORGANIZE TABLE 文を使用してインデックスの断片化を解除すると改善できる場合があります。

キャッシュのパフォーマンスの確認

キャッシュの読み込み数 (**[CacheRead]** フィールド) とヒット数 (**[CacheHits]** フィールド) が同じである場合は、この SQL 文のために処理されるすべてのオブジェクトがキャッシュに保持されています。キャッシュの読み込み数がヒット数よりも多い場合は、テーブルまたはインデックスページがサーバのキャッシュに存在しないため、データベースサーバがこれらをディスクから読み込んでいることを示します。これは、ハッシュジョインなどの場合に予想されます。ネストループジョインなどの場合、キャッシュヒット率が低いときは、クエリを効率的に実行するためのキャッシュ (バッファプール) の不足を示している可能性があります。この場合、サーバのキャッシュサイズを増やすと改善できることがあります。

無効なインデックスの識別

クエリ実行プランからは、インデックスがパフォーマンスの向上に役立っているかどうかが不明な場合がよくあります。SQL Anywhere で使用されているスキャンベースのクエリ操作の中には、インデックスを使用しないで多くのクエリに対して最高のパフォーマンスを提供するものもあります。

データの断片化の問題の識別

Runtime および **FirstRowRunTime** の実際値と推定値は、ルートノード統計値で提供されます。ノードに **RunTime** が存在する場合、**[サブツリー統計]** セクションにはこの値だけが表示されます。

RunTime の解釈は、表示される統計セクションによって異なります。**[ノード統計]** の場合、**RunTime** はこのノードだけを実行している間に、対応する演算子が費やした累積時間です。**[サ**

ブツリー統計] の場合、**RunTime** はこのノードの直下にある演算子のサブツリー全体に対して費やされた合計実行時間を表します。したがって、ほとんどの演算子では **RunTime** と **FirstRowRunTime** は独立した測定値で、個別に分析する必要があります。

FirstRowRunTime は、このノードの中間結果の最初のローを作成するために要した時間です。

テーブルスキャンまたはインデックススキャンで、ノードの **RunTime** が予想よりも大きい場合、REORGANIZE TABLE 文を実行するとパフォーマンスを向上できることがあります。sa_table_fragmentation() システムプロシージャと sa_index_density() システムプロシージャを使用して、テーブルまたはインデックスが断片化されているかどうかを判断できます。

参照

- 「REORGANIZE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE STATISTICS 文」『SQL Anywhere サーバ SQL リファレンス』
- 「オプティマイザの仕組み」321 ページ
- 「明示的な選択性推定」『SQL Anywhere サーバ SQL リファレンス』
- 「テーブルの断片化削減」241 ページ
- 「実行プランのコンポーネント」342 ページ
- 「グラフィカルなプラン内の選択性情報」339 ページ
- 「ヒント：インデックスの有効な使用」223 ページ
- 「インデックスコンサルタント」156 ページ
- 「ヒント：パフォーマンス向上のためのキャッシュの使用」212 ページ

グラフィカルなプランのノードの詳細情報

グラフィカルなプランでノードの詳細情報を表示するには、グラフィカルな図の左ウィンドウ枠でノードをクリックします。右側の **[詳細]** ウィンドウ枠と **[高度な詳細]** ウィンドウ枠にノードに関する詳細情報が表示されます。**[詳細]** ウィンドウ枠の次の 3 つのメインセクションにノードの統計値が表示されます。

- ノード統計
- サブツリー統計
- オプティマイザ統計

ノード統計は、特定のノードの実行に関する統計です。プランノードには **[詳細]** ウィンドウ枠があり、演算子に対する推定値、実測値、および平均値の統計が表示されます。どのノードについても操作を複数回実行でできます。たとえば、ネストループジョインノードの右側にリーフノードが表示されると、リーフノード演算子からローを複数回フェッチできます。この場合、リーフノード (逐次、インデックス、または RowID スキャンノード) の **[詳細]** ウィンドウ枠には、実行ごとの (平均の) 統計と実行時の実際の累積統計の両方が含まれます。

ノードがリーフノードではない場合、ノードは他のノードの中間結果を消費し、**[詳細]** ウィンドウ枠の **[サブツリー統計]** セクションには、このノードのサブツリー全体に対する推定と実際の累積統計が表示されます。SQL 要求全体を表すオプティマイザ統計情報は、ルートノードだけに存在します。オプティマイザ統計値は特に文の最適化に関連しており、最適化のゴール設定、最適化レベル設定、検討するプラン数などの値があります。

注文日で顧客を並べ替える次のクエリを考えてみます。

```
SELECT GROUPO.Customers.GivenName, GROUPO.Customers.Surname,
GROUPO.SalesOrders.OrderDate
FROM Customers KEY JOIN SalesOrders
WHERE CustomerID > 100
ORDER BY OrderDate
```

このクエリのグラフィカルなプランでは、**Hash Join (JH)** ノードが選択され、このノードに関連する情報だけが右ウィンドウ枠に表示されています。**[述部]** の説明によれば、Hash Join ノードに適用された述部は Customers.ID = SalesOrders.CustomerID : 0.79365% Statistics | Join です。Customers ノードをクリックすると、スキャン述部によれば、Customers ノードに適用された述部は Customers.ID > 100 : 100% Index; であることがわかります。

注意
次の例のクエリを実行すると、**[プランビューア]** に表示されたのとは異なるプランが取得される場合があります。データベースの設定や最近実行されたクエリなど、多くの要因によってオプティマイザによるプランの選択が影響を受ける可能性があります。

[高度な詳細] ウィンドウ枠に表示される情報は、各演算子によって異なります。ルートノードの場合、**[高度な詳細]** ウィンドウ枠に、クエリが最適化されたときに有効になっていた接続オプションの設定が表示されます。他の種類のノードでは、**[高度な詳細]** ウィンドウ枠に、特定のノードの処理で検討されたインデックスまたはマテリアライズドビューに関する情報が表示される場合があります。

グラフィカルなプランの各ノードのコンテキスト別ヘルプを表示するには、ノードを右クリックして **[ヘルプ]** をクリックします。

注意
クエリがバイパスクエリとして認識されている場合、一部の最適化ステップがバイパスされ、**[クエリオプティマイザ]** セクションと **[述部]** セクションはどちらもグラフィカルなプランに表示されません。

参照

- 「オプティマイザの仕組み」321 ページ
- 「グラフィカルなプラン」333 ページ
- 「グラフィカルなプランの表示」341 ページ
- 「高度：クエリ実行プラン」328 ページ
- 「実行プランのコンポーネント」342 ページ

グラフィカルなプラン内の選択性情報

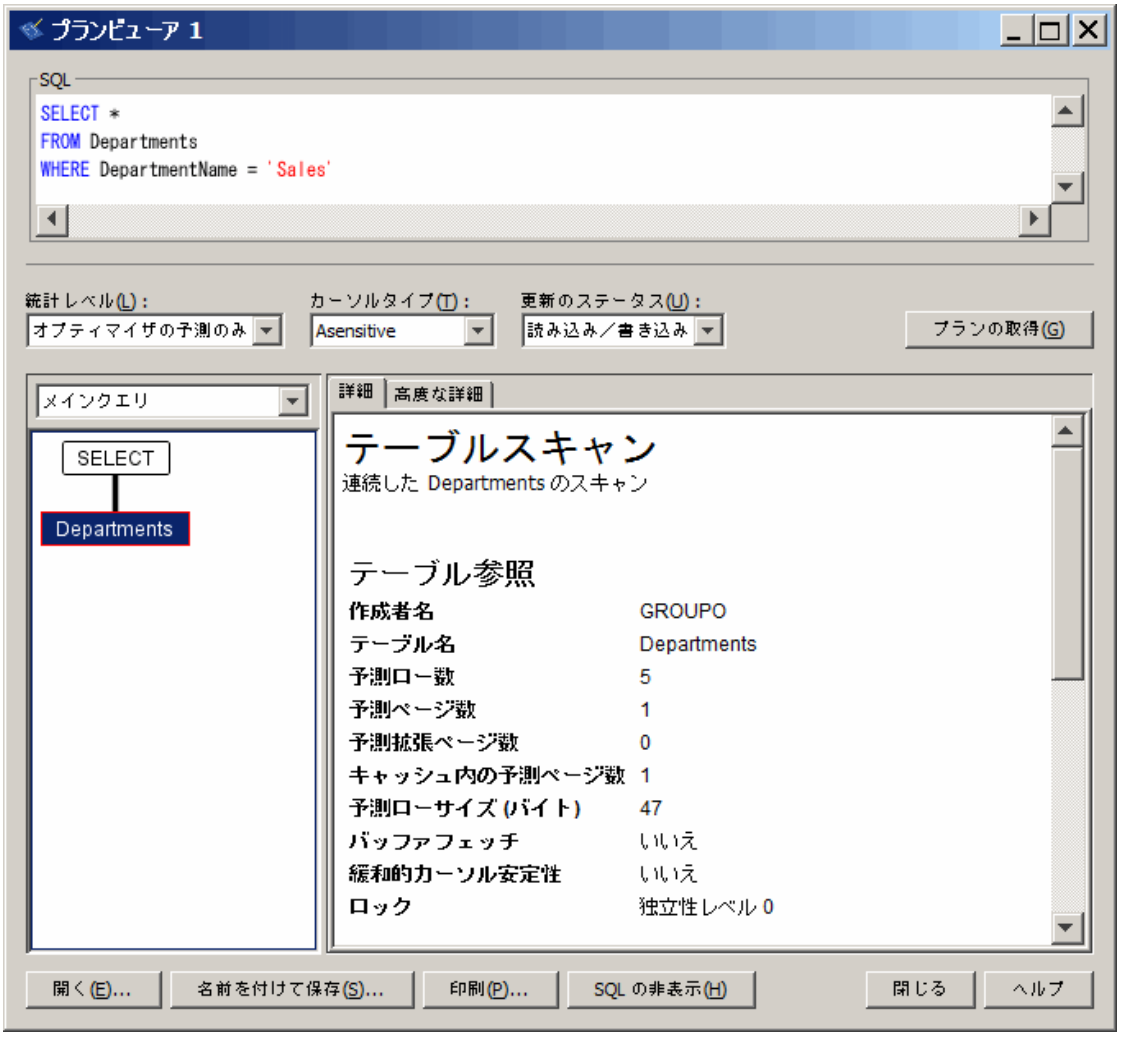
次の例では、選択されたノードが Departments テーブルのスキャンを表し、統計情報ウィンドウ枠には探索条件、選択性推定、実際の選択性として **[述部]** が表示されています。

[詳細] ウィンドウ枠の **[ノード統計]**、**[サブツリー統計]**、**[オプティマイザ統計]** の 3 つのセクションに、各ノードに関する統計値が表示されます。

ノード統計は、特定のノードの実行に関連しています。プラン内のノードがリーフノードではなく、他のノードの中間結果を消費する場合、**[詳細]** ウィンドウ枠の **[サブツリー統計]** セクションには、対象ノードのサブツリー全体に対する推定と実際の累積統計が表示されます。オプティマイザ統計情報はルートノードだけに存在し、SQL 要求全体を表します。

バイパスクエリには、選択性の情報が表示されない場合があります。バイパスクエリの詳細については、「オプティマイザの仕組み」321 ページを参照してください。

アクセスプランは、データベース内で使用可能な統計値によって決まります。この統計値は、どのクエリが実行済みかによります。ここでは、各種の統計値やプランを確認できます。



この述部記述は次のとおりです。

Departments.DepartmentName = 'Sales' : 20% Column; true 1/5 20%

この述部は次のように解釈できます。

●Departments.DepartmentName = 'Sales' は述部です。

●20% は、オプティマイザによる選択性推定です。つまり、オプティマイザは 20% のローが述部を満たすという推定に基づいてクエリアクセスを選択しています。

これは、ESTIMATE 関数で得られる出力と同じです。

●Column は推定ソースです。これは、ESTIMATE_SOURCE 関数で得られる出力と同じです。

●true 1/5 20% は、実行時における述部の実際の選択性です。述部は 5 回評価され、その内の 1 回が TRUE だったので、実際の選択性は 20% になります。

実際の選択性が推定値と大幅に異なる場合、および述部の評価回数が非常に多い場合は、不正確な推定によりクエリのパフォーマンスに重大な問題が発生している可能性があります。述部の統計値を収集することは、オプティマイザにその選択の基礎となる良質な情報を提供することとなり、パフォーマンスを改善できます。

注意
統計情報付きのグラフィカルなプランではなく、グラフィカルなプランを選択すると、最後の 2 つの統計情報は表示されません。

参照

- 「ESTIMATE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「選択性推定ソース」324 ページ

グラフィカルなプランの表示

Interactive SQL **[プランビューア]** または GRAPHICAL_PLAN 関数を使用して、グラフィカルなプランを表示できます。GRAPHICAL_PLAN 関数を使用すると、グラフィカルなプランを XML フォーマットで表示できます。

前提条件

なし。

♦ タスク

1. Interactive SQL を起動して、SQL Anywhere データベースに接続します。
2. **[ツール]** » **[プランビューアを開く]** をクリックします (または [Shift + F5] キーを押します)。
3. **[SQL]** ウィンドウ枠に文を入力します。
4. **[統計レベル]**、**[カーソルタイプ]**、**[更新のステータス]** を選択します。
5. **[プランの取得]** をクリックします。

結果

[結果] ウィンドウ枠にグラフィカルなプランが表示されます。

参照

- 「GRAPHICAL_PLAN 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「詳細とノードの統計を含むグラフィカルなプランの作成」『SQL Anywhere サーバ データベース管理』
- 「実行プランのコンポーネント」342 ページ
- 「高度：クエリ実行プラン」328 ページ
- 「グラフィカルなプランのカスタマイズ」『SQL Anywhere サーバ データベース管理』

実行プランのコンポーネント

次に、実行プランに表示される省略形を示します。

プランに表示されるアルゴリズムの詳細については、ホワイトペーパー『**Query Processing Based on SQL Anywhere 12.0.1 Architecture**』(http://www.sybase.com/detail?id=1096047) を参照してください。

短いテキストプラン	長いテキストプラン	その他の情報
	見積り済み最良プラン	オプティマイザは、特定のクエリのアクセスプランを生成してコストを計算します。この処理の最中、現在の最良プランが、より低いコストが推定される新しい最良プランで置き換えられる場合があります。最後の最良プランが、文の実行に使用する実行プランになります。見積り済み最良プランは、現在の最良プランよりも優れたプランをオプティマイザが検出した回数を示しています。数字が小さい場合は、最良プランが列挙プロセスの早い段階で決定されたことを示しています。オプティマイザは特定の文のクエリブロックごとに 1 回以上の列挙プロセスを起動するため、見積り済み最良プランは累積回数を表します。
	見積り済みプラン	オプティマイザによって生成される多数のプランが、それまでに検出された最良プランと比較してコストが高いと判断されます。見積り済みプランは、指定された文の列挙プロセスの最中にオプティマイザが検討した一部のプランまたは完全なプランの数を表します。
**	**	完全なインデックススキャン。インデックススキャンではすべてのローが読み込まれます。

短いテキストプラン	長いテキストプラン	その他の情報
DELETE	Delete	削除操作のルートノード。
DistH	HashDistinct	HashDistinct は 1 つの入力を受け取り、すべての排他ローを返します。
DistO	OrderedDistinct	OrderedDistinct では、各ローが読み込まれ、前のローと比較されます。両者が同じであれば後の入力ローは無視され、それ以外の場合は出力されます。
DP	DecodePostings	DecodePostings は、テキストインデックス内の単語の位置情報を復号化します。
DT	DerivedTable	DerivedTable は、特にクエリに 1 つ以上の外部ジョインが含まれる場合に、クエリ書き換え最適化などのさまざまな理由でプランに表示される場合があります。
EAH	HashExceptAll	SQL の集合差異演算子 EXCEPT のハッシュベースの実装が使用されたことを示します。
EAM	MergeExceptAll	SQL の集合差異演算子 EXCEPT のソートベースの実装が使用されたことを示します。
EH	HashExcept	SQL の集合差異演算子 EXCEPT のハッシュベースの実装が使用されたことを示します。
EM	MergeExcept	SQL の集合差異演算子 EXCEPT のソートベースの実装が使用されたことを示します。

短いテキストプラン	長いテキストプラン	その他の情報
Exchange	Exchange	SELECT 文の処理時にクエリ内並列処理が使用されたことを示します。
Filter	Filter	任意のタイプの述部、subselect を含む比較、EXISTS と NOT EXISTS の各サブクエリ (その他の形式の限定サブクエリ) などの探索条件の適用を示します。
GrByH	HashGroupBy	HashGroupBy では、グループごとに 1 つのローで構成されるインメモリハッシュテーブルが構築されます。入力ローが読み込まれると、ワークテーブル内で関連グループが検索されます。集合関数が更新され、グループローがワークテーブルに再び書き込まれます。グループレコードが見つからなければ、新しいグループレコードが初期化され、ワークテーブルに挿入されます。
GrByHClust	HashGroupByClustered	入力テーブルのグループ化カラム内の値はクラスタ化されているため、似た値が互いに近接して現れることがあります。ClusteredHashGroupBy は、このクラスタ化を利用します。
GrByHP	ParallelHashGroupBy	HashGroupBy の変形。
GrByHSets	HashGroupBySets	GROUPING SETS クエリを実行する場合、HashGroupBy の変形である HashGroupBySets が使用されます。

短いテキストプラン	長いテキストプラン	その他の情報
GrByO	OrderedGroupBy	OrderedGroupBy では、グループ化カラムによって順序付けされた入力が読み込まれます。各ローは、読み込まれるたびに前のローと比較されます。グループ化カラムが一致すると、現在のグループが更新されます。それ以外の場合は、現在のグループが出力され、新しいグループが開始されます。
GrByOSets	OrderedGroupBySets	GROUPING SETS クエリを実行する場合、OrderedGroupBy の変形である OrderedGroupBySets が使用されます。
GrByS	SingleRowGroupBy	GROUP BY を指定しなければ、単一ローの集合の生成に SingleRowGroupBy が使用されます。各入力ローに対して単一のグループローがメモリに格納され、更新されます。
GrBySSets	SortedGroupBySets	SortedGroupBySets は、GROUPING SETS を含む OLAP クエリを処理する場合に使用されます。
HF	HashFilter	ハッシュフィルタ (ブルームフィルタ) が使用されたことを示します。
HFP	ParallelHashFilter	ハッシュフィルタ (ブルームフィルタ) が使用されたことを示します。
HTS	HashTableScan	ハッシュテーブルスキャンが使用されたことを示します。
IAH	HashIntersectAll	SQL の集合差異演算子 INTERSECT のハッシュベースの実装が使用されたことを示します。

短いテキストプラン	長いテキストプラン	その他の情報
IAM	MergeIntersectAll	SQL の集合差異演算子 INTERSECT のソートベースの実装が使用されたことを示します。
IH	HashIntersect	SQL の集合差異演算子 INTERSECT のハッシュベースの実装が使用されたことを示します。
IM	MergeIntersect	SQL の集合差異演算子 INTERSECT のソートベースの実装が使用されたことを示します。
IN	InList	InList は、インデックスを使用して IN リスト述部を満たすことができる場合に使用されます。
table-name<*index-name*>	IndexScan、ParallelIndexScan	グラフィカルなプランで、インデックススキャンの場合は台形の中にインデックス名が表示されます。
INSENSITIVE	Insensitive	
INSERT	Insert	INSERT 操作のルートノード。
IO	IndexOnlyScan、ParallelIndexOnlyScan	クエリを満たすために必要なすべてのデータを含んだインデックスをオプティマイザが使用したことを示します。

短いテキストプラン	長いテキストプラン	その他の情報
JH	HashJoin	HashJoin は、2 つの入力のうち小さい方のインメモリハッシュテーブルを作成してから、大きい方の入力を読み込みます。次に、インメモリハッシュテーブルを調査して一致するローを検索します。見つかったローはワークテーブルに書き込まれます。小さい方の入力がメモリに収まらない場合は、HashJoin によって両方の入力が小さなワークテーブルに分割されます。これらの小さくなったワークテーブルは、小さい方の入力がメモリに収まるようになるまで再帰的に処理されます。
JHS	HashSemijoin	HashSemijoin は、左側と右側のセミジョインを実行します。
JHSP	ParallelHashSemijoin	HashJoin の変形。
JHFO	Full Outer HashJoin	HashJoin の変形。
JHA	HashAntisemijoin	HashAntisemijoin は、左側と右側の非セミジョインを実行します。
JHAP	ParallelHashAntisemijoin	HashJoin の変形。
JHO	Left Outer HashJoin	HashJoin の変形。
JHP	ParallelHashJoin	HashJoin の変形。
JHPO	ParallelLeftOuterHashJoin	HashJoin の変形。
JHR	RecursiveHashJoin	HashJoin の変形。
JHRO	RecursiveLeftOuterHashJoin	HashJoin の変形。

短いテキストプラン	長いテキストプラン	その他の情報
JM	MergeJoin	MergeJoin は 2 つの入力を読み込みます。このとき、2 つの入力はどちらもジョイン属性で順序付けされています。左側の入力のローごとに、右側の入力のローにソート順にアクセスすることで、一致する右側のローをすべて読み込みます。
JMFO	Full Outer MergeJoin	MergeJoin の変形。
JMO	Left Outer MergeJoin	MergeJoin の変形。
JNL	NestedLoopsJoin	NestedLoopsJoin では、左側のローごとに右側全体が読み込まれ、左側と右側のジョインが計算されます。
JNLA	NestedLoopsAntisemijoin	NestedLoopsAntisemijoin は左側のローごとに右側をスキャンして、入力をジョインします。
JNLFO	Full Outer NestedLoopsJoin	NestedLoopsJoin の変形。
JNLO	Left Outer NestedLoopsJoin	NestedLoopsJoin の変形。
JNLS	NestedLoopsSemijoin	NestedLoopsSemijoin は左側のローごとに右側をスキャンして、入力をジョインします。
KEYSET	Keyset	キーセット駆動型カーソルを指定します。
LOAD	Load	ロード操作のルートノード。
MultiIdx	MultipleIndexScan	論理演算子 AND または OR で組み合わされた探索条件のセットを含むクエリを満たすために、複数のインデックスを使用できる、または使用する必要がある場合は、MultipleIndexScan を使用します。

短いテキストプラン	長いテキストプラン	その他の情報
OpenString	OpenString	OpenString は、OPENSTRING 句を含む SELECT 文の FROM 句で使用されます。
	Optimization Time	指定された文のすべての列挙プロセスにオプティマイザが費やした合計時間。
PC	ProcCall	プロシージャコール (テーブル関数)。
PreFilter	PreFilter	フィルタでは、任意のタイプの述部、subselect を含む比較、EXISTS と NOT EXISTS の各サブクエリ (その他の形式の限定サブクエリ) などの探索条件が適用されます。
R	R	リバースインデックススキャン。インデックススキャンでローがインデックスから逆の順序で読み込まれます。
RL	RowLimit	RowLimit は、入力の最初の n 個のローを返し、残りのローは無視します。ローの制限は、SELECT 文の TOP n または FIRST 句によって設定されます。
ROWID	RowIdScan	グラフィカルなプランで、ロー ID スキャンの場合は長方形の中にテーブル名が表示されます。
ROWS	RowConstructor	RowConstructor は、他のアルゴリズムへの入力として使用できる仮想ローを作成する特殊な演算子です。
RR	RowReplicate	RowReplicate は、EXCEPT ALL や INTERSECT ALL などの集合操作の実行時に使用されます。

短いテキストプラン	長いテキストプラン	その他の情報
RT	RecursiveTable	再帰テーブルがクエリ内の WITH 句の結果として使用されたことを示します。WITH 句は、再帰的な UNION クエリに使用されました。
RU	RecursiveUnion	RecursiveUnion は、再帰的な UNION クエリの実行中に使用されます。
SELECT	Select	選択操作のルートノード。
seq	TableScan、ParallelTableScan	グラフィカルなプランで、テーブルスキャンの場合は長方形の中にテーブル名が表示されます。
Sort	Sort	インデックスソートまたはマージソート。
SrtN	SortTopN	SortTopN は、TOP N 句と ORDER BY 句を含むクエリで使用されます。
TermBreak	TermBreak	全文検索の TermBreaker アルゴリズム。
UA	UnionAll	UnionAll では、重複に関係なく各入力からローが読み込まれ、出力されます。このアルゴリズムは、UNION 文と UNION ALL 文を実装するために使用されます。
UPDATE	Update	更新操作のルートノード。
Window	Window	Window は、Window 関数を使用する OLAP クエリを評価する場合に使用します。
Work	Work table	中間結果を表す内部ノード。

[オプティマイザ統計] フィールドの説明

グラフィカルなプランの **[オプティマイザ統計]**、**[ローカルオブティマイザ統計]**、**[グローバルオプティマイザ統計]** セクションの説明を次に示します。これらの統計は、データベースサーバの状態と、最適化プロセスについての情報を提供します。

フィールド	説明
最適化方法	実行方式の選択に使用されたアルゴリズム。戻り値は次のとおりです。 ● バイパス (クエリ書き換え) ● バイパス (簡易クエリ書き換え) ● バイパス (ヒューリスティック) ● バイパス後、最適化 ● 最適化 ● 再利用 ● 再利用 (単純)
見積り済み最良プラン	クエリオプティマイザが異なるクエリ実行方式を列挙するとき、現在の方式の前に検出された最良な方式よりも推定された負荷が低い方式が検出された回数を追跡します。特定のクエリに対してこの発生回数を予測することは困難ですが、この数値が小さい場合、オプティマイザのアルゴリズムによって検索領域が大幅に削減され、通常は、最適化時間が短縮されることを示します。オプティマイザは特定の文のクエリブロックごとに 1 回以上の列挙プロセスを起動するため、**[見積り済み最良プラン]** は累積回数を表します。 **[見積り済み最良プラン]**、**[見積り済みプラン]**、**[最適化時間]** の値が 0 の場合、SQL Anywhere オプティマイザによって文は最適化されていません。データベースサーバはこの文をバイパスし、文の最適化を行わずに実行プランを生成したか、または文のプランはキャッシュされました。
見積り済みプラン	コストを部分的または完全に推定したこの要求に対して、オプティマイザが検討した異なるアクセスプランの数。**[見積り済み最良プラン]** と同様、この値が小さい場合は通常、最適化時間の短縮を示し、この値が大きい場合は SQL クエリがより複雑であることを示します。 **[見積り済み最良プラン]**、**[見積り済みプラン]**、**[最適化時間]** の値が 0 の場合、文は最適化されていません。データベースサーバは文をバイパスし、文の最適化を行わず実行プランを生成しています。

フィールド	説明
最適化時間	文の最適化に要した時間。 **[見積り済み最良プラン]**、**[見積り済みプラン]**、**[最適化時間]** の値が 0 の場合、文は最適化されていません。データベースサーバは文をバイパスし、文の最適化を行わず実行プランを生成しています。
予測キャッシュページ数	文の処理に使用できる現在の推定キャッシュサイズ。 非効率なアクセスプランを削減するため、オプティマイザは現在のキャッシュサイズの半分は選択された文の処理に使用できると想定します。
CurrentCacheSize	最適化時のデータベースサーバのキャッシュサイズ (キロバイト単位)。
QueryMemMaxUseful	この要求に使用できるクエリメモリのページ数。この数値が 0 の場合、文の実行プランにはメモリを大量に消費する演算子がなく、サーバのメモリガバナーの制御対象ではありません。
QueryMemNeedsGrant	この要求の実行方式に存在し、メモリを大量に消費する 1 つ以上のクエリ実行演算子に対して、メモリガバナーがメモリを付与する必要があるかどうかを示します。
QueryMemLikelyGrant	この文がすぐに実行される場合、この文に付与されるクエリメモリプールの推定ページ数。この推定値は、プラン内のメモリを大量に消費する演算子の数、データベースサーバのマルチプログラミングレベル、メモリを大量に消費する同時実行の要求数によって異なります。
QueryMemPages	すべての接続に対して、メモリを大量に消費するクエリ実行アルゴリズムで使用できるクエリメモリプール内のメモリの合計容量を表すページ数。
QueryMemActiveMax	特定の時点でクエリメモリをアクティブに使用できるタスクの最大数。

フィールド	説明
QueryMemActiveEst	安定した状態のデータベースサーバで、クエリメモリをアクティブに使用するタスクの推定平均数。
isolation_level	文の独立性レベル。文の独立性レベルは、同じトランザクション内の他の文と異なる場合があります。また、特定のベーステーブルに対して FROM 句のヒントを使用して優先される可能性があります。
optimization_goal	クエリ処理の最適化の対象を、最初のローを迅速に返すこと、または完全な結果セットを返すコストを最小限に抑えることのどちらかに指定します。
optimization_level	クエリオプティマイザがアクセスプランの検索に費やす作業量を制御します。
optimization_workload	optimization_workload 設定値 (**Mixed** または **OLAP**)。
max_query_tasks	単一のクエリの並列実行プランで使用される可能性があるタスクの最大数。
user_estimates	クエリテキストの各述部で指定されたユーザ推定を尊重するか無視するかを制御します。
予測されるタスク	クエリ内並列処理で使用可能な予測されるタスクの数。
タスクの最大数	クエリ内並列処理で使用可能なタスクの最大数。
最適化時間の構築	最適化内部の構築に費やされる時間。
最適化前の時間	最適化前に費やされる時間。
最適化時間	最適化に費やされる時間。
非最適化時間	非最適化フェーズに費やされる時間。
最適化後の時間	最適化後に費やされる時間。
最終プラン構築時間	最終プランの構築に費やされる時間。
予測最大コスト	この最適化の予測最大コスト。

フィールド	説明
ジョイン列挙アルゴリズム	ジョイン列挙で使用するアルゴリズム。可能な値は、次のとおりです。 ● ブッシーツリー 1 ● ブッシーツリー 2 ● 最適化前によるブッシーツリー ● 削除によるブッシーツリー ● 並列ブッシーツリー ● 左が深いツリー ● ブッシーツリー 3 ● メモ化による左が深いツリー
最適化前の数	最適化前ジョイン列挙アルゴリズムによるブッシーツリーで有効です。
最適化前で使用するページ	最適化前フェーズで使用されたメモリページ数。
ジョイン列挙中に使用されたページ	ジョイン列挙中に使用されたメモリページ数。
算出された最適化前の数	算出された最適化前フェーズで使用されたメモリページ数。
ジョイン列挙中に使用されたメモリページ	ジョイン列挙フェーズ中に使用されたメモリページ数。
ジョイン列挙によって使用された追加ページ	削除によるジョイン列挙によって使用された追加メモリページ数。
ジョイン列挙ランタイム	ジョイン列挙フェーズ中に費やされた時間。
初期化ランタイム	初期化フェーズ中に費やされた時間。
予測最大コストランタイム	予測最大コストフェーズ中に費やされた時間。
ランタイムの削除	削除フェーズ中に費やされた時間。
ランタイムの見積もり	見積もりフェーズ中に費やされた時間。
物理プラン生成ランタイム	物理プラン生成フェーズ中に費やされた時間。
並列ランタイム	並列フェーズ中に費やされた時間。

フィールド	説明
左が深いツリーの生成ランタイム	左が深いツリーの生成フェーズ中に費やされた時間。
論理プラン生成ランタイム	論理プラン生成フェーズ中に費やされた時間。
最適化前ランタイム	最適化前フェーズ中に費やされた時間。
クリーンアップランタイム	クリーンアップフェーズ中に費やされた時間。
パーティションランタイム	パーティションフェーズ中に費やされた時間。
ロギングランタイム	ロギングフェーズ中に費やされた時間。
その他のランタイム	その他のフェーズ中に費やされた時間。
列挙された論理ジョインの数	列挙された論理ジョインの数。
見積もられたジョインの数	見積もられたジョインの数。
削除されたジョイン	ローカルコストとグローバルコストに基づいて削除されたジョインの数。
メモ化テーブルでの操作	メモ化テーブルでの操作 (挿入、置換、検索が行われた)。

[ノード統計] フィールドの説明

次に、グラフィカルなプランの **[ノード統計]** セクションに表示されるフィールドについて説明します。

フィールド	説明
CacheHits	この演算子によるキャッシュ読み込み要求で、バッファプールで条件が満たされ、ディスク読み込み操作が不要になった要求の合計数。
CacheRead	この演算子がデータベースファイルのページ (通常、テーブルページやインデックスページなど) を読み込もうとした合計回数。
CPUTime	このノードが表す処理アルゴリズムによって発生した CPU 時間。

フィールド	説明
DiskRead	このノードの処理の結果として、ディスクから読み込まれた累積ページ数。
DiskReadTime	このノードの処理に必要なデータベースページをディスクから読み込むために要した累積時間。
DiskWrite	このノードの処理の結果として、ディスクに書き込まれた累積ページ数。
DiskWriteTime	このノードのアルゴリズムの処理に必要なデータベースページをディスクに書き込むために要した累積時間。
FirstRowRunTime	**FirstRowRunTime** 値は、このノードの中間結果の最初のローを作成するために要した時間です。
Invocations	ノードが結果を計算するために呼び出された回数。親ノードに結果を返します。ほとんどのノードは 1 回だけ呼び出されます。ただし、スキャンノードの親がネストループジョインで、ノードが複数回実行される可能性がある場合、各呼び出し後に異なるローセットを返すことがあります。
PercentTotalCost	特定のノード内で結果の計算に費やす **RunTime**。文の合計 RunTime に対する割合で表します。
QueryMemMaxUseful	この特定の演算子での使用が予想されるクエリメモリの推定容量。**Actual** 統計値で報告されるクエリメモリの実際の使用量と大幅に異なる場合、クエリオプティマイザによる結果セットのサイズの推定に潜在的な問題がある可能性があります。この推定のエラーの原因は、述部選択性推定が不正確であるか存在しないことが考えられます。

フィールド	説明
RowsReturned	要求の処理の結果として親ノードに返されたローの数。**RowsReturned** は、このノードで表されるオブジェクト (派生オブジェクトの場合がある) 内のローの数と同一であることがよくありますが、必ず同じとはかぎりません。ベーステーブルスキャンを表すリーフノードを考慮してください。**RowsReturned** 値がテーブル内のローの数よりも小さい、または大きい場合があります。最終結果の計算で、親ノードがテーブルのすべてのローを要求できない場合、**RowsReturned** 値は小さくなります。GROUP BY GROUPING SETS クエリなど、親のハッシュ GROUP BY GROUPING SETS ノードが異なるグループを計算するために入力の受け渡しを複数回要求する場合、**RowsReturned** は大きくなります。 返された推定ローと返された実際の数とが大幅に食い違っている場合、オプティマイザが不正確な選択性情報に基づいて操作していることを示している可能性があります。

フィールド	説明
RunTime	実際の時間の計測値。入出力の待機、ローのロック、テーブルのロック、内部サーバの同時制御メカニズム、実際の処理実行時間が含まれます。**RunTime** の解釈は、表示される統計セクションによって異なります。[ノード統計] の場合、**RunTime** はこのノードだけを実行している間に、ノードの対応する演算子が費やした累積時間です。[ノード統計] セクションには、この統計の推定値と実際値の両方が表示されます。 テーブルスキャンまたはインデックススキャンで、ノードの **RunTime** が予想よりも大きい場合、さらに分析を進めると問題の特定に役立つ場合があります。クエリで共有リソースに対する競合が発生し、その結果ブロックした可能性があります。sa_locks() システムプロシージャを使用して、ブロックされた接続をモニタできます。別の例として、ディスクのデータベースページレイアウトが最適化されていない場合や、テーブルが内部ページの断片化による影響を受けている場合があります。REORGANIZE TABLE 文を実行するとパフォーマンスを向上できることがあります。sa_table_fragmentation() システムプロシージャと sa_index_density() システムプロシージャを使用して、テーブルまたはインデックスが断片化されているかどうかを判断できます。

プランに使用される一般的な統計

次の統計は実際の測定値です。

統計情報	説明
CacheHits	キャッシュ内の検索において一致したデータベースページの数を返します。
CacheRead	キャッシュ内で検索されたデータベースページの数を返します。
CacheReadTable	キャッシュから読み込まれたテーブルページの数を返します。

統計情報	説明
CacheReadIndLeaf	キャッシュから読み込まれたインデックスリーフページの数を返します。
CacheReadIndInt	キャッシュから読み込まれたインデックス内部ノードページの数を返します。
DiskRead	ディスクから読み込まれたページの数を返します。
DiskReadTable	ディスクから読み込まれたテーブルページの数を返します。
DiskReadIndLeaf	ディスクから読み込まれたインデックスリーフページの数を返します。
DiskReadIndInt	ディスクから読み込まれたインデックス内部ノードページの数を返します。
DiskWrite	ディスクに書き込まれた修正ページの数を返します。
IndAdd	インデックスに追加されたエントリの数を返します。
IndLookup	インデックス内で検索されたエントリの数を返します。
FullCompare	インデックスのハッシュ値を超えて実行された比較の回数を返します。

プランに使用される一般的な推定

統計情報	説明
EstRowCount	呼び出されるたびにノードが返すローの推定数。
AvgRowCount	各呼び出しで返される平均ロー数。これは推定値ではなく、RowsReturned / Invocations として計算されます。この値が EstRowCount とまったく異なる場合は、選択性推定が不十分な場合があります。
EstRunTime	推定実行所要時間 (EstDiskReadTime、EstDiskWriteTime、EstCpuTime の合計)。

統計情報	説明
AvgRunTime	平均実行所要時間 (測定値)。
EstDiskReads	ディスクからの読み込み操作の推定回数。
AvgDiskReads	ディスクからの読み込み操作の平均回数 (測定値)。
EstDiskWrites	ディスクへの書き込み操作の推定回数。
AvgDiskWrites	ディスクへの書き込み操作の平均回数 (測定値)。
EstDiskReadTime	ディスクからローを読み込むときの推定所要時間。
EstDiskWriteTime	ディスクにローを書き込むときの推定所要時間。
EstCpuTime	プロセッサの推定実行所要時間。

SELECT、INSERT、UPDATE、DELETE に関連するプランの項目

項目	説明
Optimization Goal	クエリ処理の最適化の対象を、最初のローを迅速に返すこと、または完全な結果セットを返すコストを最小限に抑えることのどちらかに指定します。
Optimization workload	クエリ処理において、更新と読み込みを組み合わせた負荷に対して最適化するか、または大部分が読み込みベースの負荷に対して最適化するかを決定します。
ANSI update constraints	更新が許される範囲を制御します (オプションは、Off、Cursors、Strict)。
Optimization level	予約済み。
Select list	クエリによって選択される式のリスト。

項目	説明
Materialized views	オプティマイザによって検討されるマテリアライズドビューのリスト。リスト内の各エントリは view-name [view-matching-outcome] [table-list] という形式の組です。ここで *view-matching-outcome* はマテリアライズドビューの使用法を示します。この値が COSTED の場合、ビューは列挙時に使用されています。*table-list* は、このビューで置き換えられる可能性のあったクエリテーブルのリストです。 *view-matching-outcome* の値は、次のとおりです。 ● ベーステーブルが一致しません ● 権限が一致しません ● 述部が一致しません ● Select リストが一致しません ● 見積り済みです ● 失効が一致しません ● スナップショットの失効が一致しません ● オプティマイザでは使用できません ● オプティマイザでは内部で使用できません ● 定義を構築できません ● アクセスできません ● 無効になっています ● オプションが一致しません ● ビューマッチングがしきい値に到達しました ● ビューは使用されています

ロックに関連するプランの項目

項目	説明
Locked tables	すべてのロックテーブルとその独立性レベルのリスト。

スキャンに関連するプランの項目

項目	説明
Table name	テーブルの実際の名前。

項目	説明
Correlation name	テーブルのエイリアス。
Estimated rows	テーブルの推定ロー数。
Estimated pages	テーブルの推定ページ数。
Estimated row size	テーブルの推定ローサイズ。
Page maps	複数ページの読み込みにページマップが使用される場合は YES。

インデックススキャンに関連するプランの項目

項目	説明
Selectivity	範囲バウンドと一致する推定ロー数。
Index name	インデックスの名前。
Key type	PRIMARY KEY、FOREIGN KEY、CONSTRAINT (一意性制約)、UNIQUE (ユニークインデックス) のいずれか。インデックスがユニークでないセカンダリインデックスの場合、キータイプは表示されません。
Depth	インデックスの高さ。
Estimated leaf pages	リーフページの推定数。
Sequential Transitions	インデックスがどれだけクラスタ化されているかを示す、各物理インデックスの統計情報。
Random Transitions	インデックスがどれだけクラスタ化されているかを示す、各物理インデックスの統計情報。
Key Values	インデックス内のユニークなエントリの数。
Cardinality	推定ロー数と異なる場合の、インデックスのカーディナリティ。バージョン 6.0.0 以前の SQL Anywhere データベースにのみ適用されます。
Direction	FORWARD または BACKWARD。
Range bounds	範囲バウンドは、リスト (col_name=value) または col_name IN [low, high] として表示されます。

項目	説明
Primary Key Table	外部キーインデックススキャンのプライマリキーテーブル名。
Primary Key Table Estimated Rows	外部キーインデックススキャンのプライマリキーテーブル内のロー数。
Primary Key Column	外部キーインデックススキャンのプライマリキーカラム名。

ジョイン、フィルタ、事前フィルタに関連するプランの項目

項目	説明
Predicate	このノードで評価される探索条件、選択性推定、測定値。

ハッシュフィルタに関連するプランの項目

項目	説明
Build values	入力内の重複しない値の推定数。
Probe values	述部をチェックする場合の、入力内の重複しない値の推定数。
Bits	ハッシュマップを構築するために選択されたビット数。
Pages	ハッシュマップを格納するために必要なページ数。

UNION に関連するプランの項目

項目	説明
Union List	UNION 文が対象とするカラム。

GROUP BY に関連するプランの項目

項目	説明
Aggregates	すべての集合関数。
Group-by list	GROUP BY 句に指定されているすべてのカラム。

DISTINCT に関連するプランの項目

項目	説明
Distinct list	DISTINCT 句に指定されているすべてのカラム。

IN リストに関連するプランの項目

項目	説明
In List	指定したセットのすべての式。
Expression SQL	リストと比較される式。

SORT に関連するプランの項目

項目	説明
Order-by	ソート基準となるすべての式のリスト。

ロー制限に関連するプランの項目

項目	説明
Row limit count	FIRST または TOP *n* で指定された、返されるローの最大数。

参照

- 「オプティマイザの仕組み」321 ページ
- 「キャッシュとメモリガバナー」213 ページ
- 「グラフィカルなプラン内の選択性情報」339 ページ
- 「isolation_level オプション」『SQL Anywhere サーバ データベース管理』
- 「optimization_goal オプション」『SQL Anywhere サーバ データベース管理』
- 「optimization_level オプション」『SQL Anywhere サーバ データベース管理』
- 「optimization_workload オプション」『SQL Anywhere サーバ データベース管理』
- 「max_query_tasks オプション」『SQL Anywhere サーバ データベース管理』
- 「user_estimates オプション」『SQL Anywhere サーバ データベース管理』
- 「ansi_update_constraints オプション」『SQL Anywhere サーバ データベース管理』
- 「マテリアライズドビューの制限」58 ページ

高度：クエリ実行時の並列処理

SQL Anywhere では、クエリ実行に対して、クエリ間とクエリ内の 2 種類の並列処理がサポートされています。クエリ間並列処理とは、異なる要求を別個の CPU 上で同時に実行することです。各要求 (タスク) は単一のスレッド、単一のプロセッサで実行されます。

クエリ内並列処理とは、単一の要求を複数の CPU で同時処理することです。クエリが分割されて各部がマルチプロセッサのハードウェアで並列処理されます。これらの各部は交換アルゴリズムで処理されます。

クエリ内並列処理では、同時に実行されるクエリ数が使用可能なプロセッサ数より少ないことが多い場合に、負荷が分散されます。並列処理の程度は、max_query_tasks オプションを設定して制御します。

オプティマイザは、並列処理の追加コスト (ローの追加コピー、作業量の調整のための追加コスト) を推定し、パフォーマンスの向上が期待できる場合のみ並列プランを選択します。

クエリ内並列処理は、priority オプションが background に設定されている接続には使用されません。

現在要求を処理しているサーバスレッド数 (ActiveReq サーバプロパティ) が、データベースサーバが使用ライセンスを持つコンピュータの CPU コア数を最近超えた場合は、クエリ内並列処理は使用されません。正確な時間はサーバによって決められ、通常は数秒です。

並列実行

クエリで並列実行が利用されるかどうかは、次の要因によって決まります。

- 最適化時にシステムで使用可能なリソース (メモリ、キャッシュ内のデータなど)
- コンピュータの論理プロセッサ数
- データベースの格納に使用されるディスクデバイス数、プロセッサとコンピュータの I/O アーキテクチャの速度に対する相対速度
- 要求に必要な特定の代数演算子。SQL Anywhere では、次の 5 つの代数演算子がサポートされています。これらの演算子は並列実行できます。
  - 並列逐次スキャン (テーブルスキャン)
  - 並列インデックススキャン
  - 並列ハッシュジョイン、ハッシュセミジョインと非セミジョインの並列バージョン
  - 並列ネストループジョイン、ネストループセミジョインとネストループ非セミジョインの並列バージョン
  - 並列ハッシュフィルタ
  - 並列ハッシュ Group By

サポートされていない演算子を使用しているクエリでも並列実行できますが、プランで、サポートされている演算子がサポートされていない演算子の下にある必要があります (Interactive SQL での表示)。サポートされていない演算子のほとんどが上の方にあるクエリは、並列処理される確率が高くなります。たとえば、ソート演算子は並列処理できませんが、最も外側のブロックで ORDER BY を使用するクエリは、ソートをプランの一番上に置き、すべての並列演算子をその下に置くことで並列処理できます。これに対して、派生テーブルで TOP *n* と ORDER BY を使用するクエリは、ソートがプランの一番上にないので、並列処理が使用される確率が低くなります。

SQL Anywhere では、DB 領域がデフォルトで単一プラッターのディスクサブシステムにあると想定されます。このような環境では、クエリの並列実行を行う効果があっても、オプティマイザ

の単一デバイス用の I/O コストモデルによって、テーブルのデータが完全にキャッシュ内に収まっていないかぎり、オプティマイザで並列テーブルまたはインデックススキャンを選択するのが困難になります。ただし、ALTER DATABASE CALIBRATE PARALLEL READ 文を使用してディスクのサブシステムを調整することによって、オプティマイザで並列実行の効果をより正確に計算できます。ディスクのサブシステムに複数のプラッターが搭載されているときには、オプティマイザでは並行実行を伴う実行プランが選択される傾向があります。

クエリ内並列処理がアクセスプランに使用される場合、各サブツリーの並列処理の結果をマージ (UNION) する交換演算子がプランに含まれます。交換演算子の下位のサブツリー数が、並列処理の程度です。各サブツリー、またはアクセスプランのコンポーネントは、データベースサーバのタスクです。データベースサーバのカーネルでは、実行スレッド (ファイバー) が使用可能かどうかに応じて、個々の SQL 要求と同じようにこれらのタスクがスケジュールされます。このアーキテクチャでは、すべてのアクセスプランの並列処理の大部分が自動的に調整されます。並列実行タスクの処理はサーバカーネルの許可に従ってスレッド (ファイバー) にスケジュールされ、プランのコンポーネントが均等に実行されます。

参照

- 「-gn データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「max_query_tasks オプション」『SQL Anywhere サーバ データベース管理』
- 「SQL Anywhere のスレッド化」『SQL Anywhere サーバ データベース管理』
- 「-gtc データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「マルチプログラミングレベルのデータベースサーバ設定」『SQL Anywhere サーバ データベース管理』
- 「高度：クエリ実行プラン」328 ページ
- 「ALTER DATABASE 文」『SQL Anywhere サーバ SQL リファレンス』
- ホワイトペーパー『**Query Processing Based on SQL Anywhere 12.0.1 Architecture**』(http://www.sybase.com/detail?id=1096047)
- 「priority オプション」『SQL Anywhere サーバ データベース管理』
- 「データベースサーバプロパティの一覧」『SQL Anywhere サーバ データベース管理』

クエリの並列処理

クエリは、クエリが処理するロー数が、返されるロー数よりも多い場合に並列処理が使用される確率が高くなります。この場合、処理されるロー数には、スキャンされるすべてのローのサイズとすべての中間結果のサイズが含まれます。インデックスを使用してテーブルのほとんどがスキップされるので、スキャンされないローは含まれません。理想的なケースは、大きなテーブルに対する単一ローの GROUP BY であり、この場合、多数のローがスキャンされ、1 つのローだけが返されます。グループのサイズが大きい場合は、複数グループのクエリも並列処理の確率が高くなります。多数のローを削除する述部またはジョイン条件も高い確率で並列処理されます。

最適化または実行のときにクエリに並列処理が使用されない場合のリストを次に示します。

- サーバのコンピュータに複数のプロセッサがない。

- サーバのコンピュータに、複数のプロセッサを使用するためのライセンスがない。これは、NumLogicalProcessorsUsed サーバプロパティで確認できます。ただし、ハイパースレッドのプロセッサはクエリ内並列処理対象として数えられないので、コンピュータでハイパースレッドを使用している場合は、NumLogicalProcessorsUsed の値を 2 で割ります。

●max_query_tasks オプションが 1 に設定されている。

●priority オプションが background に設定されている。

●クエリを含む文が SELECT 文ではない。

●最近、ActiveReq の値が NumLogicalProcessorsUsed の値以上になったことがある (コンピュータでハイパースレッドを使用している場合はプロセッサ数を 2 で割る)。

●使用可能なタスク数が不十分である。

参照

- 「高度：クエリ実行時の並列処理」364 ページ
- 「SQL Anywhere のスレッド化」『SQL Anywhere サーバ データベース管理』
- 「max_query_tasks オプション」『SQL Anywhere サーバ データベース管理』
- 「priority オプション」『SQL Anywhere サーバ データベース管理』
- max_query_tasks、priority、NumLogicalProcessorsUsed、ActiveReq プロパティ：「データベースサーバプロパティの一覧」『SQL Anywhere サーバ データベース管理』
- 「CREATE DATABASE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

全文検索

全文検索とは

全文検索は、データベースを検索するためのより高度な方法です。全文検索を行うと、ローをスキャンする必要や単語が格納されているカラムを知る必要がなく、テーブル内の特定の単語のすべてのインスタンスを簡単に検索できます。全文検索は、「テキストインデックス」を使用することで機能します。テキストインデックスには、テキストインデックスを作成した、カラム内のすべての単語の位置情報が格納されます。特定の値を含むローを検索する場合、テキストインデックスを使用した方が、通常のインデックスを使用するよりも処理が速い可能性があります。

SQL Anywhere の全文検索機能は、単語単位であり、パターン単位でないという点で、LIKE、REGEXP、SIMILAR TO などの述部を使用した検索とは異なります。

全文検索の文字列の比較では、データベースのすべての通常の照合設定が使われます。たとえば、データベースが大文字と小文字を区別しないように設定されている場合、全文検索でも大文字と小文字が区別されません。

指定がないかぎり、全文検索では、SQL Anywhere でサポートされているすべての国際化機能が使われます。

中国語、日本語、韓国語 (CJK) データを含むデータベースで全文検索を実行する方法については、http://www.sybase.com/detail?id=1061814 にあるホワイトペーパー『**Performing Full Text Searches on Chinese, Japanese, and Korean Data in SQL Anywhere 11**』を参照してください。

全文検索を実行する 2 つの方法

全文クエリを実行するには、SELECT 文の FROM 句で CONTAINS 句を使用するか、または WHERE 句で CONTAINS 探索条件 (述部) を使用します。どちらも同じローが返されますが、FROM 句で CONTAINS 句を使用した場合は、一致するローのスコアも返されます。

次の例は、クエリでの CONTAINS 句と CONTAINS 探索条件の使用方法を示します。これらの例では、サンプルデータベースで提供されている MarketingInformation.Description テキストインデックスが使用されています。

```
SELECT *
  FROM MarketingInformation CONTAINS ( Description, 'cotton' );

SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( Description, 'cotton' );
```

全文検索を使用する前の考慮事項

通常のインデックスではなく全文インデックスを使用するかどうかを決定するときには、次の考慮事項があります。

- CONTAINS 句または CONTAINS 探索条件ではエイリアスを使用できません。

- クエリで重複する相関名を使用する場合、CONTAINS (FROM CONTAINS()) は、相関名の最初のインスタンスでのみサポートされます。たとえば、2 番目の CONTAINS 述部に A が含まれているため、次の構文ではエラーが返されます。

```
SELECT *
FROM CONTAINS(A contains-query-string) JOIN B ON A.x = B.x,
    CONTAINS(A contains-query-string) JOIN C ON A.y = C.y;
```

外部の単語区切りライブラリと事前フィルタライブラリを使用する場合は、次の追加の考慮事項があります。

- **クエリと更新**　ライブラリを使用して作成されたテキストインデックスの更新、クエリ、または変更を必要とするすべての操作に対して、外部ライブラリが引き続き使用可能になっている必要があります。

- **アンロードと再ロード**　全文インデックスに関連付けられているデータのアンロード時と再ロード時に、外部ライブラリが使用可能になっている必要があります。

- **データベースのリカバリ**　データベースをリカバリするには、外部ライブラリが使用可能である必要があります。これは、トランザクションログ内に、最終チェックポイント以降に外部ライブラリが関与した操作が存在する場合は、データベースをリカバリできないためです。

参照

- 「CONTAINS 探索条件」『SQL Anywhere サーバ SQL リファレンス』
- 「高度：外部単語区切りライブラリと事前フィルタライブラリ」442 ページ
- 「テキストインデックスの概念と参照」408 ページ
- 「国際言語と文字セット」『SQL Anywhere サーバ データベース管理』
- 「照合」『SQL Anywhere サーバ データベース管理』

テキスト設定オブジェクトの作成 (Sybase Central の場合)

テキスト設定オブジェクトは、テキストインデックスを構築したり更新するときに使用します。Sybase Central でテキスト設定オブジェクトを作成するには、**テキスト設定オブジェクト作成ウィザード**を使用します。

前提条件

所有するオブジェクトのテキスト設定を作成するには、CREATE TEXT CONFIGURATION システム権限が必要です。

他のユーザが所有するオブジェクトのテキスト設定を作成するには、CREATE ANY TEXT CONFIGURATION または CREATE ANY OBJECT システム権限が必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[テキスト設定オブジェクト]** を右クリックし、**[新規]** » **[テキスト設定オブジェクト]** をクリックします。
3. **テキスト設定オブジェクト作成ウィザード**の指示に従います。
4. **[テキスト設定オブジェクト]** ウィンドウ枠をクリックします。

結果

[テキスト設定オブジェクト] ウィンドウ枠に、新しいテキスト設定オブジェクトが表示されます。

参照

- 「CREATE TEXT CONFIGURATION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「テキスト設定オブジェクトの作成または変更時の指定項目」396 ページ
- 「テキスト設定オブジェクトの例」402 ページ
- 「データベースのテキスト設定オブジェクトの表示」370 ページ
- 「ALTER TEXT CONFIGURATION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「デフォルトのテキスト設定オブジェクト」403 ページ

テキスト設定オブジェクトの変更

Sybase Central では、管理者は、単語区切りタイプ、ストップリスト、オプション設定などのテキスト設定オブジェクトのプロパティを変更できます。

前提条件

CREATE EXTERNAL REFERENCE 権限が必要です。

内容と備考

テキストインデックスは、作成時に使用したテキスト設定オブジェクトに依存しています。このため、依存するテキストインデックスをトランケートまたは削除してください。また、テキスト

設定オブジェクトの保存された日付または時間形式のオプションを変更する場合は、希望する設定値を設定したオプションを指定してデータベースに接続してください。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[テキスト設定オブジェクト]** をクリックします。
3. テキスト設定オブジェクトを右クリックし、**[プロパティ]** をクリックします。
4. テキスト設定オブジェクトのプロパティを編集し、**[OK]**をクリックします。

結果

テキスト設定オブジェクトが変更されます。

参照

- 「テキスト設定オブジェクトの作成または変更時の指定項目」396 ページ
- 「テキスト設定オブジェクトの例」402 ページ
- 「データベースのテキスト設定オブジェクトの表示」370 ページ
- 「CREATE TEXT CONFIGURATION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TEXT CONFIGURATION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「デフォルトのテキスト設定オブジェクト」403 ページ

データベースのテキスト設定オブジェクトの表示

Sybase Central でテキスト設定オブジェクトの設定や他のプロパティを表示します。

前提条件

テキスト設定オブジェクトの所有者であるか、ALTER ANY TEXT CONFIGURATION または ALTER ANY OBJECT システム権限を持っている必要があります。

- ALTER ANY TEXT CONFIGURATION
- ALTER ANY OBJECT

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[テキスト設定オブジェクト]** をクリックします。
3. テキスト設定オブジェクトを右クリックし、**[プロパティ]** をクリックします。

結果

テキスト設定オブジェクトの設定が、表示されます。

参照

- 「テキスト設定オブジェクトの作成または変更時の指定項目」396 ページ
- 「SYSTEXTCONFIG システムビュー」『SQL Anywhere サーバ SQL リファレンス』

テキストインデックスの作成

任意のタイプのカラムのテキストインデックスを作成します。VARCHAR タイプまたは NVARCHAR タイプではないカラムは、インデックスの作成時に文字列に変換されます。

前提条件

テーブルのテキストインデックスを作成するには、テーブルの所有者であるか、次のいずれかの権限を持っている必要があります。

- CREATE ANY INDEX システム権限
- CREATE ANY OBJECT システム権限
- そのテーブルに対する REFERENCES 権限に加え、COMMENT ANY OBJECT システム権限、ALTER ANY INDEX システム権限、または ALTER ANY OBJECT システム権限

マテリアライズドビューのテキストインデックスを作成するには、マテリアライズドビューの所有者であるか、次のいずれかの権限を持っている必要があります。

- CREATE ANY INDEX システム権限
- CREATE ANY OBJECT システム権限

文またはトランザクションのスナップショットを使用する WITH HOLD 句で開かれているカーソルがある場合、テキストインデックスは作成できません。

通常のビューやテンポラリテーブルに対してテキストインデックスを作成することはできません。無効になっているマテリアライズドビューのテキストインデックスは作成できません。

内容と備考

テキストインデックスはディスク領域を消費し、再表示を必要とします。作成する場合は、クエリをサポートするために必要となるカラムのみを対象にしてください。

同じカラムを参照している複数のテキストインデックスを作成すると、予期しない結果が生じる可能性があります。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[テキストインデックス]** タブをクリックします。
3. **[ファイル]** » **[新規]** » **[テキスト インデックス]** をクリックします。
4. **インデックス作成ウィザード**の指示に従います。

[テキストインデックス] タブに新しいテキストインデックスが表示されます。また、**[テキストインデックス]** フォルダにも表示されます。

結果

テキストインデックスが作成されます。即時再表示のテキストインデックスを作成した場合は、データが自動的に設定されます。その他の再表示タイプの場合は、テキストインデックスを手動で再表示する必要があります。

参照

- 「CREATE TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「テキストインデックスの再表示」372 ページ
- 「テキストインデックスの再表示タイプ」409 ページ
- 「データ型変換」『SQL Anywhere サーバ SQL リファレンス』
- 「REFRESH TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』

テキストインデックスの再表示

テキストインデックスを再表示し、テキストインデックスのデータを更新します。テキストインデックスを再表示すると、基本となるテーブルで実施されたデータの変更を反映します。

前提条件

テキストインデックスを再表示するには、基本となるテーブルの所有者であるか、次のいずれかの権限を持っている必要があります。

- CREATE ANY INDEX システム権限
- CREATE ANY OBJECT システム権限
- ALTER ANY INDEX システム権限
- ALTER ANY OBJECT システム権限
- そのテーブルに対する REFERENCES 権限

再表示できるテキストインデックスは、AUTO REFRESH および MANUAL REFRESH として定義されているテキストインデックスだけです。IMMEDIATE として定義されているテキストインデックスは再表示できません。

内容と備考

マテリアライズドビューのテキストインデックスは、ビューが更新されたり再表示されると必ず再表示されます。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[テキストインデックス]** をクリックします。
3. テキストインデックスを右クリックし、**[データの再表示]** をクリックします。

4. 再表示の独立性レベルを選択し、**[OK]**をクリックします。

結果

テキストインデックスが再表示されます。

参照

- 「REFRESH TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「テキストインデックスの再表示タイプ」409 ページ

テキストインデックスの変更

テキストインデックスの次の特徴を変更できます。

- **再表示タイプ**　再表示タイプを AUTO REFRESH から MANUAL REFRESH に変更したり、その逆に変更することができます。再表示タイプを変更するには、ALTER TEXT INDEX 文の REFRESH 句を使用します。

  テキストインデックスを IMMEDIATE REFRESH に変更したり、IMMEDIATE REFRESH から変更することはできません。この変更を実行するには、テキストインデックスを削除して、再作成する必要があります。

- **名前**　テキストインデックスの名前を変更するには、ALTER TEXT INDEX 文の RENAME 句を使用します。

- **内容**　カラムリストを除き、インデックス付けの対象を制御する設定は、テキスト設定オブジェクトに格納されます。インデックス付けの対象を変更する場合は、テキストインデックスが参照するテキスト設定オブジェクトを変更します。依存するテキストインデックスをトランケートしてからテキスト設定オブジェクトを変更し、その後テキストインデックスを再表示する必要があります。即時再表示のテキストインデックスの場合、テキストインデックスを削除し、テキスト設定オブジェクトを変更した後にテキストインデックスを再作成する必要があります。

別のテキスト設定オブジェクトを参照するように、テキストインデックスを変更することはできません。テキストインデックスで別のテキスト設定オブジェクトを参照させるには、テキストインデックスを削除して、新しいテキスト設定オブジェクトを指定して再作成します。

参照

- 「テキストインデックスの再表示タイプ」409 ページ
- 「テキスト設定オブジェクトの概念と参照」395 ページ
- 「TRUNCATE TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REFRESH TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DROP TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_refresh_text_indexes システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「テキストインデックスの概念と参照」408 ページ

テキストインデックスの変更

テキストインデックスの名前を変更したり、その再表示タイプを変更できます。

前提条件

テーブルのテキストインデックスを変更するには、テーブルの所有者であるか、次のいずれかの権限を持っている必要があります。

- ALTER ANY INDEX システム権限
- ALTER ANY OBJECT システム権限
- そのテーブルに対する REFERENCES 権限に加え、COMMENT ANY OBJECT システム権限、CREATE ANY INDEX システム権限、または CREATE ANY OBJECT システム権限

マテリアライズドビューのテキストインデックスを変更するには、マテリアライズドビューの所有者であるか、次のいずれかの権限を持っている必要があります。

- ALTER ANY INDEX システム権限
- ALTER ANY OBJECT システム権限

別のテキスト設定オブジェクトを参照するように、テキストインデックスを変更することはできません。テキストインデックスで別のテキスト設定オブジェクトを参照させるには、テキストインデックスを削除して、新しいテキスト設定オブジェクトを指定して再作成します。

テキストインデックスを IMMEDIATE REFRESH に変更したり、IMMEDIATE REFRESH から変更することはできません。この変更を実行するには、テキストインデックスを削除して、再作成する必要があります。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[テキストインデックス]** をクリックします。
3. テキストインデックスを右クリックし、**[プロパティ]** をクリックします。
4. テキストインデックスのプロパティを編集します。

   テキストインデックスの名前は **[一般]** タブで変更できます。
5. **[OK]** をクリックします。

結果

テキストインデックスが変更されます。

参照

- 「テキストインデックスの再表示タイプ」409 ページ
- 「テキスト設定オブジェクトの概念と参照」395 ページ
- 「TRUNCATE TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REFRESH TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_refresh_text_indexes システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「テキストインデックスの概念と参照」408 ページ

テキストインデックスの単語と設定の表示 (Sybase Central の場合)

Sybase Central でテキストインデックスの単語と設定を表示します。

前提条件

テキストインデックスの完全な情報を表示するには、テーブルまたはマテリアライズドビューの所有者であるか、次のいずれかのシステム権限を持っている必要があります。

- CREATE ANY INDEX
- CREATE ANY OBJECT
- ALTER ANY INDEX
- ALTER ANY OBJECT
- DROP ANY INDEX
- DROP ANY OBJECT
- MANAGE ANY STATISTICS

[ボキャブラリ] タブの情報を表示するには、次のいずれかの権限も必要です。

- テキストインデックスが作成されたテーブルまたはマテリアライズドビューに対する SELECT 権限
- SELECT ANY TABLE システム権限

テキストインデックスを初期化する必要があります。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[テキストインデックス]** をクリックします。
3. テキストインデックス内の単語を表示するには、左ウィンドウ枠でテキストインデックスをダブルクリックし、右ウィンドウ枠で **[ボキャブラリ]** タブをクリックします。
4. インデックスが参照するテキスト設定オブジェクトや再表示タイプなど、テキストインデックスの設定を表示するには、テキストインデックスを右クリックし、**[プロパティ]** をクリックします。

結果

テキストインデックスの単語と設定が表示されます。

参照

- 「sa_text_index_stats システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSMVOPTION システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSMVOPTIONNAME システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSTAB システムビュー」『SQL Anywhere サーバ SQL リファレンス』

テキストインデックスの単語と設定の表示 (SQL の場合)

Interactive SQL でテキストインデックスの単語と設定を表示します。

前提条件

テキストインデックスに関する設定や統計の情報を表示するには、次のいずれかのシステム権限が必要です。

- MANAGE ANY STATISTICS
- CREATE ANY INDEX
- ALTER ANY INDEX
- DROP ANY INDEX
- CREATE ANY OBJECT
- ALTER ANY OBJECT
- DROP ANY OBJECT

テキストインデックスの単語を表示するには、次のいずれかの権限も必要です。

- テーブルまたはマテリアライズドビューに対する SELECT 権限
- SELECT ANY TABLE システム権限

テキストインデックスを初期化する必要があります。

♦ タスク

1. データベースに接続します。
2. sa_text_index_stats システムプロシージャを実行して、テキストインデックスに関する統計情報を表示します。

   ```
   CALL sa_text_index_stats( );
   ```

3. sa_text_index_vocab システムプロシージャを実行して、テキストインデックスの単語を表示します。

   ```
   CALL sa_text_index_vocab( );
   ```

結果

テキストインデックスの統計情報や単語が表示されます。

次の手順

テキストインデックスが作成されると、現在のデータベースオプションはテキストインデックスとともに格納されます。テキストインデックスの作成中に使用されたオプション設定を取得するには、次の文を実行します。

```
SELECT b.object_id, b.table_name, a.option_id, c.option_name, a.option_value
FROM SYSMVOPTION a, SYSTAB b, SYSMVOPTIONNAME c
WHERE a.view_object_id=b.object_id
AND b.table_type=5;
```

SYSTAB ビュー内で table_type が 5 のものは、テキストインデックスです。

参照

- 「sa_text_index_stats システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSMVOPTION システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSMVOPTIONNAME システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSTAB システムビュー」『SQL Anywhere サーバ SQL リファレンス』

全文検索のタイプ

全文検索を使用して、「単語」、「フレーズ」 (単語のシーケンス)、または「プレフィクス」を検索できます。複数の単語、フレーズ、またはプレフィクスをブール式に組み合わせることもできます。または、その式が近接検索で互いに近くに出現する必要があります。

全文検索を実行するには、SELECT 文の WHERE 句または FROM 句で CONTAINS 句を使用します。IF 探索条件の一部として全文検索を実行することもできます (SELECT IF CONTAINS... など)。

次の各項では、SQL Anywhere で使用できるさまざまな種類の全文検索を使用する方法を示します。

単語とフレーズの検索

単語のリストの全文検索を実行する場合は、フレーズ内の単語でないかぎり、単語の順序は重要ではありません。単語がフレーズ内にある場合、データベースサーバは、出現する順序と相対位置が指定されたものと厳密に一致する単語を検索します。

単語検索やフレーズ検索を実行する際、単語長の設定を超過するため、またはストップリストに含まれているために単語がクエリから削除された場合、予期しない数のローが返される可能性があります。これは、単語をクエリから削除することは探索条件を変更することと同じであるためです。たとえば、'"grown cotton"' というフレーズを検索し、grown がストップリストに入っている場合、cotton を含むインデックス付けされたローがすべて返されます。

CONTAINS 句の文法でキーワードと見なされる単語は、フレーズに含まれている場合のみ検索できます。

単語検索

サンプルデータベースでは、MarketingInformation テーブルの Description カラムに MarketingTextIndex というテキストインデックスが作成されています。次の文は、MarketingInformation テーブルの Description カラムを問い合わせて、Description カラムの値に **cotton** という単語が含まれるローを返します。

```
SELECT ID, Description
  FROM MarketingInformation
  WHERE CONTAINS ( Description, 'cotton' );
```

ID	Description
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown「cotton」construction. Shields against sun and precipitation.cotton Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown「cotton」hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% 「cotton」/20% polyester blend makes it easy to keep them clean. </body></html>

ID	Description
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying 「cotton」 shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>

次の例は、MarketingInformation テーブルを問い合わせて、各ローについて、Description カラムの値に **cotton** という単語が含まれるかどうかを示す 1 つの値を返します。

```
SELECT ID, IF CONTAINS ( Description, 'cotton' )
    THEN 1
    ELSE 0
    ENDIF AS Results
  FROM MarketingInformation;
```

ID	Results
901	0
902	0
903	0
904	0
905	0
906	1
907	0
908	1
909	1
910	1

次の例は、MarketingInformation テーブルを問い合わせて、Description カラムに **cotton** という単語が含まれる項目を検索し、一致ごとのスコアを表示します。

```
SELECT ID, ct.score, Description
  FROM MarketingInformation CONTAINS ( MarketingInformation.Description, 'cotton' ) as ct
  ORDER BY ct.score DESC;
```

ID	score	Description
908	0.9461597363521859	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"> <title>Sweatshirt</title></ head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </ body></html>
910	0.9244136988525732	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"> <title>Shorts</title></ head><body lang=EN-US> These quick-drying 「cotton」 shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></ html>

ID	score	Description
906	0.9134171046194403	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"> <title>Visor</title></ head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></ html>
909	0.8856420222728282	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"> <title>Sweatshirt</title></ head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% 「cotton」/20% polyester blend makes it easy to keep them clean. </body></ html>

フレーズ検索

フレーズの全文検索を実行する場合、フレーズを二重引用符で囲みます。指定の順序と相対位置の単語が含まれる場合、カラムは一致します。

単語がフレーズ内に含まれている場合を除き、AND や FUZZY などの CONTAINS キーワードを、検索する単語として指定することはできません (単一単語のフレーズは検索可能)。たとえば、NOT は CONTAINS キーワードですが、次の文は実行できます。

```
SELECT * FROM table-name CONTAINS ( Remarks, '"NOT"' );
```

フレーズの中で使用される特殊文字は、特殊文字として解釈されません (アスタリスクを除く)。

フレーズは、近接検索の引数としては使用できません。

次の文は、MarketingInformation テーブルの Description カラムを問い合わせて、"grown cotton" というフレーズを検索し、一致ごとのスコアを表示します。

```
SELECT ID, ct.score, Description
  FROM MarketingInformation CONTAINS ( MarketingInformation.Description, '"grown cotton"' ) as ct
  ORDER BY ct.score DESC;
```

ID	score	Description
908	1.6619019465461564	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"> <title>Sweatshirt</title></ head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </ body></html>
906	1.6043904700786786	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"> <title>Visor</title></ head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></ html>

参照

- 「全文検索結果のスコア」393 ページ
- 「プレフィクス検索」383 ページ
- 「CONTAINS 探索条件」『SQL Anywhere サーバ SQL リファレンス』

プレフィクス検索

全文検索では、単語の先頭部分を検索できる機能があります。これは「プレフィクス検索」と呼ばれます。プレフィクス検索を実行するには、検索するプレフィクスを指定した後に、アスタリスクを付けます。これは「プレフィクス単語」と呼ばれます。

CONTAINS 句のキーワードは、フレーズに含まれていないかぎり、プレフィクス検索では使用できません。

クエリ文字列に複数のプレフィクス単語 (フレーズ内のプレフィクス単語も含む) を指定することもできます (たとえば、'"shi* fab"' など)。

次の例は、MarketingInformation テーブルを問い合わせて、プレフィクス **shi** で始まる項目を検索します。

```
SELECT ID, ct.score, Description
  FROM MarketingInformation CONTAINS ( MarketingInformation.Description, 'shi*' ) AS ct
  ORDER BY ct.score DESC;
```

ID	score	Description
906	2.295363835537917	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"> <title>Visor</title></ head><body lang=EN-US> Lightweight 100% organically grown cotton construction. 「Shi」elds against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></ html>

ID	score	Description
901	1.6883275743936228	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee 「Shi」rt</title></head><body lang=EN-US> We've improved the design of this perennial favorite. A sleek and technical 「shi」rt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>
903	1.6336529491832605	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee 「Shi」rt</title></head><body lang=EN-US> A sporty, casual 「shi」rt made of recycled water bottles. It will serve you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>

ID	score	Description
902	1.6181703448678983	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><titl e>Tee 「Shi」rt</title></ head><body lang=EN-US> This simple, sleek, and lightweight technical 「shi」rt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin.</ span> </body></html>

テキストインデックスで shield という単語が出現する頻度は shirt よりも低いため、ID 906 のスコアが最も高くなります。

GENERIC テキストインデックスへのプレフィクス検索

GENERIC テキストインデックスでは、プレフィクス検索は次のように動作します。

- プレフィクス単語が MAXIMUM TERM LENGTH よりも長い場合、MAXIMUM TERM LENGTH よりも長い単語はテキストインデックスには存在できないため、クエリ文字列から削除されます。したがって、MAXIMUM TERM LENGTH が 3 のテキストインデックスで 'red appl*' を検索することは、'red' を検索することと同義になります。

- プレフィクス単語が MINIMUM TERM LENGTH よりも短く、フレーズ検索の一部でもない場合、プレフィクス検索は通常通り処理されます。したがって、MINIMUM TERM LENGTH が 5 の GENERIC テキストインデックスで、'macintosh a*' を検索すると、macintosh と a で始まる 5 文字以上の任意の単語を含むインデックス付けされたローが返されます。

- プレフィクス単語が MINIMUM TERM LENGTH よりも短いものの、フレーズ検索の一部になっている場合は、プレフィクス単語はクエリから削除されます。したがって、MINIMUM TERM LENGTH が 5 の GENERIC テキストインデックスで、'"macintosh appl* turnover"' を検索することは、「macintosh 任意の単語 turnover」を検索することと同義になります。"macintosh turnover" を含むローは検索されません。macintosh と turnover の間に単語が必要になります。

NGRAM テキストインデックスへのプレフィクス検索

NGRAM テキストインデックスの場合、NGRAM テキストインデックスには N-gram しか含まれておらず、単語の先頭に関する情報は含まれていないため、プレフィクス検索を実行すると予期しない結果が返される可能性があります。また、クエリ単語は N-gram に分割され、検索はクエ

リ単語ではなく N-gram を使用して実行されます。そのため、次のような動作に注意する必要があります。

- プレフィクス単語が N-gram の長さ (MAXIMUM TERM LENGTH) よりも短い場合、クエリはそのプレフィクス単語で始まる N-gram を含むすべてのインデックス付けされたローを返します。たとえば、3-gram のテキストインデックスの場合、'ea*' を検索すると、ea で始まる N-gram を含むすべてのインデックス付けされたローが返されます。したがって、weather と fear という単語がインデックス付けされている場合、これらの N-gram にはそれぞれ eat と ear が含まれているため、ローは一致すると見なされます。

- プレフィクス単語が N-gram の長さよりも長く、フレーズの一部ではなく、近接検索の引数ではない場合、プレフィクス単語は N-gram フレーズに変換され、アスタリスクは削除されます。たとえば、3-gram のテキストインデックスの場合、'purple blac*' を検索することは、'"pur urp rpl ple" AND "bla lac"' を検索することと同義になります。

- フレーズの場合は、次のような動作も発生します。

  - フレーズ内にプレフィクス単語以外の単語がない場合、そのプレフィクス単語は N-gram フレーズに変換され、アスタリスクは削除されます。たとえば、3-gram のテキストインデックスの場合、'"purpl*"' を検索することは、'"pur urp rpl"' を検索することと同義になります。

  - プレフィクス単語がフレーズの末尾に位置する場合は、アスタリスクは削除され、すべての単語は N-gram のフレーズに変換されます。たとえば、3-gram のテキストインデックスの場合、'"purple blac*"' を検索することは、'"pur urp rpl ple bla lac"' を検索することと同義になります。

  - プレフィクス単語がフレーズの末尾に位置していない場合、フレーズは複数のフレーズに分割され、AND で結合されます。たとえば、3-gram のテキストインデックスの場合、'"purp* blac*"' を検索することは、'"pur urp" AND "bla lac"' を検索することと同義になります。

- プレフィクス単語が近接検索の引数の場合、近接検索は AND に変換されます。たとえば、3-gram のテキストインデックスの場合、'red NEAR[1] appl*' を検索することは、'red AND "app ppl"' を検索することと同義になります。

参照

- 「アスタリスク (*) を使用できる構文」『SQL Anywhere サーバ SQL リファレンス』
- 「テキストインデックスの概念と参照」408 ページ
- 「CONTAINS 探索条件」『SQL Anywhere サーバ SQL リファレンス』

近接検索

全文検索では、単一のカラム内で互いに近接する単語を検索できる機能があります。これは「近接検索」と呼ばれます。近接検索を実行するには、2 つの単語を指定して、間に NEAR キーワードかチルダ (~) を挿入します。

NEAR キーワードを使用して、整数引数を指定し、最大距離を使用できます。たとえば、*term1* NEAR[5] *term2* は *term2* から 5 語以内に出現する *term1* のインスタンスを検索します。単語の順序に意味はなく、'term1 NEAR term2' は 'term2 NEAR term1' と同義です。

距離を指定しない場合、データベースサーバはデフォルトの距離として 10 を使用します。

NEAR キーワードの代わりに、チルダ (~) を指定することもできます。たとえば 'term1 ~ term2' のように指定します。ただし、チルダ形式を使用する場合は距離を指定できず、デフォルトの 10 が適用されます。

近接検索の引数としてフレーズを指定することはできません。

NGRAM テキストインデックスを使用した近接検索でプレフィクス単語を引数として指定すると、近接検索は AND 式に変換されます。たとえば、3-gram のテキストインデックスの場合、'red NEAR[1] appl*' を検索することは、'red AND "app ppl"' を検索することと同義になります。これは近接検索ではなくなったため、CONTAINS 句で複数のカラムが指定されている場合に、検索が 1 つのカラムに制限されることはありません。

例

MarketingInformation テーブルの Description カラムで、単語 skin から 10 語以内の距離にある単語 fabric を検索するとします。この場合、次の文を実行できます。

```
SELECT ID, "contains".score, Description
  FROM MarketingInformation CONTAINS ( Description, 'fabric ~ skin' );
```

ID	score	Description
902	1.5572371866083279	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester 「fabric」 is gentle on the earth and soft against your 「skin」. </body></html>

デフォルトの距離が 10 語なので、距離を指定する必要はありませんでした。しかし、距離を 1 語拡張することで、次のような別のローが返されます。

```
SELECT ID, "contains".score, Description
  FROM MarketingInformation CONTAINS ( Description, 'fabric NEAR[11] skin' );
```

ID	score	Description
903	1.5787803210404958	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will serve you equally well on trails or around town. The 「fabric」 has a wicking finish to pull perspiration away from your 「skin」. </body></html>
902	2.163125855043747	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester 「fabric」 is gentle on the earth and soft against your 「skin」. </body></html>

単語間の距離が近いため、ID 903 のスコアの方が高くなります。

ブール検索

全文検索を実行する場合に、複数の単語をブール演算子で区切って指定できます。SQL Anywhere では、全文検索の実行時にブール演算子 AND、OR、AND NOT がサポートされます。

全文検索での AND 演算子の使用

AND 演算子では、AND の両側に指定された単語を両方とも含むローが一致となります。AND 演算子としてアンパサンド (&) を使用することもできます。複数の単語を、間に演算子を入れないで指定した場合、AND を暗黙で指定したことになります。

単語がリストされる順番は重要ではありません。

たとえば、次の文はすべて、MarketingInformation テーブルの Description カラムで **fabric** という単語と **ski** で始まる単語を含むローを検索します。

```
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'ski* AND fabric' );
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'fabric & ski*' );
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'ski* fabric' );
```

全文検索での OR 演算子の使用

OR 演算子では、OR の両側に指定された検索語のうちの少なくとも 1 つを含むローが一致となります。OR 演算子としてパイプ記号 (|) を使用することもできます。この 2 つは同等です。

単語がリストされる順番は重要ではありません。

たとえば、次の文はいずれも、MarketingInformation テーブルの Description カラムで **fabric** という単語または **ski** で始まる単語を含むローを返します。

```
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'ski* OR fabric' );
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'fabric | ski*' );
```

全文検索での AND NOT 演算子の使用

AND NOT 演算子は、左の引数に一致し、右の引数に一致しない結果を検索します。AND NOT 演算子としてハイフン (-) を使用することもできます。この 2 つは同等です。

たとえば、次の文は同義であり、いずれも **fabric** という単語を含み、**ski** で始まる単語は含まないローを返します。

```
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'fabric AND NOT ski*' );
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'fabric -ski*' );
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'fabric & -ski*' );
```

異なるブール演算子の組み合わせ

ブール演算子は、クエリ文字列で組み合わせて使用できます。たとえば、次の文は同義であり、いずれも MarketingInformation テーブルの Description カラムで **fabric** と **skin** を含み、**cotton** は含まない項目を検索します。

```
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'skin fabric -cotton' );
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'fabric -cotton AND skin' );
```

次の文は同義であり、いずれも MarketingInformation テーブルの Description カラムで **fabric** を含むか、または **cotton** と **skin** の両方を含む項目を検索します。

```
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'fabric | cotton AND skin' );
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'cotton skin OR fabric' );
```

単語とフレーズのグループ化

単語と式はカッコを使用してグループ化できます。たとえば、次の文は MarketingInformation テーブルの Description カラムで **cotton** または **fabric** を含み、**ski** で始まる単語を含む項目を検索します。

```
SELECT ID, Description FROM MarketingInformation
  WHERE CONTAINS( MarketingInformation.Description, '( cotton OR fabric ) AND shi*' );
```

ID	Description
902	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical 「shi」rt is designed for high-intensity workouts in hot and humid weather. The recycled polyester 「fabric」 is gentle on the earth and soft against your skin. </body></html>

ID	Description
903	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual「shi」rt made of recycled water bottles. It will serve you equally well on trails or around town. The 「fabric」 has a wicking finish to pull perspiration away from your skin. </body></html>
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 construction. 「Shi」elds against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>

複数カラムに渡る検索

すべてのカラムが同じテキストインデックスの一部である場合は、1 つのクエリで複数のカラムに渡る全文検索を実行できます。

```
SELECT *
  FROM t
    WHERE CONTAINS ( t.c1, t.c2, 'term1|term2' );

SELECT *
  FROM t
  WHERE CONTAINS( t.c1, 'term1' )
    OR CONTAINS( t.c2, 'term2' );
```

t1.c1 に *term1* が含まれている、または *t1.c2* に *term2* が含まれている場合、1 つ目のクエリは一致します。

t1.c1 または *t1.c2* に、*term1* または *term2* が含まれている場合、2 つ目のクエリは一致します。この方法で contains を使用すると、一致のスコアも返されます。

参照

- 「CONTAINS 探索条件」『SQL Anywhere サーバ SQL リファレンス』
- 「全文検索結果のスコア」393 ページ

あいまい検索

あいまい検索を使用して、スペルミスや単語の変形を検索できます。あいまい検索を実行するには、FUZZY 演算子の後に二重引用符で囲まれた文字列を付けて、文字列の近似一致を検索します。たとえば、CONTAINS (Products.Description, 'FUZZY "cotton"') では、**cotton** および **coton** や **cotten** などのスペルミスが返されます。

> **注意**
> あいまい検索は、NGRAM 単語区切りを使用して作成されたテキストインデックスにしか実行できません。NGRAM 単語区切り、およびあいまい検索との関係については、「テキスト設定オブジェクトの作成または変更時の指定項目」396 ページを参照してください。

FUZZY 演算子を使用することは、文字列を長さ *n* の部分文字列に手動で分割し、それらを OR 演算子で区切ることと同じです。たとえば、NGRAM 単語区切りと MAXIMUM TERM LENGTH 3 を指定して設定されたテキストインデックスがあるとします。'FUZZY "500 main street"' を指定することは、'500 OR mai OR ain OR str OR tre OR ree OR eet' を指定することと同義になります。

FUZZY 演算子は、スコアを返す全文検索で便利です。これは、多くの近似一致が返されても、通常は最高スコアを持つ一致だけが意味のある一致だからです。

ビュー検索

ビューまたは派生テーブルで全文検索を使用するには、全文検索を実行したいベーステーブルのカラムにテキストインデックスを構築する必要があります。次の文では、サンプルデータベースの MarketingInformation テーブルにビューを作成します。ビューにはすでにテキストインデックス名があるため、そのビューで全文検索を実行できます。

MarketingInformation ベーステーブルにビューを作成するには、次の文を実行します。

```
CREATE VIEW MarketingInfoView AS
SELECT MI.ProductID AS ProdID,
    MI."Description" AS "Desc"
FROM GROUPO.MarketingInformation AS MI
WHERE MI."ID" > 3
```

次の文を使用すると、基本となるテーブルのテキストインデックスを使用してビューを問い合わせることができます。

```
SELECT *
FROM MarketingInfoView
WHERE CONTAINS ( "Desc", 'Cap OR Tee*' )
```

また、次の文を実行すると、基本テーブルのテキストインデックスを使用して派生テーブルにクエリを実行できます。

```
SELECT *
FROM (
    SELECT MI.ProductID, MI."Description"
    FROM MarketingInformation AS MI
    WHERE MI."ID" > 4 ) AS dt ( P_ID, "Desc" )
WHERE CONTAINS ( "Desc", 'Base*' )
```

注意
全文検索を実行するカラムが、ビューまたは派生テーブルの SELECT リストに含まれている必要があります。

基本となるテーブルのテキストインデックスを使用してビューを検索する場合、次のような制限事項があります。

- ビューには、TOP 句、FIRST 句、DISTINCT 句、GROUP BY 句、ORDER BY 句、UNION 句、INTERSECT 句、EXCEPT 句、または Window 関数を含めることはできない。
- ビューに集合関数を含めることはできない。
- CONTAINS クエリはビュー内のベーステーブルは参照できるが、別のビュー内にあるビュー内のベーステーブルは参照できない。

参照

- 「CONTAINS 探索条件」『SQL Anywhere サーバ SQL リファレンス』

全文検索結果のスコア

クエリの FROM 句に CONTAINS 句を含めると、一致ごとに関連するスコアが付けられます。このスコアは一致がどれだけ近いかを示しており、スコア情報を使用してデータをソートできます。

スコアは、主に次の 2 つの基準に基づいて付けられます。

- **インデックス付けされたローにその単語が出現する回数**　その単語が、インデックス付けされたローに多く出現するほどスコアは高くなります。
- **テキストインデックスにその単語が出現する回数**　その単語が、テキストインデックスに多く出現するほどスコアは低くなります。Sybase Central では、テキストインデックスの **[ボキャブラリ]** タブを表示することで、テキストインデックスに出現する各単語の回数を確認できます。単語をアルファベット順にソートするには、**term** カラムをクリックします。**freq** カラムに、その単語がテキストインデックスに出現する回数が表示されます。

また、全文検索のタイプによっては、スコアに影響する基準が他にもあります。たとえば近接検索では、検索語にどれだけ近接しているかがスコアに影響します。

スコアの使用方法

デフォルトでは、CONTAINS 句の結果セットには **contains** という相関名があり、**score** というカラムが 1 つ含まれています。"contains".score は、SELECT リスト、ORDER BY 句、またはクエリの他の部分で参照できます。ただし、contains は SQL の予約語なので、二重引用符で囲む必要があります。また、別の相関名を指定する方法もあります (たとえば、CONTAINS (expression) AS ct)。このマニュアルで使用している全文検索の例では、スコアカラムは ct.score と記載しています。

次の文は、MarketingInformation テーブルの Description カラムで **stretch** または **comfort** で始まる単語を検索します。

```
SELECT ID, ct.score, Description
   FROM MarketingInformation CONTAINS ( MarketingInformation.Description, 'stretch* | comfort*' ) AS
ct
   ORDER BY ct.score DESC;
```

ID	score	Description
910	5.570408968026068	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts provide all day 「comfort」 on or off the trails. Now with a more 「comfort」 able and 「stretch」 y fabric and an adjustable drawstring waist. </body></html>
907	3.658418186470189	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> A polycarbonate visor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of 「stretch」 to give you a snug yet 「comfort」 able fit every time you wear it. </body></html>
905	1.6750365447462499	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side vents for breathable 「comfort」 during aerobic activities. Moisture-absorbing headband liner. </body></html>

プレフィクス単語 **comfort** のインスタンスが、他の項目には 1 つしかないのに対して、項目 910 には 2 つ含まれているため、910 のスコアが最も高くなります。また、項目 910 にはプレフィクス単語 **stretch** のインスタンスも含まれています。

例 2：複数のカラムの検索

次の例では、複数のカラムに渡る全文検索を実行し、結果のスコアを表示する方法を示しています。

1. 次のようにして、Products テーブルで即時テキストインデックスを作成します。

```
CREATE TEXT INDEX scoringExampleMult
  ON Products ( Description, Name );
```

2. 次にようにして、Description カラムと Name カラムで、**cap** または **visor** という単語の全文検索を実行します。CONTAINS 句の結果には ct という相関名が割り当てられ、結果に含まれるように SELECT リストで参照されます。また、ct.score カラムは ORDER BY 句で参照され、結果はスコアの降順でソートされます。

```
SELECT Products.Description, Products.Name, ct.score
   FROM Products CONTAINS ( Products.Description, Products.Name, 'cap OR visor' ) ct
   ORDER BY ct.score DESC;
```

Description	Name	score
Cloth Visor	Visor	3.5635154905713042
Plastic Visor	Visor	3.4507856451176244
Wool cap	Baseball Cap	3.2340501745357333
Cotton Cap	Baseball Cap	3.090467108972918

複数カラムの検索のスコアは、カラム値が連結され、1 つの値としてインデックス付けされているかのように計算されます。ただし、そのフレーズと NEAR 演算子は、カラムの境界を超えて一致することはなく、複数のカラムに出現する検索語は、単一の連結された値よりもスコアが大きくなります。

3. このマニュアルの他の例を正常に動作させるためには、Products テーブルで作成したテキストインデックスを削除する必要があります。削除するには、次の文を実行します。

```
DROP TEXT INDEX scoringExampleMult ON Products;
```

テキスト設定オブジェクトの概念と参照

テキスト設定オブジェクトは、テキストインデックスの作成時または再表示時に格納される単語を制御し、全文クエリの解釈方法を制御します。各テキスト設定オブジェクトの設定は、ISYSTEXTCONFIG システムテーブルにローとして格納されます。

データベースサーバがテキストインデックスを作成または再表示する場合、テキストインデックスの作成時に指定されたテキスト設定オブジェクトの設定が使用されます。テキストインデッ

クスの作成時にテキスト設定オブジェクトを指定しなかった場合、データベースサーバは、インデックス付けされるカラムのデータ型に基づいてデフォルトのテキスト設定オブジェクトを 1 つ選択します。SQL Anywhere には、デフォルトのテキスト設定オブジェクトが 2 つあります。

既存のテキスト設定オブジェクトの設定を表示するには、SYSTEXTCONFIG システムビューを問い合わせます。

参照

- 「テキスト設定オブジェクトの例」402 ページ
- 「SYSTEXTCONFIG システムビュー」『SQL Anywhere サーバ SQL リファレンス』

テキスト設定オブジェクトの作成または変更時の指定項目

SQL Anywhere には、CHAR データに使用する default_char、および NCHAR データと CHAR データに使用する default_nchar というデフォルトのテキスト設定オブジェクトが 2 つあります。default_nchar はすべてのデータに使用できますが、文字セット変換が実行されます。

テキスト設定オブジェクトがどのように単語の分割に影響するかは、sa_char_terms および sa_nchar_terms システムプロシージャを使用してテストできます。

参照

- 「sa_nchar_terms システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_char_terms システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「テキスト設定オブジェクトの例」402 ページ
- 「デフォルトのテキスト設定オブジェクト」403 ページ
- 「テキスト設定オブジェクトの作成 (Sybase Central の場合)」369 ページ
- 「テキスト設定オブジェクトの変更」369 ページ
- 「あいまい検索」392 ページ
- 「テキストインデックスの概念と参照」408 ページ
- 「CONTAINS 探索条件」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE TEXT CONFIGURATION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TEXT CONFIGURATION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DROP TEXT CONFIGURATION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSTEXTIDX システムビュー」『SQL Anywhere サーバ SQL リファレンス』

TERM BREAKER 句 — 単語区切りアルゴリズムの指定

TERM BREAKER 設定では、文字列を単語に分割するために使用されるアルゴリズムを指定します。単語を格納する GENERIC、または N-gram を格納する NGRAM から選択できます。GENERIC では、組み込みの単語区切りアルゴリズムまたは外部単語区切りを使用できます。

次の表は、TERM BREAKER の値がテキストインデックス処理およびクエリ文字列の処理方法に与える影響を示します。

テキストインデックス	クエリ文字列
● **GENERIC テキストインデックス**　GENERIC テキストインデックスは、NGRAM テキストインデックスをパフォーマンスで上回ります。ただし、GENERIC テキストインデックスではあいまい検索は実行できません。 組み込みのアルゴリズムを使用して GENERIC テキストインデックスを作成すると、英数字以外の文字に挟まれた英数字の文字グループはデータベースサーバによって単語として処理され、位置が割り当てられます。 単語区切り外部ライブラリを使用して GENERIC テキストインデックスを作成すると、単語と単語の位置は外部ライブラリによって定義されます。 単語が単語区切りによって識別されると、単語長の制限値を超える単語やストップリストに含まれる単語は、カウントはされますがテキストインデックスには挿入されません。 ● **NGRAM テキストインデックス**　「N-gram」とは、長さ *n* の文字のグループです。*n* は MAXIMUM TERM LENGTH の値を表します。 NGRAM テキストインデックスを作成すると、データベースサーバは、英数字以外の文字に挟まれた英数字の文字グループをすべて単語として処理します。単語が定義されると、データベースサーバは単語を N-gram に分割します。その際、*n* よりも短い単語、およびストップリストに含まれる N-gram は破棄されます。 たとえば、MAXIMUM TERM LENGTH が 3 の NGRAM テキストインデックスでは、'my red table' という文字列はテキストインデックス内で red tab abl ble という N-gram として表されます。	CONTAINS クエリを解析する場合、データベースサーバはクエリ文字列からキーワードと特殊文字を抽出し、残りの単語に単語区切りアルゴリズムを適用します。たとえば、クエリ文字列が 'ab_cd* AND b*' の場合、* とキーワード AND は抽出され、文字列 ab_cd と b は単語区切りアルゴリズムに渡されて別に解析されます。 ● **GENERIC テキストインデックス**　GENERIC テキストインデックスを問い合わせると、クエリ文字列内の単語は、インデックス付けされる場合と同じ方法で処理されます。クエリ単語をテキストインデックス内の単語と比較することで、マッチングが実行されます。 ● **NGRAM テキストインデックス**　NGRAM テキストインデックスを問い合わせると、クエリ文字列内の単語は、インデックス付けされる場合と同じ方法で処理されます。クエリ単語の N-gram をインデックス付けされた単語の N-gram と比較することで、マッチングが実行されます。

テキストインデックス	クエリ文字列
N-gram の場合、元の単語の位置情報ではなく、N-gram の位置情報が格納されます。	

定義されていない場合、TERM BREAKER のデフォルトは、デフォルトのテキスト設定オブジェクトで設定された値になります。デフォルトのテキスト設定オブジェクトで単語区切りが定義されていない場合、内部単語区切りが使用されます。

参照

- 「プレフィクス検索」383 ページ
- 「デフォルトのテキスト設定オブジェクト」403 ページ
- 「CONTAINS 探索条件」『SQL Anywhere サーバ SQL リファレンス』
- TERM BREAKER 句、ALTER TEXT CONFIGURATION 文『SQL Anywhere サーバ SQL リファレンス』

MINIMUM TERM LENGTH 句 ー 単語の最小長の設定

MINIMUM TERM LENGTH 設定では、インデックスに挿入される単語、または全文クエリで検索される単語の最小長を文字数で指定します。MINIMUM TERM LENGTH は、NGRAM テキストインデックスでは関係ありません。

MINIMUM TERM LENGTH は、プレフィクス検索に対して特別な影響があります。

MINIMUM TERM LENGTH の値は 0 より大きい値にする必要があります。MAXIMUM TERM LENGTH より大きい値を設定すると、MAXIMUM TERM LENGTH が MINIMUM TERM LENGTH と等しくなるように自動的に調整されます。

定義されていない場合、MINIMUM TERM LENGTH のデフォルト値は、デフォルトのテキスト設定オブジェクトで設定された値 (通常は 1) になります。

次の表は、MINIMUM TERM LENGTH の値がテキストインデックス処理およびクエリ文字列の処理方法に与える影響を示します。

テキストインデックス	クエリ文字列
GENERIC テキストインデックス　GENERIC テキストインデックスでは、MINIMUM TERM LENGTH よりも短い単語をテキストインデックスに含めることはできません。 **NGRAM テキストインデックス**　NGRAM テキストインデックスでは、この設定は無視されます。	**GENERIC テキストインデックス**　GENERIC テキストインデックスを問い合わせる場合、MINIMUM TERM LENGTH よりも短いクエリ単語はテキストインデックスには存在できないため、無視されます。 **NGRAM テキストインデックス**　MINIMUM TERM LENGTH 設定は、NGRAM テキストインデックスの全文クエリには影響ありません。

参照

- MINIMUM TERM LENGTH 句、ALTER TEXT CONFIGURATION 文『SQL Anywhere サーバ SQL リファレンス』
- 「プレフィクス検索」383 ページ
- 「デフォルトのテキスト設定オブジェクト」403 ページ

MAXIMUM TERM LENGTH 句 ー 単語の最大長の設定

MAXIMUM TERM LENGTH 設定がどのように使用されるかは、単語区切りアルゴリズムによって異なります。

MAXIMUM TERM LENGTH の値は 60 以下の値にする必要があります。MINIMUM TERM LENGTH より小さい値を設定すると、MINIMUM TERM LENGTH が MAXIMUM TERM LENGTH と等しくなるように自動的に調整されます。

定義されていない場合、MAXIMUM TERM LENGTH のデフォルト値は、デフォルトのテキスト設定オブジェクトで設定された値 (通常は 20) になります。

次の表は、MAXIMUM TERM LENGTH の値がテキストインデックス処理およびクエリ文字列の処理方法に与える影響を示します。

テキストインデックス	クエリ文字列
GENERIC テキストインデックス　GENERIC テキストインデックスでは、MAXIMUM TERM LENGTH はテキストインデックスに挿入される単語の最大長を文字数で指定します。 **NGRAM テキストインデックス**　NGRAM テキストインデックスでは、MAXIMUM TERM LENGTH は単語が分割される N-gram の長さを決定します。N-gram の適切な長さの選択は、言語によって異なります。英語の場合の一般的な値は 4 文字または 5 文字で、中国語の場合は 2 文字または 3 文字です。	**GENERIC テキストインデックス**　GENERIC テキストインデックスでは、MAXIMUM TERM LENGTH よりも長いクエリ単語はテキストインデックスには存在できないため、無視されます。 **NGRAM テキストインデックス**　NGRAM テキストインデックスでは、クエリ単語は長さ *n* の N-gram に分割されます。*n* は MAXIMUM TERM LENGTH と同じ値になります。データベースサーバは N-gram を使用してテキストインデックスを検索します。MAXIMUM TERM LENGTH よりも短い単語はテキストインデックスの N-gram と一致しないため、無視されます。このため、引数が長さ *n* のプレフィクスでないかぎり、近接検索は機能しません。

参照

- MAXIMUM TERM LENGTH 句、ALTER TEXT CONFIGURATION 文『SQL Anywhere サーバ SQL リファレンス』
- 「デフォルトのテキスト設定オブジェクト」403 ページ

STOPLIST 句 ー ストップリストの設定

ストップリストは、テキストインデックスの作成時に無視する単語を指定します。

定義されていない場合、ストップリスト設定のデフォルト値は、デフォルトのテキスト設定オブジェクトの設定 (通常は空の状態) になります。

STOPLIST のテキストインデックスへの影響	STOPLIST のクエリ単語への影響
GENERIC テキストインデックス　GENERIC テキストインデックスでは、ストップリストに含まれる単語はテキストインデックスに挿入されません。 **NGRAM テキストインデックス**　NGRAM テキストインデックスでは、ストップリストに含まれる単語から構成される N-gram はテキストインデックスには含まれません。	**GENERIC テキストインデックス**　GENERIC テキストインデックスでは、ストップリストに含まれるクエリ単語はテキストインデックスには存在できないため、無視されます。 **NGRAM テキストインデックス**　ストップリストに含まれる単語は N-gram に分割され、N-gram は単語のフィルタリングで使用されます。同様に、クエリ単語も N-gram に分割され、ストップリスト内の N-gram と一致するものはテキストインデックスには存在できないため、削除されます。

テキスト設定オブジェクトの設定は、解析時にストップリストに適用されます。つまり、指定された単語区切りと最小長設定/最大長設定が適用されます。

NGRAM テキストインデックスのストップリストは、指定したストップリストの単語ではなく、N-gram の形式で格納されるため、予期しない結果になる場合があります。たとえば、MAXIMUM TERM LENGTH が 3 の NGRAM テキストインデックスの場合、STOPLIST 'there' を指定すると、the her ere の N-gram がストップリストとして格納されます。これは、the、her、ere の N-gram を含むすべての単語に対する問い合わせに影響します。

注意
文字列リテラルの指定に関連する制約事項は、ストップリストにも該当します。たとえば、アポストロフィはエスケープする必要があります。

Samples ディレクトリには、複数の言語用のストップリストをロードするサンプルコードが含まれています。これらのサンプルストップリストは、GENERIC テキストインデックスでのみ使用することをおすすめします。Samples ディレクトリのロケーションについては、「サンプルディレクトリ」『SQL Anywhere サーバ データベース管理』を参照してください。

参照

- 「デフォルトのテキスト設定オブジェクト」403 ページ
- STOPLIST 句、ALTER TEXT CONFIGURATION 文『SQL Anywhere サーバ SQL リファレンス』
- 「文字列リテラル」『SQL Anywhere サーバ SQL リファレンス』

PREFILTER 句 ー 外部事前フィルタアルゴリズムの指定

「事前フィルタリング」は、Word、PDF、HTML、XML などのファイルからテキストデータを抽出するプロセスです。テキストインデックス処理のコンテキストでは、事前フィルタリングを使用すると、インデックス処理するデータのみを抽出し、HTML タグなどの不要なコンテンツがインデックス処理されないようにできます。特定のドキュメントタイプ (Microsoft Word ドキュメントなど) では、全文インデックスを有用なものにするには、事前フィルタリングが必要です。

SQL Anywhere には組み込みの事前フィルタ機能はありません。ただし、要件に応じて事前フィルタリングを実行する外部事前フィルタライブラリを作成し、外部事前フィルタライブラリを指すようにテキスト設定オブジェクトを変更できます。

次の表は、PREFILTER EXTERNAL NAME の値がテキストインデックス処理およびクエリ文字列の処理方法に与える影響を示します。

テキストインデックス	クエリ文字列
GENERIC テキストインデックスと NGRAM テキストインデックス　外部事前フィルタは入力値 (ドキュメント) を受け入れ、事前フィルタライブラリによって指定されたルールに従って入力値をフィルタリングします。その結果生成されたテキストは単語区切りに渡され、その後テキストインデックスが作成または更新されます。	**GENERIC テキストインデックスと NGRAM テキストインデックス**　クエリ文字列は事前フィルタを介して渡されないため、PREFILTER EXTERNAL NAME 句の設定はクエリ文字列に影響を与えません。

SQL Anywhere インストールの *ExternalLibrariesFullText* ディレクトリには、参照用に、事前フィルタと単語区切りのサンプルコードが含まれています。このディレクトリは、*Samples* ディレクトリ内にあります。*Samples* ディレクトリのロケーションについては、「サンプルディレクトリ」『SQL Anywhere サーバ データベース管理』を参照してください。

参照

- 「外部事前フィルタライブラリ」444 ページ
- PREFILTER EXTERNAL NAME 句、ALTER TEXT CONFIGURATION 文『SQL Anywhere サーバ SQL リファレンス』

日付、時刻、およびタイムスタンプ形式の設定

テキスト設定オブジェクトが作成されると、現在の接続の date_format、time_format、timestamp_format、timestamp_with_time_zone_format の各オプションの値はテキスト設定オブジェクトに格納されます。これらのオプション値は、テキスト設定オブジェクトを使用して作成されたテキストインデックスの DATE、TIME、TIMESTAMP の各カラムがフォーマットされる方法を制御します。テキスト設定オブジェクトのこれらのオプション値を明示的に設定することはできません。設定には、テキスト設定オブジェクトを作成した接続に対して有効になっているオプション値が反映されます。ただし、これらのオプション値の変更は可能です。

参照

- 「テキスト設定オブジェクトの変更」369 ページ
- 「ALTER TEXT CONFIGURATION 文」『SQL Anywhere サーバ SQL リファレンス』

テキスト設定オブジェクトに影響するデータベースオプション

テキスト設定オブジェクトが作成されると、date_format、time_format、timestamp_format の各データベースオプションの現在の設定はテキスト設定オブジェクトに格納されます。このようになるのは、テキスト設定オブジェクトに依存するテキストインデックスの作成時および再表示時に、これらの設定が文字列の変換に影響するためです。

設定をテキスト設定オブジェクトに格納することで、依存するテキストインデックスに格納されるデータのフォーマットを変えることなく、データベースオプションの設定を変更できます。

テキストインデックスにある日付や時刻を表す文字列の形式を変更するには、次の手順に従います。

1. テキストインデックス、テキスト設定オブジェクト、およびすべての依存するテキストインデックスを削除します。
2. テキスト設定オブジェクトの作成に使用したデフォルトのテキスト設定オブジェクトとすべての依存するテキストインデックスを削除します。
3. データベースオプションを目的の形式に変更します。
4. テキスト設定オブジェクトを作成します。
5. 新しいテキスト設定オブジェクトを使用して、テキストインデックスを作成します。

> **注意**
> テキストインデックスを作成または再表示する場合は、conversion_error オプションを ON に設定する必要があります。

参照

- 「テキスト設定オブジェクトの作成または変更時の指定項目」396 ページ
- 「date_format オプション」『SQL Anywhere サーバ データベース管理』
- 「time_format オプション」『SQL Anywhere サーバ データベース管理』
- 「timestamp_format オプション」『SQL Anywhere サーバ データベース管理』
- 「conversion_error オプション」『SQL Anywhere サーバ データベース管理』

テキスト設定オブジェクトの例

データベースのすべてのテキスト設定オブジェクト、およびその設定に関するリストについては、SYSTEXTCONFIG システムビューを問い合わせてください (例：SELECT * FROM SYSTEXTCONFIG)。

テキスト設定オブジェクトで文字列がどのように単語に分割されるかは、sa_char_terms and sa_nchar_terms システムプロシージャを使用してテストできます。

デフォルトのテキスト設定オブジェクト

SQL Anywhere には、NCHAR データに使用する 「default_nchar」、および非 NCHAR データに使用する 「default_char」 という、2 つのデフォルトのテキスト設定オブジェクトがあります。これらの設定は、テキスト設定オブジェクトやテキストインデックスを初めて作成する際に作成されます。

インストール時における default_char と default_nchar の設定を以下の表に示します。これらの設定は、ほとんどの文字ベースの言語で最適であるという理由で選択されています。デフォルトのテキスト設定オブジェクトの設定を変更しないことを強くおすすめします。

設定	インストールされる値
TERM BREAKER	0 (GENERIC)
MINIMUM TERM LENGTH	1
MAXIMUM TERM LENGTH	20
STOPLIST	(空)

デフォルトのテキスト設定オブジェクトを削除すると、テキストインデックスやテキスト設定オブジェクトを次回作成する際に自動的に作成されます。

データベースサーバによってデフォルトのテキスト設定オブジェクトが作成されると、日付値と時刻値が文字列に変換される方法に影響を与えるデータベースオプションは、現在の設定からテキスト設定オブジェクトに保存されます。

参照

- 「SYSTEXTCONFIG システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_char_terms システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_nchar_terms システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「テキスト設定オブジェクトに影響するデータベースオプション」402 ページ
- 「テキスト設定オブジェクトの作成または変更時の指定項目」396 ページ

テキスト設定オブジェクトの例

次の表は、さまざまなテキスト設定オブジェクトの設定、インデックス付けされる対象に与える影響、および全文クエリ文字列の解釈方法を示したものです。すべての例で、文字列 'I'm not sure I understand' を使用しています。

設定	インデックス付けされる単語	クエリ解釈
TERM BREAKER GENERIC MINIMUM TERM LENGTH 1 MAXIMUM TERM LENGTH 20 STOPLIST ''	I m not sure I understand	("I m" AND NOT sure) AND I AND understand' 元の文字列の 'not' は 'not' というワードではなく、演算子として解釈されます。
TERM BREAKER GENERIC MINIMUM TERM LENGTH 2 MAXIMUM TERM LENGTH 20 STOPLIST 'not and'	sure understand	'understand'. not' はフレーズ "i am" と "sure" の間の演算子 (AND NOT) として解釈されるため、'sure' は削除されます。フレーズ "i am" には短すぎて削除される単語が含まれているため、AND NOT 条件 ('sure') の右側も削除されます。これにより、'understand' のみが残ります。
TERM BREAKER NGRAM MAXIMUM TERM LENGTH 3 STOPLIST 'not and'	sur ure und nde der ers rst sta tan	'und AND nde AND der AND ers AND rst AND sta AND tan'. あいまい検索の場合： 'und OR nde OR der OR ers OR rst OR sta OR tan'
TERM BREAKER GENERIC MINIMUM TERM LENGTH 1 MAXIMUM TERM LENGTH 20 STOPLIST 'not and'	I m sure I understand	'("I m" AND NOT sure) AND I AND understand'.

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

設定	インデックス付けされる単語	クエリ解釈
TERM BREAKER NGRAM MAXIMUM TERM LENGTH 20 STOPLIST 'not and'	20 文字以上の単語がないため、インデックス付けされるものはありません。 これは、MAXIMUM TERM LENGTH の GENERIC テキストインデックスへの影響と NGRAM テキストインデックスへの影響が異なることを示しています。NGRAM テキストインデックスの場合、MAXIMUM TERM LENGTH は、テキストインデックスに挿入される N-gram の長さを設定します。	クエリ文字列から 20 文字の N-gram を構成できないため、検索を実行すると空の結果セットが返されます。

CONTAINS 文字列の解釈方法の例

次の表は、テキスト設定オブジェクトの文字列の設定がどのように解釈されるかについて、例を示したものです。

解釈される文字列の列にあるカッコ内の数字は、各単語の位置情報を表しています。この数字は、説明の目的でマニュアルに記載されています。格納される実際の単語にはカッコ内の数字は含まれません。

設定	文字列	解釈される文字列
TERM BREAKER GENERIC MINIMUM TERM LENGTH 3 MAXIMUM TERM LENGTH 20	'w*'	'"w*(1)"'
	'we*'	'"we*(1)"'
	'wea*'	'"wea*(1)"'
	'we* -the'	'"we*(1)" -"the(1)"'
	'we* the'	"we*(1)" & "the(1)"'
	'for* | wonderl*'	'"for*(1)" | "wonderl*(1)"'
	'wonderlandwonderlandwonderland*'	''
	'"tr* weather"'	'"weather(1)"'
	'"tr* the weather"'	'"the(1) weather(2)"'

設定	文字列	解釈される文字列
	'"wonderlandwonderlandwonderland* wonderland"'	'"wonderland(1)"'
	'"wonderlandwonderlandwonderland* weather"'	'"weather(1)"'
	'"the_wonderlandwonderlandwonderland* weather"'	'"the(1) weather(3)"'
	'the_wonderlandwonderlandwonderland* weather'	'"the(1)" & "weather(1)"'
	'"light_a* the end" & tunnel'	'"light(1) the(3) end(4)" & "tunnel(1)"'
	light_b* the end" & tunnel'	'"light(1) the(3) end(4)" & "tunnel(1)"'
	'"light_at_b* end"'	'"light(1) end(4)"'
	'and_te*'	'"and(1) te*(2)"'
	'a_long_and_t* & journey'	'"long(2) and(3) t*(4)" & "journey(1)"'
	'weather -is'	'"weather(1)"'
TERM BREAKER NGRAM MAXIMUM TERM LENGTH 3	'w*'	'"w*(1)"'
	'we*'	'"we*(1)"'
	'wea*	'"wea(1)"'
	'we* -the'	'"we*(1)" -"the(1)"'
	'we* the'	'"we*(1)" & "the(1)"'
	'for | la*'	'"for(1)" | "la*(1)"'
	'weath*'	'"wea(1) eat(2) ath(3)"'
	'"ful weat*"'	'"ful(1) wea(2) eat(3)"'
	'"wo* la*"'	'"wo*(1)" & "la*(2)"'
	'"la* won* "'	'"la*(1)" & "won(2)"'

設定	文字列	解釈される文字列
	'"won* weat*"'	'"won(1)" & "wea(2) eat(3)"'
	'"won* weat"'	'"won(1)" & "wea(2) eat(3)"'
	'"wo* weat*"'	'"wo*(1)" & "wea(2) eat(3)"'
	'"weat* wo* "'	'"wea(1) eat(2)" & "wo*(3)"'
	'"wo* weat"'	'"wo*(1)" & "wea(2) eat(3)"'
	'"weat wo* "'	'"wea(1) eat(2) wo*(3)"'
	'w* NEAR[1] f*'	'"w*(1)" & "f*(1)"'
	'weat* NEAR[1] f*'	"wea(1) eat(2)" & "f*(1)"'
	'f* NEAR[1] weat*'	'"f*(1)" & "wea(1) eat(2)"'
	'weat NEAR[1] f*'	'"wea(1) eat(2)" & "f*(1)"'
	'f* NEAR[1] weat'	'"f*(1)" & "wea(1) eat(2)"'
	'for NEAR[1] weat*'	'"for(1)" & "wea(1) eat(2)"'
	'weat* NEAR[1] for'	'"wea(1) eat(2)" & "for(1)"'
	'and_tedi*'	'"and(1) ted(2) edi(3)"'
	'and_t*'	'"and(1) t*(2)"'
	'"and_tedi*"'	'"and(1) ted(2) edi(3)"'
	'"and-t*"'	'"and(1) t*(2)"'
	'"ligh* at_the_end of_the tun* nel"'	'"lig(1) igh(2)" & ("the(4) end(5) the(7) tun(8)" & "nel(9)")'
	'"ligh* at_the_end_of_the_tun* nel"'	'"lig(1) igh(2)" & ("the(4) end(5) the(7) tun(8)" & "nel(9)")'
	'"at_the_end of_the tun* ligh* nel"'	'"the(2) end(3) the(5) tun(6)" & ("lig(7) igh(8)" & "nel(9)")'
	'l* NEAR[1] and_t*'	"l*(1)" & "and(1) t*(2)"'

設定	文字列	解釈される文字列
	'long NEAR[1] and_t*'	'"lon(1) ong(2)" & "and(1) t*(2)"'
	'end NEAR[3] tunne*'	'"end(1)" & "tun(1) unn(2) nne(3)"'
TERM BREAKER NGRAM MAXIMUM TERM LENGTH 3 SKIPPED TOKENS IN TABLE AND IN QUERIES	'"cat in a hat"'	'"cat(1) hat(4)"'
	'"cat in_a hat"'	'"cat(1) hat(4)"'
	'"cat_in_a_hat"'	'"cat(1) hat(4)"'
	'"cat_in a_hat"'	'"cat(1) hat(4)"'
	'cat in a hat'	'"cat(1)" & "hat(1)"'
	'cat in_a hat'	'"cat(1)" & "hat(1)"'
	'"ice hat"'	'"ice(1) hat(2)"'
	'ice NEAR[1] hat'	'"ice(1)" NEAR[1] "hat(1)"'
	'ear NEAR[2] hat'	'"ear(1)" NEAR[2] "hat(1)"'
	'"ear a hat"'	'"ear(1) hat(3)"'
	'"cat hat"'	'"cat(1) hat(2)"'
	'cat NEAR[1] hat'	'"cat(1)" NEAR[1] "hat(1)"'
	'ear NEAR[1] hat'	'"ear(1)" NEAR[1] "hat(1)"'
	'"ear hat"'	'"ear(1) hat(2)"'
	'"wear a a hat"'	'"wea(1) ear(2) hat(5)"'
	'weather -is'	'"wea(1) eat(2) ath(3) the(4) her(5)"'

テキストインデックスの概念と参照

全文検索を実行すると、テーブルローではなく、「テキストインデックス」が検索されます。したがって、全文検索を実行する前に、検索するカラムにテキストインデックスを作成する必要があります。テキストインデックスには、インデックス付けされたカラム内の単語の位置情報が格納されます。テキストインデックスを使用したクエリは、テーブル内のすべての値をスキャンする必要があるクエリよりも高速になる可能性があります。

テキストインデックスを作成する場合、テキストインデックスの作成と再表示に使用される「テキスト設定オブジェクト」を指定できます。テキスト設定オブジェクトには、インデックスの構築方法に影響する設定が含まれます。テキスト設定オブジェクトを指定しない場合、データベースサーバはデフォルトの設定オブジェクトを使用します。

テキストインデックスの「再表示タイプ」も指定できます。再表示タイプによって、テキストインデックスの再表示の頻度が決まります。最後に再表示されたテキストインデックスが最も正確な結果を返します。ただし、再表示には時間がかかるため、パフォーマンスに影響する可能性があります。たとえば、インデックス付けされたテーブルを頻繁に更新する場合、基本となるデータが変更されるたびにテキストインデックスを再表示するように設定すると、パフォーマンスに影響する可能性があります。

テキストインデックスの単語の位置情報が維持されていることを確認するために、VALIDATE TEXT INDEX 文を使用できます。位置情報が維持されていない場合は、エラーが生成されます。

既存のテキストインデックスの設定を表示するには、sa_text_index_stats システムプロシージャを使用します。

参照

- 「VALIDATE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「テキストインデックスの再表示タイプ」409 ページ
- 「テキスト設定オブジェクトの概念と参照」395 ページ
- 「sa_text_index_stats システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

テキストインデックスの再表示タイプ

テキストインデックスを作成する場合、「再表示タイプ」も選択する必要があります。テキストインデックスでは、即時、自動、手動の 3 つの再表示タイプがサポートされています。再表示タイプは、テキストインデックスの作成時に定義します。即時テキストインデックスの場合を除き、再表示タイプはテキストインデックスの作成後に変更できます。

- **IMMEDIATE REFRESH**　IMMEDIATE REFRESH テキストインデックスは、基本となるテーブルまたはマテリアライズドビューのデータが変更されると再表示されます。これは基本テーブルでデータを常に最新の状態に保つ必要がある場合や、インデックス付けされたカラムが比較的短い場合、またはデータ変更の頻度が低い場合にのみ、使用することをおすすめします。

  テキストインデックスのデフォルトの再表示タイプは IMMEDIATE REFRESH です。マテリアライズドビューのテキストインデックスは、IMMEDIATE REFRESH のみサポートしています。

  AUTO REFRESH または MANUAL REFRESH のテキストインデックスを、IMMEDIATE REFRESH テキストインデックスに変更することはできません。変更したい場合は、該当するテキストインデックスを削除して、IMMEDIATE REFRESH テキストインデックスとして再作成する必要があります。

  IMMEDIATE REFRESH テキストインデックスはすべての独立性レベルをサポートします。IMMEDIATE REFRESH テキストインデックスは、作成時にデータが設定され、この初期再表示中に、テーブルまたはマテリアライズドビューに排他ロックがかけられます。

- **AUTO REFRESH**　AUTO REFRESH テキストインデックスは、指定された間隔で自動的に再表示されます。データがある程度古くなってもかまわない場合に使用することをおすすめします。古いインデックスを問い合わせると、一致するローのうち、最後に再表示されてから変更されていないものが返されます。したがって、最後の再表示以降に挿入、削除、更新されたローはクエリで返されません。

  AUTO REFRESH テキストインデックスは、次のいずれかの条件が当てはまる場合、指定した間隔よりも頻繁に再表示されることがあります。

  ○最後の再表示からの時間が再表示間隔より長い場合。

  ○保留中のすべてのローの合計長 (sa_text_index_stats システムプロシージャによって返される pending_length) が合計インデックスサイズ (sa_text_index_stats によって返される doc_length) の 20% を超える場合。

  ○削除された長さが合計インデックスサイズ (doc_length) の 50 % を超える場合。この場合には、増分更新ではなく常に再構築が実行されます。

  AUTO REFRESH テキストインデックスは、独立性レベル 0 を使用して再表示されます。

  AUTO REFRESH テキストインデックスには、作成時にはデータが含まれていないため、最初の再表示が行われるまで利用できません。最初の再表示は通常、テキストインデックスが作成されてから 1 分以内に行われます。REFRESH TEXT INDEX 文を使用して、AUTO REFRESH テキストインデックスを手動で再表示することもできます。

  AUTO REFRESH テキストインデックスは、dbunload で -g オプションを指定しない限り、再ロード時に再表示されません。

- **MANUAL REFRESH**　MANUAL REFRESH テキストインデックスは、ユーザの実行によってのみ再表示されます。基本となるテーブルのデータがめったに変更されない場合、データの古さの程度が大きくても許容できる場合、またはイベントや条件を満たした後に再表示したい場合に使用することをおすすめします。古いインデックスを問い合わせると、一致するローのうち、最後に再表示されてから変更されていないものが返されます。したがって、最後の再表示以降に挿入、削除、更新されたローはクエリで返されません。

  MANUAL REFRESH テキストインデックスには、独自の再表示方針を定義できます。次の例では、引数として渡される再表示間隔と、AUTO REFRESH テキストインデックスに使用するルールと同様のルールを使用して、すべての MANUAL REFRESH テキストインデックスが再表示されます。

```
CREATE PROCEDURE refresh_manual_text_indexes(
  refresh_interval UNSIGNED INT )
BEGIN
 FOR lp1 AS c1 CURSOR FOR
   SELECT ts.*
   FROM SYS.SYSTEXTIDX ti JOIN sa_text_index_stats( ) ts
   ON ( ts.index_id = ti.index_id )
   WHERE ti.refresh_type = 1 -- manual refresh indexes only
 DO
   BEGIN
    IF last_refresh_utc IS null
    OR cast(pending_length as float) / (
      IF doc_length=0 THEN NULL ELSE doc_length ENDIF) > 0.2
```

```
      OR DATEDIFF( MINUTE, CURRENT UTC TIMESTAMP, last_refresh_utc )
        > refresh_interval THEN
       EXECUTE IMMEDIATE 'REFRESH TEXT INDEX ' || text-index-name || ' ON "'
       || table-owner || '"."' || table-name || '";
      END IF;
     END;
    END FOR;
  END;
```

いつでも、sa_text_index_stats システムプロシージャを使用して、再表示が必要かどうか、再表示を完全再構築にするか、増分更新にするかを決定できます。

MANUAL REFRESH テキストインデックスには、作成時にはデータが含まれていないため、再表示するまで利用できません。MANUAL REFRESH テキストインデックスを再表示するには、REFRESH TEXT INDEX 文を使用します。

MANUAL REFRESH テキストインデックスは、dbunload で -g オプションを指定しない限り、再ロード時に再表示されません。

参照

- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』
- 「テキストインデックスの作成」371 ページ
- 「テキスト設定オブジェクトの作成または変更時の指定項目」396 ページ
- 「sa_text_index_stats システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REFRESH TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「isolation_level オプション」『SQL Anywhere サーバ データベース管理』
- 「sa_text_index_stats システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

チュートリアル：GENERIC テキストインデックスに対する全文検索の実行

GENERIC 単語区切りを使用するテキストインデックスに対して、全文検索を実行します。

前提条件

CREATE TEXT CONFIGURATION と CREATE TABLE のシステム権限が必要です。さらに、SELECT ANY TABLE システム権限、または、テーブル MarketingInformation に対する SELECT 権限も必要です。

♦ タスク

1. Interactive SQL を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Interactive SQL]** をクリックします。
2. **[接続]** ウィンドウで、次のフィールドに次のように入力します。
   a. **[認証]** ドロップダウンリストで **[データベース]** をクリックします。

b. **[ユーザ ID]** フィールドに、**DBA** と入力します。

c. **[パスワード]** フィールドに、**sql** と入力します。

d. **[アクション]** ドロップダウンリストで、**[ODBC データソースを使用した接続]** を選択します。

3. 次の文を実行して、myTxtConfig というテキスト設定オブジェクトを作成します。FROM 句を含めて、テンプレートとして使用するテキスト設定オブジェクトを指定する必要があります。

```
CREATE TEXT CONFIGURATION myTxtConfig FROM default_char;
```

4. 次の文を実行して、because、about、therefore、only という単語を含むストップリストを追加することにより、テキスト設定オブジェクトをカスタマイズします。次に、単語の最大長を 30 に設定します。

```
ALTER TEXT CONFIGURATION myTxtConfig
  STOPLIST 'because about therefore only';
ALTER TEXT CONFIGURATION myTxtConfig
  MAXIMUM TERM LENGTH 30;
```

5. Sybase Central を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Sybase Central]** をクリックします。

6. **[接続]** » **[SQL Anywhere 16 に接続]** をクリックします。

7. **[接続]** ウィンドウで、次のフィールドに次のように入力します。

a. **[認証]** ドロップダウンリストで **[データベース]** をクリックします。

b. **[ユーザ ID]** フィールドに、**DBA** と入力します。

c. **[パスワード]** フィールドに、**sql** と入力します。

d. **[アクション]** ドロップダウンリストで、**[ODBC データソースを使用した接続]** を選択します。

8. MarketingInformation テーブルのコピーを作成します。

a. **[テーブル]** フォルダを展開します。

b. **[MarketingInformation]** を右クリックし、**[コピー]** をクリックします。

c. **[テーブル]** フォルダを右クリックし、**[貼り付け]** をクリックします。

d. **[名前]** フィールドに、**MarketingInformation1** と入力します。

e. **[OK]** をクリックします。

9. Interactive SQL で次の文を実行し、新しいテーブルにデータを設定します。

```
INSERT INTO MarketingInformation1
  SELECT * FROM GROUPO.MarketingInformation;
```

10. サンプルデータベースの MarketingInformation1 テーブルの Description カラムに、myTxtConfig テキスト設定オブジェクトを参照するテキストインデックスを作成します。再表示間隔を 24 時間に設定します。

```
CREATE TEXT INDEX myTxtIndex ON MarketingInformation1 ( Description )
  CONFIGURATION myTxtConfig
  AUTO REFRESH EVERY 24 HOURS;
```

11. 次の文を実行して、テキストインデックスを再表示します。

```
REFRESH TEXT INDEX myTxtIndex ON MarketingInformation1;
```

12. 次の文を実行して、テキストインデックスをテストします。

    a. この文により、テキストインデックスで **cotton** または **cap** という単語が検索されます。結果はスコアの降順でソートされます。テキストインデックスで、**cap** という単語が出現する頻度は **cotton** よりも低いため、**Cap** のスコアの方が高くなります。

```
SELECT ID, Description, ct.*
  FROM MarketingInformation1
    CONTAINS ( Description, 'cotton | cap' ) ct
  ORDER BY score DESC;
```

ID	Description	Score
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball 「Cap」</title></head><body lang=EN-US> A lightweight wool「cap」with mesh side vents for breathable comfort during aerobic activities. Moisture-absorbing headband liner. </body></html>	2.2742084275032632

ID	Description	Score
904	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball 「Cap」</title></head><body lang=EN-US> This fashionable hat is ideal for glacier travel, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	1.6980426550094467
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	0.9461597363521859

ID	Description	Score
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying 「cotton」 shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	0.9244136988525732
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	0.9134171046194403

ID	Description	Score
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80%「cotton」/20% polyester blend makes it easy to keep them clean. </body></html>	0.8856420222728282

b. 次の文により、テキストインデックスで cotton という単語が検索されます。visor というワードも含むローは破棄されます。CONTAINS 句では述部が使用されるため、結果は記録されません。

```
SELECT ID, Description
  FROM MarketingInformation1
  WHERE CONTAINS( Description, 'cotton -visor' );
```

ID	Description
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown「cotton」hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>

ID	Description
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80%「cotton」/20% polyester blend makes it easy to keep them clean. </body></html>
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying 「cotton」 shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>

c. 次の文は、**cotton** という単語について各ローをテストします。ローにこの単語が含まれている場合は Results カラムに 1 が表示され、含まれていない場合は 0 が返されます。

```
SELECT ID, Description, IF CONTAINS ( Description, 'cotton' )
    THEN 1
    ELSE 0
    ENDIF AS Results
  FROM MarketingInformation1;
```

ID	Description	Results
901	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> We've improved the design of this perennial favorite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>	0
902	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>	0

ID	Description	Results
903	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will serve you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>	0
904	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier travel, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	0

ID	Description	Results
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side vents for breathable comfort during aerobic activities. Moisture-absorbing headband liner. </body></html>	0
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	1

ID	Description	Results
907	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</ title></head><body lang=EN-US> A polycarbonate visor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to give you a snug yet comfortable fit every time you wear it. </ body></html>	0
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</ title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage.</ span> </body></html>	1

ID	Description	Results
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80%「cotton」/20% polyester blend makes it easy to keep them clean. </body></html>	1
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying 「cotton」 shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	1

13. Interactive SQL と Sybase Central を閉じます。

結果

GENERIC テキストインデックスの全文検索が実行されました。

次の手順

(省略可) サンプルデータベース (*demo.db*) を元の状態にリストアします。「サンプルデータベースの再作成 (demo.db)」『SQL Anywhere 16 紹介』を参照してください。

参照

- 「テキスト設定オブジェクトの概念と参照」395 ページ
- 「CREATE TEXT CONFIGURATION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TEXT CONFIGURATION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「テキストインデックスの概念と参照」408 ページ
- 「CREATE TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』

チュートリアル：あいまい全文検索の実行

NGRAM 単語区切りを使用するテキストインデックスに対して、あいまい全文検索を実行します。

前提条件

CREATE TEXT CONFIGURATION と CREATE TABLE のシステム権限が必要です。さらに、SELECT ANY TABLE システム権限、または、テーブル MarketingInformation に対する SELECT 権限も必要です。

♦ タスク

1. Interactive SQL を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Interactive SQL]** をクリックします。

2. **[接続]** ウィンドウで、次のフィールドに入力します。
   a. **[認証]** ドロップダウンリストで **[データベース]** をクリックします。
   b. **[ユーザ ID]** フィールドに、**DBA** と入力します。
   c. **[パスワード]** フィールドに、**sql** と入力します。
   d. **[アクション]** ドロップダウンリストで、**[ODBC データソースを使用した接続]** を選択します。

3. 次の文を実行して、myFuzzyTextConfig というテキスト設定オブジェクトを作成します。FROM 句を含めて、テンプレートとして使用するテキスト設定オブジェクトを指定する必要があります。

```
CREATE TEXT CONFIGURATION myFuzzyTextConfig FROM default_char;
```

4. 次の文を実行して、単語区切りを NGRAM に変更し、単語の最大長を 3 に設定します。あいまい検索は N-gram を使用して実行されます。

```
ALTER TEXT CONFIGURATION myFuzzyTextConfig
  TERM BREAKER NGRAM;
ALTER TEXT CONFIGURATION myFuzzyTextConfig
  MAXIMUM TERM LENGTH 3;
```

5. Sybase Central を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Sybase Central]** をクリックします。

6. **[接続]** » **[SQL Anywhere 16 に接続]** をクリックします。

7. **[接続]** ウィンドウで、次のフィールドに入力します。
   a. **[認証]** ドロップダウンリストで **[データベース]** をクリックします。
   b. **[ユーザ ID]** フィールドに、**DBA** と入力します。
   c. **[パスワード]** フィールドに、**sql** と入力します。
   d. **[アクション]** ドロップダウンリストで、**[ODBC データソースを使用した接続]** を選択します。

8. MarketingInformation テーブルのコピーを作成します。
   a. Sybase Central で **[テーブル]** フォルダを拡張します。
   b. **[MarketingInformation]** を右クリックし、**[コピー]** をクリックします。
   c. **[テーブル]** フォルダを右クリックし、**[貼り付け]** をクリックします。
   d. **[名前]** フィールドに **MarketingInformation2** と入力します。**[OK]** をクリックします。

9. Interactive SQL で、次の文を実行して、MarketingInformation2 テーブルにデータを追加します。

```
INSERT INTO MarketingInformation2
  SELECT * FROM GROUPO.MarketingInformation;
```

10. 次の文を実行して、MarketingInformation2 テーブルの Description カラムに、myFuzzyTextConfig テキスト設定オブジェクトを参照するテキストインデックスを作成します。

```
CREATE TEXT INDEX myFuzzyTextIdx ON MarketingInformation2 ( Description )
  CONFIGURATION myFuzzyTextConfig;
```

11. 次の文を実行して、**coten** に似た単語を検索します。

```
SELECT MarketingInformation2.Description, ct.*
   FROM MarketingInformation2 CONTAINS ( MarketingInformation2.Description, 'FUZZY "coten"' )
ct
   ORDER BY ct.score DESC;
```

Description	Score
<html><head><meta http-equiv=「Content」-Type 「content」="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	0.9461597363521859

Description	Score
<html><head><meta http-equiv=「Content」-Type 「content」="text/html; charset=windows-1252"><title>Shorts</ title></head><body lang=EN-US> These quick-drying 「cotton」 shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	0.9244136988525732
<html><head><meta http-equiv=「Content」-Type 「content」="text/html; charset=windows-1252"><title>Visor</ title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></ html>	0.9134171046194403
<html><head><meta http-equiv=「Content」-Type 「content」="text/html; charset=windows-1252"><title>Sweatshirt</ title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% 「cotton」/ 20% polyester blend makes it easy to keep them clean. </body></html>	0.8856420222728282
<html><head><meta http-equiv=「Content」-Type 「content」="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier travel, sea-kayaking, and hiking. With concealed draw cord for windy days.</ span> </body></html>	0

Description	Score
<html><head><meta http-equiv=「Content」-Type 「content」="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side vents for breathable comfort during aerobic activities. Moisture-absorbing headband liner. </body></html>	0
<html><head><meta http-equiv=「Content」-Type 「content」="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> We've improved the design of this perennial favorite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>	0
<html><head><meta http-equiv=「Content」-Type 「content」="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will serve you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>	0
<html><head><meta http-equiv=「Content」-Type 「content」="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>	0

Description	Score
<html><head><meta http-equiv=「Content」-Type 「content」="text/html; charset=windows-1252"><title>Visor</ title></head><body lang=EN-US> A polycarbonate visor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to give you a snug yet comfortable fit every time you wear it.</ span> </body></html>	0

注意
最後の 6 つのローに、一致する N-gram を含む単語があります。ただし、これらの単語はテーブル内のすべてのローに含まれているため、スコアは割り当てられていません。

12. Interactive SQL と Sybase Central を閉じます。

結果

あいまい全文検索が実行されました。

次の手順

(省略可) サンプルデータベース (*demo.db*) を元の状態にリストアします。「サンプルデータベースの再作成 (demo.db)」『SQL Anywhere 16 紹介』を参照してください。

参照

- 「あいまい検索」392 ページ
- 「テキスト設定オブジェクトの概念と参照」395 ページ
- 「CREATE TEXT CONFIGURATION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TEXT CONFIGURATION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「テキストインデックスの概念と参照」408 ページ
- 「CREATE TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「全文検索結果のスコア」393 ページ

チュートリアル：NGRAM テキストインデックスに対する非あいまい全文検索の実行

NGRAM 単語区切りを使用するテキストインデックスに対して、非あいまい全文検索を実行します。中国語、日本語、韓国語データの全文検索を作成する際もこの手順で実行できます。

前提条件

CREATE TEXT CONFIGURATION と CREATE TABLE のシステム権限が必要です。さらに、SELECT ANY TABLE システム権限、または、テーブル MarketingInformation に対する SELECT 権限も必要です。

内容と備考

マルチバイト文字セットを含むデータベースでは、全角カンマや全角スペースなどの一部の句読表記文字やスペース文字は、英数字として処理される場合があります。

♦ タスク

1. Interactive SQL を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Interactive SQL]** をクリックします。

2. **[接続]** ウィンドウで、次のフィールドに入力します。
   a. **[認証]** ドロップダウンリストで **[データベース]** をクリックします。
   b. **[ユーザ ID]** フィールドに、**DBA** と入力します。
   c. **[パスワード]** フィールドに、**sql** と入力します。
   d. **[アクション]** ドロップダウンリストで、**[ODBC データソースを使用した接続]** を選択します。

3. 次の文を実行して、myNcharNGRAMTextConfig という NCHAR テキスト設定オブジェクトを作成します。

```
CREATE TEXT CONFIGURATION myNcharNGRAMTextConfig FROM default_nchar;
```

4. 次の文を実行して、TERM BREAKER アルゴリズムを NGRAM に変更し、MAXIMUM TERM LENGTH を 2 に設定します。

```
ALTER TEXT CONFIGURATION myNcharNGRAMTextConfig
  TERM BREAKER NGRAM;
ALTER TEXT CONFIGURATION myNcharNGRAMTextConfig
  MAXIMUM TERM LENGTH 2;
```

   中国語、日本語、韓国語データの場合は、N の値を 2 または 3 に設定することをおすすめします。検索語が 1 文字または 2 文字以内の場合は、N の値を 1 に設定します。クエリが長い場合に N の値を 1 に設定すると、実行に時間がかかる可能性があります。

5. Sybase Central を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Sybase Central]** をクリックします。

6. **[接続]** » **[SQL Anywhere 16 に接続]** をクリックします。

7. **[接続]** ウィンドウで、次のフィールドに入力します。
   a. **[認証]** ドロップダウンリストで **[データベース]** をクリックします。
   b. **[ユーザ ID]** フィールドに、**DBA** と入力します。

c. **[パスワード]** フィールドに、**sql** と入力します。

d. **[アクション]** ドロップダウンリストで、**[ODBC データソースを使用した接続]** を選択します。

8. MarketingInformation テーブルのコピーを作成します。

   a. **[テーブル]** フォルダを展開します。

   b. **[MarketingInformation]** を右クリックし、**[コピー]** をクリックします。

   c. **[テーブル]** フォルダを右クリックし、**[貼り付け]** をクリックします。

   d. **[名前]** フィールドに **MarketingInformationNgram** と入力します。

   e. **[OK]** をクリックします。

9. Interactive SQL で、次の文を実行して、MarketingInformationNgram テーブルにデータを追加します。

```
INSERT INTO MarketingInformationNgram
  SELECT * FROM GROUPO.MarketingInformation;
COMMIT;
```

10. 次の文を実行して、myNcharNGRAMTextConfig テキスト設定オブジェクトを使用することで、MarketingInformationNgram テーブルの Description カラムに IMMEDIATE REFRESH テキストインデックスを作成します。

```
CREATE TEXT INDEX ncharNGRAMTextIndex
  ON MarketingInformationNgram( Description )
    CONFIGURATION myNcharNGRAMTextConfig;
```

11. テキストインデックスをテストします。

   a. 次の文により、2-GRAM のテキストインデックスで **sw** を含む単語が検索されます。結果はスコアの降順でソートされます。

```
SELECT M.Description, ct.*
  FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 'sw' ) ct
  ORDER BY ct.score DESC;
```

Description	Score
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>「Sw」eatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded 「Sw」eatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	2.262071918398649

Description	Score
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>「Sw」eatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean. </body></html>	1.5556043490424176

b. 次の文は、**ams** を含む単語を検索します。結果はスコアの降順でソートされます。

```
SELECT M.Description, ct.*
  FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 'ams' ) ct
  ORDER BY ct.score DESC;
```

2-GRAM テキストインデックスでは、前述の文はセマンティック上、次の文と同義になります。

```
SELECT M.Description, ct.*
  FROM MarketingInformationNgram AS M
CONTAINS( M.Description, '"am ms"' ) ct
  ORDER BY ct.score DESC;
```

どちらの文も次の結果を返します。

Description	Score
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded sweatshirt with taped neck se「ams」. Comes pre-washed for softness and to lessen shrinkage. </body></html>	1.6619019465461564

Description	Score
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched se「ams」 for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean. </body></html>	1.5556043490424176

c. 次の文は、**v** の後に任意の英数字が続く単語を検索します。インデックス付けされたデータで、**ve** の出現する頻度の方が高いため、2-GRAM の **ve** を含むローのスコアは、**vi** を含むローよりも低くなります。結果はスコアの降順でソートされます。

```
SELECT M.ID, M.Description, ct.*
  FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 'v*' ) ct
  ORDER BY ct.score DESC;
```

ID	Description	Score
901	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> We'「ve」 impro「ve」d the design of this perennial fav「vo」rite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>	3.3416789108071976

ID	Description	Score
907	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>「Vi」sor</title></head><body lang=EN-US> A polycarbonate「vi」sor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to gi「ve」 you a snug yet comfortable fit e「ve」ry time you wear it. </body></html>	2.1123084896159376
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side「ve」nts for breathable comfort during aerobic acti「vi」ties. Moisture-absorbing headband liner. </body></html>	1.6750365447462499

 Copyright © 2013, SAP AG or an SAP affiliate company.

ID	Description	Score
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts pro「vi」de all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	0.9244136988525732
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>「Vi」sor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	0.9134171046194403

ID	Description	Score
904	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier tra「ve」l, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	0.7313071661212746
903	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will ser「ve」 you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>	0.6799436746197272

d. 次の文は、各ローで v を含む任意の単語を検索します。2 番目の文の後、変数には文字列 av OR ev OR iv OR ov OR rv OR ve OR vi OR vo が含まれます。結果はスコアの降順でソートされます。インデックス付けされたすべてのローに N-gram があると、スコアは 0 になります。

これは、ホワイトスペースや英数字以外の文字の前にある 1 文字を検索するための唯一の方法です。

```
CREATE VARIABLE query NVARCHAR (100);
SELECT LIST (term, ' OR ' )
INTO query
  FROM sa_text_index_vocab_nchar( 'ncharNGRAMTextIndex', 'MarketingInformationNgram',
'dba' )
```

```
  WHERE term LIKE '%v%';
SELECT M.ID, M.Description, ct.*
  FROM MarketingInformationNgram AS M
  CONTAINS( M.Description, query ) ct
  ORDER BY ct.score DESC;
```

ID	Description	Score
901	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> We'「ve」impr「ove」d the design of this perennial f「avo」rite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>	6.654350268810443
907	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>「Vi」sor</title></head><body lang=EN-US> A polycarbonate「vi」sor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to g「ive」 you a snug yet comfortable fit 「eve」ry time you wear it. </body></html>	4.265623837817126

ID	Description	Score
903	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will se「rve」 you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>	2.9386676702799504
910	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts pr「ovi」de all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	2.5481193655722336

ID	Description	Score
904	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier tr「ave」l, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	2.4293498211307214
905	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side「ve」nts for breathable comfort during aerobic act「ivi」ties. Moisture-absorbing headband liner. </body></html>	1.6750365447462499

ID	Description	Score
906	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>「Vi」sor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	0.9134171046194403
902	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>	0

 Copyright © 2013, SAP AG or an SAP affiliate company.

ID	Description	Score
908	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</ title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </ body></html>	0
909	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</ title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean.</ span> </body></html>	0

e. 次の文は、Description カラムで **ea**、**ka**、**ki** を含むローを検索します。

```
SELECT M.ID, M.Description, ct.*
  FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 'ea ka ki' ) ct
 ORDER BY ct.score DESC;
```

ID	Description	Score
904	<html><h「ea」d><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></h「ea」d><body lang=EN-US> This fashionable hat is id「ea」l for glacier travel, s「ea」-「ka」yaking, and hi「ki」ng. With conc「ea」led draw cord for windy days. </body></html>	3.4151032739119733

f. 次の文は、Description カラムで **ve** と **vi** を含み、**gg** は含まないローを検索します。

```
SELECT M.ID, M.Description, ct.*
  FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 've & vi -gg' ) ct
  ORDER BY ct.score DESC;
```

ID	Description	Score
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side「ve」nts for breathable comfort during aerobic acti「vi」ties. Moisture-absorbing headband liner. </body></html>	1.6750365447462499

12. Interactive SQL と Sybase Central を閉じます。

結果

NGRAM テキストインデックスの全文検索が実行されました。

次の手順

(省略可) サンプルデータベース (*demo.db*) を元の状態にリストアします。「サンプルデータベースの再作成 (demo.db)」『SQL Anywhere 16 紹介』を参照してください。

参照

- 「テキスト設定オブジェクトの概念と参照」395 ページ
- 「CREATE TEXT CONFIGURATION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TEXT CONFIGURATION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「テキストインデックスの概念と参照」408 ページ
- 「CREATE TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』

高度：全文検索での単語の削除

テキストインデックスは、テキストインデックスの作成に使用されるテキスト設定オブジェクトに定義された設定に従って、構築されます。次の条件のうち 1 つ以上が当てはまる場合、単語はテキストインデックスに含まれません。

- 単語がストップリストに含まれる
- 単語が単語の最小長より短い (GENERIC のみ)
- 単語が単語の最大長より長い

同じ規則はクエリ文字列にも適用されます。削除された単語は、フレーズの末尾または先頭で 0 個以上の単語と一致できます。たとえば、'the' という単語がストップリストに含まれているとします。

- 単語が AND、OR、または NEAR のいずれかの側にある場合、演算子と単語の両方が削除されます。たとえば、'the AND apple'、'the OR apple'、'the NEAR apple' を検索することは、'apple' を検索することと同義になります。
- 単語が AND NOT の右側にある場合、AND NOT と単語の両方が削除されます。たとえば、'apple AND NOT the' を検索することは、'apple' を検索することと同義になります。

  単語が AND NOT の左側にある場合は、式全体が削除され、ローは返されません。例 : 'orange and the AND NOT apple' = 'orange'
- 単語がフレーズ内にある場合、そのフレーズは削除された単語の位置で、単語と一致させることができます。たとえば、'feed the dog' の検索は、'feed the dog'、'feed my dog'、'feed any dog' などに一致します。

検索している単語がすべてテキストインデックスに含まれていない場合は、ローが返されません。たとえば、'the' と 'a' の両方がストップリストにあるとします。'a OR the' の検索では、ローが返されません。

参照

- 「テキスト設定オブジェクトの概念と参照」395 ページ

高度：外部単語区切りライブラリと事前フィルタライブラリ

外部単語区切りまたは事前フィルタライブラリを使用する理由

SQL Anywhere での全文検索はテキストインデックスを使用して実行されます。テキストインデックスが構築されたカラムの各値は、「ドキュメント」と呼ばれます。テキストインデックスが作成されると、各ドキュメントはテキストインデックスのテキスト設定で指定された、組み込みの単語区切りによって処理され、ドキュメントに含まれている「単語」 (「トークン」とも呼ばれます) およびドキュメント内の単語の位置が確認されます。組み込みの単語区切りは、クエリ文字列のドキュメント (テキストコンポーネント) に対する単語区切りの実行にも使用されます。たとえば、クエリ文字列 'rain or shine' は、OR 演算子で連結された 2 つのドキュメント 'rain' と 'shine 'で構成されています。テキスト設定で指定された組み込みの単語区切りのアルゴリズムは、ストップリストを単語に分割したり、sa_char_terms system システムプロシージャの入力を単語に分割するのにも使用されます。

アプリケーションのニーズによっても異なりますが、組み込みの GENERIC 単語区切りの一部の動作が望ましくないか、または限定的であったり、NGRAM 単語区切りがアプリケーションのニーズに合っていない場合もあります。たとえば、組み込みの GENERIC 単語区切りでは、言語固有の単語区切りは行われません。カスタムの「単語区切り」を実装する必要があるその他の理由を次に示します。

- **言語固有の単語区切りがない**　単語の構成内容に関する言語ルールは言語ごとに異なります。このため、単語区切りのルールは言語ごとに異なります。組み込みの単語区切りには、言語固有の単語区切りのルールは用意されていません。

- **アポストロフィを含む単語の処理**　"they'll" という単語は組み込みの GENERIC 単語区切りによって "they ll" として扱われます。ただし、アポストロフィを単語の一部として扱うカスタムの GENERIC 単語区切りを作成できます。

- **単語の置き換えがサポートされていない**　単語の置き換えを指定することはできません。たとえば、"they'll" という単語をインデックス付けする場合、they および will の 2 つの単語として保存する場合があります。同様に、単語の置き換えを使用すると、大文字と小文字が区別されるデータベースで大文字と小文字を区別しない検索を実行できます。

SQL Anywhere では、外部事前フィルタライブラリを使用して、データに「事前フィルタリング」を実行してからインデックス付けすることもできます。事前フィルタリングでは、インデックス付けするテキスト形式の内容のみをドキュメントから抽出できます。たとえば、XML 値を含むカラムにテキストインデックスを作成するとします。事前フィルタによって、内容にインデックス付けされないように XML タグを除外できます。

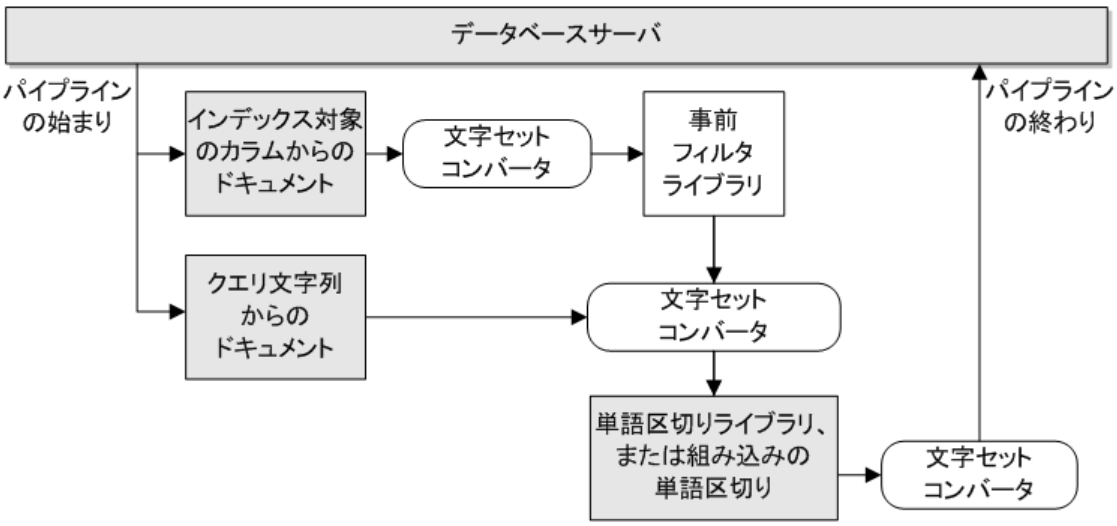
SQL Anywhere では、テキストインデックス全体の作成や更新を行うときに、カスタムの事前フィルタ、およびサードパーティの事前フィルタや単語区切りライブラリにアクセスするのに使用できる API が提供されています。このことは、外部ライブラリを使用して XML、PDF、Word のようなドキュメントフォーマットを処理したり、内容にインデックスを付ける前に不要な単語と内容を削除できることを意味します。

サンプルの事前フィルタライブラリと単語区切りライブラリを *Samples* ディレクトリにインクルードしておき、独自のライブラリの作成やサードパーティのライブラリの使用に役立てることができます。データベースサーバを実行しているシステムに Microsoft Office がインストールされている場合は、Word や Excel などの Office ドキュメント用の IFilter を使用できます。サーバに Acrobat Reader がインストールされている場合は、PDF IFilter を使用できます。

注意
NGRAM の外部単語区切りはサポートされていません。

全文パイプラインのワークフロー

次の図は、SQL Anywhere 内でインデックス付けするためにドキュメントから単語のストリームへデータが変換される方法を示しています。テキストインデックスを作成し、更新し、問い合わせるワークフローは、「パイプライン」と呼ばれます。パイプラインの必須部分は明るいグレーで表されています。矢印はパイプラインでのデータのフローを表しています。関数呼び出しは反対方向に伝達されます。



パイプラインの動作方法の概要

1. 各ドキュメントの処理は、データベースサーバがパイプラインの終わり (単語区切りまたは文字セットコンバーターのどちらか) で begin_document メソッドを呼び出すことにより開始されます。パイプライン内の各コンポーネントは、自身の begin_document の起動から戻る前に自身のプロデューサーにより begin_document を呼び出します。

2. データベースサーバは begin_document が正常に完了した後、パイプラインの終わりで get_words を呼び出します。

   - get_words の実行中に、単語区切りはプロデューサーにより get_next_piece を呼び出し、処理するデータを取得します。パイプラインに事前フィルタが存在する場合、データは get_next_piece 呼び出しの間にフィルタされます。

- 単語区切りはプロデューサーから受け取ったデータを単語区切りのルールに従って単語に分割します。

3. データベースサーバは get_words 呼び出しから返された単語に、ストップリスト制限だけでなく、最小単語長と最大単語長の設定も適用します。

4. データベースサーバは単語が返されなくなるまで get_words を呼び出し続けます。単語が返されなくなった時点で、データベースサーバは end_document を呼び出します。この呼び出しは、パイプライン内で begin_document 呼び出しと同じ方法で伝達されます。

注意
文字セットコンバーターは、必要に応じてデータベースサーバによってパイプラインに透過的に追加されます。

事前フィルタと単語区切りのコードサンプル

SQL Anywhere インストールの *ExternalLibrariesFullText* ディレクトリには、参照用に、事前フィルタと単語区切りのサンプルコードが含まれています。このディレクトリは、*Samples* ディレクトリ内にあります。*Samples* ディレクトリのロケーションについては、「サンプルディレクトリ」『SQL Anywhere サーバ データベース管理』を参照してください。

参照

- 外部事前フィルタライブラリのワークフロー445 ページ
- 外部単語区切りライブラリのワークフロー448 ページ

外部事前フィルタライブラリ

外部事前フィルタを使用するよう SQL Anywhere を設定する方法

SQL Anywhere には組み込みの事前フィルタアルゴリズムはありません。データが外部事前フィルタライブラリを経由するようにするには、次のように ALTER TEXT CONFIGURATION 文を使用してライブラリとそのエントリポイント関数を指定します。

```
ALTER TEXT CONFIGURATION my_text_config
  PREFILTER EXTERNAL NAME 'my_prefilter@myprefilterLibrary.dll'
```

この例は、my_text_config テキスト設定オブジェクトを使用してテキストインデックスを構築または更新するときに使用する事前フィルタインスタンスを、*myprefilterLibrary.dll* ライブラリの my_prefilter エントリポイント関数を使用して取得するようにデータベースサーバに指示します。

参照

- 「ALTER TEXT CONFIGURATION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「a_text_source interface」454 ページ

外部事前フィルタライブラリを設計する方法

事前フィルタライブラリは C/C++ で実装され、次の条件を満たしている必要があります。

- 事前フィルタインタフェース定義ヘッダファイル *extpfapiv1.h* を含みます。
- a_text_source インタフェースを実装します。
- a_text_source (事前フィルタ) のインスタンスおよび事前フィルタでサポートされている文字セットのラベルを初期化し、返すエントリポイント関数があります。

事前フィルタの呼び出しシーケンス

次の呼び出しシーケンスは、事前フィルタのコンシューマによって、処理されている各ドキュメントに対して実行されます。

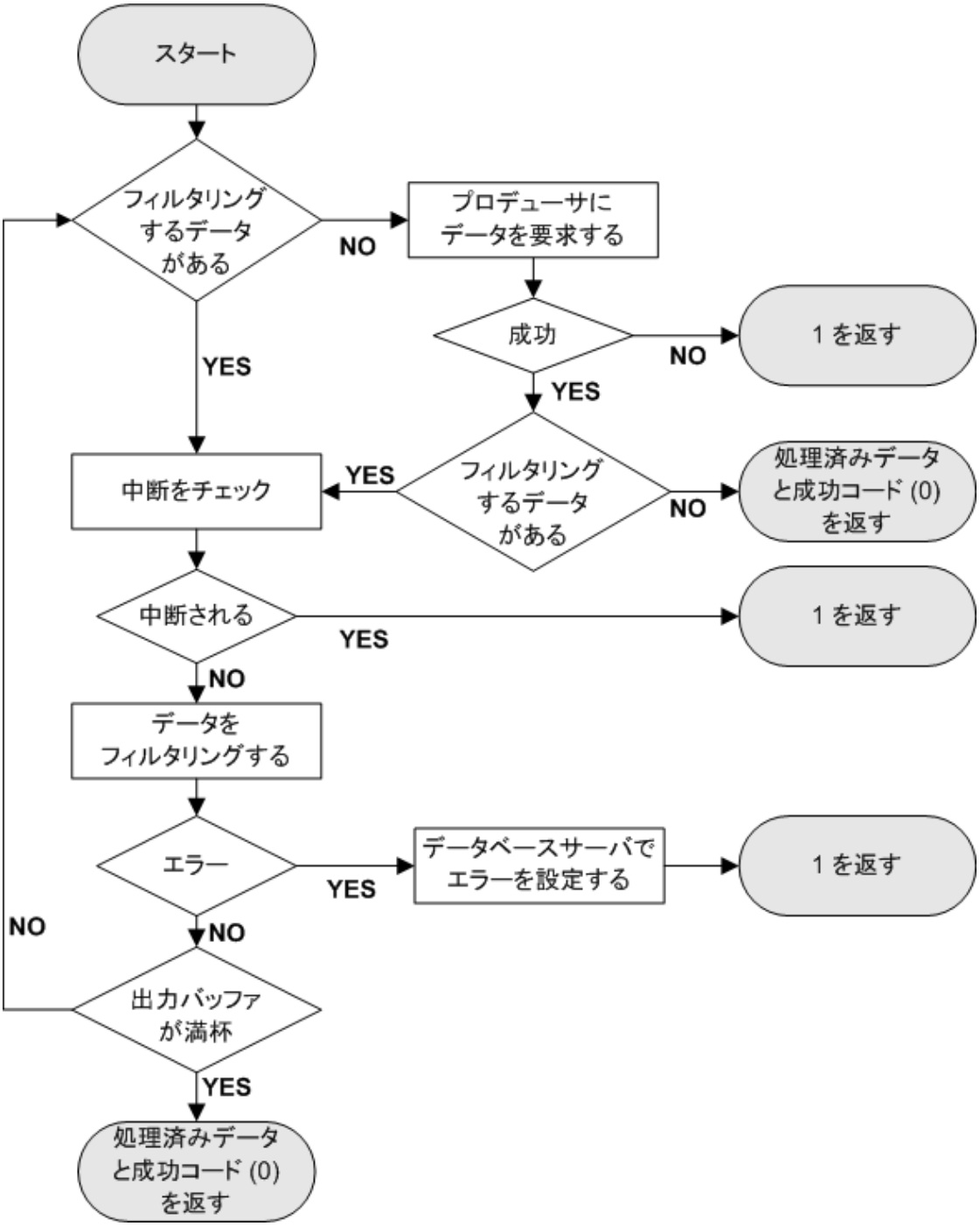
```
begin_document(a_text_source*)
get_next_piece(a_text_source*, buffer**, len*)
get_next_piece(a_text_source*, buffer**, len*)
...
end_document(a_text_source*)
```

注意
end_document は、間に begin_document 呼び出しを挟まなくても複数回呼び出すことができます。たとえば、インデックス付けされるドキュメントの 1 つが空の場合、データベースサーバでは、begin_document を呼び出さずにこのドキュメントの end_document を呼び出すことがあります。

get_next_piece 関数はフォーマット情報や受信バイトストリームからのイメージなどの不要なデータを除外し、自己割り付けバッファ内のフィルタされたデータの次のチャンクを返す必要があります。

外部事前フィルタライブラリのワークフロー

次のフローチャートは、get_next_piece 関数が呼び出されたときの論理フローを示しています。


スタート
フィルタリングするデータがある
NO
プロデューサにデータを要求する
成功
NO
1 を返す
YES
YES
中断をチェック
YES
フィルタリングするデータがある
NO
処理済みデータと成功コード (0) を返す
中断される
YES
1 を返す
NO
データをフィルタリングする
エラー
YES
データベースサーバでエラーを設定する
1 を返す
NO
NO
出力バッファが満杯
YES
処理済みデータと成功コード (0) を返す

参照

- 「a_text_source interface」454 ページ
- 「全文パイプラインのワークフロー」443 ページ
- 事前フィルタと単語区切りのコードサンプル 444 ページ

外部単語区切りライブラリ

外部単語区切りを使用するよう SQL Anywhere を設定する方法

デフォルトでは、テキスト設定オブジェクトを作成すると、そのテキスト設定オブジェクトに関連付けられたデータには組み込みの単語区切りが使用されます。データが外部単語区切りライブラリを経由するようにするには、次のように ALTER TEXT CONFIGURATION 文を使用してライブラリとそのエントリポイント関数を指定します。

```
ALTER TEXT CONFIGURATION my_text_config
  TERM BREAKER GENERIC EXTERNAL NAME 'my_termbreaker@termbreaker'
```

この例は、my_text_config テキスト設定オブジェクトに関連付けられたテキストインデックスを構築、更新、または問い合わせるとき、テキスト設定オブジェクトのストップリストの解析時、sa_char_terms システムプロシージャへの入力の処理時に使用する単語区切りインスタンスを、単語区切りライブラリの my_termbreaker エントリポイント関数を使用して取得するようにデータベースサーバに指示します。

参照

- 「ALTER TEXT CONFIGURATION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「a_word_source インタフェース」459 ページ

外部単語区切りライブラリを設計する方法

外部単語区切りライブラリは C/C++ で実装され、次の条件を満たしている必要があります。

- 単語区切りインタフェース定義ヘッダファイル *exttbapiv1.h* を含みます。
- a_word_source インタフェースを実装します。
- a_word_source (単語区切り) のインスタンスおよび単語区切りでサポートされている文字セットのラベルを初期化し、返すエントリポイント関数があります。

単語区切りの呼び出しシーケンス

次の呼び出しシーケンスは、単語区切りのコンシューマによって、処理されている各ドキュメントに対して実行されます。

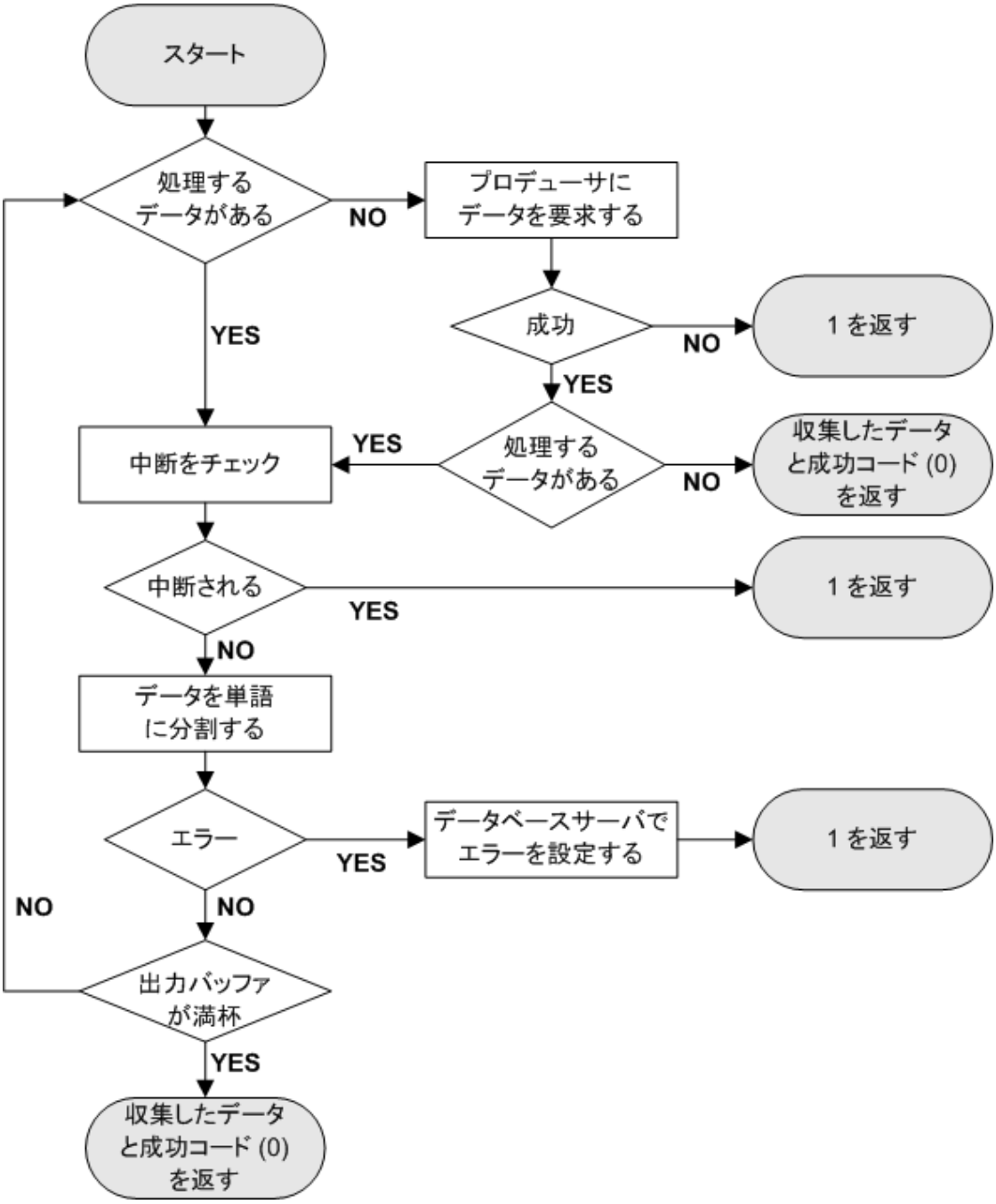
```
begin_document(a_word_source*, asql_uint32);
get_words(a_word_source*, a_term**, uint32 *num_words)
get_words(a_word_source*, a_term**, uint32 *num_words)
...
end_document(a_word_source*)
```

get_words 関数は、a_term 構造体の配列が満たされるか、または処理するデータがなくなるまで、データが単語に分割されるようにプロデューサーで get_next_piece を呼び出す必要があります。

> **注意**
> end_document は、間に begin_document 呼び出しを挟まなくても複数回呼び出すことができます。たとえば、インデックス付けされるドキュメントの 1 つが空の場合、データベースサーバでは、begin_document を呼び出さずにこのドキュメントの end_document を呼び出すことがあります。

外部単語区切りライブラリのワークフロー

次のフローチャートは、get_words 関数が呼び出されたときの論理フローを示しています。


スタート
処理する
データがある
NO
プロデューサに
データを要求する
成功
NO
1 を返す
YES
YES
処理する
データがある
NO
収集したデータ
と成功コード (0)
を返す
YES
中断をチェック
中断される
YES
1 を返す
NO
データを単語
に分割する
エラー
YES
データベースサーバで
エラーを設定する
1 を返す
NO
NO
出力バッファ
が満杯
YES
収集したデータ
と成功コード (0)
を返す

参照

- 「a_word_source インタフェース」459 ページ
- 「a_text_source interface」454 ページ
- 「a_term 構造体」463 ページ
- 「全文パイプラインのワークフロー」443 ページ
- 事前フィルタと単語区切りのコードサンプル 444 ページ

高度：外部全文ライブラリの API

テキストインデックスで事前フィルタまたは単語区切りの外部ライブラリを作成し使用するには、次の手順を完了する必要があります。

- SQL Anywhere C/C++ インタフェースの実装。
- 上記の手順で作成したコードをコンパイルおよびリンクして、動的にロードできるライブラリを作成します。
- データベースにテキスト設定オブジェクトを作成し、事前フィルタか単語区切りまたはその両方の外部ライブラリ内のエントリポイント関数を指定するように変更します。

  エントリポイント関数は、基本となるドキュメント (カラム値) が変更されるときテキストインデックスの挿入/削除中に使用される事前フィルタおよび単語区切りのオブジェクトを取得するために使用されます。外部単語区切りライブラリの場合、エントリポイント関数は、語区切りを使用するテキストインデックスに対するクエリの解析にも使用されます。

a_server_context 構造体

複数のコールバックがデータベースサーバによってサポートされ、次のタスクを実行するために a_server_context 構造体を介して全文外部ライブラリに送られます。

- エラーレポート
- 中断処理
- メッセージのロギング

構文

```
typedef struct a_server_context {
   void (SQL_CALLBACK *set_error)( a_server_context  *this
                                 , a_sql_uint32     error_code
                                 , const char       *error_string
                                 , short            error_string_length
                                 );
   a_sql_uint32 (SQL_CALLBACK *get_is_cancelled)( a_server_context *this
                                 );
   void (SQL_CALLBACK *log_message)( a_server_context *this
                                 , const char       *message_string
                                 , short            message_string_length
                                 );
   void *_context;
} a_server_context, *p_server_context;
```

メンバー

メンバー名	型	説明
set_error	void	このメソッドを使用すると、外部事前フィルタおよび単語区切りによって、エラーコードとエラー文字列を指定してデータベースサーバにエラーを設定できます。データベースサーバは現在の操作をロールバックし、次の形式でユーザにエラーコードと文字列を返します。 "Error from external library: -<error_code>: <error_string>" error_code には 17000 よりも大きい正の整数を指定します。 error_string には NULL で終了する文字列を指定します。 str_len には error_string の長さをバイト単位で指定します。
get_is_cancelled	a_sql_uint32	外部事前フィルタおよび単語区切りでこのメソッドを定期的に呼び出して、現在の操作が中断されているかどうかをチェックする必要があります。このメソッドでは、現在の操作が中断された場合は 1 が返され、中断されていない場合は 0 が返されます。1 が返された場合、呼び出し元は以降の処理を停止し、すぐに戻る必要があります。

メンバー名	型	説明
log_message	void	このメソッドを使用すると、外部事前フィルタおよび単語区切りによって、データベースサーバログにメッセージを記録できます。 *message* には NULL で終了する文字列を指定します。 msg_len にはメッセージの長さをバイト単位で指定します。
_context	void	内部で使用されます。データベースサーバコンテキストへのポインタ。

備考

a_server_context 構造体は、SQL Anywhere インストールディレクトリの *SDK¥Include* サブフォルダのヘッダファイル *exttxtcmn.h* によって定義されます。

a_server_context 構造体によって公開されるメソッドを呼び出す場合は、オペレーティングシステムの同期プリミティブを外部ライブラリで使用しないようにしてください。

a_init_pre_filter 構造体

外部事前フィルタエントリポイント関数のインスタンスの入力要件および出力要件をネゴシエートするために使用される構造体。この構造体は、事前フィルタエントリポイント関数にパラメータとして渡されます。

構文

```
typedef struct a_init_pre_filter {
   a_text_source          *in_text_source;   /* IN */
   a_text_source          *out_text_source;  /* OUT */
   const char             *desired_charset;  /* IN */
   char                   *actual_charset;   /* OUT */
   short                  is_binary;         /* IN */
} a_init_pre_filter;
```

メンバー

名前	型	説明
in_text_source	a_text_source *	作成する外部事前フィルタ (a_text_source オブジェクト) のプロデューサーへのポインタ。事前フィルタエントリポイント関数の呼び出し元によって指定されます。
out_text_source	a_text_source *	事前フィルタエントリポイント関数によって指定される外部事前フィルタ (a_text_source オブジェクト) へのポインタ。
desired_charset	const char *	エントリポイント関数の呼び出し元が、事前フィルタの出力があると期待する文字セット。is_binary フラグが 0 の場合、特にネゴシエートされないかぎり、事前フィルタへの入力の文字セットでもあります。
actual_charset	char *	外部事前フィルタライブラリが出力を生成する、(外部ライブラリによってネゴシエーションの一部として指定された) 文字セット。is_binary が 0 の場合、事前フィルタへの入力の実際の文字セットでもあります。可能な限り、ライブラリで desired_charset のデータの受け入れと作成を実行します。
is_binary	short	入力データがバイナリ (1) であるか、desired_charset (0) であるかを示します。データがバイナリの場合、データベースサーバはパイプラインでの事前フィルタの前に文字セットの変換を通知しません。

備考

a_init_pre_filter 構造体は、SQL Anywhere インストールディレクトリの *SDK¥Include* サブフォルダのヘッダファイル *extpfapiv1.h* によって定義されます。

参照

- 「a_text_source interface」454 ページ
- 「a_word_source インタフェース」459 ページ
- 「事前フィルタエントリポイント関数」466 ページ

a_text_source interface

外部事前フィルタライブラリが、全文インデックスのデータ設定または更新のためにドキュメントの事前フィルタリングを実行する場合に実装する必要があるインタフェース。

構文

```
typedef struct a_text_source {
   a_sql_uint32 ( SQL_CALLBACK *begin_document )( a_text_source *This );
   a_sql_uint32 ( SQL_CALLBACK *get_next_piece )(
                          a_text_source *This
                          , unsigned char ** buffer
                          , a_sql_uint32* len );
   a_sql_uint32 ( SQL_CALLBACK *end_document )( a_text_source *This);
   a_sql_uint64 ( SQL_CALLBACK *get_document_size )( a_text_source *This );
   a_sql_uint32 ( SQL_CALLBACK *fini_all )( a_text_source *This );
   a_server_context     *_context;

   // Only one of the following two members can have a valid pointer in a given implementation.
   // These members point to the current module's producer
   a_text_source        *_my_text_producer;
   a_word_source        *_my_word_producer;

   // Following members have been reserved for
   // future use ONLY
   a_text_source        *_my_text_consumer;
   a_word_source        *_my_word_consumer;
} a_text_source, *p_text_source;
```

メンバー

メンバー	型	説明
begin_document	a_sql_uint32	ドキュメントの処理に必要なセットアップ手順を実行します。このメソッドは、成功の場合は 0 を返し、エラーが発生したか、使用可能なドキュメントがなくなった場合は 1 を返します。

メンバー	型	説明
get_next_piece	a_sql_uint32	フィルタされた入力バイトストリームのフラグメントをフラグメントの長さとともに返します。このメソッドは、指定されたドキュメントに対して複数回呼び出され、ドキュメントのすべての入力データが消費されるか、エラーが発生するまで、各呼び出しでドキュメントの後続のチャンクを返す必要があります。 バッファは、生成されたデータを指すように事前フィルタによって入力される OUT パラメータです。メモリは事前フィルタによって管理されます。 len は、生成されたデータの長さを示す OUT パラメータです。
end_document	a_sql_uint32	指定されたドキュメントに対するフィルタリングの完了を示し、必要に応じてドキュメント固有のクリーンアップを実行します。
get_document_size	a_sql_uint64	事前フィルタによって生成されたドキュメントの合計長 (バイト単位) を返します。a_text_source オブジェクトは、現在のドキュメントのこれまでに生成されたバイトの合計数の現在の数を保持する必要があります。
fini_all	a_sql_uint32	すべてのドキュメントの処理が完了し、パイプラインが閉じられようとしているときにデータベースサーバによって呼び出されます。fini_all では最後のクリーンアップ手順が実行されます。

メンバー	型	説明
_context	a_server_context *	このメンバーを使用して、a_init_pre_filter 構造体内のエントリポイント関数に提供されたデータベースサーバのコンテキストを保持します。事前フィルタモジュールではこのコンテキストを使用してデータベースサーバとの直接通信を確立します。
_my_text_producer	a_text_source *	このメンバーを使用して、a_init_pre_filter 構造体内のエントリポイント関数に提供された事前フィルタの a_text_source プロデューサーへのポインタを保存します。このポインタは、文字セット変換が必要な場合は、エントリポイント関数が実行された後に、データベースサーバによって置き換えられる可能性があります。このため、テキストプロデューサーへのこのポインタのみが事前フィルタによって使用されます。
_my_word_producer	a_word_source *	今後の使用のために予約されており、NULL に初期化する必要があります。
_my_text_consumer	a_text_source *	今後の使用のために予約されており、NULL に初期化する必要があります。
_my_word_consumer	a_word_source *	今後の使用のために予約されており、NULL に初期化する必要があります。

備考

a_text_source インタフェースはストリームベースのデータです。データはプロデューサーから順番に引き出され、各バイトは 1 回だけ表示されます。

a_text_source インタフェースは、SQL Anywhere インストールディレクトリの *SDK¥Include* サブフォルダのヘッダファイル *extpfapiv1.h* によって定義されます。

複数の関数の呼び出しにまたがるオペレーティングシステムの同期プリミティブは、外部ライブラリに保持しないようにしてください。

参照

- 「a_server_context 構造体」450 ページ
- 「a_init_pre_filter 構造体」452 ページ
- 「事前フィルタエントリポイント関数」466 ページ

a_init_term_breaker 構造体

外部単語区切りのインスタンスの入力要件および出力要件をネゴシエートするために使用される構造体。この構造体は、単語区切りエントリポイント関数にパラメータとして渡されます。

構文

```
typedef struct a_init_term_breaker
{
    a_text_source       *in_text_source;
    const char          *desired_charset;
    a_word_source       *out_word_source;
    char                *actual_charset;
    a_term_breaker_for  term_breaker_for;
} a_init_term_breaker, *p_init_term_breaker;
```

メンバー

メンバー	型	説明
in_text_source	a_text_source *	作成する外部単語区切り (a_text_source オブジェクト) のプロデューサーへのポインタ。
out_word_source	a_word_source *	エントリポイント関数によって指定される外部単語区切り (a_word_source オブジェクト) へのポインタ。
desired_charset	const char *	エントリポイント関数の呼び出し元が、単語区切りの出力があると期待する文字セット。is_binary フラグが 0 の場合、特にネゴシエートされないかぎり、単語区切りへの入力の文字セットでもあります。

メンバー	型	説明
actual_charset	char *	外部単語区切りライブラリが出力を生成する、(外部ライブラリによってネゴシエーションの一部として指定された) 文字セット。is_binary が 0 の場合、単語区切りへの入力の実際の文字セットでもあります。可能な限り、ライブラリで desired_charset のデータの受け入れと作成を実行します。
term_breaker_for	a_term_breaker_for	単語区切りを初期化する目的： ● **TERM_BREAKER_FOR_LOAD** テキストインデックスに対する作成操作、挿入操作、更新操作、および削除操作に使用されます。事前フィルタが指定された場合、入力は事前フィルタリングされる可能性があります。 ● **TERM_BREAKER_FOR_QUERY** sa_char_term システムプロシージャへのクエリ要素、ストップリスト、および入力の解析に使用されます。TERM_BREAKER_FOR_QUERY の場合、テキストインデックスに対して外部事前フィルタライブラリが指定されていても、事前フィルタリングは行われません。

備考

a_init_term_breaker 構造体は、SQL Anywhere インストールディレクトリの *SDK¥Include* サブフォルダのヘッダファイル *exttbapiv1.h* によって定義されます。

参照

- 「sa_char_terms システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「a_term_breaker_for 列挙」459 ページ
- 「単語区切りエントリポイント関数」467 ページ
- 「a_text_source interface」454 ページ
- 「a_word_source インタフェース」459 ページ

a_term_breaker_for 列挙

パイプラインをテキストインデックスの更新中に使用するために構築するか、テキストインデックスの問い合わせに使用するために構築するかを指定する場合に使用します。

パラメータ

- **TERM_BREAKER_FOR_LOAD** テキストインデックスに対する作成操作、挿入操作、更新操作、および削除操作に使用されます。

- **TERM_BREAKER_FOR_QUERY** sa_char_term システムプロシージャへのクエリ要素、ストップリスト、および入力の解析に使用されます。TERM_BREAKER_FOR_QUERY の場合、テキストインデックスに対して外部事前フィルタライブラリが指定されていても、事前フィルタリングは行われません。

備考

データベースサーバは外部単語区切りを初期化するときに a_init_term_breaker::term_breaker_for の値を設定します。

```
typedef enum a_term_breaker_for {
    TERM_BREAKER_FOR_LOAD = 0,
    TERM_BREAKER_FOR_QUERY
} a_term_breaker_for;
```

a_term_breaker_for enumeration 構造体は、SQL Anywhere インストールディレクトリの *SDK ¥Include* サブフォルダのヘッダファイル *exttbapiv1.h* によって定義されます。

参照

- 「a_init_term_breaker 構造体」457 ページ
- 「sa_char_terms システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

a_word_source インタフェース

外部単語区切りライブラリが、テキストインデックス操作の単語区切りを実行する場合に実装する必要があるインタフェース。

構文

```
typedef struct a_word_source {
  a_sql_uint32 ( SQL_CALLBACK *begin_document )(
                        a_word_source *This
                        , a_sql_uint32 has_prefix );
  a_sql_uint32 ( SQL_CALLBACK *get_words )(
```

```
                        a_word_source *This
                        , a_term** words
                        ,a_sql_uint32 *num_words );
    a_sql_uint32 ( SQL_CALLBACK *end_document )(
                        a_word_source *This );
    a_sql_uint32 ( SQL_CALLBACK *fini_all )(
                        a_word_source *This );

    a_server_context    *_context;

    a_text_source       *_my_text_producer;
    a_word_source       *_my_word_producer;

    a_text_source       *_my_text_consumer;
    a_word_source       *_my_word_consumer;
  } a_word_source, *p_word_source;
```

メンバー

メンバー	型	説明
begin_document	a_sql_uint32	ドキュメントの処理に必要なセットアップ手順を実行します。トークン化されているドキュメントがプレフィクスクエリ単語である場合、パラメータ has_prefix は 1、not true、または TRUE に設定されます。has_prefix が TRUE に設定された場合、単語区切りは少なくとも 1 つの単語 (空の可能性があります) を返す必要があります。 パイプラインの初期化の目的が TERM_BREAKER_FOR_QUERY である場合、has_prefix は 1、not true、または TRUE のみに設定できます。 プレフィクスのトークン化の結果は、フレーズの最後の単語が実際のプレフィクス文字列であるフレーズとして扱われます。

メンバー	型	説明
get_words	a_sql_uint32	a_term 構造体の配列へのポインタを返します。このメソッドは、ドキュメントのすべての内容が単語に分解されるまで、指定されたドキュメントのループで呼び出されます。 データベースサーバでは、ドキュメント内の連続する 2 つの単語の位置の差異が 1 であると予測します。単語区切りが独自のストップリスト処理を実行している場合に、2 つの連続する単語間の差異として 1 より大きい値が返される可能性があります。これは予期される動作で、許容可能です。ただし、位置の差異が 1 で連続していない数値の場合、任意の位置は全文クエリの実行方法に影響を与え、以降の全文クエリに予期しない結果をもたらす可能性があります。
end_document	a_sql_uint32	パイプラインによるドキュメントの処理の完了を示し、ドキュメント固有のクリーンアップを実行します。
fini_all	a_sql_uint32	すべてのドキュメントの処理が完了し、パイプラインが閉じられようとしているときにデータベースサーバによって呼び出されます。fini_all では最後のクリーンアップ手順が実行されます。

メンバー	型	説明
_context	a_server_context *	a_init_term_breaker 構造体内のエントリポイント関数に提供されたデータベースサーバのコンテキスト。単語区切りモジュールではこのコンテキストを使用してデータベースサーバとの直接通信を確立します。
_my_text_producer	a_text_source *	a_init_term_breaker 構造体内のエントリポイント関数に提供された単語区切りの a_text_source プロシージャへのポインタ。このポインタは、文字セット変換が必要な場合は、エントリポイント関数が実行された後に、データベースサーバによって置き換えられる可能性があります。このため、テキストプロデューサーへのこのポインタのみが単語区切りによって使用されます。
_my_word_producer	a_word_source *	今後の使用のために予約されており、NULL に初期化する必要があります。
_my_text_consumer	a_text_source *	今後の使用のために予約されており、NULL に初期化する必要があります。
_my_word_consumer	a_word_source *	今後の使用のために予約されており、NULL に初期化する必要があります。

備考

a_word_source インタフェースは、SQL Anywhere インストールディレクトリの *SDK¥Include* サブフォルダのヘッダファイル *exttbapiv1.h* によって定義されます。

複数の関数の呼び出しにまたがるオペレーティングシステムの同期プリミティブは、外部ライブラリに保持しないようにしてください。

参照

- 「a_server_context 構造体」450 ページ
- 「a_term 構造体」463 ページ
- 「a_init_term_breaker 構造体」457 ページ
- 「a_text_source interface」454 ページ
- 「単語区切りエントリポイント関数」467 ページ

a_term 構造体

a_term 構造体は、単語、単語の長さ、単語の位置を保存します。

構文

```
typedef struct a_term
{
   unsigned char  *word;
   a_sql_uint32   len;
   a_sql_uint32   ch_len;
   a_sql_uint64   pos;
} a_term, *p_term;
```

メンバー

メンバー	型	説明
term	unsigned char *	インデックス付けされる単語。
len	a_sql_uint32	単語の長さ (バイト)。
ch_len	a_sql_uint32	単語の長さ (文字数)。

メンバー	型	説明
pos	a_sql_uint64	ドキュメント内の単語の位置。 データベースサーバでは、ドキュメント内の連続する 2 つの単語の位置の差異が 1 であると予測します。単語区切りが独自のストップリスト処理を実行している場合に、2 つの連続する単語間の差異として 1 より大きい値が返される可能性があります。これは予期される動作で、許容可能です。ただし、位置の差異が 1 で連続していない数値の場合、任意の位置は全文クエリの実行方法に影響を与え、以降の全文クエリに予期しない結果をもたらす可能性があります。

備考

各 a_term 構造体は、バイト長、文字長、ドキュメント内の位置による注釈が付けられた単語を表します。

a_term 要素の配列へのポインタは、a_word_source インタフェースの一部として実装された get_words メソッドによって OUT パラメータに返されます。

a_term 構造体は、SQL Anywhere インストールディレクトリの *SDK¥Include* サブフォルダのヘッダファイル *exttbapiv1.h* によって定義されます。

extpf_use_new_api エントリポイント関数 (事前フィルタ)

データベースサーバに、外部事前フィルタライブラリに実装されているインタフェースバージョンについて通知します。現在、バージョン 1 のインタフェースのみがサポートされています。

この関数は外部事前フィルタライブラリに必要です。

構文

extern "C" a_sql_uint32 (extpf_use_new_api)(void);

戻り値

関数は符号なし 32 ビット整数値を返します。戻り値は、*extpfapiv1.h* で定義した EXTPF_V1_API のインタフェースバージョン番号です。

備考

この関数の一般的な実装を次に示します。

```
extern "C" a_sql_uint32 ( extpf_use_new_api )( void )
{
  return EXTPF_V1_API;
}
```

exttb_use_new_api エントリポイント関数 (単語区切り)

外部単語区切りライブラリに実装されているインタフェースバージョンについての情報を提供します。現在、バージョン 1 のインタフェースのみがサポートされています。

この関数は外部単語区切りライブラリに必要です。

構文

extern "C" a_sql_uint32 (exttb_use_new_api)(void);

戻り値

関数は符号なし 32 ビット整数値を返します。戻り値は、*exttbapiv1.h* で定義した EXTTB_V1_API のインタフェースバージョン番号です。

備考

この関数の一般的な実装を次に示します。

```
extern "C" a_sql_uint32 ( exttb_use_new_api )( void )
{
  return EXTTB_V1_API;
}
```

extfn_post_load_library グローバルエントリポイント関数

この関数が実装され外部ライブラリで公開された場合、外部ライブラリがロードされバージョンチェックが実行された後、この関数はデータベースサーバによって実行されます。その後、外部ライブラリ内の定義されたその他の関数が呼び出されます。

この関数は、ライブラリ内のいずれかの関数が呼び出される前にライブラリ全体にわたる設定を行うライブラリ固有の要件がある場合にのみ必要です。

構文

extern "C" void (extfn_post_load_library)(void);

備考

外部の単語区切りライブラリと事前フィルタライブラリの両方でこの関数を実装できます。

参照

- 「extfn_post_load_library メソッド」『SQL Anywhere サーバ プログラミング』

extfn_pre_unload_library グローバルエントリポイント関数

この関数が実装され外部ライブラリで公開された場合、外部ライブラリをアンロードする前にデータベースサーバによってすぐに実行されます。

この関数は、ライブラリがアンロードされる前にライブラリ全体にわたるクリーンアップを行うライブラリ固有の要件がある場合にのみ必要です。

構文

extern "C" void (extfn_pre_unload_library)(void);

備考

外部の単語区切りライブラリと事前フィルタライブラリの両方でこの関数を実装できます。

参照

- 「extfn_pre_unload_library メソッド」『SQL Anywhere サーバ プログラミング』

事前フィルタエントリポイント関数

外部事前フィルタのインスタンスを初期化し、データの文字セットをネゴシエートするエントリポイント関数。

構文

extern "C" a_sql_uint32 (SQL_CALLBACK ****entry-point-function* **)(a_init_pre_filter ****data* **);

戻り値

エラーの場合は 1、正常に実行された場合は 0。

パラメータ

- **entry-point-function**　事前フィルタのエントリポイント関数の名前。
- **data**　a_init_pre_filter 構造体へのポインタ。

備考

この構造体は外部事前フィルタライブラリに実装される必要があります。また、複数のスレッドで同時に実行できるため再入力可能である必要があります。

この関数の呼び出し元 (データベースサーバ) は、事前フィルタのプロデューサーとして機能する a_text_source オブジェクトへのポインタを提供します。呼び出し元は、入力の文字セットも提供します。

この関数は、外部事前フィルタ (a_text_source 構造体) へのポインタを提供します。この関数は、必要に応じて actual_charset フィールドを変更して、入力 (バイナリでない場合) および出力データの文字セットもネゴシエートします。

desired_charset と actual_charset が同じでない場合、data->is_binary フィールドが 1 でないかぎり、データベースサーバは入力データに対して文字セット変換を実行します。is_binary が 0 の場合には、入力データは actual_charset によって指定された文字セットになります。

文字セットの変換が必要になると、パフォーマンスが低下する原因になる可能性があります。

このエントリポイント関数は、ユーザによって ALTER TEXT CONFIGURATION...PREFILTER EXTERNAL NAME を呼び出して指定されます。

参照

- 「a_init_pre_filter 構造体」452 ページ
- 「a_init_pre_filter 構造体」452 ページ
- 「a_text_source interface」454 ページ
- PREFILTER EXTERNAL NAME 句、ALTER TEXT CONFIGURATION 文『SQL Anywhere サーバ SQL リファレンス』

単語区切りエントリポイント関数

外部単語区切りのインスタンスを初期化し、データの文字セットをネゴシエートするエントリポイント関数。

構文

extern "C" a_sql_uint32 (SQL_CALLBACK ****entry-point-function*)(a_init_term_breaker ****data***);**

戻り値

エラーの場合は 1、正常に実行された場合は 0。

パラメータ

- **entry-point-function**　単語区切りのエントリポイント関数の名前。
- **data**　a_init_term_breaker 構造体へのポインタ。

備考

この構造体は外部単語区切りライブラリに実装される必要があります。また、複数のスレッドで同時に実行できるため再入力可能である必要があります。

この関数の呼び出し元は、単語区切りのプロデューサーとして機能する a_text_source オブジェクトへのポインタを提供します。呼び出し元は、入力の文字セットも提供する必要があります。

この関数は、外部単語区切り (a_word_source 構造体) へのポインタおよびサポートされている文字セットを呼び出し元に提供します。

desired_charset と actual_charset が同じでない場合、データベースサーバは actual_charset によって指定された文字セットへの単語区切り入力を変換します。

文字セットの変換は、パフォーマンスが低下する原因になる可能性があります。

参照

- 「a_word_source インタフェース」459 ページ
- 「a_text_source interface」454 ページ
- 「a_init_term_breaker 構造体」457 ページ

クエリ結果の要約、グループ化、ソート

クエリ結果を要約する集合関数

集合関数は、指定されたカラム中の値の要約を表示します。GROUP BY 句、HAVING 句、ORDER BY 句を使用すれば、集合関数を使用してクエリの結果をグループ化およびソートでき、UNION 演算子を使用すれば、クエリ結果を結合できます。

ORDER BY 句に定数が含まれている場合、それらの定数はオプティマイザによって解釈され、同義の ORDER BY 句に置き換えられます。たとえば、オプティマイザは ORDER BY 'a' を ORDER BY 式として解釈します。

クエリブロックに、有効な ORDER BY 句が指定された複数の集合関数が含まれているとき、それらの ORDER BY 句を単一の ORDER BY 句に論理的に結合できる場合は、そのクエリブロックを実行できます。たとえば、次の ORDER BY 句の場合は、

```
ORDER BY expression1, 'a', expression2
```

```
ORDER BY expression1, 'b', expression2, 'c', expression3
```

次の ORDER BY 句として結合されます。

```
ORDER BY expression1, expression2, expression3
```

テーブル内のすべてのロー、WHERE 句によって指定されるテーブルのサブセット、またはテーブル内の 1 つ以上のローグループに、集合関数を適用できます。集合関数を適用するそれぞれのローセットから、SQL Anywhere が単一の値を生成します。

サポートされる集合関数の一部を次に示します。

- **AVG(*expression*)**　返されたローについて提供された式の平均。
- **COUNT(*expression*)**　提供されたグループで、式が NOT NULL のロー数。
- **COUNT(*)**　各グループの中のロー数。
- **LIST(*string-expr*)**　各ローグループの中の *string-expr* に対するすべての値で構成されている、カンマで区切られたリストを含む文字列。
- **MAX(*expression*)**　返されたローの最大値。
- **MIN(*expression*)**　返されたローの最小値。
- **STDDEV(*expression*)**　返されたローの標準偏差。

- **SUM(*expression*)**　返されたローの合計。
- **VARIANCE(*expression*)**　返されたローについての式の分散。

AVG、SUM、LIST、COUNT とともにオプションのキーワード、DISTINCT を使用して、重複した値を削除してから、集合関数を適用できます。

構文が参照する式は、通常はカラム名です。より一般的な式の場合もあります。

たとえば、次の文を使用して、単価に 1 ドル加算した場合の、全製品の平均価格を調べることができます。

```
SELECT AVG ( UnitPrice + 1 )
FROM Products;
```

参照

- 「集合関数」『SQL Anywhere サーバ SQL リファレンス』

例

次のクエリは、Employees テーブルの中の年俸から、支払い給料の総額を計算します。

```
SELECT SUM( Salary )
FROM Employees;
```

集合関数を使用するには、関数名を入力し、その後ろに式を続けます。この式の値に対して、関数が作用します。この式 (この例では Salary カラム) はその関数の引数であり、カッコ内に指定します。

集合関数を使用できる場所

集合関数は、SELECT リストまたはグループ化されたクエリブロックの HAVING 句で使用できます。

WHERE 句や JOIN 条件の中では、集合関数は使用できません。しかし、SELECT リストの集合関数による SELECT クエリブロックには、多くの場合、その集合関数が適用されるローを制限する WHERE 句があります。

GROUP BY 句を含まない SELECT クエリブロックで集合関数を使用すると、その関数がすべてのローで操作される場合でも WHERE 句によって定義されたローのサブセットで操作される場合でも、常に 1 つの値が生成されます。

同一の SELECT リストで 2 つ以上の集合関数を使用でき、単一の SELECT クエリブロックで 2 つ以上の集合関数を作成できます。

参照

- 「HAVING 句：データグループの選択」477 ページ
- 「GROUP BY 句」『SQL Anywhere サーバ SQL リファレンス』

集合関数とデータ型

一部の集合関数は特定の種類のデータにだけ意味を持ちます。たとえば、SUM と AVG は、数値カラムにしか使用できません。

しかし、MIN は、文字型カラムの最小値、つまり、アルファベットの始めに最も近い値の検索に使用できます。

```
SELECT MIN( Surname )
  FROM Contacts;
```

COUNT(*)

COUNT(*) は、重複を除外しないで、指定されたテーブルのロー数を返します。NULL の入っているローを含め、各ローを個別にカウントします。この関数は、引数として式を必要としません。定義上、この関数は、特定のカラムに関する情報を使用しないからです。

次の文は、Employees テーブルの全従業員数を検出します。

```
SELECT COUNT( * )
  FROM Employees;
```

他の集合関数と同じように、COUNT(*) も、SELECT リストにある他の集合関数や WHERE 句などと結合できます。次に例を示します。

```
SELECT COUNT( * ), AVG( UnitPrice )
  FROM Products
  WHERE UnitPrice > 10;
```

COUNT( )	AVG(Products.UnitPrice)
5	18.2

DISTINCT を伴う集合関数

DISTINCT キーワードは、SUM、AVG、COUNT のオプションです。DISTINCT を使用すると、重複した値が除外されてから、合計、平均、またはカウントが計算されます。たとえば、連絡先のある都市の数を検出するには、次の文を実行します。

```
SELECT COUNT( DISTINCT City )
  FROM Contacts;
```

COUNT(DISTINCT Contacts.City)
16

クエリ内で DISTINCT を伴って集合関数を 2 つ以上使用できます。各 DISTINCT は個別に評価されます。次に例を示します。

```
SELECT COUNT( DISTINCT GivenName ) "first names",
     COUNT( DISTINCT Surname ) "last names"
  FROM Contacts;
```

first names	last names
48	60

集合関数と NULL

集合関数が作用しているカラムにあるすべての NULL は、NULL を含めてカウントする COUNT(*) 以外の関数では無視されます。カラム内のすべての値が NULL であれば、COUNT(column_name) は 0 を返します。

WHERE 句に指定されている条件を満たすローがない場合、COUNT は値 0 を返します。その他の関数は、すべて NULL を返します。例を示します。

```
SELECT COUNT( DISTINCT Name )
  FROM Products
  WHERE UnitPrice > 50;
```

COUNT(DISTINCT Name)
0

```
SELECT AVG( UnitPrice )
  FROM Products
  WHERE UnitPrice > 50;
```

AVG(Products.UnitPrice)
(NULL)

GROUP BY 句：クエリ結果のグループへの編成

GROUP BY 句は、テーブルの出力をグループに分けます。1 つ以上のカラム名によって、または計算カラムの結果によって、ローをグループ化することができます。

句の順序
WHERE 句と GROUP BY 句を使用する場合は、WHERE 句を GROUP BY 句より前に置きます。GROUPBY 句が存在する場合は、常に HAVING 句より前に置いてください。HAVING 句は指定されているが GROUP BY 句は指定されていない場合、GROUP BY () 句が指定されたと見なされます。

HAVING 句と WHERE 句の両方を 1 つのクエリに使用できます。条件が HAVING 句に置かれている場合は、グループが構成されたあとでのみ結果のローを論理的に制限します。条件が WHERE 句に置かれている場合は、グループが構成される前に論理が評価されるので、時間が節約されます。

参照

- 「クエリの集合関数」313 ページ

GROUP BY のあるクエリを実行する方法

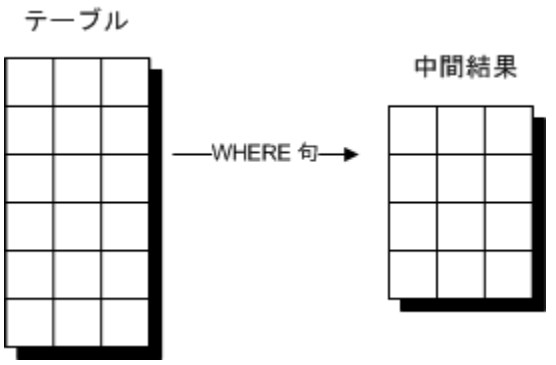
この項では、式や例で GROUP BY 句のサブ句 ROLLUP を使用します。

次のような形式の単一テーブルのクエリを考えてみます。

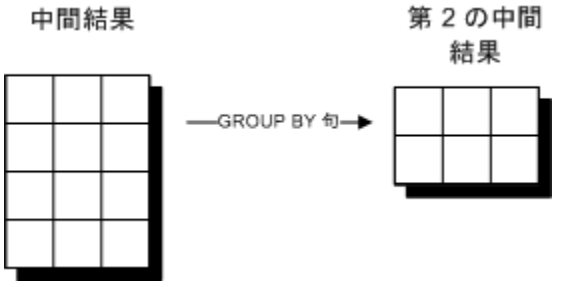
SELECT *select-list*
  FROM *table*
  WHERE *where-search-condition*
  GROUP BY [*group-by-expression* | **ROLLUP** *(group-by-expression)*]
  HAVING *having-search-condition*

このクエリは、次のように実行します。

1. **WHERE 句を適用する**　テーブルのローの一部だけで構成される中間結果が生成されます。



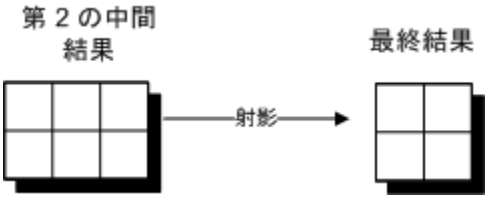
2. **結果をグループに分割する**　GROUP BY 句の指示どおりに、各グループに対してローが 1 つある中間結果が生成されます。生成された各ローには、グループごとの *group-by-expression* と、*select-list* および *having-search-condition* の集合関数の計算結果が含まれます。



3. **ROLLUP 演算があれば適用する**　ROLLUP 演算の一部として計算された小計ローが、結果セットに追加されます。

4. **HAVING 句を適用する**　第 2 の中間結果で HAVING 句の基準に満たないローは、この時点で削除されます。

5. **プロジェクトの結果を表示する**　クエリの結果セットに表示する必要のあるカラムだけが手順 3 から取り出されます。つまり、*select-list* にある式に対応するカラムだけが取り出されます。



このプロセスでは、GROUP BY 句のあるクエリについて、いくつかの要件が作成されます。

- WHERE 句が最初に評価される。したがって、どの集合関数も、WHERE 句を満たすローについてのみ評価されます。

- 最終結果セットは、分割されたローを保持している第 2 の中間結果から構築される。第 2 の中間結果は、*group-by-expression* に一致するローを保持しています。したがって、集合関数ではない式が *select-list* にある場合、同様に *group-by-expression* にもその式がなければなりません。プロジェクションの段階では関数の評価は行われません。

- *group-by-expression* にある式を、*select-list* に含まないことも可能です。その式は、結果にプロジェクションされます。

参照

- 「GROUPING SETS のショートカットとしての ROLLUP と CUBE」553 ページ

複数のカラムを使用した GROUP BY

GROUP BY 句に 1 つ以上の式をリストできます。つまり、式の組み合わせによって、テーブルをグループ化できます。

次のクエリは、まず名前別にグループ化し、次にサイズ別にグループ化した製品の平均価格をリストします。

```
SELECT Name, Size, AVG( UnitPrice )
  FROM Products
  GROUP BY Name, Size;
```

Name	Size	AVG(Products.UnitPrice)
Baseball Cap	One size fits all	9.5
Sweatshirt	Large	24

Name	Size	AVG(Products.UnitPrice)
Tee Shirt	Large	14
Tee Shirt	One size fits all	14
...	...	...

WHERE 句と GROUP BY

GROUP BY を含む文中で、WHERE 句を使用できます。WHERE 句は、GROUP BY 句より先に評価されます。WHERE 句で条件を満たさないローが削除されてから、グループ化が行われます。次に例を示します。

```
SELECT  Name, AVG( UnitPrice )
  FROM Products
  WHERE ID > 400
  GROUP BY Name;
```

クエリ結果の生成に使われるグループには、ID の値が 400 より大きいローだけが含まれます。

例

次に、1 つのクエリの中で、WHERE 句、GROUP BY 句、HAVING 句を使用する例を示します。

```
SELECT Name, SUM( Quantity )
  FROM Products
  WHERE Name LIKE '%shirt%'
  GROUP BY Name
  HAVING SUM( Quantity ) > 100;
```

Name	SUM(Products.Quantity)
Tee Shirt	157

この例では次のようになっています。

- WHERE 句によって、**shirt** という語を含む名前 (Tee Shirt、Sweatshirt) があるローだけが選択されます。
- GROUP BY 句によって、共通の名前のローが集められます。
- SUM 集合関数によって、各グループにある製品の総数が計算されます。
- HAVING 句によって、最終結果から在庫総数が 100 以下のグループが除外されます。

集合関数を伴う GROUP BY

集合関数を含む文には、GROUP BY 句がほとんど常に使用されます。その場合、集合関数はグループごとに 1 つの値を生成します。これらの値は「ベクトル集約値」と呼ばれます。これに対

して、「スカラー集約値」は、GROUP BY 句を使用しない集合関数によって生成される 1 つの値です。

例

次のクエリは、製品の種類別に平均価格をリストします。

```
SELECT Name, AVG( UnitPrice ) AS Price
  FROM Products
  GROUP BY Name;
```

Name	Price
Tee Shirt	12.333333333
Baseball Cap	9.5
Visor	7
Sweatshirt	24
...	...

集合関数と 1 つの GROUP BY 句を持つ SELECT 文が生成するベクトル集約値は、結果の各ローにカラムとして表示されます。それとは対照的に、集合関数があって GROUP BY 句がないクエリが生成するスカラー集約値は、カラムとして表示されますが、ローは 1 つだけです。次に例を示します。

```
SELECT AVG( UnitPrice )
  FROM Products;
```

AVG(Products.UnitPrice)
13.3

GROUP BY と SQL/2008 標準

SQL/2008 標準の構文には、SQL Anywhere でサポートされている構文よりもかなり多くの制約があります。GROUP BY について、SQL/2008 標準では次のように定めています。

- GROUP BY 句で指定された各 *group-by-term* は、**カラム参照**である必要があります。つまり、テーブルからカラムへの参照は、クエリ FROM 句で参照されます。これらの式は、「グループ化カラム」と呼ばれます。

- SELECT リスト、HAVING 句、または ORDER BY 句内の集合関数でない式は、グループ化カラムであるか、グループ化カラムだけを参照する必要があります。ただし、オプションの SQL/2008 言語機能 T301 "Functional dependencies" がサポートされている場合、このような参照では、グループ化カラムによって機能的に決定されたクエリ FROM 句からカラムを参照できます。

SQL Anywhere の GROUP BY 句では、*group-by-term* を、列参照、リテラル定数、変数またはホスト変数、スカラー関数、ユーザ定義関数を含む任意の式にできます。たとえば、次のクエリでは、Employee テーブルは Salary カラムに基づいて 3 つのグループに分割され、グループあたり 1 つのローが生成されます。

```
SELECT COUNT() FROM Employees
  GROUP BY (
    IF SALARY < 25000
      THEN 'low range'
    ELSE IF Salary < 50000
      THEN 'mid range'
    ELSE 'high range'
      ENDIF
    ENDIF);
```

クエリ結果に分割値を含めるには、クエリ SELECT リストに *group-by-term* を追加する必要があります。構文的に有効であるために、SQL Anywhere では、SELECT リスト項目の構文と *group-by-term* の構文が同じであることが確認されます。ただし、構文的に大きい SQL 構成では、この分析は失敗することがあります。さらに、サブクエリを含む式では等しいことの比較は行われません。

次の例では、SQL Anywhere は 2 つの IF 式が同じであることを検出し、エラーなしに結果を計算します。

```
SELECT (IF SALARY < 25000 THEN 'low range' ELSE IF Salary < 50000 THEN 'mid range' ELSE 'high
range' ENDIF ENDIF), COUNT()
FROM Employees
GROUP BY (IF SALARY < 25000 THEN 'low range' ELSE IF Salary < 50000 THEN 'mid range' ELSE
'high range' ENDIF ENDIF);
```

ただし、次のクエリには、エラーを返すサブクエリが GROUP BY 句に含まれています。

```
SELECT (Select State from Employees e WHERE e.EmployeeID = e2.EmployeeID),
    COUNT()
  FROM Employees e2
  GROUP BY (Select State from Employees e WHERE EmployeeID = e2.EmployeeID)
```

より簡潔な方法は、SELECT リスト式でエイリアスを使用し、GROUP BY 句内でエイリアスを参照することです。エイリアスを使用すると、SELECT リストと GROUP BY 句に相関サブクエリを含めることができます。次のような形で使用される SELECT リストエイリアスは、ベンダー拡張です。

```
SELECT (
  IF SALARY < 25000
    THEN 'low range'
  ELSE IF Salary < 50000
    THEN 'mid range'
  ELSE 'high range'
    ENDIF
  ENDIF) AS Salary_Range,
  COUNT() FROM Employees GROUP BY Salary_Range;
```

SQL Anywhere は、SQL/2008 言語機能 T301 (Functional dependencies) のすべての部分をサポートしているわけではありませんが、GROUP BY の単語に基づいて派生値をサポートしています。SQL Anywhere は、GROUP BY の単語、リテラル定数、(ホスト) 変数を参照する SELECT リスト式を、それらの値を変更する可能性があるスカラー関数の有無に関係なくサポートしています。例として、次のクエリでは、市と州の組み合わせ別に従業員の数がリストされます。

```
SELECT City || ' ' || State, SUBSTRING(City,1,3), COUNT()
FROM Employees
GROUP BY City, State
```

参照

- 「GROUP BY 句」『SQL Anywhere サーバ SQL リファレンス』
- 「トラブルシューティング：集合関数と外部参照」『SQL Anywhere 16 変更点とアップグレード』

HAVING 句：データグループの選択

HAVING 句は、クエリが返すローを制限します。HAVING 句は、WHERE 句が SELECT 句の条件を設定するのと同じような方法で、GROUP BY 句の条件を設定します。

HAVING 句の探索条件は、WHERE 探索条件と同じです。ただし、WHERE 探索条件では集合関数を指定できません。たとえば次の使用方法は有効です。

```
HAVING AVG( UnitPrice ) > 20
```

次の使用方法は無効です。

```
WHERE AVG( UnitPrice ) > 20
```

集合関数を伴う HAVING の使用

次の文は、集合関数を持つ HAVING 句を使用する簡単な例です。

2 種類以上のサイズまたは色がある製品をリストするには、1 種類だけの製品を含むグループは省き、Products テーブルのローを名前でグループ化するクエリが必要です。

```
SELECT Name
  FROM Products
  GROUP BY Name
  HAVING COUNT( * ) > 1;
```

Name
Tee Shirt
Baseball Cap
Visor
Sweatshirt

集合関数を伴わない HAVING の使用

HAVING 句は、集合関数がなくても使用できます。

次のクエリは、製品をグループ化し、その結果セットを name が B で始まるグループだけに限定します。

```
SELECT Name
  FROM Products
```

```
GROUP BY Name
HAVING Name LIKE 'B%';
```

Name
Baseball Cap

HAVING における 2 つ以上の条件

HAVING 句では、2 つ以上の検索条件を指定できます。これらの条件は、次の例に示すように、AND、OR、NOT 演算子と組み合わされます。

2 種類以上のサイズまたは色があり、1 種類の単価が 10 ドルを超える製品をリストするには、1 種類だけの製品を含むグループと、単価の最大値が 10 ドル以下のグループを省き、Products テーブルのローを名前でグループ化するクエリが必要です。

```
SELECT Name
  FROM Products
  GROUP BY Name
  HAVING COUNT( * ) > 1
  AND MAX( UnitPrice ) > 10;
```

Name
Tee Shirt
Sweatshirt

参照

- 「集合関数を使用できる場所」469 ページ

ORDER BY 句：クエリ結果のソート

ORDER BY 句によって、クエリ結果を 1 つ以上のカラムでソートできます。それぞれのソートは、昇順 (ASC) でも降順 (DESC) でも可能です。どちらも指定されていない場合は、ASC が使用されます。

簡単な例

次のクエリは、name 順に結果を返します。

```
SELECT ID, Name
  FROM Products
  ORDER BY Name;
```

ID	Name
400	Baseball Cap
401	Baseball Cap

ID	Name
700	Shorts
600	Sweatshirt
...	...

2 つ以上のカラムでのソート

ORDER BY 句で 2 つ以上のカラムを指定すると、ソートはネストされます。

次の文は、Products テーブルにある shirt を、name カラムで昇順ソートしてから、各 name の Quantity カラムで降順ソートします。

```
SELECT ID, Name, Quantity
  FROM Products
  WHERE Name like '%shirt%'
  ORDER BY Name, Quantity DESC;
```

ID	Name	Quantity
600	Sweatshirt	39
601	Sweatshirt	32
302	Tee Shirt	75
301	Tee Shirt	54
...	...	...

カラム位置の使用

カラム名の代わりに、SELECT リストの中のカラムの位置番号を使用できます。カラム名と SELECT リスト番号は混在できます。次の文は両方とも、前述の文と同じ結果を生成します。

```
SELECT ID, Name, Quantity
  FROM Products
  WHERE Name like '%shirt%'
  ORDER BY 2, 3 DESC;
SELECT ID, Name, Quantity
  FROM Products
  WHERE Name like '%shirt%'
  ORDER BY 2, Quantity DESC
```

SQL のほとんどのバージョンでは、SELECT リストに ORDER BY 項目があることが必須ですが、SQL Anywhere にはそのような制限はありません。次のクエリでは、SELECT リストに Quantity カラムがありませんが、Quantity 順に結果をソートします。

```
SELECT ID, Name
  FROM Products
  WHERE Name like '%shirt%'
  ORDER BY 2, Quantity DESC;
```

ORDER BY と NULL

ORDER BY では、ソート順が昇順の場合、NULL は他のすべての値の前に来ます。

ORDER BY と大文字と小文字の区別

大文字と小文字が混在するデータに対する ORDER BY 句の影響は、データベースの作成時に指定された、データベース照合順と大文字と小文字の区別によって異なります。

SELECT、UPDATE、DELETE クエリブロック内のロー制限句

FIRST 句、TOP 句、および LIMIT 句を使用することによって、WHERE 句を満たすローのサブセットを返したり、または更新や削除を実行できます。FIRST 句、TOP 句、および LIMIT 句は、ORDER BY 句を含む SELECT クエリブロック内で使用できます。FIRST 句と TOP 句は、DELETE および UPDATE クエリブロックでも使用できます。

FIRST 句、TOP 句、および LIMIT 句はロー制限句です。次に構文を示します。

row-limitation-option-1 :
FIRST | **TOP** { **ALL** | *limit-expression* } [**START AT** *startat-expression*]

row-limitation-option-2 :
LIMIT { [*offset-expression*,] *limit-expression* | *limit-expression* **OFFSET** *offset-expression* }

limit-expression : *simple-expression*

startat-expression : *simple-expression*

offset-expression : *simple-expression*

simple-expression :
integer
| *variable*
| (*simple-expression*)
| (*simple-expression* { **+** | **-** | ***** } *simple-expression*)

ロー制限句は、SELECT 句にのみ指定できます。これらの句を指定する場合は、ローの順序を意味のあるものにするために ORDER BY 句も指定する必要があります。

- ***row-limitation-option-1*** このタイプの句は、SELECT、UPDATE、または DELETE クエリブロックと一緒に使用できます。TOP 引数と START AT 引数には、ホスト変数、整数定数、または整数変数を使用した簡単な算術演算を指定できます。TOP 引数は 0 以上の値、START AT 引数は 1 以上の値に評価される必要があります。*startat-expression* が指定されていない場合は、デフォルトの 1 になります。

  式 limit-expression + startat-expression -1' は、9223372036854775807 = 2^64-1 未満の値に評価される必要があります。TOP の引数が ALL の場合、*startat-expression* で始まるすべてのローが返されます。

  TOP limit-expression START AT startat-expression 句は、LIMIT (startat-expression-1), limit-expression、または LIMIT limit-expression OFFSET (startat-expression-1) と同等です。

- ***row-limitation-option-2*** このタイプの句は SELECT クエリブロック内でのみ使用できます。LIMIT 引数と OFFSET 引数には、ホスト変数、整数定数、または整数変数を使用した簡単な算術演算を指定できます。LIMIT 引数と OFFSET 引数は 0 以上の値に評価される必要があります。*offset-expression* が指定されていない場合には、でデフォルト値の 1 になります。式 limit-expression + offset-expression は、9223372036854775807 = 2^64-1 未満に評価される必要があります。

ロー制限句 LIMIT offset-expression, limit-expression は LIMIT limit-expression OFFSET offset-expression と同等です。どちらの構成も TOP limit-expression START AT (offset-expression + 1) と同等です。

LIMIT キーワードはデフォルトでは無効になっています。LIMIT キーワードを有効にするには、reserved_keywords オプションを使用します。

例

次のクエリは、従業員を姓でソートした場合の最初の従業員に関する情報を返します。

```
SELECT FIRST *
FROM Employees
ORDER BY Surname;
```

次のクエリは、名前が姓でソートされた場合の最初の 5 人の従業員を返します。

```
SELECT TOP 5 *
FROM Employees
ORDER BY Surname;

SELECT *
FROM Employees
ORDER BY Surname
LIMIT 5;
```

TOP を使用する場合、START AT を使用してオフセットを指定できます。次の文は、姓で降順にソートした場合の 5 番目と 6 番目の従業員をリストします。

```
SELECT TOP 2 START AT 5 *
FROM Employees
ORDER BY Surname DESC;

SELECT *
FROM Employees
ORDER BY Surname DESC
LIMIT 2 OFFSET 4;

SELECT *
FROM Employees
ORDER BY Surname DESC
LIMIT 4,2;
```

矛盾のない結果を得るためには、FIRST と TOP は必ず ORDER BY 句と併用してください。FIRST または TOP を ORDER BY なしで使用すると、構文警告の要因になります。また予期しない結果を生成する可能性があります。

```
CREATE OR REPLACE VARIABLE atop INT = 10;
```

次のクエリは、名前が姓でソートされた場合の最初の 5 人の従業員を返します。

```
SELECT TOP (atop -5) *
FROM Employees
ORDER BY Surname;

SELECT *
FROM Employees
ORDER BY Surname
LIMIT (atop-5);
```

次の文は、姓で降順にソートした場合の 5 番目と 6 番目の従業員をリストします。

```
SELECT TOP (atop - 8) START AT (atop -2 -3) *
FROM Employees
ORDER BY Surname DESC;

SELECT *
FROM Employees
ORDER BY Surname DESC
LIMIT (atop - 8) OFFSET (atop -2 -3 -1);
```

参照

- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「UPDATE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DELETE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「reserved_keywords オプション」『SQL Anywhere サーバ データベース管理』

ORDER BY と GROUP BY

ORDER BY 句を使用して、特定の方法で GROUP BY の結果を順序付けできます。

例

次のクエリは、各製品の平均価格を検出し、その平均価格順に結果をソートします。

```
SELECT Name, AVG( UnitPrice )
  FROM Products
  GROUP BY Name
  ORDER BY AVG( UnitPrice );
```

Name	AVG(Products.UnitPrice)
Visor	7
Baseball Cap	9.5
Tee Shirt	12.333333333
Shorts	15
...	...

UNION、INTERSECT、EXCEPT を使用した、クエリ結果に対する集合操作

この項で説明する演算子は、2 つ以上のクエリの結果に対して集合操作を実行します。こうした操作の多くは WHERE 句や HAVING 句を使用しても実行できますが、中にはこれらの集合ベースの演算子を使用せずに実行するのは非常に困難な操作もあります。次に例を示します。

- データが正規化されていない場合、たとえテーブルが関連付けられていなくても、異なるように思われる情報を 1 つの結果セットにまとめたいと思うことがあります。
- WHERE 句や HAVING 句内では、集合演算子によって、NULL を扱う方法が異なります。WHERE 句や HAVING 句内では、NOT NULL のエントリが同じである NULL を含む 2 つのローは、同じと見なされません。2 つの NULL 値は、同じと定義されていないためです。集合演算子は、そうした 2 つのローを同じと見なします。

参照

- 「集合演算子と NULL」486 ページ
- 「EXCEPT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「INTERSECT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「UNION 文」『SQL Anywhere サーバ SQL リファレンス』

UNION 句：結果セットの結合

UNION 演算子は、2 つ以上のクエリの結果を結合して、単一の結果セットにします。

デフォルトでは、UNION 演算子は、結果セットから重複しているローを削除します。ALL オプションを使用すると、重複は削除されません。最終的な結果セットにあるカラムは、最初の結果セットのカラムと同じ名前です。UNION 演算子はいくつでも使用できます。

デフォルトでは、UNION 演算子を複数含んでいる文は、左から右に評価されます。カッコを使用して評価順を指定できます。

たとえば、次の 2 つの式は、重複ローを結果セットから削除する方法が異なるため、等しくありません。

```
x UNION ALL ( y UNION z )
```

```
(x UNION ALL y) UNION z
```

最初の式では、y と z の間の UNION で、重複が削除されます。そのセットと x の間の UNION では、重複が削除されません。2 番目の式では、x と y の間の UNION では重複が含められますが、次の z との UNION では削除されます。

EXCEPT と INTERSECT

EXCEPT 句は、2 つの結果セット間の違いをリストします。次の一般的な構成では、query-1 の結果セットにあるすべてのローがリストされますが、query-2 の結果セットにあるローは除かれます。

query-1
EXCEPT
query-2

INTERSECT 句は、2 つの結果セットの両方にあるローをリストします。次の一般的な構成では、query-1 と query-2 の両方の結果セットにあるすべてのローをリストします。

query-1
INTERSECT
query-2

UNION 句と同様に、EXCEPT と INTERSECT には ALL 修飾子を指定できます。ALL 修飾子を使用すると、結果セットから重複ローが削除されません。

参照

- 「EXCEPT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「INTERSECT 文」『SQL Anywhere サーバ SQL リファレンス』

集合操作のルール

UNION 文、EXCEPT 文、INTERSECT 文には、次のルールが適用されます。

- **優先度**　UNION 演算子と EXCEPT 演算子の優先度は同じであり、どちらも左から右へ評価されます。INTERSECT 演算子の優先度は UNION 演算子と EXCEPT 演算子よりも高く、複数の INTERSECT 演算子が使用される場合は同じく左から右へ評価されます。

- **SELECT リストの項目数は同じにする**　クエリ内のすべての SELECT リストは、式 (カラム名、算術式、集合関数など) の数を同じにします。次の文は、最初の SELECT リストが 2 番目のリストより長いので無効です。

```
SELECT store_id, city, state
  FROM stores
UNION
  SELECT store_id, city
    FROM stores_east;
```

- **データ型を一致させる**　SELECT リストで対応する式のデータ型を同じにするか、2 種類のデータ型の間で暗黙的データ変換ができるようにします。または、明示的変換を指定します。

  たとえば、CHAR データ型のカラムと INT データ型のカラムの間では、明示的変換が指定されなければ UNION、INTERSECT、または EXCEPT は不可能です。しかし、MONEY データ型のカラムと INT データ型のカラムの間では、集合操作が可能です。

- **カラム順**　集合操作の各クエリで、対応する式を同じ順序で並べます。これは、集合演算子が、SELECT 句の各クエリで指定された順に 1 対 1 で式を比較するためです。

- **複数の集合操作**　次の例のように、いくつかの集合操作を一緒に配列できます。

```
SELECT City AS Cities
  FROM Contacts
  UNION
    SELECT City
```

```
      FROM Customers
  UNION
    SELECT City
      FROM Employees;
```

UNION 文では、クエリの順番は重要ではありません。INTERSECT では、2 つ以上のクエリがある場合、順番は重要です。EXCEPT では、順番は常に重要です。

- **カラム見出し**　UNION の結果生成されるテーブルのカラム名は、文中の最初のクエリから取得されます。次の例のように、最初のクエリの SELECT リストで結果セットの新しいカラムヘッダを定義します。

```
SELECT City AS Cities
  FROM Contacts
  UNION
    SELECT City
      FROM Customers;
```

次のクエリでは、カラム見出しは UNION 句の最初のクエリで定義した City のままです。

```
SELECT City
  FROM Contacts
  UNION
    SELECT City AS Cities
      FROM Customers;
```

または、WITH 句を使用してカラム名を定義できます。次に例を示します。

```
WITH V( Cities )
AS ( SELECT City
    FROM Contacts
    UNION
      SELECT City
       FROM Customers )
SELECT * FROM V;
```

- **結果の順序付け**　SELECT 文の WITH 句を使用して、SELECT リスト内のカラム名に順序を付けられます。次に例を示します。

```
WITH V( CityName )
AS ( SELECT City AS Cities
    FROM Contacts
    UNION
      SELECT City
        FROM Customers )
SELECT * FROM V
  ORDER BY CityName;
```

また、クエリのリストの最後に単一の ORDER BY 句を使用できますが、次の例のように、カラム名ではなく整数を使用してください。

```
SELECT City AS Cities
  FROM Contacts
  UNION
    SELECT City
      FROM Customers
  ORDER BY 1;
```

集合演算子と NULL

集合演算子 UNION、EXCEPT、INTERSECT と探索条件内では、NULL を扱う方法が異なります。この違いが、集合演算子を使用する主な理由の 1 つです。

ローを比較するとき、集合演算子は、NULL 値を互いに等しいものとして扱います。対照的に、探索条件で NULL が NULL と比較された場合、結果は不定 (真ではない) となります。

この違いがもたらす結果の 1 つとして、query-1 EXCEPT ALL query-2 の結果セット内のロー数が、**常に**各クエリの結果セット内のロー数の差異であるということです。

テーブル T1 と T2 を例に説明します。各テーブルには、次のカラムがあります。

```
col1 INT,
col2 CHAR(1)
```

テーブルとデータは次のように設定されています。

```
CREATE TABLE T1 (col1 INT, col2 CHAR(1));
CREATE TABLE T2 (col1 INT, col2 CHAR(1));
INSERT INTO T1 (col1, col2) VALUES(1, 'a');
INSERT INTO T1 (col1, col2) VALUES(2, 'b');
INSERT INTO T1 (col1) VALUES(3);
INSERT INTO T1 (col1) VALUES(3);
INSERT INTO T1 (col1) VALUES(4);
INSERT INTO T1 (col1) VALUES(4);
INSERT INTO T2 (col1, col2) VALUES(1, 'a');
INSERT INTO T2 (col1, col2) VALUES(2, 'x');
INSERT INTO T2 (col1) VALUES(3);
```

テーブル内のデータは次のようになっています。

●テーブル T1

col1	col2
1	a
2	b
3	(NULL)
3	(NULL)
4	(NULL)
4	(NULL)

●テーブル T2。

col1	col2
1	a

col1	col2
2	x
3	(NULL)

T2 にもある T1 のローを要求するクエリの一例を次に示します。

```
SELECT T1.col1, T1.col2
  FROM T1 JOIN T2
  ON T1.col1 = T2.col1
  AND T1.col2 = T2.col2;
```

T1.col1	T1.col2
1	a

ロー (3, NULL) は、結果セットにありません。これは、NULL と NULL の比較が真ではないためです。対照的に、INTERSECT 演算子を使用してこの問題にアプローチすると、結果に NULL を持つローが含まれます。

```
SELECT col1, col2
  FROM T1
  INTERSECT
    SELECT col1, col2
      FROM T2;
```

col1	col2
1	a
3	(NULL)

次のクエリは、探索条件を使用して T2 にはない T1 のローをリストしています。

```
SELECT col1, col2
  FROM T1
  WHERE col1 NOT IN (
     SELECT col1
       FROM T2
       WHERE T1.col2 = T2.col2 )
  OR col2 NOT IN (
     SELECT col2
       FROM T2
       WHERE T1.col1 = T2.col1 );
```

col1	col2
2	b
3	(NULL)

col1	col2
4	(NULL)
3	(NULL)
4	(NULL)

T1 の NULL を含むローは、比較によって除外されていません。対照的に、EXCEPT ALL を使用してこの問題にアプローチすると、両方のテーブルに含まれる NULL を持つローが結果から除外されます。この場合、T2 の (3, NULL) ローは、T1 の (3, NULL) ローと同じと認識されています。

```
SELECT col1, col2
  FROM T1
  EXCEPT ALL
    SELECT col1, col2
      FROM T2;
```

col1	col2
2	b
3	(NULL)
4	(NULL)
4	(NULL)

EXCEPT 演算子を使用すると、結果がさらに制限されます。EXCEPT 演算子は、T1 から (3, NULL) のローを両方とも削除し、また (4, NULL) ローの 1 つを重複として除外しています。

```
SELECT col1, col2
  FROM T1
  EXCEPT
    SELECT col1, col2
      FROM T2;
```

col1	col2
2	b
4	(NULL)

ジョイン：複数テーブルからのデータ検索

データベースを作成する場合は、冗長エントリを多く含む 1 つの大きなテーブルにではなく、別々のテーブルに各オブジェクト固有の情報を配置して、データを正規化します。したがって、複数のテーブルから関連データを取り出すには、SQL JOIN 演算子を使用してジョイン操作を行います。ジョイン操作では、複数のテーブル (またはビュー) からの情報を使用して 1 つのより

大きいテーブルを再作成します。各種のジョインを使用すると、特定のタスクに適したさまざまな仮想テーブルを作成できます。

テーブルのリストを表示する

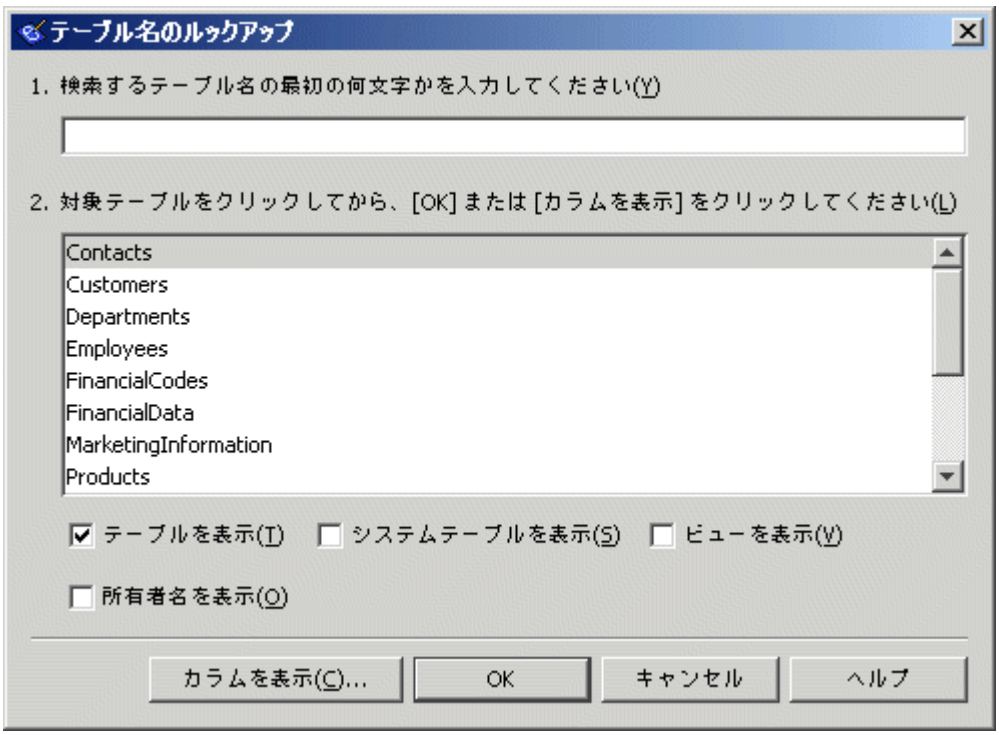
Interactive SQL から接続しているデータベースの、すべてのテーブルとそのカラムを表示できます。

前提条件

データベースに接続されている必要があります。

♦ タスク

1. Interactive SQL では、[F7] キーを押すと、接続したデータベース内のテーブルリストを表示できます。



2. テーブルを選択してから **[カラムを表示]** をクリックすると、そのテーブルのカラムが表示されます。

3. [Esc] キーを押すとテーブルリストに戻ります。ここでもう一度 [Esc] キーを押すと **[SQL 文]** ウィンドウ枠に戻ります。[Enter] キーを押すと、選択されているテーブルまたはカラム名が **[SQL 文]** ウィンドウ枠の現在カーソルが置かれている場所にコピーされます。

4. リストを終了するには、[Esc] キーを押します。

   SQL Anywhere サンプルデータベース内のテーブルの詳細については、「チュートリアル：サンプルデータベースへの接続」『SQL Anywhere サーバ データベース管理』を参照してください。

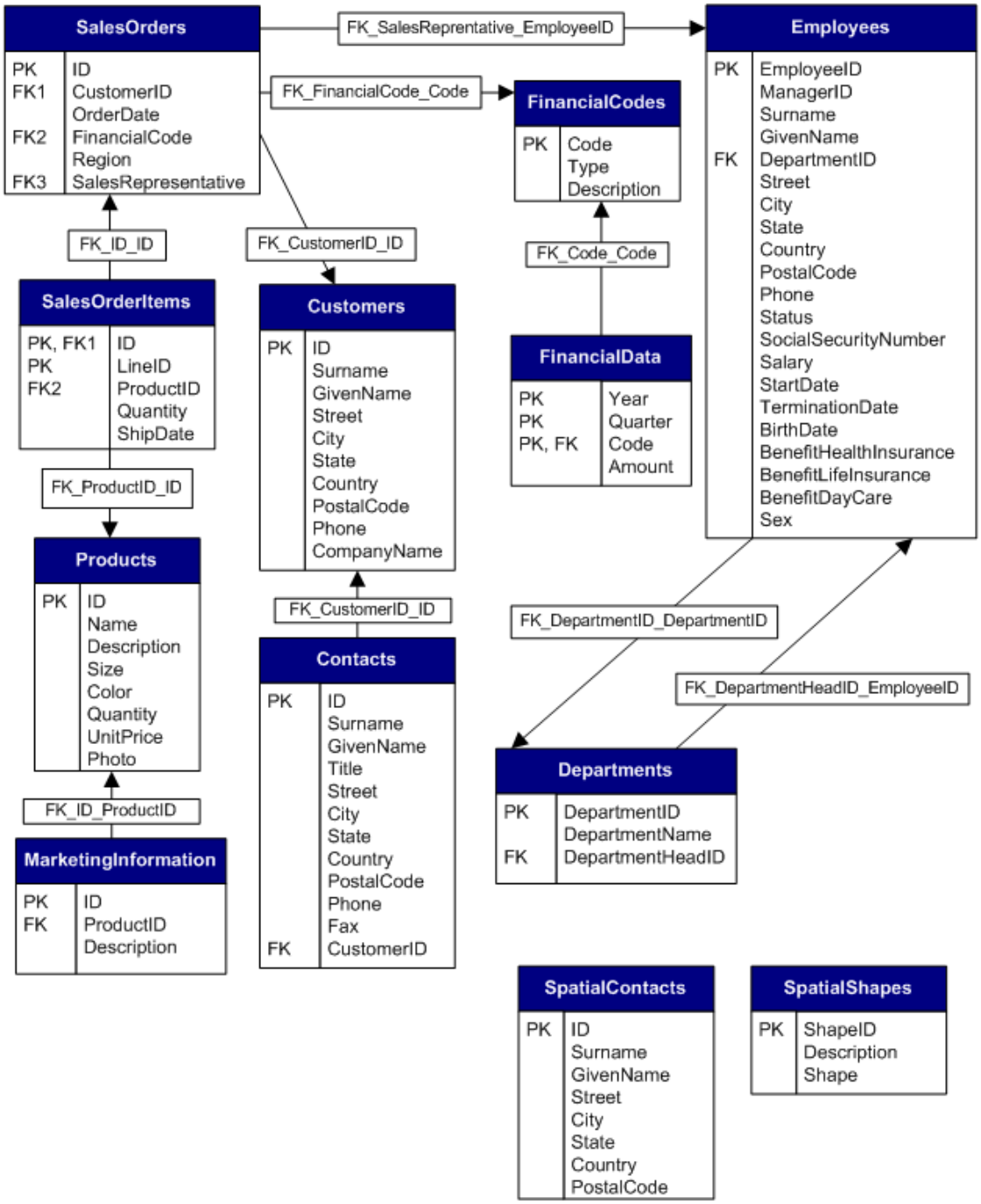
結果

接続しているデータベースの全テーブルのリストが表示されます。各テーブルのカラムを表示するオプションがあります。

サンプルデータベーススキーマ

次の図は、SQL Anywhere サンプルデータベースと、そのテーブルに関連付けられた外部キー名を示しています。一部の高度なジョインでは、これらの外部キーの役割名が必須です。

役割名の詳細については、「複数の外部キー関係がある場合のキージョイン」521 ページを参照してください。


SalesOrders
PK ID
FK1 CustomerID
OrderDate
FK2 FinancialCode
Region
FK3 SalesRepresentative
FK_SalesReprentative_EmployeeID
Employees
PK EmployeeID
ManagerID
Surname
GivenName
FK DepartmentID
Street
City
State
Country
PostalCode
Phone
Status
SocialSecurityNumber
Salary
StartDate
TerminationDate
BirthDate
BenefitHealthInsurance
BenefitLifeInsurance
BenefitDayCare
Sex
FK_FinancialCode_Code
FinancialCodes
PK Code
Type
Description
FK_ID_ID
FK_CustomerID_ID
FK_Code_Code
SalesOrderItems
PK, FK1 ID
PK LineID
FK2 ProductID
Quantity
ShipDate
Customers
PK ID
Surname
GivenName
Street
City
State
Country
PostalCode
Phone
CompanyName
FinancialData
PK Year
PK Quarter
PK, FK Code
Amount
FK_ProductID_ID
Products
PK ID
Name
Description
Size
Color
Quantity
UnitPrice
Photo
FK_CustomerID_ID
Contacts
PK ID
Surname
GivenName
Title
Street
City
State
Country
PostalCode
Phone
Fax
FK CustomerID
FK_DepartmentID_DepartmentID
FK_DepartmentHeadID_EmployeeID
Departments
PK DepartmentID
DepartmentName
FK DepartmentHeadID
FK_ID_ProductID
MarketingInformation
PK ID
FK ProductID
Description
SpatialContacts
PK ID
Surname
GivenName
Street
City
State
Country
PostalCode
SpatialShapes
PK ShapeID
Description
Shape

ジョイン操作

「ジョイン」とは、テーブル内のローを、指定したカラムの値と比較することによって結合する操作のことです。この項では、SQL Anywhere のジョイン構文の概要を説明します。

リレーショナルデータベースは別々のタイプのオブジェクトに関する情報を別々のテーブルに保存します。たとえば、あるテーブルには従業員だけの情報があり、別のテーブルには部署関連の情報があります。Employees テーブルには、従業員の名前や住所などの情報が保存されています。Departments テーブルには、部署名や部長名などの情報が入ります。

ほとんどの問い合わせに対する答えは、さまざまなテーブルの情報を組み合わせることによってのみ取得できます。たとえば、「営業部の責任者は誰か」という質問に対する回答を取得するためには、Departments テーブルで適切な従業員を特定し、Employees テーブルでその従業員名を検索します。

ジョインは複数のテーブルからの情報を取り入れた新規の仮想テーブルを作ることによって、そのような質問に答える手段です。たとえば、Employees テーブルと Departments テーブルの情報を組み合わせて、部長のリストを作成できます。FROM 句を使用して、必要な情報の入ったテーブルを特定します。

ジョインを有効なものにするには、各テーブルの適切なカラムを組み合わせてください。部長のリストを作成するには、組み合わせたテーブルの各ローに部署名とその部署を管理する従業員の名前を指定してください。特定のタイプのジョイン操作を指定するか、ON 句を使用して、複合テーブルにカラムをどのように適合させるかを調節します。

参照

- 「FROM 句」『SQL Anywhere サーバ SQL リファレンス』

ジョイン条件

「ジョイン条件」を使用するとテーブルをジョインできます。ジョイン条件とは、簡単に言えば探索条件のことです。ジョイン条件は、カラム内の値と値の関係に基づいて、ジョインしたテーブルからローのサブセットを選択します。たとえば、次のクエリでは Products テーブルと SalesOrderItems テーブルからデータが取り出されます。

```
SELECT *
FROM Products JOIN SalesOrderItems
  ON Products.ID = SalesOrderItems.ProductID;
```

このクエリのジョイン条件は、次の部分です。

```
Products.ID = SalesOrderItems.ProductID
```

このジョイン条件は、両方のテーブルのローに同じ製品 ID がある場合にかぎり、結果セットでこのローを結合できることを示しています。

ジョイン条件には明示的なものと、生成されたものがあります。「明示的ジョイン条件」とは、ON 句または WHERE 句の中に置かれたジョイン条件のことです。以下のクエリでは ON 句が使用されています。このクエリでは、2 つのテーブルの直積 (すべてのローの組み合わせ) が生成

されます。ただし、ID 番号が一致しないローは除外されます。結果は、注文の詳細が記載された顧客リストになります。

```
SELECT *
FROM Customers
JOIN SalesOrders
ON SalesOrders.CustomerID = Customers.ID;
```

これに対し、「生成されたジョイン条件」とは、KEY JOIN または NATURAL JOIN を指定すると自動的に作成されるジョイン条件のことです。キージョインの場合、ジョイン条件はテーブル間の外部キー関係に基づいて生成されます。ナチュラルジョインの場合、ジョイン条件は名前が同じカラムに基づいて生成されます。

ヒント
キージョイン構文とナチュラルジョイン構文は、どちらもショートカットです。つまり、KEY も NATURAL も指定しないで JOIN キーワードを使用してから、ON 句内で同じジョイン条件を明示的に記述しても、同じ結果が得られます。

キージョインまたはナチュラルジョインを指定した ON 句を使用すると、使用されるジョイン条件は、明示的に指定したジョイン条件と生成されたジョイン条件の「論理積」になります。ジョイン条件はキーワード AND と組み合わされます。

ジョインしたテーブル

SQL Anywhere では、次のようなジョイン条件の指定をサポートしています。

- **CROSS JOIN (クロスジョイン)**　2 つのテーブルにこのタイプのジョインを指定すると、両方のテーブルにあるローの可能な組み合わせがすべて生成されます。結果セットのサイズは、1 番目のテーブルにあるローの数と 2 番目のテーブルにあるローの数を乗算したものです。クロスジョインは、直積とも呼ばれます。クロスジョインでは ON 句を使用できません。
- **KEY JOIN (キージョイン)**　このタイプのジョイン条件では、テーブル間の外部キー関係が使用されます。ジョインタイプ (INNER、OUTER など) を指定しないで JOIN キーワードを使用する場合や、ON 句がない場合は、キージョインはデフォルトになります。
- **NATURAL JOIN (ナチュラルジョイン)**　このジョインでは、同じ名前のカラムに基づいて、ジョイン条件が自動的に生成されます。
- **ON 句を使用したジョイン**　ON 句内にジョイン条件を明示的に指定すると、このタイプのジョインになります。これをキージョインまたはナチュラルジョインと併用すると、ジョイン条件には生成されたジョイン条件と明示的ジョイン条件の両方が含まれます。KEY や NATURAL の付かない JOIN キーワードと併用すると、生成されるジョイン条件はありません。

内部ジョインと外部ジョイン

キージョイン、ナチュラルジョイン、ON 句付きジョインは、INNER、LEFT OUTER、RIGHT OUTER、FULL OUTER を指定して修飾することもできます。デフォルトは INNER です。LEFT、RIGHT、FULL を使う場合、キーワード OUTER はオプションです。

内部ジョインでは、結果の各ローがジョイン条件を満たします。

左外部ジョインまたは右外部ジョインでは、テーブルのどちらか一方のすべてのローの値が保護されます。もう一方のテーブルでは、ジョイン条件を満たさないローに NULL が返されます。たとえば右外部ジョインでは、右側が保護され、左側に NULL が入力されます。

全外部ジョインでは、両方のテーブルのすべてのローが保護され、ジョイン条件を満たさないローに NULL が入ります。

参照

● 「明示的なジョイン条件 (ON 句)」496 ページ

2 つのテーブル間のジョイン

簡単な内部ジョインの計算方法を理解するために、次のクエリを例にして考えてみましょう。これは、「在庫数と同数の受注数があったのはどの製品のサイズか」という質問への回答です。

```
SELECT DISTINCT Name, Size,
   SalesOrderItems.Quantity
FROM Products JOIN SalesOrderItems
ON Products.ID = SalesOrderItems.ProductID
  AND Products.Quantity = SalesOrderItems.Quantity;
```

Name	Size	Quantity
Baseball Cap	One size fits all	12
Visor	One size fits all	36

このクエリは次のように解釈できます。次に示すのはこのクエリの処理概念の説明であり、ジョインを含むクエリのセマンティックを例証するためのものです。ここで述べる内容は、SQL Anywhere が実際に結果セットを計算する過程を示すものではありません。

● Products テーブルと SalesOrderItems テーブルの直積を作成します。直積には 2 つのテーブルのローの組み合わせがすべて含まれます。

● 製品 ID が同じではないローはすべて除外されます (ジョイン条件が Products.ID = SalesOrderItems.ProductID であるため)。

● 数量が同じでないローはすべて除外されます (ジョイン条件が Products.Quantity = SalesOrderItems.Quantity であるため)。

● Products.Name、Products.Size、SalesOrderItems.Quantity の 3 つのカラムを持つ結果テーブルが作成されます。

● 重複するローがすべて除外されます (キーワードが DISTINCT であるため)。

参照

● 「外部ジョイン」501 ページ

3 つ以上のテーブル間のジョイン

SQL Anywhere では、ジョインできるテーブルの数に制限はありません。

3 つ以上のテーブルをジョインする場合、カッコはオプションです。カッコを使用しない場合には、SQL Anywhere では文が左から右に評価されます。そのため、A JOIN B JOIN C は (A JOIN B) JOIN C と同じです。また、次の 2 つの SELECT 文は同じです。

```
SELECT *
FROM A JOIN B JOIN C JOIN D;

SELECT *
FROM ( ( A JOIN B ) JOIN C ) JOIN D;
```

3 つ以上のテーブルをジョインした場合、そのジョインにはテーブル式が含まれます。A JOIN B JOIN C の例では、テーブル式 A JOIN B が C とジョインされます。つまり、概念上は A と B がジョインされ、その結果が C にジョインされます。

テーブル式に外部ジョインが含まれるときは、ジョインの順序が重要になります。たとえば、A JOIN B LEFT OUTER JOIN C は、(A JOIN B) LEFT OUTER JOIN C と解釈されます。テーブル式 A JOIN B が C にジョインされます。このとき、テーブル式 A JOIN B は保護され、テーブル C には NULL が入力されます。

参照

- 「外部ジョイン」501 ページ
- 「テーブル式のキージョイン」524 ページ
- 「テーブル式のナチュラルジョイン」518 ページ

互換性のあるデータ型のジョイン

2 つのテーブルをジョインする場合は、比較するカラムのデータ型は同じか互換性のあるものにしてください。

参照

- 「データ型の比較」『SQL Anywhere サーバ SQL リファレンス』

DELETE 文、UPDATE 文、INSERT 文でのジョイン

ジョインは、DELETE 文、UPDATE 文、INSERT 文、SELECT 文で使用できます。ansi_update_constraints オプションが Off に設定されていれば、ジョインを含むカーソルをいくつか更新できます。SQL Anywhere 7 より前に作成されたデータベースでは、この設定がデフォルトです。バージョン 7 以降を使って作成されたデータベースでは Cursors がデフォルトです。

参照

- 「ansi_update_constraints オプション」『SQL Anywhere サーバ データベース管理』

ANSI 以外のジョイン

SQL Anywhere は、ISO/ANSI 標準のジョインをサポートしています。また、次に示す標準以外のジョインもサポートしています。

- 「Transact-SQL の外部ジョイン (*= or =*)」
- 「ジョインで重複する相関名 (スタージョイン)」
- 「キージョイン」

REWRITE 関数を使うと、ANSI の機能に相当する ANSI 以外のジョインを確認できます。

参照

- 「REWRITE 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』

明示的なジョイン条件 (ON 句)

キージョインやナチュラルジョインの代わりに、またはこれらとともに、明示的ジョイン条件を使用してジョインを指定できます。ジョインの直後に ON 句を挿入し、ジョイン条件を指定してください。ジョイン条件は、常にその直前にあるジョインを参照します。ON 句は、ジョインのローに制限を適用します。これは、WHERE 句がクエリのローに制限を適用するのと同様です。

ON 句を使用すると、CROSS JOIN よりも使用しやすいジョインを構成できます。たとえば、SalesOrders テーブルと Employees テーブルのジョインに ON 句を適用できます。この場合、取得される結果のすべてのローで、SalesOrders テーブル内の SalesRepresentative が Employees テーブル内のものと同じになるように制限されます。各ローには、注文とその注文を担当する営業担当者についての情報が入っています。

たとえば、次のクエリでは、最初の ON 句を使用して SalesOrders を Customers にジョインします。また、2 番目の ON 句を使用して、テーブル式 (SalesOrders JOIN Customers) をベーステーブル SalesOrderItems にジョインします。

```
SELECT *
FROM SalesOrders JOIN Customers
   ON SalesOrders.CustomerID = Customers.ID
 JOIN SalesOrderItems
   ON SalesOrderItems.ID = SalesOrders.ID;
```

ON 句でのテーブルの参照

ON 句で参照されるテーブルは、その ON 句が修飾するジョインの一部である必要があります。たとえば、次の構文は無効です。

```
FROM ( A KEY JOIN B ) JOIN ( C JOIN D ON A.x = C.x )
```

ここでの問題は、ジョイン条件 A.x = C.x がテーブル A を参照していることです。テーブル A は、このジョイン条件が修飾するジョイン (この場合 C JOIN D) の一部ではありません。

ただし、ANSI/ISO 規格 SQL99 と SQL Anywhere 7.0 では、この規則には例外があります。つまり、テーブル式の間にカンマを使用すると、ジョインの ON 条件は、構文中でその ON 条件より前にある FROM 句内のテーブルを参照できます。このため、次の構文は有効になります。

```
FROM (A KEY JOIN B) , (C JOIN D ON A.x = C.x)
```

参照

- 「キージョイン」520 ページ
- 「カンマ」500 ページ

例

次の例では、SalesOrders テーブルを Employees テーブルにジョインします。結果の各ローは、SalesOrders テーブルの SalesRepresentative カラムの値と Employees テーブルの EmployeeID カラムの値が一致するローに対応しています。

```
SELECT Employees.Surname, SalesOrders.ID, SalesOrders.OrderDate
FROM SalesOrders
JOIN Employees
ON SalesOrders.SalesRepresentative = Employees.EmployeeID;
```

Surname	ID	OrderDate
Chin	2008	4/2/2001
Chin	2020	3/4/2001
Chin	2032	7/5/2001
Chin	2044	7/15/2000
Chin	2056	4/15/2001
...	...	...

次はこの例に関する説明です。

- このクエリの結果に含まれているのは、648 個のロー (SalesOrders テーブルの各ローに対応) のみです。直積における 48,600 のローのうち、2 つのテーブルで同じ従業員番号を持っているのは 648 のローだけだからです。

- 結果の順序に意味はありません。ORDER BY 句を追加すると、クエリに特定の順序を指定できます。

- ON 句によって、最終的な結果セットには含まれないカラムが組み込まれます。

生成されたジョインと ON 句

キーワード JOIN を使用し、ジョインタイプを指定していない場合、ON 句を使用しなければ、キージョインがデフォルトです。指定のない JOIN とともに ON 句を使用すると、キージョインはデフォルトにはならず、生成されたジョイン条件は何も適用されません。

たとえば、次の例はキージョインです。キーワード JOIN が使用されており、ON 句もない場合はキージョインがデフォルトだからです。

```
SELECT *
FROM A JOIN B;
```

次は、テーブル A とテーブル B のジョインであり、ジョイン条件 A.x = B.y も使用されています。したがって、このジョインはキージョインではありません。

```
SELECT *
FROM A JOIN B ON A.x = B.y;
```

KEY JOIN または NATURAL JOIN を指定し、さらに ON 句も使用すると、最終的なジョイン条件は、生成されたジョイン条件と明示的ジョイン条件の論理積になります。たとえば、次の文にはジョイン条件が 2 つ入っています。1 つはキージョインから生成されたジョイン条件で、もう 1 つは ON 句で明示的に記述されたジョイン条件です。

```
SELECT *
FROM A KEY JOIN B ON A.x = B.y;
```

キージョインによって生成されたジョイン条件が A.w = B.z の場合、次の文は上の文と同等です。

```
SELECT *
FROM A JOIN B
  ON A.x = B.y
  AND A.w = B.z;
```

参照

- 「キージョイン」520 ページ

明示的ジョイン条件の種類

ジョイン条件は、そのほとんどが等号に基づいているため「等価ジョイン」と呼ばれます。次に例を示します。

```
SELECT *
FROM Departments JOIN Employees
   ON Departments.DepartmentID = Employees.DepartmentID;
```

ただし、ジョイン条件の中で必ず等号 (=) を使うわけではありません。LIKE、SOUNDEX、BETWEEN、> (より大きい)、!= (等しくない) などの探索条件を使用できます。

例

次の例は、「在庫数以上の受注があったのはどの製品か」という質問に対する回答です。

```
SELECT DISTINCT Products.Name
FROM Products JOIN SalesOrderItems
ON Products.ID = SalesOrderItems.ProductID
   AND SalesOrderItems.Quantity > Products.Quantity;
```

参照

- 「探索条件」『SQL Anywhere サーバ SQL リファレンス』

ジョイン条件内の WHERE 句

外部ジョインを使用する場合を除き、ON 句の代わりに WHERE 句でジョイン条件を指定できます。ただし、外部ジョインがクエリに含まれる場合には、ON 句と WHERE 句には意味の違いが生じます。

ON 句は、FROM 句の一部であるため、WHERE 句よりも前に処理されます。このことは、外部ジョインについて、WHERE 句を使用することによって内部ジョインに変換できる場合を除き、結果には影響を及ぼしません。

ジョイン条件を ON 句にいれるか、WHERE 句に入れるかを決定するときには、次の規則を考慮してください。

- 外部ジョインを指定するときに WHERE 句にジョイン条件を入れると、外部ジョインが内部ジョインに変換されます。
- ON 句内の条件は、これと関連付けられた JOIN で結合するテーブル式内のテーブルのみ参照できます。ただし、WHERE 句内の条件は、その条件がジョインの一部になっていなくても、任意のテーブルを参照できます。
- ON 句はキーワード CROSS JOIN とともには使用できませんが、WHERE 句はいつでも使用できます。
- ジョイン条件が ON 句内にある場合、キージョインはデフォルトにはなりません。ただし、ジョイン条件が WHERE 句内にあると、キージョインをデフォルトにできます。

このマニュアルの例では、ON 句の中でジョイン条件を使用しています。外部ジョインを使用する場合は、これが必要です。他のジョインでも、それらがジョイン条件であって一般的な探索条件ではないことを明確にするために、ON 句の中でジョイン条件が使用されています。

参照

- 「外部ジョインとジョインの条件」502 ページ
- 「キージョイン」520 ページ

クロスジョイン

2 つのテーブルのクロスジョインによって、両方のテーブルにあるローの組み合わせで可能なものすべてが生成されます。クロスジョインは、直積とも呼ばれます。

1 番目のテーブルの各ローは、2 番目のテーブルの各ローとともに 1 回だけ出現します。したがって、結果セットのローの数は、1 番目のテーブルにあるローの数と 2 番目のテーブルにあるローの数の積から、WHERE 句による制限で除外されたローの数を減算した数になります。

クロスジョインでは ON 句を使用できません。ただし、WHERE 句で制限を設けることはできます。

クロスジョインには適用しない内部変更子と外部変更子

WHERE 句で追加した制限がある場合を除いて、両テーブルのすべてのローはいつでもクロスジョインの結果として表示されます。したがって、INNER、LEFT OUTER、RIGHT OUTER のキーワードは、クロスジョインには適用できません。

たとえば、次の文では 2 つのテーブルが結合されます。

```
SELECT *
FROM A CROSS JOIN B;
```

このクエリの結果セットには、A のすべてのカラムと B のすべてのカラムが含まれます。A の 1 つのローと B の 1 つのローの組み合わせそれぞれに対して、結果セットに 1 つのローがあります。A が n 個のロー、B が m 個のローである場合は、クエリによって $n \times m$ 個のローが返されます。

カンマ

カンマはジョイン操作とは異なりますが、似た役割を持っています。カンマは、CROSS JOIN キーワードとまったく同じ直積を作成します。ただし、JOIN キーワードはテーブル式を生成しますが、カンマはテーブル式のリストを生成します。

次は、2 つのテーブルの簡単な内部ジョインの例です。この例では、カンマと CROSS JOIN キーワードは同義です。

```
SELECT *
FROM A, B, C
WHERE A.x = B.y;

SELECT *
FROM A CROSS JOIN B CROSS JOIN C
WHERE A.x = B.y;
```

通常は、カンマをキーワード CROSS JOIN の代わりに使用できます。カンマを使用したテーブル式に含まれる生成されたジョイン条件を除いて、カンマ構文はクロスジョイン構文と同義です。

スタージョイン構文の場合、カンマには特殊な用途があります。

参照

- 「テーブル式のキージョイン」524 ページ
- 「ジョインで重複する相関名 (スタージョイン)」509 ページ

内部ジョインと外部ジョイン

キーワード INNER、LEFT OUTER、RIGHT OUTER、FULL OUTER は、キージョイン、ナチュラルジョイン、ON 句付きジョインを修飾するときに使用します。デフォルトは INNER です。これらの変更子はクロスジョインには適用されません。

内部ジョイン

デフォルトのジョインは「内部ジョイン」です。ジョイン条件を満たすローだけが結果セットに含まれます。

例

たとえば、次のクエリの結果セットで、それぞれのローには、キージョイン条件を満たす 1 つの Customers ローと 1 つの SalesOrders ローの情報が含まれます。ある顧客が発注しなかった場合は、条件が満たされないので、この顧客に対応するローは結果セットには含まれません。

```
SELECT GivenName, Surname, OrderDate
FROM Customers KEY INNER JOIN SalesOrders
ORDER BY OrderDate;
```

GivenName	Surname	OrderDate
Hardy	Mums	2000-01-02
Aram	Najarian	2000-01-03
Tommie	Wooten	2000-01-03
Alfredo	Margolis	2000-01-06
...	...	...

内部ジョインとキージョインはデフォルトなので、次のように FROM 句を使用しても前述の例と同じ結果が得られます。

```
SELECT GivenName, Surname, OrderDate
FROM Customers JOIN SalesOrders
ORDER BY OrderDate;
```

外部ジョイン

通常は、ジョイン条件を満たす場合のみローを返すジョインを作成します。これは内部ジョインと呼ばれ、クエリ時に使用されるデフォルトのジョインです。ただし、1 つのテーブルのすべてのローを保護したい場合があります。そのような場合は「外部ジョイン」を使用します。

2 つのテーブルの左または右の「外部ジョイン」を使用すると、一方のテーブルではすべてのローが保護され、他方のテーブルにはジョイン条件が満たされないときに NULL が入力されます。「左外部ジョイン」では左側のテーブルのローがすべて保護され、「右外部ジョイン」では右側テーブルのローがすべて保護されます。「全外部ジョイン」では、両方のテーブルのすべてのローが保護され、両方のテーブルが NULL 入力となります。

左外部ジョインまたは右外部ジョインのそれぞれの側のテーブル式は、「保護された」テーブル式と「NULL 入力」テーブル式と呼ばれます。左外部ジョインでは、左側のテーブル式が保護テーブル式で、右側のテーブル式は NULL 入力テーブル式です。全外部ジョインでは、左側と右側の両方のテーブル式が保護テーブル式であり、両方が NULL 入力テーブル式となります。

例

次の文にはすべての顧客が含まれます。顧客が注文していない場合には、注文情報に対応する結果のそれぞれのカラムに NULL 値が入ります。

```
SELECT Surname, OrderDate, City
FROM Customers LEFT OUTER JOIN SalesOrders
  ON Customers.ID = SalesOrders.CustomerID
WHERE Customers.State = 'NY'
ORDER BY OrderDate;
```

Surname	OrderDate	City
Thompson	(NULL)	Bancroft
Reiser	2000-01-22	Rockwood
Clarke	2000-01-27	Rockwood
Mentary	2000-01-30	Rockland
...	...	...

この文の外部ジョインは次のように解釈できます。ここで説明しているのは概念であり、SQL Anywhere が実際に結果セットを計算する過程を示すものではありません。

- 顧客からの発注ごとにローが 1 つ返されます。1 つの発注に対して 1 つのローが返されるため、顧客が 2 つ以上注文した場合には、ローも 2 つ以上返されます。これは内部ジョインの結果と同じです。ON 条件は、customer ローと sales order ローを一致させるために使用します。この手順では、WHERE 句は使用されません。

- 注文しなかったそれぞれの顧客のローが 1 行入ります。これにより、Customers テーブルのすべてのローが確実に含まれます。これらのすべてのローに対して、SalesOrders のカラムに NULL が挿入されます。キーワード OUTER が使用されているため、これらのローは追加されますが、内部ジョインには表示されません。この手順では ON 条件も WHERE 句も使用されません。

- WHERE 句を使用して、New York 在住ではない顧客のローをすべて除外します。

参照

- 「Transact-SQL の外部ジョイン (*= or =*)」505 ページ
- 「キージョイン」520 ページ

外部ジョインとジョインの条件

外部ジョインでよくある間違いは、ジョイン条件を置く場所に関するものです。WHERE 句を使用して NULL 入力テーブルを制限すると、通常、そのジョインは内部ジョインと同義になります。

これは、ほとんどの探索条件では、入力した探索条件のうち 1 つでも NULL になっていると TRUE と評価できないためです。NULL 入力テーブルに対する WHERE 句での制限によって、そ

れぞれの値は NULL と比較され、結果セットからそのローは除外されます。保護されたテーブルのローは保護されないので、このジョインは内部ジョインです。

これに対する例外は、入力値のいずれかが NULL の場合は TRUE と評価できる比較です。こうした比較には、IS NULL、IS UNKNOWN、IS FALSE、IS NOT TRUE があります。また、ISNULL や COALESCE を含む式もあります。

例

たとえば、次の文は左外部ジョインを計算します。

```
SELECT *
FROM Customers KEY LEFT OUTER JOIN SalesOrders
  ON SalesOrders.OrderDate < '2000-01-03';
```

これに対し、次の文は内部ジョインを作成します。

```
SELECT Surname, OrderDate
FROM Customers KEY LEFT OUTER JOIN SalesOrders
  WHERE SalesOrders.OrderDate < '2000-01-03';
```

この 2 つの文のうち、最初の文は次のように考えられます。まず、Customers テーブルを SalesOrders テーブルに左外部ジョインします。結果セットには Customers テーブルのすべてのローが入ります。2000 年 1 月 3 日より前に注文をしていない顧客については、sales order フィールドに NULL が入ります。

2 番目の文では、まず Customers と SalesOrders を左外部ジョインします。結果セットには Customers テーブルのすべてのローが入ります。注文をしていない顧客については、sales order フィールドに NULL が入ります。次に、2000 年 1 月 3 日以降に発注した顧客のローだけを選択することによって WHERE 条件が適用されます。発注しなかった顧客については、これらの値が NULL になります。任意の値を NULL と比較した結果は、UNKNOWN と評価されます。したがって、これらのローは削除されて、文は内部ジョインに縮小されます。

参照

- 「探索条件」『SQL Anywhere サーバ SQL リファレンス』

複雑な外部ジョイン

クエリに外部ジョインを使用したテーブル式が含まれるときは、ジョインの順序が重要になります。たとえば、A JOIN B LEFT OUTER JOIN C は、(A JOIN B) LEFT OUTER JOIN C と解釈されます。テーブル式 (A JOIN B) が C にジョインされます。このとき、テーブル式 (A JOIN B) は保護され、テーブル C には NULL が入力されます。

次に、以下の文を考えてみます。A、B、C はテーブルです。

```
SELECT *
FROM A LEFT OUTER JOIN B RIGHT OUTER JOIN C;
```

この文を理解するには、まず「SQL Anywhere では文が左から右に評価され、カッコが追加される」という規則を思い出してください。結果は次のようになります。

```
SELECT *
FROM (A LEFT OUTER JOIN B) RIGHT OUTER JOIN C;
```

次に、両方のジョインが同じタイプになるように、右外部ジョインを左外部ジョインに変換します。これを行うには、右外部ジョインにあるテーブルの位置を単純に逆にします。結果は次のようになります。

```
SELECT *
FROM C LEFT OUTER JOIN (A LEFT OUTER JOIN B);
```

ネストした外部ジョインでは、A が保護テーブルであり、B が NULL 入力テーブルです。最初の外部ジョインでは C が保護テーブルです。

この結合は次のように解釈できます。

- A を B に結合します。このとき、A のローはすべて保護されます。
- 次に、C を A と B のジョインの結果に結合します。このとき、C のローはすべて保護されます。

このジョインには ON 句がないため、デフォルトのキージョインになります。

また、外部ジョインのジョイン条件には、必ず、FROM 句内で先に参照されているテーブルだけを入れます。この制限事項は ANSI/ISO 標準に基づくものであり、あいまいさを排除するためのものです。たとえば、次の 2 つの文は構文的に正しくありません。テーブル自体が参照される前に C がジョイン条件内で参照されるからです。

```
SELECT *
FROM (A LEFT OUTER JOIN B ON B.x = C.x) JOIN C;
```

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = C.x, C;
```

参照

- 「カンマを含まないテーブル式のキージョイン」524 ページ

ビューと派生テーブルの外部ジョイン

外部ジョインは、ビューと派生テーブルにも指定できます。

次に例を示します。

```
SELECT *
FROM V LEFT OUTER JOIN A ON (V.x = A.x);
```

この例は、次のように解釈できます。

- ビュー V が計算されます。
- ジョイン条件 V.x = A.x を使用して V のローをすべて保護すると、計算されたビュー V のすべてのローが A にジョインされます。

例

次の例では、ビュー V を定義します。ここで、ビュー V は、$60,000 を上回る収入がある女性の従業員 ID と部署名を返します。

```
CREATE VIEW V AS
SELECT Employees.EmployeeID, DepartmentName
  FROM Employees JOIN Departments
    ON Employees.DepartmentID = Departments.DepartmentID
  WHERE Sex = 'F' and Salary > 60000;
```

次に、このビューを使用してそれらの女性が勤務する部署と販売地区のリストを追加します。ビュー V は保護ビューであり、SalesOrders は NULL 入力です。

```
SELECT DISTINCT V.EmployeeID, Region, V.DepartmentName
  FROM V LEFT OUTER JOIN SalesOrders
    ON V.EmployeeID = SalesOrders.SalesRepresentative;
```

EmployeeID	Region	DepartmentName
243	(NULL)	R & D
316	(NULL)	R & D
529	(NULL)	R & D
902	Eastern	Sales
...	...	...

Transact-SQL の外部ジョイン (*= or =*)

注意
Transact-SQL 外部ジョイン演算子 *= と =* は旧式であるため、将来のリリースではサポートから除外されます。

SQL Anywhere では ANSI/ISO SQL 標準に基づき、キーワード LEFT OUTER、RIGHT OUTER、FULL OUTER をサポートします。また、バージョン 12 以前の Adaptive Server Enterprise との互換性を保つために、tsql_outer_joins データベースオプションが On に設定されている場合は、LEFT OUTER と RIGHT OUTER のキーワードに対応する Transact-SQL の *= と =* もサポートします。

Transact-SQL のセマンティックには、いくつかの制限事項と潜在的な問題があります。Transact-SQL 外部ジョインの詳細については、http://www.sybase.com/detail?id=1017447 にあるホワイトペーパー『**Transact-SQL 外部ジョインのセマンティックと互換性**』を参照してください。

Transact-SQL ダイアレクトでは、FROM 句内にカンマで区切られたテーブルのリストを指定し、WHERE 句内で特殊な演算子 *= or =* を使用して外部ジョインを作成します。Adaptive Server Enterprise のバージョン 12 より前のバージョンでは、ジョイン条件を WHERE 句内に記述してください (ON はサポートされていませんでした)。

警告
外部ジョインを作成するときには、*= 構文と ON 句の構文を混在させないでください。この制限は、クエリで参照されるビューにも適用されます。

参照

- 「tsql_outer_joins オプション」『SQL Anywhere サーバ データベース管理』

例

次の左外部ジョインは全顧客をリストし、注文があればその日付を取り出します。

```
SELECT GivenName, Surname, OrderDate
FROM Customers, SalesOrders
WHERE Customers.ID *= SalesOrders.CustomerID
ORDER BY OrderDate;
```

この文は次の文と同義です。ここでは ANSI/ISO 構文が使用されています。

```
SELECT GivenName, Surname, OrderDate
FROM Customers LEFT OUTER JOIN SalesOrders
ON Customers.ID = SalesOrders.CustomerID
ORDER BY OrderDate;
```

Transact-SQL 外部ジョインの制限事項

> **注意**
> Transact-SQL 外部ジョイン演算子 *= と =* は旧式であるため、将来のリリースではサポートから除外されます。

Transact-SQL 外部ジョインにはいくつか制限があります。

- 外部ジョインと、外部ジョインの NULL 入力テーブルのカラムに対して条件を指定した場合、予想外の結果になることがあります。クエリ内の条件は結果セットからローを除外するのではなく、結果セットに含まれるローの値の方にも影響を与えます。条件を満たさないローについては、NULL 入力テーブルに NULL 値が入ります。

- ANSI/ISO SQL 構文と Transact-SQL 外部ジョイン構文を、1 つのクエリ内で混在させることはできません。ビューが外部ジョインのダイアレクトを使用して定義されている場合、そのビューのすべての外部ジョインクエリに同じダイアレクトを使用する必要があります。

- 1 つの NULL 入力テーブルを Transact-SQL 外部ジョインと通常のジョインの両方、または 2 つの外部ジョインに使用することはできません。たとえば、次の WHERE 句は、テーブル S がこの制限事項に違反しているため無効です。

  ```
  WHERE R.x *= S.x
  AND S.y = T.y
  ```

  外部ジョインと通常のジョイン句の両方で同じテーブルを使用しないようにクエリを書き直すことができない場合には、文を 2 種類のクエリに分けるか、ANSI/ISO SQL 構文のみを使用してください。

- 外部ジョインの NULL 入力テーブルを含むジョイン条件を持つサブクエリは使用できません。たとえば、次の WHERE 句は許可されません。

  ```
  WHERE R.x *= S.y
  AND EXISTS ( SELECT *
  ```

```
FROM T
WHERE T.x = S.x )
```

ビューと Transact-SQL 外部ジョイン

外部ジョインでビューを定義し、外部ジョインの NULL 入力テーブルからのカラムに対する条件でビューに問い合わせると、予期しない結果になる場合があります。クエリは NULL 入力テーブルからすべてのローを戻します。その条件に一致しないローはそのローの適切なカラム内に NULL 値を表示します。

次の規則によって、外部ジョインを含むビューを使用してカラムに実行できる更新の種類が決定します。

- INSERT 文と DELETE 文は外部ジョインビューでは使用できない。
- UPDATE 文は外部ジョインビューで使用できる。ビューの定義が WITH CHECK オプションの場合、複数のテーブルからのカラムを含む式の中の WHERE 句に、影響を受けるカラムがあると、更新は失敗する。

Transact-SQL ジョインに対する NULL の影響

Transact-SQL 外部ジョインでは、ジョインされるテーブルまたはビューの NULL 値は、互いに一致することはありません。NULL 値と他の NULL 値を比較した結果は、FALSE になります。

特殊なジョイン

この項では、セルフジョイン、スタージョイン、派生テーブルを使用したジョインなど、特殊なジョインについて説明します。

セルフジョイン

「セルフジョイン」では、異なる相関名を使用して同一テーブルを参照することによって、テーブルがそれ自体にジョインされます。

例 1

次のセルフジョインは従業員のペアのリストを作成します。各従業員の名前が全従業員の名前との組み合わせで表示されます。

```
SELECT a.GivenName, a.Surname,
    b.GivenName, b.Surname
FROM Employees AS a CROSS JOIN Employees AS b;
```

GivenName	Surname	GivenName	Surname
Fran	Whitney	Fran	Whitney
Fran	Whitney	Matthew	Cobb

GivenName	Surname	GivenName	Surname
Fran	Whitney	Philip	Chin
Fran	Whitney	Julie	Jordan
...	...	...	...

Employees テーブルには 75 個のローがあるので、このジョインには 75 × 75 = 5625 のローがあります。これには、従業員が自分自身をリストしたローも含まれます。たとえば、次のようなローです。

GivenName	Surname	GivenName	Surname
Fran	Whitney	Fran	Whitney

同じ名前を 2 回含むローを除外する場合は、互いの従業員 ID は同じではならないというジョイン条件を追加します。

```
SELECT a.GivenName, a.Surname,
    b.GivenName, b.Surname
FROM Employees AS a CROSS JOIN Employees AS b
WHERE a.EmployeeID != b.EmployeeID;
```

この重複するローを除くと、ジョインは 75 × 74 = 5550 ローで構成されます。

この新しいジョインは各従業員が自分以外の従業員とペアになったローで構成されます。しかし、各ペアの名前の表示には 2 通りの順番があるので、各ペアは 2 度表示されます。たとえば、前述のジョインには次の 2 つのローがあります。

GivenName	Surname	GivenName	Surname
Matthew	Cobb	Fran	Whitney
Fran	Whitney	Matthew	Cobb

名前の順番が重要でない場合は、同一ペアの表示が一度だけになる (75 × 74) / 2 = 2775 ローのリストを作成できます。

```
SELECT a.GivenName, a.Surname,
    b.GivenName, b.Surname
FROM Employees AS a CROSS JOIN Employees AS b
WHERE a.EmployeeID < b.EmployeeID;
```

この文は、従業員 a の EmployeeID が従業員 b の EmployeeID より小さいローのみを選択して、重複する行を削除します。

例 2

次のセルフジョインは相関名 report と manager を使用して、Employees テーブルの 2 つのインスタンスを区別し、従業員とその管理者のリストを作成します。

```
SELECT report.GivenName, report.Surname,
  manager.GivenName, manager.Surname
FROM Employees AS report JOIN Employees AS manager
  ON (report.ManagerID = manager.EmployeeID)
ORDER BY report.Surname, report.GivenName;
```

この文から、次に一部を示すテーブルが作成されます。従業員名は左側の 2 つのカラムに、マネージャ名は右側に表示されます。

GivenName	Surname	GivenName	Surname
Alex	Ahmed	Scott	Evans
Joseph	Barker	Jose	Martinez
Irene	Barletta	Scott	Evans
Jeannette	Bertrand	Jose	Martinez
...	...	...	...

例 3

次のセルフジョインによって、2 つのレベルのレポートを持つすべてのマネージャと 2 番目のレベルのレポート数のリストが生成されます。

```
SELECT higher.managerID, count(*) second_level_reports
FROM employees lower JOIN employees higher
  ON ( lower.managerID = higher.employeeID )
GROUP BY higher.managerID
ORDER BY higher.managerID DESC;
```

上記のクエリの結果には、次のローが含まれています。

ManagerID	second_level_reports
1293	30
902	23
501	22

ジョインで重複する相関名 (スタージョイン)

重複するテーブル名は、「スタージョイン」を作成するために使用します。スタージョインでは、1 つのテーブルまたはビューが複数のテーブルやビューにジョインされます。

スタージョインを作成するには、同じテーブル名、ビュー名、または相関名を FROM 句内で 2 回以上使用します。これは、ANSI/ISO SQL 標準の拡張機能です。重複名を使用しても機能は追加されませんが、使用すると特定のクエリを簡単に作成できます。

重複名は、構文が意味をなすように、必ず異なるジョイン内に置きます。同一ジョイン内でテーブル名やビュー名を 2 回使用すると、2 番目のインスタンスは無視されます。たとえば、FROM A,A と FROM A CROSS JOIN A は、どちらも FROM A と解釈されます。

次の例は SQL Anywhere で有効です。A、B、C はテーブルです。この例では、テーブル A の同一インスタンスは B と C のどちらにもジョインされます。スタージョインでジョインを分けるときにはカンマが必要です。スタージョインでのカンマの使用方法は、スタージョインの構文特有のものです。

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = B.x,
    A LEFT OUTER JOIN C ON A.y = C.y;
```

これは、次の例と同義です。

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = B.x,
    C RIGHT OUTER JOIN A ON A.y = C.y;
```

2 つの例はどちらも次の ANSI/ISO 標準構文と同義です (カッコはオプションです)。

```
SELECT *
FROM (A LEFT OUTER JOIN B ON A.x = B.x)
LEFT OUTER JOIN C ON A.y = C.y;
```

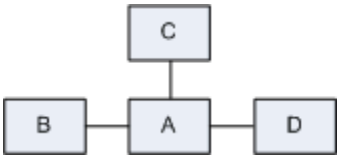
次の例では、テーブル A が 3 つのテーブル B、C、D にジョインされます。

```
SELECT *
FROM A JOIN B ON A.x = B.x,
    A JOIN C ON A.y = C.y,
    A JOIN D ON A.w = D.w;
```

上の例は、次の ANSI/ISO 標準構文と同義です (カッコはオプションです)。

```
SELECT *
FROM ((A JOIN B ON A.x = B.x)
JOIN C ON A.y = C.y)
JOIN D ON A.w = D.w;
```

複雑なジョインは図にするとわかりやすくなります。前述の例は次の図で説明できます。この図から、テーブル B、C、D がテーブル A を介してジョインされることがわかります。



注意
重複するテーブル名を使用できるのは、extended_join_syntax オプションが On (デフォルト) になっている場合だけです。

-

例 1

Rollin Overbey に発注した顧客名リストを作成します。FROM 句にある Employees テーブルのどのカラムも結果に表示されないことに注意してください。また、Customers.id や Employees.EmployeeID など、ジョインしたどのカラムも結果に表示されません。それでも、FROM 句に Employees テーブルを使用するだけで、このジョインは可能です。

```
SELECT Customers.GivenName, Customers.Surname,
   SalesOrders.OrderDate
FROM   SalesOrders KEY JOIN Customers,
       SalesOrders KEY JOIN Employees
WHERE Employees.GivenName = 'Rollin'
  AND Employees.Surname = 'Overbey'
ORDER BY SalesOrders.OrderDate;
```

GivenName	Surname	OrderDate
Tommie	Wooten	2000-01-03
Michael	Agliori	2000-01-08
Salton	Pepper	2000-01-17
Tommie	Wooten	2000-01-23
...	...	...

次の例は、ANSI/ISO 標準構文の文と同義です。

```
SELECT Customers.GivenName, Customers.Surname,
  SalesOrders.OrderDate
FROM SalesOrders JOIN Customers
  ON SalesOrders.CustomerID =
   Customers.ID
JOIN Employees
  ON SalesOrders.SalesRepresentative =
   Employees.EmployeeID
WHERE Employees.GivenName = 'Rollin'
  AND Employees.Surname = 'Overbey'
ORDER BY SalesOrders.OrderDate;
```

例 2

次の例では、「各顧客が製品ごとに発注した数はどれくらいか、また受注した営業部員のマネージャは誰か」という質問に回答します。

回答するために、まず、どの情報を取り出すのかをリストします。ここでは、製品、数量、顧客名、管理者名を取り出します。次に、これらの情報を保持しているテーブルをリストします。ここでは、Products、SalesOrderItems、Customers、Employees になります。SQL Anywhere サンプルデータベースの構造を見ると、これらのテーブルがすべて SalesOrders テーブルを介して関連していることがわかります。SalesOrders テーブルにスタージョインを作成すると、他のテーブルから情報を取り出すことができます。

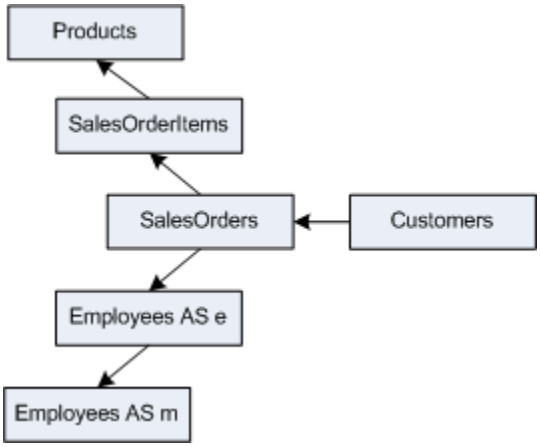
また、管理者名を取得するには、セルフジョインを作成する必要があります。Employees テーブルには、管理者の ID 番号とすべての従業員の名前はありますが、管理者名だけをリストしたカラムがないからです。

次の文は SalesOrders テーブルを中心にしてスタージョインを作成します。このジョインは、結果セットにすべての顧客が含まれるように、すべて外部ジョインになっています。注文しなかった顧客もいますが、そういう顧客の他の値は NULL になっています。結果セットのカラムは、Customers、Products、Quantity ordered、営業部員の管理者名です。

```
SELECT Customers.GivenName, Products.Name,
  SUM(SalesOrderItems.Quantity), m.GivenName
FROM SalesOrders
    KEY RIGHT OUTER JOIN Customers,
   SalesOrders
    KEY LEFT OUTER JOIN SalesOrderItems
    KEY LEFT OUTER JOIN Products,
      SalesOrders
    KEY LEFT OUTER JOIN Employees AS e
    LEFT OUTER JOIN Employees AS m
        ON (e.ManagerID = m.EmployeeID)
WHERE Customers.State = 'CA'
GROUP BY Customers.GivenName, Products.Name, m.GivenName
ORDER BY SUM(SalesOrderItems.Quantity) DESC,
   Customers.GivenName;
```

GivenName	Name	SUM(SalesOrderItems .Quantity)	GivenName
Sheng	Baseball Cap	240	Moira
Laura	Tee Shirt	192	Moira
Moe	Tee Shirt	192	Moira
Leilani	Sweatshirt	132	Moira
...	...	...	...

以下は、このスタージョインを使ったテーブルの図です。矢印は、外部ジョインの方向 (右または左) を示します。顧客の完全なリストはすべてのジョイン全体で保持されます。



次に示す ANSI/ISO 標準構文は、例 2 のスタージョインと同義です。

```
SELECT Customers.GivenName, Products.Name,
  SUM(SalesOrderItems.Quantity), m.GivenName
FROM SalesOrders LEFT OUTER JOIN SalesOrderItems
    ON SalesOrders.ID = SalesOrderItems.ID
  LEFT OUTER JOIN Products
    ON SalesOrderItems.ProductID = Products.ID
  LEFT OUTER JOIN Employees as e
    ON SalesOrders.SalesRepresentative = e.EmployeeID
  LEFT OUTER JOIN Employees as m
    ON e.ManagerID = m.EmployeeID
  RIGHT OUTER JOIN Customers
    ON SalesOrders.CustomerID = Customers.ID
WHERE Customers.State = 'CA'
GROUP BY Customers.GivenName, Products.Name, m.GivenName
ORDER BY SUM(SalesOrderItems.Quantity) DESC,
   Customers.GivenName;
```

参照

- 「サンプルデータベーススキーマ」490 ページ
- 「extended_join_syntax オプション」『SQL Anywhere サーバ データベース管理』
- 「セルフジョイン」507 ページ

派生テーブルを使用するジョイン

派生テーブルでは FROM 句内にクエリをネストできます。派生テーブルを使用すると、別のビューやテーブルを作成してそれにジョインすることなく、グループのグループ化を実行したり、グループを使用してジョインを構成することができます。

次の例では、内側の SELECT 文 (カッコに囲まれている) は顧客 ID 値でグループ分けされた派生テーブルを作成します。外側の SELECT 文はこのテーブルに相関名 sales_order_counts を割り当て、ジョイン条件を使用してそれを Customers テーブルとジョインします。

```
SELECT Surname, GivenName, number_of_orders
FROM Customers JOIN
  (  SELECT CustomerID, COUNT(*)
     FROM SalesOrders
```

```
    GROUP BY CustomerID  )
  AS sales_order_counts ( CustomerID, number_of_orders )
  ON ( Customers.ID = sales_order_counts.CustomerID )
WHERE number_of_orders > 3;
```

結果は各顧客の注文数を含む、4 件以上の注文をした顧客名のテーブルです。

参照

- 「ビューと派生テーブルのキージョイン」529 ページ
- 「ビューと派生テーブルのナチュラルジョイン」519 ページ
- 「ビューと派生テーブルの外部ジョイン」504 ページ

適用式から生成されるジョイン

適用式を使用すると、右側が左側に依存するジョインを簡単に指定できます。たとえば、適用式を使用して、テーブル式内のローごとに 1 回ずつプロシージャまたは派生テーブルを評価できます。適用式は SELECT 文の FROM 句内に配置し、ON 句を使用することはできません。

APPLY を使用すると複数のソースからローを結合できます。この動作は JOIN に似ていますが、APPLY には ON 条件を指定することはできません。APPLY と JOIN の主な相違点は、APPLY の右側は左側の現在のローによって変わる場合があるという点です。左側のローごとに、右側が再計算され、結果のローが左側のローに結合されます。左側の 1 つのローが右側の複数の行を返すケースでは、右側から返されたローの数だけ左側が重複する結果となります。

指定できる APPLY には、CROSS APPLY と OUTER APPLY の 2 つのタイプがあります。CROSS APPLY は、右側の結果を生成する左側のローのみを返します。OUTER APPLY は、CROSS APPLY が返すすべてのローと、(右側に NULL が入力されたため) 右側からローが返されない左側のすべてのローを返します。

適用式の構文は、次のとおりです。

table-expression { **CROSS** | **OUTER** } **APPLY** *table-expression*

例

次の例では、部署 ID を入力値として受け取り、その部署内で給与が 80,000 ドルを超えるすべての従業員の名前を返すプロシージャ EmployeesWithHighSalary を作成します。

```
CREATE PROCEDURE EmployeesWithHighSalary( IN dept INTEGER )
 RESULT ( Name LONG VARCHAR )
  BEGIN
   SELECT E.GivenName || ' ' || E.Surname
      FROM Employees E
      WHERE E.DepartmentID = dept AND E.Salary > 80000;
  END;
```

次のクエリでは、OUTER APPLY を使用して Departments テーブルを EmployeesWithHighSalary プロシージャの結果にジョインし、各部署で給与が 80,000 ドルを超えるすべての従業員の名前を返します。また、このクエリでは、右側が NULL のローを返し、給与が 80,000 ドルを超える従業員が存在しない各部署も示します。

```
SELECT D.DepartmentName, HS.Name
   FROM Departments D
   OUTER APPLY EmployeesWithHighSalary( D.DepartmentID ) AS HS;
```

DepartmentName	Name
R & D	Kim Lull
R & D	David Scott
R & D	John Sheffield
Sales	Moira Kelly
Finance	Mary Anne Shea
Marketing	NULL
Shipping	NULL

次のクエリでは、CROSS APPLY を使用して Departments テーブルを EmployeesWithHighSalary プロシージャの結果にジョインします。右側に NULL が指定されたローは含まれません。

```
SELECT D.DepartmentName, HS.Name
   FROM Departments D
   CROSS APPLY EmployeesWithHighSalary( D.DepartmentID ) AS HS;
```

DepartmentName	Name
R & D	Kim Lull
R & D	David Scott
R & D	John Sheffield
Sales	Moira Kelly
Finance	Mary Anne Shea

次のクエリでは、前述のクエリと同じ結果が返されますが、CROSS APPLY の右側として派生テーブルが使用されます。

```
SELECT D.DepartmentName, HS.Name
  FROM Departments D
   CROSS APPLY (
      SELECT E.GivenName || ' ' || E.Surname
         FROM Employees E
         WHERE E.DepartmentID = D.DepartmentID AND E.Salary > 80000
   ) HS( Name );
```

参照

- 「FROM 句」『SQL Anywhere サーバ SQL リファレンス』
- 「キージョイン」520 ページ
- 「クロスジョイン」499 ページ
- 「内部ジョインと外部ジョイン」500 ページ

ナチュラルジョイン

ナチュラルジョインを指定すると、同じ名前を持つカラムに基づいてジョイン条件が生成されます。生成されたジョイン条件がベーステーブルのナチュラルジョインに有効になるためには、同じ名前のカラムがどちらのテーブルにも少なくとも 1 つは存在する必要があります。共通するカラム名がなければ、エラーが発生します。

テーブル A と B が共通のカラム名を 1 つ持っており、そのカラムが x であるとします。その場合は次のようになります。

```
SELECT *
FROM A NATURAL JOIN B;
```

これは、次のクエリと同義です。

```
SELECT *
FROM A JOIN B
 ON A.x = B.x;
```

テーブル A と B が共通のカラム名を 2 つ持っており、そのカラムが a と b である場合、A NATURAL JOIN B は次のクエリと同等です。

```
A JOIN B
 ON A.a = B.a
 AND A.b = B.b;
```

例 1

たとえば、テーブル Employees と Departments には共通のカラム名 DepartmentID が 1 つあるため、ナチュラルジョインを使用してこの 2 つのテーブルをジョインできます。

```
SELECT GivenName, Surname, DepartmentName
FROM Employees NATURAL JOIN Departments
ORDER BY DepartmentName, Surname, GivenName;
```

GivenName	Surname	DepartmentName
Janet	Bigelow	Finance
Kristen	Coe	Finance
James	Coleman	Finance
Jo Ann	Davidson	Finance
...	...	...

次の文は同義です。この文では、さきほどの例で生成されたジョイン条件が明示的に指定されています。

```
SELECT GivenName, Surname, DepartmentName
FROM Employees JOIN Departments
  ON (Employees.DepartmentID = Departments.DepartmentID)
ORDER BY DepartmentName, Surname, GivenName;
```

例 2

Interactive SQL で次のクエリを実行します。

```
SELECT Surname, DepartmentName
FROM Employees NATURAL JOIN Departments;
```

Surname	DepartmentName
Whitney	R & D
Cobb	R & D
Breault	R & D
Shishov	R & D
Driscoll	R & D
...	...

SQL Anywhere は 2 つのテーブルを参照し、共通するカラム名が DepartmentID だけであると判断します。次の ON CLAUSE は内部的に生成され、ジョインの実行に使用されます。

```
FROM Employees JOIN Departments
  ON Employees.DepartmentID = Departments.DepartmentID
```

NATURAL JOIN は ON 句を入力するための単なるショートカットで、この 2 つのクエリは同じです。

NATURAL JOIN を使用した場合のエラー

NATURAL JOIN 演算子は、等価ではないカラムを等価と見なすと問題が発生する可能性があります。たとえば、次のクエリを実行すると、意図しない結果が生成されます。

```
SELECT *
FROM SalesOrders NATURAL JOIN Customers;
```

このクエリを実行してもローは返されません。内部的に次の ON 句が生成されます。

```
FROM SalesOrders JOIN Customers
  ON SalesOrders.ID = Customers.ID
```

SalesOrders テーブル内の ID カラムは注文の ID 番号です。Customers テーブル内の ID カラムは顧客の ID 番号です。これらはどれも一致しません。もちろん、一致があったとしても意味がありません。

ON 句を使用したナチュラルジョイン

NATURAL JOIN を指定し、かつ ON 句内にジョイン条件を置くと、2 つのジョイン条件の論理積が生成されます。

たとえば、次の 2 つのクエリは同義です。最初のクエリではジョイン条件 Employees.DepartmentID = Departments.DepartmentID が生成されます。このクエリには明示的ジョイン条件も含まれています。

```
SELECT GivenName, Surname, DepartmentName
FROM Employees NATURAL JOIN Departments
  ON Employees.ManagerID = Departments.DepartmentHeadID;
```

次のクエリは同義です。このクエリでは、前の例で生成されたナチュラルジョイン条件が ON 句で指定されています。

```
SELECT GivenName, Surname, DepartmentName
FROM Employees JOIN Departments
  ON Employees.ManagerID = Departments.DepartmentHeadID
   AND Employees.DepartmentID = Departments.DepartmentID;
```

テーブル式のナチュラルジョイン

ナチュラルジョインの少なくともどちらか一方の側に複数テーブルの式が 1 つ存在する場合、SQL Anywhere は、ジョイン演算子の左右にあるカラムを比較して、同じ名前のカラムを検索することでジョイン条件を生成します。

次の文を例にとります。

```
SELECT *
FROM (A JOIN B) NATURAL JOIN (C JOIN D);
```

この文には 2 つのテーブル式があります。テーブル式 A JOIN B のカラム名がテーブル式 C JOIN D のカラム名と比較されると、一致するカラム名のうちあいまいさのないペアに対してジョイン条件が生成されます。「一致するカラム名のうちあいまいさのないペア」とは、カラム名が両方のテーブル式に出現することがあっても同一テーブル式内では 2 回出現しないという意味です。

あいまいなカラム名のペアが 1 つでもあると、エラーになります。ただし、カラム名がもう一方のテーブル式内のカラム名とも一致しなければ、カラム名が同じテーブル式で 2 回出現しても構いません。

ナチュラルジョインリスト

ナチュラルジョインの少なくとも片方の側にテーブル式のリストがある場合、そのリスト内の各テーブル式に対して別のジョイン条件が生成されます。

次のテーブルを考えてみます。

●テーブル A はカラム a、b、c で構成されている。

●テーブル B は、カラム a と d で構成されている。

●テーブル C はカラム d と c で構成されている。

このような場合、SQL Anywhere ではジョイン (A,B) NATURAL JOIN C によって次の 2 つのジョイン条件が生成されます。

```
ON A.c = C.c
 AND B.d = C.d
```

A-C または B-C に共通のカラム名がなければ、エラーが発行されます。

テーブル C がカラム a、d、c で構成されている場合、ジョイン (A,B) NATURAL JOIN C は無効です。その理由は、カラム a が 3 つのテーブルすべてに出現するため、ジョインがあいまいになってしまうためです。

例

次の例は質問の答えを示しています。「販売した製品と販売担当者の情報を売り上げ別に提供してほしい」という質問に対する回答です。

```
SELECT *
FROM ( Employees KEY JOIN SalesOrders )
  NATURAL JOIN ( SalesOrderItems KEY JOIN Products );
```

これは次と同じです。

```
SELECT *
FROM ( Employees KEY JOIN SalesOrders )
  JOIN ( SalesOrderItems KEY JOIN Products )
    ON SalesOrders.ID = SalesOrderItems.ID;
```

ビューと派生テーブルのナチュラルジョイン

ANSI/ISO SQL 標準の拡張機能では、ナチュラルジョインのどちらの側にもビューまたは派生テーブルを指定できます。次の文を考えてみます。

```
SELECT *
FROM View1 NATURAL JOIN View2;
```

View1 内のカラムは View2 内のカラムと比較されます。たとえば、両方のビューにカラム EmployeeID が出現し、それと同じ名前を持つカラムがほかになければ、生成されるジョイン条件は (View1.EmployeeID = View2.EmployeeID)(View1.EmployeeID = View2.EmployeeID) になります。

例

次の例で、ナチュラルジョインに使用するビューにはカラムだけでなく式を指定することができ、これらの式やカラムはナチュラルジョインでは同じように扱われることを説明します。まず、カラム x のあるビュー V を次のように作成します。

```
CREATE VIEW V(x) AS
SELECT R.y + 1
FROM R;
```

次に、このビューから派生テーブルへのナチュラルジョインを作成します。派生テーブルには、カラム x があり、相関名 T が付けられています。

```
SELECT *
FROM V NATURAL JOIN (SELECT P.y FROM P) as T(x);
```

このジョインは、次のクエリと同義です。

```
SELECT *
FROM V JOIN (SELECT P.y FROM P) as T(x) ON (V.x = T.x);
```

キージョイン

一般的なジョインの多くは 2 つのテーブル間で外部キーによって関連付けられます。最も一般的なジョインは、外部キー値がプライマリキー値と同じになるように制限します。KEY JOIN 演算子は、外部キーの関係に基づいて 2 つのテーブルをジョインします。つまり、SQL Anywhere は、一方のテーブルのプライマリキーカラムを他方のテーブルの外部キーカラムと同等とする ON 句を生成します。キージョインを使用するには、テーブル間に外部キー関係が必要になります。この関係がない場合は、エラーになります。

キージョインは ON 句のショートカットで、この 2 つのクエリは同じです。ただし、ON 句は KEY JOIN でも使用できます。JOIN を指定しても CROSS、NATURAL、KEY を指定しない場合、または ON 句を使用する場合のデフォルトは、キージョインです。SQL Anywhere サンプルデータベースの図では、テーブル間を結ぶ線は外部キーを表します。KEY JOIN 演算子は、図の中で 1 本の線によって 2 つのテーブルがジョインされているところならどこでも使用できます。

キージョインがデフォルトの場合

次のすべてが当てはまる場合、SQL Anywhere ではキージョインがデフォルトになります。

- キーワード JOIN が使用されている。
- キーワード CROSS、NATURAL、または KEY が指定されていない。
- ON 句がない。

参照

- 「チュートリアル：サンプルデータベースへの接続」『SQL Anywhere サーバ データベース管理』

例

たとえば、次のクエリは、データベース内の外部キー関係に基づいてテーブル Products と SalesOrderItems をジョインします。

```
SELECT *
FROM Products KEY JOIN SalesOrderItems;
```

次のクエリは同義です。これには KEY がありませんが、ON 句のない JOIN はデフォルトで KEY JOIN になります。

```
SELECT *
FROM Products JOIN SalesOrderItems;
```

次のクエリも同義になります。ON 句で指定されているジョイン条件が、SQL Anywhere が SQL Anywhere サンプルデータベースの外部キー関係に基づいてこれらのテーブルに対して生成するジョイン条件と同じになるためです。

```
SELECT *
FROM Products JOIN SalesOrderItems
ON SalesOrderItems.ProductID = Products.ID;
```

ON 句を使用したキージョイン

KEY JOIN を指定し、かつ ON 句内にジョイン条件を置くと、2 つのジョイン条件の論理積が生成されます。次に例を示します。

```
SELECT *
FROM A KEY JOIN B
ON A.x = B.y;
```

A と B のキージョインによって生成されたジョイン条件が A.w = B.z の場合、前述の例は次のクエリと同等です。

```
SELECT *
FROM A JOIN B
ON A.x = B.y AND A.w = B.z;
```

複数の外部キー関係がある場合のキージョイン

外部キー関係に基づいてジョイン条件が生成されるときに、外部キー関係が 2 つ以上ある場合があります。このような場合、SQL Anywhere は、外部キーが参照するプライマリキーテーブルの相関名と、外部キーの役割名とを一致させることによって、使用する外部キー関係を決定します。

次の項では、SQL Anywhere でキージョインのジョイン条件が生成される過程について説明します。

相関名と役割名

「相関名」とは、クエリの FROM 句内で使用されるテーブル名またはビュー名のことです。元の名前か、FROM 句で定義されたエイリアスになります。

「役割名」は外部キーの名前です。役割名は、指定された外部 (子) テーブルに対してユニークでなければなりません。

外部キーに役割名を指定しない場合は、次のようにして名前が割り当てられます。

- プライマリテーブル名と同じ名前の外部キーが存在しない場合は、役割名にはプライマリテーブル名が割り当てられる。
- すでにプライマリテーブル名が別の外部キーによって使用されている場合は、役割名は、外部テーブルに対してユニークな、プライマリテーブル名と 0 埋め込みの 3 桁の数字が連結されたものになる。

外部キーの役割名がわからない場合は、Sybase Central で左ウィンドウ枠にあるデータベースコンテナーを展開すると検索できます。左ウィンドウ枠でテーブルを選択し、右ウィンドウ枠で **[制約]** タブをクリックします。右ウィンドウ枠にテーブルの外部キーのリストが表示されます。

ジョイン条件の生成

SQL Anywhere では、次のように、プライマリキーテーブルの相関名と同じ役割名を持つ外部キーが検索されます。

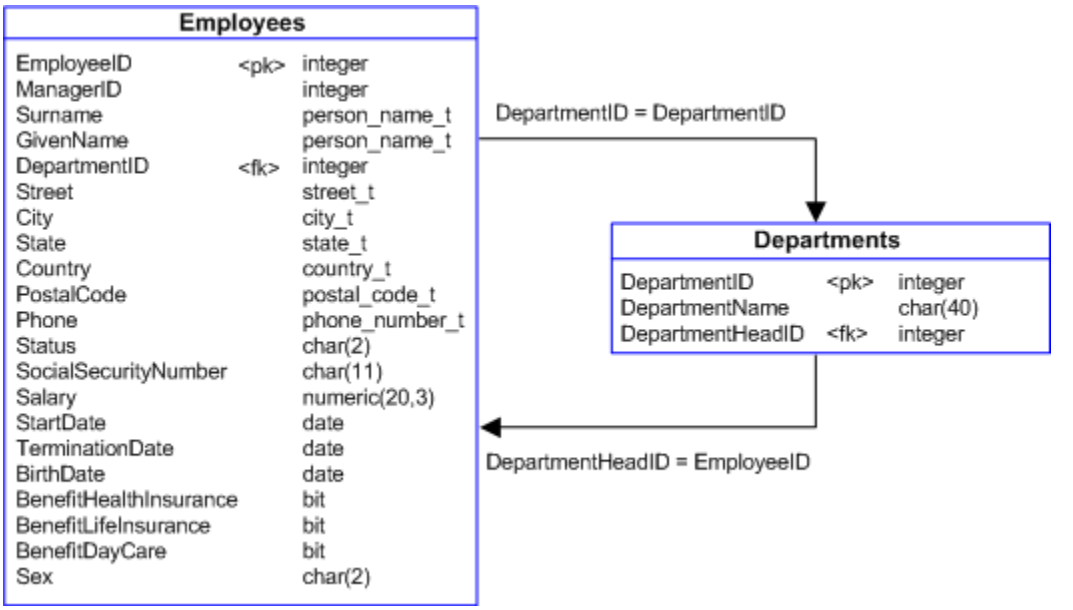
- ジョイン内のテーブルと同じ名前の外部キーが 1 つだけであれば、その外部キーが使用され、ジョイン条件が生成される。
- テーブルと同じ名前の外部キーが 2 つ以上見つかると、そのジョインはあいまいとなりエラーが発行される。
- テーブルと同じ名前の外部キーが 1 つもなければ、名前が一致しなくても外部キー関係が検索される。外部キー関係が 2 つ以上見つかれば、そのジョインはあいまいとなりエラーが発行される。

参照

- 「キージョイン操作規則」530 ページ
- 「サンプルデータベーススキーマ」490 ページ

例 1

SQL Anywhere サンプルデータベースでは、テーブル Employees とテーブル Departments の間で 2 つの外部キー関係が定義されています。Employees テーブルの外部キー FK_DepartmentID_DepartmentID は Departments テーブルを参照し、Departments テーブルの外部キー FK_DepartmentHeadID_EmployeeID は Employees テーブルを参照しています。



次のクエリはあいまいです。外部キー関係が 2 つあり、そのどちらにもプライマリキーテーブル名と同じ役割名が指定されていないからです。そのため、このクエリを実行しようとしても、結果は構文エラー SQLE_AMBIGUOUS_JOIN (-147) になります。

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees KEY JOIN Departments;
```

例 2

このクエリは、Departments テーブルに相関名 FK_DepartmentID_DepartmentID を指定して例 1 のクエリを修正したものです。ここで、外部キー FK_DepartmentID_DepartmentID にはその参照テーブルと同じ名前が付けられているため、ジョイン条件を定義するために使用されます。結果には、すべての従業員の姓と所属部署が入っています。

```
SELECT Employees.Surname,
   FK_DepartmentID_DepartmentID.DepartmentName
FROM Employees KEY JOIN Departments
   AS FK_DepartmentID_DepartmentID;
```

次のクエリは上の例と同義です。この例では、Departments テーブルのエイリアスを作成する必要はありません。このクエリでは、上の例で生成されたジョイン条件と同じものが ON 句で指定されています。

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees JOIN Departments
  ON Departments.DepartmentID = Employees.DepartmentID;
```

例 3

ある部署の責任者である従業員をすべてリストする場合は、外部キー FK_DepartmentHeadID_EmployeeID を使用し、例 1 を次のように書き換えます。このクエリは、プライマリキーテーブル Employees に相関名 FK_DepartmentHeadID_EmployeeID を指定することで外部キー FK_DepartmentHeadID_EmployeeID を使用します。

```
SELECT FK_DepartmentHeadID_EmployeeID.Surname, Departments.DepartmentName
FROM Employees AS FK_DepartmentHeadID_EmployeeID
   KEY JOIN Departments;
```

次のクエリは上の例と同義です。このクエリでは、上の例で生成されたジョイン条件が ON 句で指定されています。

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees JOIN Departments
  ON Departments.DepartmentHeadID = Employees.EmployeeID;
```

例 4

外部キーの役割名がプライマリキーテーブル名と同じ場合、相関名は不要です。たとえば、次のように、Employees テーブルに外部キー Departments を定義するとします。

```
ALTER TABLE Employees
  ADD FOREIGN KEY Departments (DepartmentID)
  REFERENCES Departments (DepartmentID);
```

ここで、KEY JOIN が 2 つのテーブル間で指定されていれば、この外部キー関係がデフォルトのジョイン条件になります。外部キー Departments が定義されていれば、次のクエリは例 3 と同義になります。

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees KEY JOIN Departments;
```

注意

この例を Interactive SQL で実行する場合は、次の文を使って SQL Anywhere サンプルデータベースへの変更をリバースする必要があります。

```
ALTER TABLE Employees DROP FOREIGN KEY Departments;
```

テーブル式のキージョイン

SQL Anywhere は、文中にあるテーブルのペアごとに外部キー関係を調べることで、テーブル式のキージョインに対してジョイン条件を生成します。

次の例では 4 つのペアのテーブルがジョインされています。

```
SELECT *
FROM (A NATURAL JOIN B) KEY JOIN (C NATURAL JOIN D);
```

テーブルのペアは A-C、A-D、B-C、B-D です。SQL Anywhere はそれぞれのペアの関係を調べてから、テーブル式全体に対して生成されるジョイン条件を作成します。処理方法はテーブル式にカンマを使用するかどうかによって異なります。したがって、次の 2 つの例では生成されたジョイン条件が異なります。A JOIN B はカンマを含まないテーブル式であり、(A,B) はテーブル式のリストです。

```
SELECT *
FROM (A JOIN B) KEY JOIN C;
```

この例は、セマンティック上、次の例と異なります。

```
SELECT *
FROM (A,B) KEY JOIN C;
```

この 2 種類のジョインの動作については以下の項を参照してください。

- 「カンマを含まないテーブル式のキージョイン」
- 「テーブル式リストのキージョイン」
- 「複数の外部キー関係がある場合のキージョイン」521 ページ

カンマを含まないテーブル式のキージョイン

ジョインされている 2 つのテーブル式のどちらにもカンマが含まれていない場合、SQL Anywhere は文中にあるテーブルのペアの外部キー関係を調べ、ジョイン条件を 1 つだけ生成します。

たとえば、次のジョインには A-C と B-C という 2 つのテーブルペアがあります。

```
(A NATURAL JOIN B) KEY JOIN C
```

C と (A NATURAL JOIN B) をジョインするために、SQL Anywhere はテーブルペア A-C と B-C の外部キー関係を調べて、ジョイン条件を 1 つだけ生成します。複数の外部キー関係が存在する

場合は、次のキージョイン決定規則に基づき、この 2 つのペアに対して 1 つのジョイン条件を生成します。

- まず、参照先となるプライマリキーテーブルのうち、その 1 つの相関名と同じ役割名を持つ単一の外部キーを指定するために A-C および B-C の両方が調べられる。この基準に合う外部キーが 1 つだけ存在する場合には、それが使用される。テーブルの相関名と同じ役割名の外部キーが 2 つ以上ある場合、そのジョインはあいまいと見なされてエラーが発行される。

- テーブルの相関名と同じ名前の外部キーがない場合は、そのテーブルの外部キー関係が検索される。見つかった外部キー関係が 1 つであれば、それが使用される。2 つ以上見つかると、そのジョインはあいまいと見なされてエラーが発行される。

- 外部キー関係がまったく存在しない場合は、エラーが発行される。

例

次の例は、営業部員とその部署をすべて検索するクエリです。

```
SELECT Employees.Surname,
       FK_DepartmentID_DepartmentID.DepartmentName
FROM ( Employees KEY JOIN Departments
       AS FK_DepartmentID_DepartmentID )
   KEY JOIN SalesOrders;
```

このクエリは次のように解釈できます。

- SQL Anywhere はテーブル式 (Employees KEY JOIN Departments as FK_DepartmentID_DepartmentID) を調べ、外部キー FK_DepartmentID_DepartmentID に基づいて、ジョイン条件 Employees.DepartmentID = FK_DepartmentID_DepartmentID.DepartmentID を生成する。

- 次に Employees/SalesOrders と Departments/SalesOrders のテーブルペアを調べる。テーブル SalesOrders と Employees 間、および、テーブル SalesOrders と Departments 間に存在できる外部キーは 1 つだけである。それ以外の場合は、ジョインはあいまいになる。この場合、テーブル SalesOrders と Employees 間には外部キー関係が 1 つだけ存在し (FK_SalesRepresentative_EmployeeID)、テーブル SalesOrders と Departments 間に外部キーは存在しない。したがって、ジョイン条件 SalesOrders.EmployeeID = Employees.SalesRepresentative が生成される。

したがって、次のクエリは上のクエリと同義です。

```
SELECT Employees.Surname, Departments.DepartmentName
FROM ( Employees JOIN Departments
   ON ( Employees.DepartmentID = Departments.DepartmentID ) )
JOIN SalesOrders
   ON ( Employees.EmployeeID = SalesOrders.SalesRepresentative );
```

テーブル式リストのキージョイン

2 つのテーブル式リストのキージョインに対してジョイン条件を生成する場合、SQL Anywhere は文中のテーブルのペアを調べて、各ペアに対してジョイン条件を生成します。最後のジョイン条件は各ペアのジョイン条件の論理積です。各ペア間には外部キー関係が存在する必要があります。

次の例では、2 つのテーブルペア A-C と B-C をジョインします。

```
SELECT *
FROM ( A,B ) KEY JOIN C;
```

SQL Anywhere では、2 つのテーブルペア A-C と B-C のそれぞれに対してジョイン条件を生成することで、C と (A,B) をジョインするためのジョイン条件が生成されます。このように複数の外部キー関係が存在する場合は、次のキージョイン規則に従って処理されます。

- 各ペアに対して、プライマリキーテーブルの相関名と同じ名前の役割名を持つ外部キーが検索される。この基準に合う外部キーが 1 つだけ存在する場合には、それが使用される。2 つ以上見つかると、そのジョインはあいまいと見なされてエラーが発行される。
- 各ペアに、テーブルの相関名と同じ名前の外部キーがない場合は、そのテーブル間の外部キー関係が検索される。見つかった外部キー関係が 1 つであれば、それが使用される。2 つ以上見つかると、そのジョインはあいまいと見なされてエラーが発行される。
- 各ペアに、外部キー関係がまったく存在しない場合は、エラーが発行される。
- 各ペアにジョイン条件を 1 つだけ決定できる場合は、AND によってジョイン条件が組み合わされる。

例

次は、ある地域で 1 つ以上の注文を受けたすべての営業部員の名前を返すクエリです。

```
SELECT DISTINCT Employees.Surname,
      FK_DepartmentID_DepartmentID.DepartmentName,
      SalesOrders.Region
FROM ( SalesOrders, Departments
      AS FK_DepartmentID_DepartmentID )
   KEY JOIN Employees;
```

Surname	DepartmentName	Region
Chin	Sales	Eastern
Chin	Sales	Western
Chin	Sales	Central
...	...	...

このクエリでは、テーブルの 2 つのペア、SalesOrders と Employees、および Departments AS FK_DepartmentID_DepartmentID と Employees を扱います。

SalesOrders と Employees のペアには、一方のテーブルと同じ役割名の外部キーはありません。ただし、2 つのテーブルに関連した外部キー (FK_SalesRepresentative_EmployeeID) が 1 つあります。これは、2 つのテーブルに関連する唯一の外部キーであり、この外部キーが使用されて、生成されたジョイン条件 (Employees.EmployeeID = SalesOrders.SalesRepresentative) になります。

Departments AS FK_DepartmentID_DepartmentID と Employees のペアでは、プライマリキーテーブルと同じ役割名を持つ外部キーは 1 つです。そのキーは FK_DepartmentID_DepartmentID で、ク

エリ内の Departments テーブルに指定した相関名と一致します。プライマリキーテーブルの相関名と同じ名前の外部キーは他にないため、このテーブルペアのジョイン条件は FK_DepartmentID_DepartmentID を使用して作成されます。生成されるジョイン条件は (Employees.DepartmentID = FK_DepartmentID_DepartmentID.DepartmentID) です。2 つのテーブルに関連する外部キーはもう 1 つありますが、この外部キーの名前はどちらのテーブルの名前とも異なるため、要因にはなりません。

最後のジョイン条件は、それぞれのテーブルペアに対して生成されたジョイン条件を 1 つにまとめます。したがって、次のクエリは同義です。

```
SELECT DISTINCT Employees.Surname,
  Departments.DepartmentName,
  SalesOrders.Region
FROM ( SalesOrders, Departments )
   JOIN Employees
   ON Employees.EmployeeID = SalesOrders.SalesRepresentative
   AND Employees.DepartmentID = Departments.DepartmentID;
```

参照

- 「複数の外部キー関係がある場合のキージョイン」521 ページ

カンマを含まないテーブル式とリストのキージョイン

テーブル式リストがカンマを含まないテーブル式を持つキージョインを介してジョインされる場合、SQL Anywhere はテーブル式リストの各テーブルに対してジョイン条件を生成します。

たとえば、次の文は、テーブル式リストと、カンマを含まないテーブル式のキージョインです。この例では、テーブル式 C NATURAL JOIN D が指定されたテーブル A と、テーブル式 C NATURAL JOIN D が指定されたテーブル B のジョイン条件が生成されます。

```
SELECT *
FROM (A,B) KEY JOIN (C NATURAL JOIN D);
```

(A,B) はテーブル式のリストで、C NATURAL JOIN D はテーブル式です。したがって、SQL Anywhere は 2 つのジョイン条件を生成する必要があります。1 つは A-C と A-D ペアのジョイン条件で、もう 1 つは B-C と B-D ペアのジョイン条件です。このように外部キー関係が複数存在する場合は、次のキージョイン規則に従って処理されます。

- テーブルペアの各セットに対して、プライマリキーテーブルの 1 つの相関名と同じ名前の役割名を持つ外部キーが検索される。この基準に合う外部キーが 1 つだけ存在する場合には、それが使用される。2 つ以上見つかると、そのジョインはあいまいになりエラーが発行される。

- テーブルペアの各セットに、テーブルの相関名と同じ名前の外部キーがない場合は、そのテーブル間の外部キー関係が検索される。存在する外部キー関係が 1 つだけであれば、それが使用される。2 つ以上見つかると、そのジョインはあいまいになりエラーが発行される。

- テーブルペアの各セットに、外部キー関係がまったく存在しない場合は、エラーが発行される。

- 各ペアにジョイン条件を 1 つだけ決定できる場合は、キーワード AND によってジョイン条件が組み合わされる。

例 1

次の 5 つのテーブルのジョインを考えてみます。

```
((A,B) JOIN (C NATURAL JOIN D) ON A.x = D.y) KEY JOIN E
```

この場合、(A,B) と E の間の条件、または C NATURAL JOIN D と E の間の条件のどちらか 1 つが生成されることで、E に対するキージョイン条件が生成されます。

(A,B) と E の間にジョイン条件が生成される場合は、A-E と B-E に 1 つずつ、合計 2 つのジョイン条件が作成される必要があります。それぞれのテーブルペア内では有効な外部キー関係が検出される必要があります。

C NATURAL JOIN D と E の間にジョイン条件が作成される場合、ジョイン条件は 1 つだけ作成されるため、C-E と D-E のペア内で検出される必要がある外部キー関係は 1 つだけです。

例 2

次は、テーブル式とテーブル式リストのキージョインの例です。この例では、営業担当兼管理者である従業員の名前と部署を示します。

```
SELECT DISTINCT Employees.Surname,
      FK_DepartmentID_DepartmentID.DepartmentName
FROM ( SalesOrders, Departments
      AS FK_DepartmentID_DepartmentID )
   KEY JOIN ( Employees JOIN Departments AS d
      ON Employees.EmployeeID = d.DepartmentHeadID );
```

次の 2 つのジョイン条件が生成されます。

- テーブルペア SalesOrders/Employees と SalesOrders/d の間には外部キー関係が 1 つだけ存在する ()。 SalesOrders.SalesRepresentative = Employees.EmployeeID.
- テーブルペア FK_DepartmentID_DepartmentID/Employees と FK_DepartmentID_DepartmentID/d の間には外部キー関係が 1 つだけ存在する ()。 FK_DepartmentID_DepartmentID.DepartmentID = Employees.DepartmentID.

この例は次の文と同義です。次の例では、相関名 Departments AS FK_DepartmentID_DepartmentID を作成する必要はありません。相関名が必要になるのは、Employees と Departments のジョインに使用する 2 つの外部キーを明示する場合だけです。

```
SELECT DISTINCT Employees.Surname,
   Departments.DepartmentName
FROM ( SalesOrders, Departments )
   JOIN ( Employees JOIN Departments AS d
      ON Employees.EmployeeID = d.DepartmentHeadID )
   ON SalesOrders.SalesRepresentative = Employees.EmployeeID
      AND Departments.DepartmentID = Employees.DepartmentID;
```

参照

- 「テーブル式リストのキージョイン」525 ページ

ビューと派生テーブルのキージョイン

キージョインにビューまたは派生テーブルが含まれている場合、SQL Anywhere ではテーブルと同じ基本手順に従います。ただし、次の点が異なります。

- 各キージョインについて、クエリとビューの FROM 句内のテーブルのペアが調べられ、すべてのペアのセットに対してジョイン条件が 1 つ生成される。この場合、ビュー内の FROM 句にカンマやジョインのキーワードが含まれているかどうかは考慮されない。
- ビューまたは派生テーブルの相関名と同じ役割名を持つ外部キーに基づいてテーブルがジョインされる。
- キージョイン内にビューまたは派生テーブルが含まれる場合、ビューまたは派生テーブル定義には UNION、INTERSECT、EXCEPT、ORDER BY、DISTINCT、GROUP BY などの集合関数や TOP、FIRST、START AT、FOR XML などの Window 関数を含めることはできない。これらが 1 つでも入っていると、エラーが返される。さらに、派生テーブルは再帰テーブル式として定義することはできない。

  派生テーブルとビューの機能は同じです。派生テーブルの場合、定義済みのビューを参照しないで、テーブルの定義を文中に記述する点だけが異なります。

例 1

次の文では、View1 がビューです。

```
SELECT *
FROM View1 KEY JOIN B;
```

View1 の定義は次のいずれかになるため、結果的に、B に対して同じジョイン条件になります (結果セットは異なりますが、ジョイン条件は同じです)。

```
SELECT *
FROM C CROSS JOIN D;

SELECT *
FROM C,D;

SELECT *
FROM C JOIN D ON (C.x = D.y);
```

いずれの場合も、View1 と B のキージョインに対してジョイン条件を生成するには、テーブルペア C-B と D-B が調べられ、ジョイン条件が 1 つだけ生成されます。ジョイン条件は複数の外部キー関係に関する規則に基づいて生成されます。ただし、ビューの参照テーブルではなく、ビューの相関名と同じ名前の外部キーを検索することが異なります。

前述のビュー定義のいずれかを使用すると、View1 KEY JOIN B の処理を次のように解釈できます。

テーブルペア C-B と D-B が調べられ、ジョイン条件が 1 つだけ生成されます。複数の外部キー関係が存在する場合には、次のキージョイン決定規則に基づいてジョイン条件が生成されます。

- まず、ビューの相関名と同じ役割名を持つ 1 つの外部キーに対して C-B と D-B が両方とも調べられる。この基準に合う外部キーが 1 つだけ存在する場合には、それが使用される。ビューの相関名と同じ役割名の外部キーが 2 つ以上ある場合、そのジョインはあいまいと見なされてエラーが発行される。

- ビューの相関名と同じ名前の外部キーがない場合は、そのテーブル間の外部キー関係が検索される。見つかった外部キー関係が 1 つであれば、それが使用される。2 つ以上見つかると、そのジョインはあいまいと見なされてエラーが発行される。

- 外部キー関係がまったく存在しない場合は、エラーが発行される。

ここで生成されたジョイン条件が B.y = D.z であると想定します。次に示す元のジョインを展開できます。たとえば、次の 2 つの文は同等です。

```
SELECT *
FROM View1 KEY JOIN B;

SELECT *
FROM View1 JOIN B ON B.y = View1.z;
```

例 2

次の例は、各部署の管理者に関する全従業員情報を含むビューです。

```
CREATE VIEW V AS
SELECT Departments.DepartmentName, Employees.*
FROM Employees JOIN Departments
  ON Employees.EmployeeID = Departments.DepartmentHeadID;
```

次のクエリはテーブル式に対するビューをジョインします。

```
SELECT *
FROM V KEY JOIN ( SalesOrders,
   Departments FK_DepartmentID_DepartmentID );
```

次のクエリは前述のクエリと同等です。

```
SELECT *
FROM V JOIN ( SalesOrders,
   Departments FK_DepartmentID_DepartmentID )
ON ( V.EmployeeID = SalesOrders.SalesRepresentative
AND V.DepartmentID =
    FK_DepartmentID_DepartmentID.DepartmentID );
```

参照

- 「再帰共通テーブル式」538 ページ
- 「複数の外部キー関係がある場合のキージョイン」521 ページ
- 「テーブル式のキージョイン」524 ページ

キージョイン操作規則

以下の規則は、これまでに説明した情報をまとめたものです。

規則 1：2 つのテーブルのキージョイン

この規則は、A KEY JOIN B に適用されます。ここで、A と B はベーステーブルまたはテンポラリテーブルです。

1. B を参照する A のすべての外部キーを見つける。

テーブル B の相関名が役割名になる外部キーが存在する場合には、それを優先外部キーとする。

2. A を参照する B のすべての外部キーを見つける。

   テーブル A の相関名が役割名になる外部キーが存在する場合には、それを優先外部キーとする。

3. 優先キーが 2 つ以上存在する場合、そのジョインはあいまいである。構文エラー SQLE_AMBIGUOUS_JOIN (-147) が発行される。

4. 優先キーが 1 つだけ存在する場合には、この KEY JOIN 式に生成されたジョイン条件を定義するために、この外部キーが選択される。

5. 優先キーがまったくない場合は、A と B 間にある他の外部キーが使用される。

   - A と B の間に外部キーが 2 つ以上ある場合、そのジョインはあいまいである。構文エラー SQLE_AMBIGUOUS_JOIN (-147) が発行される。
   - 外部キーが 1 つだけ存在する場合には、この KEY JOIN 式に生成されたジョイン条件を定義するために、この外部キーが選択される。
   - 外部キーがまったくない場合、ジョインは無効であり、エラーが生成される。

規則 2：カンマを含まないテーブル式のキージョイン

この規則は、A KEY JOIN B に適用されます。ここで、A と B はカンマを含まないテーブル式です。

1. テーブルの各ペア (テーブル式 A から 1 つとテーブル式 B から 1 つ) について、すべての外部キーをリストし、テーブル間のすべての優先キーにマークする。優先キー決定規則は上記の規則 1 で指定されている。

2. 優先キーが 2 つ以上存在する場合、そのジョインはあいまいである。構文エラー SQLE_AMBIGUOUS_JOIN (-147) が発行される。

3. 優先キーが 1 つだけ存在する場合には、この KEY JOIN 式に生成されたジョイン条件を定義するために、この外部キーが選択される。

4. 優先キーがまったくない場合は、テーブルのペア間にある他の外部キーが使用される。

   - 外部キーが 2 つ以上存在する場合、そのジョインはあいまいである。構文エラー SQLE_AMBIGUOUS_JOIN (-147) が発行される。
   - 外部キーが 1 つだけ存在する場合には、この KEY JOIN 式に生成されたジョイン条件を定義するために、この外部キーが選択される。
   - 外部キーがまったくない場合、ジョインは無効であり、エラーが生成される。

規則 3：テーブル式リストのキージョイン

この規則は、(A1, A2, ...) KEY JOIN (B1, B2, ...) に適用されます。ここで、A1、B1 はカンマを含まないテーブル式です。

1. テーブル式 Ai と Bj の各ペアに対して、規則 1 または規則 2 を適用し、テーブル式 (Ai KEY JOIN Bj) 用にユニークに生成されたジョイン条件を検出する。テーブル式のペアの KEY JOIN のいずれかが規則 1 または規則 2 によってあいまいであると判断されると、構文エラーになる。

2. この KEY JOIN 式の生成されたジョイン条件は、手順 1 のジョイン条件の論理積である。

規則 4：カンマを含まないテーブル式とリストのキージョイン

この規則は、(A1, A2, ...) KEY JOIN (B1, B2, ...) に適用されます。ここで、A1、B1 はカンマを含む可能性のあるテーブル式です。

1. テーブル式 Ai と Bj の各ペアに対して、規則 1、規則 2、または規則 3 を適用し、テーブル式 (Ai KEY JOIN Bj) 用にユニークに生成されたジョイン条件を検出する。テーブル式のペアの KEY JOIN のいずれかが規則 1、規則 2、または規則 3 によってあいまいであると判断されると、構文エラーになる。

2. この KEY JOIN 式の生成されたジョイン条件は、手順 1 のジョイン条件の論理積である。

参照

- 「'%1' を '%2' にジョインするための方法が複数あります。」『エラーメッセージ』

共通テーブル式

共通テーブル式は、WITH 句を使用して定義します。WITH 句は、SELECT 文内の SELECT キーワードに先行します。句の内容は、1 つの SELECT 文のスコープ内のみで認識される 1 つ以上のテンポラリビューを定義します。これらのテンポラリビューは、文内の他の場所から参照できます。この句の構文は、CREATE VIEW 文の構文とよく似ています。

クエリに複数の集合関数が含まれる場合や、ストアドプロシージャ内でプログラム変数を参照するビューがクエリによって定義される場合に、共通テーブル式を使用すると便利であり、共通テーブル式を使用する必要がある場合もあります。さらに、共通テーブル式は値のセットを一時的に格納する際に便利です。

例

どの部署の従業員数が最も多いかを特定する問題を例にとります。SQL Anywhere サンプルデータベースの Employees テーブルは、架空の会社で働くすべての従業員をリストし、それぞれがどの部署で働いているかを示します。次のクエリは、部署 ID コードと各部署の従業員の総数をリストします。

```
SELECT DepartmentID, COUNT( * ) AS n
FROM Employees
GROUP BY DepartmentID;
```

このクエリは、次に示すように従業員の最も多い部署を抽出するために使用できます。

```
SELECT DepartmentID, n
FROM ( SELECT DepartmentID, COUNT( * ) AS n
      FROM Employees GROUP BY DepartmentID ) AS a
WHERE a.n =
```

```
( SELECT MAX( n )
  FROM ( SELECT DepartmentID, COUNT( * ) AS n
        FROM Employees GROUP BY DepartmentID ) AS b );
```

この文は正確な結果をもたらしますが、いくつか欠点もあります。最初の欠点は、サブクエリの繰り返しによってこの文の効率が悪くなることです。2 つ目の欠点は、この文ではサブクエリ間の関係が明確でないことです。

このような問題を回避する 1 つの方法として、ビューを作成し、そのビューを使用してクエリを再作成します。この方法によって、前述の問題を回避できます。

```
CREATE VIEW CountEmployees( DepartmentID, n ) AS
  SELECT DepartmentID, COUNT( * ) AS n
  FROM Employees GROUP BY DepartmentID;

SELECT DepartmentID, n
  FROM CountEmployees
  WHERE n = ( SELECT MAX( n )
            FROM CountEmployees );
```

この方法の欠点は、ビューの作成時にデータベースサーバがシステムテーブルを更新しなければならないため、オーバヘッドが発生することです。ビューが頻繁に使用される場合は、この方法は合理的です。ただし、ビューが特定の SELECT 文内で 1 回のみ使用される場合は、代わりに次のように共通テーブル式を使用することをおすすめします。

```
WITH CountEmployees( DepartmentID, n ) AS
  ( SELECT DepartmentID, COUNT( * ) AS n
    FROM Employees GROUP BY DepartmentID )
SELECT DepartmentID, n
FROM CountEmployees
WHERE n = ( SELECT MAX( n )
          FROM CountEmployees );
```

また、従業員の最も少ない部署を検索するようにクエリを変更すると、クエリが複数のローを返す場合があることがわかります。

```
WITH CountEmployees( DepartmentID, n ) AS
  ( SELECT DepartmentID, COUNT( * ) AS n
    FROM Employees GROUP BY DepartmentID )
SELECT DepartmentID, n
FROM CountEmployees
WHERE n = ( SELECT MIN( n )
          FROM CountEmployees );
```

SQL Anywhere サンプルデータベースでは、従業員数が最も少ない 9 人編成の部署は 2 つあります。

参照

- 「複数の相関名」533 ページ
- 「複数のテーブル式」534 ページ
- 「共通テーブル式を使用できる条件」534 ページ

複数の相関名

テーブルを使用するときと同様に、共通テーブル式の複数のインスタンスに異なる相関名を付けることができます。これによって、共通テーブル式をそれ自体にジョインできます。たとえば、

次のクエリは、従業員数が同じ部署を組み合わせます。ただし、SQL Anywhere サンプルデータベースには、従業員数が同じ部は 2 つしかありません。

```
WITH CountEmployees( DepartmentID, n ) AS
   ( SELECT DepartmentID, COUNT( * ) AS n
     FROM Employees GROUP BY DepartmentID )
SELECT a.DepartmentID, a.n, b.DepartmentID, b.n
FROM CountEmployees AS a JOIN CountEmployees AS b
ON a.n = b.n AND a.DepartmentID < b.DepartmentID;
```

参照

- 「共通テーブル式」532 ページ
- 「複数のテーブル式」534 ページ
- 「共通テーブル式を使用できる条件」534 ページ

複数のテーブル式

1 つの WITH 句で、2 つ以上の共通テーブル式を定義できます。これらの定義はカンマで区切る必要があります。次の例は、支払い給与総額の最も少ない部署と、従業員数が最も多い部署をリストします。

```
WITH
  CountEmployees( DepartmentID, n ) AS
    ( SELECT DepartmentID, COUNT( * ) AS n
      FROM Employees GROUP BY DepartmentID ),
  DepartmentPayroll( DepartmentID, amount ) AS
    ( SELECT DepartmentID, SUM( Salary ) AS amount
      FROM Employees GROUP BY DepartmentID )
SELECT count.DepartmentID, count.n, pay.amount
FROM CountEmployees AS count JOIN DepartmentPayroll AS pay
ON count.DepartmentID = pay.DepartmentID
WHERE count.n = ( SELECT MAX( n ) FROM CountEmployees )
   OR pay.amount = ( SELECT MIN( amount ) FROM DepartmentPayroll );
```

参照

- 「共通テーブル式」532 ページ
- 「複数の相関名」533 ページ
- 「共通テーブル式を使用できる条件」534 ページ

共通テーブル式を使用できる条件

共通テーブル式はクエリ本体または任意のサブクエリ内から参照可能ですが、共通テーブル式を定義できるのは 3 か所のみです。

- **最上位レベルの SELECT 文**　共通テーブル式は、最上位レベルの SELECT 文内で使用できますが、サブクエリ内では使用できません。

  ```
  WITH DepartmentPayroll( DepartmentID, amount ) AS
    ( SELECT DepartmentID, SUM( Salary ) AS amount
      FROM Employees GROUP BY DepartmentID )
  SELECT DepartmentID, amount
  FROM DepartmentPayroll
  ```

```
WHERE amount = ( SELECT MAX( amount )
          FROM DepartmentPayroll );
```

- **ビュー定義内の最上位レベルの SELECT 文**　共通テーブル式は、ビューを定義する最上位レベルの SELECT 文内で使用できますが、サブクエリ内では使用できません。

```
CREATE VIEW LargestDept ( DepartmentID, Size, pay ) AS
  WITH
    CountEmployees( DepartmentID, n ) AS
      ( SELECT DepartmentID, COUNT( * ) AS n
        FROM Employees GROUP BY DepartmentID ),
    DepartmentPayroll( DepartmentID, amount ) AS
      ( SELECT DepartmentID, SUM( Salary ) AS amount
        FROM Employees GROUP BY DepartmentID )
  SELECT count.DepartmentID, count.n, pay.amount
  FROM CountEmployees count JOIN DepartmentPayroll pay
  ON count.DepartmentID = pay.DepartmentID
  WHERE count.n = ( SELECT MAX( n ) FROM CountEmployees )
    OR pay.amount = ( SELECT MAX( amount ) FROM DepartmentPayroll );
```

- **INSERT 文内の最上位レベルの SELECT 文**　共通テーブル式は、INSERT 文内の最上位レベルの SELECT 文内で使用できますが、INSERT 文内のサブクエリ内では使用できません。

```
CREATE TABLE LargestPayrolls ( DepartmentID INTEGER, Payroll NUMERIC, CurrentDate
DATE );
INSERT INTO LargestPayrolls( DepartmentID, Payroll, CurrentDate )
  WITH DepartmentPayroll( DepartmentID, amount ) AS
    ( SELECT DepartmentID, SUM( Salary ) AS amount
      FROM Employees
      GROUP BY DepartmentID )
  SELECT DepartmentID, amount, CURRENT TIMESTAMP
  FROM DepartmentPayroll
  WHERE amount = ( SELECT MAX( amount )
            FROM DepartmentPayroll );
```

参照

- 「共通テーブル式」532 ページ
- 「複数の相関名」533 ページ
- 「複数のテーブル式」534 ページ

共通テーブル式の一般的な使用例

一般的に、共通テーブル式は、単一のクエリ内でテーブル式を複数回使用しなければならない場合に役立ちます。よくある次のような状況では、共通テーブル式の使用が適切です。

- 複数の集合関数を含むクエリ
- プログラム変数への参照を含める必要がある、プロシージャ内のビュー
- 値のセットを格納するためにテンポラリビューを使用するクエリ

このリストはすべてを網羅したものではありません。共通テーブル式が役に立つ状況は、ほかにも数多くあります。

複数の集合関数

共通テーブル式は、単一のクエリ内で複数レベルの集約を使用しなければならない場合に役立ちます。前項で使用した例がこれに当てはまります。そのタスクは、従業員数が最も多い部署の部署 ID を取り出すことでした。そのために、COUNT 集合関数を使用して各部署の従業員数を計算し、MAX 関数を使用して最も従業員数の多い部署を選択しています。

支払い給与総額が最も多い部を判別するクエリを記述する場合も、似たような状況となります。SUM 集合関数を使用して各部署の支払い給与総額を計算し、MAX 関数を使用してどの部署が一番多いかを判別します。クエリに両方の関数があることが、共通テーブル式が役立つかどうかを判断する手がかりとなります。

```
WITH DepartmentPayroll( DepartmentID, amount ) AS
   ( SELECT DepartmentID, SUM( Salary ) AS amount
     FROM Employees GROUP BY DepartmentID )
SELECT DepartmentID, amount
FROM DepartmentPayroll
WHERE amount = ( SELECT MAX( amount )
           FROM DepartmentPayroll )
```

参照

- 「Window 集合関数」565 ページ

プログラム変数を参照するビュー

プログラム変数への参照を含むビューを作成すると便利な場合があります。たとえば、特定の顧客を識別する変数をプロシージャ内に定義するとします。その顧客の購入履歴を問い合わせる場合、同様の情報に何度もアクセスしたり、複数の集合関数を使用する必要が出じるのであれば、その特定の顧客に関する情報を含むビューを作成すると便利です。

ビューのスコープをプロシージャのスコープに制限する方法はないため、プログラム変数を参照するビューを作成することはできません。ビューは、一度作成すると、このような目的以外にも使用できます。しかし、プロシージャのクエリ内で共通テーブル式を使用することもできます。共通テーブル式のスコープはその文に制限されるため、変数を参照してもあいまい性はなく、変数の参照は許可されます。

次の文は、SQL Anywhere サンプルデータベース内のさまざまな販売担当者の総売上高を選択します。

```
SELECT GivenName || ' ' || Surname AS sales_rep_name,
      SalesRepresentative AS sales_rep_id,
      SUM( p.UnitPrice * i.Quantity ) AS total_sales
FROM Employees LEFT OUTER JOIN SalesOrders AS o
           INNER JOIN SalesOrderItems AS I
           INNER JOIN Products AS p
WHERE OrderDate BETWEEN '2000-01-01' AND '2001-12-31'
GROUP BY SalesRepresentative, GivenName, Surname;
```

前述のクエリは、次のプロシージャ内で使用されている共通テーブル式の基礎となります。調査する販売担当者の ID 番号と年度が入力パラメータです。次のプロシージャからわかるように、WITH 句内では、プロシージャパラメータと宣言済みのすべてのローカル変数を参照できます。

```
CREATE PROCEDURE sales_rep_total (
  IN rep  INTEGER,
```

```
  IN yyyy INTEGER )
BEGIN
  DECLARE StartDate DATE;
  DECLARE EndDate   DATE;
  SET StartDate = YMD( yyyy,  1,  1 );
  SET EndDate = YMD( yyyy, 12, 31 );
  WITH total_sales_by_rep ( sales_rep_name,
                            sales_rep_id,
                            month,
                            order_year,
                            total_sales ) AS
  ( SELECT GivenName || ' ' || Surname AS sales_rep_name,
          SalesRepresentative AS sales_rep_id,
          month( OrderDate),
          year( OrderDate ),
          SUM( p.UnitPrice * i.Quantity ) AS total_sales
    FROM Employees LEFT OUTER JOIN SalesOrders o
                   INNER JOIN SalesOrderItems I
                   INNER JOIN Products p
    WHERE OrderDate BETWEEN StartDate AND EndDate
          AND SalesRepresentative = rep
    GROUP BY year( OrderDate ), month( OrderDate ),
             GivenName, Surname, SalesRepresentative )
  SELECT sales_rep_name,
         monthname( YMD(yyyy, month, 1) ) AS month_name,
         order_year,
         total_sales
  FROM total_sales_by_rep
  WHERE total_sales =
    ( SELECT MAX( total_sales) FROM total_sales_by_rep )
  ORDER BY order_year ASC, month ASC;
END;
```

次の文を実行すると、前述のプロシージャが呼び出されます。

```
CALL sales_rep_total( 129, 2000 );
```

値を格納するビュー

SELECT 文内またはプロシージャ内に特定の値のセットを格納すると便利な場合があります。たとえば、ある会社で販売担当者の成績を分析する際の単位として、四半期ではなく、1 年を 3 期に分けた期間を採用しているとします。四半期という日付の単位は組み込まれていますが、1 年を 3 期に分けた日付の単位はないため、プロシージャ内で日付を格納する必要があります。

```
WITH thirds ( q_name, q_start, q_end ) AS
( SELECT 'T1', '2000-01-01', '2000-04-30' UNION
  SELECT 'T2', '2000-05-01', '2000-08-31' UNION
  SELECT 'T3', '2000-09-01', '2000-12-31' )
SELECT q_name,
       SalesRepresentative,
       count(*) AS num_orders,
       SUM( p.UnitPrice * i.Quantity ) AS total_sales
FROM thirds LEFT OUTER JOIN SalesOrders AS o
    ON OrderDate BETWEEN q_start and q_end
                 KEY JOIN SalesOrderItems AS I
                 KEY JOIN Products AS p
 GROUP BY q_name, SalesRepresentative
 ORDER BY q_name, SalesRepresentative;
```

この方法は、値を定期的に保守する必要があるため、注意して使用する必要があります。たとえば、前述の文は、他の年に対して使用する場合は修正する必要があります。

この方法をプロシージャ内で適用することもできます。次の例は、対象の年を引数として受け取るプロシージャを宣言しています。

```
CREATE PROCEDURE sales_by_third ( IN y INTEGER )
BEGIN
  WITH thirds ( q_name, q_start, q_end ) AS
  ( SELECT 'T1', YMD( y, 01, 01), YMD( y, 04, 30 ) UNION
    SELECT 'T2', YMD( y, 05, 01), YMD( y, 08, 31 ) UNION
    SELECT 'T3', YMD( y, 09, 01), YMD( y, 12, 31 ) )
  SELECT q_name,
         SalesRepresentative,
         count(*) AS num_orders,
         SUM( p.UnitPrice * i.Quantity ) AS total_sales
  FROM thirds LEFT OUTER JOIN SalesOrders AS o
    ON OrderDate BETWEEN q_start and q_end
          KEY JOIN SalesOrderItems AS I
          KEY JOIN Products AS p
  GROUP BY q_name, SalesRepresentative
  ORDER BY q_name, SalesRepresentative;
END;
```

次の文を実行すると、前述のプロシージャが呼び出されます。

```
CALL sales_by_third (2000);
```

再帰共通テーブル式

繰り返し実行される場合、共通テーブル式は再帰的であり、各実行によって、完全な結果セットが取得されるまで追加のローが返されます。WITH 句で WITH の直後に RECURSIVE キーワードを挿入することによって、共通テーブル式を再帰的にすることができます。1 つの WITH 句には、再帰的と非再帰的両方の複数の再帰式を含めることができます。

再帰を使用すると、ツリーまたはツリーに似たデータ構造を表すテーブルをトラバースすることが簡単になります。再帰式を使用しないで単一の文内でそのような構造をトラバースする唯一の方法は、考えられるレベルごとに 1 回ずつテーブルをそれ自体にジョインすることです。

再帰共通テーブル式に関する制限

- 他の再帰共通テーブル式への参照を、再帰共通テーブル式の定義内に含めることはできません。再帰共通テーブル式を相互に再帰的にすることはできないためです。ただし、非再帰共通テーブル式には、再帰テーブル式への参照を含めることができます。また、再帰共通テーブル式には、非再帰共通テーブル式への参照を含めることができます。

- 初期サブクエリと再帰サブクエリ間で唯一サポートされている set 演算子は、UNION ALL です。

- 再帰サブクエリ定義内では、再帰共通テーブル式への自己参照は再帰サブクエリの FROM 句内のみに含めることができ、外部ジョインの NULL 入力側に含めることはできません。

- 再帰サブクエリに DISTINCT 句、GROUP BY 句、または ORDER BY 句を含めることはできません。

●再帰サブクエリでは集合関数は使用できません。

●再帰サブクエリの暴走を防ぐために、再帰レベル数が max_recursive_iterations オプションの現在の設定を超えるとエラーが生成されます。このオプションのデフォルト値は 100 です。

例

社内の報告関係を表すテーブルがある場合、特定の 1 人に報告を行うすべての従業員を返すクエリを記述できます。

クエリの記述方法によって、再帰レベル数を制限できます。たとえば、レベル数を制限して、トップレベルの管理者のみを返すことができますが、指揮系統が予期したよりも長い場合、除外される従業員も出てきます。レベル数を制限しない場合は、従業員が除外されることはありません。ただし、ある従業員が直接または間接的に自分自身に対してレポートするなど、実行に循環が必要な場合は、無限の再帰が発生する可能性があります。この状況は、会社の従業員が取締役会の一員である場合などに、社内の管理階層内で発生することがあります。

次のクエリは、管理レベルで従業員をリストする方法を示しています。level 0 は、管理者を持たない従業員を表します。level 1 は、level 0 の管理職の 1 人に直接報告を行う従業員を表し、level 2 は、level 1 の管理職に直接報告を行う従業員を表します。以下、同様に続きます。

```
WITH RECURSIVE
  manager ( EmployeeID, ManagerID,
            GivenName, Surname, mgmt_level ) AS
( ( SELECT EmployeeID, ManagerID,       -- initial subquery
           GivenName, Surname, 0
    FROM Employees AS e
    WHERE ManagerID = EmployeeID )
  UNION ALL
  ( SELECT e.EmployeeID, e.ManagerID,   -- recursive subquery
           e.GivenName, e.Surname, m.mgmt_level + 1
    FROM Employees AS e JOIN manager AS m
     ON   e.ManagerID =  m.EmployeeID
      AND e.ManagerID <> e.EmployeeID
      AND m.mgmt_level < 20 ) )
SELECT * FROM manager
ORDER BY mgmt_level, Surname, GivenName;
```

管理レベルを 20 未満に制限する再帰クエリ内の条件 (m.mgmt leve < 20) は停止条件と呼ばれ、重要な予防措置です。これにより、テーブルデータに循環が含まれている場合に、無限再帰を回避できます。

max_recursive_iterations オプションは、暴走する再帰クエリをキャッチするために使用することもできます。このオプションのデフォルト値は 100 です。この反復数を超えた再帰クエリは終了しますが、エラーが発生します。このオプションがあれば停止条件は重要でないように見えるかもしれませんが、通常はそうではありません。各反復の間に選択されるローの数は急激に増える可能性があり、最大に達する前にパフォーマンスに深刻な影響を与えることがあります。再帰クエリ内に停止条件を設けるのは、各状況に適した制限を設定する手段です。

再帰共通テーブル式には、「初期サブクエリ」 (シード) と、各反復の間に結果セットにローを追加する「再帰サブクエリ」が含まれます。この 2 つの部分は、UNION ALL 演算子を使用してのみ接続できます。初期サブクエリは、通常の非再帰クエリで、最初に処理されます。再帰部分には、直前の繰り返しで追加されたローへの参照が含まれます。再帰は、繰り返しによって新しいローが生成されなくなったときに自動的に停止します。直前の繰り返しより前に選択されたローを参照する方法はありません。

再帰サブクエリの SELECT リストは、初期サブクエリの SELECT リストと数およびデータ型が一致する必要があります。データ型の自動変換が行えない場合は、一方のサブクエリの結果を明示的にキャストして、もう一方のサブクエリのデータ型に一致させます。

参照

- 「複数の再帰共通テーブル式」541 ページ
- 「max_recursive_iterations オプション」『SQL Anywhere サーバ データベース管理』

再帰共通テーブル式でのデータ型宣言

テンポラリビュー内のカラムのデータ型は、初期サブクエリのデータ型によって定義されます。再帰サブクエリのカラムのデータ型は、一致する必要があります。データベースサーバは、再帰サブクエリによって返された値がその初期クエリの値に一致するよう、自動的に変換しようとします。これが不可能な場合、または変換で情報が失われた場合、エラーが生成されます。

通常、初期サブクエリがリテラル値または NULL を返す場合、明示的なキャストが必要な場合がほとんどです。初期サブクエリが再帰サブクエリとは異なるカラムから値を選択する場合も、明示的なキャストが必要な場合があります。

キャストは、初期サブクエリのカラムが再帰サブクエリのカラムと同じドメインを持っていない場合に必要な場合があります。初期サブクエリでは、NULL 値に常にキャストを適用する必要があります。

たとえば、構成部品の問題は、初期サブクエリが bookcase テーブルからローを返すことによって、選択されたカラムのデータ型を継承するため、正しく動作します。

ただし、このクエリが次のように書き直された場合は、明示的なキャストが必要です。

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT NULL, 'bookcase', 1         -- ERROR! Wrong domains!
   UNION ALL
  SELECT b.component, b.subcomponent,
         p.quantity * b.quantity
  FROM parts p JOIN bookcase b
    ON p.subcomponent = b.component )
SELECT * FROM parts
ORDER BY component, subcomponent;
```

キャストを行わない場合、次の理由でエラーが発生します。

- コンポーネント名の正しいデータ型は VARCHAR だが、最初のカラムは NULL である。
- 数字 1 は SMALLINT と見なされるが、quantity カラムのデータ型は INT である。

2 番目のカラムには、キャストは不要です。初期クエリのこのカラムがすでに文字列であるためです。

初期サブクエリでデータ型をキャストすれば、クエリを意図したとおりに動作させることができます。

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT CAST( NULL AS VARCHAR ),  -- CASTs must be used
```

```
        'bookcase',              -- to declare the
        CAST( 1   AS INT )       -- correct datatypes
      UNION ALL
  SELECT b.component, b.subcomponent,
        p.quantity * b.quantity
  FROM parts p JOIN bookcase b
    ON p.subcomponent = b.component )
SELECT * FROM parts
ORDER BY component, subcomponent;
```

参照

- 「構成部品の問題」542 ページ

複数の再帰共通テーブル式

再帰クエリには、複数の再帰クエリを含めることができますが、それらの再帰クエリが共通の要素を持たず互いに素であることが条件となります。また、再帰クエリには再帰共通テーブル式と非再帰共通テーブル式を混在させることができます。共通テーブル式が 1 つでも再帰的である場合は、RECURSIVE キーワードを含める必要があります。

たとえば、前述のクエリと同じ結果を返す次のクエリは、別の非再帰共通テーブル式を使用して最短ルートの距離を選択しています。2 つめの共通テーブル式の定義は、1 つめの定義からカンマで区切られています。

```
WITH RECURSIVE
  trip ( route, destination, previous, distance, segments ) AS
    ( SELECT CAST( origin || ', ' || destination AS VARCHAR(256) ),
            destination, origin, distance, 1
      FROM travel
      WHERE origin = 'Kitchener'
      UNION ALL
      SELECT route || ', ' || v.destination,
            v.destination,
            v.origin,
            t.distance + v.distance,
            segments + 1
      FROM trip t JOIN travel v ON t.destination = v.origin
      WHERE v.destination <> 'Kitchener'
        AND v.destination <> t.previous
        AND v.origin      <> 'Pembroke'
        AND segments
              < ( SELECT count(*)/2 FROM travel ) ),
  shortest ( distance ) AS                -- Additional,
    ( SELECT MIN(distance)                  -- non-recursive
      FROM trip                         -- common table
      WHERE destination = 'Pembroke' )     -- expression
SELECT route, distance, segments FROM trip
WHERE destination = 'Pembroke' AND
      distance < 1.5 * ( SELECT distance FROM shortest )
ORDER BY distance, segments, route;
```

非再帰共通テーブル式と同様、再帰式をストアドプロシージャ内で使用した場合、ローカル変数やプロシージャパラメータへの参照を含められます。たとえば、次に定義する best_routes プロシージャは、指定した 2 つの都市の間の最短ルートを特定します。

```
CREATE PROCEDURE best_routes (
  IN initial VARCHAR(10),
```

```
  IN final   VARCHAR(10)
)
BEGIN
  WITH RECURSIVE
     trip ( route, destination, previous, distance, segments ) AS
  ( SELECT CAST( origin || ', ' || destination AS VARCHAR(256) ),
         destination, origin, distance, 1
    FROM travel
    WHERE origin = initial
      UNION ALL
    SELECT route || ', ' || v.destination,
         v.destination,           -- current endpoint
         v.origin,                -- previous endpoint
         t.distance + v.distance,  -- total distance
         segments + 1             -- total number of segments
    FROM trip t JOIN travel v ON t.destination = v.origin
    WHERE v.destination <> initial     -- Don't return to start
      AND v.destination <> t.previous  -- Prevent backtracking
      AND v.origin      <> final       -- Stop at the end
      AND segments                     -- TERMINATE RECURSION!
          < ( SELECT count(*)/2 FROM travel ) )
  SELECT route, distance, segments FROM trip
  WHERE destination = final AND
       distance < 1.4 * ( SELECT MIN( distance )
                          FROM trip
                          WHERE destination = final )
  ORDER BY distance, segments, route;
END;
```

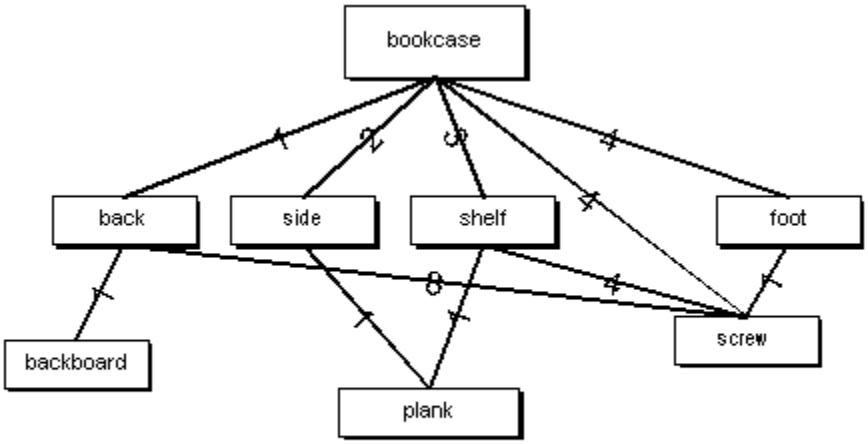
次の文を実行すると、前述のプロシージャが呼び出されます。

```
CALL best_routes ( 'Pembroke', 'Kitchener' );
```

構成部品の問題

構成部品の問題は、再帰を適用できる典型例です。この問題では、特定のオブジェクトを組み立てるために必要なコンポーネントが図で表されます。目的は、この図をデータベーステーブルを使用して表し、次に必要な要素部品の総数を計算することです。

たとえば、次の図は単純な本箱のコンポーネントを表しています。この本箱は、3 段の棚、背、そして 4 本のねじで固定される 4 本の足から構成されています。各棚は、4 つのねじで固定される板です。背には、8 つのねじで固定される別の板が使われています。



次のテーブル内の情報は、本箱の図のエッジを表しています。最初のカラムはコンポーネントを、2 番目のカラムはそのコンポーネントのサブコンポーネントの 1 つを、そして 3 番目のカラムは必要なサブコンポーネント数を示しています。

component	subcomponent	quantity
bookcase	back	1
bookcase	side	2
bookcase	shelf	3
bookcase	foot	4
bookcase	screw	4
back	backboard	1
back	screw	8
side	plank	1
shelf	plank	1
shelf	screw	4

次の文を実行して、bookcase テーブルを作成し、コンポーネントデータとサブコンポーネントデータを挿入します。

```
CREATE TABLE bookcase (
    component     VARCHAR(9),
```

```
    subcomponent   VARCHAR(9),
    quantity       INTEGER,
  PRIMARY KEY ( component, subcomponent )
);
INSERT INTO bookcase
  SELECT 'bookcase', 'back',      1 UNION
  SELECT 'bookcase', 'side',      2 UNION
  SELECT 'bookcase', 'shelf',     3 UNION
  SELECT 'bookcase', 'foot',      4 UNION
  SELECT 'bookcase', 'screw',     4 UNION
  SELECT 'back',     'backboard', 1 UNION
  SELECT 'back',     'screw',     8 UNION
  SELECT 'side',     'plank',     1 UNION
  SELECT 'shelf',    'plank',     1 UNION
  SELECT 'shelf',    'screw',     4;
```

次の文を実行して、本箱の組み立てに必要なコンポーネント、サブコンポーネント、およびその数量のリストを生成します。

```
SELECT * FROM bookcase
ORDER BY component, subcomponent;
```

次の文を実行して、本箱の組み立てに必要なサブコンポーネントとその数量のリストを生成します。

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT component, subcomponent, quantity
  FROM bookcase WHERE component = 'bookcase'
    UNION ALL
  SELECT b.component, b.subcomponent, p.quantity * b.quantity
  FROM parts p JOIN bookcase b ON p.subcomponent = b.component )
SELECT subcomponent, SUM( quantity ) AS quantity
FROM parts
WHERE subcomponent NOT IN ( SELECT component FROM bookcase )
GROUP BY subcomponent
ORDER BY subcomponent;
```

このクエリの結果を次に示します。

subcomponent	quantity
backboard	1
foot	4
plank	5
screw	24

また、このクエリを別の再帰レベルを実行するように書き直すこともできます。こうすると、メインの SELECT 文内でサブクエリを記述する必要がなくなります。次のクエリの結果は、前述のクエリの結果とまったく同じです。

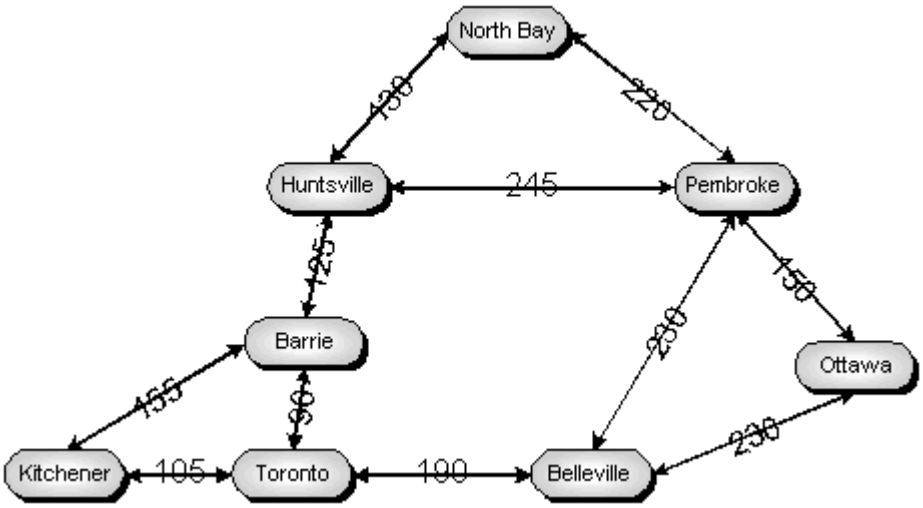
```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT component, subcomponent, quantity
  FROM bookcase WHERE component = 'bookcase'
    UNION ALL
  SELECT p.subcomponent, b.subcomponent,
```

```
    IF b.quantity IS NULL
    THEN p.quantity
    ELSE p.quantity * b.quantity
    ENDIF
  FROM parts p LEFT OUTER JOIN bookcase b
  ON p.subcomponent = b.component
    WHERE p.subcomponent IS NOT NULL
 )
SELECT component, SUM( quantity ) AS quantity
FROM parts
WHERE subcomponent IS NULL
GROUP BY component
ORDER BY component;
```

最短距離の問題

再帰共通テーブル式を使用して、命令グラフ上の望ましいパスを見つけられます。データベーステーブルの各ローは、命令エッジを表します。各ローは、出発地から目的地まで移動するときの出発地、目的地、およびコストを示します。問題の種類によって、コストは距離であったり、所要時間であったり、または別の基準であることも考えられます。再帰を使用すると、このグラフを介して可能なルートを探索できます。そして、いくつかの可能なルートから目的のルートを選ぶことができます。

たとえば、キッチナー市とペンブローク市の間を車で移動するときの望ましいルートを見つける問題を考えてみます。いくつも可能なルートがあり、それぞれ異なるいくつかの都市を中継点として通ります。目的は、最短ルートをいくつか見つけ、それらを別の適当な選択肢と比較することです。



まず、このグラフのエッジを表すテーブルを定義し、各エッジに対して 1 ローを挿入します。このグラフのすべてのエッジは双方向であるため、逆方向を表すエッジも挿入します。このために、最初のローセットを選択し、出発地と目的地を入れ替えます。たとえば、一方のローがキッチナー市からトロント市への移動を表し、もう一方のローがトロント市からキッチナー市へ戻る移動を表します。

```
CREATE TABLE travel (
   origin      VARCHAR(10),
   destination VARCHAR(10),
   distance    INT,
  PRIMARY KEY ( origin, destination )
);
INSERT INTO travel
  SELECT 'Kitchener',  'Toronto',    105 UNION
  SELECT 'Kitchener',  'Barrie',     155 UNION
  SELECT 'North Bay',  'Pembroke',   220 UNION
  SELECT 'Pembroke',   'Ottawa',     150 UNION
  SELECT 'Barrie',     'Toronto',     90 UNION
  SELECT 'Toronto',    'Belleville', 190 UNION
  SELECT 'Belleville', 'Ottawa',     230 UNION
  SELECT 'Belleville', 'Pembroke',   230 UNION
  SELECT 'Barrie',     'Huntsville', 125 UNION
  SELECT 'Huntsville', 'North Bay',  130 UNION
  SELECT 'Huntsville', 'Pembroke',   245;
INSERT INTO travel   -- Insert the return trips
  SELECT destination, origin, distance
  FROM travel;
```

次に、再帰共通テーブル式を記述します。移動はキッチナー市から始まるため、初期サブクエリは、キッチナー市を起点とする可能なすべてのパスを、その距離とともに選択することから始まります。

再帰サブクエリは、パスを拡張します。各パスに対し、再帰サブクエリは、直前のセグメントの目的地から続くセグメントを追加し、新しいセグメントの距離を加えて、各ルートの現在のトータルコストを管理します。効率性を考慮して、次の条件のいずれかに該当した場合、ルートは終了します。

●パスが出発地に戻る場合。

●パスが直前の地点に戻る場合。

●パスが最後の目的地に達した場合。

この例では、キッチナー市に戻るパスはなく、全てのパスはペンブローク市に達した場合、終了します。

再帰クエリを使用して環状のグラフを探索する場合は、再帰クエリが適切に終了することを確認することが重要です。この例の場合、前述の条件では不十分です。ルートに 2 つの中継地の間を行ったり来たりする移動が無制限に含まれる可能性があるからです。次の再帰クエリでは、指定したルート内の最大セグメント数を 7 つまでに制限することで、ルートの終わりを保証しています。

このクエリ例のポイントは現実的なルートを選択することであるため、メインクエリでは、距離が最短ルート比 50% 増未満のルートのみを選択しています。

```
WITH RECURSIVE
   trip ( route, destination, previous, distance, segments ) AS
( SELECT CAST( origin || ', ' || destination AS VARCHAR(256) ),
        destination, origin, distance, 1
  FROM travel
  WHERE origin = 'Kitchener'
    UNION ALL
  SELECT route || ', ' || v.destination,
```

```
        v.destination,          -- current endpoint
        v.origin,               -- previous endpoint
        t.distance + v.distance,  -- total distance
        segments + 1            -- total number of segments
  FROM trip t JOIN travel v ON t.destination = v.origin
  WHERE v.destination <> 'Kitchener'  -- Don't return to start
    AND v.destination <> t.previous   -- Prevent backtracking
    AND v.origin      <> 'Pembroke'   -- Stop at the end
    AND segments                      -- TERMINATE RECURSION!
        < ( SELECT count(*)/2 FROM travel ) )
SELECT route, distance, segments FROM trip
WHERE destination = 'Pembroke' AND
     distance < 1.5 * ( SELECT MIN( distance )
                        FROM trip
                        WHERE destination = 'Pembroke' )
ORDER BY distance, segments, route;
```

この文を前述のデータセットに対して実行すると、次のような結果となります。

route	distance	segments
Kitchener, Barrie, Huntsville, Pembroke	525	3
Kitchener, Toronto, Belleville, Pembroke	525	3
Kitchener, Toronto, Barrie, Huntsville, Pembroke	565	4
Kitchener, Barrie, Huntsville, North Bay, Pembroke	630	4
Kitchener, Barrie, Toronto, Belleville, Pembroke	665	4
Kitchener, Toronto, Barrie, Huntsville, North Bay, Pembroke	670	5
Kitchener, Toronto, Belleville, Ottawa, Pembroke	675	4

OLAP のサポート

OLAP (オンライン分析処理) では、単一の SQL 文内で複雑なデータ分析を実行できます。また、データベースでクエリの量を減らすことでパフォーマンスを向上しながら、結果の値を増やすことができます。SQL 文と Window 関数の拡張を使用することで、OLAP 機能を利用できます。このような SQL の拡張や機能では、多次元データ分析、データマイニング、時系列分析、傾向分析、コスト配分、ゴールシーク、例外警告などを、通常は単一の SQL 文を使用して、簡単に実行できます。

- **SELECT 文の拡張**　SELECT 文を拡張することにより、入力ローをグループ化したり、グループを分析したり、最終結果セットに検索結果を含めることができます。これらの拡張には、GROUP BY 句 (GROUPING SETS、CUBE、ROLLUP の各サブ句) と WINDOW 句に対する拡張が含まれます。

  GROUP BY 句の拡張では、入力ローを複数の方法で分割できるため、さまざまなグループを連結する結果セットを得ることができます。データマイニング用に散在した多次元結果セット (「データキューブ」とも呼ばれる) を作成することもできます。さらに、この拡張により、サブ合計のローと総合計のローを使用して、分析をより便利にすることができます。

  WINDOW 句は、入力ローのグループで分析する機会を増やすために Window 関数とともに使用されます。

- **Window 集合関数**　ほとんどの集合関数で、入力ローの処理に合わせて入力ローを上から下に移動する、設定可能なスライド「ウィンドウ」の概念がサポートされています。ウィンドウの移動とともにウィンドウ内のデータに対する追加の計算を実行できるため、セマンティック上同等なセルフジョインクエリや相関サブクエリを使用する方法よりも効率的な方法で、詳細な分析を実行できます。

  たとえば Window 集合関数を GROUP BY 句の CUBE、ROLLUP、GROUPING SETS 拡張と組み合わせることで、単一の SQL 文内における百分位数、移動平均、累積合計を効率的に計算できます。それ以外の方法では、セルフジョイン、相関サブクエリ、テンポラリテーブル、またはこれら 3 つを組み合わせなければなりません。

  Window 集合関数を使用すると、ダウジョーンズ工業平均株価の四半期の移動平均や、部門ごとの全従業員と給与の累積合計などの情報を取得できます。また、平方偏差、標準偏差、相関、回帰などの測定を計算することもできます。

- **Window ランキング関数**　Window ランキング関数を使用すると、今年出荷された総売り上げ上位 10 製品や、最低 15 企業に販売注文した販売担当者の上位 5% といった情報を取得する、単一文の SQL クエリを作ることができます。

参照

- 「GROUP BY 句の拡張」549 ページ
- 「Window 関数」558 ページ
- 「Window ランキング関数」584 ページ
- 「Window 集合関数」565 ページ

OLAP パフォーマンスの向上

OLAP パフォーマンスを向上させるには、optimization_workload データベースオプションを OLAP に設定して、調査する候補にクラスタド GROUP BY ハッシュ演算子を使用することをオプティマイザに指示しますインデックスの定義時に FOR OLAP WORKLOAD オプションを使用して、OLAP 負荷のインデックスを調整することもできます。このオプションを使用すると、データベースサーバは特定の最適化を実行します。この最適化には、同じキー内の 2 つのローの最大ページ距離に関して、クラスタド GROUP BY ハッシュ演算子で使用する統計情報の管理が含まれます。

参照

- 「optimization_workload オプション」『SQL Anywhere サーバ データベース管理』
- 「CREATE INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

GROUP BY 句の拡張

SELECT 文で標準の GROUP BY 句を使用すると、指定したグループ化の式に従って、結果セット内のローをグループ化できます。たとえば、GROUP BY columnA, columnB を指定すると、columnA と columnB のユニークな値の組み合わせでローがグループ化されます。標準の GROUP BY 句では、グループは、指定されたすべての GROUP BY 式の組み合わせの評価を反映します。

ただし、結果セットに別のグループ化またはサブグループ化を指定したい場合も考えられます。たとえば、結果で columnA と columnB のユニークな値でグループ化されたデータを表示してから、columnC の別の値でもう一度グループ化するような状況です。このような結果を得るには、GROUP BY 句で GROUPING SETS 拡張を使用します。

GROUP BY GROUPING SETS

GROUPING SETS 句は、SELECT 文の GROUP BY 句の拡張です。GROUPING SETS 句を使用すると、複数の SELECT 文を使用しなくても結果を複数の方法でグループ化できます。つまり、応答時間を減らし、パフォーマンスを向上させることができます。

たとえば、次の 2 つのクエリ文はセマンティック上同義です。ただし、2 番目のクエリでは GROUP BY GROUPING SETS 句を使用することで、グループ化基準をより効率的に定義します。

複数の SELECT 文を使用した複数のグループ化

```
SELECT NULL, NULL, NULL, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
  UNION ALL
SELECT City, State, NULL, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
GROUP BY City, State
  UNION ALL
SELECT NULL, NULL, CompanyName, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
GROUP BY CompanyName;
```

GROUPING SETS を使用した複数のグループ化

```
SELECT City, State, CompanyName, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
GROUP BY GROUPING SETS( ( City, State ), ( CompanyName ) , ( ) );
```

どちらの方法でも、次に示す同じ結果が生成されます。

	City	State	CompanyName	Cnt
1	(NULL)	(NULL)	(NULL)	8
2	(NULL)	(NULL)	Cooper Inc.	1
3	(NULL)	(NULL)	Westend Dealers	1
4	(NULL)	(NULL)	Toto's Active Wear	1
5	(NULL)	(NULL)	North Land Trading	1
6	(NULL)	(NULL)	The Ultimate	1
7	(NULL)	(NULL)	Molly's	1
8	(NULL)	(NULL)	Overland Army Navy	1
9	(NULL)	(NULL)	Out of Town Sports	1
10	'Pembroke'	'MB'	(NULL)	4
11	'Petersburg'	'KS'	(NULL)	1
12	'Drayton'	'KS'	(NULL)	3

ロー 2 ～ 9 は、CompanyName ごとにグループ化して生成されたローで、ロー 10 ～ 12 は、City と State の組み合わせでグループ化されたローです。ロー 1 は、一致したカッコ () のペアを使用して指定される空のグループ化セットで表される総合計です。空のグループ化セットは、GROUP BY に対する入力のすべてのローの分割 1 つを表します。

NULL 値は、グループ化セットで使用されない任意の式のプレースホルダとして使用されていることに注意してください。これは、結果セットが結合可能である必要があるためです。たとえば、ロー 2 ～ 9 は、クエリの 2 番目のグループ化セット (CompanyName) から得られます。グループ化セットには式として City または State が含まれないため、ロー 2 ～ 9 では City と State の値にプレースホルダの NULL が含まれますが、CompanyName の値には CompanyName に出現する重複しない値が含まれます。

NULL はプレースホルダとして使用されるため、プレースホルダの NULL とデータ内の実際の NULL を混乱しがちです。プレースホルダの NULL を NULL データと区別するには、GROUPING 関数を使用してください。

参照

- 「GROUPING 関数を使用した NULL の検出」557 ページ

例

次の例では、クエリから返される結果を調整するために GROUPING SETS を使用する方法と、結果をわかりやすく整理するために ORDER BY 句を使用する方法を示します。クエリは、各年 (Year) の四半期 (Quarter) ごとの注文の合計数と、年 (Year) ごとの合計数を返します。年 (Year)、四半期 (Quarter) の順で並べることで、結果を理解しやすくなります。

```
SELECT Year( OrderDate ) AS Year,
       Quarter( OrderDate ) AS Quarter,
       COUNT (*) AS Orders
FROM SalesOrders
GROUP BY GROUPING SETS ( ( Year, Quarter ), ( Year ) )
ORDER BY Year, Quarter;
```

このクエリは、次の結果を返します。

	Year	Quarter	Orders
1	2000	(NULL)	380
2	2000	1	87
3	2000	2	77
4	2000	3	91
5	2000	4	125
6	2001	(NULL)	268
7	2001	1	139
8	2001	2	119
9	2001	3	10

ロー 1 と 6 は、それぞれ 2000 年と 2001 年の注文の小計です。ロー 2 ～ 5 とロー 7 ～ 9 は、小計ローのディテールローに当たります。つまり、これらのローは、四半期ごとと年ごとの注文の合計数を示します。

この結果セットには、すべての年のすべての四半期の総合計がありません。総合計が含まれるようにするには、クエリで GROUPING SETS 指定に空のグループ化指定「()」を含める必要があります。

空のグループ化指定の使用

GROUP BY 句で空の GROUPING SETS 指定「()」を使用すると、結果で合計されるすべての項目について総合計のローが生成されます。総合計の行では、すべてのグループ化の式に対するすべての値にプレースホルダの NULL が含まれます。GROUPING 関数を使用すると、ローの基本となるデータで値を評価するため、プレースホルダの NULL と実際の NULL を区別できます。

重複したグループ化セットの指定

GROUPING SETS 句では、重複したグループ化指定を使用できます。この場合、SELECT 文の結果に同一のローが含まれます。

次のクエリには、重複したグループ化が含まれます。

```
SELECT City, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
GROUP BY GROUPING SETS( ( City ), ( City ) );
```

このクエリは、次の結果を返します。重複したグループ化の結果として、ロー 1 ～ 3 がロー 4 ～ 6 と同一になります。

	City	Cnt
1	'Drayton'	3
2	'Petersburg'	1
3	'Pembroke'	4
4	'Drayton'	3
5	'Petersburg'	1
6	'Pembroke'	4

適切な形式の実践

GROUP BY GROUPING SETS 句でのグループ化構文の解釈は、単純な GROUP BY 句の場合とは異なります。たとえば、GROUP BY (X, Y) は X と Y の値の異なる組み合わせによってグループ化されます。しかし GROUP BY GROUPING SETS (X, Y) の場合は、2 つの個別のグループ化セットを指定し、その 2 つのグループ化の結果がユニオンされます。つまり、結果は (X) でグループ化されてから、(Y) でグループ化された同じ結果にユニオンされます。

適切な形式を使用し、複雑な式の場合のあいまいさを避けるために、エラーが発生する可能性がある場合は指定内の各グループ化セットをカッコで囲んでください。たとえば、次の両方の文は正しく、セマンティック上同一ですが、2 番目が推奨される形式を反映した文です。

```
SELECT * FROM t GROUP BY GROUPING SETS ( X, Y );
SELECT * FROM t GROUP BY GROUPING SETS( ( X ), ( Y ) );
```

参照

- 「GROUPING 関数を使用した NULL の検出」557 ページ

GROUPING SETS のショートカットとしての ROLLUP と CUBE

複数の異なるデータ分割を単一の結果セットに連結する場合は、GROUPING SETS を使用すると便利です。ただし、多くのグループ化を指定する必要があり、かつ小計を含める場合は、ROLLUP 拡張と CUBE 拡張を使用できます。

ROLLUP 句と CUBE 句は、事前に定義された GROUPING SETS 指定のショートカットと見なすことができます。

ROLLUP は、空のグループ化セット「()」から始まり、追加する式を前の式に連結させるグループ化セットが次々と続くような、一連のグループ化を指定するのと同等です。たとえば、3 つのグループ化式 a、b、c があり、ROLLUP を指定した場合は、セット ()、(a)、(a, b)、(a, b, c) を使用して GROUPING SETS 句を指定した場合と同じ結果になります。この構成は、階層グループ化と呼ばれることもあります。

CUBE を使用すると、さらに多くのグループ化を実現できます。CUBE を指定することは、すべての可能な GROUPING SETS を指定するのと同等です。たとえば、同様の 3 つのグループ化式 a、b、c があり、CUBE を指定した場合は、セット ()、(a)、(a, b)、(a, c)、(b)、(b, c)、(c)、(a, b, c) を使用して GROUPING SETS 句を指定した場合と同じ結果になります。

ROLLUP または CUBE を指定する場合は、GROUPING 関数を使用して、結果内にあるプレースホルダの NULL を識別してください。プレースホルダの NULL は、ROLLUP または CUBE による結果セット内で暗黙的である小計のローが原因で発生します。

参照

- 「GROUPING 関数を使用した NULL の検出」557 ページ

ROLLUP 句

多くのアプリケーションに共通の要件は、グループ化属性の小計を左から右へ順番に計算することです。このパターンは、階層として参照されます。小計の計算が追加されることにより、詳細度を上げた追加のローが生成されるためです。ROLLUP キーワードを使用して ROLLUP 句を指定することにより、グループ化属性の階層を指定できます。

ROLLUP 句を使用したクエリでは、次のようなグループ化セットの階層が生成されます。ROLLUP 句に $(X_1,X_2, \ldots , X_n)$ という形式で n 個の GROUP BY 式が含まれている場合、ROLLUP 句によって次のように $n + 1$ 個のグループ化セットが生成されます。

```
{(), (X1), (X1,X2), (X1,X2,X3), ... , (X1,X2,X3, ... , Xn)}
```

例

次のクエリは、年ごとと四半期ごとに販売注文を要約し、次の表に示す結果セットを返します。

```
SELECT QUARTER( OrderDate ) AS Quarter,
       YEAR( OrderDate ) AS Year,
       COUNT( * ) AS Orders,
       GROUPING( Quarter ) AS GQ,
       GROUPING( Year ) AS GY
```

```
FROM SalesOrders
GROUP BY ROLLUP( Year, Quarter )
ORDER BY Year, Quarter;
```

このクエリは、次の結果を返します。

	Quarter	Year	Orders	GQ	GY
1	(NULL)	(NULL)	648	1	1
2	(NULL)	2000	380	1	0
3	1	2000	87	0	0
4	2	2000	77	0	0
5	3	2000	91	0	0
6	4	2000	125	0	0
7	(NULL)	2001	268	1	0
8	1	2001	139	0	0
9	2	2001	119	0	0
10	3	2001	10	0	0

結果セットの 1 番目のローは、2 年間のすべての四半期について、すべての注文の総合計 (648) を示します。

ロー 2 は 2000 年の注文の合計数 (380) を示し、ロー 3 ～ 6 はこの年の四半期ごとの注文の小計を示します。同様に、ロー 7 は 2001 年の注文の合計数 (268) を示し、ロー 8 ～ 10 はこの年の四半期ごとの小計を示します。

GROUPING 関数で返される値を使用して、総合計が含まれるローと小計のローを区別できます。ロー 2 と 7 では、四半期カラムは NULL で、GQ (Grouping by Quarter) カラムは値 1 です。これは、このローが年ごとのすべての四半期の注文の合計であることを示します。

同様に、ロー 1 の四半期 (Quarter) カラムと年 (Year) カラムは NULL で、GQ カラムと GY カラムは値 1 です。これは、このローがすべての年のすべての四半期における注文の合計であることを示します。

Transact-SQL WITH ROLLUP 構文のサポート

代わりに Transact-SQL 互換の構文である WITH ROLLUP を使用して、GROUP BY ROLLUP と同じ結果を得ることもできます。ただし、構文が多少異なり、構文に指定できるのは単純な GROUP BY 式リストだけです。

次のクエリは、前述の GROUP BY ROLLUP の例の場合と同等の結果を生成します。

```
SELECT QUARTER( OrderDate ) AS Quarter,
      YEAR( OrderDate ) AS Year,
      COUNT( * ) AS Orders,
      GROUPING( Quarter ) AS GQ,
      GROUPING( Year ) AS GY
FROM SalesOrders
GROUP BY Year, Quarter WITH ROLLUP
ORDER BY Year, Quarter;
```

参照

- 「GROUP BY 句」『SQL Anywhere サーバ SQL リファレンス』

CUBE 句

ROLLUP 句で提供される階層グループ化パターンの代わりに、データキューブを作成することもできます。データキューブとは、GROUP BY 式の可能な組み合わせを使用した入力の *n* 次元要約で、CUBE 句が使用されます。CUBE 句の結果は、各値セットの要素のすべての可能な組み合わせを含む積集合になります。複雑なデータ分析で非常に役立ちます。

CUBE 句に $(X_1,X_2, . . .,X_n)$ という形式で *n* 個の GROUPING 式が存在する場合、CUBE によって次のように 2^n 個のグループ化セットが生成されます。

```
{(), (X1), (X1,X2), (X1,X2,X3), ... , (X1,X2,X3, ...,Xn),
(X2), (X2,X3), (X2,X3,X4), ... , (X2,X3,X4, ... , Xn), ... , (Xn)}.
```

例

次のクエリは、年ごと、四半期ごと、および四半期単位に販売注文を要約し、次の表に示す結果セットを生成します。

```
SELECT QUARTER( OrderDate ) AS Quarter,
      YEAR( OrderDate ) AS Year,
      COUNT( * ) AS Orders,
      GROUPING( Quarter ) AS GQ,
      GROUPING( Year ) AS GY
FROM SalesOrders
GROUP BY CUBE ( Year, Quarter )
ORDER BY Year, Quarter;
```

このクエリは、次の結果を返します。

	Quarter	Year	Orders	GQ	GY
1	(NULL)	(NULL)	648	1	1
2	1	(NULL)	226	0	1
3	2	(NULL)	196	0	1
4	3	(NULL)	101	0	1
5	4	(NULL)	125	0	1

	Quarter	Year	Orders	GQ	GY
6	(NULL)	2000	380	1	0
7	1	2000	87	0	0
8	2	2000	77	0	0
9	3	2000	91	0	0
10	4	2000	125	0	0
11	(NULL)	2001	268	1	0
12	1	2001	139	0	0
13	2	2001	119	0	0
14	3	2000	10	0	0

結果セットの 1 番目のローは、2000 年と 2001 年を結合した、すべての四半期についての、すべての注文の総合計 (648) を示します。

ロー 2 ～ 5 は、すべての年の暦四半期ごとの販売注文を要約します。

ロー 6 と 11 の Orders は、それぞれ 2000 年と 2001 年の注文の合計を示します。

ロー 7 ～ 10 と 12 ～ 14 は、それぞれ 2000 年と 2001 年の四半期ごとの合計を示します。

GROUPING 関数で返される値を使用して、総合計が含まれるローと小計のローを区別できます。ロー 6 と 11 は、四半期 (Quarter) カラムは NULL で、GQ (Grouping by Quarter) カラムは値 1 です。これは、このローが年ごとにすべての四半期の注文 (Orders) の合計であることを示します。

注意
CUBE は指数関数的な個数のグループ化セットを生成するため、CUBE を使用して生成される結果セットは非常に大きくなることがあります。このため、64 個を超える GROUP BY 式が含まれる GROUP BY 句はサポートされていません。この上限を超える文は、SQLCODE -944 (SQLSTATE 42WA1) で失敗します。

Transact-SQL WITH CUBE 構文のサポート

代わりに Transact-SQL 互換の構文である WITH CUBE を使用して、GROUP BY CUBE と同じ結果を得ることもできます。ただし、構文が多少異なり、構文に指定できるのは単純な GROUP BY 式リストだけです。

次のクエリは、前述の GROUP BY CUBE の例の場合と同等の結果を生成します。

```
SELECT QUARTER( OrderDate ) AS Quarter,
       YEAR( OrderDate ) AS Year,
       COUNT( * ) AS Orders,
       GROUPING( Quarter ) AS GQ,
```

```
    GROUPING( Year ) AS GY
FROM SalesOrders
GROUP BY Year, Quarter WITH CUBE
ORDER BY Year, Quarter;
```

参照

- 「ROLLUP、CUBE、または GROUPING SETS 操作に対する GROUP BY リストの式が多すぎます」『エラーメッセージ』
- 「GROUP BY 句」『SQL Anywhere サーバ SQL リファレンス』

GROUPING 関数を使用した NULL の検出

ROLLUP と CUBE で作成される合計や小計のローには、グループ化で使用されなかった SELECT リストで指定したあらゆるカラムに、プレースホルダの NULL が含まれます。結果を検査しているときは、小計のローにある NULL がプレースホルダの NULL なのか、それともローの基本になるデータの評価による NULL なのかを区別できません。その結果、ディテールロー、小計ロー、総合計ローを区別することも困難になります。

GROUPING 機能を使用すると、プレースホルダの NULL を基本となるデータによる NULL と区別できます。グループ化セット指定から *group-by-expression* を 1 つ使用して GROUPING 関数を指定した場合、この関数は、プレースホルダの NULL の場合は 1 を返し、そのローの基本となるデータに存在する値 (通常は NULL) を反映している場合は 0 を返します。

たとえば、次のクエリは下の表に示される結果セットを返します。

```
SELECT Employees.EmployeeID AS Employee,
  YEAR( OrderDate ) AS Year,
  COUNT( SalesOrders.ID ) AS Orders,
  GROUPING( Employee ) AS GE,
  GROUPING( Year ) AS GY
FROM Employees LEFT OUTER JOIN SalesOrders
  ON Employees.EmployeeID = SalesOrders.SalesRepresentative
WHERE Employees.Sex IN ( 'F' )
  AND Employees.State IN ( 'TX' , 'NY' )
GROUP BY GROUPING SETS ( ( Year, Employee ), ( Year ), ( ) )
ORDER BY Year, Employee;
```

このクエリは、次の結果を返します。

	Employees	Year	Orders	GE	GY
1	(NULL)	(NULL)	54	1	1
2	(NULL)	(NULL)	0	1	0
3	102	(NULL)	0	0	0
4	390	(NULL)	0	0	0
5	1062	(NULL)	0	0	0

	Employees	Year	Orders	GE	GY
6	1090	(NULL)	0	0	0
7	1507	(NULL)	0	0	0
8	(NULL)	2000	34	1	0
9	667	2000	34	0	0
10	(NULL)	2001	20	1	0
11	667	2001	20	0	0

この例では、空のグループ化セット「()」が指定されたため、ロー 1 は注文の総合計 (54) を表します。GE と GY の両方に 1 が含まれていることに注意してください。これは、Employees カラムと Year カラムの NULL がそれぞれ Employees と Year のプレースホルダ NULL であることを示しています。

ロー 2 は小計のローです。GE カラムの 1 は、Employees カラムの NULL がプレースホルダ NULL であることを示しています。GY カラムの 0 は、Year カラムの NULL が基本となるデータの評価による結果であり、プレースホルダ NULL ではないことを示します。この場合、このローは注文のない従業員を表します。

ロー 3 ～ 7 は、従業員ごとの、Year が NULL である注文の合計数を示しています。つまり、これらは注文のない Texas と New York に住む女性従業員のものです。これらのローはロー 2 のディテールローです。つまり、ロー 2 はロー 3 ～ 7 の合計です。

ロー 8 は、2000 年のすべての従業員の分を合わせた注文数を示す小計ローです。ロー 9 は、ロー 8 の単一ディテールローです。

ロー 10 は、2001 年のすべての従業員の分を合わせた注文数を示す小計ローです。ロー 11 は、ロー 10 の単一ディテールローです。

参照

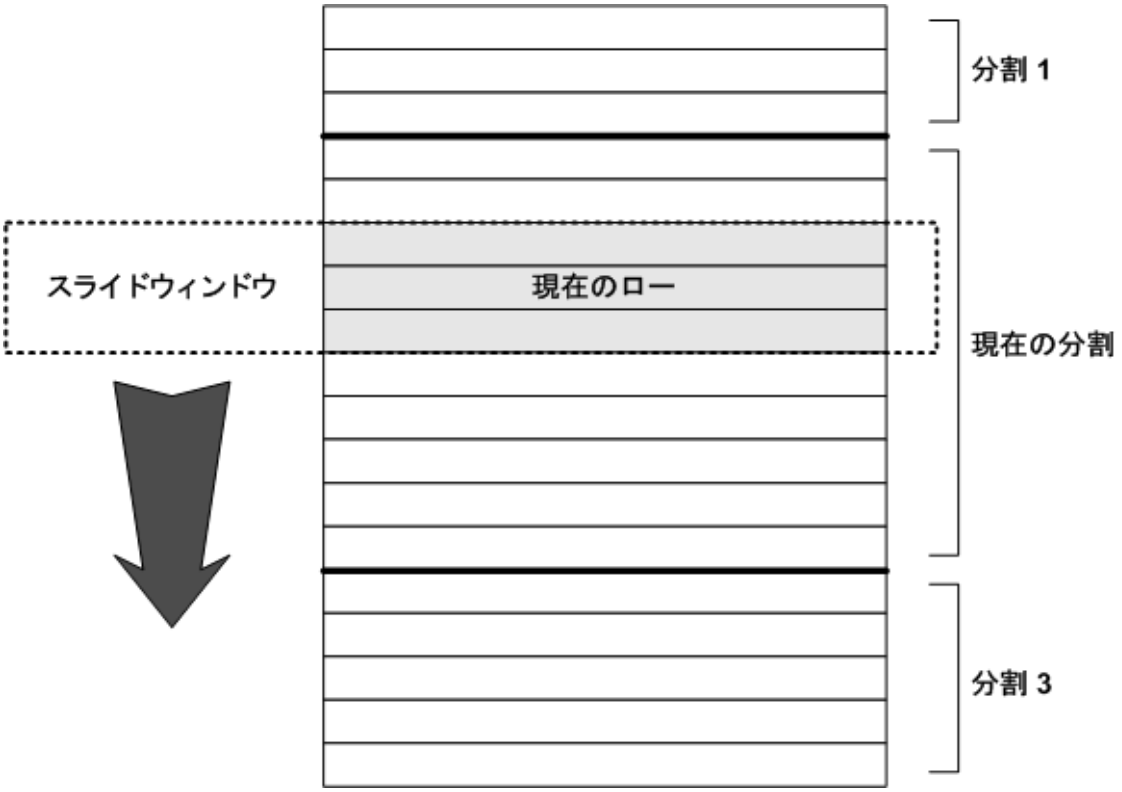
- 「GROUPING 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』

Window 関数

OLAP の機能には、入力ローの処理に合わせて入力ローを上から下に移動するスライド「ウィンドウ」の概念が含まれます。ウィンドウの移動とともにウィンドウ内のデータに対する追加の計算を実行できるため、セマンティック上同等なセルフジョインクエリや相関サブクエリを使用する方法よりも効率的な方法で、詳細な分析を実行できます。

ウィンドウの境界は、データから抽出しようとする情報を基に設定します。ウィンドウには、ウィンドウ定義内のグループ化指定に従って分割された入力データの、1 つ、複数、またはすべてのローが含まれます。ウィンドウは入力データを上から下に移動し、必須の計算を実行するために必要なローを採用します。

次の図は、入力ローが処理されるのに伴って移動するウィンドウを示しています。データ分割は、ウィンドウ定義に指定された、入力ローのグループ化を反映します。グループ化が指定されていない場合は、すべての入力ローで 1 分割であると見なされます。ウィンドウの長さ (つまりウィンドウに含まれるローの数) と、現在のローと比較したウィンドウのオフセットは、ウィンドウ定義で指定した境界を反映します。



ウィンドウ定義

SQL のウィンドウ拡張を使用して、ウィンドウの境界や、入力ローの分割や順序付けを設定できます。論理的には、GROUP BY 句で定義されたグループが作成された後、最終の SELECT リストの評価とクエリの ORDER BY 句の前に、クエリ仕様の結果を計算するセマンティックの一部として分割が作成されます。SQL 文における句の評価順は次のようになります。

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. WINDOW

6. DISTINCT

7. ORDER BY

クエリを形成する際には、評価順の影響を考慮する必要があります。たとえば、同じ SELECT クエリブロックにある Window 関数を参照する式を述部にすることはできません。ただし、クエリブロックを派生テーブルに入れることで、派生テーブルを使用して述部を指定できます。次のクエリを実行すると、Window 関数が述部に使用されているという旨のメッセージが表示されて、クエリは失敗します。

```
SELECT DepartmentID, Surname, StartDate, Salary,
      SUM( Salary ) OVER ( PARTITION BY DepartmentID
        ORDER BY StartDate
        RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW ) AS "Sum_Salary"
  FROM Employees
  WHERE State IN ( 'CA', 'UT', 'NY', 'AZ' )
    AND DepartmentID IN ( '100', '200' )
  GROUP BY DepartmentID, Surname, StartDate, Salary
  HAVING Salary > 0 AND "Sum_Salary" > 200
  ORDER BY DepartmentID, StartDate;
```

目的の結果を得るためには、派生テーブル (DT) を使用して述部を指定します。

```
SELECT * FROM ( SELECT DepartmentID, Surname, StartDate, Salary,
                  SUM( Salary ) OVER ( PARTITION BY DepartmentID
                    ORDER BY StartDate
                    RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW ) AS
"Sum_Salary"
              FROM Employees
              WHERE State IN ( 'CA', 'UT', 'NY', 'AZ' )
                AND DepartmentID IN ( '100', '200' )
              GROUP BY DepartmentID, Surname, StartDate, Salary
              HAVING Salary > 0
              ORDER BY DepartmentID, StartDate ) AS DT
  WHERE DT.Sum_Salary > 200;
```

ウィンドウ分割は GROUP BY 演算子に続くので、分割を実行する計算で、任意の集合関数 (SUM、AVG、VARIANCE など) の結果を利用できます。そのため、クエリの GROUP BY 句や ORDER BY 句だけでなく、ウィンドウを使用することでもグループ化と順序付けの操作を実行できます。

ウィンドウ指定の定義

Window 関数で操作するウィンドウを定義するときは、次の項目を 1 つまたは複数指定します。

- **分割 (PARTITION BY 句)**　PARTITION BY 句により、入力ローのグループ化方法を定義します。省略すると、入力全体が単一の分割として扱われます。指定した内容に応じて、1 つ、複数、またはすべての入力ローから分割を作成できます。2 つの分割からのデータが混合することはありません。つまり、ウィンドウが 2 つの分割の境界に達すると、分割内のデータの処理が終了してから、次の分割内のデータの処理が開始されます。ウィンドウの境界の定義方法に応じて、ウィンドウのサイズが分割の先頭と末尾で変化する可能性があります。

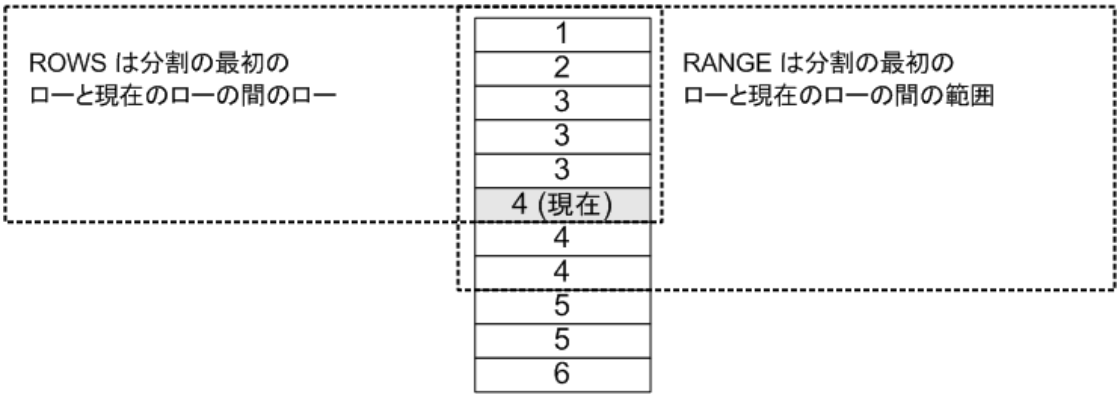
- **順序付け (ORDER BY 句)**　ORDER BY 句により、Window 関数による処理の前に、入力ローの順序を決める方法を定義します。RANGE 句を使用して境界を指定する場合、またはラン

キング関数がウィンドウを参照する場合にかぎり、ORDER BY 句が必要です。それ以外の場合、ORDER BY 句はオプションです。省略すると、データベースサーバが最も効率的な順序で入力ローを処理します。

- **境界 (RANGE 句および ROWS 句)**　現在のローは、ウィンドウの開始ローと終了ローを判断するための参照ポイントになります。ウィンドウ定義の RANGE 句と ROWS 句を使用して、境界を設定できます。RANGE は、現在のローの値からのオフセットとして**データ値の範囲**を指定することでウィンドウを定義します。範囲を計算するためにはデータが順序付けられている必要があるため、RANGE を指定する場合は ORDER BY 句も指定する必要があります。

  ROWS は、現在のローからのオフセットとして**ロー数**を指定することで、ウィンドウを定義します。

  RANGE は、データ値の範囲としてローのセットを定義するため、RANGE ウィンドウに含まれるローは現在のローを越える場合があります。この点は ROWS とは異なります。次の図は、ROWS 句と RANGE 句の違いを示したものです。



  ROWS 句と RANGE 句内では、現在のローを基準にしてウィンドウの開始ローと終了ローを (オプションで) 指定できます。これには、PRECEDING 句、BETWEEN 句、および FOLLOWING 句を使用します。これらの句には、式の他に、UNBOUNDED と CURRENT ROW の各キーワードも指定できます。ウィンドウに境界が定義されていない場合、ウィンドウ境界はデフォルトで次のようになります。

  ○ウィンドウ指定に ORDER BY 句が含まれている場合は、RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW を指定したことと同義になります。

  ○ウィンドウ指定に ORDER BY 句が含まれていない場合は、ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING を指定したことと同義になります。

  次の表は、ウィンドウ境界の例とウィンドウに含まれるローの説明を示したものです。

指定	意味
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW	分割の先頭から開始し、現在のローで終了します。累積の結果 (累積合計など) を計算する場合に使用してください。
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING	分割内のすべてのローを使用します。分割の各ローに対して集合関数の値を同一にする場合に使用してください。
ROWS BETWEEN *x* PRECEDING AND *y* FOLLOWING	現在のローから *x* の距離にある開始ローと *y* の距離にある終了ロー (境界値を含む) からなる固定サイズの移動するウィンドウを作成します。移動平均を計算する場合や隣接するロー間の値の差分を計算する場合には、この例を使用してください。 複数のローからなる移動するウィンドウでは、分割の最初のローや最後のローを計算するときに NULL になります。これは、現在のローが分割のまさに最初または最後のローである場合に、計算で使用できる直前または直後のローが存在しないためです。そのため、代わりに NULL 値が使用されます。
ROWS BETWEEN CURRENT ROW AND CURRENT ROW	1 つのロー (現在のロー) からなるウィンドウ。
RANGE BETWEEN 5 PRECEDING AND 5 FOLLOWING	ロー内の値に基づいてウィンドウを作成します。たとえば、現在のローに対して、ORDER BY 句に指定されたカラムに値 10 が含まれているとします。ウィンドウのサイズを RANGE BETWEEN 5 PRECEDING AND 5 FOLLOWING と指定した場合、最初のローにはそのカラムに 5、最後のローにはそのカラムに 15 がそれぞれ含まれるのに必要な大きさとして、ウィンドウのサイズを指定します。ウィンドウが分割を移動すると、範囲仕様を満たすのに必要なサイズに応じて、ウィンドウのサイズが変化します。

ウィンドウ指定はできるかぎり明示的にしてください。明示的に指定しないと、デフォルトが予期した結果を返さない場合があります。

連続でない値の場合は、RANGE 句を使用して、Window 関数の入力の差に起因する問題を回避します。RANGE 句を使用してウィンドウ境界が設定された場合、データベースサーバは隣接するローや重複する値を含むローを自動的に処理します。

RANGE では、符号なし整数値を使用します。ORDER BY 式のドメインと RANGE 句で指定した値のドメインに応じて、範囲式のトランケーションが発生することがあります。

ランキング関数やロー番号付け関数を使用するときは、ウィンドウの境界を指定しないでください。

ウィンドウ定義：OVER 句と WINDOW 句を使用したインライン定義

ウィンドウを定義する方法には次の 3 つがあります。

●インライン (Window 関数の OVER 句内)

●WINDOW 句内

●インラインと WINDOW 句内で部分的に指定

ただし、以降の項に示すとおり、方法によっては制限があります。

インライン定義 (Window 関数の OVER 句内)

ウィンドウ定義は、Window 関数の OVER 句に配置できます。このことを、ウィンドウを**インライン**で定義するといいます。

たとえば、次の文は、2001 年 7 月と 8 月に出荷されたすべての製品と、出荷日ごとの累積出荷数量について、サンプルデータベースに問い合わせます。ウィンドウはインラインで定義されています。

```
SELECT p.ID, p.Description, s.Quantity, s.ShipDate,
  SUM( s.Quantity ) OVER ( PARTITION BY s.ProductID
     ORDER BY s.ShipDate
     ROWS BETWEEN UNBOUNDED PRECEDING
     AND CURRENT ROW ) AS Cumulative_qty
FROM SalesOrderItems s JOIN Products p
  ON ( s.ProductID = p.ID )
WHERE s.ShipDate BETWEEN '2001-07-01' AND '2001-08-31'
ORDER BY p.ID;
```

このクエリは、次の結果を返します。

	ID	Description	Quantity	ShipDate	Cumulative_qty
1	301	V-neck	24	2001-07-16	24
2	302	Crew Neck	60	2001-07-02	60
3	302	Crew Neck	36	2001-07-13	96

	ID	Description	Quantity	ShipDate	Cumulative_qty
4	400	Cotton Cap	48	2001-07-05	48
5	400	Cotton Cap	24	2001-07-19	72
6	401	Wool Cap	48	2001-07-09	48
7	500	Cloth Visor	12	2001-07-22	12
8	501	Plastic Visor	60	2001-07-07	60
9	501	Plastic Visor	12	2001-07-12	72
10	501	Plastic Visor	12	2001-07-22	84
11	601	Zipped Sweatshirt	60	2001-07-19	60
12	700	Cotton Shorts	24	2001-07-26	24

この例では、2 つのテーブルのジョインとクエリの WHERE 句の適用後に、SUM Window 関数の計算が発生します。クエリは次のように処理されます。

1. ProductID の値を基に、入力ローを分割 (グループ化) します。
2. 各分割内で、ShipDate の値を基にローをソートします。
3. 分割内の各ローについて、Quantity の値に対して SUM 関数を評価します。このとき、各分割の最初の (ソートされた) ローからなるスライドウィンドウを使用します。

WINDOW 句の定義

前述のクエリの別の構成として、WINDOW 句を使用して、ウィンドウを使用する関数とは別にウィンドウを指定します。次に、各関数の OVER 句からウィンドウを参照します。

この例で、WINDOW 句はウィンドウ Cumulative を作成し、データを ProductID ごとに分割し、ShipDate で並べ替えます。SUM 関数は、その OVER 句でウィンドウを参照し、ROWS 句を使用してウィンドウのサイズを定義します。

```
SELECT p.ID, p.Description, s.Quantity, s.ShipDate,
   SUM( s.Quantity ) OVER ( Cumulative
   ROWS BETWEEN UNBOUNDED PRECEDING
   AND CURRENT ROW ) AS cumulative_qty
FROM SalesOrderItems s
JOIN Products p ON ( s.ProductID = p.ID )
WHERE s.ShipDate BETWEEN '2001-07-01' AND '2001-08-31'
WINDOW Cumulative AS ( PARTITION BY s.ProductID ORDER BY s.ShipDate )
ORDER BY p.ID;
```

WINDOW 句構文を使用するときは、次の制限があります。

- PARTITION BY 句を指定する場合は、WINDOWS 句内に配置する必要があります。
- ROWS 句または RANGE 句を指定する場合は、参照元関数の OVER 句内に配置する必要があります。
- ウィンドウに ORDER BY 句を指定する場合は、WINDOW 句内か参照元関数の OVER 句内に配置できますが、両方には配置できません。
- WINDOW 句は、SELECT 文の ORDER BY 句に先行する必要があります。

インラインおよび WINDOW 句の定義の組み合わせ

ウィンドウ定義の一部をインラインで指定し、残りを WINDOW 句で定義できます。次に例を示します。

```
AVG() OVER ( windowA
        ORDER BY expression )...
...
WINDOW windowA AS ( PARTITION BY expression )
```

この方法でウィンドウ定義を分割すると、次の制限が適用されます。

- Window 関数構文では PARTITION BY 句を使用できません。
- Window 関数構文または WINDOW 句のいずれかで ORDER BY 句を使用することはできますが、両方では使用できません。
- RANGE 句または ROWS 句を WINDOW 句に含めることはできません。

参照

- 「WINDOW 句」『SQL Anywhere サーバ SQL リファレンス』
- 「Window 集合関数」565 ページ
- 「Window ランキング関数」584 ページ
- 「ウィンドウ定義」559 ページ

SQL Anywhere の Window 関数

入力ローのセットに対して分析演算を実行できる関数は、Window 関数と呼ばれます。たとえば、すべてのランキング関数、およびほとんどの集合関数は「Window 関数」です。Window 関数を使用すると、データの追加分析を実行できます。この操作を行うには、入力ローを処理前に分割してソートし、次にサイズを設定可能な入力が進むウィンドウ内でローを処理します。

Window 関数には、Window 集合関数、Window ランキング関数、ロー番号付け関数の 3 種類があります。

Window 集合関数

Window 集合関数は、入力内のローの指定されたセットに対する値を返します。たとえば、Window 関数を使用して、指定した期間における会社の販売数の移動平均を計算できます。

Window 集合関数は次の 3 つのカテゴリに分類されます。

- **基本集合関数**　サポートされる基本集合関数のリストを次に示します。
  - 「SUM 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「AVG 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「MAX 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「MIN 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「MEDIAN 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「FIRST_VALUE 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「LAST_VALUE 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「COUNT 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- **標準偏差関数と平方偏差関数**　サポートされる標準偏差関数と平方偏差関数のリストを次に示します。
  - 「STDDEV 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「STDDEV_POP 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「STDDEV_SAMP 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「VAR_POP 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「VAR_SAMP 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「VARIANCE 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- **相関関数と線形回帰関数**　サポートされる相関関数と線形回帰関数のリストを次に示します。
  - 「COVAR_POP 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「COVAR_SAMP 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「REGR_AVGX 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「REGR_AVGY 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「REGR_COUNT 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「REGR_INTERCEPT 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「REGR_R2 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「REGR_SLOPE 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「REGR_SXX 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「REGR_SXY 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
  - 「REGR_SYY 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』

参照

- 「基本集合関数」566 ページ
- 「相関関数と線形回帰関数」583 ページ
- 「標準偏差関数と平方偏差関数」579 ページ

基本集合関数

複雑なデータ分析では、複数レベルの集約が必要になることがあります。GROUP BY 句に加え、またはその代わりに、ウィンドウ分割や並べ替えを使用すると、複雑な SQL クエリを非常に柔軟に構成できます。たとえば、ウィンドウ構成と単純な集合関数を組み合わせると、移動平均、移動合計、移動最小、移動最大、累積合計などの値を計算できます。

サポートしている基本集合関数を次に示します。

- **SUM 関数**　ローグループごとに、指定された式の合計を返します。
- **AVG 関数**　対象となるローセットの、数値式の平均値またはユニークな値からなるセットの平均値を返します。
- **MAX 関数**　各ローグループで見つかった式の最大値を返します。
- **MIN 関数**　各ローグループで見つかった式の最小値を返します。
- **MEDIAN 関数**　ローのセットの数値式の中央値を返します。
- **FIRST_VALUE 関数**　ウィンドウの最初のローの値を返します。この関数では、ウィンドウを指定する必要があります。
- **LAST_VALUE 関数**　ウィンドウの最後のローの値を返します。この関数では、ウィンドウを指定する必要があります。
- **COUNT 関数**　指定された式の条件を満たすローの数を返します。

参照

- 「SUM 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「AVG 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「MAX 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「MIN 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「MEDIAN 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「FIRST_VALUE 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「LAST_VALUE 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「COUNT 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「Window 関数」558 ページ

SUM 関数の例

次の例は、Window 関数として使用される SUM 関数を示したものです。クエリは、DepartmentID 別にデータを分割した結果セットを返し、在籍経験が長い人から順に従業員の給与の累積合計 (Sum_Salary) を算出します。結果セットには、カリフォルニア州、ユタ州、ニューヨーク州、またはアリゾナ州に住む従業員のみが含まれます。Sum_Salary のカラムは、従業員の給与の累積合計です。

```
SELECT DepartmentID, Surname, StartDate, Salary,
SUM( Salary ) OVER ( PARTITION BY DepartmentID
   ORDER BY StartDate
   RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
AS "Sum_Salary"
FROM Employees
WHERE State IN ( 'CA', 'UT', 'NY', 'AZ' )
  AND DepartmentID IN ( '100', '200' )
ORDER BY DepartmentID, StartDate;
```

次のテーブルは、クエリからの結果セットを示します。結果セットは、DepartmentID ごとに分割されています。

	DepartmentID	Surname	StartDate	Salary	Sum_Salary
1	100	Whitney	1984-08-28	45700.00	45700.00
2	100	Cobb	1985-01-01	62000.00	107700.00
3	100	Shishov	1986-06-07	72995.00	180695.00
4	100	Driscoll	1986-07-01	48023.69	228718.69
5	100	Guevara	1986-10-14	42998.00	271716.69
6	100	Wang	1988-09-29	68400.00	340116.69
7	100	Soo	1990-07-31	39075.00	379191.69
8	100	Diaz	1990-08-19	54900.00	434091.69
9	200	Overbey	1987-02-19	39300.00	39300.00
10	200	Martel	1989-10-16	55700.00	95000.00
11	200	Savarino	1989-11-07	72300.00	167300.00
12	200	Clark	1990-07-21	45000.00	212300.00
13	200	Goggin	1990-08-05	37900.00	250200.00

DepartmentID 100 の場合、カリフォルニア州、ユタ州、ニューヨーク州、またはアリゾナ州に住む従業員の給与の累積合計は 434,091.69 ドルで、部門 200 の従業員の累積合計は 250,200.00 ドルです。

隣接ローのデルタの計算

2 つのウィンドウ (現在のローと直前のローのそれぞれに対するウィンドウ) を使用すると、隣接ローのデルタ (変化量) を計算できます。たとえば、次のクエリの結果では、ある従業員とその直前の従業員の給与のデルタ (Delta) を計算します。

```
SELECT EmployeeID AS EmployeeNumber,
  Surname AS LastName,
  SUM( Salary ) OVER ( ORDER BY BirthDate
    ROWS BETWEEN CURRENT ROW AND CURRENT ROW )
    AS CurrentRow,
  SUM( Salary ) OVER ( ORDER BY BirthDate
    ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING )
    AS PreviousRow,
  ( CurrentRow - PreviousRow ) AS Delta
FROM Employees
WHERE State IN ( 'NY' );
```

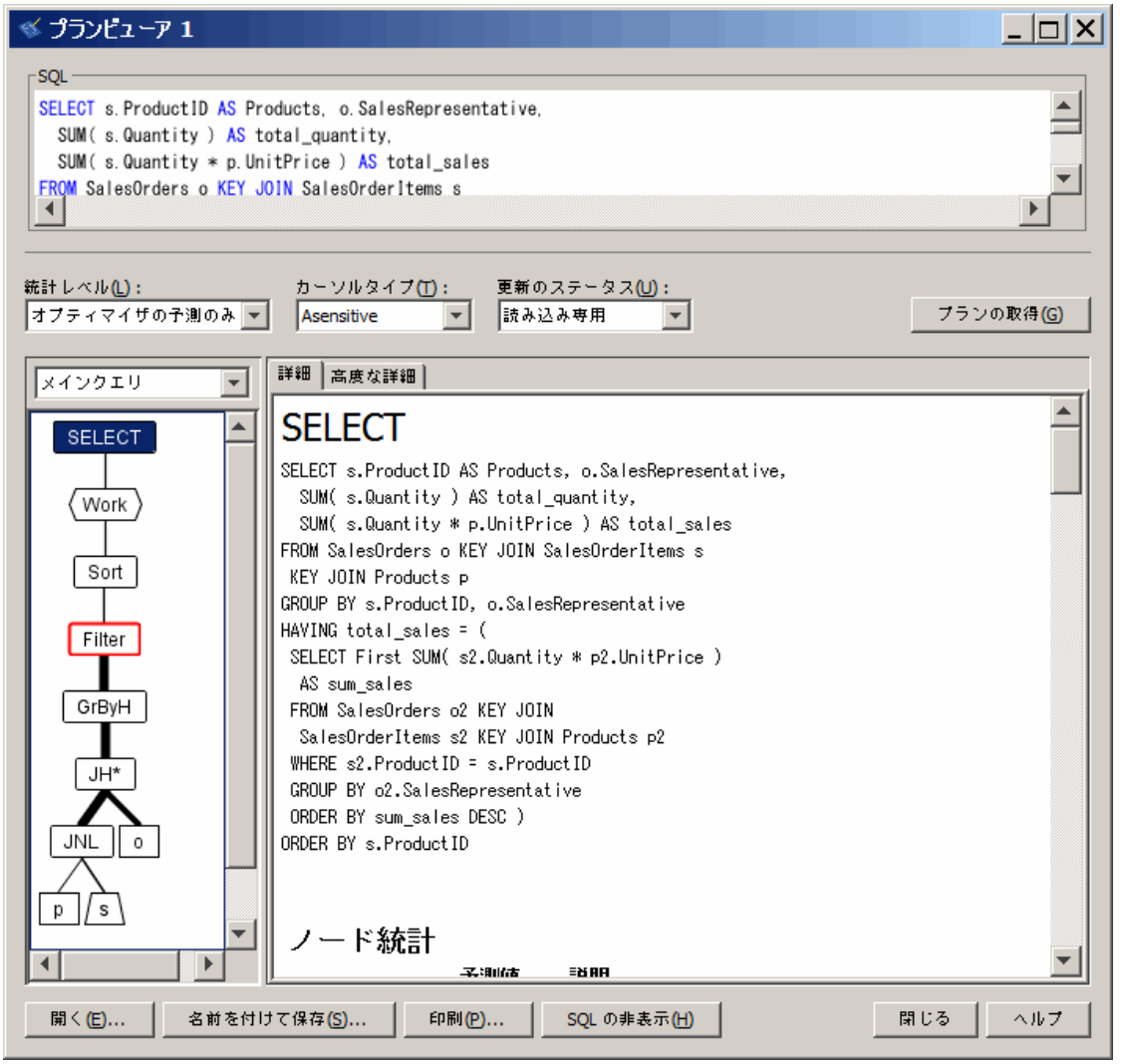
	EmployeeNumber	LastName	CurrentRow	PreviousRow	Delta
1	913	Martel	55700.000	(NULL)	(NULL)
2	1062	Blaikie	54900.000	55700.000	-800.000
3	249	Guevara	42998.000	54900.000	-11902.000
4	390	Davidson	57090.000	42998.000	14092.000
5	102	Whitney	45700.000	57090.000	-11390.000
6	1507	Wetherby	35745.000	45700.000	-9955.000
7	1751	Ahmed	34992.000	35745.000	-753.000
8	1157	Soo	39075.000	34992.000	4083.000

CurrentRow ウィンドウでは、ウィンドウのサイズが ROWS BETWEEN CURRENT ROW AND CURRENT ROW に設定されているため、SUM は現在のローのみに対して実行されます。同様に、PreviousRow ウィンドウでは、ウィンドウのサイズが ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING に設定されているため、SUM は直前のローのみに対して実行されます。最初のローには先行するローがないため、このローの PreviousRow の値は NULL です。そのため Delta の値も NULL になります。

複雑な分析

次のクエリを考えてみます。このクエリは、データベース内で製品ごとに最上位の営業担当者 (総売り上げで定義) をリストします。

```
SELECT s.ProductID AS Products, o.SalesRepresentative,
    SUM( s.Quantity ) AS total_quantity,
    SUM( s.Quantity * p.UnitPrice ) AS total_sales
  FROM SalesOrders o KEY JOIN SalesOrderItems s
   KEY JOIN Products p
  GROUP BY s.ProductID, o.SalesRepresentative
  HAVING total_sales = (
   SELECT First SUM( s2.Quantity * p2.UnitPrice )
      AS sum_sales
    FROM SalesOrders o2 KEY JOIN
      SalesOrderItems s2 KEY JOIN Products p2
    WHERE s2.ProductID = s.ProductID
    GROUP BY o2.SalesRepresentative
    ORDER BY sum_sales DESC )
  ORDER BY s.ProductID;
```



このクエリは、次の結果を返します。

	Products	SalesRepresentative	total_quantity	total_sales
1	300	299	660	5940.00
2	301	299	516	7224.00
3	302	299	336	4704.00
4	400	299	458	4122.00
5	401	902	360	3600.00

	Products	SalesRepresentative	total_quantity	total_sales
6	500	949	360	2520.00
7	501	690	360	2520.00
8	501	949	360	2520.00
9	600	299	612	14688.00
10	601	299	636	15264.00
11	700	299	1008	15120.00

元のクエリは、ProductID をサブクエリの相関外部参照として、ある製品の最高売り上げを判断する相関サブクエリを使用して作成されています。ただし、この場合のようにネストされたクエリを使用すると、コストの高いオプションになることがあります。これは、サブクエリに GROUP BY 句だけでなく、GROUP BY 句内の ORDER BY 句も含まれるからです。そのため、クエリオプティマイザは、同じセマンティックを保持したままになり、このネストされたクエリをジョインとして書き換えることができません。したがって、クエリの実行中は、外部ブロック内で計算される派生ローごとにサブクエリが評価されます。

グラフィカルプランでコストの高い Filter 述部に注意してください。オプティマイザは、クエリの実行コストの 99% がこのプラン演算子に起因するものであると推定します。サブクエリのプランでは、メインブロックのフィルタ演算子のコストが高くなる理由を明確にしています。サブクエリには、ネストループジョインが 2 つと、ハッシュ GROUP BY 演算が 1 つ、ソートが 1 つ含まれます。

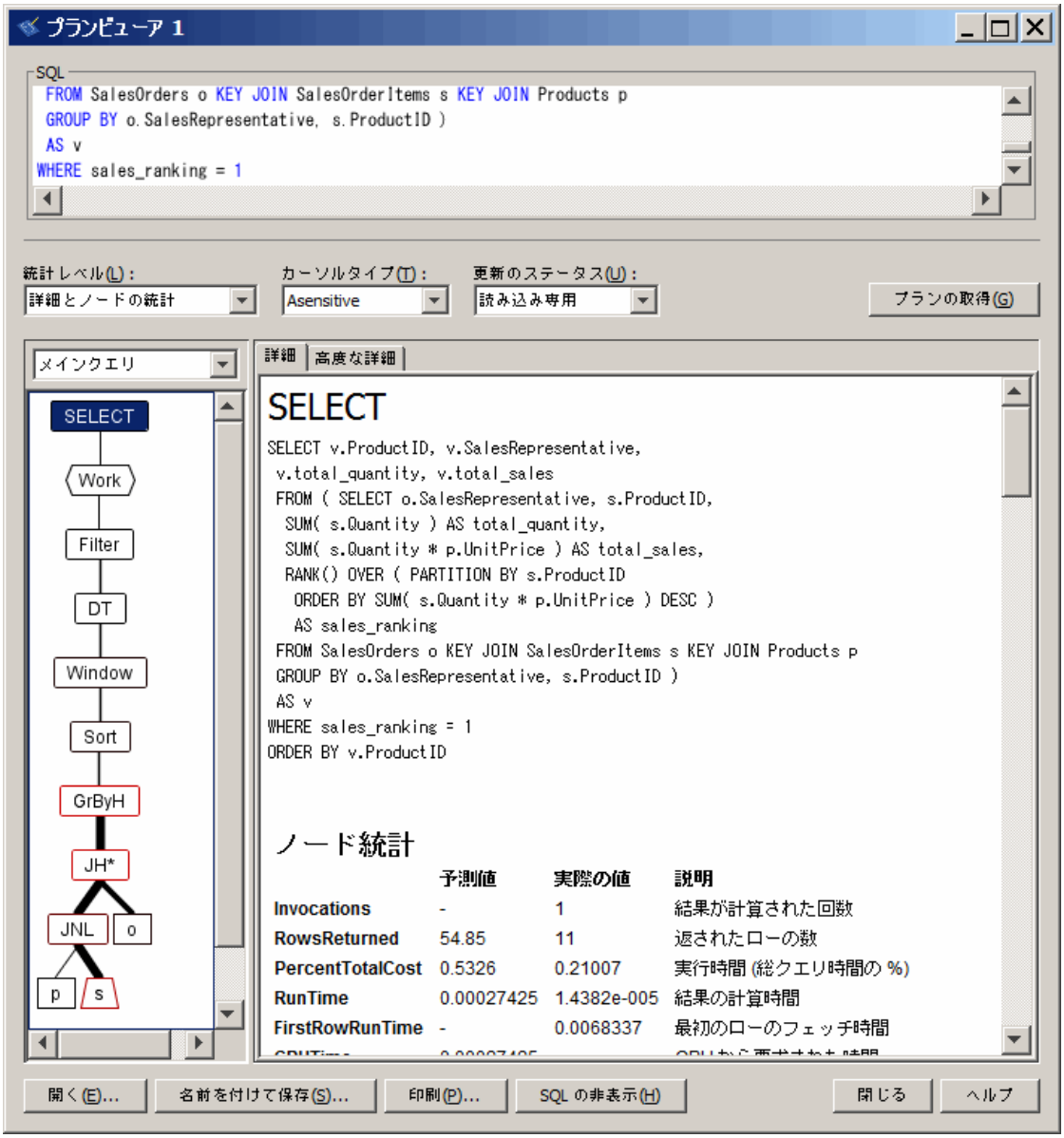
ランキング関数を使用した書き換え

ランキング関数を使用した同じクエリの書き換えでは、同じ結果が非常に効率よく計算されます。

```
SELECT v.ProductID, v.SalesRepresentative,
  v.total_quantity, v.total_sales
  FROM ( SELECT o.SalesRepresentative, s.ProductID,
          SUM( s.Quantity ) AS total_quantity,
          SUM( s.Quantity * p.UnitPrice ) AS total_sales,
```

```
        RANK() OVER ( PARTITION BY s.ProductID
          ORDER BY SUM( s.Quantity * p.UnitPrice ) DESC )
          AS sales_ranking
        FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
        GROUP BY o.SalesRepresentative, s.ProductID )
        AS v
    WHERE sales_ranking = 1
    ORDER BY v.ProductID;
```

この書き換えられたクエリは、プランがより単純になります。



GROUP BY 句が処理されてから SELECT リスト項目の評価とクエリの ORDER BY 句の処理が行われるまでに、ウィンドウ演算子が計算されます。グラフィカルなプランでわかるように、3 つのテーブルのジョイン後、ジョインされたローは SalesRepresentative 属性と ProductID 属性の組み合わせでグループ化されます。したがって、total_quantity と total_sales の SUM 集合関数は、SalesRepresentative と ProductID の組み合わせごとに計算できます。

GROUP BY 句の評価に続いて、中間結果のローを total_sales の降順にランク付けするために、ウィンドウを使用して RANK 関数が計算されます。WINDOW 指定には PARTITION BY 句が含まれます。それによって、GROUP BY 句の結果が、今度は ProductID ごとに再分割 (再グループ化) されます。そのため、RANK 関数は、総売り上げの降順で製品ごとにローをランク付けしますが、その製品を販売した営業担当者はすべてまとめられます。このランキングでは、最上位の営業担当者を特定するには、ランクが 1 ではない派生テーブルのローを拒否するように、派生テーブルの結果を制限することのみが必要となります。同順の場合 (結果セットのロー 7 と 8)、RANK は同じ値を返します。したがって、690 と 949 の両方の営業担当者が最終結果に出現します。

参照

- 「SUM 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』

AVG 関数の例

次の例では、2000 年における月別の全製品売上の移動平均を計算するための Window 関数として AVG が使用されています。WINDOW 指定で RANGE 句が使用されているので、ROWS 句の場合のように隣接ローの数で計算されるのではなく、月の値を基にウィンドウ境界が計算されます。ある月に一部またはすべての製品の販売がなかった場合などは、ROWS を使用すると異なる結果が生成されます。

```
SELECT *
 FROM ( SELECT s.ProductID,
        Month( o.OrderDate ) AS julian_month,
        SUM( s.Quantity ) AS sales,
        AVG( SUM( s.Quantity ) )
        OVER ( PARTITION BY s.ProductID
          ORDER BY Month( o.OrderDate ) ASC
          RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING )
        AS average_sales
        FROM SalesOrderItems s KEY JOIN SalesOrders o
        WHERE Year( o.OrderDate ) = 2000
        GROUP BY s.ProductID, Month( o.OrderDate ) )
 AS DT
 ORDER BY 1,2;
```

参照

- 「AVG 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』

MAX 関数の例

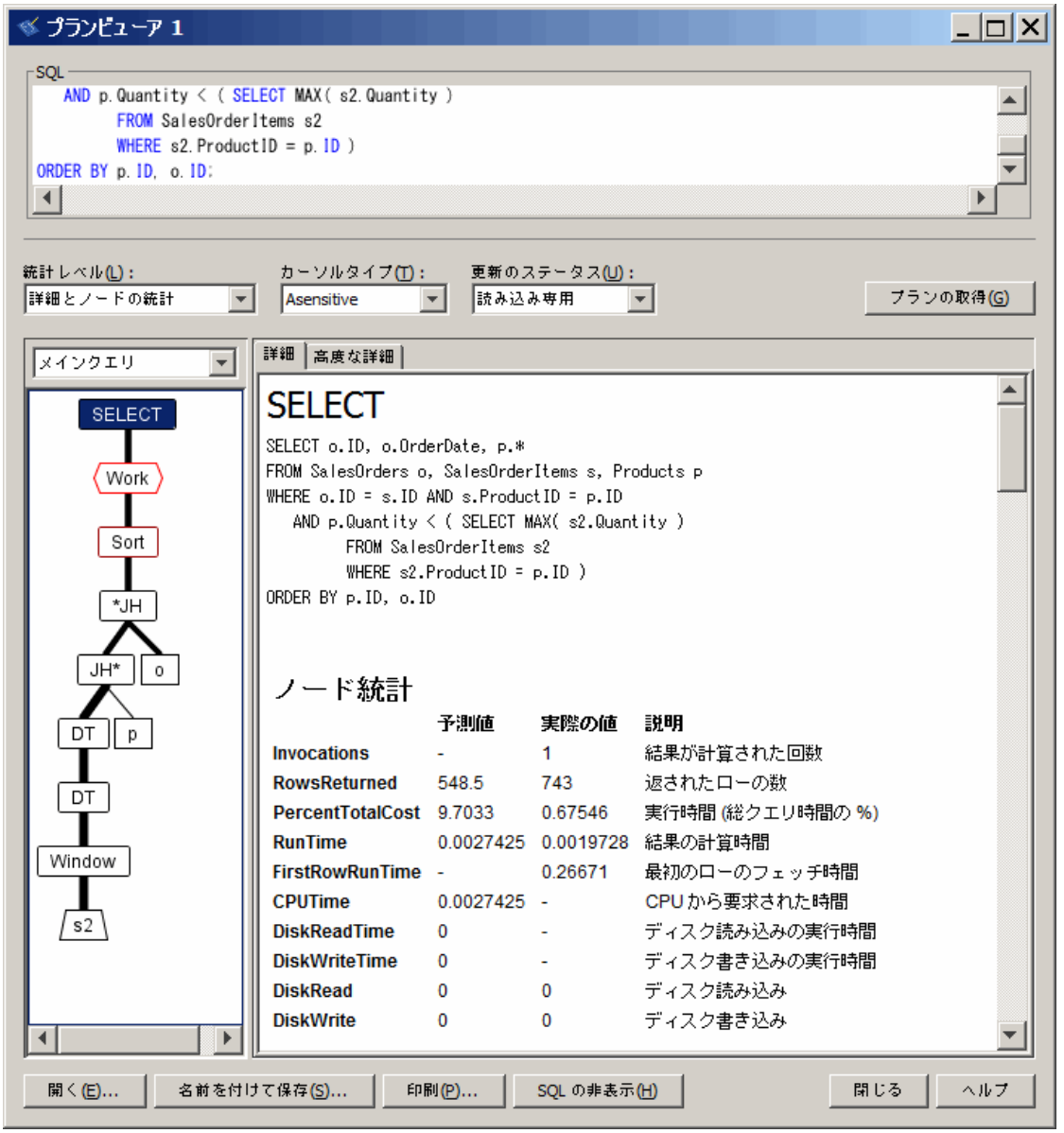
相関サブクエリの削除

場合によっては、特定のカラムの値を最大値や最小値と比較する必要があります。このようなクエリは、相関属性 (外部参照とも呼ばれる) のあるネストされたクエリとして作成されることが

よくあります。たとえば、次のクエリでは、その製品の注文 1 回あたりの最大数が製品の在庫数を超えているような製品について、すべての注文を製品情報とともにリストします。

```
SELECT o.ID, o.OrderDate, p.*
FROM SalesOrders o, SalesOrderItems s, Products p
WHERE o.ID = s.ID AND s.ProductID = p.ID
      AND p.Quantity < ( SELECT MAX( s2.Quantity )
                         FROM SalesOrderItems s2
                         WHERE s2.ProductID = p.ID )
ORDER BY p.ID, o.ID;
```

このクエリのグラフィカルなプランは、下の図のようにプランビューアに表示されます。このネストされたクエリが、クエリオプティマイザによってどのように変形されたかを確認してください。クエリは、Products テーブルと、Window 関数を含む派生テーブル (相関名 DT) のある SalesOrders テーブルとのジョインに変形されています。



オプティマイザに依存して相関サブクエリを派生テーブルによるジョインに変形するのではなく (セマンティック分析は複雑なので、この方法は単純な場合にしか使用できない)、Window 関数を使用して、このようなクエリを作成できます。

```
SELECT order_quantity.ID, o.OrderDate, p.*
 FROM ( SELECT s.ID, s.ProductID,
        MAX( s.Quantity ) OVER (
          PARTITION BY s.ProductID
          ORDER BY s.ProductID )
        AS max_quantity
        FROM SalesOrderItems s )
 AS order_quantity, Products p, SalesOrders o
```

```
WHERE p.ID = ProductID
  AND o.ID = order_quantity.ID
  AND p.Quantity < max_quantity
ORDER BY p.ID, o.ID;
```

参照

- 「MIN 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「MAX 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』

FIRST_VALUE 関数と LAST_VALUE 関数の例

FIRST_VALUE 関数と LAST_VALUE 関数は、ウィンドウの最初と最後のローの値を返します。これらの関数を使用すると、セルフジョインを使わずにクエリで複数のローの値に一度にアクセスできます。

この 2 つの関数は、ウィンドウに使用する必要があるので、他の Window 集合関数とは異なります。また、これらの関数では IGNORE NULLS 句を使用できる点も、他の Window 集合関数と異なります。IGNORE NULLS を指定すると、式の最初または最後の NULL 以外の値が返されます。指定しなければ、最初または最後の値が返されます。

例 1：グループの最初のエントリ

FIRST_VALUE 関数を使用して、一定の順序で並んでいる値グループの最初のエントリを取り出すことができます。次のクエリは、各注文について、注文の最初の品目の ProductID、つまり各注文で LineID が最も小さい品目の ProductID を返します。

クエリでは、DISTINCT キーワードを使用して重複を削除しています。このキーワードを指定しなかった場合、各注文の各品目について重複するローが返されます。

```
SELECT DISTINCT ID,
FIRST_VALUE( ProductID ) OVER ( PARTITION BY ID ORDER BY LineID )
FROM SalesOrderItems
ORDER BY ID;
```

例 2：最大売り上げに対する割合

FIRST_VALUE 関数の一般的な使用方法として、各ローの値を、現在のグループ内の最大値または最小値と比較できます。次のクエリは、各営業担当者の総売り上げを計算してから、その総売り上げと、同じ製品の最大総売り上げを比較します。結果は、最大総売り上げのパーセントで表されます。

```
SELECT s.ProductID AS prod_id, o.SalesRepresentative AS sales_rep,
   SUM( s.Quantity * p.UnitPrice ) AS total_sales,
   100 * total_sales / ( FIRST_VALUE( SUM( s.Quantity * p.UnitPrice ) )
                    OVER Sales_Window ) AS total_sales_percentage
FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
GROUP BY o.SalesRepresentative, s.ProductID
  WINDOW Sales_Window AS ( PARTITION BY s.ProductID
                      ORDER BY SUM( s.Quantity * p.UnitPrice ) DESC )
ORDER BY s.ProductID;
```

例 3 : NULL 値を設定することによるデータの高密度化

FIRST_VALUE 関数と LAST_VALUE 関数は、データの密度を高めた後で、NULL の代わりに値を設定したい場合に便利です。たとえば、1 日の総売り上げが最も高い営業担当者が表彰されるとします。次のクエリは、2001 年 4 月の第 1 週にトップ成績を上げた担当者を表示します。

```
SELECT v.OrderDate, v.SalesRepresentative AS rep_of_the_day
FROM ( SELECT o.SalesRepresentative, o.OrderDate,
           RANK() OVER ( PARTITION BY o.OrderDate
                         ORDER BY SUM( s.Quantity *
                          p.UnitPrice ) DESC ) AS sales_ranking
       FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
       GROUP BY o.SalesRepresentative, o.OrderDate ) AS v
WHERE v.sales_ranking = 1
AND v.OrderDate BETWEEN '2001-04-01' AND '2001-04-07'
ORDER BY v.OrderDate;
```

このクエリは、次の結果を返します。

OrderDate	rep_of_the_day
2001-04-01	949
2001-04-02	856
2001-04-05	902
2001-04-06	467
2001-04-07	299

ただし、売り上げがなかった日については結果は返されません。次のクエリは、データの密度を高め、売り上げがなかった日のレコードも作成されるようにします。また、LAST_VALUE 関数を使用して、表彰がなかった日には、成績順位に入れ替わりがあるまで最後にトップ成績を獲得した者の ID を NULL 値の代わりに rep_of_the_day に表示させています。

```
SELECT d.dense_order_date,
             LAST_VALUE( v.SalesRepresentative IGNORE NULLS )
                OVER ( ORDER BY d.dense_order_date )
                AS rep_of_the_day
FROM ( SELECT o.SalesRepresentative, o.OrderDate,
           RANK() OVER ( PARTITION BY o.OrderDate
                         ORDER BY SUM( s.Quantity *
                          p.UnitPrice ) DESC ) AS sales_ranking
       FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
       GROUP BY o.SalesRepresentative, o.OrderDate ) AS v
RIGHT OUTER JOIN ( SELECT DATEADD( day, row_num, '2001-04-01' )
                    AS dense_order_date
                 FROM sa_rowgenerator( 0, 6 )) AS d
ON v.OrderDate = d.dense_order_date AND sales_ranking = 1
ORDER BY d.dense_order_date;
```

このクエリは、次の結果を返します。

dense_order_date	rep_of_the_day
2001-04-01	949
2001-04-02	856
2001-04-03	856
2001-04-04	856
2001-04-05	902
2001-04-06	467
2001-04-07	299

前のクエリからの派生テーブル v を、対象日をすべて含む派生テーブル d にジョインすると、1 日ごとにローができます。ただし、この外部ジョインでは、売り上げがなかった日の SalesRepresentative カラムには NULL が含まれます。LAST_VALUE 関数を使用すると、この問題を解決できます。特定のローの rep_of_the_day を、対応する日までの SalesRepresentative の最後の NULL 以外の値と定義します。

参照

- 「FIRST_VALUE 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「LAST_VALUE 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「Window 関数」558 ページ

標準偏差関数と平方偏差関数

平方偏差関数と標準偏差関数について、標本バージョンと母集団バージョンという 2 つのバージョンがサポートされています。どちらのバージョンを選択するかは、その関数が使用される統計上のコンテキストによって変わります。

すべての平方偏差関数と標準偏差関数は、クエリの GROUP BY 句で決定されるローの分割について値を計算できるという点で、真の集合関数であるといえます。MAX や MIN などのその他の基本集合関数と同様に、入力の NULL 値は無視されます。

パフォーマンスを向上させるために、データベースサーバは平均と平均からの偏差を同時に計算します。そのためデータを 1 パスするだけですみます。

また、分析対象の式のドメインに関係なく、すべての平方偏差と標準偏差は IEEE 倍精度浮動小数点を使用して計算されます。平方偏差関数や標準偏差関数の入力が空のセットである場合、各関数は結果で NULL を返します。単一ローに対して VAR_SAMP が計算されると NULL が返されます。VAR_POP の場合は値 0 が返されます。

サポートされる標準偏差関数と平方偏差関数を次に示します。

- STDDEV 関数
- STDDEV_POP 関数
- STDDEV_SAMP 関数
- VARIANCE 関数
- VAR_POP 関数
- VAR_SAMP 関数

STDDEV 関数

この関数は、STDDEV_SAMP 関数のエイリアスです。

STDDEV_POP 関数

この関数は、数値式からなる母集団の標準偏差を DOUBLE として計算します。

例 1

次のクエリは、部門の平均給与に標準偏差を加えたものよりも多い給与を得ている従業員を示す結果セットを返します。標準偏差は、データがどれだけ平均からばらつきがあるかを計るものです。

```
SELECT *
FROM ( SELECT
   Surname AS Employee,
   DepartmentID AS Department,
   CAST( Salary as DECIMAL( 10, 2 ) )
      AS Salary,
   CAST( AVG( Salary )
      OVER ( PARTITION BY DepartmentID ) AS DECIMAL ( 10, 2 ) )
      AS Average,
   CAST( STDDEV_POP( Salary )
      OVER ( PARTITION BY DepartmentID ) AS DECIMAL ( 10, 2 ) )
      AS StandardDeviation
   FROM Employees
   GROUP BY Department, Employee, Salary )
   AS DerivedTable
WHERE Salary > Average + StandardDeviation
ORDER BY Department, Salary, Employee;
```

次のテーブルは、クエリからの結果セットを示します。すべての部門で、少なくとも 1 人は平均から著しく外れた従業員がいます。

	Employee	Department	Salary	Average	StandardDeviation
1	Lull	100	87900.00	58736.28	16829.60
2	Scheffield	100	87900.00	58736.28	16829.60
3	Scott	100	96300.00	58736.28	16829.60
4	Sterling	200	64900.00	48390.95	13869.60

	Employee	Department	Salary	Average	StandardDeviation
5	Savarino	200	72300.00	48390.95	13869.60
6	Kelly	200	87500.00	48390.95	13869.60
7	Shea	300	138948.00	59500.00	30752.40
8	Blaikie	400	54900.00	43640.67	11194.02
9	Morris	400	61300.00	43640.67	11194.02
10	Evans	400	68940.00	43640.67	11194.02
11	Martinez	500	55500.00	33752.20	9084.50

従業員 Scott は 96,300.00 ドルを得ていますが、部門の平均給与は 58,736.28 ドルです。この部門の標準偏差は 16,829.00 ドルです。つまり、平均給与以上でかつ 75,565.88 ドル (58736.28 + 16829.60 = 75565.88) に満たない給与は、平均から標準偏差内にあるということになります。従業員 Scott の給与は 96,300.00 ドルで、この値を超えています。

この例では、Surname と Salary が従業員ごとにユニークであることを想定していますが、ユニークである必要はありません。ユニーク性を保証するには、EmployeeID を GROUP BY 句に追加します。

例 2

次の文は、異なる期間における注文ごとの項目数で平均と平方偏差をリストします。

```
SELECT YEAR( ShipDate ) AS Year,
   QUARTER( ShipDate ) AS Quarter,
   AVG( Quantity ) AS Average,
   STDDEV_POP( Quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

このクエリは、次の結果を返します。

Year	Quarter	Average	Variance
2000	1	25.775148	14.2794...
2000	2	27.050847	15.0270...
...	...	...	...

STDDEV_SAMP 関数

この関数は、数値式からなるサンプルの標準偏差を DOUBLE として計算します。たとえば、次の文は、異なる四半期における注文ごとの項目数で平均と平方偏差を返します。

```
SELECT YEAR( ShipDate ) AS Year,
   QUARTER( ShipDate ) AS Quarter,
   AVG( Quantity ) AS Average,
   STDDEV_SAMP( Quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

このクエリは、次の結果を返します。

Year	Quarter	Average	Variance
2000	1	25.775148	14.3218...
2000	2	27.050847	15.0696...
...	...	...	...

VARIANCE 関数

この関数は、VAR_SAMP 関数のエイリアスです。

VAR_POP 関数

この関数は、数値式からなる母集団の統計上の平方偏差を DOUBLE として計算します。たとえば、次の文は、異なる期間における注文ごとの項目数で平均と平方偏差をリストします。

```
SELECT YEAR( ShipDate ) AS Year,
   QUARTER( ShipDate ) AS Quarter,
   AVG( Quantity ) AS Average,
   VAR_POP( quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

このクエリは、次の結果を返します。

Year	Quarter	Average	Variance
2000	1	25.775148	203.9021...
2000	2	27.050847	225.8109...
...	...	...	...

単一ローに対して VAR_POP が計算されると値 0 が返されます。

VAR_SAMP 関数

この関数は、数値式からなるサンプルの統計上の平方偏差を DOUBLE として計算します。

たとえば、次の文は、異なる期間における注文ごとの項目数で平均と平方偏差をリストします。

```
SELECT YEAR( ShipDate ) AS Year,
   QUARTER( ShipDate ) AS Quarter,
```

```
    AVG( Quantity ) AS Average,
    VAR_SAMP( Quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

このクエリは、次の結果を返します。

Year	Quarter	Average	Variance
2000	1	25.775148	205.1158...
2000	2	27.050847	227.0939...
...	...	...	...

単一ローに対して VAR_SAMP が計算されると値 NULL が返されます。

参照

- 「STDDEV_SAMP 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「STDDEV_POP 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「VAR_SAMP 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「VAR_POP 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「VAR_SAMP 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「集合関数に対応する数式」593 ページ

相関関数と線形回帰関数

さまざまな統計関数がサポートされています。関数の結果は、線形回帰の質を分析するのに役立ちます。

各関数の最初の引数は従属式 (Y で示される)、2 番目の引数は独立式 (X で示される) です。

- **COVAR_SAMP 関数**　COVAR_SAMP 関数は、(Y, X) ペアのセットの標本共平方偏差を返します。
- **COVAR_POP 関数**　COVAR_POP 関数は、(Y, X) ペアのセットの母共平方偏差を返します。
- **CORR 関数**　CORR 関数は、(Y, X) ペアのセットの相関係数を返します。
- **REGR_AVGX 関数**　REGR_AVGX 関数は、(Y, X) 値の NULL 以外のすべてのペアから x 値の平均を返します。
- **REGR_AVGY 関数**　REGR_AVGY 関数は、(Y, X) 値の NULL 以外のすべてのペアから y 値の平均を返します。
- **REGR_SLOPE 関数**　REGR_SLOPE 関数は、NULL 以外のペアに調整された線形回帰直線の傾きを計算します。

- **REGR_INTERCEPT 関数**　REGR_INTERCEPT 関数は、従属変数と独立変数に最も適切な線形回帰直線の y 切片を計算します。

- **REGR_R2 関数**　REGR_R2 関数は、回帰直線の決定係数 (「R-squared」 または「適合度」とも呼ぶ) を計算します。

- **REGR_COUNT 関数**　REGR_COUNT 関数は、入力で (Y, X) 値の NULL 以外のペアの数を返します。当該ペアの X と Y の両方が NULL 以外である場合にかぎり、線形回帰の計算で観測が使用されます。

- **REGR_SXX 関数**　この関数は、(Y, X) ペアの x 値のセットの平方和を返します。

  この関数の式は、標本平方偏差式や母平方偏差式の分子に相当します。その他の線形回帰関数と同様に、REGR_SXX は、入力で X と Y のどちらかが NULL であるような (Y, X) 値のペアを無視します。

- **REGR_SYY 関数**　この関数は、(Y, X) ペアの Y 値のセットの平方和を返します。

- **REGR_SXY 関数**　この関数は、(Y, X) ペアのセットに対して 2 つの積和の差を返します。

参照

- 「COVAR_SAMP 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「COVAR_POP 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「CORR 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「REGR_AVGX 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「REGR_AVGY 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「REGR_SLOPE 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「REGR_R2 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「REGR_COUNT 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「REGR_SXX 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「REGR_SYY 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』
- 「REGR_SXY 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』

Window ランキング関数

Window ランキング関数は、分割内の他のローに関連するローのランクを返します。サポートされているランキング関数は次のとおりです。

- CUME_DIST
- DENSE_RANK
- PERCENT_RANK
- RANK

ランキング関数は、SUM 集合関数などと同様の方法で複数の入力ローからの結果を計算しないため、集合関数とは見なされません。これらの関数は、特定の式の値に基づいて、分割内のローのランク (相対的な順序) を計算します。分割内のローの各セットは個別にランク付けされます。そのため OVER 句に PARTITION BY 句が含まれない場合は、入力全体が単一の分割として扱われます。このため、ランキング関数で使用されるウィンドウに対して、ROWS 句または

RANGE 句は指定できません。複数のランキング関数を含むクエリを作成し、それぞれの関数が入力ローを異なる状態に分割またはソートするようにできます。

すべてのランキング関数では、ランキング関数が依存する入力ローのソート順序を指定するために ORDER BY 句が必要です。ORDER BY 句に複数の式が含まれる場合は、最初の式によって隣接ローで同じ値になるときに同順の発生を避けるために、2 番目以降の式が使用されます。NULL 値はその他の値よりも前にソートされます (昇順の場合)。

RANK 関数

RANK 関数は、その他のローの値と比較した現在のローの値のランクを返します。値のランクは、値のリストがソートされた場合の順序を反映しています。

RANK 関数を使用すると、ウィンドウの ORDER BY 句で指定された式でランクが計算されます。ORDER BY 句に複数の式が含まれる場合は、最初の式によって隣接ローで同じ値になるときに同順の発生を避けるために、2 番目以降の式が使用されます。NULL 値はその他の値よりも前にソートされます (昇順の場合)。

例 1

次のクエリは、データベースで最もコストが高い製品 3 つを特定します。ウィンドウでは降順のソート順序が指定されるため、最もコストの高い製品はランクが最も低くなります。つまり、ランク付けは 1 から開始します。

```
SELECT Top 3 *
    FROM ( SELECT Description, Quantity, UnitPrice,
        RANK() OVER ( ORDER BY UnitPrice DESC ) AS Rank
        FROM Products ) AS DT
ORDER BY Rank;
```

このクエリは、次の結果を返します。

	Description	Quantity	UnitPrice	Rank
1	Zipped Sweatshirt	32	24.00	1
2	Hooded Sweatshirt	39	24.00	1
3	Cotton Shorts	80	15.00	3

ロー 1 と 2 は、UnitPrice の値が同じであるため、ランクも同じになります。これを「同順」と呼びます。

RANK 関数では、同順の後はランクの値がジャンプします。たとえば、ロー 3 のランク値は 2 ではなく 3 にジャンプします。この動作は、同順の後でジャンプが発生しない DENSE_RANK 関数と異なります。

例 2

次の SQL クエリは、ユタ州の男性および女性従業員を検索し、給与が多い順にランクします。

```
SELECT Surname, Salary, Sex,
    RANK() OVER ( ORDER BY Salary DESC ) "Rank"
    FROM Employees
WHERE State IN ( 'UT' );
```

次のテーブルは、クエリからの結果セットを示します。

	Surname	Salary	Sex	Rank
1	Shishov	72995.00	F	1
2	Wang	68400.00	M	2
3	Cobb	62000.00	M	3
4	Morris	61300.00	M	4
5	Diaz	54900.00	M	5
6	Driscoll	48023.69	M	6
7	Hildebrand	45829.00	F	7
8	Goggin	37900.00	M	8
9	Rebeiro	34576.00	M	9
10	Bigelow	31200.00	F	10
11	Lynch	24903.00	M	11

例 3

データを分割して異なる結果になるように計算できます。例 2 のクエリを使用して、性別で分割することでデータを変更できます。次の例は、従業員を性別ごとに給与の多い順でランクしたものです。

```
SELECT Surname, Salary, Sex,
    RANK ( ) OVER ( PARTITION BY Sex
    ORDER BY Salary DESC ) "Rank"
    FROM Employees
WHERE State IN ( 'UT' );
```

次のテーブルは、クエリからの結果セットを示します。

	Surname	Salary	Sex	Rank
1	Wang	68400.00	M	1
2	Cobb	62000.00	M	2
3	Morris	61300.00	M	3

	Surname	Salary	Sex	Rank
4	Diaz	54900.00	M	4
5	Driscoll	48023.69	M	5
6	Goggin	37900.00	M	6
7	Rebeiro	34576.00	M	7
8	Lynch	24903.00	M	8
9	Shishov	72995.00	F	1
10	Hildebrand	45829.00	F	2
11	Bigelow	31200.00	F	3

参照

- 「DENSE_RANK 関数」587 ページ
- 「RANK 関数 [ランキング]」『SQL Anywhere サーバ SQL リファレンス』

DENSE_RANK 関数

DENSE_RANK 関数は、RANK 関数と同様に、その他のローの値と比較した現在のローの値のランクを返します。値のランクは、値のリストがソートされた場合の順序を反映しています。ランクは、ウィンドウの ORDER BY 句で指定された式で計算されます。

DENSE_RANK 関数は、ランク値にギャップ (ジャンプ) がなく単調に増加し続ける一連のランクを返します。RANK 値とは異なり、ランク値にジャンプがないことから DENSE (密) という語が使われます。

ウィンドウが入力ローを移動するのに合わせて、ウィンドウの ORDER BY 句で指定された式でランクが計算されます。ORDER BY 句に複数の式が含まれる場合は、最初の式によって隣接ローで同じ値になるときに同順の発生を避けるために、2 番目以降の式が使用されます。NULL 値はその他の値よりも前にソートされます (昇順の場合)。

例 1

次のクエリは、データベースで最もコストが高い製品 3 つを特定します。ウィンドウでは降順のソート順序が指定されるため、最もコストの高い製品はランクが最も低くなります。つまり、ランク付けは 1 から開始します。

```
SELECT Top 3 *
  FROM ( SELECT Description, Quantity, UnitPrice,
    DENSE_RANK( ) OVER ( ORDER BY UnitPrice DESC ) AS Rank
    FROM Products ) AS DT
  ORDER BY Rank;
```

このクエリは、次の結果を返します。

	Description	Quantity	UnitPrice	Rank
1	Hooded Sweatshirt	39	24.00	1
2	Zipped Sweatshirt	32	24.00	1
3	Cotton Shorts	80	15.00	2

ロー 1 と 2 は、UnitPrice の値が同じであるため、ランクも同じになります。これを「同順」と呼びます。

DENSE_RANK 関数を使用した場合は、同順の後のランク値はジャンプしません。たとえば、ロー 3 のランク値は 2 です。この動作は、同順の後でランク値がジャンプする RANK 関数と異なります。

例 2

ウィンドウはクエリの GROUP BY 句の後に評価されるため、集合関数の値を基にランキングを判断するような複雑な要求を指定できます。

次のクエリは、地域ごとに総売り上げ上位 3 人の営業担当者と、地域ごとの総売り上げを生成します。

```
SELECT *
  FROM ( SELECT o.SalesRepresentative, o.Region,
           SUM( s.Quantity * p.UnitPrice ) AS total_sales,
           DENSE_RANK( ) OVER ( PARTITION BY o.Region,
             GROUPING( o.SalesRepresentative )
             ORDER BY total_sales DESC ) AS sales_rank
         FROM Products p, SalesOrderItems s, SalesOrders o
         WHERE p.ID = s.ProductID AND s.ID = o.ID
         GROUP BY GROUPING SETS( ( o.SalesRepresentative, o.Region ),
           o.Region ) ) AS DT
  WHERE sales_rank <= 3
  ORDER BY Region, sales_rank;
```

このクエリは、次の結果を返します。

	SalesRepresentative	Region	total_sales	sales_rank
1	299	Canada	9312.00	1
2	(NULL)	Canada	24768.00	1
3	1596	Canada	3564.00	2
4	856	Canada	2724.00	3
5	299	Central	32592.00	1

	SalesRepresentative	Region	total_sales	sales_rank
6	(NULL)	Central	134568.00	1
7	856	Central	14652.00	2
8	467	Central	14352.00	3
9	299	Eastern	21678.00	1
10	(NULL)	Eastern	142038.00	1
11	902	Eastern	15096.00	2
12	690	Eastern	14808.00	3
13	1142	South	6912.00	1
14	(NULL)	South	45262.00	1
15	667	South	6480.00	2
16	949	South	5782.00	3
17	299	Western	5640.00	1
18	(NULL)	Western	37632.00	1
19	1596	Western	5076.00	2
20	667	Western	4068.00	3

このクエリは、GROUPING SETS を使用して複数のグループ化を結合します。そのため、ウィンドウの WINDOW PARTITION 句では GROUPING 関数を使用して、特定の営業担当者を表すディテールローと、地域全体の総売り上げをリストする小計ローとを区別します。地域ごとの小計のローは、SalesRepresentative 属性に値 NULL がありますが、入力の分割ごとに結果のランキング順序が付けられるため、それぞれの小計ローのランキング値は 1 になります。これにより、ディテールローは 1 から適切にランク付けされます。

この例では、DENSE_RANK 関数により、総売り上げの集約について入力がランク付けされています。WINDOW ORDER 句では、エイリアスの設定された SELECT リスト項目が省略形として使用されます。

参照

- 「RANK 関数」585 ページ
- 「DENSE_RANK 関数 [ランキング]」『SQL Anywhere サーバ SQL リファレンス』

CUME_DIST 関数

累積分布関数 CUME_DIST は、百分位数の逆数として定義される場合があります。CUME_DIST は、ウィンドウ内の値のセットに関して、特定値の正規化された位置を計算します。関数の範囲は 0 ～ 1 です。

ウィンドウが入力ローを移動するのに合わせて、ウィンドウの ORDER BY 句で指定された式で累積分布が計算されます。ORDER BY 句に複数の式が含まれる場合は、最初の式によって隣接ローで同じ値になるときに同順の発生を避けるために、2 番目以降の式が使用されます。NULL 値はその他の値よりも前にソートされます (昇順の場合)。

次の例は、カリフォルニアに住む従業員の給与に関する累積分布を示す結果セットを返します。

```
SELECT DepartmentID, Surname, Salary,
   CUME_DIST( ) OVER ( PARTITION BY DepartmentID
     ORDER BY Salary DESC ) "Rank"
  FROM Employees
  WHERE State IN ( 'CA' );
```

このクエリは、次の結果を返します。

DepartmentID	Surname	Salary	Rank
200	Savarino	72300.00	0.333333333333333
200	Clark	45000.00	0.666666666666667
200	Overbey	39300.00	1

参照

- 「CUME_DIST 関数 [ランキング]」『SQL Anywhere サーバ SQL リファレンス』

PERCENT_RANK 関数

PERCENT 関数と同様に、PERCENT_RANK 関数はウィンドウの ORDER BY 句で指定されたカラムの値についてランクを返します。ただし、ランクは 0 ～ 1 の小数として表され、(RANK - 1)/(- 1) として計算されます。

ウィンドウが入力ローを移動するのに合わせて、ウィンドウの ORDER BY 句で指定された式でランクが計算されます。ORDER BY 句に複数の式が含まれる場合は、最初の式によって隣接ローで同じ値になるときに同順の発生を避けるために、2 番目以降の式が使用されます。NULL 値はその他の値よりも前にソートされます (昇順の場合)。

例 1

次の例は、ニューヨークの従業員の給与ランキングを性別ごとに示す結果セットを返します。結果セットは、小数のパーセンテージを使用して降順にランキングされ、性別ごとに分けられます。

```
SELECT DepartmentID, Surname, Salary, Sex,
   PERCENT_RANK( ) OVER ( PARTITION BY Sex
     ORDER BY Salary DESC ) AS PctRank
```

```
FROM Employees
WHERE State IN ( 'NY' );
```

このクエリは、次の結果を返します。

	DepartmentID	Surname	Salary	Sex	PctRank
1	200	Martel	55700.000	M	0.0
2	100	Guevara	42998.000	M	0.333333333
3	100	Soo	39075.000	M	0.666666667
4	400	Ahmed	34992.000	M	1.0
5	300	Davidson	57090.000	F	0.0
6	400	Blaikie	54900.000	F	0.333333333
7	100	Whitney	45700.000	F	0.666666667
8	400	Wetherby	35745.000	F	1.0

入力は性別 (Sex) で分割されるため、PERCENT_RANK は男性と女性で個別に評価されます。

例 2

次の例は、ユタ州とアリゾナ州の女性従業員のリストを返し、給与の多い順にランクしたものです。PERCENT_RANK 関数は、降順で累積合計を計算するのに使用します。

```
SELECT Surname, Salary,
    PERCENT_RANK ( ) OVER ( ORDER BY Salary DESC ) "Rank"
    FROM Employees
WHERE State IN ( 'UT', 'AZ' ) AND Sex IN ( 'F' );
```

このクエリは、次の結果を返します。

	Surname	Salary	Rank
1	Shishov	72995.00	0
2	Jordan	51432.00	0.25
3	Hildebrand	45829.00	0.5
4	Bigelow	31200.00	0.75
5	Bertrand	29800.00	1

PERCENT_RANK を使用した最上位と最下位の百分位数の検索

PERCENT_RANK 関数を使用して、データセット内の最上位または最下位の百分位数を検索できます。次の例では、クエリは給与額についてデータセット内で上位 5 % の男性従業員を返します。

```
SELECT *
FROM ( SELECT Surname, Salary,
      PERCENT_RANK ( ) OVER ( ORDER BY Salary DESC ) "Rank"
      FROM Employees
      WHERE Sex IN ( 'M' ) )
      AS DerivedTable ( Surname, Salary, Percent )
WHERE Percent < 0.05;
```

このクエリは、次の結果を返します。

	Surname	Salary	Percent
1	Scott	96300.00	0
2	Sheffield	87900.00	0.025
3	Lull	87900.00	0.025

参照

- 「PERCENT_RANK 関数 [ランキング]」『SQL Anywhere サーバ SQL リファレンス』

ロー番号付け関数

ロー番号付け関数は、分割内のローにユニークな番号を付けます。NUMBER と ROW_NUMBER の 2 つのロー番号付け関数がサポートされています。ROW_NUMBER 関数は、ANSI 標準準拠の関数で、NUMBER(*) 関数と同等の機能の多くを使用できるため、この関数の使用をおすすめします。どちらの関数も同様のタスクを実行しますが、NUMBER 関数には、ROW_NUMBER 関数にはない制限がいくつかあります。

参照

- 「NUMBER 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「ROW_NUMBER 関数」592 ページ

ROW_NUMBER 関数

ROW_NUMBER 関数は、結果のローにユニークな番号を付けます。この関数はランキング関数ではありませんが、ランキング関数を使用できるどのような状況でも使うことができ、動作はランキング関数に似ています。

たとえば、派生テーブルで ROW_NUMBER を使用して、ROW_NUMBER の値について、ジョインであっても制限を追加できます。

```
SELECT *
FROM ( SELECT Description, Quantity,
```

```
    ROW_NUMBER( ) OVER ( ORDER BY ID ASC ) AS RowNum
FROM Products ) AS DT
WHERE RowNum <= 3
ORDER BY RowNum;
```

このクエリは、次の結果を返します。

Description	Quantity	RowNum
Tank Top	28	1
V-neck	54	2
Crew Neck	75	3

ランキング関数の場合と同様に、ROW_NUMBER には ORDER BY 句が必要です。

ウィンドウの ORDER BY 句がユニークでない式で構成される場合は、ROW_NUMBER は非決定的な結果を返すことがあり、同順が発生したときのローの順序は予測できなくなります。

ROW_NUMBER は、分割全体に対して機能するように設計されているため、ROWS 句や RANGE 句を ROW_NUMBER 関数とともに指定することはできません。

集合関数に対応する数式

参考情報として、Window 集合関数について、等価の数式を次の 2 つの表に示します。

単純な集合関数

Function	Symbol	Formula
SUM(X)		$\sum_{i=1}^{n} x_i$
MAX(X)		$x_i : x_i \geq x_j, i \neq j \; \forall i,j \in n$
MIN(X)		$x_i : x_i \leq x_j, i \neq j \; \forall i,j \in n$
AVG(X)	$\bar{x}$	$\frac{\sum x_i}{n}$
COUNT(*)		n
VAR_SAMP(X)	s_x^2	$\frac{\sum (x_i - \bar{x})^2}{(n-1)}$
VAR_POP(X)	σ_x^2	$\frac{\sum (x_i - \bar{x})^2}{n}$
VARIANCE(X)		identical to VAR_SAMP(X)
STDDEV_SAMP(X)	s_x	$\sqrt{\frac{\sum (x_i - \bar{x})^2}{(n-1)}}$
STDDEV_POP(X)	σ_x	$\sqrt{\frac{\sum (x_i - \bar{x})^2}{n}}$
STDDEV(X)		identical to STDDEV_SAMP(X)

統計集合関数

COVAR_SAMP(Y,X)	*Co-variance*	$s_{xy} = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{(n-1)}$
COVAR_POP(Y,X)	*Co-variance*	$\sigma_{xy} = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{n}$
CORR(Y,X)	*Correlation Coefficient*	$r = \frac{\sum xy - \frac{1}{n}(\sum x)(\sum y)}{(n-1)s_x s_y}$
REGR_AVGX(Y,X)	*Independent mean*	$\bar{x}$
REGR_AVGY(Y,X)	*Dependent mean*	$\bar{y}$
REGR_SLOPE(Y,X)	*Regression Slope*	$b = r\frac{s_y}{s_x}$
REGR_INTERCEPT(Y,X)	*Regression Intercept*	$a = \bar{y} - b\bar{x}$
REGR_R2(Y,X)	*'Goodness-of-fit'*	r^2
REGR_COUNT(Y,X)	*Sample size*	n (non-null (Y, X) pairs)
REGR_SXX(Y,X)	*Sum of squares* (x)	$\sum x^2 - \frac{(\sum x)^2}{n}$
REGR_SYY(Y,X)	*Sum of squares* (y)	$\sum y^2 - \frac{(\sum y)^2}{n}$
REGR_SXY(Y,X)	*Sum of products*	$\sum xy - \frac{(\sum y)(\sum x)}{n}$

サブクエリの使用

リレーショナルデータベースを使用すると、複数のテーブルに関連するデータを保存できます。ジョインを使用して関連するテーブルからデータを抽出できるほか、「サブクエリ」を使用しても抽出できます。サブクエリとは、親の SQL 文の SELECT 句、WHERE 句、HAVING 句内にネストされた SELECT 文です。

サブクエリを使用すると、一部のクエリをジョインより簡単に記述できます。また、サブクエリを使用しないと記述できないクエリがあります。

サブクエリは、次のようなさまざまな方法に分類されます。

●1 つ以上のローを返すかどうか (単一ローと複数ローのサブクエリ)

●相関サブクエリか、非相関サブクエリか

●別のサブクエリ内でネストしているかどうか

単一ローのサブクエリと複数ローのサブクエリ

外部の文に 1 つまたは 0 個のローを返すサブクエリを、「単一ローのサブクエリ」と呼びます。比較演算子の有無にかかわらず、単一ローのサブクエリは SQL 文のどこでも使用できます。

たとえば、単一ローのサブクエリは、SELECT 句の式で使用できます。

```
SELECT (select FIRST T.x FROM T) + 1 as ITEM_1, 2 as ITEM_2,...
```

また、単一ローのサブクエリは、比較演算子とともに SELECT 句の式で使用できます。

例：

```
SELECT IF (select FIRST T.x FROM T) >= 10 THEN 1 ELSE 0 ENDIF as ITEM_1, 2 as ITEM_2,...
```

外部の文に複数のロー (ただしカラムは 1 つだけ) を返すサブクエリを、「複数ローのサブクエリ」と呼びます。複数ローのサブクエリは、IN 句、ANY 句、ALL 句または EXISTS 句で使用されるサブクエリです。

例 1：単一ローのサブクエリ

テーブル Products に製品だけの情報を、別のテーブル SalesOrdersItems には注文関連の情報を保存します。Products テーブルにはさまざまな製品の情報が入っています。SalesOrdersItems テーブルには、顧客の注文についての情報が入っています。在庫数が 50 未満になったときに製品を再注文する場合、次のクエリで「在庫数が少ない製品は何か」という問い合わせに対する回答を得ることができます。

```
SELECT ID, Name, Description, Quantity
FROM Products
WHERE Quantity < 50;
```

ただし、ほとんど注文されない製品がわずかしかないことよりも頻繁に購入される製品が少ないことの方が関心が高いため、製品が注文される頻度を考慮すると便利です。

サブクエリを使用してある顧客が注文する品目の平均数を決定し、その平均をメインクエリに使用して在庫数が少ない製品を検索します。次に示すクエリは、顧客が注文した各タイプの平均品目数の 2 倍未満の製品名とその説明を検索します。

```
SELECT Name, Description
FROM Products WHERE Quantity <  2 * (
  SELECT AVG( Quantity )
  FROM SalesOrderItems
  );
```

WHERE 句では、クエリ結果に含まれる、FROM 句にリストされるテーブルからローを選択するのにサブクエリが役立ちます。HAVING 句では、クエリ結果に含まれる、メインクエリの GROUP BY 句で指定されるローのグループを選択するのに役立ちます。

例 2：単一ローのサブクエリ

次の単一ローのサブクエリの例では、Products テーブルの製品の平均価格を計算します。そしてその平均は外部クエリの WHERE 句に渡されます。外部クエリは、平均より低い価格のすべての製品の ID、Name、UnitPrice を返します。

```
SELECT ID, Name, UnitPrice
FROM Products
WHERE UnitPrice <
 ( SELECT AVG( UnitPrice ) FROM Products )
ORDER BY UnitPrice DESC;
```

ID	Name	UnitPrice
401	Baseball Cap	10.00
300	Tee Shirt	9.00
400	Baseball Cap	9.00
500	Visor	7.00
501	Visor	7.00

例 3 : IN を使用した単純な複数ローのサブクエリ

在庫数が少ない品目を識別し、一方でそれらの品目に対する注文も識別したいとします。次のように、WHERE 句にサブクエリを含む SELECT 文を実行します。

```
SELECT *
FROM SalesOrderItems
WHERE ProductID IN
  ( SELECT ID
    FROM Products
    WHERE Quantity < 20 )
ORDER BY ShipDate DESC;
```

この例では、サブクエリは Products テーブル内の ID カラムにおいて WHERE 句の探索条件を満たすすべての値のリストを作成します。そして一連のローが返されますが、返されるカラムは 1 つだけです。IN キーワードは、それぞれの値をセットのメンバーとして扱い、メインクエリ内の各ローがセットのメンバーかどうかをテストします。

例 4 : IN、ANY、ALL の使用を比較する複数ローのサブクエリ

SQL Anywhere サンプルデータベースには、経理に関するデータを格納するテーブルが 2 つあります。FinancialCodes テーブルは、経理データとこれらの意味についてのさまざまなコードが入っているテーブルです。FinancialData テーブルから歳入項目をリストするには、次のクエリを実行します。

```
SELECT *
FROM FinancialData
WHERE Code IN
   ( SELECT Code
      FROM FinancialCodes
      WHERE type = 'revenue' );
```

Year	Quarter	Code	Amount
1999	Q1	r1	1023
1999	Q2	r1	2033
1999	Q3	r1	2998
1999	Q4	r1	3014

Year	Quarter	Code	Amount
2000	Q1	r1	3114
...	...	...	...

ANY キーワードと ALL キーワードも同様の方法で使用できます。たとえば、次のクエリは前述のクエリと同じ結果を返しますが、ANY キーワードを使用しています。

```
SELECT *
FROM FinancialData
WHERE FinancialData.Code = ANY
  (  SELECT FinancialCodes.Code
     FROM FinancialCodes
     WHERE type = 'revenue' );
```

=ANY 条件は IN 条件とまったく同じですが、ANY を < や > などの不等号とともに使用すると、サブクエリをより柔軟に使用できます。

ALL キーワードは ANY に似ています。たとえば、次のクエリは歳入以外の経理データをリストします。

```
SELECT *
FROM FinancialData
WHERE FinancialData.Code <> ALL
  (  SELECT FinancialCodes.Code
     FROM FinancialCodes
    WHERE type = 'revenue' );
```

このクエリは、NOT IN を使用した場合の次の文と同じです。

```
SELECT *
FROM FinancialData
WHERE FinancialData.Code NOT IN
  (  SELECT FinancialCodes.Code
    FROM FinancialCodes
    WHERE type = 'revenue' );
```

相関サブクエリと非相関サブクエリ

サブクエリには、親の文に定義されたオブジェクトへの参照を含めることができます。このことは「外部参照」と呼ばれます。外部参照があるサブクエリは「相関サブクエリ」と呼ばれます。相関サブクエリは、外部クエリとは別に評価することはできません。これは、サブクエリが親の文の値を使用するためです。つまり、サブクエリは親の文のローごとに実行されます。したがって、サブクエリの結果は、親の文で評価されるアクティブなローに依存します。

たとえば、次の文のサブクエリは、Products テーブル内のアクティブなローに依存する値を返します。

```
SELECT Name, Description
FROM Products
WHERE Quantity < 2 * (
  SELECT AVG( Quantity )
  FROM SalesOrderItems
  WHERE Products.ID=SalesOrderItems.ProductID );
```

この例では、このサブクエリの Products.ID カラムは外部参照です。このクエリは、在庫数が平均注文数の 2 倍より少ない製品、具体的には、メインクエリの WHERE 句によってテストされている製品の、名前と説明を抽出します。サブクエリは SalesOrderItems テーブルをスキャンしてこれを実行します。しかし、サブクエリの WHERE 句にある Products.ID カラムは、サブクエリではなく、**メイン**クエリの FROM 句に指定されているテーブルのカラムを参照します。データベースサーバは Products テーブルの各ローの間を移動して、サブクエリの WHERE 句を評価するときに、現在のローの ID 値を使用します。

サブクエリで参照されるカラムが、サブクエリの FROM 句で参照されるテーブルになくても、外部クエリの FROM 句で参照されるテーブルにあれば、クエリはエラーなく実行されます。SQL Anywhere では、サブクエリのカラムが外部クエリのテーブル名で暗黙的に修飾されます。

親の文にオブジェクトへの参照を含まないサブクエリを、「非相関サブクエリ」と呼びます。次の例では、サブクエリは正確に 1 つの値、SalesOrderItems テーブルの平均数を計算します。クエリを評価するときに、データベースサーバはこの値を一度計算し、その値を Products テーブルの Quantity フィールドにあるそれぞれの値と比較して、対応するローを選択するかどうかを決定します。

```
SELECT Name, Description
FROM Products
WHERE Quantity <  2 * (
  SELECT AVG( Quantity )
  FROM SalesOrderItems );
```

ネストされたサブクエリ

「ネストされたサブクエリ」とは、別のサブクエリの中にネストされたサブクエリです。定義できるサブクエリのネストのレベルに制限はありませんが、3 つ以上のレベルのクエリは、それ以下のレベルのクエリに比べて、実行にかなりの時間がかかります。

次の例では、ネストされたサブクエリを使用して、Fees 部の任意の品目が注文された日に出荷された注文の注文 ID とライン ID を決定します。

```
SELECT ID, LineID
FROM SalesOrderItems
WHERE ShipDate = ANY (
  SELECT OrderDate
  FROM SalesOrders
  WHERE FinancialCode IN (
    SELECT Code
    FROM FinancialCodes
    WHERE ( Description = 'Fees' ) ) );
```

ID	LineID
2001	1
2001	2
2001	3

ID	LineID
2002	1
...	...

この例では、最も内側のサブクエリが、"Fees" という説明がある経理コードのカラムを生成します。

```
SELECT Code
FROM FinancialCodes
WHERE ( Description = 'Fees' );
```

次のサブクエリは、最も内側のサブクエリが選択したコードに一致するコードを持つ品目の注文日を検索します。

```
SELECT OrderDate
FROM SalesOrders
WHERE FinancialCode
IN ( subquery-expression );
```

最後に、一番外側のクエリが、サブクエリの検索した日付のいずれかに出荷された注文の注文 ID とライン ID を検索します。

```
SELECT ID, LineID
FROM SalesOrderItems
WHERE ShipDate = ANY ( subquery-expression );
```

ジョインに代わるサブクエリの使用

たとえば、受注と受注先の日付順リストが必要な場合に、Customers ID ではなく会社名を知りたいとします。次のようなジョインを使用して、この結果を得ることができます。

ジョインの使用

2001 年 1 月 1 日以降の受注 ID、日付、各注文を行った会社名をリストするには、次のクエリを実行します。

```
SELECT SalesOrders.ID,
       SalesOrders.OrderDate,
       Customers.CompanyName
FROM SalesOrders
  KEY JOIN Customers
WHERE OrderDate > '2001/01/01'
ORDER BY OrderDate;
```

サブクエリの使用方法

次の SQL 文は、ジョインではなくサブクエリを使用して同じ結果を得ます。

```
SELECT SalesOrders.ID,
  SalesOrders.OrderDate,
  ( SELECT CompanyName FROM Customers
    WHERE Customers.ID = SalesOrders.CustomerID )
FROM SalesOrders
```

```
WHERE OrderDate > '2001/01/01'
ORDER BY OrderDate;
```

SalesOrders テーブルがサブクエリの一部でない場合でも、サブクエリは SalesOrders テーブル内の CustomerID カラムを参照します。一方、SalesOrders.CustomerID カラムは SQL 文の本文にある SalesOrders テーブルを参照します。

他のテーブルから要求されるカラムが 1 つだけである場合は、ジョインの代わりにサブクエリを使用できます。サブクエリが返すことができるカラムは 1 つだけです。この例では CompanyName カラムだけを必要としていたので、ジョインをサブクエリに変更することができました。

外部ジョインの使用

ワシントン州在住の顧客名すべてとその顧客の最も最近の受注 ID をリストするには、次のクエリを実行します。

```
SELECT  CompanyName, State,
  ( SELECT MAX( ID )
     FROM SalesOrders
    WHERE SalesOrders.CustomerID = Customers.ID )
FROM Customers
WHERE State = 'WA';
```

CompanyName	State	MAX(SalesOrders.ID)
Custom Designs	WA	2547
It's a Hit!	WA	(NULL)

It's a Hit! という会社は何も注文しなかったので、サブクエリはこの顧客については NULL を返します。内部ジョインを使用した場合、発注しなかった会社はリストされません。

外部ジョインを明示的に指定することもできます。その場合は、次のように GROUP BY 句も指定する必要があります。

```
SELECT CompanyName, State,
  MAX( SalesOrders.ID )
FROM Customers
  KEY LEFT OUTER JOIN SalesOrders
WHERE State = 'WA'
GROUP BY CompanyName, State;
```

WHERE 句でのサブクエリ

WHERE 句内のサブクエリは、ロー選択のプロセスの一部として機能します。ローの選択に使用する基準が別のテーブルの結果に依存するときに、WHERE 句内にサブクエリを使用します。

例

在庫数が平均注文数の 2 倍よりも少ない製品を検索します。

```
SELECT Name, Description
FROM Products WHERE Quantity <  2 * (
```

```
SELECT AVG( Quantity )
FROM SalesOrderItems );
```

このクエリは 2 段階で実行されます。まず、注文ごとに要求される品目の平均数を検索します。次に、どの製品の在庫数がその数の 2 倍より少ないかを検索します。

2 段階のクエリ

要求される品目の**数**は、品目のタイプ、顧客、注文ごとに、SalesOrderItems テーブルの Quantity カラムに格納されます。サブクエリは次のようになります。

```
SELECT AVG( Quantity )
FROM SalesOrderItems;
```

これによって SalesOrderItems テーブルの品目の平均数、25.851413 が返されます。

次のクエリは、前述のクエリで抽出した値の 2 倍よりも少ない在庫数の品目の名前とその説明を返します。

```
SELECT Name, Description
FROM Products
WHERE Quantity < 2*25.851413;
```

サブクエリを使用すると、この 2 つの手順を 1 つのオペレーションにまとめることができます。

WHERE 句でのサブクエリの目的

WHERE 句内でのサブクエリは、探索条件の一部です。

参照

- 「クエリ」281 ページ

HAVING 句でのサブクエリ

サブクエリは通常は WHERE 句内で探索条件として使用しますが、クエリの HAVING 句で使用することもできます。HAVING 句内のサブクエリは、HAVING 句内の他の式と同様に、ローグループの選択の一部として使用されます。

「どの製品の平均在庫数が、顧客ごとの各品目の平均注文数の 2 倍以上あるのか」という要求は、HAVING 句内にサブクエリを持つクエリになります。

例

```
SELECT Name, AVG( Quantity )
FROM Products
GROUP BY Name
HAVING AVG( Quantity ) > 2* (
  SELECT AVG( Quantity )
  FROM SalesOrderItems
 );
```

name	AVG(Products.Quantity)
Baseball Cap	62.000000
Shorts	80.000000
Tee Shirt	52.333333

クエリは次のように実行します。

●サブクエリは SalesOrderItems テーブルにある品目の平均数を計算します。

●メインクエリは Products テーブルを調べて、製品ごとの平均数を計算し、製品名でグループ化します。

●HAVING 句は、各平均数がサブクエリで検索された数量の 2 倍を超えるかどうかを確認します。超える場合、メインクエリはそのローグループを返します。超えない場合は返しません。

●SELECT 句は、グループごとに 1 つの計算ローを生成し、各製品の名前と在庫の平均数を示します。

次の例で示すように、HAVING 句には外部参照も使用できます。この例は、前述の例を若干変更したものです。

例

この例では、平均注文数が在庫数の半分よりも多い製品の ID 番号とライン ID 番号を検索します。

```
SELECT ProductID, LineID
FROM SalesOrderItems
GROUP BY ProductID, LineID
HAVING 2* AVG( Quantity ) > (
  SELECT Quantity
  FROM Products
  WHERE Products.ID = SalesOrderItems.ProductID );
```

ProductID	LineID
601	3
601	2
601	1
600	2
...	...

この例では、サブクエリは、HAVING 句にテストされるローグループに対応する製品の在庫数を生成します。サブクエリは外部参照 SalesOrderItems.ProductID を使用して、その特定製品のレコードを選択します。

比較演算子を持つサブクエリは 1 つの値を返す

このクエリは比較演算子 > を使用するので、サブクエリは 1 つの値を返します。この場合は、1 つの値を返します。Products テーブルの ID フィールドがプライマリキーなので、特定の製品 ID に対応する Products テーブルのレコードは 1 つだけになります。

サブクエリを使用した述部

サブクエリは WHERE 句または HAVING 句に置かれる式なので、サブクエリの探索条件も見なれたものになります。

次の探索条件があります。

- **サブクエリを使用した述部の比較**　メインクエリのテーブルにある各レコードについて、サブクエリが生成した 1 つの値と式の値を比較します。比較テストでは、サブクエリで提供される演算子 (=、<>、<、<=、>、<=) を使用します。
- **限定比較テスト**　サブクエリが生成した値のそれぞれのセットと式の値を比較します。
- **サブクエリセットメンバーシップテスト**　サブクエリが生成した値のセットのいずれかと、式の値が一致するかどうかを調べます。
- **存在テスト**　サブクエリがローを生成するかどうかを調べます。

参照

- 「クエリ」281 ページ

サブクエリ比較テスト

サブクエリの比較テスト (=, <>, <、<=, >, >=) は、単純な比較テストを変更したものです。サブクエリの比較テストでは、演算子の後ろに来る式がサブクエリになる点だけが異なります。このテストを使用して、メインクエリのローからの値を、サブクエリが生成する **1 つ**の値と比較します。

例

このクエリにはサブクエリ比較テストの例が含まれています。

```
SELECT Name, Description, Quantity
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

name	Description	Quantity
Tee Shirt	Tank Top	28
Baseball Cap	Wool cap	12

name	Description	Quantity
Visor	Cloth Visor	36
Visor	Plastic Visor	28
...	...	...

次のサブクエリは単一の値、つまり各顧客が発注したタイプ別平均品目数を、SalesOrderItems テーブルから取り出します。

```
SELECT AVG( Quantity )
FROM SalesOrderItems;
```

メインクエリは、各品目の在庫数をその値と比較します。

比較テストのサブクエリは 1 つの値を返す

比較テストのサブクエリは 1 つの値を返します。次の例では、SalesOrderItems テーブルから 2 つのカラムを抽出するサブクエリを持つクエリを考えてみます。

```
SELECT Name, Description, Quantity
FROM Products
WHERE Quantity <  2 * (
  SELECT AVG( Quantity ), MAX( Quantity )
  FROM SalesOrderItems);
```

エラーが返されます。

参照

- 「SELECT リストの中にカラムが 2 つ以上指定されています」『エラーメッセージ』

サブクエリと IN テスト

サブクエリセットメンバーシップテストを使用して、メインクエリからの値をサブクエリの複数の値と比較できます。

サブクエリセットメンバーシップテストは、メインクエリの各ローの 1 つのデータ値を、サブクエリが生成したデータ値の 1 つのカラムと比較します。メインクエリのデータ値がカラムのデータ値の**いずれか**と一致する場合、サブクエリは TRUE を返します。

例

Shipping 部または Finance 部の部長である従業員の名前を選択します。

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
  SELECT DepartmentHeadID
  FROM Departments
  WHERE ( DepartmentName='Finance' OR
        DepartmentName = 'Shipping' ) );
```

GivenName	Surname
Mary Anne	Shea
Jose	Martinez

この例のサブクエリは、Shipping 部と Finance 部の部長に対応する ID 番号を、Departments テーブルから抽出します。次にメインクエリが、サブクエリによって検索された 2 つの値のいずれかに一致する ID 番号を持つ従業員の名前を返します。

```
SELECT DepartmentHeadID
FROM Departments
WHERE ( DepartmentName='Finance' OR
      DepartmentName = 'Shipping' );
```

セットメンバーシップテストは =ANY テストと同等

サブクエリセットメンバーシップテストは =ANY テストと同等です。次のクエリは前述の例のクエリと同等です。

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID = ANY (
  SELECT DepartmentHeadID
  FROM Departments
  WHERE ( DepartmentName='Finance' OR
        DepartmentName = 'Shipping' ) );
```

セットメンバーシップテストの否定

サブクエリセットメンバーシップテストは、サブクエリによって生成される値に一致しないカラム値を持つローを抽出する場合にも使用できます。セットメンバーシップテストを否定するには、キーワード IN の前に NOT を挿入します。

例

このクエリのサブクエリは、Finance 部または Shipping 部の部長でない従業員の姓と名前を返します。

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID NOT IN (
  SELECT DepartmentHeadID
  FROM Departments
  WHERE ( DepartmentName='Finance' OR
        DepartmentName = 'Shipping' ) );
```

サブクエリと ANY テスト

ANY テストは、SQL 比較演算子 (=、>、<、>=、<=、!=、<>、!>、!<) のいずれかと組み合せて使用して、1 つの値をサブクエリが生成するデータ値のカラムと比較します。テストを実行するには、SQL は指定された比較演算子を使用して、テスト値をカラムのデータ値のそれぞれと比較します。**いずれかの**比較の結果が TRUE になる場合、ANY テストは TRUE を返します。

ANY を使用するサブクエリは 1 つのカラムを返します。

例

注文番号 2005 の最初の製品が出荷された後に受けた注文の注文 ID と顧客 ID を検索します。

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > ANY (
  SELECT ShipDate
  FROM SalesOrderItems
  WHERE ID=2005 );
```

ID	CustomerID
2006	105
2007	106
2008	107
2009	108
...	...

このクエリを実行すると、メインクエリが、注文番号 2005 の**すべての**製品の出荷日に対して、各注文の注文日をテストします。注文日が注文番号 2005 の **1 つ**の出荷の出荷日より後であれば、SalesOrders テーブルの注文 ID と顧客 ID が結果セットに示されます。このように ANY テストは OR 演算子に似ています。前述のクエリは、「この注文は注文番号 2005 の最初の製品が出荷された後に受けたものか、または注文番号 2005 の 2 番目の製品が出荷された後に受けたものか、または ...」というように解釈できます。

ANY 演算子の知識

ANY 演算子はやや複雑な場合があります。このクエリは、「注文番号 2005 の任意の製品が出荷された後に受けた注文を返す」と解釈してしまいがちです。しかし、それでは注文番号 2005 の**すべての**製品が出荷された後に受けた注文の注文 ID と顧客 ID を返すことになり、クエリの動作と異なります。

そうではなく、「注文番号 2005 の**少なくとも 1 つの**製品が出荷された後に受けた注文の注文 ID と顧客 ID を返す」というようにクエリを解釈してみます。キーワード SOME を使用すると、もう少し直感的な方法でクエリを表現できます。次のクエリは前述のクエリと同等です。

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > SOME (
  SELECT ShipDate
  FROM SalesOrderItems
  WHERE ID=2005 );
```

キーワード SOME はキーワード ANY と同等です。

ANY 演算子についての注意

ANY テストには、このほかに 2 つの重要な特徴があります。

- **空のサブクエリの結果セット**　サブクエリが空の結果セットを生成する場合、ANY テストは FALSE を返します。結果がない場合、少なくとも 1 つの結果が比較テストを満たしているというのは真ではないので、これは理にかなっています。

- **サブクエリの結果セットの NULL 値**　サブクエリの結果セットには少なくとも 1 つの NULL 値があることが前提です。結果セットの NULL 以外のすべてのデータ値に対して比較テストが FALSE の場合、ANY は UNKNOWN を返します。これは、比較テストが保持するサブクエリの値があるかどうか、この状況では確定できないためです。値があるかどうかは、結果セットの NULL データの**正確な**値によって異なります。

参照

- 「ANY 探索条件と SOME 探索条件」『SQL Anywhere サーバ SQL リファレンス』

サブクエリと ALL テスト

ALL テストは、SQL 比較演算子 (=、>、<、>=、<=、!=、<>、!>、!<) のいずれかと組み合せて使用して、1 つの値をサブクエリが生成するデータ値と比較します。テストを実行するには、SQL は指定された比較演算子を使用して、テスト値を結果セットのデータ値のそれぞれと比較します。すべての比較の結果が TRUE になる場合、ALL テストは TRUE を返します。

例

次の例は、注文番号 2001 のすべての製品が出荷された後に受けた注文の注文 ID と顧客 ID を検索します。

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > ALL (
  SELECT ShipDate
  FROM SalesOrderItems
  WHERE ID=2001 );
```

ID	CustomerID
2002	102
2003	103
2004	104
2005	101
...	...

このクエリを実行すると、メインクエリは、注文番号 2001 の**すべての**製品の出荷日に対して、各注文の注文日をテストします。注文日が注文番号 2001 の**すべての**出荷の出荷日より後であれ

ば、SalesOrders テーブルの注文 ID と顧客 ID が結果セットに示されます。このように ALL テストは AND 演算子に似ています。前述のクエリは、「この注文は注文番号 2001 の最初の製品が出荷される前に受けたもので、なおかつ注文番号 2001 の 2 番目の製品が出荷される前に受けたもので、なおかつ ...」というように解釈できます。

ALL 演算子についての注意

ALL テストには、このほかに 3 つの重要な特徴があります。

- **空のサブクエリの結果セット**　サブクエリが空の結果セットを生成した場合、ALL テストは TRUE を返します。結果がない場合、比較テストが結果セットのどの値に対しても適用しているというのは真なので、これは理にかなっています。

- **サブクエリの結果セットの NULL 値**　結果セットのいずれかの値に対する比較テストが FALSE の場合、ALL は FALSE を返します。すべての値が TRUE の場合は TRUE を返します。それ以外の場合は、UNKNOWN を返します。たとえば、サブクエリの結果セットに NULL 値があっても、NULL 以外のすべての値の探索条件が TRUE の場合などです。

- **ALL テストの否定**　次の 2 つの式は同じでは**ありません**。

```
NOT a = ALL (subquery)
a <> ALL (subquery)
```

参照

- 「ANY、ALL、または SOME に続くサブクエリ」612 ページ

サブクエリと EXISTS テスト

サブクエリ比較テストとセットメンバーシップテストに使用されるサブクエリは、いずれもサブクエリテーブルからデータ値を返します。しかし、場合によっては、**どの**結果をサブクエリが返すのかではなく、サブクエリが**何らかの**結果を返すのかどうかが重要であることがあります。存在テスト (EXISTS) は、サブクエリがクエリ結果のローを生成するかどうかを調べます。サブクエリが 1 つ以上の結果のローを返す場合、EXISTS テストは TRUE を返します。結果のローを返さない場合は、FALSE を返します。

例

ここでは、「2001 年 7 月 13 日よりも後に発注したのはどの顧客か」という要求を、サブクエリを使って表現してみます。

```
SELECT GivenName, Surname
FROM Customers
WHERE EXISTS (
  SELECT *
  FROM SalesOrders
  WHERE ( OrderDate > '2001-07-13' ) AND
      ( Customers.ID = SalesOrders.CustomerID ) );
```

GivenName	Surname
Almen	de Joie

GivenName	Surname
Grover	Pendelton
Ling Ling	Andrews
Bubba	Murphy

存在テストの説明

この例では、サブクエリが、Customers テーブルのローごとに、その顧客 ID が 2001 年 7 月 13 日より後に発注した顧客 ID に対応するかどうかを調べます。対応していれば、クエリはその顧客の姓と名前をメインテーブルから抽出します。

EXISTS テストはサブクエリの結果を使用しません。単にサブクエリがローを生成するかどうかを調べるだけです。このため、次の 2 つのサブクエリに適用した存在テストでも同じ結果が返されます。これらはサブクエリですから、それ自体では処理できません。サブクエリが参照する Customers テーブルは、メインクエリの一部であってサブクエリの一部ではないからです。

```
SELECT *
FROM Customers, SalesOrders
WHERE ( OrderDate > '2001-07-13' ) AND
    ( Customers.ID = SalesOrders.CustomerID )

SELECT OrderDate
FROM Customers, SalesOrders
WHERE ( OrderDate > '2001-07-13' ) AND
    ( Customers.ID = SalesOrders.CustomerID );
```

便宜上、"SELECT *" という表記を使用していますが、SalesOrders テーブルのどのカラムが SELECT 文に指定されるかどうかは問題ではありません。

存在テストの否定

EXISTS テストの論理は、NOT EXISTS という形式で否定できます。この場合、テストはサブクエリがローを返さない場合に TRUE を、ローを返す場合に FALSE を返します。

相関サブクエリ

サブクエリには Customers テーブルからの ID カラムへの参照が含まれています。メインテーブル内のカラムや式への参照は、「外部参照」と呼ばれます。また、そのサブクエリは「相関」です。概念的には、SQL は Customers テーブルを調べ、顧客ごとにサブクエリを実行して、前述のクエリを処理します。SalesOrders テーブルの注文日が 2001 年 7 月 13 日より後で、Customers テーブルと SalesOrders テーブルの顧客 ID が一致していれば、Customers テーブルからの姓と名前が表示されます。サブクエリはメインクエリを参照するので、この項のサブクエリは、前述の項のサブクエリとは異なり、サブクエリをそれだけで実行しようとするとエラーが返されます。

参照

- 「相関サブクエリと非相関サブクエリ」597 ページ

オプティマイザによるサブクエリからジョインへの自動変換

クエリオプティマイザは、サブクエリを利用するクエリの多くをジョインとして自動的に書き換えます。変換はユーザによるアクションを必要とすることなく実行されます。この項では、データベースでのクエリのパフォーマンスを理解できるように、どのサブクエリがジョインに変換できるのかを説明します。

マルチレベルのクエリをジョインで作成するために満たす必要がある基準は、演算子のタイプ、クエリの構造、サブクエリの構造によって異なります。サブクエリが WHERE 句内にある場合は、次のフォームになることに注意してください。

SELECT *select-list*
FROM *table*
WHERE
[**NOT**] *expression comparison-operator* **(** *subquery-expression* **)**
| [**NOT**] *expression comparison-operator* { **ANY** | **SOME** } **(** *subquery-expression* **)**
| [**NOT**] *expression comparison-operator* **ALL (** *subquery-expression* **)**
| [**NOT**] *expression* **IN (** *subquery-expression* **)**
| [**NOT**] **EXISTS (** *subquery-expression* **)**
GROUP BY *group-by-expression*
HAVING *search-condition*

たとえば、「Mrs. Clarke と Suresh がいつ注文し、どの担当者が注文を受けたか」という要求を考えてみます。これは次のクエリを使用して回答できます。

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID IN (
  SELECT ID
  FROM Customers
  WHERE Surname = 'Clarke' OR GivenName = 'Suresh' );
```

OrderDate	SalesRepresentative
2001-01-05	1596
2000-01-27	667
2000-11-11	467
2001-02-04	195
...	...

サブクエリは WHERE 句に名前がリストされている 2 人の顧客に対応する顧客 ID のリストを生成します。メインクエリはこの 2 人の注文に対応する注文日と担当者を検索します。

同じ問い合わせをジョインを使用して応答できます。このクエリの、2 つのテーブルのジョインを使用した代替形式を次に示します。

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
 ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

 Copyright © 2013, SAP AG or an SAP affiliate company.

この形式のクエリは SalesOrders テーブルを Customers テーブルにジョインして各顧客の注文を検索し、Suresh と Clarke のレコードだけを返します。

サブクエリは有効でもジョインが有効でない場合

サブクエリは有効でも、ジョインが有効でない場合があります。次に例を示します。

```
SELECT Name, Description, Quantity
FROM Products
WHERE Quantity <  2 * (
  SELECT AVG( Quantity )
  FROM SalesOrderItems );
```

name	Description	Quantity
Tee Shirt	Tank Top	28
Baseball Cap	Wool cap	12
Visor	Cloth Visor	36
...	...	...

この場合、内部クエリは集計クエリで外部クエリは集計クエリではないので、2 つのクエリを簡単なジョインで組み合わせることはできません。

参照

● 「ジョイン：複数テーブルからのデータ検索」488 ページ

比較演算子に続くサブクエリ

比較演算子に続くサブクエリ (=、>、<、>=、<=、!=、<>、!>、!<) は、比較と呼ばれます。オプティマイザは、サブクエリが次のような場合に、これらのサブクエリをジョインに変換します。

- メインクエリのローごとに値を 1 つずつ返す
- GROUP BY 句を含んでいない
- キーワード DISTINCT を含んでいない
- UNION クエリではない
- 集計クエリではない

例

「Suresh の製品がいつ注文され、どの担当者が注文を受けたか」という要求をサブクエリで表現したとします。

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
```

```
WHERE CustomerID = (
  SELECT ID
  FROM Customers
  WHERE GivenName = 'Suresh' );
```

このクエリは基準を満たすので、ジョインを使用するクエリに変換できます。

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
 ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

しかし、「在庫数が平均注文数の 2 倍よりも少ない製品を検索する」という要求はジョインに変換できません。これは、サブクエリに集計関数 AVG が含まれているためです。

```
SELECT Name, Description
FROM Products
WHERE Quantity <  2 * (
  SELECT AVG( Quantity )
  FROM SalesOrderItems );
```

ANY、ALL、または SOME に続くサブクエリ

キーワード ALL、ANY、または SOME に続くサブクエリは、限定比較と呼ばれます。オプティマイザは、次のような場合にこれらのサブクエリをジョインに変換します。

- メインクエリが GROUP BY 句を含んでおらず、集計クエリでない。または、サブクエリが 1 つの値を返す。
- サブクエリが GROUP BY 句を含んでいない。
- サブクエリがキーワード DISTINCT を含んでいない。
- サブクエリが UNION クエリではない。
- サブクエリが集計クエリではない。
- '*expression comparison-operator* { **ANY** | **SOME** } **(** *subquery-expression* **)**' の部分が否定されていない。
- '*expression comparison-operator* **ALL (** *subquery-expression* **)**' の部分が否定されている。

最初の 4 つの条件は、比較的簡単です。

例

「Mrs. Clarke と Suresh がいつ注文し、どの担当者が注文を受けたか」という要求は、サブクエリの形式で処理できます。

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID = ANY (
  SELECT ID
  FROM Customers
  WHERE Surname = 'Clarke' OR GivenName = 'Suresh' );
```

または、ジョインの形式で表現できます。

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
  ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

しかし、「Mrs. Clarke、Suresh、および顧客でもある従業員が、注文したのはいつか」という要求は UNION クエリとして表現されるので、ジョインには変換できません。

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID = ANY (
  SELECT ID
  FROM Customers
  WHERE Surname = 'Clarke' OR GivenName = 'Suresh'
  UNION
  SELECT EmployeeID
  FROM Employees );
```

同様に、「すべての製品の最初の出荷日の後に出荷されていない注文の注文 ID と顧客 ID を検索する」という要求は、集計クエリで表現されるため、ジョインに変換できません。

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE NOT OrderDate > ALL (
  SELECT FIRST ( ShipDate )
  FROM SalesOrderItems
  ORDER BY ShipDate );
```

ANY と ALL 演算子を使用するサブクエリの否定

5 つ目の条件はやや複雑です。次の形式のクエリがジョインに変換されます。

SELECT *select-list*
FROM *table*
WHERE NOT *expression comparison-operator* **ALL (** *subquery-expression* **)**

SELECT *select-list*
FROM *table*
WHERE *expression comparison-operator* **ANY (** *subquery-expression* **)**

ただし、次のクエリはジョインに変換されません。

SELECT *select-list*
FROM *table*
WHERE *expression comparison-operator* **ALL (** *subquery-expression* **)**

SELECT *select-list*
FROM *table*
WHERE NOT *expression comparison-operator* **ANY (** *subquery-expression* **)**

最初の 2 つのクエリも、後の 2 つのクエリも、それぞれ同等です。すでに説明したように、ANY 演算子は OR 演算子と似ていますが、引数の数が異なります。同様に、ALL 演算子は AND 演算子に似ています。たとえば、次の 2 つの式は同等です。

```
NOT ( ( X > A ) AND ( X > B ) )
( X <= A ) OR ( X <= B )
```

次の 2 つの式も同等です。

```
WHERE NOT OrderDate > ALL (
  SELECT FIRST ( ShipDate )
  FROM SalesOrderItems
  ORDER BY ShipDate )

WHERE OrderDate <= ANY (
  SELECT FIRST ( ShipDate )
  FROM SalesOrderItems
  ORDER BY ShipDate )
```

ANY と ALL の否定

一般に、次の 2 つの式は同等です。

NOT *column-name operator* **ANY (** *subquery-expression* **)**

column-name inverse-operator **ALL (** *subquery-expression* **)**

次の式も、一般に同等です。

NOT *column-name operator* **ALL (** *subquery-expression* **)**

column-name inverse-operator **ANY (** *subquery-expression* **)**

inverse-operator は、次の表に示すように、*operator* を否定することによって取得されます。

operator	inverse-operator
=	<>
<	=>
>	=<
=<	>
=>	<
<>	=

IN に続くサブクエリ

オプティマイザは、IN キーワードが次のような場合に続くサブクエリのみを変換します。

- メインクエリが GROUP BY 句を含んでおらず、集計クエリでない。または、サブクエリが 1 つの値を返す。
- サブクエリが GROUP BY 句を含んでいない。
- サブクエリがキーワード DISTINCT を含んでいない。

●サブクエリが UNION クエリではない。

●サブクエリが集計クエリではない。

●'expression IN (subquery-expression)' の部分が否定されていない。

例

「部長でもある従業員の名前を検索する」という要求は、次のクエリで表現されますが、この要求は条件を満たすため、ジョインされたクエリに変換されます。

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
  SELECT DepartmentHeadID
  FROM Departments
  WHERE ( DepartmentName ='Finance' OR
        DepartmentName = 'Shipping' ) );
```

しかし、「部長か顧客のいずれかである従業員の名前を検索する」という要求は、UNION クエリで表現されているとジョインに変換されません。

IN 演算子に続く UNION クエリは変換できない

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
  SELECT DepartmentHeadID
  FROM Departments
  WHERE ( DepartmentName='Finance' OR
        DepartmentName = 'Shipping' )
  UNION
  SELECT CustomerID
  FROM SalesOrders);
```

同様に、「部長ではない従業員の名前を検索する」という要求は、次に示す否定のサブクエリで表現されますが、変換されません。

```
SELECT GivenName, Surname
FROM Employees
 WHERE NOT EmployeeID IN (
  SELECT DepartmentHeadID
  FROM Departments
  WHERE ( DepartmentName='Finance' OR
        DepartmentName = 'Shipping' ) );
```

IN サブクエリまたは ANY サブクエリがジョインに変換されるために必要な条件は、同じです。これは、2 つの式が論理的には同等であるためです。

ANY 演算子を使用するクエリに変換される、IN 演算子を使用するクエリ

SQL Anywhere は、IN 演算子を使用するクエリを ANY 演算子を使用するクエリに変換し、サブクエリをジョインに変換するかどうかを決定することがあります。たとえば、次の 2 つの式は同等です。

WHERE *column-name* **IN(** *subquery-expression* **)**

WHERE *column-name* **= ANY(** *subquery-expression* **)**

同様に、次の 2 つのクエリは同等です。

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
  SELECT DepartmentHeadID
  FROM Departments
  WHERE ( DepartmentName='Finance' OR
        DepartmentName = 'Shipping' ) );

SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID = ANY (
  SELECT DepartmentHeadID
  FROM Departments
  WHERE ( DepartmentName='Finance' OR
        DepartmentName = 'Shipping' ) );
```

EXISTS に続くサブクエリ

オプティマイザは次のような場合のみ、EXISTS キーワードに続くサブクエリを変換します。

- メインクエリが GROUP BY 句を含んでおらず、集計クエリでない。または、サブクエリが 1 つの値を返す。
- 'EXISTS (subquery)' の部分が否定されていない。
- サブクエリが相関である。つまり、外部参照を含んでいる。

例

「どの顧客が 2001 年 7 月 13 日以降に発注したか」という要求は、外部参照 **Customers.ID** = **SalesOrders.CustomerID** を含む否定されていないサブクエリを持つクエリで表現できるので、次のジョインで表すことができます。

```
SELECT GivenName, Surname
FROM Customers
WHERE EXISTS (
  SELECT *
  FROM SalesOrders
  WHERE ( OrderDate > '2001-07-13' ) AND
      ( Customers.ID = SalesOrders.CustomerID ) );
```

EXISTS キーワードは、空の結果セットをチェックするようデータベースサーバに通知するものです。内部ジョインが使用されていると、データベースサーバは、FROM 句内のすべてのテーブルからのデータがあるローのみを自動的に表示します。つまり、次のクエリは、サブクエリを持つクエリが返すものと同じローを返します。

```
SELECT DISTINCT GivenName, Surname
FROM Customers, SalesOrders
WHERE ( SalesOrders.OrderDate > '2001-07-13' ) AND
    ( Customers.ID = SalesOrders.CustomerID );
```

データ操作文

データの追加、変更、削除に使用する文はデータ操作文と呼ばれ、ANSI SQL のデータ操作言語 (DML) 文部分のサブセットとなっています。

主要な DML 文を次に示します。

- **INSERT 文**　テーブルまたはビューに新規のローを追加します。
- **UPDATE 文**　一連のテーブルまたはビューのローを変更します。
- **DELETE 文**　一連のテーブルまたはビューのローを削除します。
- **MERGE 文**　テーブルまたはビューの特定のローを追加、変更、削除します。

前述の文に加えて、LOAD TABLE 文と TRUNCATE TABLE 文は、データのバルクロードや削除を行う場合に便利です。

参照

- 「INSERT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「UPDATE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DELETE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「MERGE 文」『SQL Anywhere サーバ SQL リファレンス』

データ操作の権限

データ操作文は、修正するデータベーステーブルに対する適切な権限を持っている場合のみ実行できます。データベースの管理者とデータベースオブジェクトの所有者は、GRANT 文と REVOKE 文を使用して、だれがどのデータ操作機能にアクセスできるのかを決定します。

権限は個々のユーザ、ロール、ユーザ拡張ロールに付与できます。

参照

- 「ユーザのセキュリティ (ロールと権限)」『SQL Anywhere サーバ データベース管理』

トランザクションとデータ操作

データを修正すると、各データ操作文によって影響を受ける各ローの新旧の状態がコピーされて、ロールバックログに格納されます。トランザクションを開始し、間違いに気づいてトランザクションをロールバックすると、データベースを前の状態に回復できます。

参照

- 「トランザクションと独立性レベル」861 ページ

永続的なデータ変更

COMMIT 文は、すべての変更を永続的なものにします。

COMMIT 文は、まとまった意味を持つ文のグループの後で使用してください。たとえば、ある顧客の口座から別の顧客の口座に金銭を振り込む場合、振り込み前と後の合計金額が同一である必要があるため、振り込まれる側の口座にその金額を加え、その後で振り込む側の口座からその金額を削除して、最後にコミットします。

auto_commit オプションを On に設定すると、Interactive SQL に対して変更を自動的にコミットするように指定できます。これは Interactive SQL のオプションです。auto_commit を On に設定すると、INSERT 文、UPDATE 文、DELTE 文を実行するたびに Interactive SQL が COMMIT 文を発行します。このため、パフォーマンスが大幅に低下することがあります。このような場合には、auto_commit オプションを Off に設定することをおすすめします。

注意して COMMIT 文を使用すること
このチュートリアルにある例を試してみる場合、データベースの変更を確定しても問題がないと確信するまでは、どのような変更に対してもコミットしないように注意してください。

参照

- 「Interactive SQL オプション」『SQL Anywhere サーバ データベース管理』
- 「COMMIT 文」『SQL Anywhere サーバ SQL リファレンス』

変更のキャンセル

コミットされていない変更はすべてキャンセルできます。SQL では、ROLLBACK 文を使用することによって最後のコミット以降に加えた変更をすべて取り消せます。この文は、最後に変更を確定した後にデータベースに対して行われたすべての変更を取り消します。

参照

- 「ROLLBACK 文」『SQL Anywhere サーバ SQL リファレンス』

トランザクションとデータリカバリ

SQL Anywhere は、システム障害や停電が発生した場合にデータベースの整合性を保護します。データベースサーバをリストアするためのオプションが複数用意されています。たとえば、SQL Anywhere によって別のドライブに保存されたログファイルを使用して、データをリストアできます。リカバリのためにログファイルを使用する場合、SQL Anywhere はデータベースを頻繁に更新する必要がないため、データベースサーバのパフォーマンスは向上します。

トランザクション処理により、データベースサーバはデータが一貫性を保っていることを識別できます。トランザクション処理は、なんらかの理由でトランザクションが正常に完了しなかった場合に、トランザクション全体が取り消されるか、ロールバックしたことを確認します。トランザクションが失敗しても、データベースには影響ありません。

SQL Anywhere のトランザクション処理は、トランザクションの途中でシステムがダウンした場合でも、トランザクションの内容は確実に処理されることを保証します。

参照

- 「バックアップとデータリカバリ」『SQL Anywhere サーバ データベース管理』

参照整合性

データを挿入、更新、削除したときに、SQL Anywhere により、データ内に一般的なエラーがあるかどうか自動的に検査が行われます。このタイプの妥当性検査は、データベースに含まれるテーブル内やテーブル間のデータの整合性を検査するため、「参照整合性の確保」と呼ばれます。

参照

- 「エンティティ整合性と参照整合性」850 ページ

INSERT を使用したデータの追加

INSERT 文を使用してデータベースにローを追加します。INSERT 文には 2 つの形式があります。VALUES キーワードまたは SELECT 文を使用できます。

値を使う INSERT

VALUES キーワードで新しいロー内の一部、またはすべてのカラムの値を指定します。VALUES キーワードを使った INSERT 文の簡略バージョンの構文は、次のとおりです。

INSERT [**INTO**] *table-name* [(*column-name*, ...)]
VALUES (*expression*, ...)

SELECT * によるクエリを実行した結果に表示される順で、テーブルの各カラムに値を入力すると、カラム名のリストを省略できます。

SELECT からの INSERT

INSERT 文に SELECT 文を使用して、1 つ以上のテーブルから値を引き出せます。データを挿入するテーブルに多数のカラムがある場合は、WITH AUTO NAME を使用して構文を簡単にすることもできます。WITH AUTO NAME を使用する場合、カラム名を指定する必要があるのは、INSERT 文と SELECT 文の両方ではなく、SELECT 文のみです。SELECT 文の名前には、カラム参照かエイリアスの式を指定してください。

SELECT 文を使用した INSERT 文の簡略バージョンの構文は、次のとおりです。

INSERT [**INTO**] *table-name*
[**WITH AUTO NAME**] *select-statement*

参照

- 「INSERT 文」『SQL Anywhere サーバ SQL リファレンス』

ローの全カラムへの値の挿入

INSERT 文を使用して、ローの全カラムに値を挿入できます。

前提条件

そのテーブルに対する INSERT 権限が必要です。ON EXISTING UPDATE 句が指定された場合は、そのテーブルに対する UPDATE 権限も必要になります。

元の CREATE TABLE 文にあるカラム名と同じ順序で値を入力します。

値をカッコで囲みます。

すべての文字データを一重引用符で囲みます。

追加する各ローには、別の INSERT 文を使用します。

♦ タスク

● 各カラムの値を含む INSERT 文を実行します。

結果

指定された値は、新規ローの各カラムに挿入されます。

例

次の INSERT 文は、Departments テーブルに新規のローを追加して、そのローのすべてのカラムに値を指定します。

```
INSERT INTO GROUPO.Departments
VALUES ( 702, 'Eastern Sales', 902 );
```

参照

● 「INSERT 文」『SQL Anywhere サーバ SQL リファレンス』

指定カラムへの値の挿入

指定されたカラムと指定されていないカラムに挿入される値

値は INSERT 文で指定された内容に従ってローに挿入されます。カラムに値が入力されていない場合、挿入される値はカラム設定 (NULL 値やデフォルト値を挿入するなど) によって異なります。挿入操作が失敗して、エラーが返される場合もあります。次の表は、挿入される値 (該当する場合) とカラム設定に基づいた結果を示しています。

挿入される値	NULL 入力可	NULL 入力不可	NULL 入力可 (DEFAULT 指定)	NULL 入力不可 (DEFAULT 指定)	NULL 入力不可 (DEFAULT AUTOINCREMENT または DEFAULT [UTC] TIMESTAMP 指定)
<なし>	NULL	SQL エラー	デフォルト値	デフォルト値	デフォルト値
NULL	NULL	SQL エラー	NULL	SQL エラー	デフォルト値
指定された値	指定された値	指定された値	指定された値	指定された値	指定された値

デフォルトでは、テーブルの作成時にカラム定義で NOT NULL と明示的に記述しないかぎり、カラムに NULL を使用できます。allow_nulls_by_default オプションを使用して、デフォルトを変更できます。また、ALTER TABLE 文を使用して、特定のカラムに NULL 値を許可するかどうかを変更できます。

制約を使用したカラムデータの制限

カラムまたはドメインに対する制約を作成できます。制約によって、追加可能または追加不可能なデータの種類が決定されます。

NULL の明示的挿入

NULL を入力すると、カラムに NULL を明示的に挿入できます。NULL を引用符で囲まないでください。囲むと文字列として扱われます。たとえば、次の文は DepartmentHeadID カラムに NULL を明示的に挿入します。

```
INSERT INTO Departments
VALUES ( 703, 'Western Sales', NULL );
```

デフォルトを使用した値の指定

カラムが値を受け取らなくても、ローを挿入したら常にデフォルト値が自動的に挿入されるように、カラムを定義できます。これを設定するには、カラムにデフォルト値を設定します。

参照

- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「allow_nulls_by_default オプション」『SQL Anywhere サーバ データベース管理』
- 「テーブルとカラム制約」843 ページ
- 「カラムデフォルト」836 ページ

指定カラムへの値の挿入

カラムとその値を指定するだけで、ローにあるカラムにデータを追加できます。

前提条件

そのテーブルに対する INSERT 権限が必要です。ON EXISTING UPDATE 句が指定された場合は、そのテーブルに対する UPDATE 権限も必要になります。

内容と備考

指定するカラムの順序はテーブル内のカラムの順序と一致させる必要はなく、挿入する値を指定する順序と一致させてください。

そのカラムリストにない他のすべてのカラムは、NULL 入力可、またはデフォルト値を持つように定義します。デフォルト値の入ったカラムを省略すると、そのカラムにはデフォルトが挿入されます。

♦ タスク

● INSERT INTO 文を実行して、指定カラムにデータを追加します。

たとえば次の文は、DepartmentID と DepartmentName の 2 つのカラムだけにデータを追加します。

```
INSERT INTO GROUPO.Departments ( DepartmentID, DepartmentName )
VALUES ( 703, 'Western Sales' );
```

DepartmentHeadID にはデフォルトの値はありませんが、NULL. は受け入れます。そのため、NULL は自動的にそのカラムに割り当てられます。

結果

データが、指定されたカラムに挿入されます。

参照

● 「INSERT 文」『SQL Anywhere サーバ SQL リファレンス』

SELECT を使用した新しいローの追加

1 つ以上のテーブルから他のテーブルへ値を引き出すには、INSERT 文に SELECT 句を使用します。SELECT 句により、ローにあるカラムの一部またはすべてに値を挿入できます。

一部のカラムだけに対する値の挿入は、既存のテーブルから値を取得する場合に便利です。その場合、UPDATE 文を使用して他のカラムの値を追加できます。

テーブルにある一部のカラムにのみ値を挿入するには、その前にデフォルト値が存在するか、または値を挿入しないカラムに NULL を指定しているか確認します。こうしないと、エラーが表示されます。

あるテーブルから他のテーブルにローを挿入する場合、2 つのテーブルは互換性のある構造にします。すなわち、一致するカラムを、同じデータ型または SQL Anywhere が自動的に変換できるデータ型にします。

一部のカラムへのデータ挿入

SELECT 文を使用して、VALUES 句を使用する場合と同様に、ローにある一部のカラムにのみデータを追加できます。INSERT 句でデータを追加するカラムを指定するだけです。

同じテーブルからのデータの挿入

同じテーブルにある他のデータに基づいたテーブルへ、データを挿入できます。本質的には、これはローの全部または一部をコピーすることを意味します。

たとえば、Products テーブルに既存の製品に基づく新しい製品を挿入できます。次の文は Products テーブルに新しい項目の Extra Large Tee Shirt (Tank Top、V-neck、Crew Neck の各種) を追加します。ID 番号は既存サイズのシャツ番号に 30 を加えます。

```
INSERT INTO Products
SELECT ID + 30, Name, Description,
   'Extra large', Color, 50, UnitPrice, NULL
FROM Products
WHERE Name = 'Tee Shirt';
```

例

2 つのテーブルでカラムの順序が同じである場合、どちらのテーブルのカラム名も指定する必要はありません。たとえば、NewProducts というテーブルのスキーマが Products テーブルのスキーマと同じであり、NewProducts テーブルには Products テーブルに追加する製品情報の一部のローが含まれているとします。この場合、次の文を実行できます。

```
INSERT Products
SELECT *
FROM NewProducts;
```

ドキュメントとイメージの挿入

ドキュメントまたはイメージをデータベースに格納するには、ファイルの内容を変数に読み込んで、その変数を INSERT 文の値として指定するアプリケーションを記述します。

テーブルへのファイル内容の挿入には、xp_read_file システムプロシージャも使用できます。ファイルの内容を Interactive SQL から挿入する場合や、完全なプログラミング言語を提供しない他の環境から挿入する場合に、このプロシージャを使用すると便利です。

例

この例では、テーブルを作成してテーブルのカラムにイメージを挿入します。これらの手順は Interactive SQL から実行します。

1. イメージを保持するテーブルを作成します。

```
CREATE TABLE Pictures
( C1 INT DEFAULT AUTOINCREMENT PRIMARY KEY,
  Filename VARCHAR(254),
  Picture LONG BINARY );
```

2. データベースサーバの現在の作業ディレクトリにある *portrait.gif* の内容をテーブルに挿入します。

```
INSERT INTO Pictures ( Filename, Picture )
VALUES ( 'portrait.gif',
  xp_read_file( 'portrait.gif' ) );
```

参照

- 「xp_read_file システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「xp_read_file での OPENXML の使用」673 ページ
- 「準備文の概要」『SQL Anywhere サーバ プログラミング』
- 「BLOB の考慮事項」『SQL Anywhere サーバ データベース管理』
- 「SET 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「INSERT 文」『SQL Anywhere サーバ SQL リファレンス』

高度：挿入されたローに対するディスク割り付け

SQL Anywhere は可能な場合ローを連続して格納する

新しいローは、データベースファイルのページサイズよりも小さい場合、常に単一のページに保管されます。現在のページに新しいローを保存する十分な空き領域がない場合、SQL Anywhere はローを新しいページに書き込みます。たとえば、新しいローが 600 バイトの領域を必要とするときに、ページの一部が埋まっていて 500 バイトしか使用できない場合、SQL Anywhere は新しいページにローを配置します。

ディスク上のテーブルページがさらに連続するように、SQL Anywhere はテーブルページを 8 ページのブロック単位で割り付けます。たとえば、1 ページの割り付けが必要な場合は、8 ページを割り付け、必要な 1 ページをブロックに挿入してから、ブロックの残りの 7 ページを埋めます。また、空きページビットマップを使用して、DB 領域内で連続するページブロックを検索します。次に、64 KB のグループを読み込み、ビットマップを使用して関連ページを検索し、逐次スキャンを実行します。このため、逐次スキャンの効率が高まります。

SQL Anywhere ではローをどのような順序でも保存できる

SQL Anywhere はページの領域を検索し、受け取った順序でローを挿入します。それぞれのローを 1 ページに割り当てますが、テーブル内で選択したロケーションは、ローが挿入された順序と一致しない場合があります。たとえば、データベースサーバは、長いローを隣接して保管するためにページを新しくする必要があることがあります。次のローが短い場合、そのローは前のページの空いているロケーションに配置されます。

すべてのテーブルのローには順序が付いていません。ローを受け取ったり処理したりする順序が重要である場合、SELECT 文で ORDER BY 句を使用し、結果に順序を付けます。テーブル内のローの順序に依存するアプリケーションは、警告なしに失敗することがあります。

テーブルのローを特定の順序にすることが頻繁に必要になる場合は、クエリの ORDER BY 句で指定したカラムにインデックスを作成することを検討してください。

NULL カラム用の領域は予約されない

デフォルトでは、SQL Anywhere は、ローを挿入する場合は必ず、ローを作成時の値で格納するために必要な領域だけを予約します。NULL 値、またはテキスト文字列などの拡張する可能性のあるフィールドを格納するための追加領域は予約しません。

SQL Anywhere に対して、テーブルの作成時に PCTFREE オプションを使用して領域を予約するよう強制できます。

一度挿入されたローの識別子は不変

一度ページ上にホーム位置が割り当てられると、ローは決してそのページから移動しません。更新によりローのいずれかの値が変更され、割り当てられているページに適合しなくなると、ローは分割され、追加情報が別のページに挿入されます。

この特性には注意が必要です。特に、ローの挿入時に SQL Anywhere は追加領域を許可しないためです。たとえば、大量の空のローを 1 つのテーブルに挿入して、UPDATE 文を使用して一度に 1 カラムずつ値を入力するとします。この結果、1 つのローにあるほとんどすべての値は別々のページに保存されます。1 つのローからすべての値を取り出すために、データベースサーバは複数のディスクページを読み込まなければならない場合があります。この簡単な操作にかなりの時間がかかることになります。

新しいローへのデータ配置を挿入時に行うことを検討してください。ローは一度挿入されれば、データを保持するのに十分な領域を確保します。

データベースファイルは縮小しない

データベースにローを挿入してから削除すると、SQL Anywhere はローが使用していた領域を自動的に再利用します。したがって、SQL Anywhere は別のローが以前に使用していた領域に新しいローを挿入します。

SQL Anywhere は、各ページの空き領域の量の記録を保持しています。新しいローを挿入するよう要求されると、まず既存のページの領域の記録を検索します。既存のページで十分な領域を見つけると、新しいローをそのページに配置し、必要であればそのページの内容を再編成します。見つけられない場合には、新しいページを開始します。

いくつかのローを削除し、空き領域を使用できる程度の小さなローを新しく挿入しなかった場合、時間の経過とともにデータベース内の情報がまばらになることがあります。その場合は、テーブルを再ロードするか、REORGANIZE TABLE 文を使用してテーブルの断片化を解除できます。

参照

- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REORGANIZE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

UPDATE によるデータの変更

UPDATE 文では、変更するローと、それらのローで特定のカラムの新しい値として使用する式を指定します。

UPDATE 文を使用して、テーブルにある単一のロー、ローのグループ、またはすべてのローを変更できます。他のデータ操作文 (INSERT、MERGE、DELETE) と異なり、UPDATE 文では複数のテーブル内のローを同時に修正することもできます。すべての場合で、UPDATE 文の実行はアトミックです。すべてのローがエラーなく修正されるか、すべて修正されません。たとえば、修正される値の 1 つが誤ったデータ型であったり、新しい値によって CHECK 制約違反が発生したりした場合、UPDATE は失敗し、操作全体がロールバックされます。

UPDATE 構文

UPDATE 文の構文の簡略バージョンは次のとおりです。

UPDATE *table-name*
SET *column_name* = *expression*
WHERE *search-condition*

会社 Newton Ent.(SQL Anywhere サンプルデータベースの Customers テーブル内の会社) が Einstein, Inc. に吸収される場合は、次のような文を使用して会社名を更新できます。

```
UPDATE Customers
SET CompanyName = 'Einstein, Inc.'
WHERE CompanyName = 'Newton Ent.';
```

WHERE 句で任意の式を使用できます。入力された会社名のスペルがわからなければ、次のような文を使用して Newton という会社名を更新してみます。

```
UPDATE Customers
SET CompanyName = 'Einstein, Inc.'
WHERE CompanyName LIKE 'Newton%';
```

探索条件は更新されるカラムを参照する必要はありません。Newton Entertainments の会社 ID は 109 です。ID 値はテーブルのプライマリキーなので、次の文を使用して正しいローを確実に更新できます。

```
UPDATE Customers
SET CompanyName = 'Einstein, Inc.'
WHERE ID = 109;
```

ヒント
また、InteractiveSQL で結果セットからのローを修正することもできます。

SET 句

SET 句は、更新されるカラムとその新しい値を指定します。WHERE 句は、更新する必要があるローを決定します。WHERE 句がない場合、指定されたすべてのローのカラムが SET 句の値によって更新されます。

SET 句で指定できる式は、定数リテラル、ホストまたは SQL 変数、サブクエリ、CURRENT TIMESTAMP などの特別値、別のテーブルから引き出された式の値、またはこれらの組み合わせです。また、SET 句で DEFAULT を指定すると、そのベーステーブルのカラムのデフォルト値を指定できます。式のデータ型が修正するカラムのデータ型と異なる場合は、可能であれば、データベースサーバによって式はカラムの型に自動的に変換されます。変換できない場合は、データ例外が発生し、UPDATE 文は失敗します。

SET 句は、カラム値の修正以外に、変数の値を設定するために使用できます。次の例は、テーブル T の更新以外に、変数 @var に値を代入します。

```
UPDATE T
SET @var = expression1, col1 = expression2
WHERE...;
```

これは、SELECT 文とそれに続く UPDATE を連続して実行することとほぼ同じです。

```
SELECT @var = expression1
FROM T
WHERE... ;
UPDATE T SET col1 = expression2
WHERE...;
```

UPDATE 文での変数の代入の利点は、書き込みロックを保持したまま、変数の値を文の実行の中で設定できる点です。これにより、他の接続から同時に実行される更新アクティビティによる予期しない値の代入を防ぎます。

WHERE 句

WHERE 句は、UPDATE 文で指定したテーブルまたはテーブル式の直積に *search-condition* を適用することによって、更新されるローを指定します。たとえば、次の文は "One Size Fits All" を "Extra Large Tee Shirt" に書き換えます。

```
UPDATE Products
SET Size  = 'Extra Large'
WHERE Name = 'Tee Shirt'
  AND Size = 'One Size Fits All';
```

複雑な UPDATE 文

より複雑な形式の UPDATE 文では、ジョインによる更新やその他のタイプのテーブル式が可能です。

たとえば、UPDATE 文の構文 1 は次のとおりです。

UPDATE [*row-limitation*] *table-name*
SET *set-item*[, ...]
FROM *table-expression* [, ...]]
[**WHERE** *search-condition*]
[**ORDER BY** *expression* [**ASC** | **DESC**] , ...]
[**OPTION(** *query-hint*, ... **)**]

この形式の UPDATE 文のセマンティックは、最初に各 *table-expression* からのローのすべての組み合わせで構成される結果セットを計算し、次に WHERE 句の *search-condition* を適用してから、結果ローを ORDER BY 句を使用して順序付けします。この計算の結果は、修正されるローのセットになります。各 *table-expression* は、ベーステーブル、ビュー、派生テーブルのジョインで構成できます。この構文では、他のテーブルにあるカラムの値を持つ 1 つまたは複数のテーブルを更新できます。クエリオプティマイザによって操作が並べ替えられ、UPDATE 文のより効率的な実行方式が作成される場合があります。

ベーステーブルのローが、修正されるローのセット内に複数回現れる場合、ローの新しい値が各操作の試行で異なっていれば、そのローは複数回更新されます。BEFORE ROW UPDATE トリガが存在する場合は、トリガの UPDATE OF *column-list* 句に応じて、個々のロー操作のたびに BEFORE ROW UPDATE トリガが起動されます。AFTER ROW UPDATE トリガも、トリガの UPDATE OF *column-list* 句に応じて各ロー操作とともに起動されますが、ローの値が実際に変更される場合のみです。

トリガは、トリガのタイプと各トリガ定義の ORDER 句の値に基づいて、更新されるテーブルごとに起動されます。ただし、UPDATE 文によって複数のテーブルが修正される場合、テーブルが更新される順序は保証されません。

次の例では、Products テーブルに BEFORE ROW UPDATE トリガおよび AFTER STATEMENT UPDATE トリガを作成します。各トリガによって、データベースサーバメッセージウィンドウにメッセージが出力されます。

```
CREATE OR REPLACE TRIGGER trigger0
BEFORE UPDATE
ON Products
REFERENCING OLD AS old_product NEW AS new_product
FOR EACH ROW
BEGIN
  PRINT ('BEFORE row: PK value: ' || old_product.ID || ' New Price: ' || new_product.UnitPrice );
END;

CREATE OR REPLACE TRIGGER trigger1
AFTER UPDATE
ON Products
REFERENCING NEW AS new_product
FOR EACH STATEMENT
BEGIN
  DECLARE @pk INTEGER;
  DECLARE @newUnitPrice DECIMAL(12,2);
  DECLARE @err_notfound EXCEPTION FOR SQLSTATE VALUE '02000';
  DECLARE new_curs CURSOR FOR
   SELECT ID, UnitPrice FROM new_product;
  OPEN new_curs;
  LoopGetRow:
    LOOP
      FETCH NEXT new_curs INTO @pk, @newUnitPrice;
      IF SQLSTATE = @err_notfound THEN
        LEAVE LoopGetRow
      END IF;
      PRINT ('AFTER stmt: PK value: ' || @pk || ' Unit price: ' || @newUnitPrice );
  END LOOP LoopGetRow;
  CLOSE new_curs
END;
```

次に、Products テーブルと SalesOrderItems テーブルのジョインに UPDATE 文を実行し、2001 年 4 月 1 日以降に出荷された、大きい注文が少なくとも 1 つある製品を 5% 割引するとします。

```
UPDATE Products p JOIN SalesOrderItems s ON (p.ID = s.ProductID)
SET p.UnitPrice = p.UnitPrice * 0.95
WHERE s.ShipDate > '2001-04-01' AND s.Quantity >= 72;
```

データベースサーバメッセージウィンドウには、次の情報が表示されます。

```
BEFORE row: PK value: 700 New Price: 14.25
BEFORE row: PK value: 302 New Price: 13.30
BEFORE row: PK value: 700 New Price: 13.54
AFTER stmt: PK value: 700 Unit price: 14.25
AFTER stmt: PK value: 302 Unit price: 13.30
AFTER stmt: PK value: 700 Unit price: 13.54
```

メッセージは、Product 700 が 2 回更新されたことを示しています。これは、Product 700 が UPDATE 文の検索条件と一致する 2 つの異なる注文に含まれていたためです。更新の重複は、BEFORE ROW トリガと AFTER STATEMENT トリガの両方からわかります。各ロー操作により、各トリガ呼び出しの「OLD」 値および「NEW」 値はそれに応じて変更されます。AFTER STATEMENT トリガでは、REFERENCING 句によって形成されるテンポラリテーブル内のローの順序は、ローが修正された順序と一致しない場合があり、それらのローの正確な順序は保証されません。

更新の重複のために、Product 700 の UnitPrice は 2 回割引され、最初の $15.00 から、$14.25 ではなく $13.54 (9.75% の割引が発生) に下がりました。この意図しない結果を回避するには、ジョインではなく EXISTS サブクエリを使用して、各 Product ローの修正が多くても 1 回になることを保証するように UPDATE 文を作成します。書き換えられた UPDATE 文では、EXISTS サブクエリと、FROM 句を許可する代替 UPDATE 文の構文の両方が使用されます。

```
UPDATE Products AS p
SET p.UnitPrice = p.UnitPrice * 0.95
FROM Products AS p
WHERE EXISTS(
   SELECT *
   FROM SalesOrderItems s
   WHERE p.ID = s.ProductID
     AND s.ShipDate > '2001-04-01'
     AND s.Quantity >= 72);
```

UPDATE と制約違反

実行中に UPDATE 文が参照整合性制約に違反した場合、文の動作は wait_for_commit オプションの設定によって制御されます。wait_for_commit オプションが Off に設定されていて、参照制約違反が発生した場合、UPDATE 文の結果は直ちに自動的にロールバックされ、エラーメッセージが表示されます。wait_for_commit オプションが On に設定されている場合、UPDATE 文によって発生した参照整合性制約違反は一時的に無視され、接続によって COMMIT が実行されるときにチェックされます。

修正されるベーステーブルに、プライマリキー、UNIQUE 制約、またはユニークインデックスがある場合、UPDATE 文のローごとの実行によって一意性制約違反が発生する場合があります。たとえば、テーブル T のプライマリキーカラムの値すべてを増分する次のような UPDATE 文を発行する場合があります。

```
UPDATE T SET PKcol = PKcol + 1;
```

UPDATE 文の実行中に一意性違反が発生した場合、データベースサーバによって次の処理が自動的に行われます。

1. 修正されるローの新しい値と古い値が、修正されるベーステーブルと同じスキーマを持つテンポラリテーブルにコピーされます。

2. 元のローがベーステーブルから削除されます。この削除操作の結果として DELETE トリガは起動されません。

UPDATE 文の実行中に、正常に更新されるローと一時的に削除されるローは、評価の順序によって異なり、保証されません。弱い独立性レベル (独立性レベル 0、1、または 2) で実行されている他の接続からの SQL 要求の動作は、これらの一時的に削除されるローの影響を受ける場合があります。修正されるテーブルの BEFORE または AFTER ROW トリガには、トリガの REFERENCING 句に従って各ローの古い値と新しい値が渡されますが、修正されるテーブルに対して ROW トリガによって個別の SQL 文が発行される場合、テンポラリテーブルに保持されているローは失われます。

UPDATE 文による各ローの修正が完了した後、テンポラリテーブルに保持されているローはベーステーブルに戻されます。一意性違反がまだ発生する場合は、UPDATE 文全体がロールバックされます。テンポラリテーブルに保持されているすべてのローがベーステーブルに正常に再挿入された場合にのみ、AFTER STATEMENT トリガが起動されます。

修正されるベーステーブルが ON DELETE CASCADE、ON DELETE SET NULL、ON DELETE DEFAULT、ON UPDATE CASCADE、ON UPDATE SET NULL、ON UPDATE DEFAULT などの参照整合性制約アクションのターゲットである場合、ローを一時的に格納するためにデータベースサーバによって保持テーブルは使用されません。

参照

- 「UPDATE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「Interactive SQL での結果セット」『SQL Anywhere サーバ データベース管理』
- 「ansi_update_constraints オプション」『SQL Anywhere サーバ データベース管理』
- 「DELETE 文または UPDATE 文の整合性検査」857 ページ
- 「更新時のロック」899 ページ

INSERT によるデータの変更

INSERT 文の ON EXISTING 句を使用して、テーブル内の既存のローを (プライマリキールックアップに基づいて) 新しい値で更新できます。この句は、プライマリキーが設定されたテーブルでのみ使用できます。プライマリキーがないテーブル、またはプロキシテーブルでこの句を使用すると、構文エラーになります。

ON EXISTING 句を指定すると、サーバは各入力ローに対してプライマリキールックアップを実行します。対応するローが存在しない場合は、新しいローが挿入されます。すでにテーブルに存在するローに対しては、次の操作を選択できます。

- 重複するキー値に対してエラーを生成する。ON EXISTING 句を指定しない場合は、これがデフォルトの動作です。
- 入力ローを無視して、エラーを生成しない。
- 既存のローを入力ロー内の値で更新する。

参照

- 「INSERT 文」『SQL Anywhere サーバ SQL リファレンス』

DELETE によるデータの削除

DELETE 文は、次のような単純な形式になっています。

DELETE [**FROM**] *table-name*
WHERE *column-name* = *expression*

次のような、より複雑な形式も使用できます。

DELETE [**FROM**] *table-name*
FROM *table-list*
WHERE *search-condition*

WHERE 句

WHERE 句を使用して削除するローを指定します。WHERE 句がないと、DELETE 文によってテーブルのすべてのローが削除されます。

FROM 句

DELETE 文の 2 番目に位置する FROM 句には、テーブルからデータを選択し、最初に指定されたテーブルから一致するデータを削除する、特別な働きがあります。FROM 句で選択したローに、削除の条件を指定します。

参照

- 「DELETE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「Interactive SQL での結果セット」『SQL Anywhere サーバ データベース管理』

例

この例では、SQL Anywhere のサンプルデータベースを使用します。この文を実行するには、wait_for_commit オプションを On に設定してください。次の文は現在の接続に関してのみ、この操作を実行します。

```
SET TEMPORARY OPTION wait_for_commit = 'On';
```

これによって、外部キーに参照されるプライマリキーが含まれるローでも削除できますが、対応する外部キーも削除しないかぎり、COMMIT は許可されません。

次のビューでは、製品と販売した製品の値を表示します。

```
CREATE VIEW ProductPopularity as
SELECT  Products.ID,
  SUM( Products.UnitPrice * SalesOrderItems.Quantity )
  AS "Value Sold"
FROM  Products JOIN SalesOrderItems
ON Products.ID = SalesOrderItems.ProductID
GROUP BY Products.ID;
```

このビューを使用して、売り上げが 20,000 ドル未満の製品を Products テーブルから削除できます。

```
DELETE
FROM Products
FROM Products NATURAL JOIN ProductPopularity
WHERE "Value Sold" < 20000;
```

次の ROLLBACK 文を実行して、データベースのこれらの変更をキャンセルしてください。

```
ROLLBACK;
```

ヒント
また、InteractiveSQL の結果セットから、データベーステーブルのローを削除することもできます。

参照

- 「Interactive SQL での結果セット」『SQL Anywhere サーバ データベース管理』

テーブルからのすべてのローの削除

TRUNCATE TABLE 文を使用すると、テーブルにあるすべてのローを簡単に削除できます。この方法は、条件を指定しない DELETE 文よりもすばやく処理できます。これは、DELETE 文が各変更のログを取るのに対し、TRUNCATE 文では個々のローの削除が記録されないためです。

DROP TABLE 文を実行しないかぎり、TRUNCATE TABLE 文によって空になったテーブルの定義は、インデックスや他の関連オブジェクトとともにデータベースに残されます。

他のテーブルに参照整合性制約で参照するローがあると、TRUNCATE TABLE 文を使用できません。外部テーブルからローを削除するか、外部テーブルをトランケートしてからプライマリテーブルをトランケートします。

ベーステーブルのトランケート操作またはバルクロードの実行操作を行うことにより、インデックス (通常のインデックスまたはテキストインデックス) 内と従属マテリアライズドビュー内のデータが古くなります。最初にインデックスや従属したマテリアライズドビューのデータをトランケートし、INPUT 文を実行してから、インデックスとマテリアライズドビューを再構築または再表示してください。

TRUNCATE TABLE 構文

TRUNCATE TABLE 文の構文は次のとおりです。

TRUNCATE TABLE *table-name*

たとえば、SalesOrders テーブルのすべてのデータを削除するには、次のように入力します。

```
TRUNCATE TABLE SalesOrders;
```

TRUNCATE TABLE 文はテーブルで定義されたトリガを呼び出しません。

次の ROLLBACK 文を実行して、データベースのこれらの変更をキャンセルしてください。

```
ROLLBACK;
```

参照

- 「TRUNCATE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「TRUNCATE TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』

SQL のダイアレクトと互換性

この項では、Transact-SQL の互換性と、他の SQL ソフトウェアにはない SQL Anywhere の機能について説明します。

SQL Anywhere は、SQL-92 を中心とする米国連邦情報処理規格刊行物 (FIPS PUB) 127 に全面的に準拠しています。また、若干の例外がありますが、ISO/IEC JTC 1/SC 32 9075-2008 の 第 9 部に記載されているように、ISO/ANSI SQL2008 の中核となる仕様にも準拠しています。準拠に関する情報については、SQL Anywhere の各機能のリファレンスマニュアルを参照してください。

SQL Flagger を使用した SQL 準拠のテスト

SQL Anywhere では、データベースサーバと SQL プリプロセッサ (sqlpp) で、ベンダー拡張であるか、特定の ISO/ANSI SQL 標準に準拠しないか、Ultra Light でサポートされていない SQL 文を特定できます。この機能は、SQL Flagger と呼ばれます。ISO/ANSI 9075-1999 SQL 標準 (このマニュアルでは SQL/1999 と呼びます) のオプションの SQL 言語機能 F812 として最初に導入されました。SQL Flagger によって、アプリケーション開発者は SQL 言語の特定のサブセットに違反する SQL 言語要素を特定できます。SQL Flagger を使用すると、SQL 標準のコア機能への準拠、またはコア機能とオプション機能の組み合わせへの準拠も確認できます。また、SQL Flagger は、SQL Anywhere で Ultra Light アプリケーションのプロトタイプを作成するときに、使用している SQL が Ultra Light でサポートされていることを確認するためにも使用できます。

空間データサポートは SQL/MM 標準の第 3 部 (ISO/IEC 13249-3) として標準化されているため、空間機能、操作、構文は SQL Flagger ではサポートされておらず、ベンダー拡張として通知されます。

SQL Flagger では、SQL 文の構文とセマンティックの両要素が分析の対象ですが、準拠はコンパイル時に静的にチェックされます。構文の準拠のテスト例として、INSERT 文にオプションの INTO キーワードがない場合を考えます (たとえば、INSERT Products VALUES(...))。これは、SQL Anywhere による SQL 言語の文法上の拡張です。ANSI SQL/2008 標準では INTO キーワードの使用が必須と定められているので、INTO キーワードがない INSERT 文は、ベンダー拡張として通知されます。ただし、Ultra Light アプリケーションでは INTO キーワードはオプションです。

キーのジョインもベンダー拡張として通知されます。ON 句を使用しないで、キーワード JOIN を使用する場合、キージョインがデフォルトで使用されます。キージョインは、既存の外部キー関係を使用して、テーブルをジョインします。キージョインは Ultra Light ではサポートされていません。たとえば、次のクエリは、Products テーブルと SalesOrderItems テーブル間の暗黙的なジョイン条件を指定しています。このクエリは、SQL Flagger でベンダー拡張として通知されます。

```
SELECT * FROM Products JOIN SalesOrderItems;
```

SQL Flagger 機能は、SQL 文の実行に依存しません。すべての通知論理は単に静的なコンパイル時の処理として行われます。

参照

- 「SQLFLAGGER 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「空間標準への準拠」『SQL Anywhere サーバ 空間データサポート』
- 「SQL プリプロセッサ」『SQL Anywhere サーバ プログラミング』
- 「INSERT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「キージョイン」520 ページ

SQL Flagger の起動

SQL Anywhere では、複数の方法で SQL Flagger を起動し、単一の SQL 文、または SQL 文のバッチをチェックできます。

- **SQLFLAGGER 関数**　SQLFLAGGER 関数は、文字列引数として渡された単一の SQL 文またはバッチが特定の SQL 標準に準拠しているかどうかを分析します。単一の文またはバッチは解析されますが、実行はされません。

- **sa_ansi_standard_packages システムプロシージャ**　sa_ansi_standard_packages システムプロシージャは、単一の文またはバッチで、ANSI SQL/2008、SQL/2003、または SQL/1999 国際標準のオプションの SQL 言語機能またはパッケージが使用されていないかどうかを分析します。単一の文またはバッチは解析されますが、実行はされません。

- **sql_flagger_error_level オプションと sql_flagger_warning_level オプション**　sql_flagger_error_level オプションと sql_flagger_warning_level オプションは、文が接続に対して準備または実行されると、SQL Flagger を起動します。文がオプション設定 (特定の ANSI 標準または Ultra Light) に準拠しない場合、文はエラー (SQLSTATE 0AW03) で終了するか、警告 (SQLSTATE 01W07) を返します。どちらの処理を行うかはオプション設定で決まります。文が準拠する場合は、文の実行が正常に続行されます。

- **SQL プリプロセッサ (sqlpp)**　SQL プリプロセッサ (sqlpp) には、Embedded SQL アプリケーション内の静的 SQL 文をコンパイル時に通知する機能があります。この機能は、Ultra Light アプリケーションを開発するときに特に便利です。この機能で、SQL 文に Ultra Light との互換性があることを確認できます。

参照

- 「バッチ」107 ページ
- 「SQLFLAGGER 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「sql_flagger_error_level オプション」『SQL Anywhere サーバ データベース管理』
- 「sql_flagger_warning_level オプション」『SQL Anywhere サーバ データベース管理』
- 「sa_ansi_standard_packages システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「SQL プリプロセッサ」『SQL Anywhere サーバ プログラミング』
- 「SQL プリプロセッサ」『SQL Anywhere サーバ プログラミング』

標準と互換性

データベースサーバと SQL プリプロセッサで使用される通知機能は、ANSI/ISO SQL/2008 国際標準の第 1 部 (フレームワーク) で定義されている SQL Flagger 機能に従っています。SQL Flagger では、SQL 言語構成の準拠を決定するときに、次の ANSI SQL 標準がサポートされます。

- SQL/1992 の Entry レベル、Intermediate レベル、Full レベル
- SQL/1999 のコアと SQL/1999 のオプションパッケージ
- SQL/2003 のコアと SQL/2003 のオプションパッケージ
- SQL/2008 のコアと SQL/2008 のオプションパッケージ

注意
SQL Flagger では、SQL/1992 (全レベル) はサポートされなくなりました。

SQL Flagger では、Ultra Light SQL に準拠しない文も特定できます。たとえば、Ultra Light では、スキーマオブジェクトの CREATE と ALTER 機能が限られています。

SQL 文はすべて SQL Flagger で分析できます。ただし、スキーマオブジェクトを作成または変更するほとんどの文 (テーブル、インデックス、マテリアライズドビュー、パブリケーション、サブスクリプション、プロキシテーブルを作成する文を含む) は、ANSI SQL 標準のベンダー拡張なので、準拠しないと通知されます。

SET OPTION 文とそのオプション要素は、どの SQL 標準に対しても、Ultra Light との互換性についても、準拠しないと通知されません。

参照

- 「Ultra Light SQL 言語の要素」『Ultra Light データベース管理とリファレンス』
- 「SET OPTION 文」『SQL Anywhere サーバ SQL リファレンス』

他の SQL ソフトウェアとは異なる SQL Anywhere の機能

SQL Anywhere には、豊富な SQL 機能があります。それらの機能には、ローごとのトリガ、文ごとのトリガ、INSTEAD OF トリガ、SQL ストアドプロシージャ、ユーザ定義関数、RECURSIVE UNION クエリ、共通テーブル式、テーブル関数、LATERAL 派生テーブル、統合全文検索、Window 集合関数、正規表現検索、XML サポート、マテリアライズドビュー、スナップショットアイソレーション、参照整合性などがあります。この項では、他の SQL データベースソフトウェアとは異なる、SQL Anywhere でサポートされている特定の機能について説明します。

日付

SQL Anywhere には日付、時刻、タイムスタンプのデータ型があり、年、月、日、時、分、秒、小数点以下の秒が含まれます。日付フィールドへの挿入または更新、および日付フィールド間の比較については、フリーフォーマットの日付がサポートされています。

また、日付に関しては以下の演算が可能です。

- **日付 + 整数**　指定された値の日数を日付に加えます。
- **日付 - 整数**　指定された値の日数を日付から引きます。
- **日付 - 日付**　2 つの日付間の日数の差を計算します。
- **日付 + 時刻**　日付と時刻からタイムスタンプを作成します。

SQL Anywhere は、INTERVAL 日付タイプはサポートしていません。これは SQL/2008 標準の SQL 言語機能 F052 です。ただし、SQL Anywhere には、DATEADD などの日付と時刻の演算に使用できる数多くの関数があります。

エンティティ整合性と参照整合性

SQL Anywhere は、CREATE TABLE 文および ALTER TABLE 文の PRIMARY KEY 句と FOREIGN KEY 句によってエンティティ整合性と参照整合性の両方をサポートしています。

PRIMARY KEY [**CLUSTERED**] (*column-name* [**ASC** | **DESC**], ...)
[**NOT NULL**] **FOREIGN KEY** [*role-name*]
　　[(*column-name* [**ASC** | **DESC**], ...)]
　REFERENCES *table-name* [(*column-name*, ...)]
　　[**MATCH** [**UNIQUE** | **SIMPLE** | **FULL**]]
　　[**ON UPDATE** [**CASCADE** | **RESTRICT** | **SET DEFAULT** | **SET NULL**]]
　　[**ON DELETE** [**CASCADE** | **RESTRICT** | **SET DEFAULT** | **SET NULL**]]
　　[**CHECK ON COMMIT**] [**CLUSTERED**]

PRIMARY KEY 句では、テーブルのプライマリキーを宣言します。SQL Anywhere はプライマリキーカラムにユニークなインデックスを作成して、宣言されたプライマリキーがユニークであるよう管理します。このインデックスは、次の 2 つの SQL Anywhere 拡張によってカスタマイズできます。

- **CLUSTERED**　CLUSTERED キーワードは、プライマリキーインデックスはクラスタドインデックスであり、したがって、インデックス内の連続したインデックスエントリはテーブルの物理的に隣接するローをポイントすることを示します。
- **ASC | DESC**　プライマリキーインデックスの各インデックスカラムのソートの程度 (昇順または降順) をカスタマイズできます。このカスタマイズを使用して、プライマリキーインデックスのソートの程度が、特定の SQL クエリに必要な、そのクエリの文の ORDER BY 句で指定されたソートの程度に一致するようにできます。

FOREIGN KEY 句は、2 つのテーブルの関係を定義します。その関係は、このテーブルのカラムが他のテーブルのプライマリキーの値を保有することによって確立しています。SQL Anywhere は、参照整合性制約を確保するように定義された FOREIGN KEY ごとに自動的にインデックスを構築します。制約のセマンティックおよびこのインデックスの物理特性は、次のようにカスタマイズできます。

- **CLUSTERED**　CLUSTERED キーワードは、外部キーインデックスはクラスタドインデックスであり、したがって、インデックス内の連続したインデックスエントリは外部テーブルの物理的に隣接するローをポイントすることを示します。
- **ASC | DESC**　外部キーインデックスの各インデックスカラムのソートの程度 (昇順または降順) をカスタマイズできます。外部キーインデックスのソートの程度は、プライマリキーイ

ンデックスのソートの程度と異なる場合があります。ソートの程度のカスタマイズを使用して、外部キーインデックスのソートの程度が、アプリケーションの特定の SQL クエリに必要な、そのクエリの文の ORDER BY 句で指定されたソートの程度に一致するようにできます。

- **MATCH 句**　SQL Anywhere は、MATCH 句をサポートしています。これは SQL/2008 標準の SQL 言語機能 F741 です。また、SQL Anywhere は、MATCH UNIQUE もサポートしています。これは、UNIQUE インデックスを追加する必要なしにプライマリテーブルと外部テーブル間の 1 対 1 の関係を確保します。

ユニークインデックス

SQL Anywhere は、NULL 入力可能のカラムへのユニークインデックス (ユニークセカンダリインデックスとも呼ばれる) の作成をサポートしています。デフォルトで、各インデックスキーはユニークであるか、少なくとも 1 つのカラムで NULL を持つ必要があります。たとえば、2 つのインデックスエントリ ('a'、NULL) と ('a'、NULL) はそれぞれユニークインデックス値と見なされます。SQL Anywhere は、ユニークセカンダリインデックスをサポートしています。このインデックスでは、NULL 値が各ドメインで特別値として扱われます。これは、WITH NULLS NOT DISTINCT 句を使用して行われます。このようなインデックスでは、2 つの値のペア ('a'、NULL) と ('a'、NULL) は重複と見なされます。

ジョイン

SQL Anywhere は、INNER ジョイン、LEFT OUTER ジョイン、RIGHT OUTER ジョイン、FULL OUTER ジョインをサポートしています。明示的ジョイン述部に加えて、SQL Anywhere は、NATURAL ジョインとベンダー拡張 (KEY ジョインとも呼ばれる) をサポートしています。ベンダー拡張では、テーブルの外部キー関係に基づいて暗黙的ジョイン述部を指定します。

CHAR、NCHAR、BINARY データ型

SQL Anywhere 内部では、固定長文字列型と可変長文字列型 (CHAR、NCHAR、または BINARY) は区別されません。SQL Anywhere では、後続ブランクは、このような値がデータベースに挿入されるときに文字列タイプからトランケートされません。SQL Anywhere では、NULL 値と空の文字列は区別されます。デフォルトでは、SQL Anywhere データベースは大文字と小文字を区別しない照合を使用して、大文字と小文字を区別しない文字列比較をサポートしています。SQL Anywhere では、固定長文字列タイプにブランクが埋め込まれることはありません。ブランク埋め込みセマンティックは、各文字列比較の実行中にシミュレートされます。これらのセマンティックは、他の SQL ソフトウェアの文字列比較とは微妙に異なる場合があります。

UPDATE 文

SQL Anywhere は、オプションの SQL 言語機能 T111 を部分的にサポートしています。この機能では、UPDATE 文によってジョインがあるビューを参照できます。また、UPDATE 文と UPDATE WHERE CURRENT OF 文によって、文の SET 句で複数のテーブルを参照できます。さらに、UPDATE 文の FROM 句をジョインおよび派生テーブルを含む任意のテーブル式で構成できます。

SQL Anywhere では、UPDATE、INSERT、MERGE、DELETE の各文を別の SQL 文に派生テーブルとして埋め込むこともできます。このサポートの利点の 1 つは、UPDATE 文によって変更されたローのセットを返すクエリを簡単な方法で構築できることです。

テーブル関数

SQL Anywhere では、ストアドプロシージャの結果セットを文の FROM 句内のテーブルとして参照できます。この機能は、一般にテーブル関数と呼ばれています。テーブル関数は、SQL/2008 標準の SQL 言語機能 T326 です。この標準では、テーブル関数は TABLE キーワードを使用して指定されます。SQL Anywhere では、TABLE キーワードを使用する必要はありません。ストアドプロシージャは FROM 句で直接参照され、オプションで相関名と結果セットのスキーマの指定がプロシージャによって返されます。

次の例では、ストアドプロシージャの結果 ShowCustomerProducts をベーステーブル Products にジョインします。ストアドプロシージャの参照には、WITH 句を使用して、プロシージャの結果のスキーマの明示的な宣言が付随しています。

```
SELECT sp.ident, sp.quantity, Products.name
FROM ShowCustomerProducts( 149 )
    WITH ( ident INT, description CHAR(20), quantity INT ) sp
  JOIN Products ON sp.ident = Products.ID
```

マテリアライズドビュー

SQL Anywhere はマテリアライズドビューをサポートしています。これは、SQL クエリから直接的または間接的に参照できる、事前に計算された結果セットです。SQL Anywhere では、CREATE MATERIALIZED VIEW 文を使用して、即時保存ビューと手動保存ビューの両方を作成できます。他のデータベース製品では、異なる語を使用してこの機能を記述する場合があります。

カーソル

SQL Anywhere は、SQL/2008 標準のオプションの SQL 言語機能 F431 をサポートしています。SQL Anywhere では、明示的に FORWARD ONLY を宣言されないかぎり、すべてのカーソルは双方向にスクロール可能です。アプリケーションでは、FETCH 文によって相対位置または絶対位置を使用するか、ODBC などの他のアプリケーションプログラミングインタフェースによって同等のものを使用して、カーソルによってスクロールできます。

SQL Anywhere は、value-sensitive カーソルとローメンバーシップ sensitive カーソルをサポートしています。INSENSITIVE、KEYSET-DRIVEN、SENSITIVE カーソルなどの一般的にサポートされているカーソルタイプがサポートされます。Embedded SQL を使用しているときは、カーソル位置は FETCH 文上で自由に移動できます。カーソルは現在位置に対して前後させるか、カーソルの最初または最後からレコード数を指定して移動できます。

デフォルトで、Embedded SQL および SQL プロシージャ、ユーザ定義関数、トリガのカーソルは更新可能です。これらのカーソルは、FOR UPDATE 句を使用して明示的に更新可能にできます。ただし、FOR UPDATE 句を単独で指定しても、カーソルの結果セットのローに対するロックは取得されません。結果セットのローが他のトランザクションによって変更されないようにするには、次のいずれかを指定します。

- **FOR UPDATE BY LOCK**　データベースサーバは、この句によって、結果セットのフェッチされたローに対する意図的ローロックを取得します。このロックは長期間のロックであり、トランザクションがコミットまたはロールバックされるまで保持されます。

- **FOR UPDATE BY { VALUES | TIMESTAMP }**　SQL Anywhere データベースサーバは、keyset-driven カーソルを利用して、結果セットをスクロールするときにローが変更または削除されたときにアプリケーションへ通知できるようにします。

エイリアスの参照

SQL Anywhere では、クエリの SELECT リスト内のエイリアスの式を、クエリの他の部分で参照できます。他のほとんどの SQL システムと SQL/2008 標準にはこの機能はありません。たとえば、次のような SQL クエリを指定できます。

```
SELECT column-or-expression AS alias-name
FROM table-reference
WHERE alias-name = expression
```

エイリアスは、SELECT ブロックの任意の場所に使用できます。エイリアスは他の SELECT リスト式にも使用できます。この場合、追加のエイリアスが定義されます。循環的なエイリアスの参照は許可されません。式に指定されたエイリアスが SELECT ブロックのネームスペースにある変数名またはカラム名と同じ場合は、エイリアス定義によってそのカラムまたは変数が遮断されます。ただし、そのような場合は、テーブルによってカラム名を明示的に修飾できます。

スナップショットアイソレーション

SQL Anywhere は、マルチバージョン同時実行性制御 (MVCC) とも呼ばれる、スナップショットアイソレーションをサポートしています。スナップショットアイソレーションをサポートする他の SQL 実装では、書き込み競合、つまり 2 つ以上のトランザクションによる同じローへの同時更新は、COMMIT 時にのみ明らかになります。この場合、通常は最初の COMMIT が優先され、競合に関与するその他のトランザクションはアボートされる必要があります。

SQL Anywhere では、スナップショットトランザクションが ANSI アイソレーションレベルでトランザクション実行と共存できるように、ローへの書き込み操作によってローの書き込みロックが取得されます。このため、SQL Anywhere での書き込み競合ではブロックが発生しますが、正確な動作は BLOCKING と BLOCKING_TIMEOUT の接続オプションによって制御できます。

参照

- 「日付と時刻関数」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「文字データ型」『SQL Anywhere サーバ SQL リファレンス』
- 「キージョイン」520 ページ
- 「FROM 句」『SQL Anywhere サーバ SQL リファレンス』
- 「マテリアライズドビュー」53 ページ
- 「SQL Anywhere のカーソル」『SQL Anywhere サーバ プログラミング』
- 「DECLARE CURSOR 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』
- 「スナップショットアイソレーション」868 ページ

Watcom SQL

SQL Anywhere がサポートしている SQL ダイアレクトは Watcom SQL と呼ばれています。SQL Anywhere は、最初にリリースされた 1992 年当時には Watcom SQL と呼ばれていました。SQL Anywhere がサポートする SQL ダイアレクトについては、現在でも Watcom SQL という用語が使用されています。

SQL Anywhere は、Sybase Adaptive Server Enterprise がサポートする SQL ダイアレクトである Transact-SQL の大部分のサブセットもサポートしています。

参照

● 「Transact-SQL との互換性」640 ページ

Transact-SQL との互換性

SQL Anywhere は、Sybase Adaptive Server Enterprise がサポートする SQL ダイアレクトである Transact-SQL の大部分のサブセットをサポートしています。この項では、SQL Anywhere と Adaptive Server Enterprise 間の SQL の互換性について説明します。

目的

次に、SQL Anywhere の Transact-SQL に対するサポートの目的を示します。

- **アプリケーションの移植性**　アプリケーション、ストアドプロシージャ、バッチファイルの多くは、Adaptive Server Enterprise と SQL Anywhere データベースの両方で使用できるように作成できます。

- **データの移植性**　SQL Anywhere データベースと Adaptive Server Enterprise データベースの間では、最小限の作業でデータの交換とレプリケートができます。

目的は、Adaptive Server Enterprise と SQL Anywhere の両方で動作するアプリケーションを作成することです。既存の Adaptive Server Enterprise アプリケーションの多くは、SQL Anywhere データベースで実行するには多少の変更が必要です。

Transact-SQL のサポート方法

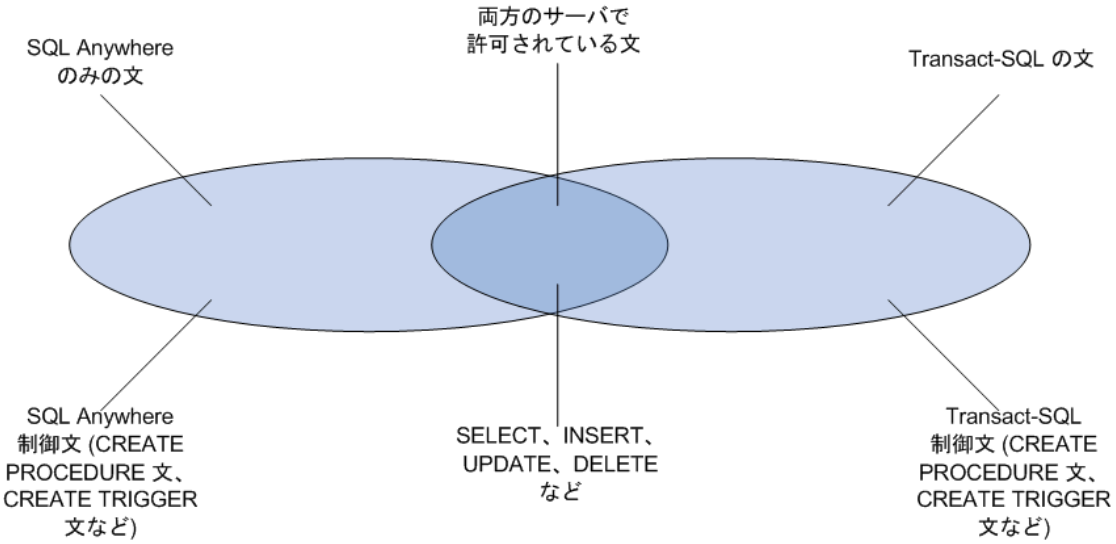
次に、SQL Anywhere の Transact-SQL に対するサポート方法を示します。

●SQL 文の多くは、SQL Anywhere と Adaptive Server Enterprise 間で互換性があります。

●特にプロシージャ、トリガ、バッチで使用されるプロシージャ言語で書かれた文のいくつかでは、これまでの SQL Anywhere のバージョンでサポートされる構文とともに、別の Transact-SQL 文がサポートされます。このような文については、SQL Anywhere では 2 種類の SQL ダイアレクトがサポートされています。これらのダイアレクトは Transact-SQL (Adaptive Server Enterprise のダイアレクト) および Watcom-SQL (SQL Anywhere のダイアレクト) と呼ばれています。

●プロシージャ、トリガ、バッチは Transact-SQL または Watcom-SQL ダイアレクトのどちらでも実行されます。バッチまたはプロシージャ内では、1 つのダイアレクトの制御文だけを通して使用します。たとえば、ダイアレクトごとに異なったフロー制御文があります。

2 つのダイアレクトが重なり合う状況を、次の図に示します。



類似点と相違点

SQL Anywhere は、既存のデータを処理するときの Transact-SQL 言語の要素、関数、文のほとんどをサポートします。たとえば SQL Anywhere は、すべての数値関数、集合関数、日付と時刻関数をサポートし、1 つを除くすべての文字列関数をサポートしています。別の例として、ジョインを使う拡張された DELETE 文と UPDATE 文がサポートされます。

さらに、SQL Anywhere では、Transact-SQL のストアドプロシージャ言語 (CREATE PROCEDURE 文、CREATE TRIGGER 文、制御文など) と、多くの Transact-SQL データのデータ定義言語文がサポートされます。

それぞれの製品にサポートされる構造と設定については設計上の相違点があります。デバイス管理、ユーザ管理、バックアップなどの管理タスクの多くはシステム固有のものです。ここでは、SQL Anywhere は、Transact-SQL のシステムテーブルをビューとして提供しますが、意味のないテーブルにはローがありません。また、SQL Anywhere は、一般的な管理タスクの一部として一連のシステムプロシージャを提供します。

この項では、最初に、相違点の多いシステムレベルの問題を検討します。互換性の高い、ダイアレクトのデータ操作とデータ定義言語については、後で説明します。

Transact-SQL のみ

SQL Anywhere がサポートする SQL 文には、一方のダイアレクトに属し、他方には属さないものがあります。このような 2 つのダイアレクトは、1 つのプロシージャ、トリガ、またはバッチ内で混在させることはできません。たとえば、次の文は SQL Anywhere ではサポートされますが、それは Transact-SQL ダイアレクトのみに属するものとしてです。

●Transact-SQL 制御文の IF と WHILE

●Transact-SQL の EXECUTE 文

●Transact-SQL の CREATE PROCEDURE 文と CREATE TRIGGER 文

- Transact-SQL の BEGIN TRANSACTION 文
- セミコロンで区切られて**いない** SQL 文は、Transact-SQL のプロシージャまたはバッチに属しています。

SQL Anywhere のみ

Adaptive Server Enterprise では、次の文をサポートしません。

- LOOP 制御文および FOR 制御文
- IF と WHILE の SQL Anywhere バージョン
- CALL 文
- SIGNAL 文
- CREATE PROCEDURE 文、CREATE FUNCTION 文、CREATE TRIGGER 文の SQL Anywhere バージョン
- セミコロンで区切られた SQL 文

説明

1 つのプロシージャ、トリガ、またはバッチ内では、2 つのダイアレクトを混在させることはできません。

- Transact-SQL のみの文を、両方のダイアレクトに属する文とともにバッチ、プロシージャ、またはトリガ内に含めることができます。
- Adaptive Server Enterprise によってサポートされない文を、両サーバによってサポートされる文とともに、バッチ、プロシージャ、またはトリガ内に含めることができます。
- Transact-SQL のみの文を、SQL Anywhere のみの文とともにバッチ、プロシージャ、またはトリガに含めることはできません。

Adaptive Server Enterprise のアーキテクチャ

Adaptive Server Enterprise と SQL Anywhere は、それぞれの明確な目的に合わせたアーキテクチャを持つ、相互に補完的な製品です。

この項では、Adaptive Server Enterprise と SQL Anywhere のアーキテクチャの違いについて説明します。また、互換性のあるデータベース管理のために SQL Anywhere に含まれている、Adaptive Server Enterprise に似たツールについても説明します。

サーバとデータベース

サーバとデータベースの関係は、Adaptive Server Enterprise と SQL Anywhere では異なります。

Adaptive Server Enterprise では各データベースはサーバ内に存在し、各サーバは複数のデータベースを持つことができます。ユーザはサーバに対するログイン権限を持っている場合、サーバに接続できます。サーバに接続すると、ユーザはサーバ上のパーミッションを持つ各データベースを使用できます。master データベースに保持されている、システム全体に適用されるシステムテーブルは、サーバ上のすべてのデータベースに共通な情報を含んでいます。

SQL Anywhere に master データベースがない

SQL Anywhere では、Adaptive Server Enterprise の master データベースに対応するレベルはありません。その代わり、それぞれのデータベースが独立したエンティティであり、自分のシステムテーブルを持っています。ユーザは、サーバに対してではなく 1 つのデータベースに対する接続権限を持ちます。ユーザが接続する場合、個々のデータベースに対する接続となります。master データベースレベルで維持されているシステム全体にわたる一連のシステムテーブルはありません。SQL Anywhere の各データベースサーバは、複数のデータベースを動的にロード、アンロードできます。ユーザはそれぞれのデータベースに対する独立した接続を維持できます。

SQL Anywhere は、Transact-SQL のサポートと Open Server のサポートに対して、Adaptive Server Enterprise に似た方法でタスクを実行するツールを提供します。たとえば、SQL Anywhere は、Adaptive Server Enterprise の sp_addlogin システムプロシージャの実装を提供し、同等の処理、つまりデータベースへのユーザの追加を実行します。

ファイル操作文

SQL Anywhere では、Transact-SQL 文 DUMP DATABASE と LOAD DATABASE を使用したバックアップとリストアはサポートされていません。SQL Anywhere には、構文が異なる BACKUP DATABASE 文と RESTORE DATABASE 文があります。

参照

- 「Open Server としての SQL Anywhere」『SQL Anywhere サーバ データベース管理』

デバイス管理

SQL Anywhere と Adaptive Server Enterprise は、用途の違いを反映して、異なったモデルを使用してデバイスとディスク領域を管理します。Adaptive Server Enterprise は多種多様な Transact-SQL 文を使用する包括的なリソース管理スキームを作成しますが、SQL Anywhere は独自のリソースを自動的に管理し、そのデータベースは通常のオペレーティングシステムファイルです。

SQL Anywhere は、DISK INT、DISK MIRROR、DISK REFIT、DISK REINIT、DISK REMIRROR、DISK UNMIRROR などの Transact-SQL DISK 文をサポートしません。

参照

- 「データベースファイルのタイプ」『SQL Anywhere サーバ データベース管理』

デフォルトとルール

SQL Anywhere は、Transact-SQL の CREATE DEFAULT 文や CREATE RULE 文はサポートしません。CREATE DOMAIN 文を使用すると、デフォルトとルール (いわゆる、CHECK 条件) がドメ

インの定義に組み込まれるので、Transact-SQL の CREATE DEFAULT 文と CREATE RULE 文に対して、類似した機能性が与えられます。

SQL Anywhere では、ドメインはデフォルト値とそれに対応した CHECK 条件を持つことができ、その CHECK 条件はそのデータ型に基づいて定義されたすべてのカラムに適用されます。ドメインを作成するには、CREATE DOMAIN 文を使用します。

デフォルト値とルール、または CHECK 条件は、個々のカラムについて、CREATE TABLE 文または ALTER TABLE 文を使用して定義できます。

Adaptive Server Enterprise では、CREATE DEFAULT 文は名前付きデフォルトを作成します。このデフォルトは、特定のカラムにバインドすることによって、カラムのデフォルト値として使用できます。また、sp_bindefault システムプロシージャを使用してドメインのカラムにバインドすると、ドメインのすべてのカラムのデフォルト値として使用できます。CREATE RULE 文は、名前のついたルールを作成します。このルールは、特定のカラムにバインドすることによって、さまざまなカラムのドメイン定義に使用でき、データ型にバインドすると、そのドメインのすべてのカラムのルールとして使用できます。ルールをデータ型やカラムにバインドするには、sp_bindrule システムプロシージャを使用します。

参照

- 「CREATE DOMAIN 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「探索条件」『SQL Anywhere サーバ SQL リファレンス』

システムテーブル

SQL Anywhere には、独自のシステムテーブルのほかに、Adaptive Server Enterprise のシステムテーブルに対応する箇所をシミュレートするビューがあります。

SQL Anywhere のシステムテーブルは各データベース内に格納されていますが、Adaptive Server Enterprise のシステムテーブルの場合、一部は各データベース内、一部は master データベース内に格納されています。SQL Anywhere のアーキテクチャは master データベースを含みません。

Adaptive Server Enterprise では、データベース所有者 (ユーザ dbo) がシステムテーブルを所有します。SQL Anywhere では、システム所有者 (ユーザ SYS) がシステムテーブルを所有します。ユーザ dbo は、SQL Anywhere が提供する Adaptive Server Enterprise と互換性のあるシステムビューを所有します。

参照

- 「Transact-SQL 互換のビュー」『SQL Anywhere サーバ SQL リファレンス』

管理上の役割

Adaptive Server Enterprise は SQL Anywhere より管理上の役割が充実しています。Adaptive Server Enterprise では、複数のログインアカウントに役割を付与でき、1 つのアカウントが複数の役割を処理できますが、独特の役割のセットもあります。

Adaptive Server Enterprise の役割

Adaptive Server Enterprise の独特の役割には次のものがあります。

- **システム管理者**　特定のアプリケーションに関連していない一般的な管理タスクを行い、あらゆるデータベースオブジェクトにアクセスできます。
- **システムセキュリティ担当者**　Adaptive Server Enterprise のセキュリティが問題となるタスクを行いますが、データベースオブジェクトには特別のパーミッションを持ちません。
- **データベース所有者**　自分が所有者であるデータベース内のオブジェクトに対して、完全な権限を持ちます。また、データベースにユーザを追加したり、データベース内でオブジェクトの作成や文の実行を行うのに必要な権限を他のユーザに付与したりできます。
- **データ定義文**　CREATE TABLE や CREATE VIEW などの特定のデータ定義文に対する権限をユーザに与え、ユーザがデータベースオブジェクトを作成できるようにします。
- **オブジェクト所有者**　各データベースオブジェクトには所有者があり、そのオブジェクトにアクセスする権限を他のユーザに与えることができます。オブジェクトの所有者は、自動的にそのオブジェクトに対するすべての権限を持ちます。

- データベース管理者 (SYS_AUTH_DBA_ROLE 互換ロール) は、Adaptive Server Enterprise のデータベース所有者のように、所有するデータベース内のすべてのオブジェクト (SYS が所有するオブジェクトは除く) に対して完全な権限を持ち、他のユーザにデータベース内でのオブジェクト作成と文実行に必要な権限を付与できます。デフォルトのデータベース管理者は、ユーザ DBA です。
- リソースロール (SYS_AUTH_RESOURCE_ROLE 互換ロール) は、ユーザがデータベース内でオブジェクトを作成するのを許可します。これは Adaptive Server Enterprise で個々の CREATE 文にパーミッションを付与する代わりのものです。
- SQL Anywhere のオブジェクト所有者は、Adaptive Server Enterprise の所有者と同じです。オブジェクト所有者は権限を付与することも含め、そのオブジェクトに関するすべての権限を自動的に持ちます。

Adaptive Server Enterprise と SQL Anywhere の両方に含まれるデータにスムーズにアクセスするために、適切な権限を持つユーザ ID をデータベースに作成し、このユーザ ID からオブジェクトを作成してください。同じユーザ ID を両方の環境で使用すると、オブジェクト名と修飾子が 2 つのデータベースで同一になり、互換性のあるアクセスが可能になります。

ユーザとグループ

次に、SQL Anywhere がサポートする、Adaptive Server Enterprise のユーザとグループ管理のシステムプロシージャを示します。

システムプロシージャ	説明
sp_addlogin	Adaptive Server Enterprise では、ユーザをサーバに追加する。SQL Anywhere では、ユーザをデータベースに追加します。
sp_adduser	Adaptive Server Enterprise と SQL Anywhere の両方で、ユーザをデータベースに追加する。Adaptive Server Enterprise では、これは sp_addlogin とは異なるタスクですが、SQL Anywhere では同じです。
sp_addgroup	グループをデータベースに追加する。
sp_changegroup	ユーザをグループに追加するか、ユーザをあるグループから別のグループに移動する。
sp_droplogin	Adaptive Server Enterprise では、ユーザをサーバから削除する。SQL Anywhere では、ユーザをデータベースから削除します。
sp_dropuser	ユーザをデータベースから削除する。
sp_dropgroup	グループをデータベースから削除する。

Adaptive Server Enterprise では、ログイン ID はサーバ全体に適用されます。SQL Anywhere では、ユーザは個々のデータベースに属します。

データベースオブジェクトに適用される権限

個々のデータベースオブジェクトに対する権限を付与する GRANT 文と REVOKE 文は、Adaptive Server Enterprise と SQL Anywhere でよく似ています。どちらの文も、データベースのテーブルとビューに対する SELECT、INSERT、DELETE、UPDATE、REFERENCES 権限を付与でき、データベースのテーブルの選択したカラムに対する UPDATE 権限を付与できます。どちらも、ストアドプロシージャに対する EXECUTE 権限を付与できます。

たとえば、次の文は Adaptive Server Enterprise と SQL Anywhere の両方で有効です。

```
GRANT INSERT, DELETE
ON Employees
TO MARY, SALES;
```

この文は、Employees テーブルで INSERT 文と DELETE 文を使用するのに必要な権限を MARY というユーザと SALES グループに付与します。

SQL Anywhere と Adaptive Server Enterprise の両方で、WITH GRANT OPTION 句は、権限を付与されるユーザがそれを他のユーザに付与することを許可します。ただし、SQL Anywhere は WITH GRANT OPTION を GRANT EXECUTE 文で使用することを許可しません。SQL Anywhere では、WITH GRANT OPTION はユーザにのみ指定できます。グループのメンバーは、グループに付与されている WITH GRANT OPTION を継承しません。

データベース全体に適用される権限

Adaptive Server Enterprise と SQL Anywhere は、データベース全体に適用される権限に異なったモデルを使用します。SQL Anywhere は DBA ロールを使用して、データベース内での完全な権限をユーザに許可します。Adaptive Server Enterprise のシステム管理者は、この権限をサーバ上のすべてのデータベースに対して使用できます。しかし、SQL Anywhere データベース上での DBA ロールは、Adaptive Server Enterprise のデータベース所有者のパーミッションとは異なります。Adaptive Server Enterprise のデータベース所有者は、他のユーザが所有するオブジェクトに対するパーミッションを得るには、Adaptive Server Enterprise の SETUSER 文を使用してください。

参照

- 「Adaptive Server Enterprise のシステムプロシージャとカタログプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「ユーザとグループ」645 ページ

Transact-SQL と互換性のあるデータベース

データベースを作成するとき、または既存のデータベースで作業中の場合にデータベースを再構築するときは、適切なオプションを選択すると、SQL Anywhere と Adaptive Server Enterprise の相違点のいくつかを解除できます。他の相違点は、SQL Anywhere では SET TEMPORARY OPTION 文、Adaptive Server Enterprise では SET 文を使用して接続レベルオプションを設定することで制御できます。

大文字と小文字を区別するデータベースの作成

デフォルトでは、Adaptive Server Enterprise データベースでの文字列比較は大文字と小文字を区別しますが、SQL Anywhere では大文字と小文字を区別しません。

SQL Anywhere を使用して Adaptive Server Enterprise と互換性のあるデータベースを構築する場合は、大文字と小文字を区別するオプションを選択します。

- Sybase Central を使用している場合は、このオプションは**データベース作成ウィザード**にあります。
- dbinit ユーティリティを使用している場合は、-c オプションを指定します。
- CREATE DATABASE 文を使用する場合は、CASE RESPECT 句を指定します。

比較の後続ブランクを無視

SQL Anywhere を使用して Adaptive Server Enterprise と互換性のあるデータベースを構築するときには、比較するときに後続ブランクを無視するオプションを選択します。

- Sybase Central を使用している場合は、このオプションは**データベース作成ウィザード**にあります。
- dbinit ユーティリティを使用している場合は、-b オプションを指定します。
- CREATE DATABASE 文を使用する場合は、BLANK PADDING ON 句を指定します。

このオプションを選択すると、Adaptive Server Enterprise と SQL Anywhere では次の 2 つの文字列を等しいと見なします。

```
'ignore the trailing blanks   '
'ignore the trailing blanks'
```

このオプションを選択しないと、SQL Anywhere ではこの 2 つの文字列は異なっていると見なします。

このオプションを選択すると、クライアントアプリケーションでフェッチした文字列にブランクが埋め込まれます。

古いシステムビューの削除

SQL Anywhere の古いバージョンでは 2 つのシステムビューを使用していましたが、互換性を確保しようとすると、それらの名前は Adaptive Server Enterprise のシステムビューと矛盾します。その 2 つのビューは SYSCOLUMNS と SYSINDEXES です。Open Client または JDBC のインタフェースを使用している場合は、これらのビューを除外してデータベースを作成します。この処理には dbinit -k オプションを使用します。

データベースの作成時にこのオプションを使用しなかった場合、SELECT * FROM SYSCOLUMNS; を実行すると、「SQLE_AMBIGUOUS_TABLE_NAME」というエラーが発生します。

参照

- 「SET OPTION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「データベースオプション」『SQL Anywhere サーバ データベース管理』

Transact-SQL と互換性のあるデータベースの作成 (Sybase Central の場合)

Sybase Central を使用して Transact-SQL と互換性のあるデータベースを作成できます。

前提条件

デフォルトでは、SERVER OPERATOR システム権限が必要です。-gu データベースサーバオプションを使用すると、必要な権限を変更できます。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[ツール]** » **[SQL Anywhere 16]** » **[データベースの作成]** をクリックします。
3. ウィザードの指示に従います。

   [追加設定の指定] 画面で、**[Adaptive Server Enterprise のエミュレート]** をクリックして、**[次へ]** をクリックします。

4. ウィザードの残りの指示に従います。

結果

Transact-SQL と互換性のあるデータベースが作成されます。このデータベースは、ブランクが埋め込まれ、大文字と小文字が区別されます。また、SYS.SYSCOLUMNS と SYS.SYSINDEXES のシステムビューはありません。

参照

- 「Transact-SQL と SQL/2008 の互換性オプション」『SQL Anywhere サーバ データベース管理』

Transact-SQL と互換性のあるデータベースの作成 (コマンドラインの場合)

コマンドラインを使用して Transact-SQL と互換性のあるデータベースを作成できます。この項の残りの部分では、設定する必要があるオプションについて説明します。

前提条件

この作業を実行するための前提条件はありません。

♦ タスク

- 次の dbinit コマンドを実行します。

```
dbinit -b -c -k -dba DBA,sql db-name.db
```

このコマンドでは、-b はデータベースにブランクを埋め込み、-c はデータベースで大文字と小文字が区別されるようになり、-k は SYS.SYSCOLUMNS と SYS.SYSINDEXES のシステムビューが作成されないようにします。

結果

Transact-SQL と互換性のあるデータベースが作成されます。

参照

- 「初期化ユーティリティ (dbinit)」『SQL Anywhere サーバ データベース管理』
- 「Transact-SQL と SQL/2008 の互換性オプション」『SQL Anywhere サーバ データベース管理』

Transact-SQL と互換性のあるデータベースの作成 (SQL の場合)

SQL を使用して Transact-SQL と互換性のあるデータベースを作成できます。

前提条件

デフォルトでは、SERVER OPERATOR システム権限が必要です。-gu データベースサーバオプションを使用すると、必要な権限を変更できます。

♦ タスク

1. SQL Anywhere データベースに接続します。

2. Interactive SQL で次の文を入力します。

```
CREATE DATABASE 'db-name.db'
DBA USER 'DBA' DBA PASSWORD 'sql'
ASE COMPATIBLE
CASE RESPECT
BLANK PADDING ON;
```

この文で ASE COMPATIBLE 句を使用すると、SYS.SYSCOLUMNS と SYS.SYSINDEXES のシステムビューが作成されないようになります。

結果

Transact-SQL と互換性のあるデータベースが作成されます。このデータベースは、ブランクが埋め込まれ、大文字と小文字が区別されます。また、SYS.SYSCOLUMNS と SYS.SYSINDEXES のシステムビューはありません。

参照

- 「CREATE DATABASE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「Transact-SQL と SQL/2008 の互換性オプション」『SQL Anywhere サーバ データベース管理』

Transact-SQL 互換性のオプション

SQL Anywhere のデータベースオプションの設定は、SET OPTION 文を使用します。データベースオプションの設定値のいくつかは Transact-SQL の動作を規定します。

allow_nulls_by_default オプションの設定

デフォルトでは、Adaptive Server Enterprise はカラムに対して NULL を許可すると定義しないかぎり、新しいカラムに NULL を許可しません。SQL Anywhere はデフォルトで新しいカラムに NULL を認めますが、これは SQL/2008 ISO 標準と互換性があります。

Adaptive Server Enterprise を SQL/2008 互換で動作させるには、sp_dboption システムプロシージャを使用して allow_nulls_by_default オプションを true に設定します。

SQL Anywhere を Transact-SQL 互換で動作させるには、allow_nulls_by_default オプションを Off に設定します。これを行うには、次のように SET OPTION 文を使用します。

```
SET OPTION PUBLIC.allow_nulls_by_default = 'Off';
```

quoted_identifier オプションの設定

Adaptive Server Enterprise のデフォルトの識別子と文字列の処理は、SQL/2008 ISO 標準に一致する SQL Anywhere とは異なります。

quoted_identifier オプションは、Adaptive Server Enterprise と SQL Anywhere の両方で使用できます。識別子と文字列が互換性を持って処理されるように、両方のデータベースでこのオプションが同じ値に設定されていることを確認します。

SQL/2008 に準拠させるには、Adaptive Server Enterprise と SQL Anywhere の両方で quoted_identifier オプションを On に設定します。

Transact-SQL に準拠させるには、Adaptive Server Enterprise と SQL Anywhere の両方で quoted_identifier オプションを Off に設定します。この設定では、キーワードと同じように複数の同一の識別子を二重引用符で囲んで使用することはできません。quoted_identifier を Off に設定しないで、アプリケーション内の SQL 文で使用しているすべての文字列を二重引用符ではなく一重引用符で囲む方法もあります。

string_rtruncation オプションの設定

Adaptive Server Enterprise と SQL Anywhere では、string_rtruncation オプションがサポートされています。このオプションは、INSERT 文または UPDATE 文字列がトランケートされたときのエラーメッセージの表示を制御します。各データベースのオプション設定は同じ値にしてください。

参照

- 「互換性オプション」『SQL Anywhere サーバ データベース管理』
- 「quoted_identifier オプション」『SQL Anywhere サーバ データベース管理』
- 「string_rtruncation オプション」『SQL Anywhere サーバ データベース管理』

大文字と小文字の区別

データベースにおける大文字と小文字の区別は、次のことに関連があります。

- **データ**　データの大文字と小文字を区別するかしないかは、インデックスなどに反映されます。
- **識別子**　識別子には、テーブル名、カラム名などがあります。
- **パスワード**　SQL Anywhere データベースのパスワードは常に大文字と小文字が区別されます。

データの大文字と小文字の区別

SQL Anywhere では、比較におけるデータの大文字と小文字の区別は、データベース作成時に決定します。SQL Anywhere データベースは、デフォルトでは、データは常に入力されたとおりの大文字と小文字で保持されていますが、比較において大文字と小文字を区別しません。

Adaptive Server Enterprise での大文字と小文字の区別は、Adaptive Server Enterprise システムにインストールされているソート順によって異なります。大文字と小文字の区別は、シングルバイト文字セットの場合は、Adaptive Server Enterprise のソート順を再設定して変更できます。

識別子の大文字と小文字の区別

SQL Anywhere は大文字と小文字を区別する識別子をサポートしません。Adaptive Server Enterprise では、識別子の大文字と小文字の区別はデータの大文字と小文字の区別に従います。SQL Anywhere データベースのデフォルトのユーザ ID は、DBA です。

Adaptive Server Enterprise では、ドメイン名の大文字と小文字を区別します。SQL Anywhere では、Java データ型を例外として、ドメイン名の大文字と小文字を区別しません。

パスワードの大文字と小文字の区別

SQL Anywhere のパスワードでは、常に大文字と小文字が区別されます。DBA ユーザ ID のデフォルトのパスワードは、小文字の **sql** です。

Adaptive Server Enterprise では、ユーザ ID とパスワードの大文字と小文字の区別は、サーバの大文字と小文字の区別に従います。

オブジェクト名の互換性

データベースオブジェクトは、一定の「ネームスペース」内にユニークな名前を持っていなければなりません。ネームスペース外では重複した名前も許可されます。データベースオブジェクトによっては、Adaptive Server Enterprise と SQL Anywhere で異なるネームスペースを持っています。

Adaptive Server Enterprise は、トリガ名について SQL Anywhere よりも制限の多いネームスペースを持っています。トリガ名はデータベース内でユニークでなければなりません。互換性のある SQL を作成するには、Adaptive Server Enterprise の、データベース内でユニークなトリガ名を必要とする制限を採用してください。

Transact-SQL の特殊な TIMESTAMP カラムとデータ型

SQL Anywhere は Transact-SQL の特殊な TIMESTAMP カラムをサポートします。TIMESTAMP カラムは、TSEQUAL システム関数とともに、ローが更新されたかどうかチェックするのに使用されます。

timestamp の 2 つの意味
SQL Anywhere は正確な日付と時間の情報を持つ TIMESTAMP データ型を持っています。これは、Transact-SQL の特殊な TIMESTAMP カラムとデータ型とは異なります。

SQL Anywhere での Transact-SQL の TIMESTAMP カラムの作成

Transact-SQL の TIMESTAMP カラムを作成するには、(SQL Anywhere の) TIMESTAMP データ型とタイムスタンプのデフォルト設定値を持つカラムを作成します。このカラムにはどのような名前も付けられますが、通常は timestamp が使われます。

次に、Transact-SQL の TIMESTAMP カラムを含む CREATE TABLE 文の例を示します

```
CREATE TABLE tablename (
  column_1 INTEGER,
```

```
  column_2 TIMESTAMP DEFAULT TIMESTAMP
);
```

次の ALTER TABLE 文は、SalesOrders テーブルに Transact-SQL の TIMESTAMP カラムを追加します。

```
ALTER TABLE SalesOrders
ADD timestamp TIMESTAMP DEFAULT TIMESTAMP;
```

Adaptive Server Enterprise ではカラム名が timestamp で、データ型の指定がなければ自動的に TIMESTAMP データ型が与えられます。SQL Anywhere ではデータ型を明示的に割り当てる必要があります。

TIMESTAMP カラムのデータ型

Adaptive Server Enterprise では、TIMESTAMP カラムはドメインの NULL を許容する VARBINARY(8) として扱われます。SQL Anywhere では、TIMESTAMP カラムは日付と時間からなる TIMESTAMP データ型として扱われ、時間は秒以下小数点 6 桁からなります。

後で更新するためにテーブルからフェッチするときは、TIMESTAMP 値がフェッチされる先の変数は、カラムの記述に従わなければなりません。

Interactive SQL では、ローの値の違いを調べるために timestamp_format オプションを設定することが必要な場合があります。次の文は、秒の小数点以下 6 桁すべてを表示するように timestamp_format オプションを設定します。

```
SET OPTION timestamp_format='YYYY-MM-DD HH:NN:SS.SSSSSS';
```

小数点以下 6 桁の数字すべてが表示されなければ、TIMESTAMP カラムの値は等しいように見えても、実際は等しくないことがあります。

更新時の TSEQUAL の使用

TSEQUAL システム関数を使えば、TIMESTAMP カラムが更新されたかどうかがわかります。

アプリケーションが TIMESTAMP カラムを変数に SELECT したとします。選択されたロー内の 1 つの UPDATE が送信されたときに、アプリケーションは TSEQUAL 関数を使用してそのローが修正されたかどうかをチェックできます。TSEQUAL 関数は、テーブルの TIMESTAMP 値を SELECT で取得した TIMESTAMP 値と比較します。これらの値が同じなら、変更はありません。値が違う場合は、ローが SELECT の実行以降に変更されています。次に例を示します。

```
CREATE VARIABLE old_ts_value TIMESTAMP;

SELECT timestamp into old_ts_value
FROM publishers
WHERE pub_id = '0736';

UPDATE publishers
SET city = 'Springfield'
WHERE pub_id = '0736'
AND TSEQUAL(timestamp, old_ts_value);
```

特殊な IDENTITY カラム

IDENTITY カラムは、送り状番号や従業員番号のような連続番号を格納します。このカラムは、自動生成されます。IDENTITY カラムの値はテーブルの各ローをユニークに識別します。

Adaptive Server Enterprise では、データベースの各テーブルは IDENTITY カラムを 1 つだけ持つことができます。データ型は numeric、位取りは 0 (ゼロ)、NULL 値は許可されません。

SQL Anywhere では、IDENTITY カラムはカラムのデフォルトとして設定されます。連続番号でない値も、INSERT 文を使用してカラムに明示的に挿入できます。Adaptive Server Enterprise は、identity_insert オプションが **on** に設定されないかぎり、identity カラムへの INSERT を許可しません。SQL Anywhere では、NOT NULL プロパティを設定する必要があります。また、1 つのカラムのみが IDENTITY カラムであることを確認する必要があります。SQL Anywhere では、IDENTITY カラムにどの数値データ型でも使用できます。パフォーマンスのためには、整数データ型の使用をおすすめします。

SQL Anywhere では、IDENTITY カラムとカラムのデフォルト設定の AUTOINCREMENT は同じです。

IDENTITY カラムを作成するには、次の CREATE TABLE 構文を使用します。ただし、*n* にはテーブルに挿入される可能性のある最大ロー数の値を保持できる大きさである必要があります。

```
CREATE TABLE table-name (
  ...
  column-name numeric(n,0) IDENTITY NOT NULL,
  ...
)
```

@@identity を使用する IDENTITY カラム値の検索

初めてローを 1 つテーブルに挿入すると、IDENTITY カラムに 1 という値が割り当てられます。その後は挿入するたびに、カラムの値が 1 ずつ増えます。最後に IDENTITY カラムに挿入した値は、@@identity グローバル変数として使用できます。

参照

- 「@@identity グローバル変数」『SQL Anywhere サーバ SQL リファレンス』

互換性のある SQL 文

この項では、複数のデータベース管理システムで使用する SQL を記述するための一般的なガイドラインについて説明します。また、SQL 文レベルでの Adaptive Server Enterprise と SQL Anywhere 間の互換性に関する問題についても説明します。

移植可能な SQL を記述するための一般的なガイドライン

複数のデータベース管理システムで使用するために SQL を記述する場合は、SQL 文をできるかぎり明示的にします。指定した SQL 文を複数のサーバがサポートしている場合でも、デフォルトの動作が各システムで同じであると仮定するのは間違っていることもあります。

SQL Anywhere では、データベースサーバと SQL プリプロセッサ (sqlpp) で、ベンダー拡張であるか、特定の ISO/ANSI SQL 標準に準拠しないか、Ultra Light でサポートされていない SQL 文を特定できます。この機能は SQL Flagger と呼ばれます。

次に、互換性のある SQL を記述するときの一般的なガイドラインを示します。

- デフォルトの動作を使用せずに、使用可能なオプションをすべて含めます。
- 演算子の優先順位のデフォルトが同じであるとするのではなく、文中の実行の順序をカッコを使用して明確にします。
- Adaptive Server Enterprise に移植できるように、変数名には @ をプレフィクスとして付ける Transact-SQL の規則に従います。
- プロシージャ、トリガ、バッチでは、BEGIN 文の直後で変数とカーソルを宣言します。Adaptive Server Enterprise では、プロシージャ、トリガ、バッチ内のどこででも宣言できますが、SQL Anywhere では、BEGIN 文の直後で行う必要があります。
- データベース内の識別子として、Adaptive Server Enterprise または SQL Anywhere の予約語を使用しないでください。
- 大きいネームスペースを想定します。たとえば、各インデックスにはユニークな名前を持たせるようにします。

参照

- 「SQL Flagger を使用した SQL 準拠のテスト」633 ページ

Transact-SQL と互換性のあるテーブル

SQL Anywhere は、制約とデフォルト定義をデータ型定義内にカプセル化できるドメインをサポートします。Adaptive Server Anywhere はまた、CREATE TABLE 文の明示的なデフォルトと CHECK 条件もサポートします。ただし、名前付きデフォルトはサポートしません。

NULL

SQL Anywhere と Adaptive Server Enterprise は、NULL の処理に関して異なる点があります。Adaptive Server Enterprise では、NULL は値として処理されることがあります。

たとえば、Adaptive Server Enterprise のユニークインデックスには、NULL 値があるローと、NULL 値以外は同一のローを同時に格納できません。SQL Anywhere では、このようなローでもユニークインデックスに格納できます。

デフォルトでは、Adaptive Server Enterprise のカラムは NOT NULL に設定されていますが、SQL Anywhere のデフォルト設定値は NULL です。この設定は、allow_nulls_by_default オプションを使用して制御できます。NULL または NOT NULL を明示的に指定して、データ定義文を転送可能にします。

テンポラリテーブル

テンポラリテーブルを作成するには、CREATE TABLE 文のテーブル名の先頭にシャープ記号 (#) を付けます。このようなテンポラリテーブルは、SQL Anywhere の宣言テンポラリテーブルであり、現在の接続でしか使用できません。

テーブルの物理的配置は、Adaptive Server Enterprise と SQL Anywhere では実行方法が異なります。SQL Anywhere は **ON** *segment-name* 句をサポートしますが、*segment-name* は SQL Anywhere の DB 領域を参照します。

参照

- 「SQL Flagger を使用した SQL 準拠のテスト」633 ページ
- 「Transact-SQL 互換性のオプション」650 ページ
- 「DECLARE LOCAL TEMPORARY TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

Transact-SQL と互換性のあるクエリ

SQL Anywhere と Adaptive Server Enterprise の両方のデータベースで実行されるクエリの記述方法には、基準が 2 つあります。

- クエリ中のデータ型、式、探索条件が互換性を持つこと。

- 文そのものの構文が互換性を持つこと。

この項では、クエリ中のデータ型、式、探索条件が互換性を持つことを前提としています。quoted_identifier オプションが OFF に設定され、それが Adaptive Server Enterprise のデフォルトの設定値であって、SQL Anywhere のデフォルトの設定値ではない、という例を想定します。

Transact-SQL ダイアレクトの SQL Anywhere の実装は Watcom SQL ダイアレクトのクエリ式構文のほとんどをサポートしていますが、Adaptive Server Enterprise はそれらの SQL 構成の一部をサポートしていません。Transact-SQL クエリで、SQL Anywhere は次の SQL 構成をサポートします。

- 識別子を指定する逆引用符 `、二重引用符 "、角カッコ []
- UNION、EXCEPT、INTERSECT クエリ式
- 派生テーブル
- テーブル関数
- 全文検索用の CONTAINS テーブル式
- REGEXP、SIMILAR、IS DISTINCT FROM、CONTAINS 述部
- ユーザ定義 SQL 関数または外部関数
- LEFT、RIGHT、FULL 外部ジョイン
- GROUP BY ROLLUP、CUBE、GROUPING SETS
- TOP N START AT M
- Window 集合関数および統計分析関数や線形回帰関数などの他の分析関数

要約すると、SQL Anywhere の Transact-SQL ダイアレクトは次の構成をサポートします。

構文

query-expression:
{ *query-expression* **EXCEPT** [**ALL**] *query-expression*
| *query-expression* **INTERSECT** [**ALL**] *query-expression*
| *query-expression* **UNION** [**ALL**] *query-expression*
| *query-specification* }
[**ORDER BY** { *expression* | *integer* }
[**ASC** | **DESC**], ...]
[**FOR READ ONLY** | *for-update-clause*]
[**FOR XML** *xml-mode*]

query-specification:
SELECT [**ALL** | **DISTINCT**] [*cursor-range*] *select-list*
[**INTO** *#temporary-table-name*]
[**FROM** *table-expression*, ...]
[**WHERE** *search-condition*]
[**GROUP BY** *group-by-term*, ...]
[**HAVING** *search-condition*]
[**WINDOW** *window-specification*, ...]

パラメータ

select-list:
*table-name.**
| *
| *expression*
| *alias-name* = *expression*
| *expression as identifier*
| *expression as string*

table-expression: 「FROM 句」『SQL Anywhere サーバ SQL リファレンス』 を参照。

group-by-term: 「GROUP BY 句」『SQL Anywhere サーバ SQL リファレンス』 を参照。

for-update-clause: FOR UPDATE または FOR READ ONLY 句、SELECT 文『SQL Anywhere サーバ SQL リファレンス』 を参照。

xml-mode: 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』 を参照。

alias-name:
identifier | '*string*' | "*string*" | `*string*`

cursor-range:
{ **FIRST** | **TOP***constant-or-variable* } [**START AT** *constant-or-variable*]

Transact-SQL-table-reference:
[*owner* .]*table-name* [[**AS**] *correlation-name*]
[(**INDEX** *index_name* [**PREFETCH** *size*][**LRU** | **MRU**])]

注意

- FROM 句の Watcom SQL 構文に加えて、SQL Anywhere は特定の Adaptive Server Enterprise テーブルヒントの Transact-SQL 構文をサポートします。テーブル参照について、*Transact-SQL-table-reference* は INDEX ヒントキーワードと PREFETCH、MRU、LRU キャッシュヒントをサポートします。PREFETCH、MRU、LRU は SQL Anywhere では無視されます。

- SQL Anywhere は、GROUP BY 句に含まれていないカラムの参照を可能にする GROUP BY 句への Transact-SQL 拡張をサポートしません。

  SQL Anywhere は Transact-SQL GROUP BY ALL 構成もサポートしません。

- SQL Anywhere は比較演算子 *= と =* を使用して、Transact-SQL 外部ジョイン構成のサブセットをサポートします。

- SQL Anywhere の Transact-SQL ダイアレクトは派生テーブル内に埋め込まれている場合を除き、共通テーブル式をサポートしません。このため、SQL Anywhere の Transact-SQL ダイアレクトは再帰 UNION クエリをサポートしません。この機能が必要な場合は、Use the Watcom SQL ダイアレクトを使用してください。

- テーブル仕様のパフォーマンスパラメータ部分は解析されますが、効果はありません。

- HOLDLOCK キーワードは SQL Anywhere によってサポートされます。HOLDLOCK の場合、データページが不要になっても共有ロックが解放されないため、指定されたテーブルやビューの共有ロックの制限はより厳しくなります。クエリは、HOLDLOCK が指定されたテーブルで独立性レベル 3 で実行されます。

- HOLDLOCK オプションは、それが指定されたテーブルまたはビューにだけ、しかも、そのオプションが使用された文で定義されたトランザクションの間だけ適用されます。独立性レベルを 3 に設定すると、トランザクション内の各 SELECT に適用されます。1 つのクエリの中で HOLDLOCK と NOHOLDLOCK の両方のオプションは指定できません。

- NOHOLDLOCK キーワードは、SQL Anywhere によって認識されますが、効果はありません。

- Transact-SQL は SELECT 文を使用してローカル変数に値を割り当てます。

  ```
  SELECT @localvar = 42;
  ```

  SQL Anywhere でこれに対応する文は、SET 文です。

  ```
  SET @localvar = 42;
  ```

- Adaptive Server Enterprise では、次のものをサポートしません。

  - SELECT...INTO *host-variable-list*
  - SELECT...INTO *variable-list*
  - EXCEPT [ALL] または INTERSECT [ALL]
  - START AT 句
  - SQL Anywhere 定義のテーブルヒント
  - テーブル関数
  - FULL OUTER JOIN
  - FOR UPDATE BY { LOCK | TIMESTAMP }
  - Window 集合関数および線形回帰関数

- SQL Anywhere は、次に示す Adaptive Server Enterprise Transact-SQL SELECT 構文のキーワードと句をサポートしません。
  - SHARED キーワード
  - PARTITION キーワード
  - COMPUTE 句
  - FOR BROWSE 句
  - GROUP BY ALL 句
  - PLAN 句
  - ISOLATION 句
- SQL Anywhere は、識別子またはエイリアスで次の文字をサポートしていません。
  - " (二重引用符)
  - 制御文字 (0X20 未満の文字)
  - 円記号
  - 角カッコ
  - 逆引用符

参照

- 「SQL Flagger を使用した SQL 準拠のテスト」633 ページ
- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「OLAP のサポート」547 ページ
- 「GROUP BY と SQL/2008 標準」475 ページ
- 「FROM 句」『SQL Anywhere サーバ SQL リファレンス』
- 「Transact-SQL の外部ジョイン (*= or =*)」505 ページ

ジョインの互換性

Transact-SQL の SQL Anywhere の実装では、キーワード JOIN、LEFT OUTER JOIN、RIGHT OUTER JOIN、FULL OUTER JOIN を使用して SQL/2008 標準からジョイン構文を指定できるのに加えて、文の WHERE 句で特別な比較演算子 *= と =* を使用する既存の Transact-SQL 外部ジョイン構文を指定できます。

> **注意**
> Transact-SQL 外部ジョイン演算子 *= と =* は旧式であるため、将来のリリースではサポートから除外されます。

参照

- 「ジョイン：複数テーブルからのデータ検索」488 ページ
- 「FROM 句」『SQL Anywhere サーバ SQL リファレンス』
- 「Transact-SQL の外部ジョイン (*= or =*)」505 ページ
- 「SQL Flagger を使用した SQL 準拠のテスト」633 ページ

Transact-SQL のプロシージャ言語

「ストアドプロシージャ言語」は、SQL のうちの、ストアドプロシージャ、トリガ、バッチで使用されている部分です。

SQL Anywhere は、SQL/2008 に基づく Watcom SQL ダイアレクトに加えて、Transact-SQL ストアドプロシージャ言語の大部分をサポートします。

Transact-SQL のストアドプロシージャ

SQL Anywhere のネイティブダイアレクト Watcom-SQL は、ISO/ANSI SQL/2008 標準に基づいています。このため、Watcom-SQL のストアドプロシージャダイアレクトは、Transact-SQL ダイアレクトとは多くの点で異なります。概念と機能は似ていますが、構文が異なります。概念が似ているため、Transact-SQL の SQL Anywhere でのサポートは Watcom-SQL と Transact-SQL 間の自動変換を行えます。ただし、プロシージャはどちらかの言語だけで記述する必要があり、混在させることはできません。

SQL Anywhere での Transact-SQL のストアドプロシージャのサポート

ここでは、次に示す Transact-SQL ストアドプロシージャに対する SQL Anywhere のサポートについて説明します。

- パラメータを引き渡す
- 結果セットを返す
- ステータス情報を返す
- パラメータにデフォルト値を提供する
- 制御文
- エラー処理
- ユーザ定義関数

Transact-SQL のトリガ

トリガの互換性を保つには、トリガ機能とトリガ構文の互換性が必要です。この項では、Transact-SQL と SQL Anywhere トリガの機能の互換性の概要について説明します。

Adaptive Server Enterprise は、文レベルの AFTER トリガをサポートしています。つまり、このトリガは、トリガを起動する文が完了してから実行されます。SQL Anywhere でサポートされる Watcom-SQL ダイアレクトは、ローレベルの BEFORE、AFTER、INSTEAD OF トリガと、文レベルの AFTER および INSTEAD OF トリガをサポートしています。

ローレベルのトリガは、Transact-SQL 互換性機能の一部ではありません。

サポートされない Transact-SQL トリガまたは異なる Transact-SQL トリガの説明

Transact-SQL トリガの機能の中で、SQL Anywhere ではサポートされない機能、または、SQL Anywhere では異なる機能を次に示します。

- **他のトリガを起動するトリガ**　トリガが別のトリガを起動することがあります。この状況での SQL Anywhere と Adaptive Server Enterprise の対応は、やや異なります。Adaptive Server

Enterprise のデフォルトでは、トリガは設定可能なネストレベル (デフォルトは 16) まで、他のトリガを起動します。ネストレベルの設定は、Adaptive Server Enterprise のネストされたトリガオプションを使用します。SQL Anywhere では、メモリが不足していないかぎり、トリガは無制限に他のトリガを起動できます。

- **自分自身を起動するトリガ**　トリガが自分自身を起動する動作を行うことがあります。この状況での SQL Anywhere と Adaptive Server Enterprise の対応は、やや異なります。SQL Anywhere では、デフォルトで、Transact-SQL でないトリガは自分自身を再帰的に起動できます。一方、Transact-SQL ダイアレクトのトリガは自分自身を再帰的に起動することはできません。ただし、Transact-SQL ダイアレクトのトリガについては、SET 文 [T-SQL] の self_recursion オプションを使用して、トリガが自分自身を再帰的に呼び出すことを許可できます。

  Adaptive Server Enterprise のデフォルトでは、トリガは自分自身を再帰的に呼び出すことはできません。再帰を許可するには、self_recursion オプションを使用します。

- **トリガでの ROLLBACK 文はサポートされない**　Adaptive Server Enterprise は、トリガ中で、そのトリガを含むトランザクション全体をロールバックする ROLLBACK TRANSACTION 文を許可します。SQL Anywhere は、トリガで ROLLBACK (または ROLLBACK TRANSACTION) 文を許可しません。トリガとなる動作とそのトリガがともにアトミックステートメントを構成するためです。

  SQL Anywhere は、Adaptive Server Enterprise と互換性のある ROLLBACK TRIGGER 文を提供します。この文は、トリガ内で動作を取り消すために使用します。

- **ORDER 句はサポートされない**　Transact-SQL トリガでは ORDER *nn* 句を使用できません。trigger_order の値は自動的に 1 に設定されます。このため、文レベルのトリガがすでにある場合は、エラーが返され T-SQL トリガが作成されることがあります。これは、SYSTRIGGER システムテーブルには table_id、event、trigger_time、trigger_order にユニークインデックスがあるためです。特定のイベント (挿入、更新、削除) では、文レベルのトリガは常に AFTER であり、trigger_order は設定できません。このため、その他のトリガで 1 以外の順序に設定されないことを前提として、テーブルごとに 1 つのみを設定できます。

参照

- 「トリガ」94 ページ
- 「ストアドプロシージャ、トリガ、バッチ、ユーザ定義関数」77 ページ
- 「SET 文 [T-SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「ROLLBACK TRIGGER 文」『SQL Anywhere サーバ SQL リファレンス』

Transact-SQL のバッチ

Transact-SQL では、バッチは一緒に送信されてグループとして次々に実行される一連の SQL 文です。バッチは SQL スクリプトファイルとして保存されます。Interactive SQL を使用してバッチ対話形式で実行できます。

プロシージャで使われる制御文はバッチでも使えます。SQL Anywhere はバッチ中の制御文の使用をサポートします。また Transact-SQL のように、デリミタのない、バッチの終わりを示す GO 文で終わる文のグループをサポートします。

SQL スクリプトファイルに保存されているバッチでは、Interactive SQL はこれらのファイルにあるパラメータの使用をサポートします。

参照

- 「PARAMETERS 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』

ストアドプロシージャの自動変換

Transact-SQL 代替構文のサポートに加えて、SQL Anywhere は Watcom SQL と Transact-SQL の間で文を変換する手助けをします。次に示す SQL 言語組み込み関数は、SQL 文に関する情報を返して自動変換できるようにします。

- **SQLDIALECT(statement)** Watcom-SQL または Transact-SQL を返します。
- **WATCOM SQL(statement)** Watcom-SQL 構文を返します。
- **TRANSACTSQL(statement)** Transact-SQL 構文を返します。

これらは関数です。Interactive SQL の SELECT 文を使用してアクセスできます。たとえば、次の文は値 Watcom-SQL を返します。

```
SELECT SQLDIALECT( 'SELECT * FROM Employees' );
```

ストアドプロシージャの変換

たとえば、Watcom-SQL と Transact-SQL のような SQL ダイアレクト間でストアドプロシージャを変換します。

前提条件

そのプロシージャの所有者であるか、または次のいずれかの権限を持っていることが必要です。

- ALTER ANY PROCEDURE システム権限
- ALTER ANY OBJECT システム権限

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[プロシージャとファンクション]** フォルダをクリックし、リストからストアドプロシージャを 1 つ選択します。
3. 右ウィンドウ枠で、**[SQL]** タブをクリックして、テキストウィンドウをクリックします。
4. **[ファイル]** をクリックし、**[変換]** オプションの 1 つをクリックします。

   右ウィンドウ枠に、選択したダイアレクトでプロシージャが表示されます。選択されたダイアレクトが保存されているプロシージャと異なるときは、データベースサーバが変換を行います。変換されなかった行はコメントとして表示されます。

5. 変換されなかった行を書き直します。
6. **[ファイル]** » **[保存]** をクリックします。

結果

ストアドプロシージャが変換され、データベースに保存されます。

参照

- 「ストアドプロシージャの自動変換」662 ページ
- 「SQLDIALECT 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「TRANSACTSQL 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「WATCOMSQL 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』

Transact-SQL プロシージャから返される結果セット

SQL Anywhere は RESULT 句を使用して、返される結果セットを指定します。Transact-SQL プロシージャでは、最初のクエリのカラム名またはエイリアス名が呼び出し環境に返されます。

Transact-SQL プロシージャの例

次の Transact-SQL プロシージャは、Transact-SQL ストアドプロシージャが結果セットを返す方法を示します。

```
CREATE PROCEDURE ShowDepartment (@deptname VARCHAR(30))
AS
  SELECT Employees.Surname, Employees.GivenName
  FROM Departments, Employees
  WHERE Departments.DepartmentName = @deptname
  AND Departments.DepartmentID = Employees.DepartmentID;
```

Watcom SQL プロシージャの例

次に、これに対応する SQL Anywhere のプロシージャを示します。

```
CREATE PROCEDURE ShowDepartment(in deptname VARCHAR(30))
RESULT ( LastName CHAR(20), FirstName CHAR(20))
BEGIN
  SELECT Employees.Surname, Employees.GivenName
  FROM Departments, Employees
  WHERE Departments.DepartmentName = deptname
  AND Departments.DepartmentID = Employees.DepartmentID
END;
```

参照

- 「結果セット」115 ページ

Transact-SQL プロシージャの中の変数

SQL Anywhere は SET 文を使用して、プロシージャ内の変数に値を割り当てます。Transact-SQL では、空のテーブルリストの SELECT 文、または SET 文を使用して値を割り当てます。次のプロシージャは、Transact-SQL 構文の働きを示します。

```
CREATE PROCEDURE multiply
        @mult1 int,
        @mult2 int,
        @result int output
AS
SELECT @result = @mult1 * @mult2;
```

このプロシージャを呼び出すには、次のようにします。

```
CREATE VARIABLE @product int
go
EXECUTE multiply 5, 6, @product OUTPUT
go
```

変数 @product の値は、プロシージャの実行後 30 になります。

参照

- 「Transact-SQL と互換性のあるクエリ」656 ページ
- 「SET 文」『SQL Anywhere サーバ SQL リファレンス』

Transact-SQL プロシージャでのエラー処理

デフォルトでのプロシージャのエラー処理は、Watcom SQL と Transact-SQL とでは違います。Watcom SQL のデフォルトでは、エラーが起こった場合にプロシージャは終了し、呼び出した環境に SQLSTATE 値と SQLCODE 値を返します。

EXCEPTION 文を使って、Watcom SQL ストアドプロシージャに明示的なエラー処理を組み込むことができます。または、ON EXCEPTION RESUME 文を使って、エラーが起こった次の文から実行を再開するよう、プロシージャに指示することもできます。

Transact-SQL のプロシージャでエラーが起こった場合は、次の文から実行が継続されます。最後に実行された文のエラーステータスは、グローバル変数 @@error に保存されます。文の後ろにあるこの変数をチェックして、プロシージャから強制的に返すことができます。たとえば、次の文は、エラーが起こると終了させます。

```
IF @@error != 0 RETURN
```

プロシージャが実行を終了したときの戻り値で、プロシージャが成功したかどうかがわかります。この戻り値は整数で、次のように指定してアクセスできます。

```
DECLARE @Status INT
EXECUTE @Status = proc_sample
IF @Status = 0
  PRINT 'procedure succeeded'
ELSE
  PRINT 'procedure failed'
```

次のテーブルは、組み込みプロシージャの戻り値とその意味を示します。

値	定義	SQL Anywhere SQLSTATE
0	プロシージャはエラーを起こすことなく実行された	
-1	オブジェクトが見つからない	42W33、52W02、52003、52W07、42W05
-2	データ型エラー	53018
-3	プロセスはデッドロックの対象となった	40001、40W06
-4	パーミッションエラー	42501
-5	構文エラー	42W04
-6	その他のユーザエラー	
-7	リソースエラー、たとえば領域が足りない	08W26
-10	致命的な内部の矛盾	40W01
-11	致命的な内部の矛盾	40000
-13	データベースが壊れている	WI004
-14	ハードウェアエラー	08W17、40W03、40W04

SQL Anywhere SQLSTATE が該当しない場合、デフォルト値の -6 が返されます。

RETURN 文は、この表の値以外の、ユーザが意味を定義した整数値を返すこともできます。

RAISERROR 文を使用するプロシージャ

RAISERROR 文を使用してユーザ定義エラーを作成します。RAISERROR 文は、SIGNAL 文と同じように機能します。

RAISERROR 文そのものは、プロシージャを終了しません。ただし、ユーザ定義エラー後の実行を制御するために、RETURN 文やグローバル変数 @@error のテストと組み合わせることができます。

on_tsql_error データベースオプションを Continue に設定すると、RAISERROR 文は実行終了時にエラーを通知しません。代わりに、プロシージャが完了し、RAISERROR のステータスコードとメッセージを保存してから、最新の RAISERROR を返します。RAISERROR を返したプロシージャが他のプロシージャから呼び出された場合、最も外側のプロシージャが終了してから

RAISERROR が返されます。on_tsql_error オプションをデフォルト値 (Conditional) に設定した場合、continue_after_raiserror オプションは、RAISERROR 文の実行後の動作を制御します。on_tsql_error オプションを Stop または Continue に設定した場合、その設定は continue_after_raiserror の設定より優先されます。

中間レベルの RAISERROR ステータスとコードは、プロシージャが終了すると失われます。結果が返されるときに RAISERROR とともにエラーが発生した場合は、エラー情報が返され、RAISERROR 情報は失われます。アプリケーションでは、別の実行ポイントでグローバル変数 @@error を検査して、中間の RAISERROR ステータスを問い合わせることができます。

参照

- 「RAISERROR 文」『SQL Anywhere サーバ SQL リファレンス』

Watcom SQL での Transact-SQL のようなエラー処理

CREATE PROCEDURE 文に ON EXCEPTION RESUME 句を追加して、Watcom SQL のプロシージャが Transact-SQL に似た方法でエラー処理を行うようにすることができます。

```
CREATE PROCEDURE sample_proc()
ON EXCEPTION RESUME
BEGIN
  ...
END
```

ON EXCEPTION RESUME 句があると、明示的な例外処理コードは実行されません。このため、明示的なエラー処理ではこの句を避けます。

参照

- 「CREATE PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』

データベースにおける XML

Extensible Markup Language (XML) は、構造化データをテキスト形式で表します。XML は、大規模な電子出版の課題を満たすために設計されました。

XML は、HTML のように単純なマークアップ言語ですが、SGML のように柔軟性があります。XML は、階層型で、その主な目的は、人間とコンピュータの両方が作成し、読むことのできるデータの構造を記述することです。

XML では、さまざまな形式のデータを記述する、一連の静的な要素は提供されておらず、ユーザが要素を定義できます。そのため、XML を使用して多くの種類の構造化データを記述できます。XML 文書では、オプションとして文書型定義 (DTD) または XML スキーマを使用し、XML ファイルで使用される構造、要素、属性を定義できます。

SQL Anywhere で XML を使用する方法はいくつかあります。

- データベースに XML 文書を格納する
- リレーショナルデータを XML としてエクスポートする
- データベースに XML をインポートする
- リレーショナルデータを XML として問い合わせる

XML の詳細については、http://www.w3.org/XML/を参照してください。

リレーショナルデータベースにおける XML 文書の格納

SQL Anywhere は、データベースで XML 文書を格納するために使用できる 2 つのデータ型、XML データ型と LONG VARCHAR データ型をサポートしています。これらのデータ型は、いずれも XML 文を文字列としてデータベースに格納します。

XML データ型は、データベースサーバの文字セットエンコードを使用します。XML エンコード属性は、データベースサーバが使用するエンコードと一致する必要があります。XML エンコード属性は、自動文字セット変換の実行方法を指定しません。

XML データ型は、文字列と相互にキャスト可能なすべてのデータ型と、相互にキャストできます。文字列が XML にキャストされるときに、整形式かどうかはチェックされません。

リレーショナルデータから要素を生成するとき、XML で無効な文字は、データが XML 型でないかぎりエスケープされます。たとえば、次の内容の <product> という要素を生成するとします。この要素の内容には、不等号 (より小、より大) が含まれています。

```
<hat>bowler</hat>
```

要素の内容を XML 型として指定するクエリを記述した場合、次のように不等号はマークアップされません。

```
SELECT XMLFOREST( CAST( '<hat>bowler</hat>' AS XML ) AS product );
```

結果は次のようになります。

```
<product><hat>bowler</hat></product>
```

しかし、たとえば次のように、クエリでこの要素の内容を XML 型として指定しない場合があります。

```
SELECT XMLFOREST( '<hat>bowler</hat>' AS product );
```

この場合、不等号がエンティティの参照で置換されます。

```
<product>&lt;hat&gt;bowler&lt;/hat&gt;</product>
```

属性は、データ型に関係なく常にマークアップされます。

参照

- 「不正な XML 名のエンコーディングのルール」679 ページ
- 「XML データ型」『SQL Anywhere サーバ SQL リファレンス』

XML としてエクスポートされるリレーショナルデータ

SQL Anywhere は、リレーショナルデータを XML としてエクスポートするために、Interactive SQL OUTPUT 文と ADO.NET DataSet オブジェクトの 2 つの方法を提供しています。

FOR XML 句と SQL/XML 関数を使用して、データベースのリレーショナルデータから結果セットを XML として生成できます。次に、UNLOAD 文または xp_write_file システムプロシージャを使用して、生成された XML をファイルにエクスポートできます。

Interactive SQL から XML としてエクスポートされるリレーショナルデータ

Interactive SQL OUTPUT 文は、XML フォーマットをサポートしており、クエリ結果を生成された XML ファイルに出力します。

この生成された XML ファイルは、UTF-8 でエンコードされており、埋め込み DTD が含まれます。XML ファイルでは、バイナリ値は、2 桁の 16 進数文字列として表されるバイナリデータとして文字データ (CDATA) ブロック内にエンコードされます。

INPUT 文は、XML をファイルフォーマットとして受け入れません。ただし、OPENXML 演算子または ADO.NET DataSet オブジェクトを使用して XML をインポートできます。

参照

- 「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「XML 文書をリレーショナルデータとしてインポートする方法」669 ページ

DataSet オブジェクトを使用して XML としてエクスポートされるリレーショナルデータ

ADO.NET DataSet オブジェクトを使用して、DataSet の内容を XML 文書に保存できます。一度、データベースのクエリ結果などを DataSet に入力すると、DataSet からスキーマのみか、スキーマとデータの両方を XML ファイルに保存できます。WriteXml メソッドは、スキーマとデータの両方を XML ファイルに保存します。WriteXmlSchema メソッドは、スキーマのみを XML ファイルに保存します。SQL Anywhere .NET データプロバイダを使用して DataSet オブジェクトに入力することが可能です。

DataSet を使用してリレーショナルデータを XML としてエクスポートする方法については、「SACommand : ExecuteNonQuery を使用したローの挿入、削除、更新」『SQL Anywhere サーバ プログラミング』を参照してください。

XML 文書をリレーショナルデータとしてインポートする方法

SQL Anywhere は、XML をデータベースにインポートする 2 種類の方法をサポートしています。

- OPENXML 演算子を使用して、XML 文書から結果セットを生成する。
- ADO.NET DataSet オブジェクトを使用して、XML 文書から DataSet にデータかスキーマまたはその両方を読み込む

OPENXML 演算子を使用した XML のインポート

クエリの FROM 句で OPENXML 演算子を使用すると、XML 文書から結果セットを生成できます。OPENXML は、XPath クエリ言語のサブセットを使用して、XML 文書からノードを選択します。

XPath 式の使用

OPENXML を使用すると、XML 文書が解析され、結果はツリーとしてモデル化されます。このツリーはノードで構成されています。XPath 式は、ツリー内のノードを選択するために使用されます。次のリストは、一般的に使用される XPath 式の一部を示しています。

- **/** XML 文書のルートノードを示します。
- **//** ルートのすべての子孫を示します。ルートノードも含まれます。
- **. (単一のピリオド)** XML 文書のカレントノードを示します。
- **.//** カレントノードのすべての子孫を示します。カレントノードも含まれます。
- **..** カレントノードの親ノードを示します。

- **./@attributename** *attributename* という名前を持つ、カレントノードの属性を示します。
- **./childname** カレントノードの子で、*childname* という名前を持つ要素を示します。

次の XML 文書を考えてみます。

```
<inventory>
  <product ID="301" size="Medium">Tee Shirt
    <quantity>54</quantity>
  </product>
  <product ID="302" size="One Size fits all">Tee Shirt
    <quantity>75</quantity>
  </product>
  <product ID="400" size="One Size fits all">Baseball Cap
    <quantity>112</quantity>
  </product>
</inventory>
```

<inventory> 要素は、ルートノードです。この要素は、次の XPath 式を使用して参照できます。

```
/inventory
```

カレントノードが <quantity> 要素であると仮定します。このノードは、次の XPath 式を使用して参照できます。

```
.
```

<inventory> 要素の子である <product> 要素をすべて検出するには、次の XPath 式を使用します。

```
/inventory/product
```

カレントノードが <product> 要素のときに、size 属性を参照したい場合は、次の XPath 式を使用します。

```
./@size
```

XPath クエリ言語の詳細については、http://www.w3.org/TR/xpath を参照してください。

OPENXML を使用した結果セットの生成

OPENXML の最初の *xpath-query* 引数に一致するごとに、結果セットにローが 1 つ生成されます。WITH 句は、結果セットのスキーマと、結果セット内で各カラムに値がどのように格納されるかを指定します。次のクエリを例にとります。

```
SELECT * FROM OPENXML(
'<inventory>
  <product>Tee Shirt
    <quantity>54</quantity>
    <color>Orange</color>
  </product>
  <product>Baseball Cap
    <quantity>112</quantity>
    <color>Black</color>
  </product>
</inventory>',
'/inventory/product' )
WITH ( Name CHAR (25) './text()',
       Quantity CHAR(3) 'quantity',
       Color CHAR(20) 'color');
```

最初の *xpath-query* 引数は、/inventory/product です。そして XML には <product> 要素が 2 つあるため、このクエリによってローが 2 つ生成されます。

WITH 句は、カラムが Name、Quantity、Color の 3 つであることを指定します。これらのカラムの値は、<product>、<quantity>、<color> の各要素から取得されます。前述のクエリは、次の結果を生成します。

Name	Quantity	Color
Tee Shirt	54	Orange
Baseball Cap	112	Black

OPENXML を使用したエッジテーブルの生成

OPENXML 演算子を使用すると、XML 文書内の各要素に対応する各行から成るエッジテーブルを生成できます。エッジテーブルを生成すると、SQL を使用して結果セット内のデータを問い合わせできます。

次の SQL 文は、XML 文書を 1 つ含むテーブルを作成します。このクエリが生成する XML には、<root> と呼ばれるルート要素があります。このルート要素は、XMLELEMENT 関数を使用して生成されています。また、ELEMENTS 修飾子を持つ FOR XML AUTO を使用して、Employees テーブル、SalesOrders テーブル、Customers テーブルの指定された各カラムに対応する要素が生成されています。

```
CREATE TABLE IF NOT EXISTS xmldata (xmldoc XML);
INSERT INTO xmldata WITH AUTO NAME
   SELECT XMLELEMENT( NAME root,
       (SELECT EmployeeID, Employees.GivenName, Employees.Surname,
         Customers.ID, Customers.GivenName, Customers.Surname, Customers.Phone, CompanyName,
         SalesOrders.ID, OrderDate, Region
       FROM Employees
       KEY JOIN SalesOrders
       KEY JOIN Customers
       ORDER BY EmployeeID, Customers.ID, SalesOrders.ID
       FOR XML AUTO, ELEMENTS)) AS xmldoc;
SELECT xmldoc FROM xmldata;
```

生成される XML ロックは、次のようになります (結果は読みやすいようにフォーマットされています。クエリから返される結果は 1 つの連続した文字列です)。

```
<root>
 <Employees>
   <EmployeeID>129</EmployeeID>
   <GivenName>Philip</GivenName>
   <Surname>Chin</Surname>

   <Customers>
     <ID>101</ID>
     <GivenName>Michaels</GivenName>
     <Surname>Devlin</Surname>
     <Phone>2015558966</Phone>
     <CompanyName>The Power Group</CompanyName>
      <SalesOrders>
        <ID>2560</ID>
        <OrderDate>2001-03-16</OrderDate>
```

```
        <Region>Eastern</Region>
      </SalesOrders>
    </Customers>

    <Customers>
      <ID>103</ID>
      <GivenName>Erin</GivenName>
      <Surname>Niedringhaus</Surname>
      <Phone>2155556513</Phone>
      <CompanyName>Darling Associates</CompanyName>
      <SalesOrders>
        <ID>2451</ID>
        <OrderDate>2000-12-15</OrderDate>
        <Region>Eastern</Region>
      </SalesOrders>
    </Customers>

    <Customers>
      <ID>104</ID>
      <GivenName>Meghan</GivenName>
      <Surname>Mason</Surname>
      <Phone>6155555463</Phone>
      <CompanyName>P.S.C.</CompanyName>
      <SalesOrders>
        <ID>2331</ID>
        <OrderDate>2000-09-17</OrderDate>
        <Region>South</Region>
      </SalesOrders>

      <SalesOrders>
        <ID>2342</ID>
        <OrderDate>2000-09-28</OrderDate>
        <Region>South</Region>
      </SalesOrders>
    </Customers>
    ...
  </Employees>
  ...
  <Employees>
  ...
  </Employees>
</root>
```

次のクエリは、descendant-or-self (//*) XPath 式を使用して、前述の XML 文書内の各要素とのマッチングを行っています。次に、各要素に対し id メタプロパティを使用して、ノードの ID を取得しています。また、parent (../) XPath 式を id メタプロパティとともに使用して、親ノードを取得しています。localname メタプロパティは、各要素の名前を取得するために使用されています。メタプロパティ名では大文字と小文字が区別されます。そのため、ID や LOCALNAME はメタプロパティ名として使用できません。

```
CREATE OR REPLACE VARIABLE x XML;
SELECT xmldoc INTO x FROM xmldata;

SELECT *
FROM OPENXML( x, '//*' )
WITH (ID INT '@mp:id',
      parent INT '../@mp:id',
      name CHAR(25) '@mp:localname',
      text LONG VARCHAR 'text()' )
ORDER BY ID;
```

このクエリによって生成される結果セットには、XML 文書内の各ノードの ID、親ノードの ID、各要素の名前と内容が表示されています。

ID	parent	name	text
5	(NULL)	root	(NULL)
16	5	Employees	(NULL)
28	16	EmployeeID	129
55	16	GivenName	Phillip
82	16	Surname	Chin
...	...	...	...

xp_read_file での OPENXML の使用

これまで XMLELEMENT のようなプロシージャで生成された XML を使用してきましたが、xp_read_file プロシージャを使用して、ファイルからの XML を読み込んで解析することもできます。次のクエリを使用してファイル *c:¥temp¥inventory.xml* が書き込まれたとします。

```
SELECT xp_write_file( 'c:¥¥temp¥¥inventory.xml',
'<inventory>
  <product>Tee Shirt
    <quantity>54</quantity>
    <color>Orange</color>
  </product>
  <product>Baseball Cap
    <quantity>112</quantity>
    <color>Black</color>
  </product>
</inventory>'
);
```

この場合、次の文を使用してファイル内の XML を読み込み、解析できます。

```
SELECT *
FROM OPENXML( xp_read_file( 'c:¥¥temp¥¥inventory.xml' ),
  '//*' )
WITH (ID INT '@mp:id',
      parent INT '../@mp:id',
      name CHAR(128) '@mp:localname',
      text LONG VARCHAR 'text()' )
ORDER BY ID;
```

カラム内の XML の問い合わせ

XML を含むカラムを持つテーブルがある場合、OPENXML を使用して、カラム内のすべての XML 値を一度に問い合わせできます。これには、ラテラル派生テーブルを使用します。

次の文は、ManagerID と Reports という 2 つのカラムを持つテーブルを作成します。Reports カラムには、Employees テーブルから生成された XML データが含まれます。

```
CREATE TABLE IF NOT EXISTS xmltest (ManagerID INT, Reports XML);
INSERT INTO xmltest
  SELECT ManagerID, XMLELEMENT( NAME reports,
    XMLAGG( XMLELEMENT( NAME e, EmployeeID)))
  FROM Employees
  GROUP BY ManagerID;
```

次のクエリを実行して、テストテーブル内のデータを表示してください。

```
SELECT *
FROM xmltest
ORDER BY ManagerID;
```

このクエリは、次の結果を生成します。

ManagerID	Reports
501	<reports> <e>102</e> <e>105</e> <e>160</e> <e>243</e> ... </reports>
703	<reports> <e>191</e> <e>750</e> <e>868</e> <e>921</e> ... </reports>
902	<reports> <e>129</e> <e>195</e> <e>299</e> <e>467</e> ... </reports>
1293	<reports> <e>148</e> <e>390</e> <e>586</e> <e>757</e> ... </reports>
...	...

次のクエリは、ラテラル派生テーブルを使用して、2 つのカラムを持つ結果セットを生成しています。1 つのカラムは、各マネージャの ID をリストします。もう 1 つのカラムは、そのマネージャに報告を行う各従業員の ID をリストします。

```
SELECT ManagerID, EmployeeID
FROM xmltest, LATERAL( OPENXML( xmltest.Reports, '//e' )
WITH (EmployeeID INT '.') ) DerivedTable
ORDER BY ManagerID, EmployeeID;
```

このクエリは、次の結果を生成します。

ManagerID	EmployeeID
501	102
501	105
501	160
501	243
...	...

参照

- 「XMLELEMENT 関数 [文字列]」『SQL Anywhere サーバ SQL リファレンス』
- 「FOR XML AUTO」682 ページ
- 「OPENXML 演算子」『SQL Anywhere サーバ SQL リファレンス』
- 「FROM 句」『SQL Anywhere サーバ SQL リファレンス』

DataSet オブジェクトを使用した XML のインポート

ADO.NET DataSet オブジェクトを使用して、XML 文書から DataSet にデータかスキーマまたはその両方を読み込めます。

- ReadXml メソッドは、スキーマとデータの両方を含む XML 文書から DataSet に投入を行います。
- ReadXmlSchema メソッドは、XML 文書からスキーマのみを読み込みます。一度 DataSet に XML 文書のデータが入力されると、DataSet からの変更に基づいてデータベース内のテーブルを更新できます。

また、SQL Anywhere .NET データプロバイダを使用して、DataSet オブジェクトを操作できます。

参照

- 「SADataAdapter：概要」『SQL Anywhere サーバ プログラミング』

デフォルトの XML ネームスペースの定義

デフォルトのネームスペースは、xmlns="URI" の形式の属性で、XML 文書の要素に定義します。次の例では、文書には http://www.iAnywhere.com/EmployeeDemo という URI にバインドされるデフォルトのネームスペースがあります。

```
<x xmlns="http://www.iAnywhere.com/EmployeeDemo"/>
```

要素の名前にプレフィクスがない場合は、その要素およびその要素のすべての子孫要素にデフォルトのネームスペースが適用されます。コロンによって、プレフィクスと残りの要素名は区切ら

れます。たとえば、<x/> にはプレフィクスがないが、<p:x/> にはプレフィクス p があります。xmlns:prefix="URI" の形式の属性によって、プレフィクスにバインドされるネームスペースを定義します。次の例では、文書がプレフィクス p を前述の例と同じ URI にバインドします。

```
<x xmlns:p="http://www.iAnywhere.com/EmployeeDemo"/>
```

デフォルトのネームスペースが属性に適用されることはありません。プレフィクスがある場合を除き、属性は常に NULL ネームスペース URI にバインドされます。次の例では、ルート要素と子要素には iAnywhere1 ネームスペースがあり、x 属性には NULL ネームスペース URI、y 属性には iAnywhere2 ネームスペースがあります。

```
<root xmlns="iAnywhere1" xmlns:p="iAnywhere2">
<child x='1' p:y='2' />
</root>
```

XML 文書を、OPENXML クエリの *namespace-declaration* 引数として渡すと、文書のルート要素に定義されたネームスペースはクエリに適用されます。ルート要素以降の残りの文書はすべて無視されます。次の例では、p1 は文書では iAnywhere1 にバインドされ、*namespace-declaration* 引数では p2 にバインドされます。クエリはプレフィクス p2 を使用できます。

```
SELECT *
FROM OPENXML('<p1:x xmlns:p1="iAnywhere1">123</p1:x>', '/p2:x', 1, '<root xmlns:p2="iAnywhere1"/
>')
WITH ( c1 int '.' );
```

要素を一致させる場合、プレフィクスがバインドされる URI を正確に指定する必要があります。上の例では、xpath クエリの x 名と文書の x 要素は、どちらも iAnywhere1 ネームスペースがあるため、一致となります。

要素を一致させる場合、プレフィクスがバインドされる URI を正確に指定する必要があります。上の例では、xpath クエリの x 名と文書の x 要素は、どちらも iAnywhere1 ネームスペースがあるため、一致となります。xpath 要素 x のプレフィクスは、*xml-data* 内で x 要素に対して定義されたネームスペースと一致する、*namespace-declaration* 内で定義されたネームスペース iAnywhere1 を参照します。

OPENXML 演算子の *namespace-declaration* では、デフォルトのネームスペースを使用しないでください。NULL ネームスペースを含む任意の URI にバインドされる x 要素と一致する、/*:x 形式のワイルドカードクエリを使用してください。または、特定のプレフィクスに必要な URI をバインドし、それをクエリで使用します。

参照

- 「OPENXML 演算子」『SQL Anywhere サーバ SQL リファレンス』

XML としてのクエリ結果

SQL Anywhere は、リレーショナルデータからクエリ結果を XML として取得する 2 種類の方法をサポートしています。

- **FOR XML 句**　FOR XML 句を SELECT 文で使用して、XML 文書を生成できます。

- **SQL/XML**　SQL Anywhere は、リレーショナルデータから XML 文書を生成する、ドラフト段階の SQL/XML 標準に基づく関数をサポートしています。

SQL Anywhere がサポートする FOR XML 句と SQL/XML 関数により、リレーショナルデータから XML を生成する選択肢が 2 通り提供されます。通常は、どちらかの方法を使用して同じ XML を生成できます。

たとえば、このクエリは、FOR XML AUTO を使用して XML を生成しています。

```
SELECT ID, Name
FROM Products
WHERE Color='black'
FOR XML AUTO;
```

次のクエリは、XMLELEMENT 関数を使用して XML を生成しています。

```
SELECT XMLELEMENT(NAME product,
        XMLATTRIBUTES(ID, Name))
FROM Products
WHERE Color='black';
```

どちらのクエリも次の XML を生成します (結果セットは読みやすいようにフォーマットされています)。

```
<product ID="302" Name="Tee Shirt"/>
<product ID="400" Name="Baseball Cap"/>
<product ID="501" Name="Visor"/>
<product ID="700" Name="Shorts"/>
```

ヒント

ネストの深い文書を生成する場合は、FOR XML EXPLICIT クエリの方が SQL/XML クエリよりも効率的である可能性が高くなります。これは、EXPLICIT モードクエリは、通常 UNION を使用してネストを生成するのに対し、SQL/XML はサブクエリを使用して必要なネストを生成するためです。

参照

- 「クエリ結果を XML として取り出すための FOR XML 句の使用」677 ページ
- 「クエリ結果を XML として取得するための SQL/XML の使用」695 ページ
- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』

クエリ結果を XML として取り出すための FOR XML 句の使用

SQL Anywhere では、SELECT 文内で FOR XML 句を使用して、データベースに対し SQL クエリを実行し、結果を XML 文書として返すことができます。XML 文書は、XML 型です。

FOR XML 句は、サブクエリ、GROUP BY 句または集合関数のあるクエリ、ビュー定義など、どのような SELECT 文内でも使用できます。

SQL Anywhere は、FOR XML 句を使用して生成された XML 文書に対しスキーマは生成しません。

FOR XML 句内で、生成される XML のフォーマットを制御する 3 つの XML モードのうちの 1 つを指定できます。

- **RAW**　クエリに一致する各ローを <row> XML 要素として、各カラムを属性として表します。
- **AUTO**　クエリ結果をネストされた XML 要素として返します。SELECT リスト内で参照される各テーブルは、XML 内で要素として表されます。要素のネスト順は、SELECT リスト内のカラムの順序に基づきます。
- **EXPLICIT**　希望するネストに関する情報を含むクエリを記述できます。そのため、生成される XML の形式を制御できます。

次の項では、FOR XML 句の 3 つのモードに共通する、バイナリデータ、NULL 値、無効な XML 名に関連する動作について説明します。また、FOR XML 句の使用例も示します。

参照

- 「XML データ型」『SQL Anywhere サーバ SQL リファレンス』
- 「FOR XML の例」680 ページ
- 「FOR XML RAW」680 ページ
- 「FOR XML AUTO」682 ページ
- 「FOR XML EXPLICIT」685 ページ

FOR XML とバイナリデータ

SELECT 文で FOR XML 句を使用すると、使用するモードに関わらず、すべての BINARY、LONG BINARY、IMAGE、または VARBINARY カラムは、自動的に Base64 エンコードフォーマットで表される属性または要素として出力されます。

OPENXML を使用して XML から結果セットを生成する場合、OPENXML は、BINARY、LONG BINARY、IMAGE、VARBINARY の各データ型を Base64 でエンコードされていると見なし、自動的にデコードします。

参照

- 「OPENXML 演算子」『SQL Anywhere サーバ SQL リファレンス』

FOR XML と NULL 値

デフォルトでは、NULL 値を含む要素と属性は、結果から省略されます。for_xml_null_treatment オプションを使用すると、この動作を制御できます。

NULL 会社名を含む Customers テーブル内のエントリを考えてみます。

```
INSERT INTO Customers( ID, Surname, GivenName, Street, City, Phone)
VALUES (100,'Robert','Michael', '100 Anywhere Lane','Smallville','519-555-3344');
```

for_xml_null_treatment オプションを Omit (デフォルト) に設定して次のクエリを実行すると、属性は NULL カラム値について生成されません。

```
SELECT ID, GivenName, Surname, CompanyName
FROM Customers
WHERE GivenName LIKE 'Michael%'
ORDER BY ID
FOR XML RAW;
```

この場合、CompanyName 属性は Michael Robert について生成されません。

```
<row ID="100" GivenName="Michael" Surname="Robert"/>
<row ID="101" GivenName="Michaels" Surname="Devlin" CompanyName="The Power Group"/>
<row ID="110" GivenName="Michael" Surname="Agliori" CompanyName="The Pep Squad"/>
```

for_xml_null_treatment オプションを Empty に設定すると、空の属性も結果に含まれます。

```
<row ID="100" GivenName="Michael" Surname="Robert" CompanyName=""/>
<row ID="101" GivenName="Michaels" Surname="Devlin" CompanyName="The Power Group"/>
<row ID="110" GivenName="Michael" Surname="Agliori" CompanyName="The Pep Squad"/>
```

この場合、空の CompanyName 属性が Michael Robert について生成されています。

参照

- 「for_xml_null_treatment オプション」『SQL Anywhere サーバ データベース管理』

不正な XML 名のエンコーディングのルール

SQL Anywhere は、次のルールを使用して、XML 名として有効ではない名前 (たとえば、スペースを含むカラム名) をエンコードします。

XML には、SQL 名のルールとは異なる名前のルールがあります。たとえば XML 名にはスペースを使用できません。カラム名などの SQL 名が XML 名に変換されると、XML 名で有効でない文字はエンコードされるかエスケープされます。

エンコードされた各文字について、エンコーディングは文字の Unicode のコードポイント値が基になり、16 進数で表されます。

- ほとんどの文字のコードポイント値は、16 ビット、または 4 桁の 16 進数で表すことができ、_xHHHH_ というエンコーディングが使用されます。このような文字に対応する Unicode 文字の UTF-16 値は、16 ビットワード 1 つです。

- コードポイント値が 16 ビットよりも多く必要な文字では、8 桁の 16 進数が使用され、エンコーディングは _xHHHHHHHH_ になります。このような文字に対応する Unicode 文字の UTF-16 値は、16 ビットワード 2 つです。ただし、エンコーディングには UTF-16 値ではなく、Unicode のコードポイント値 (通常は 16 進数で 5 または 6 桁) が使用されます。

  たとえば、次のクエリには、スペースを持つカラム名が含まれています。

  ```
  SELECT EmployeeID AS "Employee ID"
  FROM Employees
  FOR XML RAW;
  ```

  そのため、次の結果が返されます。

  ```
  <row Employee_x0020_ID="102"/>
  <row Employee_x0020_ID="105"/>
  ```

```
<row Employee_x0020_ID="129"/>
<row Employee_x0020_ID="148"/>
...
```

●アンダースコア (_) は、次に文字 x が続く場合、エスケープされます。たとえば、名前 Linu_x は Linu_x005F_x のようにエンコーディングされます。

●コロン (:) は、エスケープされません。そのため、FOR XML クエリを使用して、名前空間宣言と修飾された要素名、属性名を生成できます。

ヒント
Interactive SQL で FOR XML 句を含むクエリを実行する場合は、truncation_length オプションを設定するとカラム長を増やせます。

参照

- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「truncation_length オプション [Interactive SQL]」『SQL Anywhere サーバ データベース管理』

FOR XML の例

以降の例は、SELECT 文内での FOR XML 句の使用方法を示します。

●次の例は、サブクエリ内での FOR XML 句の使用方法を示します。

```
SELECT XMLELEMENT( NAME root,
   (SELECT * FROM Employees FOR XML RAW) );
```

●次の例は、GROUP BY 句と集合関数のあるクエリ内での FOR XML 句の使用方法を示します。

```
SELECT Name, AVG(UnitPrice) AS Price
FROM Products
GROUP BY Name
FOR XML RAW;
```

●次の例は、ビュー定義内での FOR XML 句の使用方法を示します。

```
CREATE VIEW EmployeesDepartments
AS SELECT Surname, GivenName, DepartmentName
FROM Employees JOIN Departments
ON Employees.DepartmentID = Departments.DepartmentID
FOR XML AUTO;
```

FOR XML RAW

クエリ内で FOR XML RAW を指定すると、各ローは、<row> 要素として表され、各カラムは、<row> 要素の属性となります。

構文

FOR XML RAW[, **ELEMENTS**]

パラメータ

ELEMENTS　このパラメータを指定すると、FOR XML RAW は、結果における各カラムに対し属性の代わりに XML 要素を生成します。NULL 値がある場合は、その要素は、生成される XML 文書から省略されます。次のクエリは、<EmployeeID> 要素と <DepartmentName> 要素を生成します。

```
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
  ON Employees.DepartmentID=Departments.DepartmentID
FOR XML RAW, ELEMENTS;
```

このクエリは、次の結果を返します。

```
<row>
  <EmployeeID>102</EmployeeID>
  <DepartmentName>R &amp; D</DepartmentName>
</row>
<row>
  <EmployeeID>105</EmployeeID>
  <DepartmentName>R &amp; D</DepartmentName>
</row>
<row>
  <EmployeeID>160</EmployeeID>
  <DepartmentName>R &amp; D</DepartmentName>
</row>
<row>
  <EmployeeID>243</EmployeeID>
  <DepartmentName>R &amp; D</DepartmentName>
</row>
...
```

使用法

BINARY、LONG BINARY、IMAGE、VARBINARY カラムのデータは、FOR XML RAW を含むクエリを実行すると、自動的に Base64 エンコードフォーマットで返されます。

デフォルトでは、NULL 値は、結果から省略されます。for_xml_null_treatment オプションを使用すると、この動作を制御できます。

FOR XML RAW は、整形式 XML 文書を返しません。これは、文書に単一のルートノードが含まれないためです。<root> 要素が必要な場合は、1 つの方法として、XMLELEMENT 関数を使用して挿入できます。次に例を示します。

```
SELECT XMLELEMENT( NAME root,
            (SELECT EmployeeID AS id, GivenName AS name
            FROM Employees FOR XML RAW) );
```

XML 文書内で使用される属性名や要素名は、エイリアスを指定して変更できます。次のクエリは、ID 属性の名前を product_ID に変更します。

```
SELECT ID AS product_ID
FROM Products
WHERE Color='black'
FOR XML RAW;
```

このクエリは、次の結果を返します。

```
<row product_ID="302"/>
<row product_ID="400"/>
```

```
<row product_ID="501"/>
<row product_ID="700"/>
```

結果の順序は、特に指定しないかぎり、オプティマイザが選択するプランによって決まります。特定の順序で結果を表示したい場合は、次のように、クエリに ORDER BY 句を含めてください。

```
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
  ON Employees.DepartmentID=Departments.DepartmentID
ORDER BY EmployeeID
FOR XML RAW;
```

例

従業員が所属する部署の情報を取り出したい場合、次のように入力します。

```
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
  ON Employees.DepartmentID=Departments.DepartmentID
FOR XML RAW;
```

次の XML 文書が返されます。

```
<row EmployeeID="102" DepartmentName="R &amp; D"/>
<row EmployeeID="105" DepartmentName="R &amp; D"/>
<row EmployeeID="160" DepartmentName="R &amp; D"/>
<row EmployeeID="243" DepartmentName="R &amp; D"/>
...
```

参照

- 「FOR XML と NULL 値」678 ページ
- 「XMLELEMENT 関数 [文字列]」『SQL Anywhere サーバ SQL リファレンス』

FOR XML AUTO

AUTO モードは、XML 文書内にネストされた要素を生成します。

ELEMENTS 句を省略すると、SELECT リスト内で参照される各テーブルは、生成された XML 内で要素として表されます。ネストの順序は、SELECT リスト内でカラムが参照される順序に基づきます。SELECT リスト内の各カラムに対して属性が作成されます。

ELEMENTS 句があると、SELECT リスト内で参照される各テーブルおよびカラムは、生成された XML 内で要素として表されます。ネストの順序は、SELECT リスト内でカラムが参照される順序に基づきます。SELECT リスト内の各カラムに対して要素が作成されます。

構文

FOR XML AUTO[, **ELEMENTS**]

パラメータ

ELEMENTS　このパラメータを指定すると、FOR XML AUTO は、結果における各カラムに対し属性の代わりに XML 要素を生成します。次に例を示します。

```
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
```

```
  ON Employees.DepartmentID=Departments.DepartmentID
ORDER BY EmployeeID
FOR XML AUTO, ELEMENTS;
```

この場合、結果セット内の各カラムは、<Employees> 要素または <Departments> 要素の属性としてではなく、別個の要素として返されています。NULL 値がある場合は、その要素は、生成される XML 文書から省略されます。

```
<Employees>
  <EmployeeID>102</EmployeeID>
  <Departments>
    <DepartmentName>R &amp; D</DepartmentName>
  </Departments>
</Employees>
<Employees>
  <EmployeeID>105</EmployeeID>
  <Departments>
    <DepartmentName>R &amp; D</DepartmentName>
  </Departments>
</Employees>
<Employees>
  <EmployeeID>129</EmployeeID>
  <Departments>
  <DepartmentName>Sales</DepartmentName>
  </Departments>
</Employees>
...
```

使用法

FOR XML AUTO を使用してクエリを実行すると、BINARY、LONG BINARY、IMAGE、VARBINARY カラムのデータは、自動的に Base64 エンコードフォーマットで返されます。デフォルトでは、NULL 値は、結果から省略されます。for_xml_null_treatment オプションを EMPTY に設定すると、NULL 値を空の属性として返すことができます。

特に指定しないかぎり、データベースサーバは、テーブルのローを意味のない順序で返します。特定の順序で結果を表示したい場合、または親要素に複数の子を持たせたい場合は、クエリに ORDER BY 句を含めて、すべての子が隣接するようにします。ORDER BY 句を指定しないと、結果のネストはオプティマイザが選択するプランによって決まり、必要なネストが得られないことがあります。

FOR XML AUTO は、整形式 XML 文書を返しません。これは、文書に単一のルートノードが含まれないためです。<root> 要素が必要な場合は、1 つの方法として、XMLELEMENT 関数を使用して挿入できます。次に例を示します。

```
SELECT XMLELEMENT( NAME root,
                   (SELECT EmployeeID AS id, GivenName AS name
                   FROM Employees FOR XML AUTO ) );
```

XML 文書内で使用される属性名や要素名は、エイリアスを指定して変更できます。次のクエリは、ID 属性の名前を product_ID に変更します。

```
SELECT ID AS product_ID
FROM Products
WHERE Color='Black'
FOR XML AUTO;
```

次の XML が生成されます。

```
<Products product_ID="302"/>
<Products product_ID="400"/>
<Products product_ID="501"/>
<Products product_ID="700"/>
```

テーブルの名前をエイリアスに変更することもできます。次のクエリは、テーブルを product_info に名前を変更します。

```
SELECT ID AS product_ID
FROM Products AS product_info
WHERE Color='Black'
FOR XML AUTO;
```

次の XML が生成されます。

```
<product_info product_ID="302"/>
<product_info product_ID="400"/>
<product_info product_ID="501"/>
<product_info product_ID="700"/>
```

例

次のクエリは、<employee> 要素と <department> 要素の両方を含む XML を生成します。<employee> 要素 (SELECT リストで最初にリストされたテーブル) は、<department> 要素の親です。

```
SELECT EmployeeID, DepartmentName
FROM Employees AS employee
JOIN Departments AS department
  ON employee.DepartmentID=department.DepartmentID
ORDER BY EmployeeID
FOR XML AUTO;
```

前述のクエリによって、次の XML が生成されます。

```
<employee EmployeeID="102">
  <department DepartmentName="R &amp; D"/>
</employee>
<employee EmployeeID="105">
  <department DepartmentName="R &amp; D"/>
</employee>
<employee EmployeeID="129">
  <department DepartmentName="Sales;"/>
</employee>
<employee EmployeeID="148">
  <department DepartmentName="Finance;"/>
</employee>
...
```

次のように、SELECT リスト内でカラムの順序を変更するとします。

```
SELECT DepartmentName, EmployeeID
FROM Employees AS employee JOIN Departments AS department
  ON employee.DepartmentID=department.DepartmentID
ORDER BY 1, 2
FOR XML AUTO;
```

結果は次のようにネストされます。

```
<department DepartmentName="Finance">
  <employee EmployeeID="148"/>
```

```
    <employee EmployeeID="390"/>
    <employee EmployeeID="586"/>
    ...
  </department>
  <department DepartmentName="Marketing">
    <employee EmployeeID="184"/>
    <employee EmployeeID="207"/>
    <employee EmployeeID="318"/>
    ...
  </department>
  ...
```

ここでも、クエリによって生成された XML には、<employee> 要素と <department> 要素の両方が含まれています。しかしこの場合は、<department> 要素が <employee> 要素の親となっています。

参照

- 「for_xml_null_treatment オプション」『SQL Anywhere サーバ データベース管理』
- 「XMLELEMENT 関数 [文字列]」『SQL Anywhere サーバ SQL リファレンス』

FOR XML EXPLICIT

FOR XML EXPLICIT を使用して、クエリが返す XML 文書の構造を制御できます。クエリは特定の方法で記述して、必要なネストに関する情報がクエリ結果内で指定されるようにしてください。FOR XML EXPLICIT がサポートするオプションのディレクティブを使用すると、個別のカラムの扱いを設定できます。たとえば、あるカラムが要素内容と属性内容のどちらとして表示されるかを制御できます。また、あるカラムが生成された XML に含まれるのではなく、結果の順序付けのみに使用されるように制御できます。

パラメータ

EXPLICIT モードでは、SELECT 文の最初の 2 つのカラムに、それぞれ名前 **Tag** と **Parent** を付けてください。Tag と Parent はメタデータカラムで、それらの値は、クエリが返す XML 文書内の要素の親子関係、またはネストを決定するために使用されます。

- **Tag カラム**　これは、SELECT リスト内で最初に指定されるカラムです。Tag カラムは、現在の要素のタグ番号を格納します。タグ番号として許可されている値は、1 から 255 までです。

- **Parent カラム**　このカラムは、現在の要素の親のタグ番号を格納します。このカラムの値が NULL の場合、そのローは XML 階層のトップレベルに位置付けられています。

たとえば、FOR XML EXPLICIT が指定されていない場合に、次の結果セットを返すクエリを考えてみます

Tag	Parent	GivenName!1	ID!2
1	NULL	'Beth'	NULL
2	NULL	NULL	'102'

この例では、Tag カラムの値は、結果セット内の各要素のタグ番号です。両方のローの Parent カラムには、値 NULL が含まれています。両要素とも階層のトップレベルに生成され、クエリに FOR XML EXPLICIT 句が含まれる場合は、次の結果が得られます。

```
<GivenName>Beth</GivenName>
<ID>102</ID>
```

しかし、2 番目のローの Parent カラムが値 1 を持つ場合は、結果は次のようになります。

```
<GivenName>Beth
  <ID>102</ID>
</GivenName>
```

クエリへのデータカラムの追加

Tag カラムと Parent カラムに加えて、クエリには、1 つ以上のデータカラムを含めてください。これらのデータカラムの名前は、タグ付け中にカラムが解釈される方法を制御します。各カラム名は、感嘆符 (!) で区切られるフィールドに分割されます。次のフィールドをデータカラムに指定できます。

ElementName!*TagNumber*!*AttributeName*!*Directive*

ElementName　要素の名前。ある特定のローに関して、ローに対して生成される要素名は、一致するタグ番号を持つ最初のカラムの *ElementName* フィールドから取得されます。同じ *TagNumber* を持つ複数のカラムがある場合は、*ElementName* は、同じ *TagNumber* を持つ後続のカラムについては無視されます。前述の例では、最初のローは、<GivenName> と呼ばれる要素を生成します。

TagNumber　要素のタグ番号。ある特定のタグ値を持つローに関して、*TagNumber* フィールドに同じ値を持つすべてのカラムは、そのローに対応する要素に内容を提供します。

AttributeName　カラム値が *ElementName* 要素の属性であることを指定します。たとえば、データカラムが productID!1!Color という名前の場合、Color は <productID> 要素の属性として表示されます。

Directive　このオプションフィールドを使用して、XML 文書のフォーマットをさらに制御できます。*Directive* に対して次の値のいずれか 1 つを指定できます。

- **hide**　結果を生成するときにこのカラムが無視されることを示します。このディレクティブは、テーブルを順序付ける目的のみに使用されるカラムを含めるために使用できます。属性名は無視され、結果には含まれません。
- **element**　カラム値が、属性としてではなく、名前 *AttributeName* を持つ、ネストされた要素として挿入されることを示します。
- **xml**　カラム値が、引用されずに挿入されることを示します。*AttributeName* が指定されている場合は、値はその名前を持つ要素として挿入されます。それ以外の場合は、値は、要素がラップされずに挿入されます。このディレクティブが使用されていない場合は、カラムが XML 型でないかぎり、マークアップ文字でエスケープされます。たとえば、値 <a/> は、<a/> として挿入されます。
- **cdata**　カラム値が CDATA セクションとして挿入されることを示します。*AttributeName* は無視されます。

使用法

BINARY、LONG BINARY、IMAGE、VARBINARY カラムのデータは、FOR XML EXPLICIT を含むクエリを実行すると、自動的に Base64 エンコードフォーマットで返されます。デフォルトでは、結果セット内のすべての NULL 値は省略されます。for_xml_null_treatment オプションの設定を変更すると、この動作を変更できます。

参照

- 「for_xml_null_treatment オプション」『SQL Anywhere サーバ データベース管理』
- 「FOR XML と NULL 値」678 ページ
- 「cdata ディレクティブの使用」693 ページ
- 「xml ディレクティブの使用」692 ページ
- 「element ディレクティブの使用」690 ページ
- 「hide ディレクティブの使用」691 ページ
- 「EXPLICIT モードのクエリの記述」687 ページ
- 「クエリへのデータカラムの追加」686 ページ
- 「パラメータ」685 ページ

EXPLICIT モードのクエリの記述

次の XML 文書を生成するクエリを FOR XML EXPLICIT を使用して記述するとします。

```
<employee employeeID='129'>
  <customer customerID='107' region='Eastern'/>
  <customer customerID='119' region='Western'/>
  <customer customerID='131' region='Eastern'/>
</employee>
<employee employeeID='195'>
  <customer customerID='109' region='Eastern'/>
  <customer customerID='121' region='Central'/>
</employee>
```

このためには、次の結果セットを指定された順序どおりに返す SELECT 文を記述し、クエリに FOR XML EXPLICIT を追加します。

Tag	Parent	employee!1!employeeID	customer!2!customerID	customer!2!region
1	NULL	129	NULL	NULL
2	1	129	107	Eastern
2	1	129	119	Western
2	1	129	131	Central
1	NULL	195	NULL	NULL
2	1	195	109	Eastern
2	1	195	121	Central

クエリを記述すると、ある特定のローの一部のカラムのみが、生成された XML 文書の一部となります。カラムは、*TagNumber* フィールド (カラム名の 2 つめのフィールド) の値が、Tag カラムの値と一致する場合のみ、XML 文書に含められます。

この例では、Tag カラムに値 1 を持つ 2 つのローの場合に 3 番目のカラムが使用されます。4 番目と 5 番目のカラムでは、Tag カラムに値 2 を持つローの場合に値が使用されます。要素名は、カラム名の最初のフィールドから取得されます。この例の場合、<employee> 要素と <customer> 要素が作成されます。

属性名は、カラム名の 3 番目のフィールドから取得されます。したがって、<employee> 要素に対して **employeeID** 属性が作成され、<customer> 要素に対して **customerID** 属性と **region** 属性が作成されます。

次の手順では、SQL Anywhere サンプルデータベースを使用して、この項の最初にある XML 文書に似た XML 文書を生成する FOR XML EXPLICIT クエリを構成する方法を説明します。

例：FOR XML EXPLICIT クエリの記述

1. トップレベルの要素を生成する SELECT 文を記述します。

   この例では、クエリの最初の SELECT 文は、<employee> 要素を生成します。クエリの最初の 2 つの値は、Tag カラム値と Parent カラム値にしてください。<employee> 要素は、階層のトップにあるため、Tag 値に 1 を、Parent 値に NULL を割り当ててください。

   > **注意**
   > UNION を使用する EXPLICIT モードのクエリを記述する場合、最初の SELECT 文で指定されたカラム名のみが使用されます。要素名または属性名として使用するカラム名は、最初の SELECT 文で指定してください。これは、後続の SELECT 文で指定されたカラム名は無視されるためです。

2. 前述のテーブルの <employee> 要素を生成するためには、最初の SELECT 文は、次のようになります。

```
SELECT
    1          AS tag,
    NULL       AS parent,
    EmployeeID AS [employee!1!employeeID],
    NULL       AS [customer!2!customerID],
    NULL       AS [customer!2!region]
FROM Employees;
```

3. 子要素を生成する SELECT 文を記述します。

   2 つめのクエリは、<customer> 要素を生成します。これは EXPLICIT モードクエリのため、すべての SELECT 文において、最初の 2 つの値に Tag 値と Parent 値を指定してください。<customer> 要素には、タグ番号 2 を与えます。また、この要素は <employee> 要素の子であるため、Parent 値は 1 になります。最初の SELECT 文で、すでに EmployeeID、CustomerID、Region は属性であると指定しています。

```
SELECT
    2,
    1,
    EmployeeID,
```

```
        CustomerID,
        Region
    FROM Employees KEY JOIN SalesOrders
```

4. クエリに UNION DISTINCT を追加して、2 つの SELECT 文を結合します。

```
    SELECT
        1         AS tag,
        NULL      AS parent,
        EmployeeID AS [employee!1!employeeID],
        NULL      AS [customer!2!customerID],
        NULL      AS [customer!2!region]
    FROM Employees
    UNION DISTINCT
    SELECT
        2,
        1,
        EmployeeID,
        CustomerID,
        Region
    FROM Employees KEY JOIN SalesOrders
```

5. ORDER BY 句を追加して、結果内のローの順序を指定します。ローの順序は、生成される文書内で使用される順序です。

```
SELECT
    1         AS tag,
    NULL      AS parent,
    EmployeeID AS [employee!1!employeeID],
    NULL      AS [customer!2!customerID],
    NULL      AS [customer!2!region]
FROM Employees
UNION DISTINCT
SELECT
    2,
    1,
    EmployeeID,
    CustomerID,
    Region
FROM Employees KEY JOIN SalesOrders
ORDER BY 3, 1
FOR XML EXPLICIT;
```

FOR XML EXPLICIT の例

次のクエリ例は、従業員による発注に関する情報を取り出します。この例では、<employee>、<order>、<department> という 3 種類の要素があります。<employee> 要素は ID 属性と name 属性を持ち、<order> 要素は date 属性を、また <department> 要素は name 属性を持ちます。

```
SELECT
    1         tag,
    NULL      parent,
    EmployeeID  [employee!1!id],
    GivenName   [employee!1!name],
    NULL      [order!2!date],
    NULL      [department!3!name]
FROM Employees
UNION DISTINCT
SELECT
    2,
    1,
    EmployeeID,
```

```
        NULL,
        OrderDate,
        NULL
FROM Employees KEY JOIN SalesOrders
UNION DISTINCT
SELECT
        3,
        1,
        EmployeeID,
        NULL,
        NULL,
        DepartmentName
FROM Employees e JOIN Departments d
  ON e.DepartmentID=d.DepartmentID
ORDER BY 3, 1
FOR XML EXPLICIT;
```

このクエリから次の結果が得られます。

```
<employee id="102" name="Fran">
  <department name="R &amp; D"/>
</employee>
<employee id="105" name="Matthew">
  <department name="R &amp; D"/>
</employee>
<employee id="129" name="Philip">
  <order date="2000-07-24"/>
  <order date="2000-07-13"/>
  <order date="2000-06-24"/>
  <order date="2000-06-08"/>
  ...
  <department name="Sales"/>
</employee>
<employee id="148" name="Julie">
  <department name="Finance"/>
</employee>
...
```

element ディレクティブの使用

属性ではなくサブ要素を生成する場合は、次のように、クエリに element ディレクティブを追加します。

```
SELECT
        1         tag,
        NULL      parent,
        EmployeeID [employee!1!id!element],
        GivenName  [employee!1!name!element],
        NULL      [order!2!date!element],
        NULL      [department!3!name!element]
FROM Employees
UNION DISTINCT
SELECT
        2,
        1,
        EmployeeID,
        NULL,
        OrderDate,
        NULL
FROM Employees KEY JOIN SalesOrders
UNION DISTINCT
SELECT
        3,
```

```
        1,
        EmployeeID,
        NULL,
        NULL,
        DepartmentName
FROM Employees e JOIN Departments d
  ON e.DepartmentID=d.DepartmentID
ORDER BY 3, 1
FOR XML EXPLICIT;
```

このクエリから次の結果が得られます。

```
<employee>
  <id>102</id>
  <name>Fran</name>
  <department>
    <name>R &amp; D</name>
  </department>
</employee>
<employee>
  <id>105</id>
  <name>Matthew</name>
  <department>
    <name>R &amp; D</name>
  </department>
</employee>
<employee>
  <id>129</id>
  <name>Philip</name>
  <order>
    <date>2000-07-24</date>
  </order>
  <order>
    <date>2000-07-13</date>
  </order>
  <order>
    <date>2000-06-24</date>
  </order>
  ...
  <department>
    <name>Sales</name>
  </department>
</employee>
...
```

hide ディレクティブの使用

次のクエリでは、employee ID は、結果の順序付けに使用されていますが、hide ディレクティブが指定されているため、結果には表示されません。

```
SELECT
        1          tag,
        NULL       parent,
        EmployeeID  [employee!1!id!hide],
        GivenName   [employee!1!name],
        NULL       [order!2!date],
        NULL       [department!3!name]
FROM Employees
UNION DISTINCT
SELECT
        2,
        1,
        EmployeeID,
```

```
        NULL,
        OrderDate,
        NULL
FROM Employees KEY JOIN SalesOrders
UNION DISTINCT
SELECT
        3,
        1,
        EmployeeID,
        NULL,
        NULL,
        DepartmentName
FROM Employees e JOIN Departments d
  ON e.DepartmentID=d.DepartmentID
ORDER BY 3, 1
FOR XML EXPLICIT;
```

このクエリは、次の結果を返します。

```
<employee name="Fran">
  <department name="R &amp; D"/>
</employee>
<employee name="Matthew">
  <department name="R &amp; D"/>
</employee>
<employee name="Philip">
  <order date="2000-04-21"/>
  <order date="2001-07-23"/>
  <order date="2000-12-30"/>
  <order date="2000-12-20"/>
  ...
  <department name="Sales"/>
</employee>
<employee name="Julie">
  <department name="Finance"/>
</employee>
...
```

xml ディレクティブの使用

デフォルトでは、FOR XML EXPLICIT クエリの結果に XML 文字として有効ではない文字が含まれる場合、カラムが XML 型でないかぎり、無効な文字はエスケープされます。詳細については、「不正な XML 名のエンコーディングのルール」679 ページを参照してください。

たとえば、次のクエリは、アンパサンド (&) を含む XML を生成します。

```
SELECT
        1           AS tag,
        NULL          AS parent,
        ID           AS [customer!1!id!element],
        CompanyName    AS [customer!1!company!element]
FROM Customers
WHERE ID = '115'
FOR XML EXPLICIT;
```

このクエリによって生成される結果では、アンパサンドはエスケープされます。これは、このカラムが XML 型ではないためです。

```
<customer><id>115</id>
<company>Sterling &amp; Co.</company>
</customer>
```

xml ディレクティブは、生成される XML にカラム値がエスケープされずに挿入されることを示します。前述のクエリに xml ディレクティブを付けて実行します。

```
SELECT
      1            AS tag,
      NULL           AS parent,
      ID           AS [customer!1!id!element],
      CompanyName    AS [customer!1!company!xml]
FROM Customers
WHERE ID = '115'
FOR XML EXPLICIT;
```

結果内で、アンパサンドはエスケープされていません。

```
<customer>
 <id>115</id>
 <company>Sterling & Co.</company>
</customer>
```

この XML は、アンパサンドが含まれるため、整形式ではありません。アンパサンドは、XML においては特別な文字です。クエリを使用して XML を生成する場合は、その XML が整形式であり、妥当であることを確認してください。SQL Anywhere は、生成される XML が整形式または妥当であるかをチェックしません。

xml ディレクティブを指定すると、*AttributeName* フィールドは属性ではなく要素を生成するために使用されます。

cdata ディレクティブの使用

次のクエリは、cdata ディレクティブを使用して、製品名を CDATA セクションに入れて返します。

```
SELECT
      1            AS tag,
      NULL           AS parent,
      ID           AS [product!1!id],
      Description     AS [product!1!!cdata]
FROM Products
FOR XML EXPLICIT;
```

このクエリによって生成される結果は、各製品の説明を CDATA セクション内にリストします。CDATA セクションに含まれるデータは、引用されません。

```
<product id="300">
  <![CDATA[Tank Top]]>
</product>
<product id="301">
  <![CDATA[V-neck]]>
</product>
<product id="302">
  <![CDATA[Crew Neck]]>
</product>
<product id="400">
  <![CDATA[Cotton Cap]]>
</product>
...
```

結果を表示するための Interactive SQL の使用

FOR XML クエリの結果は文字列で返されます。多くの場合、文字列の結果は長くなります。Interactive SQL には、**[ウィンドウで表示]** オプションを使用して整形式 XML 文書の構造を表示する機能があります。

FOR XML クエリの結果は、<?xml?> タグを含め、任意でタグのペアで囲んで (<root>...</root> など)、整形式 XML 文書にキャストできます。次のクエリは、これを行う方法を示します。

```
SELECT XMLCONCAT( CAST('<?xml version="1.0"?>' AS XML),
  XMLELEMENT( NAME root, (
    SELECT
        1          AS tag,
        NULL       AS parent,
        EmployeeID AS [employee!1!employeeID],
        NULL       AS [customer!2!customerID],
        NULL       AS [customer!2!region],
        NULL       AS [custname!3!given_name!element],
        NULL       AS [custname!3!surname!element]
    FROM Employees
    UNION DISTINCT
    SELECT
        2,
        1,
        EmployeeID,
        CustomerID,
        Region,
        NULL,
        NULL
    FROM Employees KEY JOIN SalesOrders
    UNION DISTINCT
    SELECT
        3,
        2,
        EmployeeID,
        CustomerID,
        NULL,
        Customers.GivenName,
        Customers.SurName
    FROM SalesOrders
    JOIN Customers
       ON SalesOrders.CustomerID = Customers.ID
    JOIN Employees
       ON SalesOrders.SalesRepresentative = Employees.EmployeeID
    ORDER BY 3, 4, 1
    FOR XML EXPLICIT
  ) )
);
```

Interactive SQL のカラムの **[トランケーションの長さ]** の値は、カラム全体をフェッチできる大きさに設定する必要があります。これは、**[ツール]** » **[オプション]** メニューを使用するか、次のような Interactive SQL 文を実行して行うことができます。

```
SET OPTION truncation_length = 80000;
```

XML 文書の結果を表示するには、**[結果]** ウィンドウ枠でカラムの内容をダブルクリックし、**[XML アウトライン]** タブを選択します。

参照

- 「Interactive SQL での HTML データと XML データの表示」『SQL Anywhere サーバ データベース管理』

クエリ結果を XML として取得するための SQL/XML の使用

SQL/XML は、XML を SQL 言語に機能統合する方法を定める、ドラフト段階の標準です。SQL/XML は、XML とともに SQL を使用する方法を定めています。サポートされる関数を使用して、リレーショナルデータから XML 文書を構成するクエリを記述できます。

無効な名前と SQL/XML

SQL/XML では、スペースを含む式など、有効な XML 名でない式は、FOR XML 句と同様にエスケープされます。XML 型の要素の内容は、引用されません。

XML データ型の使用については、「リレーショナルデータベースにおける XML 文書の格納」667 ページを参照してください。

参照

- 「不正な XML 名のエンコーディングのルール」679 ページ

XMLAGG 関数の使用

XMLAGG 関数は、XML 要素の集合から XML 要素のフォレストを生成するために使用されます。XMLAGG は、集合関数で、クエリ内のすべてのローに対して単一の集約された XML 結果を生成します。

次のクエリでは、XMLAGG は、各ローに対し <name> 要素を生成するために使用されています。<name> 要素は、従業員名で順序付けされています。ORDER BY 句は、XML 要素を順序付けるために指定されています。

```
SELECT XMLELEMENT( NAME Departments,
                   XMLATTRIBUTES ( DepartmentID ),
                   XMLAGG( XMLELEMENT( NAME name,
                           Surname )
                           ORDER BY Surname )
                 ) AS department_list
FROM Employees
GROUP BY DepartmentID
ORDER BY DepartmentID;
```

このクエリは、次の結果を生成します。

department_list
<Departments DepartmentID="100"> <name>Breault</name> <name>Cobb</name> <name>Diaz</name> <name>Driscoll</name> ... </Departments>
<Departments DepartmentID="200"> <name>Chao</name> <name>Chin</name> <name>Clark</name> <name>Dill</name> ... </Departments>
<Departments DepartmentID="300"> <name>Bigelow</name> <name>Coe</name> <name>Coleman</name> <name>Davidson</name> ... </Departments>
...

参照

- 「XMLAGG 関数 [集合]」『SQL Anywhere サーバ SQL リファレンス』

XMLCONCAT 関数の使用

XMLCONCAT 関数は、渡されるすべての XML 値を連結して、XML 要素のフォレストを作成します。たとえば、次のクエリは、Employees テーブルの従業員ごとに、<given_name> 要素と <surname> 要素を連結します。

```
SELECT XMLCONCAT( XMLELEMENT( NAME given_name, GivenName ),
                  XMLELEMENT( NAME surname, Surname )
                 ) AS "Employee_Name"
FROM Employees;
```

このクエリは、次の結果を返します。

Employee_Name
<given_name>Fran</given_name> <surname>Whitney</surname>
<given_name>Matthew</given_name> <surname>Cobb</surname>
<given_name>Philip</given_name> <surname>Chin</surname>

Employee_Name
<given_name>Julie</given_name> <surname>Jordan</surname>
...

参照

- 「XMLCONCAT 関数 [文字列]」『SQL Anywhere サーバ SQL リファレンス』

XMLELEMENT 関数の使用

XMLELEMENT 関数は、リレーショナルデータから XML 要素を構成します。生成される要素の内容を指定できます。また、その要素の属性と、属性の内容も指定できます。

ネストされた要素の生成

次のクエリは、ネストされた XML を生成します。製品ごとに <product_info> 要素が生成され、その中に各製品の名前、数量、説明を示す要素が生成されます。

```
SELECT ID,
XMLELEMENT( NAME product_info,
        XMLELEMENT( NAME item_name, Products.name ),
        XMLELEMENT( NAME quantity_left, Products.Quantity ),
        XMLELEMENT( NAME description, Products.Size || ' ' ||
                Products.Color || ' ' || Products.name )
      ) AS results
FROM Products
WHERE Quantity > 30;
```

このクエリは、次の結果を生成します。

ID	results
301	<product_info> <item_name>Tee Shirt </item_name> <quantity_left>54 </quantity_left> <description>Medium Orange Tee Shirt</description> </product_info>
302	<product_info> <item_name>Tee Shirt </item_name> <quantity_left>75 </quantity_left> <description>One Size fits all Black Tee Shirt </description> </product_info>

ID	results
400	<product_info> <item_name>Baseball Cap </item_name> <quantity_left>112 </quantity_left> <description>One Size fits all Black Baseball Cap </description> </product_info>
...	...

要素の内容の指定

XMLELEMENT 関数を使用して、要素の内容を指定できます。次の文は、内容 **hat** を持つ XML 要素を生成します。

```
SELECT ID, XMLELEMENT( NAME product_type, 'hat' )
FROM Products
WHERE Name IN ( 'Baseball Cap', 'Visor' );
```

属性を持つ要素の生成

クエリに XMLATTRIBUTES 引数を含めると、要素に属性を追加できます。この引数は、属性名と内容を指定します。次の文は、各品目の name、Color、UnitPrice に対して属性を生成します。

```
SELECT ID, XMLELEMENT( NAME item_description,
                    XMLATTRIBUTES( Name,
                                   Color,
                                   UnitPrice )
                  ) AS item_description_element
FROM Products
WHERE ID > 400;
```

AS 句を指定して、属性に名前を付けることができます。

```
SELECT ID, XMLELEMENT( NAME item_description,
                    XMLATTRIBUTES ( Color AS color,
                                    UnitPrice AS price ),
                    Products.Name
                  ) AS products
FROM Products
WHERE ID > 400;
```

例

次の例では HTTP Web サービスで XMLELEMENT を使用します。

```
CREATE OR REPLACE PROCEDURE "DBA"."http_header_example_with_table_proc"()
RESULT ( res LONG VARCHAR )
BEGIN
   DECLARE var LONG VARCHAR;
   DECLARE varval LONG VARCHAR;
   DECLARE I INT;
   DECLARE res LONG VARCHAR;
   DECLARE htmltable XML;
   SET var  = NULL;
loop_h:
```

```
    LOOP
       SET var = NEXT_HTTP_HEADER( var );
       IF var IS NULL THEN LEAVE loop_h END IF;
       SET varval = http_header( var );
       -- ... do some action for <var,varval> pair...
       SET htmltable = htmltable ||
          XMLELEMENT( name "tr",
          XMLATTRIBUTES( 'left' AS "align", 'top' AS "valign" ),
          XMLELEMENT( name "td", var ),
          XMLELEMENT( name "td", varval ) )  ;
    END LOOP;

    SET res = XMLELEMENT( NAME "table",
       XMLATTRIBUTES( '' AS "BORDER", '10' as "CELLPADDING", '0' AS "CELLSPACING" ),

       XMLELEMENT( NAME "th",
          XMLATTRIBUTES( 'left' AS "align", 'top' AS "valign" ),
          'Header Name' ),

       XMLELEMENT( NAME "th",
           XMLATTRIBUTES( 'left' AS "align", 'top' AS "valign" ),
           'Header Value' ),

       htmltable);
    SELECT res;
 END;
```

参照

- 「XMLELEMENT 関数 [文字列]」『SQL Anywhere サーバ SQL リファレンス』

XMLFOREST 関数の使用

XMLFOREST は、XML 要素のフォレストを構成します。各 XMLFOREST 引数に対して、1 つの要素が生成されます。

次のクエリは、<item_description> 要素を生成します。この要素には、<name>、<color>、<price> 要素があります。

```
SELECT ID, XMLELEMENT( NAME item_description,
                  XMLFOREST( Name as name,
                             Color as color,
                             UnitPrice AS price )
                  ) AS product_info
FROM Products
WHERE ID > 400;
```

このクエリによって、次の結果が生成されます。

ID	product_info
401	<item_description> <name>Baseball Cap</name> <color>White</color> <price>10.00</price> </item_description>

ID	product_info
500	<item_description> <name>Visor</name> <color>White</color> <price>7.00</price> </item_description>
501	<item_description> <name>Visor</name> <color>Black</color> <price>7.00</price> </item_description>
...	...

参照

- 「XMLFOREST 関数 [文字列]」『SQL Anywhere サーバ SQL リファレンス』

XMLGEN 関数の使用

XMLGEN 関数は、XQuery コンストラクタに基づいて XML 値を生成するために使用されます。

次のクエリによって生成される XML は、SQL Anywhere サンプルデータベース内の顧客の注文に関する情報を提供します。このクエリでは、次の変数参照を使用します。

- **{$ID}** SalesOrders テーブルの ID カラムの値を使用して、<ID> 要素の内容を生成します。
- **{$OrderDate}** SalesOrders テーブルの OrderDate カラムの値を使用して、<date> 要素の内容を生成します。
- **{$Customers}** Customers テーブルの CompanyName カラムから <customer> 要素の内容を生成します。

```
SELECT XMLGEN ( '<order>
        <ID>{$ID}</ID>
        <date>{$OrderDate}</date>
        <customer>{$Customers}</customer>
        </order>',
        SalesOrders.ID,
        SalesOrders.OrderDate,
        Customers.CompanyName AS Customers
        ) AS order_info
FROM SalesOrders JOIN Customers
ON Customers.ID = SalesOrders.CustomerID
ORDER BY SalesOrders.CustomerID;
```

このクエリは、次の結果を生成します。

order_info
<order> <ID>2001</ID> <date>2000-03-16</date> <customer>The Power Group</customer> </order>
<order> <ID>2005</ID> <date>2001-03-26</date> <customer>The Power Group</customer> </order>
<order> <ID>2125</ID> <date>2001-06-24</date> <customer>The Power Group</customer> </order>
<order> <ID>2206</ID> <date>2000-04-16</date> <customer>The Power Group</customer> </order>
...

属性の生成

注文 ID 番号を <order> 要素の属性としたい場合は、次のようにクエリを記述します (変数参照が二重引用符で囲まれいます。これは、属性値を指定しているためです)。

```
SELECT XMLGEN ( '<order ID="{$ID}">
                <date>{$OrderDate}</date>
                <customer>{$Customers}</customer>
                </order>',
               SalesOrders.ID,
               SalesOrders.OrderDate,
               Customers.CompanyName AS Customers
             ) AS order_info
FROM SalesOrders JOIN Customers
ON Customers.ID = SalesOrders.CustomerID
ORDER BY SalesOrders.OrderDate;
```

このクエリは、次の結果を生成します。

order_info
<order ID="2131"> <date>2000-01-02</date> <customer>BoSox Club</customer> </order>
<order ID="2065"> <date>2000-01-03</date> <customer>Bloomfield's</customer> </order>

order_info
<order ID="2126"> <date>2000-01-03</date> <customer>Leisure Time</customer> </order>
<order ID="2127"> <date>2000-01-06</date> <customer>Creative Customs Inc.</customer> </order>
...

両方の結果セットにおいて、顧客名 Bloomfield's は、Bloomfield's と引用されています。これは、アポストロフィは XML において特別な文字であり、<customer> 要素が生成される元となったカラムは XML 型ではないためです。

XMLGEN での無効な文字の引用の詳細については、「無効な名前と SQL/XML」695 ページを参照してください。

XML 文書のヘッダ情報の指定

SQL Anywhere がサポートする FOR XML 句と SQL/XML 関数は、生成する XML 文書にバージョン宣言情報を含めません。XMLGEN 関数を使用すると、ヘッダ情報を生成できます。

```
SELECT XMLGEN( '<?xml version="1.0"
               encoding="ISO-8859-1" ?>
               <r>{$x}</r>',
               (SELECT GivenName, Surname
      FROM Customers FOR XML RAW) AS x );
```

これは、次の結果を生成します。

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<r>
 <row GivenName="Michaels" Surname="Devlin"/>
 <row GivenName="Beth" Surname="Reiser"/>
 <row GivenName="Erin" Surname="Niedringhaus"/>
 <row GivenName="Meghan" Surname="Mason"/>
 ...
</r>
```

参照

- 「XMLGEN 関数 [文字列]」『SQL Anywhere サーバ SQL リファレンス』

データベース内の JSON

JavaScript Object Notation (JSON) は、言語に依存しないテキストベースのデータ交換フォーマットで、JavaScript データの直列化のために開発されました。

JSON には、次の 4 つの基本型があります。文字列、数値、ブール、および NULL です。また、JSON には、次の 2 つの構造化型もあります。オブジェクトと配列です。

JSON フォーマットの詳細については、http://www.json.org を参照してください。

クエリ結果を JSON として取り出すための FOR JSON 句の使用

SQL Anywhere では、SELECT 文内で FOR JSON 句を使用して、データベースに対し SQL クエリを実行して、結果を JSON 文書として返すことができます。

FOR JSON 句は、サブクエリ、GROUP BY 句または集合関数のあるクエリ、ビュー定義など、どのような SELECT 文内でも使用できます。FOR JSON 句の使用は、配列、オブジェクト、スカラ要素で構成される JSON 配列としてのリレーショナルデータを示します。

FOR JSON 句内で、生成される JSON のフォーマットを制御する次の JSON モードのうちの 1 つを指定できます。

- **RAW**　クエリ結果をフラットにされた JSON 表現として返します。このモードはより冗長性が高いですが、解析が容易です。
- **AUTO**　クエリのジョインに基づいて、クエリの結果をネストされた JSON オブジェクトとして返します。
- **EXPLICIT**　カラムデータの表現方法を指定できます。カラムを単一の値、オブジェクト、またはネストされたオブジェクトとして指定して、統一された配列または異なる配列を生成できます。

参照

- 「FOR JSON RAW」703 ページ
- 「FOR JSON AUTO」704 ページ
- 「FOR JSON EXPLICIT」705 ページ
- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ARRAY コンストラクタ [複合]」『SQL Anywhere サーバ SQL リファレンス』

FOR JSON RAW

クエリで FOR JSON RAW を使用すると、各ローがフラットにされた JSON 表現として返されます。

構文

FOR JSON RAW

使用法

この句は、解析と理解が最も容易な方法であるため、クエリ結果を JSON オブジェクトとして取得する推奨の方法です。

例

次のクエリは、FOR JSON RAW を使用して、Employees テーブルから従業員情報を返します。

```
SELECT
  emp.EmployeeID,
  so.CustomerID,
  so.Region
FROM Employees AS emp KEY JOIN SalesOrders AS so WHERE emp.EmployeeID <= 195
ORDER BY 1
FOR JSON RAW;
```

結果を階層でネストするような、FOR JSON AUTO を使用した場合に返される結果とは異なり、FOR JSON RAW はフラット化された結果セットを返します。

```
[
  { "EmployeeID" : 129, "CustomerID" : 107, "Region" : "Eastern" },
  { "EmployeeID" : 129, "CustomerID" : 119, "Region" : "Western" },
  ...
  { "EmployeeID" : 129, "CustomerID" : 131, "Region" : "Eastern" },
  { "EmployeeID" " 195, "CustomerID" : 176, "Region" : "Eastern" }
]
```

参照

- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』

FOR JSON AUTO

クエリで FOR JSON AUTO を指定すると、クエリのジョインに基づいて、クエリが JSON オブジェクトのネストされた階層を返します。

構文

FOR JSON AUTO

使用法

結果セットで JSON オブジェクト間の階層関係を表示する場合は、クエリで FOR JSON AUTO 句を使用します。

例

次の例は、**emp** オブジェクトの JSON 配列を返します。それぞれに EmployeeID と **so** オブジェクトが含まれます。**so** オブジェクトは、CustomerID と地域で構成されるオブジェクトの配列です。

```
SELECT
  emp.EmployeeID,
```

```
  so.CustomerID,
  so.Region
FROM Employees AS emp KEY JOIN SalesOrders AS so WHERE emp.EmployeeID <= 195
ORDER BY 1
FOR JSON AUTO;
```

FOR JSON RAW とは異なり、FOR JSON AUTO を使用すると、データのネストされた階層が返され、**emp** または Employee オブジェクトが CustomerID データの配列を含む **so** または SalesOrders オブジェクトで構成されています。

```
[
  { "emp":
    { "EmployeeID" : 129,
    "so" : [
      { "CustomerID" : 107 , "Region" : "Eastern" },
      ...
      { "CustomerID" : 131 , "Region" : "Eastern" }
    ]
    }
  },
  { "emp" :
    { "EmployeeID" : 195,
    "so" : [
      { "CustomerID" : 109 , "Region" : "Eastern" },
      ...
      { "CustomerID" : 176 , "Region" : "Eastern" }
        ]
    }
  }
]
```

参照

- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ARRAY コンストラクタ [複合]」『SQL Anywhere サーバ SQL リファレンス』

FOR JSON EXPLICIT

配列で FOR JSON EXPLICIT を指定すると、列を単純な値、オブジェクト、およびネストされた階層オブジェクトとして指定して、統一された配列または異なる配列を生成できます。

構文

FOR JSON EXPLICIT

使用法

FOR JSON EXPLICIT は、カラムエイリアスを使用して、詳細なフォーマット仕様を提供します。エイリアスが存在しない場合は、指定されたカラムが値として出力されます。エイリアスは、ネストされた構造内で値 (またはオブジェクト) を表すのに必要です。

最初の 2 つのカラム select-list **tag** と **parent** に名前を付ける必要があります。複数のクエリを使用すると、各クエリ内のタグと親の関係を指定して、JSON 出力を返すことができます。

エイリアスディレクティブのフォーマットは [encapsulating_object!tag!name!qualifier] で、

- **!** がディレクティブの基準を区切ります。
- **encapsulating_object** は、select-list 項目のカプセル化 (配列) オブジェクトを生成します。
- **tag** は、後続のクエリで使用されるカラムの識別子を定義します。ネスティングの基準 (親に対して) も作成します。
- **name** は、(名前と値のペア) オブジェクトの名前を割り当てます。
- **qualifier** は **element** (デフォルト) または **hide** のいずれかで、結果セットの要素を難読化します。

例

次のクエリは、FOR JSON EXPLICIT を使用して、Employees テーブルから従業員情報を返します。

```
SELECT
  1              AS tag,
  NULL             AS parent,
  emp.EmployeeID   AS [!1!EmployeeID],
  so.CustomerID    AS [!1!CustomerID],
  so.Region        AS [!1!Region]
FROM Employees AS emp KEY JOIN SalesOrders AS so WHERE emp.EmployeeID <= 195
ORDER BY 3
FOR JSON EXPLICIT;
```

結果は FOR JSON RAW の例と同じです。

```
[
  { "EmployeeID" : 129, "CustomerID" : 107, "Region" : "Eastern" },
  { "EmployeeID" : 129, "CustomerID" : 119, "Region" : "Western" },
  ...
  { "EmployeeID" : 129, "CustomerID" : 131, "Region" : "Eastern" },
  { "EmployeeID" " 195, "CustomerID" : 176, "Region" : "Eastern" }
]
```

次の例は、FOR JSON AUTO の例に似た結果を返します。

```
SELECT
    1            AS tag,
    NULL           AS parent,
    emp.EmployeeID AS [emp!1!EmployeeID],
    null          AS [so!2!CustomerID],
    null          AS [!2!Region]
FROM Employees as emp where emp.EmployeeID <= 195
UNION ALL
SELECT
    2,
    1,
    emp.EmployeeID,
    so.CustomerID,
    so.Region
FROM Employees as emp KEY JOIN SalesOrders as so where emp.EmployeeID <= 195
ORDER BY 3, 1
FOR JSON EXPLICIT;
```

上記のクエリは、次の結果を返します。

```
[
  {"emp": [{"EmployeeID":102}]},
  {"emp":[{"EmployeeID":105}]},
  {"emp":
    [{"EmployeeID":129,
      "so":[
        {"CustomerID":101,"Region":"Eastern"},
        ...
        {"CustomerID":205,"Region":"Eastern"}
      ]
    }]
  },
  {"emp":[{"EmployeeID":148}]},
  {"emp":[{"EmployeeID":160}]},
  {"emp":[{"EmployeeID":184}]},
  {"emp":[{"EmployeeID":191}]},
  {"emp":
    [{"EmployeeID":195,
      "so":[
        {"CustomerID":101,"Region":"Eastern"},
    ...
        {"CustomerID":209,"Region":"Western"}
      ]
    }]
  }
]
```

配列の順序と受注のない従業員を含めること以外で、上記のフォーマットが FOR JSON AUTO の結果と異なるのは、**emp** が構造の配列であることだけです。FOR JSON AUTO では、**emp** だけが単一のオブジェクトを持っていると理解されます。FOR JSON EXPLICIT は、集合をサポートする配列のカプセル化を使用します。

次の例では、**emp** のカプセル化を削除して、値として Region を返します。この例は、FOR JSON EXPLICIT モードが細かいフォーマット制御を行って、RAW モードと AUTO モード間で対象を生成する方法を示します。

```
SELECT
    1            AS tag,
    NULL         AS parent,
    emp.EmployeeID AS [!1!EmployeeID],          // remove "emp" encapsulation
    null         AS [so!2!id],                  // change "CustomerID" to just "id"
    null         AS [!2!]                       // stipulate that region should be emitted as a value
FROM Employees AS emp WHERE emp.EmployeeID <= 195
UNION ALL
SELECT
    2,
    1,
    emp.EmployeeID,
    so.CustomerID,
    so.Region
FROM Employees as emp KEY JOIN SalesOrders AS so WHERE emp.EmployeeID <= 195
ORDER BY 3, 1
FOR JSON EXPLICIT;
```

クエリの結果で、**so** はオブジェクトの配列ではなくなっていますが、2 次元の配列となっています。

```
[
  {"EmployeeID":102},{"EmployeeID":105},{"EmployeeID":129,
    "so":[
      [{"id":101},"Eastern"],
```

```
      ...
      [{"id":205},"Eastern"]
    ]
  },
  {"EmployeeID":148},
  {"EmployeeID":160},
  {"EmployeeID":184},
  {"EmployeeID":191},
  {"EmployeeID":195,
    "so":[
      [{"id":101},"Eastern"],
      ...
      [{"id":209},"Western"]
    ]
  }
]
```

次の例は、FOR JSON RAW を使用した場合に似ていますが、employeeID、CustomerID、Region が名前／値のペアとしてではなく、値として出力されます。

```
SELECT
    1           AS tag,
    NULL            AS parent,
    emp.EmployeeID,    // no alias directives
    so.CustomerID,
    so.Region
FROM Employees AS emp KEY JOIN SalesOrders AS so WHERE emp.EmployeeID <= 195
ORDER BY 3
FOR JSON EXPLICIT;
```

クエリが、EmployeeID、CustomerID、Region で構成される 2 次元配列を生成する結果を返します。

```
[
  [129,107,"Eastern"],
  ...
  [195,176,"Eastern"]
]
```

参照

- 「ARRAY コンストラクタ [複合]」『SQL Anywhere サーバ SQL リファレンス』

データのインポートとエクスポート

「バルクオペレーション」という用語は、データのインポートやエクスポートのプロセスを説明するために使用されます。バルクオペレーションは一般的なエンドユーザアプリケーションの一部ではなく、実行するには特別な権限が必要です。バルクオペレーションは同時実行性とトランザクションログに影響する可能性があるため、ユーザがデータベースに接続していないときに実行する必要があります。

データをインポートおよびエクスポートする際の、一般的な状況を次に示します。

- 新しいデータベースに最初のデータセットをインポートする
- データベースの構造を修正した場合などに、新しいデータベースを構築する
- スプレッドシートなど、他のアプリケーションで使うためにデータベースからデータをエクスポートする
- レプリケーションまたは同期に使用するデータベースの抽出を作成する
- 破損したデータベースを修復する
- データベースを再構築してパフォーマンスを向上させる
- 新しいバージョンのデータベースソフトウェアを入手し、ソフトウェアアップグレードを完了する

バルクオペレーションのパフォーマンスの側面

バルクオペレーションのパフォーマンスは、オペレーションがデータベースサーバの内部と外部のどちらに対するものかなどの要因によって決まります。

内部バルクオペレーション

内部バルクオペレーションは、LOAD TABLE および UNLOAD 文を使用してデータベース サーバによって実行されるインポートおよびエクスポート操作であり、**サーバ側**バルクオペレーションとも呼ばれます。

内部バルクオペレーションを実行する場合は、ASCII テキストファイルまたは Adaptive Server Enterprise の BCP ファイルからロードしたり、これらのファイルにアンロードすることができます。これらのファイルは、データベース サーバと同じコンピュータに存在することも、クライアントコンピュータに存在することもできます。書き込みまたは読み込み対象のファイルのパスは、データベースサーバからの相対パスとして指定します。内部バルクオペレーションは、データベースのデータのインポートおよびエクスポート方法としては最速です。

外部バルクオペレーション

外部バルクオペレーションは、INPUT および OUTPUT 文を使用して Interactive SQL などのクライアントによって実行されるインポートおよびエクスポート操作であり、**クライアント側**バルクオペレーションとも呼ばれます。クライアントが INPUT 文を発行すると、INPUT 文で指定され

たファイルの処理時に読み込まれた各ローに対して、トランザクションログに INSERT 文が記録されます。このため、クライアント側ロードはサーバ側ロードよりかなり遅くなります。また、INPUT 中に INSERT トリガが起動されます。

OUTPUT 文により、SELECT 文の結果セットを複数の異なるファイルフォーマットで書き出すことが可能になります。

外部バルクオペレーションの場合、読み込みまたは書き込み対象のファイルのパスは、クライアントアプリケーションが実行されているコンピュータからの相対パスとして指定します。

参照

- 「LOAD TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「UNLOAD 文」『SQL Anywhere サーバ SQL リファレンス』
- 「INPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「データのインポートのパフォーマンスに関するヒント」711 ページ
- 「-b データベースサーバオプション」『SQL Anywhere サーバ データベース管理』

バルクオペレーションのデータリカバリの問題

バルクオペレーションモード (-b サーバオプション) でデータベースサーバを実行できます。このオプションを使用した場合、データベースサーバでは一部の重要な機能を実行しません。具体的には、次のとおりです。

関数	影響
トランザクションログの管理	変更の記録がありません。COMMIT を実行するたびにチェックポイントが設定されます。
レコードのロック	重大な影響はありません。

また、バルクロードしたデータをリカバリ時にも使用できるようにすることが必要な場合もあります。そのためには、元のデータソースを元の場所にそのまま残します。また、LOAD TABLE 文の一部のロギングオプションを使用して、バルクロードしたデータをトランザクションログに記録することもできます。

警告
バルクオペレーションモードでは、メディア障害に対してデータベースが防御されないので、このモードの使用前後には、データベースのバックアップを行ってください。

参照

- 「-b データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「LOAD TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

データインポート

データのインポートは、バルクオペレーションとしてのデータベースへのデータの読み込みに関連する管理作業です。SQL Anywhere を使用して、次の作業を実行できます。

- テキストファイルからテーブル全体またはテーブルの一部をインポートする
- 変数からデータをインポートする
- スクリプトでインポート手順を自動化して、複数のテーブルを連続的にインポートする
- テーブルにデータを挿入または追加する
- テーブル内のデータを置換する
- インポートの前またはインポート中にテーブルを作成する
- クライアントコンピュータにあるファイルからデータをロードする
- BCP フォーマット句を使用して、SQL Anywhere と Adaptive Server Enterprise の間でファイルを転送する

まったく新しいデータベースを作成する場合、パフォーマンスを最適化するには、LOAD TABLE を使用してデータをロードしてください。

参照

- 「LOAD TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「UNLOAD 文」『SQL Anywhere サーバ SQL リファレンス』
- 「INPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「データのインポートのパフォーマンスに関するヒント」711 ページ
- 「バルクオペレーションのパフォーマンスの側面」709 ページ
- 「-b データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「インポート用のテーブル構造」728 ページ
- 「クライアントコンピュータ上のデータへのアクセス」747 ページ
- 「データベースの再構築」749 ページ

データのインポートのパフォーマンスに関するヒント

大量のデータをインポートするには時間がかかる場合があります。時間を節約するには次のような方法があります。

- データファイルとデータベースは、物理的に別のディスクドライブに置く。ロード中のディスクヘッドの動きを削減できます。
- データベースのサイズを拡張する。ALTER DBSPACE 文を使うと、領域が必要になったときに少しずつサイズ拡張する代わりに、領域が必要になる前の段階でデータベースを大幅に拡張できます。また、大量のデータをロードする場合のパフォーマンスを改善でき、ファイルシステム内でデータベースの断片化を防ぐことができます。

●データのロードにテンポラリテーブルを使用する。ローカルまたはグローバルのテンポラリテーブルは、データセットを繰り返しロードする必要がある場合、または異なる構造を持つテーブルをマージする必要がある場合に役立ちます。

●LOAD TABLE 文を使用している場合は、-b オプション (バルクオペレーションモード) を指定せずにデータベースサーバを起動する。

●INPUT 文または OUTPUT 文を使用している場合は、データベースサーバと同じコンピュータ上で Interactive SQL またはクライアントアプリケーションを実行する。ネットワークを介してデータをロードすると、通信のために余分な負荷がかかります。新しいデータは、データベースサーバがビジー状態ではないときに徐々にロードすることをおすすめします。

参照

- 「LOAD TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「INPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「-b データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「ALTER DBSPACE 文」『SQL Anywhere サーバ SQL リファレンス』

インポートウィザードを使用したデータのインポート

Interactive SQL の**インポートウィザード**は、データのソース、フォーマット、インポート先テーブルを選択するために使用します。データはテキストファイル、固定フォーマットファイル、シェイプファイルから、既存のテーブルや新しいテーブルにインポートできます。

前提条件

データを既存のテーブルにインポートする場合は、そのテーブルの所有者であるか、そのテーブルに対する SELECT 権限と INSERT 権限を持っているか、SELECT ANY TABLE システム権限と INSERT ANY TABLE システム権限を持っている必要があります。

データを新しいテーブルにインポートする場合、CREATE TABLE、CREATE ANY TABLE、または CREATE ANY OBJECT システム権限を持っている必要があります。

内容と備考

また、**インポートウィザード**を使用すると、次の間でデータのインポートを行うこともできます。

●SQL Anywhere データベースと Ultra Light データベースなど、タイプの異なるデータベース。

●SQL Anywhere 16 データベースと SQL Anywhere 11 データベースなど、バージョンの異なるデータベース (各データベースの ODBC ドライバがある場合)。

次の場合は、Interactive SQL の**インポートウィザード**を使用します。

●データのインポートと同時にテーブルを作成する場合

●テキスト以外のフォーマットでのデータのインポートにポイントアンドクリックインタフェースを使用する場合

♦ タスク

1. Interactive SQL で、**[データ]** » **[インポート]** をクリックします。
2. **インポートウィザード**の指示に従います。

結果

データが、指定されたデータベースにインポートされます。

例

データをファイルから SQL Anywhere サンプルデータベースにインポートするには、次の手順を実行します。

1. 次の値を持つテキストファイルを作成し、*newProducts.csv* という名前で保存します。

```
ID,Name,Description,Size,Color,Quantity,UnitPrice
701,Shorts,Spandex Cycling,Small,Black,500,23.00
702,Shorts,Spandex Cycling,Medium,Black,500,23.00
703,Shorts,Spandex Cycling,Large,Black,200,25.00
```

2. Interactive SQL で、**[データ]** » **[インポート]** をクリックします。
3. **[テキストファイル]** をクリックしてから、**[次へ]** をクリックします。
4. **[参照]** をクリックして、**newProducts.csv** が格納されているフォルダに移動します。

   このファイルは、データベースサーバコンピュータではなくクライアントコンピュータに関連して配置されます。
5. **newProducts.csv** をクリックしてから、**[開く]** をクリックします。
6. **[既存のテーブル]** をクリックします。
7. **[Product]** をクリックしてから、**[次へ]** をクリックします。
8. **[フィールド セパレータ]** リストで、**[カンマ (,)]** をクリックします。
9. **[その他のオプション]** リストで、**[最初の行にカラム名を含める]** をクリックします。
10. **[次へ]** をクリックします。
11. **[インポート]** をクリックします。
12. **[閉じる]** をクリックします。

    インポートが完了すると、ウィザードで作成した SQL 文が履歴リストに保存されます。

    生成した SQL INPUT 文を表示するには、**[SQL]** メニューで **[前の SQL]** をクリックします。

インポートウィザードによって生成された INPUT 文が **[SQL 文]** ウィンドウ枠に次のように表示されます。

```
-- Generated by the Import Wizard
input into "GROUPO"."Products" from 'C:¥¥LocalTemp¥¥newSwimwear.csv'
format text escapes on escape character '¥¥'
delimited by ',' encoding 'Cp1252' skip 1
```

データを SQL Anywhere サンプルデータベースから Ultra Light データベースにインポートするには、次の手順を実行します。

1. Ultra Lite データベース (*C:¥Users¥Public¥Documents¥SQL Anywhere 16¥Samples¥UltraLite ¥CustDB¥custdb.udb* など) に接続します。
2. Interactive SQL で、**[データ]** » **[インポート]** をクリックします。
3. **[データベース]** をクリックします。**[次へ]** をクリックします。
4. **[データベースタイプ]** リストで、**[SQL Anywhere]** をクリックします。
5. **[アクション]** ドロップダウンリストで、**[ODBC データソースを使用した接続]** をクリックします。
6. **[ODBC データソース名]** をクリックし、その下のボックスに **SQL Anywhere 16 Demo** と入力します。
7. **[次へ]** をクリックします。
8. **[テーブル名]** リストで、**[Customers]** をクリックします。**[次へ]** をクリックします。
9. **[新しいテーブル]** をクリックします。
10. **[テーブル名]** フィールドに **SQLAnyCustomers** と入力します。
11. **[インポート]** をクリックします。
12. **[閉じる]** をクリックします。
13. 生成された SQL 文を表示するには、**[SQL]** » **[前の SQL]** をクリックします。

    インポートウィザードによって生成された INPUT 文が **[SQL 文]** ウィンドウ枠に次のように表示されます。

    ```
    -- Generated by the Import Wizard
    input using 'dsn=SQL Anywhere 16 Demo;CON=""'
    from "GROUPO.Customers" into "SQLAnyCustomers"
    create table on
    ```

INPUT 文を使用したデータのインポート

INPUT 文は、異なるファイルフォーマットのデータを既存のテーブルや新しいテーブルにインポートするために使用します。データベースの ODBC ドライバが存在する場合は、USING 句を

使用すると、異なる種類のデータベースや異なるバージョンの SQL Anywhere データベースからデータをインポートできます。

INPUT 文を使用すると、TEXT および FIXED フォーマットのデータをインポートできます。他のファイルフォーマットのデータをインポートするには、USING 句と ODBC データソースを使用します。

デフォルトの入力フォーマットを使用したり、INPUT 文ごとにファイルフォーマットを指定することができます。INPUT 文は Interactive SQL 文なので、IF 文などの複合文やストアドプロシージャでは使用できません。

INPUT 文は、ファイルまたは他のデータベースからデータをインポートする場合に使用します。

マテリアライズドビューに関する考慮事項

即時ビューでは、基本となるテーブルにデータをバルクロードしようとするとエラーが返されます。最初にビューのデータをトランケートしてから、バルクロードオペレーションを実行します。

手動ビューでは、基本となるテーブルにデータをバルクロードできます。ただし、ビュー内のデータは再表示されるまで古いままです。

テーブルへのバルクロードオペレーション (INPUT など) を実行する際は、先に従属するマテリアライズドビューのデータをトランケートすることを検討してください。データをロードしたら、ビューを再表示します。

テキストインデックスに関する考慮事項

即時テキストインデックスでは、基本となるテーブルで INPUT などのバルクロードオペレーションを実行してからテキストインデックスを更新すると、自動更新にもかかわらず時間がかかる場合があります。手動テキストインデックスでは、再表示にも時間がかかる場合があります。

テーブルへのバルクロードオペレーション (INPUT など) を実行する際は、先に従属するテキストインデックスを削除することを検討してください。データをロードしたら、テキストインデックスを再作成します。

データベースに対する影響

INPUT 文を使用すると、変更内容はトランザクションログに記録されます。メディア障害が発生した場合は、詳細な変更の記録があります。ただし、この方法ではすべてのローがトランザクションログに書き込まれるので、この方法で大量のデータをインポートすると、パフォーマンスに影響が及びます。

対照的に、LOAD TABLE 文では各ローがトランザクションログに保存されないため、INPUT 文よりも高速な可能性があります。ただし、INPUT 文ではより多くのデータベースとファイルフォーマットがサポートされています。

参照

- 「INPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「TRUNCATE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DROP TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』

INPUT 文を使用したデータのインポート

Interactive SQL を使用して、データをテキストファイルまたはカンマ区切り (CSV) ファイルからデータベースにインポートできます。

前提条件

テーブルの所有者であるか、次の権限を持っている必要があります。

- テーブルに対する INSERT 権限、または INSERT ANY TABLE システム権限
- テーブルに対する SELECT 権限、または SELECT ANY TABLE システム権限

内容と備考

INPUT 文は Interactive SQL 文なので、IF 文などの複合文やストアドプロシージャでは使用できません。

♦ タスク

1. 次の値を持つ *newSwimwear.csv* という名前のテキストファイルを作成し、*C:¥LocalTemp* ディレクトリに保存します。

```
ID,Name,Description,Size,Color,Quantity,UnitPrice
800,Swimsuit,Lycra,Small,Blue,10,81.00
801,Swimsuit,Lycra,Medium,Blue,10,81.00
802,Swimsuit,Lycra,Large,Blue,7,85.00
```

2. Interactive SQL を開き、SQL Anywhere サンプルデータベースに接続します。

3. **[SQL 文]** ウィンドウ枠に INPUT 文を入力します。

```
INPUT INTO Products
FROM C:¥LocalTemp¥newSwimwear.csv
FORMAT TEXT
SKIP 1;
```

   この文では、インポート先テーブルの名前は Products で、*newSwimwear.csv* はデータファイルの名前です。カラム名が含まれているファイルの最初の行はスキップされます。このファイルはクライアントコンピュータに関連して配置されます。

4. 文を実行します。

   インポートに成功すると、**[メッセージ]** タブに、データのインポートに要した時間が表示されます。インポートに失敗した場合は、インポートに失敗した理由を示すメッセージが表示されます。

結果

指定されたデータベースにデータがインポートされます。

例

INPUT 文を使用して Excel CSV ファイルからデータを入力するには、次の手順を実行します。

1. Excel で、Excel ファイルのデータを CSV ファイルに保存します。たとえば、そのファイルに *newSales.csv* という名前を付けます。
2. Interactive SQL で、SQL Anywhere データベース (サンプルデータベースなど) に接続します。
3. **imported_sales** という名前のテーブルを作成し、必要なカラムを追加します。
4. Excel によって CSV ファイルの最初の行に配置されたカラム名をスキップするための SKIP 句を使用して、INPUT 文を実行します。

```
INPUT INTO imported_sales FROM 'C:¥¥LocalTemp¥¥newSales.csv' SKIP 1;
```

参照

- 「INPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』

LOAD TABLE 文を使用したデータのインポート

LOAD TABLE 文は、データベースサーバまたはクライアントコンピュータにあるデータを、テキストフォーマットや ASCII フォーマットで既存のテーブルにインポートするために使用します。

また、LOAD TABLE 文を使用すると、別のテーブルのカラムや値の式 (関数やシステムプロシージャの結果など) からデータをインポートすることもできます。

LOAD TABLE 文はテーブルにローを追加しますが、置き換えることはしません。

LOAD TABLE 文 (WITH ROW LOGGING および WITH CONTENT LOGGING オプションなし) を使用すると、INPUT 文を使用するよりもずっと早くデータをロードできます。

LOAD TABLE 文を使用してロードされたデータに対してはトリガは起動しません。

マテリアライズドビューに関する考慮事項

即時ビューでは、基本となるテーブルにデータをバルクロードしようとするとエラーが返されます。最初にビューのデータをトランケートしてから、バルクロードオペレーションを実行します。

手動ビューでは、基本となるテーブルにデータをバルクロードできます。ただし、ビュー内のデータは再表示されるまで古いままです。

テーブルへのバルクロードオペレーション (LOAD TABLE など) を実行する際は、先に従属するマテリアライズドビューのデータをトランケートすることを検討してください。データをロードしたら、ビューを再表示します。

テキストインデックスに関する考慮事項

即時テキストインデックスでは、基本となるテーブルで LOAD TABLE などのバルクロードオペレーションを実行してからテキストインデックスを更新すると、自動更新にもかかわらず時間がかかる場合があります。手動テキストインデックスでは、再表示にも時間がかかる場合があります。

テーブルへのバルクロードオペレーション (LOAD TABLE など) を実行する際は、先に従属するテキストインデックスを削除することを検討してください。データをロードしたら、テキストインデックスを再作成します。

データベースリカバリと同期の考慮事項

デフォルトでは、データをファイルからロードする場合 (LOAD TABLE table-name FROM filename; など)、LOAD TABLE 文のみがトランザクションログに記録され、ロードされる実際のデータのローは記録されません。このため、元のロードファイルの変更、移動、削除後にトランザクションログを使用してデータベースのリカバリを行おうとすると、問題が発生します。また、同期やレプリケーションを行うデータベースが新しいデータを取得できません。

リカバリや同期の考慮事項に対応するため、LOAD TABLE 文では 2 つのロギングオプションを使用できます。WITH ROW LOGGING を使用すると、ローをロードするごとに、トランザクションログに INSERT 文を作成します。WITH CONTENT LOGGING を使用すると、ロードしたローをチャンクにグループ化して、そのチャンクをトランザクションログに記録します。これらのオプションにより、ロードデータのソースがなくなっても、ロード操作を繰り返すことができます。

データベースのミラーリングの考慮事項

データベースでミラーリングを行っている場合は、LOAD TABLE 文の使用に注意してください。たとえば、ファイルからデータをロードする場合、ミラーサーバでファイルをロードできるかどうかや、ミラーデータベースがロードを処理するまでにロード元のソースのデータが変更されるかどうかに注意してください。これらのいずれかのリスクが存在する場合は、LOAD TABLE 文のロギングレベルとして WITH ROW LOGGING または WITH CONTENT LOGGING のいずれかを指定してください。これにより、ミラーデータベースにロードされるデータが、ミラーリングされているデータベースにロードされたデータと同一になります。

参照

- 「CREATE TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DROP TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「クライアントコンピュータ上のデータへのアクセス」747 ページ
- 「データベースミラーリング」『SQL Anywhere サーバ データベース管理』
- 「INPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「LOAD TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「TRUNCATE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』

INSERT 文を使用したデータのインポート

INSERT 文は、データベースにローを追加するために使用します。INSERT 文にはインポート先テーブルにインポートするデータが含まれているため、この文は対話型入力と見なされます。リモートデータアクセスに INSERT 文を使用して、ファイルではなく別のデータベースからデータをインポートすることもできます。

次の場合は、INSERT 文を使用してデータをインポートします。

- 1 つのテーブルに少量のデータをインポートする場合
- ファイルフォーマットが柔軟な場合
- ファイルではなく外部データベースからリモートデータをインポートする場合

INSERT 文では、ON EXISTING 句を使用して、挿入するローがすでに挿入先のテーブルに存在する場合に実行するアクションを指定できます。ただし、ON EXISTING 条件を満たすローが数多く存在する可能性がある場合は、代わりに MERGE 文を使用してください。MERGE 文を使用すると、一致するローに対して実行するアクションをより詳細に制御できます。また、一致を定義するためのより高度な構文も使用できます。

マテリアライズドビューに関する考慮事項

即時ビューでは、基本となるテーブルにデータをバルクロードしようとするとエラーが返されます。最初にビューのデータをトランケートしてから、バルクロードオペレーションを実行します。

手動ビューでは、基本となるテーブルにデータをバルクロードできます。ただし、ビュー内のデータは再表示されるまで古いままです。

テーブルへのバルクロードオペレーション (INSERT など) を実行する際は、先に従属するマテリアライズドビューのデータをトランケートすることを検討してください。データをロードしたら、ビューを再表示します。

テキストインデックスに関する考慮事項

即時テキストインデックスでは、基本となるテーブルで INSERT などのバルクロードオペレーションを実行してからテキストインデックスを更新すると、自動更新にもかかわらず時間がかかる場合があります。手動テキストインデックスでは、再表示にも時間がかかる場合があります。

テーブルへのバルクロードオペレーション (INSERT など) を実行する際は、先に従属するテキストインデックスを削除することを検討してください。データをロードしたら、テキストインデックスを再作成します。

データベースに対する影響

INSERT 文を使用すると、変更内容はトランザクションログに記録されます。データベースファイルに関するメディア障害が発生した場合は、変更内容に関する情報をトランザクションログからリカバリできます。

参照

- 「トランザクションログ」『SQL Anywhere サーバ データベース管理』
- 「MERGE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「INSERT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「TRUNCATE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DROP TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「LOAD TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「INPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』

MERGE 文を使用したデータのインポート

MERGE 文は、更新操作を実行し、大量のテーブルデータを更新するために使用します。データのマージ時に、ソースデータのローがターゲットデータのローに一致する場合または一致しない場合に実行するアクションを指定できます。

マージ動作の定義

MERGE 文の構文の省略形を以下に示します。

MERGE INTO *target-object*
USING *source-object*
ON *merge-search-condition*
{ WHEN MATCHED | WHEN NOT MATCHED } [...]

データベースによってマージ操作が実行されると、*source-object* のローと *target-object* のローが比較され、ON 句に含まれる定義に基づいて一致する行または一致しない行が検索されます。*merge-search-condition* が true になるローが少なくとも 1 つ *target-table* に存在する場合、*source-object* のローは一致と見なされます。

source-object は、ベーステーブル、ビュー、マテリアライズドビュー、派生テーブル、プロシージャの結果のいずれかです。*target-object* には、これらのオブジェクトのうち、マテリアライズドビューとプロシージャ以外の任意のオブジェクトを指定できます。

ANSI SQL/2008 標準では、マージ操作中に *target-object* のローを *source-object* の複数のローで更新することは許可されません。

source-object のローが一致または不一致と見なされると、それぞれ一致の場合と不一致の場合の WHEN 句 (WHEN MATCHED または WHEN NOT MATCHED) に対して評価されます。WHEN MATCHED 句では、*target-object* のローに対して実行するアクションを定義します (たとえば、WHEN MATCHED ... UPDATE は、*target-object* のローを更新することを指定します)。WHEN NOT MATCHED 句では、*source-object* の一致しないローを使用して、*target-object* に対して実行するアクションを定義します。

WHEN 句は無制限に指定でき、指定した順序で処理されます。また、WHEN 句内で AND 句を使用し、ローのサブセットに対するアクションを指定することもできます。たとえば、次の WHEN 句では、一致したローの Quantity カラムの値に応じて、異なるアクションを実行するよう定義しています。

```
WHEN MATCHED AND myTargetTable.Quantity<=500 THEN SKIP
WHEN MATCHED AND myTargetTable.Quantity>500 THEN UPDATE SET
myTargetTable.Quantity=500
```

マージ操作における分岐

一致するローおよび一致しないローをアクションごとにグループ化することを「分岐化」と呼び、各グループを「分岐」と呼びます。「分岐」は、単一の WHEN MATCHED 句または WHEN NOT MATCHED 句と同等です。たとえば、ある分岐に *source-object* の一致しないローのセットが含まれている場合は、それらのローを挿入する必要があります。分岐アクションの実行は、すべての分岐アクティビティを完了してから (*source-object* のすべてのローを評価してから) 開始されます。データベースサーバは、WHEN 句が指定された順序に従って、分岐アクションの実行を開始します。

source-object の一致しないロー、または *source-object* と *target-object* の一致するローのペアが分岐に入ると、後続の分岐に対して評価されません。したがって、WHEN 句を指定する順序は重要です。

一致または不一致と見なされる *source-object* のローのうち、いずれの分岐にも属さない (つまり、いずれの WHEN 句も満たさない) ものは無視されます。これは、WHEN 句に AND 句が含まれていて、ローがいずれの AND 句の条件も満たさない場合に発生します。この場合、アクションが定義されていないため、ローは無視されます。

データを変更するアクションは、個々の INSERT、UPDATE、DELETE 文としてトランザクションログに記録されます。

ターゲットテーブルに定義されたトリガ

通常、マージ操作中に各 INSERT、UPDATE、DELETE 文を実行すると、トリガが起動されます。たとえば、UPDATE アクションが定義された分岐の処理時に、データベースサーバは次の内容を実行します。

1. すべての BEFORE UPDATE トリガを起動する
2. ローの候補セットに対して UPDATE 文を実行すると同時に、すべてのローレベルの UPDATE トリガを起動する
3. AFTER UPDATE トリガを起動する

target-table に対してトリガを起動すると、別の分岐で更新されるローに影響が及ぶ場合、マージ操作で競合が発生する可能性があります。たとえば、ロー B を削除するトリガを起動するアクションをロー A に対して実行するとします。しかし、ロー B には独自のアクションが定義されており、まだ実行されていません。ローに対してアクションを実行できないと、マージ操作は失敗し、すべての変更がロールバックされ、エラーが返されます。

複数のトリガアクションが定義されたトリガは、同じトリガに各トリガアクションを 1 つずつ指定したものと見なされます (つまり、各トリガに 1 つのトリガアクションを指定して、別個のトリガを定義したのと同等になります)。

即時マテリアライズドビューに関する考慮事項

MERGE 文によって多数のローを更新すると、データベースサーバのパフォーマンスに影響する場合があります。多数のローを更新する場合は、従属する即時マテリアライズドビューのデータ

をトランケートしてから、テーブルで MERGE 文を実行することを検討してください。MERGE 文を実行したら、REFRESH MATERIALIZED VIEW 文を実行します。

テキストインデックスに関する考慮事項

MERGE 文によって多数のローを更新すると、データベースサーバのパフォーマンスに影響する場合があります。テーブルで MERGE 文を実行する際は、先に従属するテキストインデックスを削除することを検討してください。MERGE 文を実行したら、テキストインデックスを再作成します。

参照

- 「MERGE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「TRUNCATE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DROP TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』

例 1

ジャケットとセーターを販売する小さい会社を経営しているとします。ジャケットの素材の価格が 5% 上昇したため、それに合わせて価格を調整したいとします。次の CREATE TABLE 文を使用して、販売するジャケットとセーターの現在の価格情報を保持する myProducts という小さいテーブルを作成します。その後の INSERT 文で、myProducts にデータを入力します。この例では、CREATE TABLE 権限が必要です。

```
CREATE TABLE myProducts (
  product_id    NUMERIC(10),
  product_name  CHAR(20),
  product_size  CHAR(20),
  product_price NUMERIC(14,2));
INSERT INTO myProducts VALUES (1, 'Jacket', 'Small', 29.99);
INSERT INTO myProducts VALUES (2, 'Jacket', 'Medium', 29.99);
INSERT INTO myProducts VALUES (3, 'Jacket', 'Large', 39.99);
INSERT INTO myProducts VALUES (4, 'Sweater', 'Small', 18.99);
INSERT INTO myProducts VALUES (5, 'Sweater', 'Medium', 18.99);
INSERT INTO myProducts VALUES (6, 'Sweater', 'Large', 19.99);
SELECT * FROM myProducts;
```

product_id	product_name	product_size	product_price
1	Jacket	Small	29.99
2	Jacket	Medium	29.99
3	Jacket	Large	39.99
4	Sweater	Small	18.99
5	Sweater	Medium	18.99
6	Sweater	Large	19.99

さらに、次の文を使用して、ジャケットの価格変更に関する情報を保持する myPrices という別のテーブルを作成します。マージ操作を実行する前に myPrices テーブルの内容を確認できるように、末尾に SELECT 文を追加します。

```
CREATE TABLE myPrices (
  product_id    NUMERIC(10),
  product_name  CHAR(20),
  product_size  CHAR(20),
  product_price NUMERIC(14,2),
  new_price     NUMERIC(14,2));
INSERT INTO myPrices (product_id) VALUES (1);
INSERT INTO myPrices (product_id) VALUES (2);
INSERT INTO myPrices (product_id) VALUES (3);
INSERT INTO myPrices (product_id) VALUES (4);
INSERT INTO myPrices (product_id) VALUES (5);
INSERT INTO myPrices (product_id) VALUES (6);
SELECT * FROM myPrices;
```

product_id	product_name	product_size	product_price	new_price
1	(NULL)	(NULL)	(NULL)	(NULL)
2	(NULL)	(NULL)	(NULL)	(NULL)
3	(NULL)	(NULL)	(NULL)	(NULL)
4	(NULL)	(NULL)	(NULL)	(NULL)
5	(NULL)	(NULL)	(NULL)	(NULL)
6	(NULL)	(NULL)	(NULL)	(NULL)

次の MERGE 文を使用して、myProducts テーブルのデータを myPrices テーブルにマージします。*source-object* は、product_name が Jacket のローのみを含むようフィルタリングされた派生テーブルです。また、ON 句では、*target-object* と *source-object* の product_id カラムの値が一致する場合にローが一致するよう指定しています。

```
MERGE INTO myPrices p
USING ( SELECT
    product_id,
    product_name,
    product_size,
    product_price
  FROM myProducts
  WHERE product_name='Jacket') pp
ON (p.product_id = pp.product_id)
WHEN MATCHED THEN
  UPDATE SET
    p.product_id=pp.product_id,
    p.product_name=pp.product_name,
    p.product_size=pp.product_size,
    p.product_price=pp.product_price,
    p.new_price=pp.product_price * 1.05;
SELECT * FROM myPrices;
```

product_id	product_name	product_size	product_price	new_price
1	Jacket	Small	29.99	31.49
2	Jacket	Medium	29.99	31.49
3	Jacket	Large	39.99	41.99
4	(NULL)	(NULL)	(NULL)	(NULL)
5	(NULL)	(NULL)	(NULL)	(NULL)
6	(NULL)	(NULL)	(NULL)	(NULL)

product_id が 4、5、6 のカラムの値は、NULL のままになります。これは、myProducts テーブルで製品が (product_name='Jacket') であるいずれのローとも一致しなかったためです。

例 2

次の例では、myTargetTable のプライマリキー値を使用してローを一致させ、mySourceTable テーブルと myTargetTable テーブルのローをマージします。mySourceTable のローに、myTargetTable のプライマリキーカラムと同じ値が含まれている場合、ローは一致していると見なされます。

```
MERGE INTO myTargetTable
  USING mySourceTable ON PRIMARY KEY
  WHEN NOT MATCHED THEN INSERT
  WHEN MATCHED THEN UPDATE;
```

WHEN NOT MATCHED THEN INSERT 句は、mySourceTable にはあって myTargetTable にはないローを、myTargetTable に追加することを指定します。WHEN MATCHED THEN UPDATE 句は、myTargetTable の一致するローを mySourceTable の値に更新することを指定します。

次の構文は前述の構文と同等です。ここでは、myTargetTable にはカラム (I1, I2, .. I*n*) があり、プライマリキーがカラム (I1, I2) に定義されていることを前提としています。mySourceTable にはカラム (U1, U2, .. U*n*) があります。

```
MERGE INTO myTargetTable ( I1, I2, .. ., In )
  USING mySourceTable ON myTargetTable.I1 = mySourceTable.U1
    AND myTargetTable.I2 = mySourceTable.U2
  WHEN NOT MATCHED
    THEN INSERT ( I1, I2, .. In )
      VALUES ( mySourceTable.U1, mySourceTable.U2, ..., mySourceTable.Un )
  WHEN MATCHED
    THEN UPDATE SET
    myTargetTable.I1 = mySourceTable.U1,
    myTargetTable.I2 = mySourceTable.U2,
    ...
    myTargetTable.In = mySourceTable.Un;
```

RAISERROR アクションの使用

一致または不一致のアクションに指定できるアクションの 1 つが、RAISERROR です。RAISERROR を使用すると、WHEN 句の条件を満たした場合に、マージ操作を失敗させることができます。

RAISERROR を指定すると、データベースサーバはデフォルトで SQLSTATE 23510 および SQLCODE -1254 を返します。必要に応じて、RAISERROR キーワードの後に *error_number* パラメータを指定し、返される SQLCODE をカスタマイズできます。

カスタム SQLCODE の指定は、後でエラーの発生した状況を特定する場合に便利です。

カスタム SQLCODE には 17000 よりも大きい正の整数を指定してください。数または変数のいずれとしても指定できます。

次の文で、SQLCODE のカスタマイズによって返される内容にどのような影響が出るのかを簡単に示します。この例では、CREATE TABLE 権限が必要です。

次のように targetTable を作成します。

```
CREATE TABLE targetTable( c1 int );
INSERT INTO targetTable VALUES( 1 );
```

次の文は、SQLSTATE = '23510' および SQLCODE = -1254 のエラーを返します。

```
MERGE INTO targetTable
  USING (SELECT 1 c1 ) AS sourceData
  ON targetTable.c1 = sourceData.c1
  WHEN MATCHED THEN RAISERROR;
SELECT sqlstate, sqlcode;
```

次の文は、SQLSTATE = '23510' および SQLCODE = -17001 のエラーを返します。

```
MERGE INTO targetTable
  USING (SELECT 1 c1 ) AS sourceData
  ON targetTable.c1 = sourceData.c1
  WHEN MATCHED THEN RAISERROR 17001
  WHEN NOT MATCHED THEN RAISERROR 17002;
SELECT sqlstate, sqlcode;
```

次の文は、SQLSTATE = '23510' および SQLCODE = -17002 のエラーを返します。

```
MERGE INTO targetTable
  USING (SELECT 2 c1 ) AS sourceData
  ON targetTable.c1 = sourceData.c1
  WHEN MATCHED THEN RAISERROR 17001
  WHEN NOT MATCHED THEN RAISERROR 17002;
SELECT sqlstate, sqlcode;
```

プロキシテーブルを使用したデータのインポートに関するヒント

プロキシテーブルは、メタデータを含むローカルテーブルです。リモートデータベースサーバのテーブルに、ローカルテーブルであるかのようにアクセスするときに使用します。これによって、データを直接インポートできます。

次の場合は、プロキシテーブルを使用してデータをインポートします。

- リモートデータにアクセスできる場合

●データを別のデータベースから直接インポートする場合

データベースに対する影響

プロキシテーブルを使用すると、変更内容はトランザクションログに記録されます。データベースファイルに関するメディア障害が発生した場合は、変更内容に関する情報をトランザクションログからリカバリできます。

プロキシテーブルの使用方法

プロキシテーブルを作成し、SELECT 句を指定した INSERT 文を使用してリモートデータベースからデータベース内の永久テーブルにデータを挿入します。

参照

- 「リモートデータアクセス」 771 ページ
- 「INSERT 文」『SQL Anywhere サーバ SQL リファレンス』

インポート中の変換エラー

外部ソースからロードされたデータは、エラーを含む場合があります。たとえば、無効な日付や数字が含まれている可能性があります。conversion_error データベースオプションを使用すると、変換エラーを無視し、無効な値を NULL 値に変換できます。

参照

- 「conversion_error オプション」『SQL Anywhere サーバ データベース管理』
- 「SET OPTION 文」『SQL Anywhere サーバ SQL リファレンス』

テーブルのインポート (Interactive SQL インポートウィザードの場合)

Interactive SQL を使用して、データをテキストファイル、任意のデータベースの別のテーブル、またはシェイプファイルからデータベース内のテーブルにインポートできます。

前提条件

ユーザ本人が所有するテーブルを作成するには、CREATE TABLE 権限が必要です。他のユーザが所有するテーブルを作成するには、CREATE ANY TABLE または CREATE ANY OBJECT のシステム権限が必要です。

♦ タスク

1. Interactive SQL で、**[データ]** » **[インポート]** をクリックします。
2. **[テキストファイル]** をクリックしてから、**[次へ]** をクリックします。
3. **[ファイル名]** フィールドで、**[参照]** をクリックしてファイルを追加します。

4. **[新しいテーブル]** をクリックして、**[テーブル名]** に入力します。
5. **[次へ]** をクリックします。
6. テキストファイルの場合は、そのファイルを読み込む方法を指定して **[次へ]** をクリックします。
7. カラム名やデータ型の変更を行って、**[インポート]** をクリックします。
8. **[閉じる]** をクリックします。

結果

データが指定されたテーブルにインポートされます。

テーブルのインポート (SQL の場合)

SQL を使用して、データをテキストファイル、任意のデータベースの別のテーブル、またはシェイプファイルからデータベース内のテーブルにインポートできます。

前提条件

ユーザ本人が所有するテーブルを作成するには、CREATE TABLE 権限が必要です。他のユーザが所有するテーブルを作成するには、CREATE ANY TABLE または CREATE ANY OBJECT のシステム権限が必要です。

データのインポート (ロード) に必要な権限は、-gl データベースオプションの設定値とデータのインポート元によって異なります。データのロードに必要な権限についての詳細は、LOAD TABLE 文の項を参照してください。

♦ タスク

1. CREATE TABLE 文を使用してインポート先テーブルを作成します。次に例を示します。

```
CREATE TABLE Departments (
DepartmentID          integer NOT NULL,
DepartmentName        char(40) NOT NULL,
DepartmentHeadID      integer NULL,
CONSTRAINT DepartmentsKey PRIMARY KEY (DepartmentID) );
```

2. LOAD TABLE 文を実行します。次に例を示します。

```
LOAD TABLE Departments
FROM 'C:¥¥ServerTemp¥¥Departments.csv';
```

3. 値の中の後続ブランクをそのままにするには、LOAD TABLE 文中で STRIP OFF 句を使用します。デフォルトの設定 (STRIP RTRIM) では、値は後続ブランクが取り除かれてから挿入されます。

   LOAD TABLE 文はファイルの内容を、テーブルの既存のローに追加します。既存のローを置き換えるわけではありません。テーブルからすべてのローを削除するには、TRUNCATE TABLE 文を使います。

FROM 句は、データベースサーバコンピュータ上のファイルを指定します。

TRUNCATE TABLE 文と LOAD TABLE 文は、カスケード型削除のようなトリガの起動や参照整合性に関わるアクションの実行を行うことはありません。

結果

データが指定されたテーブルにインポートされます。

参照

- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「LOAD TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「TRUNCATE 文」『SQL Anywhere サーバ SQL リファレンス』

インポート用のテーブル構造

ソースデータの構造は、インポート先テーブル自体の構造と一致する必要はありません。たとえば、カラムのデータ型が異なる、順序が異なる、またはロード先のテーブルのカラム数を超えるインポートデータの値があるなどです。

テーブルまたはデータの並べ替え

インポートするデータの構造がインポート先テーブルの構造と一致しないことがわかっている場合は、次の操作を行うことができます。

- LOAD TABLE 文でロードするカラムの名前リストを入力します。
- INSERT 文の一種とグローバルテンポラリテーブルを使用して、インポートデータをテーブルに合うように並べ替えることができます。
- INPUT 文を使用して、カラムの特定のセットまたは順序を指定できます。

カラムに NULL 値を入力できるようにする

インポート中のファイルにテーブルのカラムのサブセットへのデータがある場合、またはカラムの順序が異なる場合は、LOAD TABLE 文の DEFAULTS オプションを使用して、ブランクを埋めて一致しないテーブル構造をマージすることもできます。

- DEFAULTS オプションが OFF の場合は、カラムリストにないカラムすべてに NULL が割り当てられます。DEFAULTS オプションが OFF で、NULL 入力不可のカラムがカラムリストから省かれている場合は、データベースサーバは、空の文字列をカラムの型に変換しようとします。
- DEFAULTS オプションが ON で、カラムにデフォルト値が入っている場合は、その値が使用されます。

たとえば、Customers テーブルの City カラムにデフォルト値を定義してから、次のような LOAD TABLE 文を使用して、データベースサーバコンピュータ上の *C:¥ServerTemp* ディレクトリにある *newCustomers.csv* というファイルから、Customers テーブルに新しいローをロードできます。

```
ALTER TABLE Customers
ALTER City DEFAULT 'Waterloo';

LOAD TABLE Customers ( Surname, GivenName, Street, State, Phone )
FROM 'C:¥¥ServerTemp¥¥newCustomers.csv'
DEFAULTS ON;
```

City カラムには値が入力されていないため、デフォルト値が入力されます。DEFAULTS OFF が指定されている場合は、City カラムには空の文字列が割り当てられます。

異なるテーブル構造のマージ

INSERT 文の一種とグローバルテンポラリテーブルを使用して、インポートデータをテーブルに合うように並べ替えます。

前提条件

グローバルテンポラリテーブルを作成するには、次のいずれかのシステム権限を持っている必要があります。

- CREATE TABLE
- CREATE ANY TABLE
- CREATE ANY OBJECT

データのインポート (ロード) に必要な権限は、-gl データベースオプションの設定値とデータのインポート元によって異なります。データのロードに必要な権限についての詳細は、LOAD TABLE 文の項を参照してください。

INSERT 文を使用するには、テーブルの所有者であるか、次のいずれかの権限を持っている必要があります。

- INSERT ANY TABLE システム権限
- そのテーブルに対する INSERT 権限

さらに、ON EXISTING UPDATE 句が指定されている場合は、UPDATE ANY TABLE システム権限を持っているか、テーブルに対する UPDATE 権限を持っている必要があります。

♦ タスク

1. **[SQL 文]** ウィンドウ枠で、入力ファイルと構造が一致するグローバルテンポラリテーブルを作成します。

   CREATE TABLE 文を使用して、グローバルテンポラリテーブルを作成できます。

2. LOAD TABLE 文を使用して、作成したグローバルテンポラリテーブルにデータをロードします。

   データベース接続を閉じると、グローバルテンポラリテーブル内のデータは消去されます。ただし、テーブル定義は残ります。この定義は、次にデータベースに接続するときに使用できます。

3. INSERT 文と SELECT 句を使用し、テンポラリテーブルからデータを抽出して要約し、データベースの 1 つまたは複数の永久テーブルにコピーします。

結果

データが、永久データベーステーブルにロードされます。

例

次は前述した手順の例です。

```
CREATE GLOBAL TEMPORARY TABLE TempProducts
(
   ID              integer NOT NULL,
   Name              char(15) NOT NULL,
   Description        char(30) NOT NULL,
   Size             char(18) NOT NULL,
   Color            char(18) NOT NULL,
   Quantity          integer NOT NULL,
   UnitPrice         numeric(15,2) NOT NULL,
   CONSTRAINT ProductsKey PRIMARY KEY (ID)
)
ON COMMIT PRESERVE ROWS;

LOAD TABLE TempProducts
FROM 'C:¥¥ServerTemp¥¥newProducts.csv'
SKIP 1;

INSERT INTO Products WITH AUTO NAME
   (SELECT Name, Description, ID, Size, Color, Quantity,
         UnitPrice * 1.25 AS UnitPrice
   FROM TempProducts);
```

グローバルテンポラリテーブル内の品目の価格は、ローを Products テーブルに挿入する前に 25% 増しに調整されます。

参照

- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「LOAD TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「INSERT 文」『SQL Anywhere サーバ SQL リファレンス』

データエクスポート

データのエクスポートは、データベースからのデータの書き出しに関する管理タスクです。データのエクスポートは、データベースの大部分を共有する必要がある場合、または特定の基準に従ってデータベースの一部を抽出する必要がある場合に便利です。SQL Anywhere を使用して、次の作業を実行できます。

- 個々のテーブル、クエリ結果、またはテーブルスキーマをエクスポートする
- 複数のテーブルを連続的にエクスポートできるようにするために、エクスポートを自動化するスクリプトを作成する

- 多くの異なるファイルフォーマットにエクスポートする
- クライアントコンピュータにあるファイルにデータをエクスポートする場合
- BCP FORMAT 句を使用して、SQL Anywhere と Adaptive Server Enterprise の間でファイルをエクスポートする

データをエクスポートする前に、所有しているリソースの種類、およびデータベースからエクスポートする情報の種類を判断します。

データベース全体をエクスポートする場合は、パフォーマンスを考慮して、データをエクスポートするのではなくデータベースをアンロードしてください。

エクスポートの制限事項

Microsoft Excel ODBC ドライバを使用して、SQL Anywhere データベースから Excel データベースにデータをエクスポートすると、以下のデータ型の変更が発生することがあります。

- CHAR、LONG VARCHAR、NCHAR、NVARCHAR、または LONG NVARCHAR データ型として格納されているデータをエクスポートする場合、データは VARCHAR (Excel ドライバによってサポートされる最も近い型) として格納されます。

  Microsoft Excel ODBC ドライバがサポートするテキストカラムの幅は、255 文字までです。

- MONEY データ型と SMALLMONEY データ型として格納されているデータは、CURRENCY データ型にエクスポートされます。それ以外の場合、数値データは数値としてエクスポートされます。

参照

- 「UNLOAD 文」『SQL Anywhere サーバ SQL リファレンス』
- 「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「バルクオペレーションのパフォーマンスの側面」709 ページ
- 「Interactive SQL での NULL 値の処理設定」742 ページ
- 「クライアントコンピュータ上のデータへのアクセス」747 ページ
- 「データベースの再構築」749 ページ

エクスポートウィザードを使用したデータのエクスポート

Interactive SQL の**エクスポートウィザード**は、特定のフォーマットのクエリ結果をファイルやデータベースにエクスポートするために使用します。

前提条件

問い合わせるテーブルの所有者であるか、そのテーブルに対する SELECT 権限を持っているか、または SELECT ANY TABLE システム権限を持っていることが必要です。

♦ タスク

1. クエリを実行します。

2. Interactive SQL で、**[データ]** » **[エクスポート]** をクリックします。
3. **エクスポートウィザード**の指示に従います。

結果

クエリ結果は、指定されたファイルまたはデータベースにエクスポートされます。

例

1. サンプルデータベースに接続している状態で、次のクエリを実行します。Employees テーブルに対する SELECT 権限を持っているか、SELECT ANY TABLE システム権限を持っている必要があります。

```
SELECT * FROM Employees WHERE State = 'GA';
```

2. 結果セットには、ジョージア州に住むすべての従業員のリストが含まれます。
3. **[データ]** » **[エクスポート]** をクリックします。
4. **[データベース]** をクリックしてから、**[次へ]** をクリックします。
5. **[データベースタイプ]** リストで、**[Ultra Light]** をクリックします。
6. **[ユーザ ID]** フィールドに、**DBA** と入力します。
7. **[パスワード]** フィールドに **sql** と入力します。
8. **[データベースファイル]** フィールドに、*C:¥Users¥Public¥Documents¥SQL Anywhere 16¥Samples¥UltraLite¥CustDB¥custdb.udb* と入力します。
9. **[次へ]** をクリックします。
10. **[新しいテーブルを作成]** をクリックします。
11. **[テーブル名]** フィールドに **GAEmployees** と入力します。
12. **[エクスポート]** をクリックします。
13. **[閉じる]** をクリックします。
14. **[SQL]** » **[前の SQL]** をクリックします。

    エクスポートウィザードによって作成および使用された OUTPUT USING 文が **[SQL 文]** ウィンドウ枠に表示されます。

```
-- Generated by the Export Wizard
output using 'driver=UltraLite 16;UID=DBA;PWD=***;
DBF=C:¥¥Users¥¥Public¥¥Documents¥¥SQL Anywhere 16¥¥Samples¥¥Ultralite¥¥custdb¥
¥custdb.udb'
into "GAEmployees"
create table on
```

OUTPUT 文を使用したデータのエクスポートに関するヒント

OUTPUT 文を使用して、データベースからクエリ結果、テーブル、またはビューをエクスポートします。

OUTPUT 文は、SELECT 文の結果セットを複数の異なるファイルフォーマットで書き出すことができるため、互換性が重要な場合に役立ちます。デフォルトの出力フォーマットを使用したり、OUTPUT 文ごとにファイルフォーマットを指定することができます。Interactive SQL では、複数の OUTPUT 文が含まれた SQL スクリプトファイルを実行できます。

Interactive SQL のデフォルトの出力フォーマットは、Interactive SQL の **[オプション]** ウィンドウ (Interactive SQL で **[ツール]** » **[オプション]** をクリック) の **[インポート／エクスポート]** タブで指定します。

次の場合は、Interactive SQL の OUTPUT 文を使用します。

- テキスト以外のフォーマットでテーブルまたはビューのすべてまたは一部をエクスポートする場合
- SQL スクリプトファイルを使用してエクスポート処理を自動化する場合

データベースに対する影響

OUTPUT 文、UNLOAD 文、UNLOAD TABLE 文のいずれかを選択できる場合は、パフォーマンスを考慮して UNLOAD TABLE 文を選択します。

OUTPUT 文を使用して大量のデータをエクスポートすると、パフォーマンスに影響が及びます。可能であれば OUTPUT 文はサーバと同じコンピュータ上で使用して、ネットワークを介してデータを大量に送信しないようにしてください。

参照

- 「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』

Excel ファイルへのデータのエクスポート

Interactive SQL で OUTPUT 文を使用して、データをデータベースから Excel ファイルにエクスポートできます。

前提条件

そのテーブルに対する SELECT 権限、または SELECT ANY TABLE システム権限が必要です。

Excel ODBC ドライバが 32 ビットドライバである場合は、32 ビットバージョンの Interactive SQL を使用して、アーキテクチャの不一致エラーを回避する必要があります。

♦ タスク

1. Interactive SQL で、SQL Anywhere データベースに接続します。

2. READONLY 句を使用して OUTPUT 文を実行します。次に例を示します。

```
SELECT * FROM SalesOrders;
OUTPUT USING 'Driver=Microsoft Excel Driver (*.xls);DBQ=sales.xls;READONLY=0'
INTO "newSalesData";
```

sales.xls という名前の新しい Excel ファイルが作成されます。このファイルには、newSalesData というワークシートが含まれています。

結果

指定されたファイルにデータがエクスポートされます。

例

次の例では、サンプルデータベース内の Employees テーブルのデータを、*demo2* という名前のデータベース内の新しいテーブルにエクスポートします。2 つ目のデータベースへの接続に使用するユーザ ID には、CREATE TABLE、CREATE ANY TABLE、または CREATE ANY OBJECT システム権限が必要です。

```
SELECT * FROM Employees;
OUTPUT USING 'DRIVER=SQL Anywhere 16;UID=DBA;PWD=sql;DBN=demo2;CON=""'
   INTO "newEmployees"
   CREATE TABLE ON;
```

参照

- 「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』

CSV ファイルへのデータのエクスポート

Interactive SQL で OUTPUT 文を使用して、データをデータベースから CSV ファイルにエクスポートできます。

前提条件

SELECT ANY TABLE システム権限、またはテーブルに対する SELECT 権限が必要です。

♦ タスク

1. Interactive SQL で、SQL Anywhere データベースに接続します。

2. FORMAT TEXT、QUOTE ""、WITH COLUMN NAMES 句を使用して OUTPUT 文を実行し、ファイルの最初の行にカラム名を持つカンマ区切りフォーマットファイルを作成します。文字列値は引用符で囲まれます。次に例を示します。

```
SELECT * FROM SalesOrders;
OUTPUT TO 'C:¥¥LocalTemp¥¥newSales.csv'
   FORMAT TEXT
   QUOTE ""
   WITH COLUMN NAMES;
```

結果

指定された CSV ファイルにデータがエクスポートされます。

例

次の例では、SQL Anywhere サンプルデータベース内の Employees テーブルのデータを、*C:¥LocalTemp* ディレクトリにある *Employees.csv* にエクスポートします。

```
SELECT * FROM Employees;
OUTPUT TO C:¥LocalTemp¥Employees.csv
   FORMAT TEXT;
```

参照

- 「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』

UNLOAD TABLE 文を使用したデータのエクスポートに関するヒント

UNLOAD TABLE 文を使用すると、テキストフォーマットだけでデータを効率的にエクスポートできます。UNLOAD TABLE 文では、1 行に 1 ローずつ、値をカンマで区切ってエクスポートします。再ロードを速くするために、データはプライマリキー値の順にエクスポートされます。

次の場合は、UNLOAD TABLE 文を使用します。

- テキストフォーマットでテーブル全体をエクスポートする場合
- データベースパフォーマンスを考慮する場合
- クライアントコンピュータにあるファイルにデータをエクスポートする場合

UNLOAD TABLE 文を使用するには、適切な権限を持っている必要があります。たとえば、-gl データベースサーバオプションの設定が NONE でない限り、ほとんどの場合、SELECT ANY TABLE システム権限があれば十分です。

UNLOAD TABLE 文を使用できるユーザの詳細については、「-gl データベースサーバオプション」『SQL Anywhere サーバ データベース管理』を参照してください。

データベースに対する影響

OUTPUT 文、UNLOAD 文、UNLOAD TABLE 文のいずれかを選択できる場合は、パフォーマンスを考慮して UNLOAD TABLE 文を選択します。

UNLOAD TABLE 文は、アンロード中にテーブル全体に排他ロックを配置します。

例

SQL Anywhere のサンプルデータベースを使用し、次の文を実行して Employees テーブルをテキストファイル *Employees.csv* にアンロードできます。

```
UNLOAD TABLE Employees TO 'C:¥¥ServerTemp¥¥Employees.csv';
```

この形式の UNLOAD TABLE 文を使用すると、ファイルパスはデータベースサーバコンピュータに対する相対パスとなります。

参照

- 「クライアントコンピュータ上のデータへのアクセス」747 ページ
- 「UNLOAD 文」『SQL Anywhere サーバ SQL リファレンス』
- 「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』

UNLOAD 文を使用したデータのエクスポートに関するヒント

UNLOAD 文は、クエリ結果をファイルにエクスポートするという点で OUTPUT 文に似ています。ただし、UNLOAD 文はテキストフォーマットでより効率的にデータをエクスポートします。UNLOAD 文は、1 行に 1 ローずつ、値をカンマで区切ってエクスポートします。

次の場合は、UNLOAD 文を使用してデータをアンロードします。

- パフォーマンスが問題で、クエリ結果をエクスポートする場合
- テキストフォーマットで出力データを格納する場合
- アプリケーションにエクスポート文を埋め込む場合
- クライアントコンピュータにあるファイルにデータをエクスポートする場合

UNLOAD 文を SELECT とともに使用するには、適切な権限を持っている必要があります。たとえば、-gl データベースサーバオプションの設定が NONE でない限り、ほとんどの場合、SELECT ANY TABLE システム権限があれば十分です。少なくとも、UNLOAD 文で指定されたテーブルに対して SELECT を実行できる権限を持っている必要があります。

UNLOAD 文を使用できるユーザの詳細については、「-gl データベースサーバオプション」『SQL Anywhere サーバ データベース管理』を参照してください。

データベースに対する影響

OUTPUT 文、UNLOAD 文、UNLOAD TABLE 文のいずれかを選択できる場合は、パフォーマンスを考慮して UNLOAD TABLE 文を選択します。

UNLOAD 文と SELECT は、現在の独立性レベルで実行されます。

例

SQL Anywhere のサンプルデータベースを使用し、次の文を実行して Employees テーブルのサブセットをテキストファイル *GAEmployees.csv* にアンロードできます。

```
UNLOAD
SELECT * FROM Employees
WHERE State = 'GA'
TO 'C:¥¥ServerTemp¥¥GAEmployees.csv'
QUOTE '';
```

この形式の UNLOAD TABLE 文を使用すると、ファイルパスはデータベースサーバコンピュータに対する相対パスとなります。

参照

- 「クライアントコンピュータ上のデータへのアクセス」747 ページ
- 「UNLOAD 文」『SQL Anywhere サーバ SQL リファレンス』
- 「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』

アンロードユーティリティ (dbunload) を使用したデータのエクスポートに関するヒント

アンロードユーティリティ (dbunload) を使用して、1 つ、複数、またはすべてのデータベーステーブルをエクスポートします。テーブルデータとテーブルスキーマをエクスポートできます。データベースのテーブルを配置し直す場合にも dbunload を使用でき、必要な SQL スクリプトファイルを作成して必要に応じて修正します。コマンドファイルを使用すると、異なるデータベースに同じテーブルを作成できます。テーブルは、構造のみ、データのみ、または構造とデータの両方をアンロードできます。-ac オプションを使用して、既存のデータベースに直接アンロードすることもできます。

次のような場合に dbunload を使用します。

- データベースを再構築または抽出する場合
- テキストフォーマットでデータをエクスポートする場合
- 大量のデータをすばやく処理する必要がある場合
- ファイルフォーマット要件が柔軟な場合

> **注意**
> アンロードユーティリティ (dbunload) と Sybase Central の**データベースアンロードウィザード**は、機能的に同じものです。どちらを使っても同じ結果が得られます。

参照

- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』

データベースアンロードウィザードを使用したデータのエクスポートに関するヒント

データベースアンロードウィザードは、既存のデータベースを新しいデータベースにアンロードするために使用します。

データベースアンロードウィザードを使用してデータベースをアンロードする際、データベース内のすべてのオブジェクトをアンロードするのか、またはデータベースからテーブルのサブセットのみをアンロードするのかを選択できます。**[所有者フィルタの設定]** ウィンドウ内で選択したユーザ用のテーブルのみが、**データベースアンロードウィザード**に表示されます。特定のデータベースユーザに属するテーブルを表示する場合、アンロードするデータベースを右クリックし、**[所有者フィルタの設定]** をクリックしてから、表示されるウィンドウでユーザを選択します。

データベースアンロードウィザードを使用すると、データベース全体をカンマ区切りのテキストフォーマットでアンロードし、データベース全体の再作成に必要な SQL スクリプトファイルを作成することもできます。これは、SQL Remote を抽出するときや、同一または少しだけ修正した構造を持つデータベースを新しく作成するときに便利です。**データベースアンロードウィザード**は、SQL Anywhere 内での再使用を目的として SQL Anywhere ファイルをエクスポートするときに便利です。

データベースアンロードウィザードでは、再ロードファイルではなく、既存のデータベースまたは新しいデータベースへ再ロードするオプションもあります。

> **注意**
> アンロードユーティリティ (dbunload) と**データベースアンロードウィザード**は、機能的に同じものです。どちらを使っても同じ結果が得られます。

参照

- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』

データベースファイルまたは稼働しているデータベースのアンロード

Sybase Central では、**データベース アンロード ウィザード**を使用して、停止または実行中のデータベースをアンロードできます。

前提条件

変数へのアンロード時には、権限は必要ありません。それ以外の場合、必要な権限は、データベースサーバの -gl オプションによって異なります。

- -gl オプションが ALL に設定されている場合は、そのテーブルの所有者であるか、そのテーブルに対する SELECT 権限を持っているか、または SELECT ANY TABLE システム権限を持っていることが必要です。
- -gl オプションが DBA に設定されている場合は、SELECT ANY TABLE システム権限が必要です。
- -gl オプションが NONE に設定されている場合、UNLOAD は使用できません。

クライアントコンピュータにあるファイルにアンロードする場合

- WRITE CLIENT FILE 権限が必要です。
- そのファイルが置かれているディレクトリに対する書き込みパーミッションが必要です。
- allow_write_client_file データベースオプションが有効になっている必要があります。
- write_client_file セキュリティ機能が有効になっている必要があります。

内容と備考

> **注意**
> テーブルだけをアンロードするときには、テーブルを所有するユーザ ID はアンロードされません。テーブルを所有するユーザ ID を新しいデータベースに作成してからテーブルを再ロードする必要があります。

♦ タスク

1. **[ツール]** » **[SQL Anywhere 16]** » **[データベースのアンロード]** をクリックします。
2. **[データベースアンロードウィザード]** の指示に従います。

結果

指定されたデータベースがアンロードされます。

参照

- 「UNLOAD 文」『SQL Anywhere サーバ SQL リファレンス』

[データのアンロード] ウィンドウを使用したデータのエクスポート

[データのアンロード] ウィンドウを使用して、Sybase Central のテーブルをアンロードできます。

前提条件

そのテーブルの所有者であるか、そのテーブルに対する SELECT 権限を持っているか、または SELECT ANY TABLE システム権限を持っていることが必要です。

内容と備考

Sybase Central の **[データのアンロード]** ウィンドウを使用して、データベース内の 1 つ以上のテーブルをアンロードできます。この機能は、**データベースアンロードウィザード**またはアンロードユーティリティ (dbunload) でも使用できますが、このウィンドウを使用すると、**データベースアンロードウィザード**全体を実行する代わりに 1 ステップでテーブルをアンロードできます。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[テーブル]** をダブルクリックします。
3. データをエクスポートするテーブルを右クリックして、**[データのアンロード]** をクリックします。

4. **[データのアンロード]** ウィンドウの入力を完了します。**[OK]** をクリックします。

結果

データは、指定されたファイルに保存されます。

OUTPUT 文を使用したクエリ結果のエクスポート

OUTPUT 文を使用して、Interactive SQL のクエリ結果をエクスポートできます。

前提条件

問い合わせるテーブルの所有者であるか、そのテーブルに対する SELECT 権限を持っているか、または SELECT ANY TABLE システム権限を持っていることが必要です。

内容と備考

APPEND 句と VERBOSE 句を組み合わせると、結果とメッセージを両方とも既存のファイルに追加できます。たとえば、**OUTPUT TO '***ファイル名***' APPEND VERBOSE** と入力します。

OUTPUT 文の APPEND 句と VERBOSE 句は、Interactive SQL の以前のバージョンの演算子 >#、>>#、>&、>>& と同じです。これらの演算子を使用してデータをリダイレクトできますが、Interactive SQL 文を使用すると、出力がより正確になり、コードが読み取りやすくなります。

BCP FORMAT 句は、SQL Anywhere と Adaptive Server Enterprise 間でファイルのインポートとエクスポートを実行するために使用します。

♦ タスク

1. Interactive SQL の **[SQL 文]** ウィンドウ枠にクエリを入力します。
2. クエリの最後に **OUTPUT TO '***ファイル名***'** と入力します。

   たとえば、Employees テーブル全体をファイル *Employees.csv* にエクスポートする場合は、次のクエリを入力します。

   ```
   SELECT * FROM Employees;
   OUTPUT TO 'C:¥¥LocalTemp¥¥Employees.csv';
   ```

3. クエリ結果をエクスポートして、別のファイルに結果を追加するには、APPEND 句を使用します。次に例を示します。

   ```
   SELECT * FROM Employees;
   OUTPUT TO 'C:¥¥LocalTemp¥¥Employees.csv'
   APPEND;
   ```

   クエリ結果をエクスポートして、メッセージをインクルードするには、VERBOSE 句を使用します。次に例を示します。

   ```
   SELECT * FROM Employees;
   OUTPUT TO 'C:¥¥LocalTemp¥¥Employees.csv'
   VERBOSE;
   ```

4. **[SQL]** » **[実行]** をクリックします。

   エクスポートに成功すると、**[メッセージ]** タブに、クエリ結果セットのエクスポートに要した時間、エクスポートされたデータのファイル名とパス、書き込まれたローの数が表示されます。エクスポートに失敗した場合は、エクスポートに失敗したことを示すメッセージが表示されます。

結果

クエリ結果は、指定されたロケーションにエクスポートされます。

参照

- 「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「Adaptive Server Enterprise の互換性」770 ページ

UNLOAD 文を使用したクエリ結果のエクスポート

UNLOAD 文を使用して、Interactive SQL のクエリ結果をエクスポートできます。

前提条件

変数へのアンロード時には、権限は必要ありません。それ以外の場合、必要な権限は、データベースサーバの -gl オプションによって異なります。

- -gl オプションが ALL に設定されている場合は、そのテーブルの所有者であるか、そのテーブルに対する SELECT 権限を持っているか、または SELECT ANY TABLE システム権限を持っていることが必要です。

- -gl オプションが DBA に設定されている場合は、SELECT ANY TABLE システム権限が必要です。

- -gl オプションが NONE に設定されている場合、UNLOAD は使用できません。

クライアントコンピュータにあるファイルにアンロードする場合

- WRITE CLIENT FILE 権限が必要です。

- そのファイルが置かれているディレクトリに対する書き込みパーミッションが必要です。

- allow_write_client_file データベースオプションが有効になっている必要があります。

- write_client_file セキュリティ機能が有効になっている必要があります。

内容と備考

BCP FORMAT 句は、SQL Anywhere と Adaptive Server Enterprise 間でファイルのインポートとエクスポートを実行するために使用します。

♦ タスク

● **[SQL 文]** ウィンドウ枠で、UNLOAD 文を実行します。次に例を示します。

```
UNLOAD
SELECT * FROM Employees
TO 'C:¥¥ServerTemp¥¥Employees.csv';
```

エクスポートに成功すると、**[メッセージ]** タブに、クエリ結果セットのエクスポートに要した時間、エクスポートされたデータのファイル名とパス、書き込まれたローの数が表示されます。エクスポートに失敗した場合は、エクスポートに失敗したことを示すメッセージが表示されます。

この形式の UNLOAD TABLE 文を使用すると、ファイルパスはデータベースサーバコンピュータに対する相対パスとなります。

結果

クエリ結果は、指定されたロケーションにエクスポートされます。

参照

● 「UNLOAD 文」『SQL Anywhere サーバ SQL リファレンス』
● 「Adaptive Server Enterprise の互換性」770 ページ

Interactive SQL での NULL 値の処理設定

Interactive SQL の **[結果]** ウィンドウ枠を設定することにより、OUTPUT 文を使用したときに NULL 値をどのように表すのかを指定できます。

前提条件

この作業を実行するための前提条件はありません。

♦ タスク

1. Interactive SQL で、**[ツール]** » **[オプション]** をクリックします。
2. **[SQL Anywhere]** をクリックします。
3. **[結果]** タブをクリックします。
4. **[Null 値の代替文字]** フィールドに、NULL に使用する値を入力します。
5. **[OK]** をクリックします。

結果

NULL 値の代わりに表示される値が変更されます。

参照

- 「SET OPTION 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「output_nulls オプション [Interactive SQL]」『SQL Anywhere サーバ データベース管理』
- 「IFNULL 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』

データベースのエクスポート (Sybase Central の場合)

Sybase Central の**データベースアンロードウィザード**を使用して、データをデータベースから再ロードファイル、新規データベース、または既存のデータベースにアンロードできます。

前提条件

変数へのアンロード時には、権限は必要ありません。それ以外の場合、必要な権限は、データベースサーバの -gl オプションによって異なります。

- -gl オプションが ALL に設定されている場合は、そのテーブルの所有者であるか、そのテーブルに対する SELECT 権限を持っているか、または SELECT ANY TABLE システム権限を持っていることが必要です。
- -gl オプションが DBA に設定されている場合は、SELECT ANY TABLE システム権限が必要です。
- -gl オプションが NONE に設定されている場合、UNLOAD は使用できません。

クライアントコンピュータにあるファイルにアンロードする場合

- WRITE CLIENT FILE 権限が必要です。
- そのファイルが置かれているディレクトリに対する書き込みパーミッションが必要です。
- allow_write_client_file データベースオプションが有効になっている必要があります。
- write_client_file セキュリティ機能が有効になっている必要があります。

♦ タスク

1. **[ツール] » [SQL Anywhere 16] » [データベースのアンロード]** をクリックします。
2. **[データベースアンロードウィザード]** の指示に従います。

結果

データは、指定されたロケーションにアンロードされます。

参照

- 「UNLOAD 文」『SQL Anywhere サーバ SQL リファレンス』
- 「-gl データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「データベースのエクスポート (コマンドラインの場合)」744 ページ

データベースのエクスポート (コマンドラインの場合)

コマンドラインのアンロードユーティリティ (dbunload) を使用して、データをデータベースから再ロードファイル、新規データベース、または既存のデータベースにアンロードできます。

前提条件

再ロードを伴わないアンロードの場合は、SELECT ANY TABLE システム権限が必要です。再ロードを伴うアンロードの場合は、SELECT ANY TABLE システム権限と SERVER OPERATOR システム権限が必要です。

♦ タスク

- アンロード (dbunload) ユーティリティを実行し、-c オプションを使用して接続パラメータを指定します。

オプション	アクション
データベース全体をアンロードする	データベース全体をサーバコンピュータ上のディレクトリ *C:¥ServerTemp¥DataFiles* にアンロードするには、次の手順に従います。 dbunload -c "DBN=demo;UID=DBA;PWD=sql" C:¥ServerTemp¥DataFiles
データのみエクスポートする	-d オプションと -ss オプションを使用します。次に例を示します。 dbunload -c "DBN=demo;UID=DBA;PWD=sql" -d -ss C:¥ServerTemp¥DataFiles
スキーマのみエクスポートする	-n オプションを使用します。次に例を示します。 dbunload -c "DBN=demo;UID=DBA;PWD=sql" -n

スキーマの再作成とテーブルの再ロードに必要な文は、クライアントのカレントディレクトリの *reload.sql* に記述されています。

結果

データは、指定されたロケーションにアンロードされます。

参照

- 「-gl データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』
- 「データベースのエクスポート (Sybase Central の場合)」 743 ページ
- 「UNLOAD 文」『SQL Anywhere サーバ SQL リファレンス』

テーブルのエクスポート (SQL の場合)

Interactive SQL から UNLOAD TABLE 文を実行して、テーブルをエクスポートできます。

前提条件

変数へのアンロード時には、権限は必要ありません。それ以外の場合、必要な権限は、データベースサーバの -gl オプションによって異なります。

- -gl オプションが ALL に設定されている場合は、そのテーブルの所有者であるか、そのテーブルに対する SELECT 権限を持っているか、または SELECT ANY TABLE システム権限を持っていることが必要です。
- -gl オプションが DBA に設定されている場合は、SELECT ANY TABLE システム権限が必要です。
- -gl オプションが NONE に設定されている場合、UNLOAD は使用できません。

クライアントコンピュータにあるファイルにアンロードする場合

- WRITE CLIENT FILE 権限が必要です。
- そのファイルが置かれているディレクトリに対する書き込みパーミッションが必要です。
- allow_write_client_file データベースオプションが有効になっている必要があります。
- write_client_file セキュリティ機能が有効になっている必要があります。

内容と備考

テーブル内のすべてのデータを選択して、クエリ結果をエクスポートすることにより、テーブルをエクスポートできます。

♦ タスク

- UNLOAD TABLE 文を実行します。次に例を示します。

```
UNLOAD TABLE Departments
TO 'C:¥¥ServerTemp¥¥Departments.csv';
```

この文は、SQL Anywhere サンプルデータベースの Departments テーブルを、クライアントコンピュータではなくデータベースサーバコンピュータのディレクトリにある *Departments.csv* ファイルにアンロードします。このファイルのパスは SQL リテラルで指定されるため、円記号は 2 つ重ねることによりエスケープされ、'¥n' または '¥x' などのエスケープシーケンスの変換が回避されます。

テーブルの各ローは出力ファイルの 1 行に書き出されます。また、カラム名はエクスポートされません。各カラムはカンマで区切られます。デリミタに使う文字は、DELIMITED BY 句で変更できます。フィールドは固定幅ではありません。エクスポートされるのは、各エントリの文字だけで、カラム幅全体ではありません。

結果

データが指定されたファイルにエクスポートされます。

参照

- 「エクスポートウィザードを使用したデータのエクスポート」 731 ページ
- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』
- 「-gl データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「UNLOAD 文」『SQL Anywhere サーバ SQL リファレンス』

テーブルのエクスポート (コマンドラインの場合)

コマンドラインのアンロードユーティリティ (dbunload) を実行して、テーブルをエクスポートできます。

前提条件

再ロードを伴わないアンロードの場合は、SELECT ANY TABLE システム権限が必要です。再ロードを伴うアンロードの場合は、SELECT ANY TABLE システム権限と SERVER OPERATOR システム権限が必要です。

内容と備考

カンマ (,) をデリミタとしてテーブル名を区切ると、複数のテーブルをアンロードできます。

♦ タスク

- 次のコマンドを実行します。

```
dbunload -c "DBN=demo;UID=DBA;PWD=sql" -t Employees C:¥ServerTemp¥DataFiles
```

  このコマンドの -c ではデータベース接続パラメータを指定し、-t ではエクスポートする 1 つまたは複数のテーブルの名前を指定します。この dbunload コマンドは、データを SQL Anywhere サンプルデータベース (デフォルトのデータベースサーバ上で実行されていると仮定) からサーバコンピュータ上の *C:¥ServerTemp¥DataFiles* ディレクトリのファイルセットにアンロードします。データファイルからテーブルを再構築するのに必要な SQL スクリプトファイルは、デフォルト名 *reload.sql* としてクライアントのカレントディレクトリに作成されます。

結果

データは、指定されたロケーションにエクスポートされます。

参照

- 「-gl データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「UNLOAD 文」『SQL Anywhere サーバ SQL リファレンス』
- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』

クライアントコンピュータ上のデータへのアクセス

SQL Anywhere では、ファイルをデータベースサーバコンピュータにコピーしたり、データベースサーバコンピュータからファイルをコピーすることなく、SQL の文と関数を使用して、クライアントコンピュータ上のファイルからデータをロードしたり、ファイルにデータをアンロードすることができます。これを実行するために、データベースサーバは Command Sequence 通信プロトコル (CmdSeq) ファイルハンドラを使用して転送を開始します。CmdSeq ファイルハンドラは、データベースサーバがクライアントコンピュータとのデータの転送を必要としているクライアントアプリケーションから要求を受信した後、応答を送信する前に呼び出されます。このファイルハンドラは、クライアントの複数のファイルの同時転送およびインターリーブ転送を任意の時点でサポートしています。たとえば、クライアントアプリケーションにより実行された文で複数のファイルの同時転送が必要な場合、データベースサーバはそれを開始することができます。

CmdSeq ファイルハンドラを使用してクライアントデータの転送を実行することにより、アプリケーションは特殊なコードを使用する必要がなく、次に示す SQL コンポーネントを使用した機能をすぐに利用することができます。

- **READ_CLIENT_FILE 関数**　READ_CLIENT_FILE 関数は、クライアントコンピュータ上の指定したファイルからデータを読み込み、ファイルの内容を表す LONG BINARY 値を返します。この関数は、BLOB が使用可能な SQL コードの任意の位置で使用できます。可能な場合は、文で実体化の実行を明示的に指定していないかぎり、READ_CLIENT_FILE 関数によって返されるデータはメモリ内で実体化されません。たとえば、LOAD TABLE 文は実体化を行わずにクライアントファイルからデータをストリーミングします。READ_CLIENT_FILE 関数によって返された値を接続変数に割り当てると、データベースサーバはクライアントファイルの内容を取得して実体化します。

- **WRITE_CLIENT_FILE 関数**　WRITE_CLIENT_FILE 関数は、クライアントコンピュータにある、指定したファイルにデータを書き込みます。

- **READ CLIENT FILE システム権限**　READ CLIENT FILE システム権限によって、クライアントコンピュータのファイルを読み込むことができます。

- **WRITE CLIENT FILE システム権限**　WRITE CLIENT FILE システム権限によって、クライアントコンピュータのファイルに書き込むことができます。

- **LOAD TABLE ... USING CLIENT FILE 句**　USING CLIENT FILE 句を使用すると、クライアントコンピュータにあるファイル内のデータを使用してテーブルをロードできます。たとえば、LOAD TABLE ... USING CLIENT FILE 'my-file.txt'; と指定すると、クライアントコンピュータから *my-file.txt* というファイルがロードされます。

- **LOAD TABLE ... USING VALUE 句**　USING VALUE 句を使用すると、BLOB 式を値として指定できます。BLOB 式では、READ_CLIENT_FILE 関数を使用してクライアントコンピュータ上のファイルから BLOB をロードできます。たとえば、LOAD TABLE ... USING VALUE READ_CLIENT_FILE('my-file') というように指定します。*my-file* はクライアントコンピュータ上のファイルです。

- **UNLOAD TABLE ... INTO CLIENT FILE 句**　INTO CLIENT FILE 句を使用すると、データのアンロード先としてクライアントコンピュータにあるファイルを指定できます。

- **UNLOAD TABLE ... INTO VARIABLE 句**　INTO VARIABLE 句を使用すると、データのアンロード先として変数を指定できます。

- **read_client_file および write_client_file セキュリティ機能**　read_client_file および write_client_file セキュリティ機能は、クライアントファイルの読み込みまたは書き込みを行う文の使用を制御します。

プロシージャからのクライアントファイルの読み込みと書き込みを許可するには、関数またはその他の間接文、コールバック関数を登録します。コールバック関数が呼び出され、直接要求していないクライアント転送をアプリケーションが許可することが確認されます。

参照

- 「セキュリティ機能キーの作成」『SQL Anywhere サーバ データベース管理』
- 「-sf データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「UNLOAD 文」『SQL Anywhere サーバ SQL リファレンス』
- 「LOAD TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「READ_CLIENT_FILE 関数 [文字列]」『SQL Anywhere サーバ SQL リファレンス』
- 「WRITE_CLIENT_FILE 関数 [文字列]」『SQL Anywhere サーバ SQL リファレンス』
- 「SQLSetConnectAttr 拡張接続属性」『SQL Anywhere サーバ プログラミング』 (ODBC)
- 「db_register_a_callback 関数」『SQL Anywhere サーバ プログラミング』 (ESQL)
- 「JDBC コールバック」『SQL Anywhere サーバ プログラミング』 (JDBC)

クライアント側データセキュリティ

SQL Anywhere は、クライアントファイルの転送において、データベースサーバコンピュータとは異なる場所に存在することの多いクライアントコンピュータ上のデータを不正に転送できないようにする手段を提供しています。

そのため、データベースサーバは各文の実行元を追跡し、文がクライアントアプリケーションから直接受信されたものかどうかを判断します。クライアントからの新しいファイルの転送を開始する際に、データベースサーバは文の実行元に関する情報を付加します。文がクライアントアプリケーションから直接送信された場合は、CmdSeq ファイルハンドラによってファイルの転送が許可されます。文がクライアントアプリケーションから直接送信されたものではない場合、アプリケーションは検証のコールバックを登録する必要があります。コールバックが登録されないと、転送が拒否されて文が失敗し、エラーが発生します。

また、接続が正常に確立されるまで、クライアントデータの転送は許可されません。この制限により、接続文字列やログインプロシージャを使用した不正アクセスを防止できます。

許可されたユーザを装ってシステムへのアクセスを試みるユーザからの保護を実現するには、転送データの暗号化を行ってください。

また、SQL Anywhere では、さまざまなレベルでアクセスを制御する次のセキュリティメカニズムも提供しています。

- **サーバレベルのセキュリティ**　read_client_file および write_client_file セキュリティ機能により、サーバワイドでクライアント側の転送をすべて無効にできます。

- **アプリケーションレベルおよび DBA レベルのセキュリティ**　allow_read_client_file および allow_write_client_file データベースオプションにより、データベース、ユーザ、接続レベルでのアクセス制御を実現します。たとえば、アプリケーションの接続後にこのデータベースオプションを OFF に設定することにより、アプリケーションがクライアント側の転送に使用されるのを防止できます。

- **ユーザレベルのセキュリティ**　READ CLIENT FILE および WRITE CLIENT FILE システム権限は、それぞれクライアントコンピュータからのデータの読み込みと、クライアントコンピュータへのデータの書き込みに関して、ユーザレベルのアクセス制御を提供します。

参照

- 「-sf データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「allow_read_client_file オプション」『SQL Anywhere サーバ データベース管理』

クライアント側データのロード時のリカバリ

トランザクションログから LOAD TABLE 文のリカバリが必要になった場合、データのロードに使用したクライアントコンピュータ上のファイルは SQL Anywhere ですでに使用できないか、変更されている可能性が高いため、元のデータが使用できなくなります。このような状況が発生するのを防ぐため、ロギングがオフになっていないことを確認してください。また、データのロード時に WITH ROW LOGGING または WITH CONTENT LOGGING 句のいずれかを指定してください。これらの句を指定することにより、ロードしているデータがトランザクションログに記録されるため、リカバリ時にトランザクションログのリプレイが可能になります。

WITH ROW LOGGING では、挿入された各ローがトランザクションログに INSERT 文として記録されます。WITH CONTENT LOGGING では、挿入されたデータが、データベースサーバがリカバリ時に処理するためにチャンク単位でトランザクションログに記録されます。どちらの方法も、リカバリ時にクライアント側データをロードできるようにするには適しています。ただし、同期を行っているデータベースにデータをロードする場合は、WITH CONTENT LOGGING は使用できません。

次のいずれかの LOAD TABLE 文を指定する際に、ロギングレベルが指定されていない場合は、WITH CONTENT LOGGING がデフォルトの動作となります。

- LOAD TABLE...USING CLIENT FILE *client-filename-expression*

- LOAD TABLE...USING VALUE *value-expression*

- LOAD TABLE...USING COLUMN *column-expression*

データベースの再構築

データベースの再構築は、データベース全体のアンロードと再ロードを伴うインポートとエクスポートの一種です。再構築 (アンロード／ロード) ツールと抽出ツールを使用して、データベースの再構築、既存のデータベースの一部からの新しいデータベースの作成、未使用のページの排除を行います。

データベースは、Sybase Central や dbunload で再構築できます。

注意
データベースを再構築する場合、特に元のデータベースを再構築したデータベースに置き換える場合は、再構築を実行する前にデータベースのバックアップを作成するようにしてください。

インポートとエクスポートの場合、データの送信先はデータベース内またはデータベース外になります。インポートでは、データはデータベースに読み込まれます。エクスポートでは、データはデータベースから書き出されます。情報を、SQL Anywhere 以外の別のデータベースから受け取ったり、別のデータベースに送信したりすることがよくあります。

暗号化オプション -ek、-ep または -et を指定する場合は、*reload.sql* ファイルの LOAD TABLE 文に暗号化キーを含めます。ハードコーディングされたキーはセキュリティの低下を招くため、*reload.sql* ファイルのパラメータには暗号化キーが指定されます。Interactive SQL 内で *reload.sql* ファイルを実行する際に、パラメータとして暗号化キーを指定してください。READ 文にキーを指定していない場合、Interactive SQL でキーの入力を要求するプロンプトが表示されます。

ロードとアンロードでは、SQL Anywhere のデータベースからデータとスキーマを取り出し、それらを別の SQL Anywhere データベースに入れ直します。アンロードプロシージャでは、データファイルと、テーブルを正確に再作成するために必要なテーブル定義を含む *reload.sql* ファイルが生成されます。*reload.sql* スクリプトを実行すると、テーブルが再作成され、そこに元のデータがロードされます。

データベースの再構築には時間がかかる可能性があり、大量のディスク領域が必要になることがあります。また、データベースのアンロードと再ロード中は、そのデータベースを使用できません。このため、明確な目的がないかぎり、運用環境でデータベースを再構築しないでください。

特定の SQL Anywhere データベースから別の SQL Anywhere データベースへ

再構築では、通常 SQL Anywhere データベースからデータをコピーし、それを別の SQL Anywhere データベースに再ロードします。アンロードと再ロードは関連しています。通常はどちらか一方ではなく、両方を行います。

再構築とエクスポート

再構築がエクスポートとは異なる点は、再構築では、データの他にテーブル定義とスキーマのエクスポートとインポートが行われることです。再構築処理のアンロード部分では、テキストフォーマットのデータファイルと、テーブルとその他の定義が含まれる *reload.sql* ファイルが生成されます。*reload.sql* スクリプトを実行すると、テーブルが再作成され、そこにデータがロードされます。

SQL Remote または Mobile Link を使用している場合は、(古いデータベースから新しいデータベースを作成して) データベースを抽出することを検討します。

レプリケートするデータベースの再構築

データベース再構築のプロシージャは、データベースがレプリケーションに関連するかしないかによって異なります。データベースがレプリケーションに関連する場合は、操作中はトランザクションログのオフセットを保存しておいてください。これは、Message Agent がこの情報を必要とするためです。データベースがレプリケーションに関連しない場合、処理はもっと簡単です。

参照

- 「データベース再構築時のダウン時間の最短化」759 ページ
- 「同期やレプリケーションに関連するデータベースの再構築 (コマンドラインの場合)」754 ページ
- 「同期やレプリケーションに関連しないデータベースの再構築」752 ページ
- 「データベースの照合の変更」『SQL Anywhere サーバ データベース管理』
- 「マテリアライズドビューの手動再表示」63 ページ
- 「Interactive SQL ユーティリティ (dbisql)」『SQL Anywhere サーバ データベース管理』
- 「内部アンロードと外部アンロード、内部再ロードと外部再ロード」『SQL Anywhere サーバ データベース管理』
- 「データベース抽出」760 ページ
- 「バックアップとデータリカバリ」『SQL Anywhere サーバ データベース管理』

データベースを再構築する理由

さまざまな理由で、データベースの再構築を検討します。次の処理を行う場合は、データベースを再構築します。

- **データベースのファイルフォーマットのアップグレード**　アップグレードユーティリティを適用すると一部の新機能が使用可能になります。ただし、データベースのファイルフォーマットのアップグレードを必要とする機能もあります。ファイルフォーマットのアップグレードとは、データベースをアンロードして再ロードすることです。

  新しいバージョンの SQL Anywhere データベースサーバは、データベースをアップグレードしないで使用できます。新しいシステムテーブルまたはデータベースオプションにアクセスする必要がある新しいバージョンの機能を使用する場合は、アップグレードユーティリティを使用してデータベースをアップグレードしてください。アップグレードユーティリティでは、データをアンロードまたは再ロードしません。

  データベースファイルフォーマットの変更を伴う新しいバージョンの SQL Anywhere を使用する場合は、データベースをアンロードして再ロードしてください。データベースをバックアップしてから再構築してください。

  > **注意**
  > バージョン 9 より前からアップグレードする場合は、データベースファイルを再構築する必要があります。バージョン 10.0.0 以降からアップグレードする場合は、アップグレードユーティリティを使用するか、データベースを再構築します。

- **ディスク領域を再利用する場合**　データを削除しても、データベースは縮小されません。代わりに、空のページが、再使用できるように空き領域としてマーク付けされます。データベースを再構築しないかぎり、空のページがデータベースから削除されることはありません。データベースからデータを大量に削除し、データをそれ以上追加しない場合は、データベースを再構築するとディスク領域を再利用できます。

- **データベースパフォーマンスを向上させる場合**　データベースを再構築すると、パフォーマンスが向上する場合があります。プライマリキーの順にデータベースをアンロードして再ロードできるので、関連するローが同じページまたは周辺のページに表示されるため、関連情報に速くアクセスできる。

注意
テーブルが極端に断片化されているためにパフォーマンスが低下していることが判明した場合は、テーブルを再編成します。

参照

- 「SQL Anywhere 16 へのアップグレード方法」『SQL Anywhere 16 変更点とアップグレード』
- 「データベースミラーリングシステムでのアップグレードと再構築」『SQL Anywhere 16 変更点とアップグレード』
- 「REORGANIZE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「アップグレードユーティリティ (dbupgrad)」『SQL Anywhere サーバ データベース管理』
- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』

アンロードユーティリティを使用したデータベースの再構築に関するヒント

アンロードユーティリティ (dbunload) を使用して、データベース全体をカンマ区切りのテキストフォーマットでアンロードし、データベース全体の再作成に必要な SQL スクリプトファイルを作成することもできます。たとえば、SQL Remote 抽出を作成したり、同一または少しだけ修正した構造を持つデータベースを新しく作成するために、これらのファイルを使用できます。

次の場合は、アンロード (dbunload) ユーティリティを使用します。

- データベースを再構築する場合、またはデータベースからデータを抽出する場合
- テキストフォーマットでエクスポートする場合
- 大量のデータをすばやく処理する必要がある場合
- ファイルフォーマット要件が柔軟な場合

注意
アンロードユーティリティ (dbunload) と**データベースアンロードウィザード**は、機能的に同じものです。どちらを使っても同じ結果が得られます。Interactive SQL の SQL OUTPUT 文または SQL UNLOAD 文を使用することによっても、データベースをアンロードできます。

参照

- 「同期やレプリケーションに関連しないデータベースの再構築」752 ページ
- 「同期やレプリケーションに関連するデータベースの再構築 (コマンドラインの場合)」754 ページ

同期やレプリケーションに関連しないデータベースの再構築

アンロードユーティリティ (dbunload) を使用すると、データベースをアンロードして新規データベースに再構築したり、既存のデータベースに再ロードしたり、既存のデータベースを置き換えたりすることができます。

前提条件

同期やレプリケーションに関連しないデータベースの場合にのみ、次の手順を使用してください。

SELECT ANY TABLE および SERVER OPERATOR システム権限が必要です。

内容と備考

-an オプションと -ar オプションは、パーソナルサーバへの接続、または共有メモリ経由によるネットワークサーバへの接続にのみ適用されます。-ar や -an オプションを指定すると、Sybase Central で**データベースアンロードウィザード**を使用した場合より高速で処理できますが、-ac を指定すると処理は**データベースアンロードウィザード**よりも遅くなります。

dbunload には他にもオプションがあり、実行中のデータベースまたは実行されていないデータベースを指定したり、データベースパラメータを指定することもできます。

♦ タスク

1. 次のいずれかのオプションを指定してアンロードユーティリティ (dbunload) を実行します。

処理	使用するオプション	例
新規データベースへの再構築	-an	dbunload -c "DBF=demo.db;UID=DBA;PWD=sql" -an DemoBackup.db
既存のデータベースへの再ロード	-ac	dbunload -c "DBF=demo.db;UID=DBA;PWD=sql" -ac "UID=DBA;PWD=sql;DBF=mynewdemo.db"
既存のデータベースの置換	-ar	dbunload -c "DBF=demo.db;UID=DBA;PWD=sql" -ar

   これらのオプションの 1 つを使用した場合、ディスク上にデータの中間コピーが作成されないため、コマンドラインではアンロードディレクトリを指定する必要なありません。このため、データのセキュリティが向上します。

2. 再ロードしたデータベースを使用する前に、データベースを停止し、トランザクションログを圧縮します。

結果

データベースは、指定されたロケーションにアンロードされます。

参照

- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』

同期やレプリケーションに関連するデータベースの再構築 (コマンドラインの場合)

dbunload -ar オプションを使用して、同期やレプリケーションに関連するデータベースを再構築できます。その際は、同期やレプリケーションに干渉しない方法でデータベースがアンロードおよび再ロードされます。

前提条件

SELECT ANY TABLE および SERVER OPERATOR システム権限が必要です。

すべてのサブスクリプションを同期してから、Mobile Link 同期に参加するデータベースを再構築する必要があります。

内容と備考

この項の内容は、SQL Anywhere の Mobile Link クライアント (dbmlsync を使用するクライアント) と SQL Remote に適用されます。

同期やレプリケーションは、トランザクションログのオフセットに基づいています。データベースを再構築すると、古いトランザクションログのオフセットは新しいログのオフセットとは異なるため、古いログは使用できなくなります。このため、同期やレプリケーションに参加しているときは特に、データベースを確実にバックアップすることが重要です。

注意
dbunload には他にもオプションがあり、実行中のデータベースまたは実行されていないデータベースを指定したり、データベースパラメータを指定することもできます。

♦ タスク

1. データベースを停止します。
2. データベースファイルとトランザクションログファイルを安全なロケーションにコピーしてオフラインでフルバックアップを実行します。
3. 次 dbunload のコマンドを実行して、データベースを再構築します。

```
dbunload -c connection-string -ar directory
```

connection-string は適切な権限による接続を表し、*directory* はレプリケーション環境で使用した古いトランザクションログのディレクトリを表します。データベースへのその他の接続はありません。

-ar オプションは、パーソナルサーバへの接続、または共有メモリ経由によるネットワークサーバへの接続にのみ適用されます。

4. 新しいデータベースを停止してから、データベースの復元後に通常実行する妥当性検査を実行します。

5. 必要な運用オプションを使用してデータベースを起動します。これで、再ロードされたデータベースにアクセスできます。

結果

データベースは再ロードされ、起動します。

参照

- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』
- 「データベースの検証 (Sybase Central の場合)」『SQL Anywhere サーバ データベース管理』
- 「データベースの再構築」 749 ページ
- 「Mobile Link のアップグレード」『SQL Anywhere 16 変更点とアップグレード』
- 「SQL Remote のアップグレード」『SQL Anywhere 16 変更点とアップグレード』

同期やレプリケーションに関連するデータベースの再構築 (手動の場合)

同期やレプリケーションに関連するデータベースを手動で再構築できます。

前提条件

データベースを再構築するには、SELECT ANY TABLE および SERVER OPERATOR システム権限が必要です。

すべてのサブスクリプションを同期してから、Mobile Link 同期に参加するデータベースを再構築する必要があります。

内容と備考

この項の内容は、SQL Anywhere の Mobile Link クライアント (dbmlsync を使用するクライアント) と SQL Remote に適用されます。

同期やレプリケーションは、トランザクションログのオフセットに基づいています。データベースを再構築すると、古いトランザクションログのオフセットは新しいログのオフセットとは異なるため、古いログは使用できなくなります。このため、同期やレプリケーションに参加しているときは特に、データベースを確実にバックアップすることが重要です。

♦ **タスク**

1. データベースを停止します。
2. データベースファイルとトランザクションログファイルを安全なロケーションにコピーして、オフラインでフルバックアップを実行します。
3. dbtran ユーティリティを実行してデータベースの現在のトランザクションログファイルの開始オフセットと終了オフセットを表示します。

   終了オフセットは手順 8 で使用するために記録しておきます。
4. 現在のトランザクションログファイルの名前を変更して、アンロード処理中に修正されないようにします。このファイルを dbremote のオフラインログディレクトリに置きます。
5. データベースを再構築します。
6. 新しいデータベースを停止します。
7. 新しいデータベースで使用されていたトランザクションログファイルを消去します。
8. 新しいデータベースで dblog を実行します。手順 3 で記録した終了オフセットを -z オプションに指定し、相対オフセットを 0 に設定します。

   ```
   dblog -x 0 -z 0000698242 -ir -is database-name.db
   ```
9. Message Agent を実行するときは、コマンドラインに元のオフラインディレクトリのパスを入力します。
10. データベースを起動します。これで、再ロードされたデータベースにアクセスできます。

結果

データベースが再ロードされ、起動します。

参照

- 「データベースの検証 (Sybase Central の場合)」『SQL Anywhere サーバ データベース管理』
- 「データベースの再構築」 749 ページ
- 「トランザクションログユーティリティ (dblog)」『SQL Anywhere サーバ データベース管理』
- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』
- 「ログ変換ユーティリティ (dbtran)」『SQL Anywhere サーバ データベース管理』
- 「トランザクションログユーティリティ (dblog)」『SQL Anywhere サーバ データベース管理』

UNLOAD TABLE 文を使用したデータベースの再構築に関するヒント

UNLOAD TABLE 文を使用すると、特定の文字コードでデータを効率的にエクスポートできます。テキストフォーマットでデータをエクスポートする場合は、UNLOAD TABLE 文を使用してデータベースを再構築してください。

データベースに対する影響

UNLOAD TABLE 文は、テーブル全体に排他ロックを配置します。

参照

- 「UNLOAD 文」『SQL Anywhere サーバ SQL リファレンス』

テーブルデータのエクスポート

アンロードユーティリティではテーブルデータのみをアンロードできます。

前提条件

問い合わせるテーブルの所有者であるか、そのテーブルに対する SELECT 権限を持っているか、または SELECT ANY TABLE システム権限を持っていることが必要です。

内容と備考

スキーマの再作成と指定したテーブルの再ロードに必要な文は、現在のローカルディレクトリの *reload.sql* に記述されています。

カンマでテーブル名を区切ると、複数のテーブルをアンロードできます。

♦ タスク

- dbunload コマンドを実行します。その際には -c オプションを使用して接続パラメータを指定し、-t オプションを使用してデータをエクスポートするテーブルを指定し、-ss オプションを使用してカラム統計を表示しないかどうかを指定し、-d オプションを使用してデータのみをアンロードするかどうかを指定します。

  たとえば、Employees テーブルからデータをエクスポートするには、次のコマンドを実行します。

  ```
  dbunload -c "DBN=demo;UID=DBA;PWD=sql" -ss -d -t Employees C:¥ServerTemp¥DataFiles
  ```

  reload.sql ファイルはクライアントのカレントディレクトリに書き込まれ、Employees テーブルのデータを再ロードするために必要な LOAD TABLE 文が含まれます。データファイルはサーバディレクトリ *C:¥ServerTemp¥DataFiles* に書き込まれます。

結果

指定されたディレクトリにテーブルデータがエクスポートされます。

参照

- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』

テーブルスキーマのエクスポート

アンロードユーティリティには、テーブルスキーマのみをアンロードするためのオプションがあります。

前提条件

そのテーブルの所有者であるか、そのテーブルに対する SELECT 権限を持っているか、または SELECT ANY TABLE システム権限を持っていることが必要です。

内容と備考

スキーマの再作成と指定されたテーブルの再ロードに必要な文は、クライアントのカレントディレクトリの *reload.sql* に記述されています。

カンマをデリミタとしてテーブル名を区切ると、複数のテーブルをアンロードできます。

♦ タスク

- dbunload コマンドを実行します。その際には -c オプションを使用して接続パラメータを指定し、-t オプションを使用してデータをエクスポートするテーブルを指定し、-n オプションを指定してスキーマのみをアンロードするかどうかを指定します。

  たとえば、Employees テーブルからスキーマのみをエクスポートするには、次のコマンドを実行します。

  ```
  dbunload -c "DBN=demo;UID=DBA;PWD=sql" -n -t Employees
  ```

結果

テーブルスキーマがエクスポートされます。

参照

- 「アンロードユーティリティ (dbunload)」『SQL Anywhere サーバ データベース管理』

データベースの再ロード

データベースは、コマンドラインから再ロードできます。再ロードでは、空のデータベースファイルを作成し、*reload.sql* ファイルを使用してスキーマを作成し、別の SQL Anywhere データベースからアンロードしたすべてのデータを新規に作成したテーブルに挿入します。

前提条件

reload.sql ファイルが必要です。

♦ タスク

1. dbinit ユーティリティを実行して空の新しいデータベースファイルを作成します。

2. 新しいデータベースに接続します。

3. *reload.sql* スクリプトを実行します。

結果

データベースが再ロードされます。

例

次のコマンドは、*mynewdemo.db* という名前のファイルを作成します。

```
dbinit -dba DBA,sql mynewdemo.db
```

次のコマンドは、*reload.sql* スクリプトを現在のディレクトリにロードして実行します。

```
dbisql -c "DBF=mynewdemo;UID=DBA;PWD=sql" reload.sql
```

参照

- 「初期化ユーティリティ (dbinit)」『SQL Anywhere サーバ データベース管理』
- 「Interactive SQL ユーティリティ (dbisql)」『SQL Anywhere サーバ データベース管理』
- 「データベースの再構築」 749 ページ

データベース再構築時のダウン時間の最短化

バックアップユーティリティやログ変換ユーティリティを使用して、データベースを再構築するときのダウン時間を最小限に抑えることができます。

前提条件

データベースファイルのバックアップコピーを作成してから、データベースを再構築することをおすすめします。

トランザクションログの名前を変更するようなバックアップが他に予定されていないことを確認します。ログの名前が変更されている場合は、名前が変更されたトランザクションログのトランザクションを、再構築されたデータベースに適切な順序で適用してください。

BACKUP DATABASE システム権限が必要です。

♦ タスク

1. dbbackup -r を使用し、データベースとログのバックアップを作成してトランザクションログの名前を変更します。

2. バックアップデータベースを別のコンピュータ上で再構築します。

3. 運用サーバ上で再度 dbbackup -r を実行してトランザクションログの名前を変更してください。

4. トランザクションログに対して dbtran ユーティリティを実行し、再構築したデータベースにトランザクションを適用します。

5. 運用サーバを停止し、データベースとトランザクションログをコピーします。
6. 再構築したデータベースを運用サーバにコピーします。
7. 手順 5 のトランザクションログに対して dbtran を実行します。
8. 再構築されたデータベースをパーソナルサーバ (dbeng16) で開始し、ユーザが接続できないことを確認します。
9. 手順 8 のトランザクションを適用します。
10. データベースサーバを停止し、データベースをネットワークサーバ (dbsrv16) で開始して、ユーザが接続できるようにします。

結果

データベースを再構築するときのダウン時間が最小限に抑えられます。

参照

- 「バックアップユーティリティ (dbbackup)」『SQL Anywhere サーバ データベース管理』
- 「ログ変換ユーティリティ (dbtran)」『SQL Anywhere サーバ データベース管理』

データベース抽出

データベースの抽出は、SQL Remote で使用します。SQL Anywhere の統合データベースから SQL Anywhere のリモートデータベースを抽出します。

Sybase Central の**データベース抽出ウィザード**または抽出ユーティリティを使用して、データベースを抽出できます。SQL Remote レプリケーション用に統合データベースからリモートデータベースを作成する場合は、抽出ユーティリティ (dbxtract) を使用してください。

参照

- 「抽出ユーティリティ (dbxtract)」『SQL Remote』
- 「リモートデータベースの抽出」『SQL Remote』
- 「プロトタイプをカスタマイズすることによる Mobile Link リモートデータベースの配備」『Mobile Link クライアント管理』

SQL Anywhere へのデータベース移行

sa_migrate システムプロシージャまたは**データベース移行ウィザード**を使用して、次のソースからテーブルをインポートします。

- SQL Anywhere
- Ultra Light
- Sybase Adaptive Server Enterprise
- IBM DB2
- Microsoft SQL Server
- Microsoft Access
- Oracle
- MySQL
- Advantage Database Server
- リモートサーバに接続する汎用 ODBC ドライバ

データベース移行ウィザードやシステムプロシージャの sa_migrate セットを使用してデータを移行する前に、まず「ターゲットデータベース」を作成してください。ターゲットデータベースとは、データの移行先となるデータベースのことです。

参照

- 「データベースの作成」『SQL Anywhere サーバ データベース管理』

データベース移行ウィザードの使用

Sybase Central では、**データベース移行ウィザード**を使用すると、リモートデータベースに接続するためのリモートサーバ、および (必要に応じて) 現在のユーザをリモートデータベースに接続するための外部ログインを作成できます。

前提条件

事前にリモートサーバを作成しておく必要があります。ターゲットデータベース内にそのテーブルを所有するユーザが必要です。

CREATE PROXY TABLE システム権限と CREATE TABLE システム権限の両方を持っているか、次のシステム権限をすべて持っている必要があります。

- CREATE ANY TABLE
- ALTER ANY TABLE
- DROP ANY TABLE
- INSERT ANY TABLE
- SELECT ANY TABLE
- CREATE ANY INDEX

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。

2. **[ツール]** » **[SQL Anywhere 16]** » **[データベースの移行]** をクリックします。
3. **[次へ]** をクリックします。
4. 対象のデータベースを選択して **[次へ]** をクリックします。
5. リモートデータベースへの接続に使用するリモートサーバを選択し、**[次へ]** をクリックします。

   リモートサーバ用の外部ログインも作成できます。デフォルトでは、SQL Anywhere は、現在のユーザに代わってリモートサーバに接続する場合に、常にそのユーザのユーザ ID とパスワードを使用します。ただし、リモートサーバに現在のユーザと同じユーザ ID とパスワードで定義されたユーザがいない場合は、外部ログインを作成する必要があります。外部ログインは、現在のユーザ用に代替ログイン名とパスワードを割り当ててリモートサーバに接続できるようにします。
6. 移行するテーブルを選択し、**[次へ]** をクリックします。

   システムテーブルは移行できないので、このリストにはシステムテーブルは表示されません。
7. ターゲットデータベースのテーブルを所有するユーザを選択し、**[次へ]** をクリックします。
8. リモートテーブルからデータと外部キーを移行するかどうか、また移行プロセスのために作成されたプロキシテーブルを保持するかどうかを選択し、**[次へ]** をクリックします。
9. **[完了]** をクリックします。

結果

指定されたテーブルが移行されます。

参照

- 「CREATE SERVER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「リモートサーバの作成 (Sybase Central の場合)」775 ページ
- 「ユーザの作成 (Sybase Central の場合)」『SQL Anywhere サーバ データベース管理』
- 「外部ログインの作成 (Sybase Central の場合)」789 ページ

sa_migrate システムプロシージャ

sa_migrate システムプロシージャを使用して、リモートデータを移行します。テーブルや外部キーのマッピングを削除する場合は、拡張メソッドを使用します。

参照

- 「sa_migrate システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_migrate_create_fks システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_migrate_create_remote_fks_list システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_migrate_create_remote_table_list システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_migrate_create_tables システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_migrate_data システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_migrate_drop_proxy_tables システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

sa_migrate システムプロシージャを使用したすべてのテーブルの移行

sa_migrate システムプロシージャを使用して、すべてのテーブルを移行できます。

前提条件

次のシステム権限が必要です。

●CREATE TABLE または CREATE ANY TABLE (ベーステーブルの所有者でない場合)

●SELECT ANY TABLE (ベーステーブルの所有者でない場合)

●INSERT ANY TABLE (ベーステーブルの所有者でない場合)

●ALTER ANY TABLE (ベーステーブルの所有者でない場合)

●CREATE ANY INDEX (ベーステーブルの所有者でない場合)

●DROP ANY TABLE (ベーステーブルの所有者でない場合)

ターゲットデータベース内に移行対象のテーブルを所有するユーザが必要です。

外部ログインを作成するには、MANAGE ANY USER システム権限が必要です。

内容と備考

リモートデータベースで、テーブルの所有者が異なっていても名前が同じ場合、それらのテーブルはターゲットデータベースに移行すると 1 人の所有者に属します。したがって、一度に移行するテーブルは、1 人の所有者に関連付けられたテーブルだけにしてください。

ターゲットデータベースに移行後、テーブルをすべて同じユーザによって所有されないようにする場合は、ターゲットデータベース上の所有者ごとに、*local_table_owner* 引数と *owner_name* 引数を指定して sa_migrate プロシージャを実行します。

♦ タスク

1. Interactive SQL からターゲットデータベースに接続します。

2. リモートサーバを作成してリモートデータベースに接続します。

3. (省略可) リモートデータベースに接続できるように外部ログインを作成します。この手順は、ユーザがターゲットデータベースとリモートデータベースで異なるパスワードを使用している場合、またはターゲットデータベースで使用しているユーザ ID とは異なるユーザ ID でリモートデータベースにログインする場合にのみ必要です。

4. **[SQL 文]** ウィンドウ枠で、sa_migrate システムプロシージャを実行します。table_name パラメータと owner_name パラメータの両方に NULL を指定すると、データベース内のシステムテーブルを含む、すべてのテーブルが移行されます。

   例：

   ```
   CALL sa_migrate( 'local_user1', 'rmt_server1', NULL, 'remote_user1', NULL, 1, 1, 1 );
   ```

結果

このプロシージャは複数のプロシージャを順番に呼び出し、指定された基準を使用してユーザ remote_user1 に属するリモートテーブルをすべて移行します。

参照

- 「ユーザの作成 (Sybase Central の場合)」『SQL Anywhere サーバ データベース管理』
- 「sa_migrate システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「リモートサーバ」772 ページ
- 「外部ログイン」789 ページ

データベースマイグレーションシステムプロシージャを使用した個別のテーブルの移行

データベースマイグレーションシステムプロシージャを使用して、個別のテーブルを移行できます。

前提条件

次のシステム権限が必要です。

- CREATE ANY TABLE
- CREATE ANY INDEX
- INSERT ANY TABLE
- SELECT ANY TABLE
- ALTER ANY TABLE
- DROP ANY TABLE

事前にリモートサーバを作成しておく必要があります。ターゲットデータベース内にそのテーブルを所有するユーザが必要です。

外部ログインを作成するには、MANAGE ANY USER システム権限が必要です。

内容と備考

table-name と *owner-name* パラメータには、NULL を入力しないでください。両方に NULL を指定すると、システムテーブルを含む、データベース内のすべてのテーブルが移行します。また、リモートデータベースで、テーブルの所有者が異なっていても名前が同じ場合、それらのテーブルはターゲットデータベースに移行すると 1 人の所有者に属します。一度に移行するテーブルは、1 人の所有者に関連付けられたテーブルだけにすることをおすすめします。

♦ タスク

1. ターゲットデータベースを作成します。

2. Interactive SQL からターゲットデータベースに接続します。

3. (省略可) リモートデータベースに接続できるように外部ログインを作成します。外部ログインは、ユーザがターゲットデータベースとリモートデータベースで異なるパスワードを使用している場合、またはターゲットデータベースで使用しているユーザ ID とは異なるユーザ ID でリモートデータベースにログインする場合にのみ必要です。

4. sa_migrate_create_remote_table_list システムプロシージャを実行します。次に例を示します。

   ```
   CALL sa_migrate_create_remote_table_list( 'rmt_server1',
       NULL, 'remote_user1', 'mydb' );
   ```

   リモートサーバの名前を指定してください。

   このプロシージャにより、移行するリモートテーブルのリストが dbo.migrate_remote_table_list テーブルに移植されます。このテーブルから、移行しないリモートテーブルのローを削除できます。

5. sa_migrate_create_tables システムプロシージャを実行します。次に例を示します。

   ```
   CALL sa_migrate_create_tables( 'local_user1' );
   ```

   このプロシージャは、dbo.migrate_remote_table_list からリモートテーブルのリストを取り出し、リストにあるテーブルごとにプロキシテーブルとベーステーブルを作成します。また、移行したテーブルのプライマリキーインデックスもすべて作成します。

6. リモートテーブルからターゲットデータベース上のベーステーブルにデータを移行する場合は、sa_migrate_data システムプロシージャを実行します。次に例を示します。

   ```
   CALL sa_migrate_data( 'local_user1' );
   ```

   このプロシージャは、各リモートテーブルから、sa_migrate_create_tables プロシージャで作成されたベーステーブルにデータを移行します。

   リモートデータベースから外部キーを移行しない場合は、手順 10 に進みます。

7. sa_migrate_create_remote_fks_list システムプロシージャを実行します。次に例を示します。

   ```
   CALL sa_migrate_create_remote_fks_list( 'rmt_server1' );
   ```

このプロシージャは、dbo.migrate_remote_table_list にリストされている各リモートテーブルに関連した外部キーのリストを、テーブル dbo.migrate_remote_fks_list に設定します。

再作成しない外部キーマッピングを、ローカルのベーステーブルから削除できます。

8. sa_migrate_create_fks システムプロシージャを実行します。次に例を示します。

```
CALL sa_migrate_create_fks( 'local_user1' );
```

このプロシージャは、dbo.migrate_remote_fks_list に定義されている外部キーマッピングをベーステーブル上に作成します。

9. 移行用に作成されたプロキシテーブルを削除する場合は、sa_migrate_drop_proxy_tables システムプロシージャを実行します。次に例を示します。

```
CALL sa_migrate_drop_proxy_tables( 'local_user1' );
```

結果

このプロシージャは、移行用に作成したプロキシテーブルをすべて削除し、移行プロセスを完了します。

参照

- 「sa_migrate_create_remote_table_list システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_migrate_create_tables システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_migrate_data システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_migrate_create_remote_fks_list システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_migrate_create_fks システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_migrate_drop_proxy_tables システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「リモートサーバ」772 ページ
- 「外部ログイン」789 ページ

SQL スクリプトファイル

「SQL スクリプトファイル」は、SQL 文を含むテキストファイルであり、同じ SQL 文を繰り返し実行する場合に便利です。スクリプトファイルは手動で作成できますが、データベースユーティリティによって自動的に作成することもできます。たとえば、アンロードユーティリティ (dbunload) を使うと、データベースの再構築に必要な SQL 文で構成されたスクリプトファイルを作成できます。

SQL スクリプトファイルの作成

SQL スクリプトファイルの作成には好きなテキストエディタを使用できますが、SQL スクリプトファイルの作成には、Interactive SQL の使用をおすすめします。実行される SQL 文にはコメント行を含めることができます。

> **注意**
> Interactive SQL では、SQL スクリプトファイルをお気に入りから **[SQL 文]** ウィンドウ枠にロードすることができます。

参照

- 「コメント」『SQL Anywhere サーバ SQL リファレンス』
- 「READ 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「Interactive SQL のカスタマイズ」『SQL Anywhere サーバ データベース管理』
- 「SQL スクリプトファイル、SQL 文、および接続をお気に入りに追加する」『SQL Anywhere サーバ データベース管理』

SQL スクリプトファイルをロードしないで実行

Interactive SQL では、SQL スクリプトファイルを **[SQL 文]** ウィンドウ枠にロードすることなく実行できます。

前提条件

Interactive SQL が *.sql* ファイルのデフォルトエディタとして設定されていることを確認します。

Interactive SQL で、**[ツール]** » **[オプション]** » **[一般]** をクリックしてから、**[Interactive SQL を .SQL ファイルとプランファイルのデフォルトエディタにする]** をクリックします。

必要な権限は、実行する文によって異なります。

内容と備考

[スクリプトの実行] メニュー項目の機能は、READ 文と同じです。

♦ タスク

1. Interactive SQL で、**[ファイル]** » **[スクリプトの実行]** をクリックします。
2. ファイルを検索し、**[開く]** をクリックします。

結果

指定されたファイルの内容がすぐに実行されます。実行の進行状況を示す **[ステータス]** ウィンドウが表示されます。

参照

- 「READ 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「Interactive SQL のカスタマイズ」『SQL Anywhere サーバ データベース管理』
- 「SQL スクリプトファイル、SQL 文、および接続をお気に入りに追加する」『SQL Anywhere サーバ データベース管理』

Interactive SQL の READ 文を使用した SQL スクリプトファイルの実行

Interactive SQL の READ 文を使用することによっても、**[SQL 文]** ウィンドウ枠にロードすることなく、SQL スクリプトファイルを実行できます。

前提条件

必要な権限は、実行する文によって異なります。

♦ タスク

- **[SQL 文]** ウィンドウ枠で、次の例のような文を実行します。

  ```
  READ 'C:¥¥LocalTemp¥¥filename.sql';
  ```

  この文の *C:¥LocalTemp¥filename.sql* には、ファイルのパス、名前、拡張子を指定します。パスにスペースが含まれている場合にのみ、一重引用符 (例文参照) が必要です。一重引用符を使用する場合、円記号は 2 つ重ねることによりエスケープされ、'¥n' または '¥x' などのエスケープシーケンスの変換が回避されます。

結果

SQL スクリプトファイルが実行されます。

参照

- 「READ 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「Interactive SQL のカスタマイズ」『SQL Anywhere サーバ データベース管理』
- 「SQL スクリプトファイル、SQL 文、および接続をお気に入りに追加する」『SQL Anywhere サーバ データベース管理』

バッチモードでの SQL スクリプトファイルの実行 (コマンドラインの場合)

Interactive SQL のコマンドライン引数として SQL スクリプトファイルを指定できます。

前提条件

必要な権限は、実行する文によって異なります。

♦ タスク

- dbisql ユーティリティを実行し、コマンドライン引数として SQL スクリプトファイルを指定します。

結果

SQL スクリプトファイルが実行されます。

例

次のコマンドは、SQL Anywhere サンプルデータベースに対して SQL スクリプトファイル *myscript.sql* を実行します。

```
dbisql -c "DSN=SQL Anywhere 16 Demo" myscript.sql
```

参照

- 「Interactive SQL ユーティリティ (dbisql)」『SQL Anywhere サーバ データベース管理』

ファイルから [SQL 文] ウィンドウ枠への SQL スクリプトのロード

Interactive SQL では、SQL スクリプトファイルを **[SQL 文]** ウィンドウ枠にロードして、そこから直接実行できます。

前提条件

Interactive SQL が *.sql* ファイルのデフォルトエディタとして設定されていることを確認します。

Interactive SQL で、**[ツール]** » **[オプション]** » **[一般]** をクリックしてから、**[Interactive SQL を .SQL ファイルとプランファイルのデフォルトエディタにする]** をクリックします。

必要な権限は、実行する文によって異なります。

♦ タスク

1. **[ファイル]** » **[開く]** をクリックします。
2. ファイルを検索し、**[開く]** をクリックします。

結果

文が **[SQL 文]** ウィンドウ枠に表示され、読み込み、編集、実行を行うことができます。

ファイルへのデータベース出力の書き込み

Interactive SQL では、文の実行結果セットデータは、次の文が実行されると **[結果]** ウィンドウ枠の **[結果]** タブから消えます。データの記録を残すには、個々の文を別ファイルに出力して保存します。

前提条件

必要な権限は、実行する文によって異なります。

♦ タスク

- たとえば、2 つの SELECT 文 *statement1* と *statement2* がある場合、次のようにしてそれぞれの実行結果を *file1* と *file2* に出力します。

```
statement1; OUTPUT TO file1;statement2; OUTPUT TO file2;
```

結果

各 SQL 文の出力が個別のファイルに保存されます。

例

次の文は、クエリの結果を *C:¥LocalTemp* ディレクトリ内の *Employees.csv* という名前のファイルに保存します。

```
SELECT * FROM Employees;
OUTPUT TO 'C:¥¥LocalTemp¥¥Employees.csv';
```

参照

- 「SELECT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ SQL リファレンス』
- 「UNLOAD 文を使用したデータのエクスポートに関するヒント」736 ページ

Adaptive Server Enterprise の互換性

BCP FORMAT 句を使用して、SQL Anywhere と Adaptive Server Enterprise 間でファイルのインポートとエクスポートを実行できます。Adaptive Server Enterprise で使用するために SQL Anywhere から BLOB データをエクスポートしている場合は、UNLOAD TABLE 文で BCP フォーマット句を使用します。

データを SQL Anywhere にインポートできるように、BCP の out コマンドを使用して Adaptive Server Enterprise からファイルをエクスポートする場合、データはテキスト／ASCII フォーマットとし、カンマ区切りにしてください。テキスト／ASCII フォーマットでデータをエクスポートするには、BCP out コマンドの -c オプションを使用できます。-t オプションを使用すると、区切り文字 (デフォルトはタブ) を変更できます。区切り文字を変更しない場合は、SQL Anywhere データベースにデータをインポートするときに、LOAD TABLE 文に **DELIMITED BY '¥x09'** を指定してください。

参照

- 「LOAD TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「UNLOAD 文」『SQL Anywhere サーバ SQL リファレンス』

 Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0

リモートデータアクセス

SQL Anywhere のリモートデータアクセス機能によって、他のデータソースのデータにアクセスできます。この機能を使用して、データを SQL Anywhere データベースにマイグレートできます。また、データベース間でデータを問い合わせできます。

リモートデータアクセスを使用して、次のことを実行できます。

- insert-select を使用してデータをあるロケーションから別のロケーションに移動するのに SQL Anywhere を使用する。
- Sybase ASE、Oracle Database、IBM DB2 などのリレーショナルデータベースのデータにアクセスする。
- Excel スプレッドシート、Microsoft Access データベース、FoxPro、テキストファイルのデータにアクセスする。
- ODBC インタフェースをサポートするデータソースにアクセスする。
- ローカルデータとリモートデータ間でジョインを実行する。ただしパフォーマンスは、すべてのデータが単一の SQL Anywhere データベース内にある場合よりかなり低速になります。
- 別々の SQL Anywhere データベースのテーブル間でジョインを実行する。パフォーマンスの制限は、他のリモートデータソースの場合と同様です。
- 通常は SQL Anywhere の機能を持たないデータソースに対して、SQL Anywhere の機能を使用する。たとえば、Oracle データベースに格納されているデータに対して Java の機能を使用したり、スプレッドシートでサブクエリをしたりできます。データを取り出してから操作することによって、リモートデータソースではサポートされていない機能を SQL Anywhere が補います。
- FORWARD TO 文を使用して、リモートサーバに直接アクセスする。
- 他のサーバへのリモートプロシージャコールを実行する。

SQL Anywhere によって、次に示す外部データソースへのアクセスが可能になります。

- SQL Anywhere
- Adaptive Server Enterprise
- Advantage Database Server
- IBM DB2
- Microsoft Access
- Microsoft SQL Server
- Oracle MySQL
- Oracle Database
- SAP HANA
- SAP Sybase IQ
- Ultra Light
- その他の ODBC データソース

注意
Mac OS X 上で実行している Ultra Light データベースには、リモートサーバを作成できません。

利用可能なプラットフォームについては、http://www.sybase.com/detail?id=1002288 を参照してください。

リモートテーブルのマッピング

クライアントアプリケーション側から見ると、接続している SQL Anywhere 内にすべてのテーブルがあるかのように見えます。リモートテーブルに関わるクエリが実行されると、内部では対象となるデータソースを割り出し、外部にあるそのデータソースにアクセスしてデータを取り出します。

リモートテーブルのデータにアクセスするには、次の設定を行う必要があります。

1. リモートオブジェクトが置かれるリモートサーバを定義します。これにはサーバのクラスとリモートサーバの場所が含まれます。これを行うには CREATE SERVER 文を使用します。

2. リモートサーバ上のデータベースへのアクセスに必要なクレデンシャルが、接続されているデータベースへのアクセスに必要なクレデンシャルと異なる場合は、リモートサーバのユーザログイン情報を定義します。これを行うには CREATE EXTERNLOGIN 文を使用します。

3. プロキシテーブルの定義を作成します。これによってリモートテーブルに対するローカルプロキシテーブルのマッピングを指定します。リモートテーブルが置かれているサーバ、リモートテーブルのデータベース名、所有者名、テーブル名、カラム名を指定します。これを行うには CREATE EXISTING TABLE 文を使用します。また、CREATE TABLE 文を使用しても、リモートサーバで新しいテーブルを作成できます。

リモートサーバの定義、外部ログイン、プロキシテーブルのマッピングを管理するには、Sybase Central を使用するか、Interactive SQL などのツールを使用して SQL 文を実行します。

警告
Microsoft Access、Microsoft SQL Server、Sybase Adaptive Server Enterprise などの一部のリモートサーバでは、COMMIT や ROLLBACK を実行してもカーソルは保存されません。このようなリモートサーバでは、Sybase Central の **[データ]** タブを使用してプロキシテーブルの内容を表示したり変更することはできません。ただし、オートコミット機能が無効 (Interactive SQL のデフォルトの動作) になっている場合は、Interactive SQL を使用してプロキシテーブルのデータを表示および編集できます。Oracle Database、IBM DB2、SQL Anywhere などのその他の RDBMS では、この制限はありません。

リモートサーバ

リモートオブジェクトをローカルのプロキシテーブルにマッピングするには、リモートオブジェクトが置かれるリモートサーバを、あらかじめ定義しておきます。リモートサーバを定義するときは、サーバのクラスを選択します。

「サーバクラス」は、リモートサーバとの対話に使用するアクセス方法を指定します。異なるタイプのリモートサーバには、異なるアクセス方法が必要です。SQL Anywhere は、サーバ機能に関する詳細情報をそのサーバクラスから得ます。SQL Anywhere はこれらの情報に基づいて、リモートサーバとのアクセスを調整します。

サーバのクラスは次のとおりです。

- **SAODBC** SQL Anywhere
- **ULODBC** UltraLite

> **注意**
> Mac OS X 上で実行している Ultra Light データベースには、リモートサーバを作成できません。

- **ADSODBC** Advantage Database Server
- **ASEODBC** Sybase Adaptive Server Enterprise (バージョン 10 以降)
- **DB2ODBC** IBM DB2
- **HANAODBC** SAP HANA
- **IQODBC** SAP Sybase IQ
- **MSACCESSODBC** Microsoft Access
- **MSSODBC** Microsoft SQL Server
- **MYSQLODBC** Oracle MySQL
- **ODBC** その他の ODBC データソース
- **ORAODBC** Oracle Database サーバ (バージョン 8.0 以降)

> **注意**
> リモートデータアクセスを使用する際に、Unicode をサポートしていない ODBC ドライバを使用すると、その ODBC ドライバから受け取るデータに対して、文字セット変換が実行されません。

リモートサーバを定義すると、ISYSSERVER システムテーブルにエントリが追加されます。

参照

- 「リモートデータアクセスのサーバクラス」808 ページ

リモートサーバの作成 (SQL の場合)

CREATE SERVER 文を使用して、リモートサーバ定義を設定します。

前提条件

SERVER OPERATOR システム権限が必要です。

内容と備考

各リモートサーバにアクセスするには ODBC ドライバを使用します。データベースごとにリモートサーバの定義が必要です。

データソースの識別には接続文字列を使用します。UNIX プラットフォームでは、ODBC ドライバも接続文字列で参照される必要があります。

♦ リモートサーバの作成

- CREATE SERVER 文を使用して、リモートサーバにリンクするリモートデータアクセスサーバを定義します。

  たとえば、次の文はリモートサーバ RemoteASE を定義します。SQL Anywhere データベースサーバは、USING 句で指定された ODBC 接続文字列を使用して Adaptive Server Enterprise データベースサーバに接続します。

  ```
  CREATE SERVER RemoteASE
  CLASS 'ASEODBC'
  USING 'DRIVER=SYBASE ASE ODBC
  Driver;Server=TestASE;Port=5000;Database=testdb;UID=username;PWD=password';
  ```

  次の説明は、CREATE SERVER 文の各部の分析を示します。

  - **SERVER**　この句を使用して、リモートサーバに名前を付けます。たとえば、RemoteASE はリモートサーバの名前です。
  - **CLASS**　この句を使用して、SQL Anywhere データベースサーバがリモートサーバと通信する方法を示します。この例では、ASEODBC はリモートサーバが Adaptive Server Enterprise (ASE) であり、ASE ODBC ドライバを使用して接続が確立されることを示します。
  - **USING**　この句は、リモートサーバの ODBC 接続文字列を指定します。この例では ASE ODBC ドライバが指定されています。

結果

CREATE SERVER 文は、ISYSSERVER システムテーブルにエントリを作成します。

次の手順

必要な場合は、外部ログイン情報を作成します。

例

次の文は、リモートサーバ RemoteSA を定義します。SQL Anywhere データベースサーバは、USING 句で指定された ODBC データソース名 (DSN) を使用して、SQL Anywhere データベースサーバに接続します。

```
CREATE SERVER RemoteSA
CLASS 'SAODBC'
USING 'SQL Anywhere 16 CustDB';
```

次の文は、リモートサーバ RemoteLinuxSA を定義します。SQL Anywhere データベースサーバは、USING 句で指定された ODBC データソース名 (DSN) を使用して、SQL Anywhere データベースサーバに接続します。UNIX プラットフォームでは、ODBC ドライバは、接続文字列で指定する必要があります。

```
CREATE SERVER RemoteLinuxSA
CLASS 'SAODBC'
USING 'DRIVER=SQL Anywhere 16;DSN=my_sa_dsn';
```

次の文は、リモートサーバ RemoteLinuxASE を定義します。SQL Anywhere データベースサーバは、USING 句で指定された ODBC データソース名 (DSN) を使用して、Adaptive Server Enterprise (ASE) データベースサーバに接続します。UNIX プラットフォームでは、ODBC ドライバは、接続文字列で指定する必要があります。

```
CREATE SERVER RemoteLinuxASE
CLASS 'ASEODBC'
USING '/opt/sybase/ase_odbc_1500/DataAccess/ODBC/lib/libsybdrvodb.so;DSN=my_ase_dsn';
```

次の文は、リモートサーバ RemoteAccessDB を定義します。SQL Anywhere データベースサーバは、USING 句で指定された ODBC DSN MyAccessDataSource を使用して、Microsoft Access データベースに接続します。

```
CREATE SERVER RemoteAccessDB
CLASS 'MSACCESSODBC'
USING 'MyAccessDataSource';
```

参照

- 「外部ログインの作成 (Sybase Central の場合)」789 ページ
- 「プロキシテーブルの作成 (SQL の場合)」793 ページ
- 「CREATE SERVER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE EXTERNLOGIN 文」『SQL Anywhere サーバ SQL リファレンス』

リモートサーバの作成 (Sybase Central の場合)

管理者は、Sybase Central を使用してリモートサーバ定義を作成できます。

前提条件

MANAGE ANY USER と SERVER OPERATOR のシステム権限が必要です。

♦ リモートサーバの作成

1. SQL Anywhere 16 プラグインを使用して、ホストデータベースに接続します。
2. 左ウィンドウ枠で、**[リモートサーバ]** をダブルクリックします。
3. **[ファイル]** » **[新規]** » **[リモートサーバ]** をクリックします。

4. **[新しいリモートサーバの名前を指定してください。]** フィールドにリモートサーバの名前を入力し、**[次へ]** をクリックします。

5. リモートサーバのタイプを選択して、**[次へ]**をクリックします。

6. 接続タイプを選択し、**[接続情報を指定してください。]** フィールドに次に示す接続情報を入力します。

   ●ODBC ベースの接続では、データソース名を指定するか、ODBC Driver パラメータとその他の接続パラメータを指定します。

   ●JDBC ベースの接続では、*computer-name*:*port-number* の形式で URL を指定します。

   データアクセスメソッド (JDBC か ODBC) は、SQL Anywhere がリモートデータベースへのアクセスに使用するためのものです。Sybase Central が Adaptive Server Anywhere に接続する方法をこれで決めているわけではありません。

   JDBC ベースのリモートサーバアクセスは、現在のリリースではサポートされていません。

7. **[次へ]** をクリックします。

8. リモートサーバを読み込み専用にするかどうかを指定し、**[次へ]** をクリックします。

9. **[現在のユーザの外部ログインを作成する]** をクリックし、必須フィールドの入力を完成させます。

   デフォルトでは、SQL Anywhere は、現在のユーザに代わってリモートサーバに接続する場合に、常にそのユーザのユーザ ID とパスワードを使用します。ただし、リモートサーバに現在のユーザと同じユーザ ID とパスワードで定義されたユーザがいない場合は、外部ログインを作成する必要があります。外部ログインは、現在のユーザ用に代替ログイン名とパスワードを割り当ててリモートサーバに接続できるようにします。

10. **[接続テスト]** をクリックしてリモートサーバ接続をテストします。

11. **[完了]** をクリックします。

結果

リモートサーバは、指定された定義で作成されます。

次の手順

必要な場合は、外部ログイン情報を作成します。

参照

- 「外部ログインの作成 (Sybase Central の場合)」789 ページ
- 「プロキシテーブルの作成 (Sybase Central の場合)」792 ページ
- 「CREATE SERVER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE EXTERNLOGIN 文」『SQL Anywhere サーバ SQL リファレンス』

リモートサーバの削除 (SQL の場合)

管理者は DROP SERVER 文を使用してリモートサーバを削除できます。

前提条件

SERVER OPERATOR システム権限が必要です。

内容と備考

リモートサーバを削除する前に、そのリモートサーバに定義されているプロキシテーブルをすべて削除する必要があります。次のクエリを使用すると、リモートサーバ *server-name* に定義されているプロキシサーバを特定できます。

```
SELECT st.table_name, sp.remote_location, sp.existing_obj
   FROM sysproxytab sp
   JOIN sysserver ss ON ss.srvid = sp.srvid
   JOIN systab st ON sp.table_object_id = st.object_id
   WHERE ss.srvname = 'server-name';
```

♦ リモートサーバの削除

1. ホストデータベースに接続します。
2. DROP SERVER 文を実行します。

```
DROP SERVER server-name;
```

結果

リモートサーバが削除されます。

例

次の文は RemoteASE という名前のリモートサーバを削除します。

```
DROP SERVER RemoteASE;
```

参照

- 「リモートサーバの削除 (Sybase Central の場合)」 777 ページ
- 「DROP SERVER 文」『SQL Anywhere サーバ SQL リファレンス』

リモートサーバの削除 (Sybase Central の場合)

管理者は、Sybase Central でリモートサーバを削除できます。

前提条件

SERVER OPERATOR システム権限が必要です。

内容と備考

リモートサーバを削除する前に、そのリモートサーバに定義されているプロキシテーブルをすべて削除する必要があります。Sybase Central は、リモートサーバに定義されているプロキシテーブルを自動的に特定して、それらを最初に削除します。

♦ リモートサーバの削除

1. SQL Anywhere 16 プラグインを使用して、ホストデータベースに接続します。
2. 左ウィンドウ枠で、**[リモートサーバ]** をダブルクリックします。
3. リモートサーバを選択し、**[編集]** » **[削除]** をクリックします。

結果

リモートサーバが削除されます。

参照

- 「リモートサーバの削除 (SQL の場合)」 777 ページ
- 「DROP SERVER 文」『SQL Anywhere サーバ SQL リファレンス』

リモートサーバの変更 (SQL の場合)

管理者は、Interactive SQL でリモートサーバのプロパティを変更できます。

前提条件

SERVER OPERATOR システム権限が必要です。

内容と備考

ALTER SERVER 文は、サーバの既知の機能の有効化または無効化にも使用できます。

♦ リモートサーバのプロパティを変更する

リモートサーバの設定の変更は、次回リモートサーバに接続する際に有効になります。

1. ホストデータベースに接続します。
2. ALTER SERVER 文を実行します。

結果

リモートサーバのプロパティが変更されます。

例

次の文は、RemoteASE という名前のサーバのサーバクラスを ASEODBC に変更します。

```
ALTER SERVER RemoteASE
CLASS 'ASEODBC';
```

参照

- 「ALTER SERVER 文」『SQL Anywhere サーバ SQL リファレンス』

リモートサーバの変更 (Sybase Central の場合)

Sybase Central で、リモートサーバのプロパティを変更できます。

前提条件

SERVER OPERATOR システム権限が必要です。

内容と備考

リモートサーバの設定の変更は、次回リモートサーバに接続する際に有効になります。

♦ リモートサーバのプロパティを変更する

1. SQL Anywhere 16 プラグインを使用して、ホストデータベースに接続します。
2. 左ウィンドウ枠で、**[リモートサーバ]** をダブルクリックします。
3. リモートサーバを選択し、**[ファイル]** » **[プロパティ]** をクリックします。
4. リモートサーバの設定を変更し、**[OK]**をクリックします。

結果

リモートサーバのプロパティが変更されます。

参照

- 「ALTER SERVER 文」『SQL Anywhere サーバ SQL リファレンス』

リモートサーバ上のテーブルのリスト (SQL の場合)

システムプロシージャを使用すると、リモートサーバ上の全テーブルの制限されたリスト、または包括的なリストを表示できます。

前提条件

なし。

♦ リモートサーバ上のテーブルのリスト

- sp_remote_tables システムプロシージャを呼び出し、リモートサーバ上のテーブルのリストを返します。

@table_name または *@table_owner* を指定すると、テーブルのリストは一致するテーブルにのみ制限されます。

結果

すべてのテーブルのリスト、またはテーブルの制限されたリストが返されます。

例

GROUPO が所有する RemoteSA という名前のリモートサーバで、データベースのすべてのテーブルのリストを取得するには、次の文を実行します。

```
CALL sp_remote_tables('RemoteSA', null, 'GROUPO');
```

RemoteASE という名前の Adaptive Server Enterprise サーバで、運用データベースのすべてのテーブルのリストを取得するには、次の文を実行します。

```
CALL sp_remote_tables('RemoteASE', null, 'Fred', 'Production');
```

Excel という名前のリモートサーバから使用可能な、すべての Microsoft Excel ワークシートのリストを取得するには、次の文を実行します。

```
CALL sp_remote_tables('Excel');
```

参照

- 「sp_remote_tables システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

リモートサーバの機能

sp_servercaps システムプロシージャは、リモートサーバの機能に関する情報を表示します。SQL Anywhere はこの機能の情報を使用して、リモートサーバに渡すことができる SQL 文の量を判断します。

また、SYSCAPABILITY と SYSCAPABILITYNAME のシステムビューを問い合わせると、リモートサーバの機能情報も参照できます。これらのシステムビューは、SQL Anywhere が最初にリモートサーバに接続するまでは空の状態です。

sp_servercaps システムプロシージャを使用する場合は、*server-name* には CREATE SERVER 文で使用した *server-name* と同じ名前を指定してください。

次のように、sp_servercaps ストアドプロシージャを実行します。

```
CALL sp_servercaps('server-name');
```

参照

- 「sp_servercaps システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSCAPABILITY システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSCAPABILITYNAME システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE SERVER 文」『SQL Anywhere サーバ SQL リファレンス』

ディレクトリアクセスサーバ

「ディレクトリアクセスサーバ」は、データベースサーバを実行しているコンピュータのローカルファイル構造にアクセスするためのリモートサーバです。ディレクトリアクセスサーバに接続したら、プロキシテーブルを使用して、そのコンピュータ上のサブフォルダにアクセスします。データベースユーザがディレクトリアクセスサーバを使用するには、外部ログインが必要です。

ディレクトリアクセスサーバの作成後に変更することはできません。ディレクトリアクセスサーバの変更が必要な場合は、削除してから別の設定で再作成してください。まず、ディレクトリアクセスサーバを参照しているプロキシテーブルをすべて削除し、ディレクトリアクセスサーバを再作成してから、プロキシテーブルを再作成します。

プロキシテーブルのフォーマットを以下に示します。

- **permissions VARCHAR(10)**　POSIX スタイルのパーミッション文字列 (drwxrwxrwx など)。
- **size BIGINT**　ファイルのサイズ (バイト単位)。
- **access_date_time TIMESTAMP**　ファイルが最後にアクセスされた日付と時刻 (たとえば、2010-02-08 11:00:24.000)。
- **modified_date_time TIMESTAMP**　ファイルが最後に変更された日付と時刻 (たとえば、2009-07-28 10:50:11.000)。
- **create_date_time TIMESTAMP**　ファイルが作成された日付と時刻 (たとえば、2008-12-18 10:32:26.000)。
- **owner VARCHAR(20)**　ファイル作成者のユーザ ID (たとえば、Linux の root)。Windows の場合、この値は常に "0" です。
- **file_name VARCHAR(260)**　相対パスを含むファイル名 (たとえば、bin¥perl.exe)。
- **contents LONG BINARY**　このカラムが結果セットで明示的に参照される場合、ファイルの内容。

ディレクトリアクセスサーバの作成 (Sybase Central の場合)

管理者は、Sybase Central の **[ディレクトリ アクセス サーバの作成ウィザード]** を使用して、ディレクトリアクセスサーバを作成できます。

前提条件

MANAGE ANY USER と SERVER OPERATOR のシステム権限が必要です。

ユーザ本人が所有するプロキシテーブルを作成するには、CREATE PROXY TABLE システム権限が必要です。他のユーザが所有するプロキシテーブルを作成するには、CREATE ANY TABLE または CREATE ANY OBJECT のシステム権限が必要です。

内容と備考

ディレクトリアクセスサーバを作成するときは、アクセスできるサブフォルダ数を制御したり、ディレクトリアクセスサーバで既存のファイルを変更できるかどうかを制御できます。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してホストデータベースに接続します。
2. 左ウィンドウ枠で **[ディレクトリアクセスサーバ]** を右クリックし、**[新規]** » **[ディレクトリアクセスサーバ]** をクリックします。
3. **ディレクトリアクセスサーバの作成ウィザード**の指示に従います。
4. Sybase Central の左ウィンドウ枠で **[テーブル]** を右クリックし、**[新規]** » **[プロキシテーブル]** をクリックします。
5. **プロキシテーブル作成ウィザード**の指示に従います。

結果

ディレクトリアクセスサーバは作成され、設定されます。

参照

- 「CREATE SERVER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE EXTERNLOGIN 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE EXISTING TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

ディレクトリアクセスサーバの作成 (SQL の場合)

管理者は、Interactive SQL で CREATE SERVER 文を使用して、ディレクトリアクセスサーバを作成できます。

前提条件

SERVER OPERATOR および MANAGE ANY USER システム権限が必要です。

ユーザ本人が所有するプロキシテーブルを作成するには、CREATE PROXY TABLE システム権限が必要です。他のユーザが所有するプロキシテーブルを作成するには、CREATE ANY TABLE または CREATE ANY OBJECT のシステム権限が必要です。

♦ タスク

1. CREATE SERVER 文を使用して、リモートサーバを作成します。

```
CREATE SERVER my_dir_server
CLASS 'DIRECTORY'
USING 'ROOT=c:¥Program Files;SUBDIRS=3';
```

2. CREATE EXTERNLOGIN 文を使用して、外部ログインを作成します。

   ```
   CREATE EXTERNLOGIN DBA TO my_dir_server;
   ```

3. CREATE EXISTING TABLE 文を使用して、ディレクトリのプロキシテーブルを作成します。

   ```
   CREATE EXISTING TABLE my_program_files AT 'my_dir_server;;;.';
   ```

   この例では、my_program_files がプロキシテーブルの名前、my_dir_server がディレクトリアクセスサーバの名前です。

4. プロキシテーブルのローを表示します。

   ```
   SELECT * FROM my_program_files ORDER BY file_name;
   ```

5. sp_remote_tables システムプロシージャを使用すると、データベースサーバを実行しているコンピュータ上の *c:¥mydir* にあるすべてのサブフォルダを表示できます。

   ```
   CALL sp_remote_tables( 'my_dir_server' );
   ```

結果

ディレクトリアクセスサーバは作成され、設定されます。

参照

- 「CREATE SERVER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE EXTERNLOGIN 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE EXISTING TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

例：動的なディレクトリアクセスサーバ (SQL の場合)

この例では、管理者は CREATE SERVER 文とともにディレクトリアクセスサーバのルートとサブディレクトリのレベルを表す変数を使用して、動的なディレクトリアクセスサーバを作成できます。

前提条件

SERVER OPERATOR システム権限が必要です。

内容と備考

あなたは DBA であり、あるときは server1 という名前のデータベースサーバが起動されるコンピュータ A のデータベース、またはあるときは server2 という名前のデータベースサーバが起動されるコンピュータ B のデータベースを利用しているとします。ここで、コンピュータ A のローカルドライブ *c:¥temp* を参照するディレクトリアクセスサーバと、コンピュータ B のネットワークサーバドライブ *d:¥temp* をセットアップしたいとします。さらに、プロキシテーブルをセットアップして、すべてのユーザがそこから自分のプライベートディレクトリのリストを取得できるようにしたいとします。次の手順に従って、CREATE SERVER 文の USING 句と CREATE EXISTING TABLE 文の AT 句の変数を使用することによって、単一のディレクトリアクセスサーバと単一のプロキシテーブルを作成すれば、そのような要求を満たすことができます。

♦ タスク

1. この例では、接続しているサーバの名前が server1 であるとして、次のディレクトリが存在するとします。

```
c:¥temp¥dba
c:¥temp¥updater
c:¥temp¥browser
```

ディレクトリアクセスサーバのルートとサブディレクトリのレベルを表す変数を使用して、ディレクトリアクセスサーバを作成します。

```
CREATE SERVER dir
CLASS 'DIRECTORY'
USING 'root={@directory};subdirs={@subdirs}';
```

2. ディレクトリアクセスサーバの使用を許可するそれぞれのユーザに明示的な外部ログインを作成します。

```
CREATE EXTERNLOGIN "DBA" TO dir;
CREATE EXTERNLOGIN "UPDATER" TO dir;
CREATE EXTERNLOGIN "BROWSER" TO dir;
```

3. ディレクトリアクセスサーバとその関連するプロキシテーブルを動的に設定するための変数を作成します。

```
CREATE VARIABLE @directory LONG VARCHAR;
SET @directory = 'c:¥¥temp';

CREATE VARIABLE @subdirs VARCHAR(10);
SET @subdirs = '7';

CREATE VARIABLE @curuser VARCHAR(128);
SET @curuser = 'updater';

CREATE VARIABLE @server VARCHAR(128);
SET @server = 'dir';
```

4. ディレクトリアクセスサーバ @server の *@directory¥@curuser* を参照するプロキシテーブルを作成します。

```
CREATE EXISTING TABLE dbo.userdir AT '{@server};;;{@curuser}';
```

5. 変数はもう必要ではないため、次の文を実行して削除します。

```
DROP VARIABLE @server;
DROP VARIABLE @curuser;
DROP VARIABLE @subdirs;
DROP VARIABLE @directory;
```

6. ユーザが自分のユーザディレクトリの内容を表示するためのプロシージャを作成します。

```
CREATE OR REPLACE PROCEDURE dbo.listmydir()
SQL SECURITY INVOKER
BEGIN
   DECLARE @directory LONG VARCHAR;
   DECLARE @subdirs VARCHAR(10);
   DECLARE @server VARCHAR(128);
   DECLARE @curuser VARCHAR(128);
```

```
    -- for this example we always use the "dir" remote directory access server
    SET @server = 'dir';

    -- the root directory is based on the name of the server the user is connected to
    SET @directory = if property('name') = 'server1' then 'c:¥¥temp'
                 else 'd:¥¥temp' endif;

    -- the subdir limit is based on the connected user
    SET @curuser = user_name();

    -- all users get a subdir limit of 7 except "browser" who gets a limit of 1
    SET @subdirs = convert( varchar(10), if @curuser = 'browser' then 1 else 7 endif);

    -- with all the variables set above, the proxy table dbo.userdir
    -- now points to @directory¥@curuser and has a subdir limit of @subdirs
    SELECT * FROM dbo.userdir;

    DROP REMOTE CONNECTION TO dir CLOSE CURRENT;
END;
```

この手順の最後のステップでは、リモート接続を終了して、ユーザが (sp_remote_tables システムプロシージャなどを使用して) ディレクトリアクセスサーバのリモートテーブルをリストできないようにします。

7. ストアドプロシージャの一般的な用途に必要な権限を設定します。

```
GRANT SELECT ON dbo.userdir TO PUBLIC;
GRANT EXECUTE ON dbo.listmydir TO PUBLIC;
```

8. データベースサーバから切断して、ユーザ UPDATER (パスワード 'update') またはユーザ BROWSER (パスワード 'browse') として再接続します。次のクエリを実行します。

```
CALL dbo.listmydir()
```

結果

動的なディレクトリアクセスサーバが作成され、設定されます。

参照

- 「CREATE SERVER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE EXTERNLOGIN 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE EXISTING TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DROP REMOTE CONNECTION 文」『SQL Anywhere サーバ SQL リファレンス』

ディレクトリアクセスプロキシテーブルでのクエリ

パフォーマンスを向上させるには、テーブルスキャンを発生させるクエリを使用するときに内容カラムを選択しないようにします。可能であれば、ファイル名を使用してディレクトリアクセスプロキシテーブルの内容を取得します。ファイル名を述部として使用すると、指定したファイルのみがディレクトリアクセスサーバによって読み込まれるため、パフォーマンスは向上します。ファイル名が不明の場合は、最初にファイルのリストを取得するクエリを実行してから、リスト内の各ファイルのクエリを発行してその内容を取得します。

例 1

次に示すクエリの実行は遅い場合があります (ディレクトリ内のファイルの数とサイズによって異なります)。述部と一致するものを見つけるために、ディレクトリ内のすべてのファイルの内容をディレクトリアクセスサーバで読み込む必要があるためです。

```
SELECT contents FROM DirAccessProxyTable WHERE file_name LIKE 'something%';
```

例 2

次に示すクエリは、ディレクトリスキャンを発生させないで、単一のファイルの内容を返します。

```
SELECT contents FROM DirAccessProxyTable WHERE file_name = 'something';
```

例 3

次に示すクエリの実行も遅い場合があります (ディレクトリ内のファイルの数とサイズによって異なります)。論理和 (OR) があるために、テーブルスキャンをディレクトリアクセスサーバで実行する必要があるためです。

```
SELECT contents FROM DirAccessProxyTable WHERE file_name = 'something' OR size = 10;
```

例 4

ファイル名をリテラル定数としてクエリ内で指定する代わりに、ファイル名の値を変数に入れて、その変数をクエリで使用できます。

```
DECLARE @filename LONG VARCHAR;
SET @filename = 'something';
SELECT contents FROM DirAccessProxyTable WHERE file_name = @filename;
```

デリミタの一貫性

ディレクトリアクセスプロキシテーブルを問い合わせるときは、パス名のデリミタの使用法が一貫している必要があります。プラットフォームのネイティブデリミタを使用することをおすすめします。Windows では 「¥」 を使用し、UNIX では 「/」 を使用します。Windows では 「/」 もサーバによってデリミタとして認識されますが、リモートデータアクセスでは、ファイル名は常に一貫したデリミタを使用して返されます。そのため、デリミタが一貫していないクエリではローは返されません。

例

次のクエリではローは返されません。

```
SELECT contents FROM DirAccessProxyTable WHERE filename = 'some/dir¥thing';
```

ディレクトリアクセスサーバのプロキシテーブルの削除 (Sybase Central の場合)

管理者は Sybase Central を使用して、ディレクトリアクセスサーバに関連づけられているプロキシテーブルを削除できます。

前提条件

所有者であるか、または DROP ANY TABLE と DROP ANY OBJECT のいずれかのシステム権限を持っている必要があります。

内容と備考

ディレクトリアクセスサーバを削除できるようにする前に、そのディレクトリアクセスサーバに関連付けられているすべてのプロキシテーブルを削除する必要があります。

♦ プロキシテーブルの削除

1. SQL Anywhere 16 プラグインを使用して、ホストデータベースに接続します。
2. 左ウィンドウ枠で、**[ディレクトリアクセスサーバ]** をダブルクリックし、削除したいプロキシテーブルを保持しているディレクトリアクセスサーバをクリックします。
3. 右ウィンドウ枠で、**[プロキシテーブル]** タブをクリックします。
4. プロキシテーブルを選択し、**[編集]** » **[削除]** をクリックします。
5. **[はい]** をクリックします。

結果

プロキシテーブルが削除されます。

次の手順

ディレクトリアクセスサーバに関連付けられているすべてのプロキシテーブルが削除されたら、そのディレクトリアクセスサーバを削除できます。

参照

- 「ディレクトリアクセスサーバの削除 (Sybase Central の場合)」787 ページ
- 「DROP SERVER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DROP TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

ディレクトリアクセスサーバの削除 (Sybase Central の場合)

管理者は、Sybase Central を使用してディレクトリアクセスサーバを削除できます。

前提条件

SERVER OPERATOR システム権限が必要です。

ディレクトリアクセスサーバを削除する前に、そのディレクトリアクセスサーバに定義されているプロキシテーブルをすべて削除する必要があります。次のクエリを使用すると、ディレクトリアクセスサーバ *server-name* に定義されているプロキシサーバを特定できます。

```
SELECT st.table_name, sp.remote_location, sp.existing_obj
   FROM sysproxytab sp
   JOIN sysserver ss ON ss.srvid = sp.srvid
   JOIN systab st ON sp.table_object_id = st.object_id
   WHERE ss.srvname = 'server-name';
```

内容と備考

既存のディレクトリアクセスサーバを変更することはできません。DROP SERVER 文を使用して既存のディレクトリアクセスサーバを削除し、新しいサーバを作成してください。

♦ ディレクトリアクセスサーバの削除

1. SQL Anywhere 16 プラグインを使用して、ホストデータベースに接続します。
2. 左ウィンドウ枠で、**[ディレクトリアクセスサーバ]** をダブルクリックします。
3. ディレクトリアクセスサーバを選択し、**[編集]** » **[削除]** をクリックします。

結果

ディレクトリアクセスサーバが削除されます。

参照

- 「プロキシテーブルの削除 (Sybase Central の場合)」794 ページ

ディレクトリアクセスサーバの削除 (SQL の場合)

管理者は、Interactive SQL を使用してディレクトリアクセスサーバを削除できます。

前提条件

SERVER OPERATOR システム権限が必要です。

ディレクトリアクセスサーバを削除する前に、そのディレクトリアクセスサーバに定義されているプロキシテーブルをすべて削除する必要があります。次のクエリを使用すると、ディレクトリアクセスサーバ *server-name* に定義されているプロキシサーバを特定できます。

```
SELECT st.table_name, sp.remote_location, sp.existing_obj
   FROM sysproxytab sp
   JOIN sysserver ss ON ss.srvid = sp.srvid
   JOIN systab st ON sp.table_object_id = st.object_id
   WHERE ss.srvname = 'server-name';
```

内容と備考

既存のディレクトリアクセスサーバを変更することはできません。DROP SERVER 文を使用して既存のディレクトリアクセスサーバを削除し、新しいサーバを作成してください。

♦ ディレクトリアクセスサーバの削除

1. ホストデータベースに接続します。
2. ディレクトリアクセスサーバに関連付けられている各プロキシテーブルに対して、DROP TABLE 文を実行します。

   ```
   DROP TABLE my_program_files;
   ```

3. そのディレクトリアクセスサーバに対して、DROP SERVER 文を実行します。

   ```
   DROP SERVER my_dir_server;
   ```

結果

ディレクトリアクセスサーバが削除されます。

参照

- 「プロキシテーブルの削除 (Sybase Central の場合)」794 ページ

外部ログイン

デフォルトでは、SQL Anywhere は、クライアントに代わってリモートサーバに接続するときは、常にそのクライアントの名前とパスワードを使用します。外部ログインを作成することによって、このデフォルトを上書きできます。外部ログインとは、リモートサーバと通信するときに使用する代替ログイン名とパスワードのことです。

参照

- 「Windows 統合ログイン」『SQL Anywhere サーバ データベース管理』

外部ログインの作成 (Sybase Central の場合)

Sybase Central を使用して、リモートサーバとの通信に使用する外部ログインを作成します。

前提条件

MANAGE ANY USER システム権限が必要です。

♦ 外部ログインの作成

1. SQL Anywhere 16 プラグインを使用して、ホストデータベースに接続します。
2. 左ウィンドウ枠で、**[リモートサーバ]** をダブルクリックします。
3. リモートサーバを選択し、右ウィンドウ枠で **[外部ログイン]** タブをクリックします。
4. **[ファイル]**[新規]**[外部ログイン]** をクリックします。 »

5. **外部ログイン作成ウィザード**の指示に従います。

結果

外部ログインが作成されます。

参照

- 「CREATE EXTERNLOGIN 文」『SQL Anywhere サーバ SQL リファレンス』

外部ログインの削除 (Sybase Central の場合)

Sybase Central を使用して、必要のない外部ログインを削除できます。

前提条件

MANAGE ANY USER システム権限が必要です。

♦ 外部ログインの削除

1. SQL Anywhere 16 プラグインを使用して、ホストデータベースに接続します。
2. 左ウィンドウ枠で、**[リモートサーバ]** をダブルクリックします。
3. リモートサーバを選択し、右ウィンドウ枠で **[外部ログイン]** タブをクリックします。
4. 外部ログインを選択し、**[編集]** » **[削除]** をクリックします。
5. **[はい]** をクリックします。

結果

外部ログインが削除されます。

参照

- 「DROP EXTERNLOGIN 文」『SQL Anywhere サーバ SQL リファレンス』

プロキシテーブル

リモートオブジェクトに対応するローカルの「プロキシテーブル」を作成すると、意識することなくリモートデータにアクセスできるようになります。プロキシテーブルを使用すると、リモートデータベースがプロキシテーブルの候補としてエクスポートする任意のオブジェクト (テーブル、ビュー、マテリアライズドビューを含む) にアクセスできます。プロキシテーブルを作成するには、次の文のいずれかを使用します。

- リモートサーバにすでにテーブルが存在する場合は、CREATE EXISTING TABLE 文を使用します。この文は、リモートサーバの既存のテーブルのプロキシテーブルを定義します。

- リモートサーバにテーブルが存在しない場合は、CREATE TABLE 文を使用します。この文はリモートサーバに新しいテーブルを作成して、そのテーブルのプロキシテーブルを定義します。

注意
セーブポイント内では、プロキシテーブルを変更することはできません。

プロキシテーブルでトリガを起動する場合は、プロキシテーブルの所有者の権限ではなく、トリガを起動するユーザの権限を使用します。

参照

- 「トランザクション内のセーブポイント」864 ページ

プロキシテーブルのロケーション

AT キーワードを CREATE TABLE 文と CREATE EXISTING TABLE 文とともに使用して、既存のオブジェクトのロケーションを定義します。ロケーション文字列は、ピリオドかセミコロンで区切られた 4 つの部分からなります。セミコロンデリミタを使用すると、データベースフィールドと所有者フィールドでファイル名と拡張子を使用できます。

AT 句の構文は次のようになります。

```
... AT 'server.database.owner.table-name'
```

- **server**　CREATE SERVER 文で指定されたもので、Adaptive Server Anywhere がサーバを識別する名前です。このフィールドはすべてのリモートデータソースに必須です。

- **database**　データベースフィールドの意味は、データソースによって異なります。このフィールドは使用しないで、空にしておく場合があります。ただし、その場合でもデリミタは必要です。

  データソースが Adaptive Server Enterprise の場合は、*database* によってテーブルを保管しているデータベースが指定されます。たとえば、master または pubs2 などです。

  データソースが SQL Anywhere の場合は、このフィールドは適用されません。入力しないでおきます。

  データソースが Excel、Lotus Notes、または Access の場合は、テーブルが保管されているファイルの名前を入力します。ファイル名にピリオドがある場合には、セミコロンデリミタを使用してください。

- **owner**　データベースが所有者の概念をサポートしている場合、所有者名を表します。このフィールドは、何人かの所有者が同じ名前でテーブルを所有する場合にだけ必要です。

- **table-name**　このフィールドはテーブルの名前を指定します。Excel スプレッドシートの場合、これはブックのシートの名前になります。*table-name* を入力しない場合、リモートテーブル名はローカルのプロキシテーブル名と同じであると見なされます。

例

以下はロケーション文字列の使用例です。

●SQL Anywhere：

'RemoteSA..GROUPO.Employees'

●Adaptive Server Enterprise：

'RemoteASE.pubs2.dbo.publishers'

●Excel：

'RemoteExcel;d:¥pcdb¥quarter3.xls;;sheet1$'

●Access：

'RemoteAccessDB;¥¥server1¥production¥inventory.mdb;;parts'

プロキシテーブルの作成 (Sybase Central の場合)

Sybase Central を使用してプロキシテーブルを作成できます。

前提条件

ユーザ本人が所有するプロキシテーブルを作成するには、CREATE PROXY TABLE システム権限が必要です。他のユーザが所有するプロキシテーブルを作成するには、CREATE ANY TABLE または CREATE ANY OBJECT のシステム権限が必要です。

内容と備考

Sybase Central は、システムテーブルのプロキシテーブルの作成をサポートしません。ただし、システムテーブルのプロキシテーブルは、CREATE EXISTING TABLE 文を使用して作成できます。

♦ プロキシテーブルの作成

1. SQL Anywhere 16 プラグインを使用して、ホストデータベースに接続します。
2. 左ウィンドウ枠で、**[リモートサーバ]** をダブルクリックします。
3. リモートサーバを選択し、右ウィンドウ枠で **[プロキシテーブル]** タブを選択します。
4. **[ファイル]**[新規]**[プロキシ テーブル]** をクリックします。 »
5. **プロキシテーブル作成ウィザード**の指示に従います。

結果

プロキシテーブルが作成されます。

参照

- 「CREATE EXISTING TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

プロキシテーブルの作成 (SQL の場合)

Interactive SQL で CREATE TABLE 文または CREATE EXISTING TABLE 文を使用して、プロキシテーブルを作成できます。

前提条件

ユーザ本人が所有するプロキシテーブルを作成するには、CREATE PROXY TABLE システム権限が必要です。他のユーザが所有するプロキシテーブルを作成するには、CREATE ANY TABLE または CREATE ANY OBJECT のシステム権限が必要です。

内容と備考

AT 句とともに CREATE TABLE 文を使用すると、リモートサーバに新しいテーブルを作成し、そのテーブルに対するプロキシテーブルをローカルサーバに作成します。カラムは SQL Anywhere のデータ型を使用して定義します。SQL Anywhere は、リモートサーバのネイティブの型にデータを自動的に変換します。

CREATE TABLE 文を使用してローカルとリモートの両方のテーブルを作成してから、引き続き DROP TABLE 文を使用してプロキシテーブルを削除すると、リモートテーブルも削除されます。ただし、DROP TABLE 文を使用して、CREATE EXISTING TABLE 文を使用して作成されたプロキシテーブルを削除できます。この場合、リモートテーブル名は削除されません。

CREATE EXISTING TABLE 文は、リモートサーバ上にある既存のテーブルにマッピングするプロキシテーブルを作成します。SQL Anywhere は、リモートロケーションのオブジェクトからカラム属性とインデックス情報を導出します。

♦ CREATE EXISTING TABLE 文を使用したプロキシテーブルの作成

1. ホストデータベースに接続します。
2. CREATE EXISTING TABLE 文を実行します。

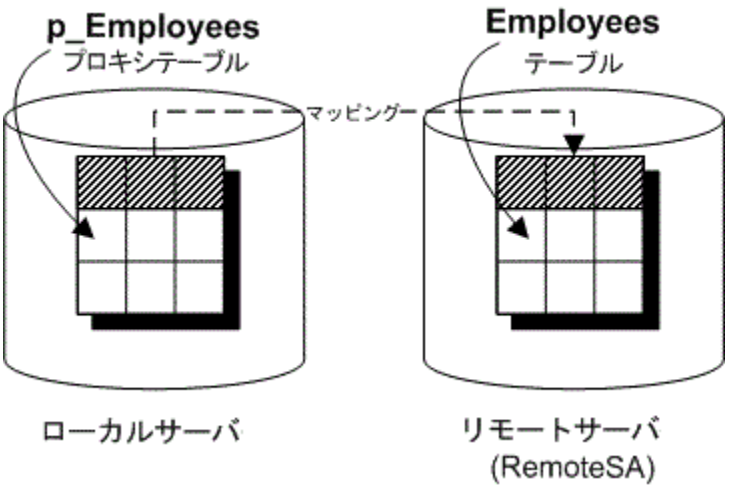
結果

プロキシテーブルが作成されます。

例

サーバ RemoteSA のリモートテーブル Employees にマッピングするローカルサーバ上に、p_Employees というプロキシテーブルを作成するには、次の構文を使用します。

```
CREATE EXISTING TABLE p_Employees
AT 'RemoteSA..GROUPO.Employees';
```



次の文は、プロキシテーブル a1 を Microsoft Access ファイル *mydbfile.mdb* にマッピングします。AT 句では、セミコロン (;) をデリミタとして使用しています。Microsoft Access 用に定義されているサーバの名前は access です。

```
CREATE EXISTING TABLE a1
AT 'access;d:¥mydbfile.mdb;;a1';
```

次の文は、リモートサーバ RemoteSA に Employees というテーブルを作成し、リモートテーブルにマッピングする Members というプロキシテーブルを作成します。

```
CREATE TABLE Members
( membership_id INTEGER NOT NULL,
member_name CHAR( 30 ) NOT NULL,
office_held CHAR( 20 ) NULL )
AT 'RemoteSA..GROUPO.Employees';
```

参照

- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE EXISTING TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

プロキシテーブルの削除 (Sybase Central の場合)

管理者は Sybase Central を使用して、リモートサーバに関連付けられているプロキシテーブルを削除できます。

前提条件

所有者であるか、または DROP ANY TABLE と DROP ANY OBJECT のいずれかのシステム権限を持っている必要があります。

内容と備考

リモートサーバを削除できるようにする前に、そのリモートサーバに関連付けられているすべてのプロキシテーブルを削除する必要があります。

♦ プロキシテーブルの削除

1. SQL Anywhere 16 プラグインを使用して、ホストデータベースに接続します。
2. 左ウィンドウ枠で、**[リモートサーバ]** をダブルクリックします。
3. 右ウィンドウ枠で、**[プロキシテーブル]** タブをクリックします。
4. プロキシテーブルを選択し、**[編集]** » **[削除]** をクリックします。
5. **[はい]** をクリックします。

結果

プロキシテーブルが削除されます。

次の手順

リモートサーバに関連付けられているすべてのプロキシテーブルが削除されたら、そのリモートサーバを削除できます。

参照

- 「リモートサーバの削除 (Sybase Central の場合)」 777 ページ
- 「DROP SERVER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DROP TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

リモートテーブルのカラムのリスト

CREATE EXISTING TABLE 文を実行する前に、リモートテーブルのカラムのリストを取得すると便利な場合があります。sp_remote_columns システムプロシージャは、リモートテーブルにあるカラムのリストとそのデータ型の説明を生成します。sp_remote_columns システムプロシージャの構文は次のとおりです。

```
CALL sp_remote_columns( @server_name, @table_name [, @table_owner [, @table_qualifier ] ] )
```

テーブル名、所有者、またはデータベース名を指定すると、カラムのリストはその指定に当てはまるものだけに限定されます。

たとえば、asetest という名前の Adaptive Server Enterprise サーバの production データベースにある sysobjects テーブルのカラムのリストを取得するには、次のように指定します。

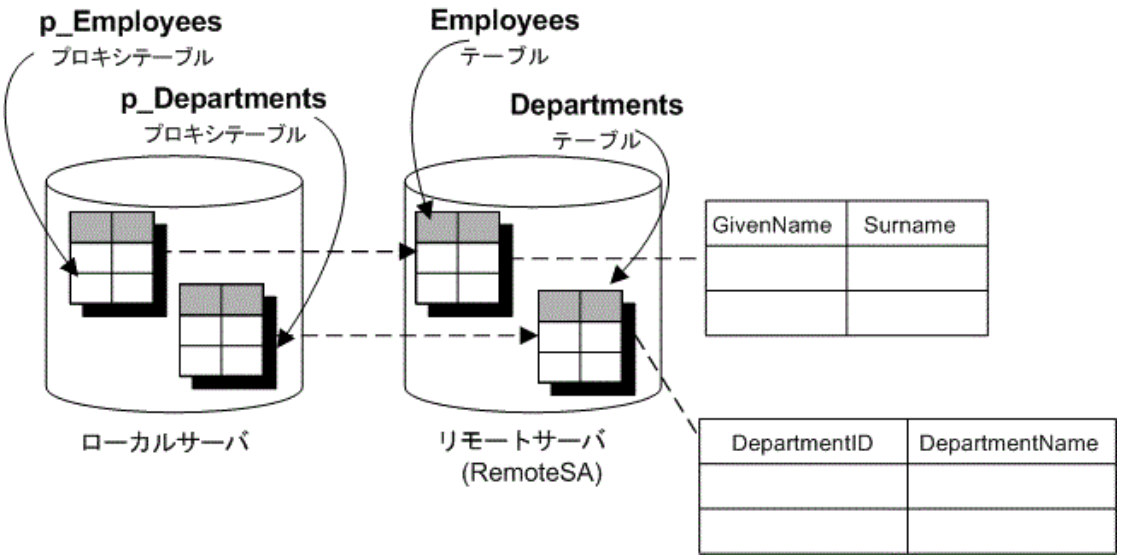
```
CALL sp_remote_columns('asetest, 'sysobjects', null, 'production');
```

参照

- 「sp_remote_columns システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

リモートテーブル間のジョイン

次の図は、リモートサーバ RemoteSA にある SQL Anywhere サンプルデータベースのリモートテーブル Employees と Departments にマッピングされている、ローカルデータベースサーバのプロキシテーブルを示しています。



異なる SQL Anywhere データベースのテーブル間にジョインを使用できます。次の例では、データベースを 1 つだけ使用した簡単なケースについて説明して、その仕組みを示します。

例

2 つのリモートテーブル間のジョインを実行します。

1. *empty.db* という名前の新しいデータベースを作成します。

   このデータベースにはデータがありません。このデータベースは、リモートオブジェクトを定義して、SQL Anywhere サンプルデータベースにアクセスするためだけに使用します。

2. *empty.db* を実行するデータベースサーバを起動します。これを行うには、次のコマンドを実行します。

   ```
   dbsrv16 empty
   ```

3. Interactive SQL から DBA ユーザとして *empty.db* に接続します。

4. 新しいデータベースで、RemoteSA という名前のリモートサーバを作成します。このサーバのサーバクラスは SAODBC で、接続文字列は SQL Anywhere 16 Demo ODBC データソースを参照します。

   ```
   CREATE SERVER RemoteSA
   CLASS 'SAODBC'
   USING 'SQL Anywhere 16 Demo';
   ```

5. この例では、ローカルデータベースと同じユーザ ID とパスワードをリモートデータベースで使用するので、外部ログインは必要ありません。

   場合によっては、リモートサーバのデータベースに接続するとき、ユーザ ID とパスワードを入力します。新しいデータベースの場合は、リモートサーバへの外部ログインを作成します。例では簡素化するために、ローカルのログイン名とリモートのユーザ ID はどちらも DBA とします。

   ```
   CREATE EXTERNLOGIN DBA
   TO RemoteSA
   REMOTE LOGIN DBA
   IDENTIFIED BY sql;
   ```

6. p_Employees プロキシテーブルを定義します。

   ```
   CREATE EXISTING TABLE p_Employees
   AT 'RemoteSA..GROUPO.Employees';
   ```

7. p_Departments プロキシテーブルを定義します。

   ```
   CREATE EXISTING TABLE p_Departments
   AT 'RemoteSA..GROUPO.Departments';
   ```

8. SELECT 文にプロキシテーブルを使用して、ジョインを実行します。

   ```
   SELECT GivenName, Surname, DepartmentName
   FROM p_Employees JOIN p_Departments
   ON p_Employees.DepartmentID = p_Departments.DepartmentID
   ORDER BY Surname;
   ```

複数のローカルデータベースのテーブル間のジョイン

SQL Anywhere サーバでは、複数のローカルデータベースを同時に稼働できます。他のローカル SQL Anywhere データベース内のテーブルをリモートテーブルとして定義することによって、データベース間のジョインを実行できます。

複数のデータベースの指定の詳細については、「CREATE SERVER 文の USING 句」810 ページを参照してください。

例

データベース db1 を使用しているときに、データベース db2 内のテーブルのデータにアクセスするとします。この場合は、データベース db2 のテーブルを示すプロキシテーブル定義を設定します。RemoteSA という名前の SQL Anywhere サーバ上で、db1、db2、db3 の 3 つのデータベースが使用可能だとします。

1. ODBC を使用している場合、アクセスするデータベースのそれぞれに ODBC データソース名を作成します。

2. ジョインの実行元のデータベースに接続します。たとえば db1 に接続します。

3. アクセスするその他のローカルデータベースのそれぞれに、CREATE SERVER 文を実行します。これによって、SQL Anywhere サーバへの「ループバック」接続が設定されます。

```
CREATE SERVER remote_db2
CLASS 'SAODBC'
USING 'RemoteSA_db2';
CREATE SERVER remote_db3
CLASS 'SAODBC'
USING 'RemoteSA_db3';
```

4. アクセスする他のデータベースにあるテーブルに CREATE EXISTING TABLE 文を実行して、プロキシテーブルの定義を作成します。

```
CREATE EXISTING TABLE Employees
AT 'remote_db2...Employees';
```

ネイティブ文とリモートサーバ

FORWARD TO 文を使用して、1 つ以上の文をネイティブの構文でリモートサーバに送信できます。この文は、2 つの方法で使用できます。

●1 つの文をリモートサーバに送信する。

●SQL Anywhere をパススルーモードにして一連の文をリモートサーバに送信する。

FORWARD TO 文を使用して、サーバが正しく設定されていることを検証できます。リモートサーバに文を送信して、SQL Anywhere がエラーメッセージを返さなければ、リモートサーバは正しく設定されています。

プロシージャまたはバッチ内では FORWARD TO 文を使用できません。

指定したサーバに接続できない場合、メッセージがユーザに返されます。接続が確立された場合は、クライアントプログラムが認識できるフォームに結果が変換されます。

例 1

次の文は、バージョン文字列をセレクトすることによって、RemoteASE というサーバへの接続を検証します。

```
FORWARD TO RemoteASE {SELECT @@version};
```

例 2

次の文は、サーバ RemoteASE とのパススルーセッションを示します。

```
FORWARD TO RemoteASE;
   SELECT * FROM titles;
   SELECT * FROM authors;
FORWARD TO;
```

参照

● 「FORWARD TO 文」『SQL Anywhere サーバ SQL リファレンス』

リモートプロシージャコール (RPC)

SQL Anywhere ユーザは、次のリモートサーバへのプロシージャコールを発行できます。

- SQL Anywhere
- Adaptive Server Enterprise
- Oracle Database
- IBM DB2

SQL Anywhere では、複数の結果セットのフェッチなどの、リモートプロシージャからの結果セットのフェッチをサポートしています。また、リモートファンクションを使用してリモートプロシージャとファンクションから戻り値をフェッチできます。リモートプロシージャは、SELECT 文の FROM 句で使用できます。

リモートプロシージャのデータ型

リモートプロシージャコールのパラメータおよび RETURNS 値には、次のデータ型を使用できます。

- [UNSIGNED] SMALLINT
- [UNSIGNED] INTEGER
- [UNSIGNED] BIGINT
- [UNSIGNED] TINYINT
- TIME
- DATE
- TIMESTAMP
- REAL
- DOUBLE
- CHAR
- BIT
- データ型 LONG VARCHAR、LONG NVARCHAR、LONG BINARY は、IN パラメータには指定できますが、OUT パラメータ、INOUT パラメータ、RETURNS 値には指定できません。
- データ型 NUMERIC と DECIMAL は、IN パラメータには指定できますが、OUT パラメータ、INOUT パラメータ、RETURNS 値には指定できません。

リモートプロシージャの作成 (SQL の場合)

Interactive SQL でリモートプロシージャおよびリモートファンクションを作成できます。

前提条件

ユーザ本人が所有するプロシージャおよびファンクションを作成するには、CREATE PROCEDURE システム権限が必要です。他のユーザが所有するプロシージャおよびファンクションを作成するには、CREATE ANY PROCEDURE または CREATE ANY OBJECT の権限が必要です。外部プロシージャとファンクションを作成するには、CREATE EXTERNAL REFERENCE システム権限も必要です。

内容と備考

リモートプロシージャが結果セットを返すことができる場合は、たとえすべてのケースで結果セットを返せるわけではなくても、ローカルプロシージャ定義には RESULT 句を含めてください。

♦ リモートプロシージャの作成

1. ホストデータベースに接続します。
2. プロシージャまたはファンクションを定義する文を実行します。

   次に例を示します。

```
CREATE PROCEDURE RemoteProc()
AT 'bostonase.master.dbo.sp_proc';

CREATE FUNCTION RemoteFunc()
RETURNS INTEGER
AT 'bostonasa..dbo.sp_func';
```

   構文はローカルプロシージャの定義と似ています。ロケーション文字列は、プロシージャのロケーションを定義します。

結果

リモートプロシージャまたはファンクションが作成されます。

例

リモートプロシージャを呼び出すときにパラメータを指定する例を次に示します。

```
CREATE PROCEDURE RemoteUser ( IN username CHAR( 30 ) )
AT 'bostonase.master.dbo.sp_helpuser';
CALL RemoteUser( 'joe' );
```

この例は、リモートサーバ RemoteSA で定義されたファンクションへのインタフェースを作成します。

```
CREATE FUNCTION proxy_maxorder()
RETURNS INTEGER
AT 'RemoteSA;;DBA;maxorder';
```

参照

- 「CREATE FUNCTION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』

リモートプロシージャの作成 (Sybase Central の場合)

管理者は、Sybase Central を使用してリモートプロシージャを作成できます。

前提条件

ユーザ本人が所有するプロシージャおよびファンクションを作成するには、CREATE PROCEDURE システム権限が必要です。他のユーザが所有するプロシージャおよびファンクションを作成するには、CREATE ANY PROCEDURE または CREATE ANY OBJECT の権限が必要です。外部プロシージャとファンクションを作成するには、CREATE EXTERNAL REFERENCE システム権限も必要です。

♦ リモートプロシージャの作成

リモートプロシージャが結果セットを返すことができる場合は、たとえすべてのケースで結果セットを返せるわけではなくても、ローカルプロシージャ定義には RESULT 句を含めてください。

1. SQL Anywhere 16 プラグインを使用して、ホストデータベースに接続します。
2. 左ウィンドウ枠で、**[リモートサーバ]** をダブルクリックします。
3. リモートサーバを選択し、右ウィンドウ枠で **[リモートプロシージャ]** タブをクリックします。
4. **[ファイル]** » **[新規]** » **[リモートプロシージャ]** をクリックします。
5. **[リモートプロシージャ作成ウィザード]** の指示に従います。

結果

リモートプロシージャが作成されます。

参照

- 「CREATE FUNCTION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』

リモートプロシージャの削除 (SQL の場合)

管理者は SQL 文を使用してリモートプロシージャおよびリモートファンクションを削除できます。

前提条件

権限は不要

♦ リモートプロシージャの削除

- プロシージャまたはファンクションを削除する文を実行します。

```
DROP PROCEDURE RemoteProc;
DROP FUNCTION RemoteFunc;
```

結果

リモートプロシージャまたはファンクションが削除されます。

参照

- 「DROP FUNCTION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DROP PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』

リモートプロシージャの削除 (Sybase Central の場合)

管理者は、Sybase Central でリモートプロシージャおよびリモートファンクションを削除できます。

前提条件

権限は不要

♦ リモートプロシージャの削除

1. SQL Anywhere 16 プラグインを使用して、ホストデータベースに接続します。
2. 左ウィンドウ枠で、**[リモートサーバ]** をダブルクリックします。
3. リモートサーバを選択し、右ウィンドウ枠で **[リモートプロシージャ]** タブをクリックします。
4. リモートプロシージャまたはリモート関数を選択し、**[編集]** » **[削除]** をクリックします。
5. **[はい]** をクリックします。

結果

リモートプロシージャが削除されます。

参照

- 「DROP FUNCTION 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DROP PROCEDURE 文」『SQL Anywhere サーバ SQL リファレンス』

トランザクションの管理とリモートデータ

トランザクションを使って、複数の SQL 文をグループ化して 1 つの単位として処理するようにできます。これによって、SQL 文の実行結果はすべてデータベースにコミットされるか、1 つもコミットされないかのいずれかになります。

リモートテーブルでのトランザクション管理は、SQL Anywhere のローカルテーブルでのトランザクション管理とほとんど同じですが、多少異なる点があります。

参照

- 「トランザクションと独立性レベル」861 ページ

リモートトランザクション管理

リモートサーバに関連するトランザクションを管理する方法として、2 フェーズコミットプロトコルが使用されます。SQL Anywhere は、ほとんどの場合においてトランザクションの整合性を保証します。しかし、1 つのトランザクションで 2 つ以上のリモートサーバが呼び出されるときには、分散した作業単位が未定の状態で残る可能性があります。2 フェーズコミットプロトコルを使用する場合でも、リカバリ処理は含まれません。

ユーザのトランザクションを管理する通常の論理は、次のようになっています。

1. SQL Anywhere は、BEGIN TRANSACTION 通知でリモートサーバの作業を開始します。

2. トランザクションのコミットの準備が整うと、SQL Anywhere は、トランザクションの一部であったリモートサーバのそれぞれに PREPARE TRANSACTION 通知を送信します。これによって、リモートサーバがトランザクションをコミットする準備が整っていることを確実にします。

3. PREPARE TRANSACTION 要求が失敗すると、すべてのリモートサーバは現在のトランザクションをロールバックするよう指示されます。

   PREPARE TRANSACTION 要求がすべて成功すると、サーバはトランザクションに関わるリモートサーバのそれぞれに、COMMIT TRANSACTION 要求を送信します。

BEGIN TRANSACTION によって開始すればどのようば文でも、トランザクションを開始できます。BEGIN TRANSACTION を明示しない場合は、SQL 文はリモートサーバに送信されて、1 つのリモートの作業単位として実行されます。

トランザクション管理の制限

トランザクション管理には、次のような制限があります。

- セーブポイントはリモートサーバに伝達されません。

- ネストされた BEGIN TRANSACTION 文と COMMIT TRANSACTION 文がリモートサーバに関わるトランザクションに含まれている場合は、一番外側の組の文だけが処理されます。BEGIN TRANSACTION 文と COMMIT TRANSACTION 文を含む一番内側の組は、リモートサーバに転送されません。

内部オペレーション

この項では、クライアントアプリケーションの裏側で行われている SQL Anywhere のリモートサーバに対するオペレーションについて説明します。

クエリで実行される内部オペレーション

次の手順は、ローカルとリモートの両方の、すべてのクエリに実行されます。

クエリの解析

文は、クライアントから受信されると、データベースサーバによって解析されます。有効な SQL Anywhere の SQL 文でないと、エラーが発生します。

クエリの正規化

クエリ内で参照されているオブジェクトが検証され、いくつかのデータ型の互換性が検査されます。

次のクエリを例にとります。

```
SELECT *
FROM t1
WHERE c1 = 10;
```

クエリの正規化の段階で、カラム c1 を持つテーブル t1 がシステムテーブルに存在することを確認します。また、カラム c1 のデータ型が値 10 と互換性があることも確認します。たとえば、カラムのデータ型が TIMESTAMP であれば、この文は拒否されます。

クエリの前処理

クエリの前処理では、クエリの最適化の準備をします。ここで SQL 文の表現が変更されることもあるので、SQL Anywhere がリモートサーバに引き渡す実際の SQL 文は、セマンティック上は同じであっても構文が元のものと同じとはかぎりません。

前処理は、ビューによって参照されるテーブルでクエリが操作できるよう、ビューの拡張を行います。処理を効率化するため、式が並べ替えられ、サブクエリが変更される場合があります。たとえば、いくつかのサブクエリがジョインに変換される場合があります。

サーバの機能

以降の手順は、SQL 文の型と、作業に関わるリモートサーバの機能によって異なります。

SQL Anywhere では、各リモートサーバに機能が定義されています。これらの機能は、ISYSCAPABILITIES システムテーブルに格納され、リモートサーバへの最初の接続の間に初期化されます。

一般的なサーバクラスである ODBC は、ODBC ドライバから返される情報によって厳密にリモートサーバの機能を判別します。DB2ODBC などのその他のサーバクラスには、リモートサーバタイプの機能情報についてより詳細な情報があり、その情報を使用して、ドライバから返される情報を補います。

ISYSCAPABILITIES にサーバが追加されると、以後、そのリモートサーバの機能情報はそのシステムテーブルから取り出されるようになります。

リモートサーバは特定の SQL 文の全機能をサポートしているとはかぎらないので、SQL Anywhere では、クエリがリモートサーバに対応できるようになるまで、文を単純なコンポーネントに分割する必要があります。リモートサーバに渡されない SQL 機能は、SQL Anywhere 自身が評価します。

たとえば、あるクエリに ORDER BY 文があるとします。リモートサーバが ORDER BY を実行できない場合、ORDER BY を除いて、文がリモートサーバに送信されます。SQL Anywhere は、結果が返されると、ORDER BY を実行してから結果をユーザに返します。したがって、ユーザは、SQL Anywhere でサポートされる全種類の SQL を使用できます。

文の完全なパススルー

効率性を考慮して、SQL Anywhere では、文に含まれるできるだけ多くの要素をリモートサーバに渡します。多くの場合、これは元々 SQL Anywhere に指定された完全な文です。

SQL Anywhere は、次のような場合に完全な文を渡します。

- 文内のテーブルがどれも同じリモートサーバに存在している。
- リモートサーバが文内のすべての構文を処理できる。

まれに、リモートサーバが作業を行うよりも、SQL Anywhere がいくつかの作業を行った方が効率が良い場合があります。たとえば、SQL Anywhere のソートアルゴリズムの方が優れていることがあります。この場合は、ALTER SERVER 文を使用して、リモートサーバの機能の変更を検討します。

参照

- 「ALTER SERVER 文」『SQL Anywhere サーバ SQL リファレンス』

文の部分的なパススルー

ある文に複数のサーバへの参照が含まれている場合、またはリモートサーバではサポートされていない SQL 機能を文が使用している場合は、クエリは複数の単純な部分に分解されます。

SELECT

引き渡すことのできない部分が取り除かれながら、SELECT 文は分解されていきます。取り除かれた部分の処理は SQL Anywhere によって実行されます。たとえば、次の文内にある ATAN2 関数をリモートサーバが処理できないとします。

```
SELECT a,b,c
WHERE ATAN2( b, 10 ) > 3
AND c = 10;
```

リモートサーバに送信される文は、次のように変換されます。

```
SELECT a,b,c WHERE c = 10;
```

次に、SQL Anywhere がローカルで WHERE ATAN2(b, 10) > 3 を中間結果セットに適用します。

ジョイン

2 つのテーブルがジョインされると、1 つは外部テーブルとなります。外部テーブルは、テーブルに適用する WHERE 条件に基づいてスキャンされます。検出された、条件に合うどのローに対しても、内部テーブルと呼ばれるもう 1 つのテーブルがスキャンされ、ジョイン条件に一致するローが検出されます。

リモートテーブルが参照されるときにも同じアルゴリズムが使用されます。通常は、ローカルテーブルよりリモートテーブルを検索する方がコストがかかるので (ネットワーク I/O のため)、できるかぎりリモートテーブルをジョインの一番外側のテーブルにします。

UPDATE と DELETE

条件に合うローが検出されたとき、SQL Anywhere が UPDATE 文または DELETE 文全体をリモートサーバに渡すことができない場合は、元の WHERE 句のできるだけ多くの部分を持つ SELECT 文に文を変更して、WHERE CURRENT OF *cursor-name* を指定する位置付け UPDATE 文または DELETE 文を続けます。

たとえば、リモートサーバが関数 ATAN2 をサポートしていないとします。

```
UPDATE t1
SET a = atan2( b, 10 )
WHERE b > 5;
```

この文は次のように変換されます。

```
SELECT a,b
FROM t1
WHERE  b > 5;
```

ローが検出されるたびに、SQL Anywhere は a の新しい値を計算して、次の文を実行します。

```
UPDATE t1
SET a = 'new value'
WHERE CURRENT OF CURSOR;
```

new value と等しい値が a にある場合、位置付け UPDATE は必要なく、リモートに送信されません。

テーブルスキャンを必要とする UPDATE 文または DELETE 文を処理するには、位置付け UPDATE または DELETE (WHERE CURRENT OF *cursor-name*) を実行する機能をリモートのデータソースがサポートしていなければなりません。データソースによってはこの機能をサポートしていません。

テンポラリテーブルは更新できない
中間テンポラリテーブルが必要な場合は、UPDATE または DELETE を実行できません。これは ORDER BY を持つクエリと、サブクエリを持つクエリのいくつかで発生します。

リモートデータアクセスのトラブルシューティング

この項では、リモートサーバへのアクセスのトラブルシューティングのヒントをいくつか説明します。

リモートデータに対してサポートされない機能

次の SQL Anywhere 機能はリモートデータではサポートされていません。

- リモートテーブルでの ALTER TABLE 文
- プロキシテーブルに定義されたトリガ
- SQL Remote
- リモートテーブルを参照する外部キー
- READTEXT 関数、WRITETEXT 関数、TEXTPTR 関数
- 位置付け UPDATE 文と DELETE 文
- 中間テンポラリテーブルを必要とする UPDATE 文と DELETE 文
- リモートデータに対して開かれたカーソルの後方スクロール。フェッチ文は NEXT または RELATIVE 1 でなければなりません。
- プロキシテーブルを参照する式を含む関数の呼び出し
- リモートテーブルのカラムにリモートサーバに関するキーワードがある場合、そのカラムのデータにはアクセスできない。CREATE EXISTING TABLE 文を実行して定義をインポートすることはできますが、そのカラムを選択することはできません。

大文字と小文字の区別

SQL Anywhere データベースの大文字と小文字の区別の設定は、アクセスするリモートサーバが使用する設定に合わせてください。

デフォルトでは、SQL Anywhere データベースは大文字と小文字を区別しないで作成されます。この設定では、大文字と小文字を区別するデータベースから選択を行ったときに、予期しない結果が発生することがあります。ORDER BY または文字列比較がリモートサーバに送信されるか、それともローカルの SQL Anywhere によって評価されるかによって、発生する結果が異なります。

接続のテスト

次の手順で、リモートサーバに接続できることを確認します。

- Interactive SQL などのクライアントツールを使用してリモートサーバに接続できることを確認してから、SQL Anywhere を設定します。
- リモートサーバに対して簡単なパススルー文を実行して、接続とリモートログインの設定を確認します。次に例を示します。

```
FORWARD TO RemoteSA {SELECT @@version};
```

- リモートサーバとの対話をトレースするために、リモートトレーシングを有効にします。次に例を示します。

```
SET OPTION cis_option = 7;
```

  リモートトレーシングを有効にすると、データベースサーバのメッセージウィンドウにトレーシング情報が表示されます。この出力をファイルに記録するには、データベースサーバの起動時に -o サーバオプションを指定します。

参照

- 「cis_option オプション」『SQL Anywhere サーバ データベース管理』
- 「-o データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「PASSTHROUGH 文 [SQL Remote]」『SQL Anywhere サーバ SQL リファレンス』

クエリ上でブロックされるクエリ

クエリが実行する個々のタスクをサポートするのに十分なスレッドが使用可能でなければなりません。必要なタスクの数を提供できないと、クエリはそのクエリ上でブロックされます。

参照

- 「トランザクションのブロックとデッドロック」881 ページ

ODBC を使用したリモートデータアクセスの接続

ODBC を介してリモートデータベースにアクセスする場合、リモートサーバへの接続には名前が付けられます。名前を使用して接続を削除し、リモート要求をキャンセルできます。

接続の名前は、ASACIS_*conn-name* のようになります。*conn-name* は、ローカル接続の接続 ID です。接続 ID は、sa_conn_info ストアドプロシージャから取得できます。

参照

- 「sa_conn_info システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

リモートデータアクセスのサーバクラス

CREATE SERVER 文に指定するサーバクラスは、リモート接続の動作を決定します。サーバクラスは、サーバの機能の詳細な情報を SQL Anywhere に提供します。SQL Anywhere は、サーバの機能に合わせて SQL 文をフォーマットします。

サーバクラスはすべて ODBC ベースです。各サーバクラスには各種のユニークな特性があります。リモートデータアクセス用にサーバを設定するには、この特性を知っておく必要があります。サーバクラスのカテゴリ全般についての情報とともに、個々のサーバクラスに固有の情報を参照する必要があります。

サーバのクラスは次のとおりです。

- SAODBC
- ULODBC
- ADSODBC
- ASEODBC
- DB2ODBC
- HANAODBC
- IQODBC
- MIRROR
- MSACCESSODBC
- MSSODBC
- MYSQLODBC
- ODBC
- ORAODBC

注意
リモートデータアクセスを使用する際に、Unicode をサポートしていない ODBC ドライバを使用すると、その ODBC ドライバから受け取るデータに対して、文字セット変換が実行されません。

ODBC 外部サーバ定義

ODBC ベースのリモートサーバを定義する最も一般的な方法は、ODBC データソースを基にすることです。これを実行するために、ODBC データソースアドミニストレータを使用して、データソースを作成することができます。

データソースを定義すると、CREATE SERVER 文の USING 句は ODBC データソース名 (DSN) を参照します。

たとえば、データソース名も mydb2 である mydb2 という名前の IBM DB2 サーバを定義するには、次の文を使用します。

```
CREATE SERVER mydb2
CLASS 'DB2ODBC'
USING 'mydb2';
```

使用するドライバはデータベースサーバのビット設定と一致する必要があります。

Windows では、ビット設定がデータベースサーバと一致するシステムデータソース名 (System DSN) を定義する必要もあります。たとえば、32 ビットのシステム DSN を作成するには、32 ビットの ODBC データソースアドミニストレータを使用します。ユーザ DSN にはビット設定はありません。

データソースの代わりに接続文字列を使用

データソース名を使用しない代わりに、CREATE SERVER 文の USING 句に接続文字列を指定します。これを実行するには、使用している ODBC ドライバの接続パラメータが必要です。たとえば、次は SQL Anywhere データベースサーバへの接続の例です。

```
CREATE SERVER TestSA
CLASS 'SAODBC'
USING 'DRIVER=SQL Anywhere 16;HOST=myhost;Server=TestSA;DBN=sample';
```

この例で定義される接続先の TestSA という名前のデータベースサーバは、myhost というコンピュータで実行され、データベースが sample という名前で、使用するプロトコルが TCP/IP です。

参照

各 ODBC サーバクラスに固有の情報については、次を参照してください。

- 「サーバクラス SAODBC」811 ページ
- 「サーバクラス ULODBC」812 ページ
- 「サーバクラス ADSODBC」812 ページ
- 「サーバクラス ASEODBC」814 ページ
- 「サーバクラス DB2ODBC」817 ページ
- 「サーバクラス HANAODBC」819 ページ
- 「サーバクラス IQODBC」820 ページ
- 「サーバクラス MSACCESSODBC」821 ページ
- 「サーバクラス MSSODBC」823 ページ
- 「サーバクラス MYSQLODBC」825 ページ
- 「サーバクラス ODBC」827 ページ
- 「サーバクラス ORAODBC」829 ページ
- 「ODBC データソース」『SQL Anywhere サーバ データベース管理』
- 「CREATE SERVER 文」『SQL Anywhere サーバ SQL リファレンス』

CREATE SERVER 文の USING 句

アクセスするリモートの SQL Anywhere データベースごとに、個別の CREATE SERVER 文を発行してください。たとえば、TestSA という名前の SQL Anywhere サーバが Banana というコンピュータで稼働していて、3 つのデータベース (db1、db2、db3) を所有する場合は、次のようなリモートサーバを設定します。

```
CREATE SERVER TestSAdb1
CLASS 'SAODBC'
USING 'DRIVER=SQL Anywhere 16;HOST=Banana;Server=TestSA;DBN=db1';

CREATE SERVER TestSAdb2
CLASS 'SAODBC'
USING 'DRIVER=SQL Anywhere 16;HOST=Banana;Server=TestSA;DBN=db2';

CREATE SERVER TestSAdb3
CLASS 'SAODBC'
USING 'DRIVER=SQL Anywhere 16;HOST=Banana;Server=TestSA;DBN=db3';
```

データベース名を指定しない場合、リモート接続にはリモートの SQL Anywhere サーバのデフォルトデータベースが使用されます。

参照

- 「CREATE SERVER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「接続パラメータ」『SQL Anywhere サーバ データベース管理』

サーバクラス SAODBC

サーバクラス SAODBC のリモートサーバは、SQL Anywhere データベースサーバです。SQL Anywhere データソースの設定には、特別な要件はありません。

複数のデータベースをサポートする SQL Anywhere データベースサーバにアクセスするには、各データベースへの接続を定義する ODBC データソース名を作成します。これらの ODBC データソース名のそれぞれに対して、CREATE SERVER 文を実行します。

例

CREATE SERVER 文の USING 句で接続文字列を指定して、SQL Anywhere データベースに接続します。

```
CREATE SERVER TestSA
CLASS 'SAODBC'
USING 'DRIVER=SQL Anywhere 16;HOST=myhost;Server=TestSA;DBN=sample';
```

サーバクラス MIRROR

サーバクラス MIRROR のリモートサーバは、SQL Anywhere データベースサーバです。MIRROR サーバクラスは、ODBC を介してリモート SQL Anywhere サーバへの接続を行います。ただし、リモートサーバの作成時には、SYS.SYSMIRRORSERVER カタログテーブルにあるミラーサーバ名が USING 句に含まれます。リモートデータアクセスレイヤは、このミラーサーバ名を使用して、リモート SQL Anywhere サーバに対する接続文字列を作成します。

注意

リモートデータアクセスミラーサーバのテーブルにマッピングされたプロキシテーブルを照会すると、リモートデータアクセスレイヤは SYS.SYSMIRRORSERVER カタログテーブルと SYS.SYSMIRRORSERVEROPTION カタログテーブルの両方を参照して、リモートデータアクセスミラーサーバによって指定された SA サーバへの接続を確立するために使用する接続文字列を決定します。

例

MyMirrorServer に接続するためのリモートデータアクセスミラーサーバを設定するには、次のような文を実行します。

```
CREATE SERVER remote_server_name
CLASS 'MIRROR'
USING 'MirrorServer=MyMirrorServer';
```

注意
他のリモートデータアクセスサーバのクラスとは異なり、リモートデータミラーアクセスサーバへの接続は、リモート接続が切断されると自動的に再接続されます。

サーバクラス ULODBC

サーバクラスが ULODBC のリモートサーバは Ultra Light データベースサーバです。Ultra Light データベースへの接続を定義する ODBC データソース名を作成します。ODBC データソース名に対して、CREATE SERVER 文を実行します。

Ultra Light は、SQL Anywhere で使用可能なデータ型のサブセットをサポートしているため、Ultra Light のデータ型と SQL Anywhere のデータ型には 1 対 1 のマッピングが存在します。

注意
Mac OS X 上で実行している Ultra Light データベースには、リモートサーバを作成できません。

参照

- 「Ultra Light SQL データ型」『Ultra Light データベース管理とリファレンス』

例

CREATE SERVER 文の USING 句で接続文字列を指定して、Ultra Light データベースに接続します。

```
CREATE SERVER TestUL
CLASS 'ULODBC'
USING 'DRIVER=UltraLite 16;UID=DBA;PWD=sql;DBF=custdb.udb'
```

サーバクラス ADSODBC

CREATE TABLE 文を実行するときに、SQL Anywhere は、次のデータ型変換を使用して、データ型を対応する Advantage Database Server のデータ型に自動的に変換します。

SQL Anywhere データ型	ADS のデフォルトデータ型
BIT	Logical
VARBIT(*n*)	Binary(*n*)
LONG VARBIT	Binary(2G)
TINYINT	Integer
SMALLINT	Integer
INTEGER	Integer

SQL Anywhere データ型	ADS のデフォルトデータ型
BIGINT	Numeric(32)
UNSIGNED TINYINT	Numeric(11)
UNSIGNED SMALLINT	Numeric(11)
UNSIGNED INTEGER	Numeric(11)
UNSIGNED BIGINT	Numeric(32)
CHAR(*n*)	Character(*n*)
VARCHAR(*n*)	VarChar(*n*)
LONG VARCHAR	VarChar(65000)
NCHAR(*n*)	NChar(*n*)
NVARCHAR(*n*)	NVarChar(*n*)
LONG NVARCHAR	NVarChar(32500)
BINARY(*n*)	Binary(*n*)
VARBINARY(*n*)	Binary(*n*)
LONG BINARY	Binary(2G)
DECIMAL(*precision*, *scale*)	Numeric(*precision*+3)
NUMERIC(*precision*, *scale*)	Numeric(*precision*+3)
SMALLMONEY	Money
MONEY	Money
REAL	Double
DOUBLE	Double
FLOAT(*n*)	Double
DATE	Date
TIME	Time

SQL Anywhere データ型	ADS のデフォルトデータ型
TIMESTAMP	TimeStamp
TIMESTAMP WITH TIMEZONE	Char(254)
XML	Binary(2G)
ST_GEOMETRY	Binary(2G)
UNIQUEIDENTIFIER	Binary(2G)

サーバクラス ASEODBC

サーバクラスが ASEODBC のリモートサーバは、Adaptive Server Enterprise (バージョン 10 以降) データベースサーバです。SQL Anywhere では、クラスが ASEODBC のリモートの Adaptive Server Enterprise データベースサーバに接続するために、Adaptive Server Enterprise ODBC ドライバと Open Client 接続ライブラリのインストールが必要です。

注意

- Open Client はバージョン 11.1.1、EBF 7886 以降が必要です。Open Client をインストールして Adaptive Server Enterprise サーバへの接続を検証してから、ODBC をインストールして SQL Anywhere を設定してください。Sybase ODBC ドライバはバージョン 11.1.1、EBF 7911 以降が必要です。

- quoted_identifier オプションのローカル設定は、Adaptive Server Enterprise の引用符付き識別子の使用を制御します。たとえば、quoted_identifier オプションをローカルで Off に設定すると、Adaptive Server Enterprise に対して引用符付き識別子がオフになります。

- **[Configuration Manager]** でユーザデータソースを次の属性で設定します。

  - **[General] タブ**　**[Data Source Name]** に任意の値を入力します。この値は、CREATE SERVER 文の USING 句に使用されます。

    サーバ名は Sybase interfaces ファイルにあるサーバ名と一致させてください。

  - **[Advanced] タブ**　**[Application Using Threads]** オプションと **[Enable Quoted Identifiers]** オプションをクリックします。

  - **[Connection] タブ**　[charset] フィールドを、SQL Anywhere の文字セットに一致するように設定します。

    [language] フィールドを、エラーメッセージを表示したい言語に設定します。

  - **[Performance] タブ**　**[Prepare Method]** を 「2-Full」 に設定します。

    最高のパフォーマンスを得るには、**[Fetch Array Size]** をできるだけ大きな値に設定します。これはメモリ内にキャッシュされるローの数なので、この値を大きくすると必要なメ

モリ量が増大します。Adaptive Server Enterprise では 100 の値を使用することをおすすめします。

[Select Method] を 「0-Cursor」 に設定します。

[Packet Size] をできるだけ大きな値に設定します。Adaptive Server Enterprise では -1 の値を使用することをおすすめします。

[Connection Cache] を 1 に設定します。

データ型変換：ODBC と Adaptive Server Enterprise

CREATE TABLE 文を実行するときに、SQL Anywhere は、データ型を対応する Adaptive Server Enterprise のデータ型に自動的に変換します。次の表に、SQL Anywhere から Adaptive Server Enterprise へのデータ型変換を示します。

SQL Anywhere データ型	Adaptive Server Enterprise のデフォルトデータ型
BIT	bit
VARBIT(*n*)	if (n <= 255) varbinary(*n*) else image
LONG VARBIT	image
TINYINT	tinyint
SMALLINT	smallint
INT, INTEGER	int
BIGINT	numeric(20,0)
UNSIGNED TINYINT	tinyint
UNSIGNED SMALLINT	int
UNSIGNED INTEGER	numeric(11,0)
UNSIGNED BIGINT	numeric(20,0)
CHAR(*n*)	if (n <= 255) char(*n*) else text
VARCHAR(*n*)	if (n <= 255) varchar(*n*) else text
LONG VARCHAR	text
NCHAR(*n*)	if (n <= 255) nchar(*n*) else ntext
NVARCHAR(*n*)	if (n <= 255) nvarchar(*n*) else ntext

SQL Anywhere データ型	Adaptive Server Enterprise のデフォルトデータ型
LONG NVARCHAR	ntext
BINARY(*n*)	if (n <= 255) binary(*n*) else image
VARBINARY(*n*)	if (n <= 255) varbinary(*n*) else image
LONG BINARY	image
DECIMAL(*prec*,*scale*)	decimal(*prec*,*scale*)
NUMERIC(*prec*,*scale*)	numeric(*prec*,*scale*)
SMALLMONEY	numeric(10,4)
MONEY	numeric(19,4)
REAL	real
DOUBLE	float
FLOAT(*n*)	float(*n*)
DATE	datetime
TIME	datetime
SMALLDATETIME	smalldatetime
TIMESTAMP	datetime
TIMESTAMP WITH TIMEZONE	varchar(254)
XML	text
ST_GEOMETRY	image
UNIQUEIDENTIFIER	binary(16)

例

CREATE SERVER 文の USING 句で接続文字列を指定して、Adaptive Server Enterprise データベースに接続します。

```
CREATE SERVER TestASE
CLASS 'ASEODBC'
USING 'DRIVER=SYBASE ASE ODBC
Driver;Server=TestASE;Port=5000;Database=testdb;UID=username;PWD=password'
```

サーバクラス DB2ODBC

サーバクラスが DB2ODBC のリモートサーバは IBM DB2 データベースサーバです。

注意

- iAnywhere は、IBM の DB2 Connect バージョン 5 (修正パック WR09044 付き) の使用を確認しています。この製品の説明に従って、ODBC 構成の設定とテストを実行してください。SQL Anywhere には、IBM DB2 データソースの設定について特別な要件はありません。

- 以下は、mydb2 という ODBC データソースを持つ IBM DB2 サーバの CREATE EXISTING TABLE 文の例です。

```
CREATE EXISTING TABLE ibmcol
AT 'mydb2..sysibm.syscolumns';
```

データ型変換：IBM DB2

CREATE TABLE 文を実行するときに、SQL Anywhere は、データ型を対応する IBM DB2 のデータ型に自動的に変換します。次の表に、SQL Anywhere から IBM DB2 へのデータ型変換を示します。

SQL Anywhere データ型	IBM DB2 のデフォルトデータ型
BIT	smallint
VARBIT(*n*)	if (n <= 4000) varchar(*n*) for bit data else long varchar for bit data
LONG VARBIT	long varchar for bit data
TINYINT	smallint
SMALLINT	smallint
INTEGER	int
BIGINT	decimal(20,0)
UNSIGNED TINYINT	int
UNSIGNED SMALLINT	int
UNSIGNED INTEGER	decimal(11.0)
UNSIGNED BIGINT	decimal(20,0)
CHAR(*n*)	if (n < 255) char(*n*) else if (n <= 4000) varchar(*n*) else long varchar
VARCHAR(*n*)	if (n <= 4000) varchar(*n*) else long varchar

SQL Anywhere データ型	IBM DB2 のデフォルトデータ型
LONG VARCHAR	long varchar
NCHAR(*n*)	サポートされていない
NVARCHAR(*n*)	サポートされていない
LONG NVARCHAR	サポートされていない
BINARY(*n*)	if (n <= 4000) varchar(*n*) for bit data else long varchar for bit data
VARBINARY(*n*)	if (n <= 4000) varchar(*n*) for bit data else long varchar for bit data
LONG BINARY	long varchar for bit data
DECIMAL(*prec*,*scale*)	decimal(*prec*,*scale*)
NUMERIC(*prec*,*scale*)	decimal(*prec*,*scale*)
SMALLMONEY	decimal(10,4)
MONEY	decimal(19,4)
REAL	real
DOUBLE	float
FLOAT(*n*)	float(*n*)
DATE	date
TIME	time
TIMESTAMP	timestamp
TIMESTAMP WITH TIMEZONE	varchar(254)
XML	long varchar for bit data
ST_GEOMETRY	long varchar for bit data
UNIQUEIDENTIFIER	varchar(16) for bit data

サーバクラス HANAODBC

サーバクラスが HANAODBC のリモートサーバは SAP HANA データベースサーバです。

注意

●以下は、mySAPHANA という ODBC データソースを持つ SAP HANA データベースサーバの CREATE EXISTING TABLE 文の例です。

```
CREATE EXISTING TABLE hanatable
AT 'mySAPHANA..dbo.hanatable';
```

データ型変換：SAP HANA

CREATE TABLE 文を実行するときに、SQL Anywhere は、データ型を対応する SAP HANA のデータ型に自動的に変換します。次の表に、SQL Anywhere から SAP HANA へのデータ型変換を示します。

SQL Anywhere データ型	SAP HANA のデフォルトデータ型
BIT	TINYINT
VARBIT(*n*)	if (n <= 5000) VARBINARY(*n*) else BLOB
LONG VARBIT	BLOB
TINYINT	TINYINT
SMALLINT	SMALLINT
INTEGER	INTEGER
BIGINT	BIGINT
UNSIGNED TINYINT	TINYINT
UNSIGNED SMALLINT	INTEGER
UNSIGNED INTEGER	BIGINT
UNSIGNED BIGINT	DECIMAL(20,0)
CHAR(*n*)	if (n <= 5000) VARCHAR(*n*) else CLOB
VARCHAR(*n*	if (n <= 5000) VARCHAR(*n*) else CLOB
LONG VARCHAR	CLOB
NCHAR(*n*)	if (n <= 5000) NVARCHAR(*n*) else NCLOB
NVARCHAR(*n*)	if (n <= 5000) NVARCHAR(*n*) else NCLOB

SQL Anywhere データ型	SAP HANA のデフォルトデータ型
LONG NVARCHAR	NCLOB
BINARY(*n*)	if (n <= 5000) VARBINARY(*n*) else BLOB
VARBINARY(*n*)	if (n <= 5000) VARBINARY(*n*) else BLOB
LONG BINARY	BLOB
DECIMAL(*precision*, *scale*)	DECIMAL(*precision*, *scale*)
NUMERIC(*precision*, *scale*)	DECIMAL(*precision*, *scale*)
SMALLMONEY	DECIMAL(13,4)
MONEY	DECIMAL(19.4)
REAL	REAL
DOUBLE	FLOAT
FLOAT(*n*)	FLOAT
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP
TIMESTAMP WITH TIMEZONE	VARCHAR(254)
XML	BLOB
ST_GEOMETRY	BLOB
UNIQUEIDENTIFIER	VARBINARY(16)

サーバクラス IQODBC

サーバクラスが IQODBC のリモートサーバは SAP Sybase IQ データベースサーバです。SAP Sybase IQ データソースの設定には、特別な要件はありません。

複数のデータベースをサポートする SAP Sybase IQ データベースサーバにアクセスするには、各データベースへの接続を定義する ODBC データソース名を作成します。これらの ODBC データソース名のそれぞれに対して、CREATE SERVER 文を実行します。

参照

- 「CREATE SERVER 文の USING 句」810 ページ

サーバクラス MSACCESSODBC

Access データベースは *.mdb* ファイルに格納されます。ODBC マネージャを使用して、ODBC データソースを作成し、これらのファイルの 1 つにマッピングします。新しい *.mdb* ファイルは、ODBC マネージャを使って作成できます。SQL Anywhere でテーブルを作成するときにデフォルトを指定しないと、このデータベースファイルがデフォルトになります。

ODBC データソースが access という名前であると仮定した場合、次のいずれかの文を使用してデータにアクセスできます。

- ```
  CREATE TABLE tab1 (a int, b char(10))
  AT 'access...tab1';
  ```
- ```
  CREATE TABLE tab1 (a int, b char(10))
  AT 'access;d:¥¥pcdb¥¥data.mdb;;tab1';
  ```
- ```
  CREATE EXISTING TABLE tab1
  AT 'access;d:¥¥pcdb¥¥data.mdb;;tab1';
  ```

Access では所有者名の修飾をサポートしないので、これはブランクのままにしてください。

データ型変換：Microsoft Access

SQL Anywhere データ型	Microsoft Access のデフォルトデータ型
BIT	TINYINT
VARBIT(*n*)	if (n <= 4000) BINARY(*n*) else IMAGE
LONG VARBIT	IMAGE
TINYINT	TINYINT
SMALLINT	SMALLINT
INTEGER	INTEGER
BIGINT	DECIMAL(19,0)
UNSIGNED TINYINT	TINYINT
UNSIGNED SMALLINT	INTEGER
UNSIGNED INTEGER	DECIMAL(11,0)
UNSIGNED BIGINT	DECIMAL(20,0)
  ```

SQL Anywhere データ型	Microsoft Access のデフォルトデータ型
CHAR(*n*)	if (n < 255) CHARACTER(*n*) else TEXT
VARCHAR(*n*)	if (n < 255) CHARACTER(*n*) else TEXT
LONG VARCHAR	TEXT
NCHAR(*n*)	Not supported
NVARCHAR(*n*)	Not supported
LONG NVARCHAR	Not supported
BINARY(*n*)	if (n <= 4000) BINARY(*n*) else IMAGE
VARBINARY(*n*)	if (n <= 4000) BINARY(*n*) else IMAGE
LONG BINARY	IMAGE
DECIMAL(*precision*, *scale*)	DECIMAL(*precision*, *scale*)
NUMERIC(*precision*, *scale*)	DECIMAL(*precision*, *scale*)
SMALLMONEY	MONEY
MONEY	MONEY
REAL	REAL
DOUBLE	FLOAT
FLOAT(*n*)	FLOAT
DATE	DATETIME
TIME	DATETIME
TIMESTAMP	DATETIME
TIMESTAMP WITH TIMEZONE	CHARACTER(254)
XML	XML
ST_GEOMETRY	IMAGE
UNIQUEIDENTIFIER	BINARY(16)

サーバクラス MSSODBC

サーバクラス MSSODBC を使用して、Microsoft SQL Server に、そのいずれかの ODBC ドライバを介してアクセスします。

注意

- これまでに使用されてきた Microsoft SQL Server ODBC ドライバのバージョンは次のとおりです。

  - Microsoft SQL Server ODBC Driver バージョン 06.01.7601

  - Microsoft SQL Server Native Client バージョン 10.00.1600

- 次に、Microsoft SQL Server の例を示します。

```
CREATE SERVER mysqlserver
CLASS 'MSSODBC'
USING 'DSN=MSSODBC_cli';

CREATE EXISTING TABLE accounts
AT 'mysqlserver.master.dbo.accounts';
```

- quoted_identifier オプションのローカル設定は、Microsoft SQL Server の引用符付き識別子の使用を制御します。たとえば、quoted_identifier オプションをローカルで Off に設定すると、Microsoft SQL Server に対して引用符付き識別子がオフになります。

データ型変換：Microsoft SQL Server

CREATE TABLE 文を実行するときに、SQL Anywhere は、次のデータ型変換を使用して、データ型を対応する Microsoft SQL Server のデータ型に自動的に変換します。

SQL Anywhere データ型	Microsoft SQLServer のデフォルトデータ型
BIT	bit
VARBIT(*n*)	if (n <= 255) varbinary(*n*) else image
LONG VARBIT	image
TINYINT	tinyint
SMALLINT	smallint
INTEGER	int
BIGINT	numeric(20,0)
UNSIGNED TINYINT	tinyint
UNSIGNED SMALLINT	int

SQL Anywhere データ型	Microsoft SQLServer のデフォルトデータ型
UNSIGNED INTEGER	numeric(11,0)
UNSIGNED BIGINT	numeric(20,0)
CHAR(*n*)	if (n <= 255) char(*n*) else text
VARCHAR(*n*)	if (n <= 255) varchar(*n*) else text
LONG VARCHAR	text
NCHAR(*n*)	if (n <= 4000) nchar(*n*) else ntext
NVARCHAR(*n*)	if (n <= 4000) nvarchar(*n*) else ntext
LONG NVARCHAR	ntext
BINARY(*n*)	if (n <= 255) binary(*n*) else image
VARBINARY(*n*)	if (n <= 255) varbinary(*n*) else image
LONG BINARY	image
DECIMAL(*precision*, *scale*)	decimal(*precision*, *scale*)
NUMERIC(*precision*, *scale*)	numeric(*precision*, *scale*)
SMALLMONEY	smallmoney
MONEY	money
REAL	real
DOUBLE	float
FLOAT(*n*)	float(*n*)
DATE	datetime
TIME	datetime
SMALLDATETIME	smalldatetime
DATETIME	datetime
TIMESTAMP	datetime
TIMESTAMP WITH TIMEZONE	varchar(254)

SQL Anywhere データ型	Microsoft SQLServer のデフォルトデータ型
XML	xml
ST_GEOMETRY	image
UNIQUEIDENTIFIER	binary(16)

サーバクラス MYSQLODBC

CREATE TABLE 文を実行するときに、SQL Anywhere は、次のデータ型変換を使用して、データ型を対応する MySQL のデータ型に自動的に変換します。

SQL Anywhere データ型	MySQL のデフォルトデータ型
BIT	bit(1)
VARBIT(*n*)	if (n <= 4000) varbinary(*n*) else longblob
LONG VARBIT	longblob
TINYINT	tinyint unsigned
SMALLINT	smallint
INTEGER	int
BIGINT	bigint
UNSIGNED TINYINT	tinyint unsigned
UNSIGNED SMALLINT	int
UNSIGNED INTEGER	bigint
UNSIGNED BIGINT	decimal(20,0)
CHAR(*n*)	if (n < 255) char(*n*) else if (n <= 4000) varchar(*n*) else longtext
VARCHAR(*n*)	if (n <= 4000) varchar(*n*) else longtext
LONG VARCHAR	longtext
NCHAR(*n*)	if (n < 255) national character(*n*) else if (n <= 4000) national character varying(*n*) else longtext

SQL Anywhere データ型	MySQL のデフォルトデータ型
NVARCHAR(*n*)	if (n <= 4000) national character varying(*n*) else longtext
LONG NVARCHAR	longtext
BINARY(*n*)	if (n <= 4000) varbinary(*n*) else longblob
VARBINARY(*n*)	if (n <= 4000) varbinary(*n*) else longblob
LONG BINARY	longblob
DECIMAL(*precision*, *scale*)	decimal(*precision*, *scale*)
NUMERIC(*precision*, *scale*)	decimal(*precision*, *scale*)
SMALLMONEY	decimal(10,4)
MONEY	decimal(19,4)
REAL	real
DOUBLE	float
FLOAT(*n*)	float(*n*)
DATE	date
TIME	time
TIMESTAMP	datetime
TIMESTAMP WITH TIMEZONE	varchar(254)
XML	longblob
ST_GEOMETRY	longblob
UNIQUEIDENTIFIER	varbinary(16)

例

CREATE SERVER 文の USING 句で接続文字列を指定して、MySQL データベースに接続します。

```
CREATE SERVER TestMySQL
CLASS 'MYSQLODBC'
USING 'DRIVER=MySQL ODBC 5.1
Driver;DATABASE=mydatabase;SERVER=mySQLHost;UID=me;PWD=secret'
```

サーバクラス ODBC

独自のサーバクラスを持たない ODBC データソースでは、サーバクラス **ODBC** を使用します。任意の ODBC ドライバを使用できます。iAnywhere は次の ODBC データソースの使用を確認しています。

- 「Microsoft Excel (Microsoft 3.51.171300)」
- 「Microsoft FoxPro (Microsoft 3.51.171300)」
- 「Lotus Notes SQL」

最新バージョンの Microsoft ODBC ドライバは、Microsoft Data Access Components (MDAC) とともに、Microsoft ダウンロードセンタからダウンロードできます。MDAC 2.0 には、上に示すバージョンの Microsoft ドライバが含まれています。

Microsoft Excel (Microsoft 3.51.171300)

Excel では、それぞれの Excel ブックはいくつかのテーブルを持つデータベースであると、論理的に考えられます。テーブルはブック内のシートにマッピングされます。ODBC ドライバマネージャで ODBC データソース名を設定する場合は、そのデータソースに関連付けられたデフォルトのブック名を指定します。ただし、CREATE TABLE 文を実行する場合には、デフォルトを無効にしてロケーションの文字列にブック名を指定できます。これによって、1 つの ODBC DSN を使用してすべての Excel ブックにアクセスできます。

Microsoft Excel ODBC ドライバに接続する excel という名前のリモートサーバを作成します。

```
CREATE SERVER excel
CLASS 'ODBC'
USING 'DRIVER=Microsoft Excel Driver (*.xls);DBQ=d:¥¥work1.xls;READONLY=0;DriverID=790'
```

mywork というシート (テーブル) を持つ *work1.xls* というブックを作成するには、次のようにします。

```
CREATE TABLE mywork (a int, b char(20))
AT 'excel;d:¥¥work1.xls;;mywork';
```

2 番目のシート (テーブル) を作成するには、次のような文を実行します。

```
CREATE TABLE mywork2 (x float, y int)
AT 'excel;d:¥¥work1.xls;;mywork2';
```

CREATE EXISTING 文を使用して、既存のシートを SQL Anywhere にインポートできます。ここでは、シートの最初のローには、カラム名が入ることを前提としています。

```
CREATE EXISTING TABLE mywork
AT'excel;d:¥¥work1;;mywork';
```

SQL Anywhere によって、テーブルが見つからないと表示された場合は、マッピングするカラムとローの範囲を明示的に指定する必要があります。次に例を示します。

```
CREATE EXISTING TABLE mywork
AT 'excel;d:¥¥work1;;mywork$';
```

シート名に $ を追加すると、ワークシート全体が選択されることを示します。

AT で指定するロケーションの文字列で、フィールドセパレータとしてピリオドの代わりにセミコロンが使用されていることに注意してください。これは、ファイル名にピリオドが使用されるためです。Excel では所有者名フィールドをサポートしないので、これはブランクのままにしてください。

削除はサポートされていません。また、Excel ドライバは位置付け更新をサポートしないため、更新も可能でない場合があります。

例

次の文は、ODBC DSN を使用して Excel ワークブック *LogFile.xlsx* にアクセスし、そのシートを SQL Anywhere にインポートする、TestExcel という名前のデータベースサーバを作成します。

```
CREATE SERVER TestExcel
CLASS 'ODBC'
USING 'DRIVER=Microsoft Excel Driver (*.xls);DBQ=c:¥¥temp¥
¥LogFile.xlsx;READONLY=0;DriverID=790'

CREATE EXISTING TABLE MyWorkbook
AT 'TestExcel;c:¥¥temp¥¥LogFile.xlsx;;Logfile$';

SELECT * FROM MyWorkbook;
```

Microsoft FoxPro (Microsoft 3.51.171300)

FoxPro の各テーブルを 1 つの FoxPro データベースファイル (*.dbc*) にまとめて格納したり、各テーブルをそれぞれの *.dbf* ファイルに格納することができます。*.dbf* ファイルを使用するときは、ファイル名がロケーションの文字列に入っていることを確認してください。入っていない場合は、SQL Anywhere が起動されたディレクトリが使用されます。

```
CREATE TABLE fox1 (a int, b char(20))
AT 'foxpro;d:¥¥pcdb;;fox1';
```

この文は、ODBC ドライバマネージャで **Free Table Directory** オプションが選択されている場合、*d:¥pcdb¥fox1.dbf* というファイルを作成します。

Lotus Notes SQL

このドライバを取得するには、Lotus NotesSQL の Web サイト (http://www.ibm.com/developerworks/lotus/products/notesdomino/notessql/) を参照してください。Notes データがリレーショナルテーブルにどのようにマッピングされるかについては、ドライバに付属のマニュアルを参照してください。SQL Anywhere テーブルは、Notes フォームに簡単にマッピングできます。

Lotus Notes の連絡先にアクセスするよう SQL Anywhere を設定する方法を次に示します。

- Lotus Notes プログラムフォルダがパスに含まれていることを確認します (C:¥Program Files (x86)¥IBM¥Lotus¥Notes など)。

- NotesSQL ODBC ドライバを使用して 32 ビットの ODBC データソースを作成します。この例では *names.nsf* データベースを使用します。**特殊文字のマップ**オプションを有効にしてください。この例では、**データソース名**は *my_notes_dsn* です。

●32 ビットのデータベースサーバに接続された Interactive SQL を使用して、リモートデータアクセスサーバを作成します。次に例を示します。

```
CREATE SERVER NotesContacts
CLASS 'ODBC'
USING 'my_notes_dsn';
```

●Lotus Notes サーバ用の外部ログインを作成します。次に例を示します。

```
CREATE EXTERNLOGIN "DBA" TO "NotesContacts"
REMOTE LOGIN 'John Doe/SYBASE' IDENTIFIED BY 'MyNotesPassword';
```

●Person フォームの一部のカラムを SQL Anywhere テーブルにマッピングします。

```
CREATE EXISTING TABLE PersonDetails
( DisplayName CHAR(254),
  DisplayMailAddress CHAR(254),
  JobTitle CHAR(254),
  CompanyName CHAR(254),
  Department CHAR(254),
  Location CHAR(254),
  OfficePhoneNumber CHAR(254) )
AT 'NotesContacts...Person';
```

●次のようにテーブルを問い合わせます。

```
SELECT * FROM PersonDetails
WHERE Location LIKE 'Waterloo%';
```

サーバクラス ORAODBC

サーバクラスが ORAODBC のリモートサーバは、Oracle Database バージョン 8.0 以降です。

注意

●iAnywhere は、Oracle Database バージョン 8.0.03 の ODBC ドライバの使用を確認しています。この製品の説明に従って、ODBC 構成の設定とテストを実行してください。

●以下は、myora という Oracle Database サーバの CREATE EXISTING TABLE 文の例です。

```
CREATE EXISTING TABLE employees
AT 'myora.database.owner.employees';
```

データ型変換：Oracle Database

CREATE TABLE 文を実行するときに、SQL Anywhere は、次のデータ型変換を使用して、データ型を対応する Oracle Database のデータ型に自動的に変換します。

SQL Anywhere データ型	Oracle Database のデータ型
BIT	number(1,0)
VARBIT(*n*)	if (n <= 255) raw(*n*) else long raw

SQL Anywhere データ型	Oracle Database のデータ型
LONG VARBIT	long raw
TINYINT	number(3,0)
SMALLINT	number(5,0)
INTEGER	number(11,0)
BIGINT	number(20,0)
UNSIGNED TINYINT	number(3,0)
UNSIGNED SMALLINT	number(5,0)
UNSIGNED INTEGER	number(11,0)
UNSIGNED BIGINT	number(20,0)
CHAR(*n*)	if (n <= 255) char(*n*) else long
VARCHAR(*n*)	if (n <= 2000) varchar(*n*) else long
LONG VARCHAR	long
NCHAR(*n*)	if (n <= 255) nchar(*n*) else nclob
NVARCHAR(*n*)	if (n <= 2000) nvarchar(*n*) else nclob
LONG NVARCHAR	nclob
BINARY(*n*)	if (n > 255) long raw else raw(*n*)
VARBINARY(*n*)	if (n > 255) long raw else raw(*n*)
LONG BINARY	long raw
DECIMAL(*precision*, *scale*)	number(*precision*, *scale*)
NUMERIC(*precision*, *scale*)	number(*precision*, *scale*)
SMALLMONEY	numeric(13,4)
MONEY	number(19,4)
REAL	real
DOUBLE	float

SQL Anywhere データ型	Oracle Database のデータ型
FLOAT(*n*)	float
DATE	date
TIME	date
TIMESTAMP	date
TIMESTAMP WITH TIMEZONE	varchar(254)
XML	long raw
ST_GEOMETRY	long raw
UNIQUEIDENTIFIER	raw(16)

例

CREATE SERVER 文の USING 句で接続文字列を指定して、Oracle データベースに接続します。

```
CREATE SERVER TestOracle
CLASS 'ORAODBC'
USING 'DRIVER=Oracle ODBC Driver;DBQ=mydatabase;UID=username;PWD=password'
```

データ整合性

データに整合性があるということは、データが有効、つまり適切であり正確で、データベースの関係構造が保たれていることを意味します。参照整合性制約を使うと、データベースの関係構造を確保できます。また、この規則によって、各テーブル間でデータの一貫性を保つことができます。データベースに整合性制約を構築すると、データの一貫性を確実に保つことができます。

参照整合性を確保する方法はいくつかあります。たとえば、テーブルとカラムに制約と検査制約を課すことで、各エントリが正しいことを保証できます。また、適切なデータ型を使ってカラムのプロパティを設定したり、特別なデフォルト値を設定したりすることもできます。

SQL Anywhere では、データベースへのデータの入力方法を細かく規定するためのストアドプロシージャをサポートしています。また、トリガも作成できます。トリガは特殊なストアドプロシージャで、特定のカラムの更新など、特定のアクションが行われると自動的に実行されます。

参照

●「ストアドプロシージャ、トリガ、バッチ、ユーザ定義関数」77 ページ

データが有効でなくなる状況

適切な検査が行われないと、データベース内のデータが有効でなくなる可能性があります。それぞれのケースは、この項で説明されている機能を使用して回避できます。

正しくない情報

●オペレータが売り上げトランザクションの日付を間違って入力した。

●オペレータが一桁間違えて入力し、社員の給料が 10 分の 1 になった。

データの重複

●組織のデータベースの Departments テーブルに、2 人の異なる従業員が同じ部署 (DepartmentID は 200) を新しく追加した。

失われた外部キー関係

●300 という DepartmentID で識別される部署が閉鎖になったのに、1 人の従業員レコードだけが他の部署へ移されていない。

整合性制約

データベースのデータの有効性を確保するには、データが有効かどうかを判断するための検査と、データが従う規則 (ビジネスルール) が必要です。通常、ビジネスルールは、検査制約、ユーザ定義データ型、適切なトランザクションを使用することによって実装されます。

データベースに組み込まれる制約は、クライアントアプリケーション内に組み込まれる制約や、データベースのユーザに指示される制約よりも高い信頼性を備えています。データベースに組み込まれた制約はデータベースの定義の一部であり、すべてのアプリケーションに対して常に適

用されるからです。データベース内に制約が設定されると、それ以降に発生するすべてのデータベースとの対話に対してその制約が適用されます。

これに対して、クライアントアプリケーションに組み込まれた制約は、アプリケーションが変更されるたびに無効となる可能性があります。また、複数のアプリケーションに組み込んだり、1 つのアプリケーションの複数箇所に組み込んだりする必要があります。

データベースの内容が変更される状況

クライアントアプリケーションから SQL 文が送られてくると、データベーステーブル内の情報に変更が生じます。データベースの情報を実際に変更する SQL 文は、それほど多くはありません。次の文で変更が行われます。

- UPDATE 文を使って、テーブルのローを更新する。
- DELETE 文を使って、テーブルのローを削除する。
- INSERT 文を使って、テーブルに新しいローを挿入する。

データの整合性を維持するためのツール

データ整合性を確保するため、デフォルト値、データ制約、データベースの参照構造を保つための制約を利用できます。

デフォルト値

各エントリのデータの信頼性を高めるために、カラムにデフォルト値を設定できます。次に例を示します。

- すべてのユーザ、またはクライアントアプリケーションによるトランザクション日を記録するデフォルト値を、CURRENT DATE にする。
- 新しいローの追加以外で、特にユーザが操作しなくてもカラムのデフォルト値を自動的に 1 ずつ大きくしていく。このようにすると、(発注書などの) 項の値をユニークに、また連続の値にできます。

プライマリキー

プライマリキーによって、指定されたテーブルのすべてのローをテーブル内でユニークに識別できることが保証されます。

テーブルとカラム制約

以下の制約によって、データベース内のデータ構造が維持され、リレーショナルデータベースのテーブル間の関連が定義されます。

- **参照整合性制約**　データ整合性は「RI 制約」 (参照整合性制約の場合) とも呼ばれ、参照整合性制約を使用することによって維持することもできます。RI 制約は、カラムやテーブルに設

定されるデータルールであり、データの形式を制御します。RI 制約は、リレーショナルデータベースのテーブル間の関係を定義します。

- **NOT NULL 制約**　NOT NULL 値制約を適用すると、カラムに NULL 値が入力されるのを防ぐことができる。

- **CHECK 制約**　検査制約をカラムに適用すると、カラム内のデータが常に一定の条件を満たすようにできる。たとえば、Salary (給与) カラムに上限と下限を設定して入力エラーを防げます。

  検査制約を使って、それぞれのカラム値の差を検査できる。たとえば、図書館データベースで、DateReturned (返却日) エントリが DateBorrowed (貸出日) エントリよりも必ず後になるように指定できます。

カラム制約はドメインから継承できます。

高度な整合性規則のためのトリガ

「トリガ」はデータベースに格納されたストアドプロシージャの一種で、特定のテーブル情報が修正されると自動的に実行されます。トリガはデータベース管理者や開発者にとって、データの信頼性を維持するための強力な手段です。データ整合性の確保にはトリガも使用できます。トリガを使うと検査条件をより高度な形で適用できる。

参照

- 「カラムデフォルト」836 ページ
- 「プライマリキー」17 ページ
- 「エンティティ整合性と参照整合性」850 ページ
- 「テーブルとカラム制約」843 ページ
- 「ストアドプロシージャ、トリガ、バッチ、ユーザ定義関数」77 ページ

整合性制約を実装するための SQL 文

整合性制約を実装するための SQL 文を次に示します。

- **CREATE TABLE 文**　テーブルの作成時の整合性制約を実装します。

- **ALTER TABLE 文**　既存のテーブルに対して、整合性制約の追加や制約の修正を行います。

- **CREATE TRIGGER 文**　より複雑なビジネスルールを確保するトリガを作成します。

- **CREATE DOMAIN 文**　ユーザ定義のデータ型を作成します。データ型の定義には制約を含めることができます。

参照

- 「SQL 文」『SQL Anywhere サーバ SQL リファレンス』

カラムデフォルト

カラムデフォルトを設定すると、データベーステーブルに新しいローが追加された場合、常に指定された値が一定のカラムへ自動的に設定されます。デフォルト値の設定には、クライアントアプリケーション側からの処理は特に必要ありません。一方、クライアントアプリケーションがカラムに値を設定すると、その値がデフォルト値に上書きされます。

カラムデフォルトは、ローが挿入された日付や時刻、入力するユーザのユーザ ID などの情報を、カラムにすばやく自動的に入力する場合に便利です。カラムデフォルトはデータの整合性を向上させますが、確実なものではありません。クライアントアプリケーションはデフォルト値を自由に上書きできます。

@ で始まる変数を使用してデフォルト値が定義された場合、デフォルトとして使用される値は、DML 文または LOAD 文が実行されるときの変数の値です。

サポートされているデフォルト値

SQL では、次のデフォルト値をサポートしています。

- CREATE TABLE 文または ALTER TABLE 文で指定した文字列。
- CREATE TABLE 文または ALTER TABLE 文で指定した数値。
- AUTOINCREMENT：自動的に増分される数値。既存のカラムの最大値に、自動的に 1 を加えます。
- GLOBAL AUTOINCREMENT：複数のデータベースにユニークなプライマリキーを保証します。
- NEWID 関数を使用して生成されるユニバーサルユニーク識別子 (UUID)。
- CURRENT DATE、TIME、または TIMESTAMP。
- データベースユーザの CURRENT USER。
- NULL 値。
- データベースオブジェクトを参照していない定数式。

カラムデフォルトの作成

カラムデフォルトは、テーブルの作成時に CREATE TABLE 文を使って作成するか、後で ALTER TABLE 文を使って追加します。

例

次に示す文は、SalesOrders テーブルの ID カラムにデフォルト設定を追加して、クライアントアプリケーションが指定しないかぎり、自動的に 1 ずつ値を増加させます。SQL Anywhere サンプルデータベースでは、このカラムは AUTOINCREMENT に設定済みです。

```
ALTER TABLE SalesOrders
ALTER ID DEFAULT AUTOINCREMENT;
```

参照

- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

カラムデフォルトの変更

カラムデフォルトは、作成と同様に、ALTER TABLE 文を使って修正または削除できます。

次に示す文は、OrderDate というカラムのデフォルト値を、CURRENT DATE に変更します。

```
ALTER TABLE SalesOrders
ALTER OrderDate DEFAULT CURRENT DATE;
```

削除する場合は、カラムデフォルトに NULL を設定します。次に示す文は、OrderDate カラムからデフォルトを削除します。

```
ALTER TABLE SalesOrders
ALTER OrderDate DEFAULT NULL;
```

カラムデフォルトの設定

Sybase Central で、カラムデフォルトの追加、修正、削除を行うには、**[カラムプロパティ]** ウィンドウの **[値]** タブを使用します。

前提条件

そのカラムが属しているテーブルの所有者であるか、または次のいずれかの権限を持っていることが必要です。

- そのテーブルに対する ALTER 権限
- ALTER ANY TABLE システム権限
- ALTER ANY OBJECT システム権限

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[テーブル]** をダブルクリックします。
3. テーブルをダブルクリックします。
4. **[カラム]** タブをクリックします。
5. カラムを右クリックして、**[プロパティ]** をクリックします。
6. **[値]** タブをクリックします。

7. 必要に応じてカラムデフォルトを変更します。

結果

カラムのプロパティが変更されます。

参照

- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

現在の日付／時刻デフォルト

データ型が DATE、TIME、TIMESTAMP のカラムには、CURRENT DATE、CURRENT TIME、CURRENT TIMESTAMP をデフォルトとして使用できます。指定するデフォルト値は、カラムのデータ型に一致させてください。

CURRENT DATE をデフォルトにすると便利な例

次のような場合、CURRENT DATE をデフォルト値として設定すると便利です。

- 顧客データベースで、電話があった日付を記録する。
- 売り上げ入力データベースで、注文の日付を記録する。
- 図書館データベースで、会員が本を借りた日を記録する。

CURRENT TIMESTAMP

CURRENT TIMESTAMP は、CURRENT DATE と同じように使えるデフォルト値で、さらに細かい情報を記録できます。たとえば、交渉管理アプリケーションのユーザは、一日に何度も同一の顧客とやり取りすることがあります。CURRENT TIMESTAMP のデフォルトを使用すると、それぞれの連絡先が区別しやすくなります。

CURRENT TIMESTAMP は日付と時刻を 100 万分の 1 秒単位で記録するので、データベースに記録されている各イベントの順序が重要である場合にも便利です。

DEFAULT TIMESTAMP

DEFAULT TIMESTAMP は、テーブル内の各ローが最後に修正された日時を示します。カラムの宣言に DEFAULT TIMESTAMP が指定されている場合は、ローを挿入するとタイムスタンプのデフォルト値が割り付けられます。この値は、ローが更新されるたびに現在の日付と時刻に基づいて更新されます。挿入されたローにタイムスタンプのデフォルト値を割り付け、そのローが更新されてもタイムスタンプを更新しない場合は、DEFAULT TIMESTAMP の代わりに DEFAULT CURRENT TIMESTAMP を使用します。

参照

- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「SQL データ型」『SQL Anywhere サーバ SQL リファレンス』

ユーザ ID デフォルト

DEFAULT USER をカラムのデフォルト値として指定しておくと、そのデータをデータベースに入力したユーザを確実に特定できます。たとえば、売り上げ歩合制の営業員がデータベースを使用する場合、このような情報が必要になります。

ユーザ ID デフォルトをテーブルのプライマリキーに設定すると、不定期に接続するユーザがいる場合に情報更新の競合を防ぐことができます。このようなユーザは、データベースから必要な部分を携帯端末にコピーして、マルチユーザデータベースに接続していない状態でデータを修正し、後でサーバにアクセスしてトランザクションログを送ることができます。

LAST USER 特別値は、ローを最後に更新したユーザの名前を返します。DEFAULT TIMESTAMP と結合すると、LAST USER のデフォルト値を使用して、ローを最後に変更したユーザと日時の両方を記録できます (ただし、別々のローに記録されます)。

参照

- 「LAST USER 特別値」『SQL Anywhere サーバ SQL リファレンス』

オートインクリメントデフォルト

オートインクリメントデフォルトは、値それ自体には意味がない数値データフィールドで役立ちます。この機能は、新しく作成されたローの該当するカラムに、カラムの他の値よりも大きいユニークな値を割り当てます。注文伝票番号の記録、顧客からの問い合わせの電話の識別など、番号自体に意味がないエントリ番号のカラムに、オートインクリメントを適用できます。

AUTOINCREMENT カラムは、通常はプライマリキーカラム、つまりユニークな値を保持するように制約されたカラムになります。

AUTOINCREMENT カラムへ直前に追加された値は、グローバル変数 @@identity を使って取得できます。

AUTOINCREMENT と負の値

AUTOINCREMENT は正の整数を想定しています。

テーブルが作成されたときの AUTOINCREMENT の初期値は 0 です。意図的に負の値が設定されたカラムが挿入されると、初期値 0 はそのカラムの最大値としてそのまま残されます。特に初期値が設定されていない挿入に対しては、オートインクリメントにより 1 が設定され、それ以降も必ず正の値が設定されます。

AUTOINCREMENT と IDENTITY カラム

Transact-SQL アプリケーションでは、オートインクリメントのデフォルト値が設定されるカラムを IDENTITY カラムと呼びます。

参照

- 「AUTOINCREMENT カラムのあるテーブルの再ロード」『SQL Anywhere 16 変更点とアップグレード』
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「GLOBAL AUTOINCREMENT デフォルト」840 ページ
- 「シーケンス値と AUTOINCREMENT 値からの選択」929 ページ
- 「GLOBAL AUTOINCREMENT カラム」『SQL Remote』
- 「sa_reset_identity システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「@@identity グローバル変数」『SQL Anywhere サーバ SQL リファレンス』
- 「エンティティ整合性」850 ページ
- 「特殊な IDENTITY カラム」654 ページ

GLOBAL AUTOINCREMENT デフォルト

GLOBAL AUTOINCREMENT デフォルトは、SQL Remote レプリケーション環境または Mobile Link 同期環境で複数のデータベースを使用するときのために用意されたものです。複数のデータベースにユニークなプライマリキーを保証します。

このオプションは AUTOINCREMENT と同じですが、ドメインはパーティションに分割されます。各分割には同じ数の値が含まれます。データベースの各コピーにユニークなグローバルデータベース ID 番号を割り当てます。SQL Anywhere では、データベースのデフォルト値は、そのデータベース番号でユニークに識別された分割からのみ設定されます。

この分割サイズには任意の正の整数を設定できますが、通常、分割サイズは、サイズの値がすべての分割で不足しないように選択されます。

カラムの型が BIGINT または UNSIGNED BIGINT である場合、デフォルトの分割サイズは 2^{32} = 4294967296 です。それ以外の型のカラムの場合、デフォルトの分割サイズは 2^{16} = 65536 です。特に、カラムの型が INT または BIGINT ではない場合は、これらのデフォルト値が適切ではないことがあるため、分割サイズを明示的に指定するのが最も賢明です。

このオプションを使用する場合、各データベース内のパブリックオプション global_database_id は、ユニークな正の整数に設定します。この値は、データベースをユニークに識別し、デフォルト値の割り当て元の分割を示します。使用できる値の範囲は $np + 1$ ～ $(n+1)p$ です。ここで、n はパブリックオプション global_database_id の値を表し、p は分割サイズを表します。たとえば、分割サイズを 1000、global_database_id を 3 に設定すると、範囲は 3001 ～4000 になります。

前の値が $(n + 1)p$ 未満であれば、このカラム内でこれまで使用した最大値より 1 大きい値が次のデフォルト値になります。カラムに値が含まれていない場合、最初のデフォルト値は $np + 1$ になります。デフォルトのカラム値は、現在の分割以外のカラムの値の影響を受けません。つまり、$np + 1$ より小さいか $p(n + 1)$ より大きい数には影響されません。Mobile Link 同期を介して別のデータベースからレプリケートされた場合に、このような値が存在する可能性があります。

public オプション global_database_id は、負の値に設定できないため、選択された値は常に正になります。ID 番号の最大値を制限するのは、カラムのデータ型と分割サイズだけです。

public オプション global_database_id がデフォルト値の 2147483647 に設定されると、NULL 値がカラムに挿入されます。NULL 値が許可されていない場合に、ローの挿入を試みるとエラーが発

生します。たとえば、テーブルのプライマリキーにカラムが含まれている場合に、この状況が発生します。

デフォルトの NULL 値は、分割で値が不足したときにも生成されます。この場合には、別の分割からデフォルト値を選択できるように、データベースに global_database_id の新しい値を割り当ててください。カラムで NULL が許可されていない場合に NULL 値を挿入しようとすると、エラーが発生します。未使用の値が残り少ないことを検出し、このような状態を処理するには、GlobalAutoincrement タイプのイベントを作成します。

GLOBAL AUTOINCREMENT カラムは、通常はプライマリキーカラム、つまりユニークな値を保持するように制約されたカラムになります。

これ以外の場合にも GLOBAL AUTOINCREMENT のデフォルトを適用できますが、データベースのパフォーマンスが低下することがあります。たとえば、各カラムの次の値が 64 ビットの符号付き整数として格納されている場合に、2^{31} - 1 より大きい値または大きい double または numeric の値が使用されると、オーバーフローが発生して負の値になることがあります。

AUTOINCREMENT カラムへ直前に追加された値は、グローバル変数 @@identity を使って取得できます。

参照

- 「GLOBAL AUTOINCREMENT」『Mobile Link サーバ管理』
- 「エンティティ整合性」850 ページ
- 「@@identity グローバル変数」『SQL Anywhere サーバ SQL リファレンス』
- 「イベント」『SQL Anywhere サーバ データベース管理』
- 「GLOBAL AUTOINCREMENT カラム」『SQL Remote』
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「AUTOINCREMENT カラムのあるテーブルの再ロード」『SQL Anywhere 16 変更点とアップグレード』
- 「オートインクリメントデフォルト」839 ページ
- 「シーケンス値と AUTOINCREMENT 値からの選択」929 ページ
- 「sa_reset_identity システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

NEWID デフォルト

ユニバーサルユニーク識別子 (UUID) を使用して、テーブルのユニークなローを識別できます。UUID は、グローバルユニーク識別子 (GUID) とも呼ばれます。この値は、1 台のコンピュータで生成された値が、他のコンピュータで生成された値と一致しないように生成されます。したがって、これらの値は、レプリケーション環境と同期環境でキーとして使用できます。

プライマリキーとして使用する場合、GLOBAL AUTOINCREMENT 値と比べると、UUID 値にはいくつかのトレードオフがあります。次に例を示します。

- 各リモートデータベースにユニークなデータベース ID を割り当てる必要がないので、GLOBAL AUTOINCREMENT より UUID の方が簡単に設定できます。システムのデータベース数や個々のテーブルのロー数を考慮する必要もありません。抽出ユーティリティ (dbxtract) を使用してデータベース ID の割り当てを処理できます。GLOBAL AUTOINCREMENT では

通常、BIGINT データ型を使用する場合はこの点を考慮する必要はありませんが、BIGINT より小さいデータ型を使用する場合は考慮する必要があります。

- UUID 値は GLOBAL AUTOINCREMENT に必要な値よりかなり大きいため、プライマリテーブルと外部テーブルの両方でより多くのテーブル領域が必要です。また、UUID を使用すると、これらのカラムのインデックスの効率も悪くなります。つまり、GLOBAL AUTOINCREMENT の方がパフォーマンスに優れています。

- UUID には暗黙的な順序付けがありません。たとえば、A と B が UUID 値で、A が B よりも大きい場合に、A と B が同じコンピュータ上で生成されたとしても、A が B の後で生成されたとはかぎりません。暗黙的な順序付けが必要な場合は、追加のカラムとインデックスが必要になります。

参照

- 「NEWID 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』
- 「UNIQUEIDENTIFIER データ型」『SQL Anywhere サーバ SQL リファレンス』

NULL デフォルト

NULL 値を許容するカラムに NULL デフォルトを指定しても、デフォルトを指定しなくても、まったく同じです。ローを挿入するクライアントが特に値を指定しなければ、そのローは自動的に NULL 値が設定されます。

NULL デフォルトは、カラムの入力を省略できる場合、または入力データが入手できないことがある場合に使います。

参照

- 「NULL 特別値」『SQL Anywhere サーバ SQL リファレンス』

文字列と数値デフォルト

カラムが文字列または数値のデータ型であれば、特定の文字列または数値をデフォルトに指定できます。指定するデフォルト値は、カラムのデータ型に変換可能なものにしてください。

文字列と数値デフォルトは、カラムに同じデータを入力することが多いときに便利です。たとえば、ある会社に本社 city_1 と小規模の事業所 city_2 の 2 つのオフィスがある場合、所在地カラムのデフォルトを city_1 にしておけば入力が簡単になります。

定数式デフォルト

定数式もデフォルト値として使用できます。ただし、データベースオブジェクトを参照していない場合にかぎります。たとえば次の式では、本日より 15 日後というエントリを、デフォルト値としてカラムに設定できます。

```
... DEFAULT ( DATEADD( day, 15, GETDATE() ) );
```

テーブルとカラム制約

基本的なテーブル構造 (カラムの番号、名前、データ型、テーブルの名前とロケーション) 以外にも、CREATE TABLE 文と ALTER TABLE 文を使って、データの整合性を保つためのさまざまなテーブル属性値を指定できます。制約を使って、カラムに含まれる値や、異なるカラムに含まれるそれぞれの値の関係に制限を課すことができます。制約は、テーブル全体、または個別のカラムに対して適用できます。

この項では、制約を使用してテーブルに正しい値が確実に入力されるようにする方法について説明します。

カラムに対する検査制約

カラムの値がある基準を満たすようにするため、検査条件を使用します。これらの条件は、データが正しいかどうかを確認するためのものであったり、企業のポリシーや内部規定などが反映される厳密なものであったりします。個々のカラムに対する検査条件は、カラム内の値を一定の範囲に収める必要がある場合に便利です。

検査条件が使用されると、その後はローの修正前に値が条件に対して評価されます。検査制約が設定された値を更新すると、その値の制約が検査されるだけでなく、残りのローの制約も検査されます。

カラムに対する検査制約に変数は使用できません。カラム検査制約内で @ で始まる文字列は、制約が有効にされているカラムの名前で置き換えられます。

カラムのデータ型がドメインの場合、カラムはドメインに定義されている検査制約を継承します。

注意
カラムの検査は、FALSE が返された場合にエラーになります。UNKNOWN が返されても、動作は TRUE が返される場合と同じで、その値が受け入れられます。

参照

- 「ドメインから継承されるカラム検査制約」845 ページ
- 「探索条件」『SQL Anywhere サーバ SQL リファレンス』

例 1

データのフォーマットを規定します。たとえば、テーブルに電話番号のカラムがあるとして、その電話番号カラムが同じフォーマットで入力されるようにします。北米地域の電話番号に対する制約の例を次に示します。

```
ALTER TABLE Customers
ALTER Phone
CHECK ( Phone LIKE '(___) ___-____' );
```

この検査条件が使用されると、たとえば Phone の値を 9835 に設定しようとしても、変更されません。

例 2

エントリに入力されるデータが、あらかじめ決められたいくつかの値のいずれかになるように設定できます。たとえば、City カラムに、その会社の事業所の所在地など、いくつかの許可された都市のいずれかしか入力できないようにするには、次のように制約を使用します。

```
ALTER TABLE Customers
ALTER City
CHECK ( City IN ( 'city_1', 'city_2', 'city_3' ) );
```

データベースの作成時に特に指定をしなかった場合、文字列の比較において大文字と小文字は区別されません。

例 3

日付や数値が一定の範囲内に収まるように設定できます。従業員の入社日 StartDate を会社の創立から現在までの日付にする文の例を示します。StartDate がこの 2 つの日付の間になるように、次の制約を使用します。

```
ALTER TABLE Employees
ALTER StartDate
CHECK ( StartDate BETWEEN '1983/06/27'
            AND CURRENT DATE );
```

日付形式はいくつかの種類が用意されています。ここで使用した YYYY/MM/DD 形式は、現在のオプション設定に関係なく必ず認識される特長があります。

テーブルに対する検査制約

テーブルに対して検査条件を適用すると、たとえば単一のローで入力または修正された 2 つの値の関係の正当性を保証できます。

制約に名前を付けると、制約は個別にシステムテーブル内に格納され、個別に置換、削除できます。この方法の方が柔軟性が高いため、検査制約に名前を付けるか、できるかぎり個別にカラム制約を使用することをおすすめします。

たとえば Employees テーブルに制約を追加して、TerminationDate が常に StartDate と同じかそれ以降になるようにすることができます。

```
ALTER TABLE Employees
  ADD CONSTRAINT valid_term_date
  CHECK( TerminationDate >= StartDate );
```

テーブル検査制約内には変数を指定できますが、変数の名前は @ で始まるようにします。使用される値は、DML 文または LOAD 文が実行されるときの変数の値です。

参照

- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

ドメインから継承されるカラム検査制約

ドメインに検査制約を付加できます。そのドメインで定義されたカラムには、検査制約も継承されます。そのカラムに明示的に検査制約を設定した場合は、ドメインよりも優先されます。たとえば、このドメイン定義における CHECK 句は、カラムに挿入される値は正の整数だけであることを要求します。

```
CREATE DATATYPE positive_integer INT
CHECK ( @col > 0 );
```

positive_integer ドメインを使用して定義されたカラムでは、検査制約が明示的に指定されていないかぎり、正の整数のみが受け入れられます。@ 記号のプレフィクスを持つ変数はすべて、検査制約が評価される時点でそのカラムの名前に置き換えられるので、@ 記号のプレフィクスさえあれば @col 以外の変数名を使用しても問題ありません。

ALTER TABLE 文と DELETE CHECK 句を組み合せると、ドメインから継承されたものを含め、テーブル定義からすべての検査制約を削除できます。

ドメインでカラムが定義されたあとに、ドメイン定義の制約が変更された場合は、そのカラムに変更は適用されません。カラムは作成時にドメインの制約を継承しますが、それ以降は、両者の間に関係はありません。

参照

- 「ドメイン」『SQL Anywhere サーバ SQL リファレンス』
- 「カラムに対する検査制約」843 ページ

制約の管理

Sybase Central では、カラム制約の追加、変更、削除を、テーブルまたは **[カラムプロパティ]** ウィンドウの **[制約]** タブで実行できます。

前提条件

そのカラムが属しているテーブルの所有者であるか、または次のいずれかの権限を持っていることが必要です。

- そのテーブルに対する ALTER 権限
- ALTER ANY TABLE システム権限
- ALTER ANY OBJECT システム権限

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[テーブル]** をダブルクリックします。
3. 変更するテーブルをダブルクリックします。
4. 右ウィンドウ枠で、**[制約]** タブをクリックし、既存の制約を変更するか、新しい制約を追加します。

結果

カラムの制約が表示されます。

次の手順

必要に応じて制約を修正します。

参照

- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

一意性制約の追加

Sybase Central のカラムの一意性制約を作成または削除できます。

前提条件

そのテーブルの所有者であるか、または次のいずれかの権限を持っていることが必要です。

- テーブルに対する ALTER 権限と、ALTER ANY INDEX、COMMENT ANY OBJECT、CREATE ANY INDEX、または CREATE ANY OBJECT システム権限のいずれか
- ALTER ANY TABLE システム権限と、ALTER ANY INDEX、COMMENT ANY OBJECT、CREATE ANY INDEX、または CREATE ANY OBJECT システム権限のいずれか
- ALTER ANY OBJECT システム権限

内容と備考

一意性制約には空間カラムは設定できません。

カラムの場合、一意性制約では、カラム値をユニークにする必要があることを指定します。テーブルの場合、一意性制約では、テーブル内のユニークなローを識別する 1 つ以上のカラムを指定します。テーブル内の異なるローが、指定されているすべてのカラムで同じ値を持つことはできません。1 つのテーブルには、複数の一意性制約を設定できます。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[テーブル]** をダブルクリックします。
3. 変更するテーブルをクリックします。
4. 右ウィンドウ枠で、**[制約]** タブをクリックします。
5. **[制約]** テーブルを右クリックし、**[新規]** » **[一意性制約]** をクリックします。
6. **一意性制約作成ウィザード**の指示に従います。

結果

一意性制約が作成されます。

検査制約の変更と削除の方法

テーブル上の既存の検査制約を変更するには、いくつかの方法があります。

●テーブルまたはカラムに新しい検査制約を追加できます。

●カラムの検査制約を NULL に設定すると削除できます。次に、Customers テーブルの Phone カラムから検査制約を削除する文の例を示します。

```
ALTER TABLE Customers
ALTER Phone CHECK NULL;
```

●検査制約の追加と同じ方法で、カラムの検査制約を置換できます。次に、Customers テーブルの Phone カラムの検査制約を追加または置換する文の例を示します。

```
ALTER TABLE Customers
ALTER Phone
CHECK ( Phone LIKE '___-___-____' );
```

●テーブルに定義された検査制約を変更できます。

○ALTER TABLE と ADD *table-constraint* 句を使って新しい検査制約を追加できます。

○制約名を定義済みの場合は、制約を個別に変更できます。

○制約名を定義していない場合は、ALTER TABLE DELETE CHECK を使って、既存のすべての検査制約 (カラム検査制約、ドメインから継承した検査制約など) を削除してから、新しい検査制約を追加できます。

ALTER TABLE 文で DELETE CHECK 句を使用するには、次のように指定します。

```
ALTER TABLE table-name
DELETE CHECK;
```

Sybase Central では、テーブル検査制約とカラム検査制約の両方を追加、変更、削除できます。

テーブルからカラムを削除しても、そのカラムと関連付けられていた検査制約はテーブル制約から削除されません。制約を削除しないと、テーブルに対してデータの挿入や、単に問い合わせを行っただけでもエラーメッセージが生成されます。

注意
テーブル検査制約は FALSE が返された場合にエラーとなります。UNKNOWN が返されても、動作は TRUE が返される場合と同じで、その値が受け入れられます。

参照

- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「カラム '%1' が見つかりません。」『エラーメッセージ』
- 「制約の管理」845 ページ

ドメイン

「ドメイン」とはユーザ定義データ型のことです。これを他の属性と一緒に使用して、値の許容範囲を制限したりデフォルト値を設定したりできます。ドメインを使うと既存のデータ型を拡張できます。通常、値の許容範囲の制限には検査制約が使われます。さらに、ドメインではデフォルト値を設定できます。値は NULL であってもそうでなくてもかまいません。

独自のドメインを定義することには次のような利点があります。

- 不適切な値が入力された場合の一般的なエラーを防ぐ。ドメインに設定した制約で、値を一定の範囲またはフォーマットに保持させたいすべてのカラムと変数に、ある範囲内またはフォーマットの値だけを保持させることができます。たとえば、あるデータ型によって、データベースに入力されるクレジットカード番号に確実に正しい桁数が入るようにできます。

- アプリケーションやデータベースの構造をわかりやすくする。

- 利便性。たとえば、テーブルの識別子をすべて正の整数にし、デフォルト値としてオートインクリメントにするとします。テーブルを新しく作成するたびにこうした制約を設定するよりも、新しいドメインを定義して識別子がそのドメイン型の値以外をとらないように設定する方が、作業が少なくて済みます。

参照

- 「ドメイン」『SQL Anywhere サーバ SQL リファレンス』

ドメインの作成

ユーザ定義ドメインの作成

前提条件

CREATE DATATYPE または CREATE ANY OBJECT システム権限が必要です。

内容と備考

SQL Anywhere には定義済みドメインがいくつかあります (通貨データドメインの MONEY など)。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で **[ドメイン]** を右クリックし、**[新規]** » **[ドメイン]** をクリックします。

3. **ドメイン作成ウィザード**の指示に従います。

結果

新しいドメインが作成されます。

参照

- 「CREATE DOMAIN 文」『SQL Anywhere サーバ SQL リファレンス』

カラムへのドメインの適用

Sybase Central では、カラムでドメイン (ユーザ定義データ型) を使用するように変更できます。

前提条件

そのカラムが属しているテーブルの所有者であるか、または次のいずれかの権限を持っていることが必要です。

- そのテーブルに対する ALTER 権限
- ALTER ANY TABLE システム権限
- ALTER ANY OBJECT システム権限

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[テーブル]** をダブルクリックします。
3. テーブルをクリックします。
4. 右ウィンドウ枠で、**[カラム]** タブをクリックします。
5. カラムを右クリックし、**[プロパティ]** をクリックします。
6. **[データ型]** タブをクリックし、**[ドメイン]** をクリックします。
7. **[ドメイン]** リストからドメインを選択します。
8. **[OK]** をクリックします。

結果

カラムは選択されたドメインを使用します。

参照

- 「ドメイン」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

ドメインの削除

Sybase Central を使用してユーザ定義データ型 (ドメイン) を削除できます。

前提条件

DROP DATATYPE または DROP ANY OBJECT システム権限が必要です。

データベース内の変数やカラムによって使用されているドメインは削除できません。そのドメインを使用しているカラムや変数をすべて削除または変更してから、ドメインを削除してください。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[ドメイン]** をダブルクリックします。
3. 右ウィンドウ枠で、ドメインを右クリックし、**[削除]** をクリックします。
4. **[はい]** をクリックします。

結果

ドメインは削除されます。

参照

- 「DROP DOMAIN 文」『SQL Anywhere サーバ SQL リファレンス』

エンティティ整合性と参照整合性

データベースの関係構造によって、データベースサーバはデータベース内の情報を識別し、各テーブル内のすべてのローで (データベーススキーマに定義されている) テーブル間の関係が維持されていることを保証できます。

エンティティ整合性

ユーザがローを挿入または更新すると、データベースサーバによってそのテーブルのプライマリキーの有効性が確実に保持されます。つまり、テーブル内の各ローがプライマリキーによりユニークに識別されます。

例 1

SQL Anywhere サンプルデータベースの Employees テーブルでは、employee ID をプライマリキーとして使用します。新しい従業員がこのテーブルに追加されると、データベースサーバは新しい employee ID 値がユニークであり、NULL でないことを検査します。

例 2

SQL Anywhere サンプルデータベースの SalesOrderItems テーブルは、プライマリキーとして 2 つのカラムを使用します。

このテーブルは注文された製品の情報を格納します。ID カラムには注文番号が入っていますが、1 つの注文番号に対して複数の製品が注文される場合があるため、このカラムだけではプライマリキーになりません。一方、LineID カラムは製品に対応する行を識別します。ID カラムと LineID カラムの 2 つがセットになって、ある製品をユニークに指定できるので、この 2 つがプライマリキーになります。

クライアントアプリケーションがエンティティ整合性に違反する場合

エンティティ整合性は、プライマリキーの値がユニークで、かつそこに NULL が含まれていないことが必要です。クライアントアプリケーションが重複するプライマリキー値を追加または更新すると、エンティティ整合性が破られます。エンティティ整合性の違反が検出されると、新しい情報はデータベースに追加されず、クライアントアプリケーションにエラーが返されます。

整合性の違反をどのようにしてユーザに通知し、どう処理させるかは、アプリケーション側の問題です。適切な処置といっても、ユーザに対して重複しない値をプライマリキーに指定するよう促すことしかできません。

プライマリキーによるエンティティ整合性の確保

各テーブルにプライマリキーが設定されれば、エンティティ整合性を確保するためにクライアントアプリケーション開発者やデータベース管理者がそれ以上することはありません。

テーブルの所有者は、テーブルの作成時にプライマリキーを指定します。後日、テーブル構造を修正した場合は、プライマリキーも再定義します。

参照

- 「プライマリキー」17 ページ
- 「CREATE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』

参照整合性

外部キー関係を有効にするには、外部キーのエントリが参照先のテーブルのローのプライマリキー値に対応していなければなりません。場合によっては、プライマリキー以外のユニークなカラムが参照先になります。

外部キーは、通常別のテーブルにあるプライマリキーまたは一意性制約を参照します。そのプライマリキーが存在しない場合、対応する外部キーは「オーファン」と呼ばれます。SQL Anywhere は、データベースに参照整合性に違反するローがないかを自動的に確認します。このプロセスを

「参照整合性の検証」と呼びます。データベースサーバは、オーファン数をカウントし参照整合性を調べます。

複数カラムの外部キーを使用する場合、MATCH 句を使用して、孤立したローの構成内容と参照整合性違反の構成内容を対比できます。MATCH 句では、また、キーの一意性を指定することで、一意性を別に宣言する必要がなくなります。

次に、指定できる MATCH タイプを示します。

- **MATCH [UNIQUE] SIMPLE**　外部キーテーブルのローに一致が発生するのは、すべてのカラム値がプライマリキーテーブルのローにある対応するカラム値と一致する場合です。外部キーテーブルでローが孤立するのは、外部キーの少なくとも 1 つのカラム値が NULL である場合です。

  MATCH SIMPLE はデフォルトの動作です。

  UNIQUE キーワードを指定すると、NULL 以外のキー値に対して、参照テーブルで一致が 1 つのみになります。

- **MATCH [UNIQUE] FULL**　外部キーテーブルのローに一致が発生するのは、いずれの値も NULL ではなく、値がプライマリキーテーブルのローにある対応するカラム値と一致する場合です。ローが孤立するのは、外部キーのすべてのカラム値が NULL の場合です。

  UNIQUE キーワードを指定すると、NULL 以外のキー値に対して、参照テーブルで一致が 1 つのみになります。

例 1

SQL Anywhere サンプルデータベースには、Employees テーブルと Departments テーブルが含まれています。Employees テーブルのプライマリキーは employee ID、また Departments テーブルのプライマリキーは department ID です。Employees テーブルから見ると、department ID は department テーブルの「外部キー」になります。Employees テーブルのそれぞれの department ID が、Departments テーブルの department ID と完全に一致しているためです。

外部キー関係は、多対 1 の関係です。Employees テーブルの中には、同じ department ID エントリを含む複数のエントリがありますが、department ID は Departments テーブルのプライマリキーであるため、その中には 1 つしかありません。外部キーが、重複したエントリもしくは NULL 値を含む Departments テーブルのカラムを参照できると、Departments テーブルのどのローが正しい参照先かを決められなくなります。これは、外部キーカラムを NOT NULL として定義することによって回避されます。

例 2

データベースに、事業所の所在地をリストする office テーブルが含まれているとします。Employees テーブルには、その従業員が勤務する事業所の所在地を示すために、office テーブルに対する外部キーが指定されています。ここでデータベースの設計者は、従業員が雇用されたときに事業所の所在地を指定しないでおくことができます。これは、まだ配置先が決定していない場合や、外で働く場合に対応するためです。このような場合、外部キーはオプションで、NULL 値を使用できます。

例 3

次の文を実行して、複合プライマリキーを作成します。

```
CREATE TABLE pt(
    pk1 INT NOT NULL,
    pk2 INT NOT NULL,
    str VARCHAR(10)
    PRIMARY KEY ( pk1, pk2 ));
```

次の文は、プライマリキーとカラムの順序が異なる外部キーと、外部キーカラムに対して異なるソートの程度を作成します。これは、外部キーインデックスを作成する場合に使用されます。

```
CREATE TABLE ft1(
    fpk INT PRIMARY KEY,
    ref1 INT,
    ref2 INT );

ALTER TABLE ft1 ADD FOREIGN KEY ( ref2 ASC, ref1 DESC)
    REFERENCES pt ( pk2, pk1 ) MATCH SIMPLE;
```

次の文を実行して、プライマリキーとカラムの順序が同じだが、外部キーインデックスに対してソートの程度が異なる外部キーを作成します。また、この例では、MATCH FULL 句を使用して、両方のカラムが NULL だった場合に孤立したローになることを指定します。UNIQUE 句は、NULL ではないカラムの **pt** テーブルと **ft2** テーブルの間に 1 対 1 の関係を適用します。

```
CREATE TABLE ft2(
    fpk INT PRIMARY KEY,
    ref1 INT,
    ref2 INT );

ALTER TABLE ft2 ADD FOREIGN KEY ( ref1, ref2 DESC )
    REFERENCES pt ( pk1, pk2 ) MATCH UNIQUE FULL;
```

参照循環性

参照元テーブルと参照先テーブルは別である必要はありません。テーブルに、それ自体を参照する外部キーを含めることができます。これを自己参照テーブルと呼びます。自己参照テーブルは、参照循環性の特殊な場合です。

例 1

SQL Anywhere サンプルデータベースには、従業員 (employee) 情報が入ったテーブルが 1 つと、部署 (department) 情報の入ったテーブルが 1 つあります。

```
CREATE TABLE "GROUPO"."Employees" (
   "EmployeeID"          int NOT NULL
  ,"ManagerID"           int NULL
  ,"Surname"             "person_name_t" NOT NULL
  ,"GivenName"           "person_name_t" NOT NULL
  ,"DepartmentID"        int NOT NULL
  ,"Street"              "street_t" NOT NULL
  ,"City"                "city_t" NOT NULL
  ,"State"               "state_t" NULL
  ,"Country"             "country_t" NULL
  ,"PostalCode"          "postal_code_t" NULL
  ,"Phone"               "phone_number_t" NULL
  ,"Status"              char(2) NULL
```

```
    ,"SocialSecurityNumber"          char(11) NOT NULL
    ,"Salary"                        numeric(20,3) NOT NULL
    ,"StartDate"                     date NOT NULL
    ,"TerminationDate"               date NULL
    ,"BirthDate"                     date NULL
    ,"BenefitHealthInsurance"        bit NULL
    ,"BenefitLifeInsurance"          bit NULL
    ,"BenefitDayCare"                bit NULL
    ,"Sex"                           char(2) NULL CONSTRAINT "Sexes" check(Sex in( 'F','M','NA' ) )
    ,CONSTRAINT "EmployeesKey" PRIMARY KEY ("EmployeeID")
  )

  ALTER TABLE "GROUPO"."Employees"
    ADD CONSTRAINT "SSN" UNIQUE ( "SocialSecurityNumber" )

  CREATE TABLE "GROUPO"."Departments" (
    "DepartmentID"                   int NOT NULL
    ,"DepartmentName"                char(40) NOT NULL
    ,"DepartmentHeadID"              int NULL
    ,CONSTRAINT "DepartmentsKey" PRIMARY KEY ("DepartmentID")
    ,CONSTRAINT "DepartmentRange" check(DepartmentID > 0 and DepartmentID <= 999)
  )
```

Employees テーブルには、プライマリキー "EmployeeID" と候補キー"SocialSecurityNumber" があります。Departments テーブルには、プライマリキー"DepartmentID" があります。Employees テーブルは、外部キーの定義によって Departments テーブルに関連付けられます。

```
  ALTER TABLE "GROUPO"."Employees"
    ADD NOT NULL FOREIGN KEY "FK_DepartmentID_DepartmentID" ("DepartmentID")
    REFERENCES "GROUPO"."Departments" ("DepartmentID")
```

特定の従業員の所属部署名を探せるように、その従業員の部署名を Employees テーブルに格納しておく必要はありません。その代わり Employees テーブルには、Departments テーブルの DepartmentID 値の 1 つと一致する部署番号の入った "DepartmentID" カラムがあります。

Employees テーブルは、Employees と Departments 間の多対 1 の関係を宣言する上記の参照整合性制約によって、Departments テーブルを参照します。さらに、Employees テーブルの外部キーカラムである DepartmentID が NOT NULL と宣言されているため、これは必須の関係です。しかし、これは Employees テーブルと Departments テーブル間の唯一の関係ではありません。Departments テーブル自体に、各部署の長を示すための Employees テーブルへの外部キーがあります。

```
  ALTER TABLE "GROUPO"."Departments"
    ADD FOREIGN KEY "FK_DepartmentHeadID_EmployeeID" ("DepartmentHeadID")
    REFERENCES "GROUPO"."Employees" ("EmployeeID")
    ON DELETE SET NULL
```

これは、Departments テーブルと Employees テーブル間のオプションの多対 1 の関係です。参照整合性制約だけでは複数の部署の長が同じになることを回避できないため、多対 1 になっています。その結果、Employees テーブルと Departments テーブルは参照循環性を形成し、それぞれに他方への外部キーがあります。

外部キーによる参照整合性の確保

プライマリキーと同様、外部キーは CREATE TABLE 文または ALTER TABLE 文を使って作成します。外部キーを作成すると、外部キーに指定したカラムに入力できる値が、関連付けたテーブルのプライマリキー値として存在する値にかぎられます。

参照整合性の喪失

次のような場合、データベースの参照整合性が失われる可能性があります。

- プライマリキーの値が更新または削除された場合。そのプライマリキーを参照している外部キーがすべて無効になります。
- 外部テーブルに新しく追加されたローに、プライマリキーと対応しない外部キーの値が入力された場合。データベースが無効になります。

SQL Anywhere は、この両方に対する防護策を備えています。

クライアントアプリケーションが参照整合性に違反する場合

クライアントアプリケーションがあるテーブルからプライマリキーの値を更新または削除したときに、データベース内にそのプライマリキーを参照する外部キーがある場合、参照整合性に違反する危険性があります。

例

サーバがプライマリキーの更新または削除を許可して、それを参照する外部キーに何も変更を加えないと、外部キーの参照は無効になります。KEY JOIN 句を使用した SELECT 文など、外部キーの参照を使う処理は、参照先のテーブルに対応する値がないためエラーになります。

SQL Anywhere は、エンティティ整合性に違反しそうになると、単にデータ入力を拒否してエラーメッセージを表示するだけですが、参照整合性の違反への対応は複雑になる場合があります。参照整合性を維持するための参照整合性アクションとして知られている手段はいくつかあります。

参照整合性アクション

他から参照されているプライマリキーの更新や削除に対して参照整合性を維持する最も単純な方法は、更新や削除を禁止することです。しかしそれ以外にも、参照整合性を保つために各外部キーを操作することもできます。データベース管理者やテーブル所有者は、CREATE TABLE 文と ALTER TABLE 文を使って、変更されたプライマリキーを参照している外部キーに対し、整合性の違反が発生したときに実行するアクションを指定できます。

注意
参照整合性アクションは、ユニークな値に対する「論理的な」更新ではなく「物理的な」更新によってトリガされます。たとえば、大文字と小文字を区別しないデータベースでも、プライマリキーの値を *SAMPLE-VALUE* から *sample-value* に更新すると、2 つの値は論理的には同じであっても参照整合性アクションがトリガされます。

次の参照整合性アクションは、プライマリキーの更新と削除に対してそれぞれ別に指定できます。

- **RESTRICT**　参照されているプライマリキーの値をユーザが変更しようとした場合、エラーを生成してその変更を防止します。これが参照整合性アクションのデフォルトです。
- **SET NULL**　変更されたプライマリキーを参照するすべての外部キーの値を NULL にします。
- **SET DEFAULT**　変更されたプライマリキーを参照するすべての外部キーを、そのカラムの (テーブル定義で指定されている) デフォルト値にします。
- **CASCADE**　ON UPDATE とともに使用すると、更新されたプライマリキーを参照するすべての外部キーを新しい値に更新します。ON DELETE とともに使用すると、削除されたプライマリキーを参照する外部キーを含むすべてのローを削除します。

参照整合性アクションはシステムトリガとして実装されます。トリガはプライマリテーブル上で定義され、セカンダリテーブルの所有者の権限を使って実行されます。つまり、特に権限が与えられていなくても、所有者の違うテーブルに対するカスケード処理ができることになります。

参照整合性の検査

参照性制約に違反する可能性があるとして RESTRICT などの制限が定義されている外部キーの検査は、デフォルトで文の実行時に行われます。CHECK ON COMMIT 句を指定した場合、検査が行われるのはトランザクションのコミット時にかぎられます。

検査のタイミングを制御するデータベースオプション

参照整合性に違反する操作を制限するよう定義されている外部キーは、wait_for_commit データベースオプションを使ってその動作を制御できます。このオプションは、CHECK ON COMMIT 句によって無効になります。

デフォルトでは wait_for_commit が Off に設定されており、データベースに整合性違反を生じさせる可能性のある操作は実行できません。たとえば、従業員がまだ存在している部署に対する DELETE は実行できません。次の文を実行すると、エラーになります。

```
DELETE FROM Departments
WHERE DepartmentID = 200;
```

wait_for_commit を On にすると、実際にコミットが実行されるまで参照整合性は検査されません。データベースの整合性が失われていると、データベースはコミットの実行を許可せずにエラーを返します。このモードの場合、データベースユーザはまだ従業員がいる部署を削除できますが、以下の操作が終了しないかぎり変更をコミットできません。

- その部署に所属する従業員を削除するか、もしくは他の部署に移す。
- DepartmentID ローを Departments テーブルに戻す。
- トランザクションをロールバックして、DELETE 操作を取り消す。

INSERT 文の整合性検査

SQL Anywhere は、INSERT 文の実行時に整合性検査を実行します。たとえば、部署を新設しようとする場合に、すでに使用されている DepartmentID 値を指定すると仮定します。

```
INSERT
INTO Departments ( DepartmentID, DepartmentName, DepartmentHeadID )
VALUES ( 200, 'Eastern Sales', 902 );
```

テーブルのプライマリキーがユニークでなくなるため、INSERT は拒否されます。DepartmentID カラムはプライマリキーなので、重複する値は許可されません。

関係に違反する値の挿入

次の文は SalesOrders テーブルに新しいローを挿入しますが、Employees テーブル内には存在しない SalesRepresentative ID を誤って指定します。

```
INSERT
INTO SalesOrders ( ID, CustomerID, OrderDate, SalesRepresentative)
VALUES ( 2700, 186, '2000-10-19', 284 );
```

Employees テーブルと SalesOrders テーブルとの関係は、SalesOrders テーブルの SalesRepresentative カラムと Employees テーブルの EmployeeID カラムに基づいた 1 対多の関係です。プライマリテーブル (Employees) にレコードが入力されている場合にのみ、外部テーブル (SalesOrders) 内の対応するレコードを挿入できます。

外部キー

Employees テーブルのプライマリキーは従業員 ID 番号です。SalesRepresentative テーブル内の営業担当者 ID 番号は、Employees テーブルの外部キーです。つまり、SalesOrders テーブル内にある各営業担当者 ID 番号と、Employees テーブル内にある同じ従業員の従業員 ID 番号を一致させる必要があります。

営業担当者 284 に対して注文を追加しようとすると、エラーが発生します。

Employees テーブルには、その ID 番号を持つ従業員は存在しません。これによって、有効な営業担当者 ID のない注文が挿入されるのを防ぐことができます。

参照

- 「テーブル間の関係」『SQL Anywhere 16 紹介』

DELETE 文または UPDATE 文の整合性検査

更新オペレーションまたは削除オペレーションを行う場合にも、外部キーエラーが発生する可能性があります。たとえば、Departments テーブルから R&D 部を削除するとします。DepartmentID フィールドは Departments テーブルのプライマリキーなので、1 対多の関係の「1」の側を構成します (対応する外部キーは Employees テーブルの DepartmentID フィールドで、「多」の側を構成します)。1 対多の関係の「1」のレコードは、対応する「多」のレコードすべてが削除されるまで削除できません。

DELETE 文の参照整合性エラー

Departments テーブルの R&D 部 (DepartmentID は 100) を削除するとします。R&D 部を参照するその他のレコードがデータベース内にあることを示すエラーがレポートされて、削除オペレーションが実行されません。R&D 部を削除するには、次のように、その部署に所属しているすべての従業員を削除する必要があります。

```
DELETE
FROM Employees
WHERE DepartmentID = 100;
```

これで、R&D 部に所属するすべての従業員が削除されたため、R&D 部を削除することができます。

```
DELETE
FROM Departments
WHERE DepartmentID = 100;
```

次の ROLLBACK 文を入力して、データベースのこれらの変更をキャンセルしてください。

```
ROLLBACK;
```

UPDATE 文の参照整合性エラー

たとえば、Employees テーブルの DepartmentID フィールドを変更するとします。DepartmentID フィールドは Employees テーブルの外部キーなので、1 対多の関係の「多」の側を構成します (対応するプライマリキーは Departments テーブルの DepartmentID フィールドで、「1」の側を構成します)。1 対多の関係の「多」の側のレコードは、「1」の側のレコードに対応しないかぎり、つまり参照するプライマリキーがなければ変更できません。

たとえば次の UPDATE 文を実行すると、整合性エラーが発生します。

```
UPDATE Employees
SET DepartmentID = 600
WHERE DepartmentID = 100;
```

DepartmentID が 600 の部署は Departments テーブルに存在しないため、エラーが発生します。

Employees テーブルの DepartmentID フィールドの値を変更するには、Departments テーブルの既存の値に対応する必要があります。次に例を示します。

```
UPDATE Employees
SET DepartmentID = 300
WHERE DepartmentID = 100;
```

この文は、DepartmentID 300 は既存の Finance 部に対応するので実行できます。

次の ROLLBACK 文を入力して、データベースのこれらの変更をキャンセルしてください。

```
ROLLBACK;
```

コミット時の整合性の検査

前述の例では、それぞれの文の実行時にデータベースの整合性が検査されています。データベースの整合性を失わせる可能性があるオペレーションは実行されません。

wait_for_commit オプションを使用すると、コミット時まで整合性が検査されないようにデータベースを設定することができます。これは、変更を加えている間にデータの一貫性が一時的に失われる可能性があるような変更の必要がある場合に便利です。たとえば、Employees テーブルと Departments テーブルの R&D 部を削除するとします。これらのテーブルには相互参照があり、かつ削除は一度に 1 テーブルずつ実行する必要があるため、削除中はテーブル間で一貫性が失われます。この場合、データベースでは削除が完了するまでコミットを実行できません。wait_for_commit オプションを On に設定して、コミットが実行されるまでデータの一貫性が失われることを許容してください。

また、プライマリキーに加えた変更に合わせて、外部キーが自動的に修正されるように定義することもできます。上の例では、Employees テーブルから Departments テーブルへの外部キーが ON DELETE CASCADE を使用して定義された場合、部署 ID を削除すると、Employees テーブルの対応するエントリが自動的に削除されます。

ここまでの例では、整合性のないデータベースが確定的なものとしてコミットされることはありません。変更することによってデータベースの整合性が失われる可能性がある場合、SQL Anywhere では代替の動作もサポートされています。

参照

- 「wait_for_commit オプション」『SQL Anywhere サーバ データベース管理』
- 「データ整合性」833 ページ

システムテーブルの整合性ルール

データベースの整合性検査と規則に関する情報は、すべて次に示すシステムテーブルに入っています。この情報を表示するには、次に示す対応するシステムビューを使用してください。

システムビュー	説明
SYS.SYSCONSTRAINT	SYS.SYSCONSTRAINT システムビュー内の各ローには、データベース内の制約が記述されています。現在サポートされている制約には、テーブルとカラムの検査、プライマリキー、外部キー、一意性制約が含まれます。 テーブルとカラムの検査制約の場合、実際の検査条件は SYS.ISYSCHECK システムテーブルに含まれています。
SYS.SYSCHECK	SYS.SYSCHECK システムビューの各ローは、SYS.SYSCONSTRAINT システムビューにリストされている検査制約を定義します。
SYS.SYSFKEY	SYS.SYSFKEY システムビューの各ローは、キーに定義した一致タイプなど、外部キーに関して記述します。

システムビュー	説明
SYS.SYSIDX	SYS.SYSIDX システムビューの各ローは、データベース内のインデックスを定義します。
SYS.SYSTRIGGER	SYS.SYSTRIGGER システムビューの各ローは、参照トリガアクション (ON DELETE CASCADE など) が設定された外部キー制約を自動的に作成するトリガなどの、データベース内のトリガ 1 つを示します。 referential_action カラムには、アクションがカスケード (C)、削除 (D)、NULL 設定 (N)、制限 (R) のいずれであるかを示すアルファベット 1 文字が格納されています。 event カラムには、各アクションを実行するイベントを示すアルファベット 1 文字が格納されます (A=挿入と削除、B=挿入と更新、C=更新、D=削除、E=削除と更新、I=挿入、U=更新、M=挿入、削除、更新)。 trigger_time カラムには、アクションの実行がイベントのトリガの後 (A) または前 (B) のどちらに行われるかが格納されます。

参照

- 「SYSCONSTRAINT システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSCHECK システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSFKEY システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSIDX システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSTRIGGER システムビュー」『SQL Anywhere サーバ SQL リファレンス』

トランザクションと独立性レベル

データの整合性を確保するには、データベースの情報が「一貫している」状態を識別できることが重要です。一貫性の概念は例によってよく理解できます。

一貫性の例

預金口座を扱うデータベースを使っていて、ある顧客の口座から別の口座に送金したいとします。送金前と送金後では、データベースの状態は一貫しています。しかし、引き落としの後、次の口座に振り込まれるまでのデータベースの状態は一貫していません。送金処理において、顧客の口座残高が送金前と同じ段階では、データベースの状態は一貫性があります。ある程度送金されると、データベースの状態は一貫性がなくなります。引き落としと振り込みの両方を処理するか、またはどちらも処理しないようにする必要があります。

トランザクションは作業の論理単位

「トランザクション」は、作業の論理単位です。各トランザクションは、1 つのタスクを実行し、データベースをある一貫した状態から別の状態へ変更する、論理的に関連する一連の文です。一貫した状態の性質は、使用しているデータベースに依存します。

各トランザクション内の文は不可分の単位として処理されます。すべての文が実行されるか、またはどの文も実行されません。各トランザクションの最後に、変更を「コミット」し確定します。トランザクション内の文のいくつかが正しく処理されない場合は、途中の変更が取り消されるか、または「ロールバック」されます。これを、トランザクションが「アトミック」であるといいます。

複数の文をトランザクションにグループ分けすることは、メディア障害またはシステム障害時にデータの一貫性を保護するため、またはデータベースの同時操作を管理するために重要です。トランザクションは安全にインターリーブされ、各トランザクションが完了すると、データベース内の情報が一貫しているポイントにマークが付けられます。データベースを 1 つの一貫した状態から別の状態へ変更するタスクを実行するために各トランザクションを設計します。

通常の操作においてシステム障害やデータベースクラッシュが発生した場合、SQL Anywhere はそのデータベースが次に起動されたときにデータを自動的にリカバリします。自動リカバリでは、完了済みのすべてのトランザクションをリカバリし、障害発生時にコミットされていなかったトランザクションをロールバックします。トランザクションのアトミックな性質により、データベースは確実に一貫した状態にリカバリされます。

参照

- 「バックアップとデータリカバリ」『SQL Anywhere サーバ データベース管理』
- 「同時実行性」863 ページ

トランザクション

SQL Anywhere では、文はトランザクションにまとめます。トランザクションをコミットすると、データベースの変更が永続的なものになります。データを変更すると、変更がトランザクションログに記録されます。変更は、COMMIT 文を入力するまでは永続的なものにはなりません。

トランザクションは次のいずれかのイベントで開始します。

●データベースへの接続後、最初の文

●トランザクションの終了後、最初の文

トランザクションは次のいずれかのイベントで完了します。

●データベースの変更を確定する COMMIT 文

●トランザクションで行われたすべての変更を取り消す ROLLBACK 文

●オートコミットが実行される文。ALTER、CREATE、COMMENT、DROP など、データ定義文の大半は自動的にコミットを実行します。

●データベースへの接続を解除すると、暗黙的なロールバックが実行されます。

●ODBC と JDBC には各文の後で COMMIT 文を実行するオートコミットの設定があります。デフォルトでは、ODBC と JDBC のオートコミットの設定は ON にする必要があり、各文は単一のトランザクションとして処理されます。トランザクション設計機能を活用する場合は、オートコミットの設定を OFF にします。

●chained データベースオプションを Off にしておくと、各文の後にオートコミットを実行したのと同様の処理が行われます。デフォルトでは、jConnect または Open Client アプリケーションを使用する接続では chained は Off に設定されています。

SQL Anywhere コンソールユーティリティ (dbconsole) を使用してデータベースに接続することで、未処理のトランザクションがある接続を決定できます。**[接続]** ウィンドウ枠を検査して、未処理のトランザクションがある接続を確認します。

Interactive SQL のオプション

Interactive SQL は、いつ、どのようにトランザクションを終了するかを指定する次の 2 つのオプションを提供します。

●auto_commit オプションを On に設定すると、Interactive SQL では、文が正常に実行された場合は結果が自動的にコミットされ、失敗した場合は ROLLBACK が自動的に実行されます。

●commit_on_exit オプションは、Interactive SQL を終了したときにコミットされていない変更をどうするかを決定します。このオプションを On (デフォルト) に設定すると、Interactive SQL は COMMIT を実行し、それ以外の場合はコミットされていない変更を ROLLBACK 文で取り消します。

Interactive SQL でのデータソースの使用
デフォルトでは、ODBC はオートコミットモードで動作します。Interactive SQL で auto_commit オプションを Off に設定しても、ODBC 設定が Interactive SQL の設定を上書きします。ODBC の設定は、SQL_ATTR_AUTOCOMMIT 接続属性を使用して変更できます。ODBC オートコミットは chained オプションから独立しています。

SQL Anywhere も、BEGIN TRANSACTION などの Transact-SQL 文をサポートし、Adaptive Server Enterprise との互換性を保ちます。

トランザクションの開始タイミングの決定

TransactionStartTime データベースプロパティは、COMMIT または ROLLBACK の後にデータベースが最初に修正された時間を返します。このプロパティを使用すると、すべてのアクティブな接続で最も早いトランザクションの開始時刻を調べることができます。次に例を示します。

```
BEGIN
  DECLARE connid int;
  DECLARE earliest char(50);
  DECLARE connstart char(50);
  SET connid=next_connection(null);
  SET earliest = NULL;
  lp: LOOP
  IF connid IS NULL THEN LEAVE lp END IF;
    SET connstart = CONNECTION_PROPERTY('TransactionStartTime',connid);
    IF connstart <> '' THEN
      IF earliest IS NULL
      OR CAST(connstart AS TIMESTAMP) < CAST(earliest AS TIMESTAMP) THEN
        SET earliest = connstart;
      END IF;
    END IF;
    SET connid=next_connection(connid);
  END LOOP;
  SELECT earliest
END
```

参照

- 「SQL 文」『SQL Anywhere サーバ SQL リファレンス』
- 「オートコミットまたは手動コミットモード」『SQL Anywhere サーバ プログラミング』
- 「chained オプション」『SQL Anywhere サーバ データベース管理』
- 「auto_commit オプション [Interactive SQL]」『SQL Anywhere サーバ データベース管理』
- 「commit_on_exit オプション [Interactive SQL]」『SQL Anywhere サーバ データベース管理』
- 「Transact-SQL との互換性」640 ページ
- 「SQL Anywhere コンソールユーティリティ (dbconsole)」『SQL Anywhere サーバ データベース管理』
- 「未処理のトランザクションがある接続の特定 (SQL の場合)」『SQL Anywhere サーバ データベース管理』

同時実行性

同時実行性とは複数のトランザクションを同時に処理するためにデータベースサーバで必要な機能です。データベースサーバ内に特殊なメカニズムがない場合、同時に発生したトランザクションがお互いに干渉して、情報の一貫性が失われる可能性があります。

同時実行性に関する知識の必要な方

同時実行性は、すべてのデータベース管理者と開発者に関係のある問題です。シングルユーザデータベースで作業する場合でも、複数のアプリケーションからの要求や、単一のアプリケーションの複数の接続からの要求を処理する場合には、同時性を考慮してください。これらの複数のアプリケーションや接続は、ネットワーク上でマルチユーザが経験するのとまったく同様に、お互いに干渉し合います。

同時実行性に影響するトランザクションサイズ

SQL 文をトランザクションにまとめる方法は、データの整合性とシステムのパフォーマンスに大きな影響を与えます。トランザクションが短かすぎて論理的にひとまとまりにならない場合、データベースの一貫性が失われる可能性があります。トランザクションが長すぎて複数の互いに関係のない操作が含まれていると、本来なら安全にデータベースにコミットできたはずの操作が、不要な ROLLBACK によって取り消される可能性が高くなります。

トランザクションが長い場合、他のトランザクションが同時処理することを防ぐため、同時実行性が低下します。

アプリケーションのタイプや環境要因によって、トランザクションの適切な長さを決定する要素は数多くあります。

参照

- 「SQL Anywhere データベースサーバ」『SQL Anywhere サーバ データベース管理』

トランザクション内のセーブポイント

「セーブポイント」を使用して関連する文をグループ分けすることによって、トランザクション内の重要な状態を識別し、そこに戻ることができます。

SAVEPOINT 文は、トランザクションの中に中間ポイントを定義します。そのポイント以降の変更はすべて ROLLBACK TO SAVEPOINT 文を使ってキャンセルできます。RELEASE SAVEPOINT 文の実行後、またはトランザクションの終了後は、セーブポイントは使えなくなります。セーブポイントは、COMMIT には影響しません。COMMIT が実行されると、トランザクション中に加えられたすべての変更がデータベースの中で永続的なものになります。

RELEASE SAVEPOINT や ROLLBACK TO SAVEPOINT 文では、ロックは解放されません。ロックはトランザクションの最後でのみ解放されます。

セーブポイントの命名とネスト

名前を付けてネストしたセーブポイントを使って、トランザクション内に多数のアクティブなセーブポイントを設定できます。SAVEPOINT と RELEASE SAVEPOINT の間の更新も、その前のセーブポイントにロールバックするか、トランザクション全体をロールバックすればキャンセルできます。トランザクション内の変更は、トランザクションがコミットされるまで確定していません。トランザクションが終了すると、セーブポイントはすべて解放されます。

セーブポイントはバルクオペレーションモードでは使用できません。セーブポイントを使用しても、オーバヘッドはほとんど増えません。

独立性レベルと一貫性

SQL Anywhere では、あるトランザクションの操作が、同時に実行される他のトランザクションの操作で認識される程度を制御できます。この制御には、「独立性レベル」というデータベースオプションを使用します。

また、対応するテーブルヒントを使用して、クエリ内の個々のテーブルの独立性レベルを制御することもできます。

SQL Anywhere が提供するインタフェースを次に示します。

独立性レベル	特性
0 - コミットされない読み出し	●書き込みロックの有無にかかわらず、ローで読み込みが許可される。 ●読み込みロックは適用されない。 ●同時トランザクションがローを変更しないこと、またはローに対しての変更がロールバックされないことは保証されない。 ●テーブルヒント NOLOCK と READUNCOMMITTED に対応する。 ●ダーティリード、繰り返し不可能読み出し、幻ローを許可する。
1 - コミットされた読み出し	●書き込みロックのないローでは読み込みのみ許可される。 ●現在のローでの読み込みに対してのみ読み込みロックが取得されて保持されるが、カーソルがローから移動すると解放される。 ●トランザクション中にデータが変更されないという保証はない。 ●テーブルヒント READCOMMITTED に対応する。 ●ダーティリードを防ぐ。 ●繰り返し不可能読み出しと幻ローを許可する。

独立性レベル	特性
2 - 繰り返し可能読み出し	●書き込みロックのないローでは読み込みのみ許可される。 ●結果セットの各ローとして取得された読み込みロックが読み込まれ、トランザクションが終了するまで保持される。 ●テーブルヒント REPEATABLEREAD に対応する。 ●ダーティリードと繰り返し不可能読み出しを防ぐ。 ●幻ローを許可する。
3 - 直列化可能	●書き込みロックのない結果内のローに対しては読み込みのみ許可される。 ●カーソルが開いているときに取得された読み込みロックは、トランザクションの終了時まで保持される。 ●テーブルヒント HOLDLOCK と SERIALIZABLE に対応する。 ●ダーティリード、繰り返し不可能読み出し、幻ローを防ぐ。
snapshot [1]	●読み込みロックは適用されない。 ●すべてのローで読み込みが許可される。 ●コミットされたデータのデータベーススナップショットは、トランザクションが最初のローの読み込みまたは更新を行った時点で作成される。
statement-snapshot[1]	●読み込みロックは適用されない。 ●すべてのローで読み込みが許可される。 ●コミットされたデータのデータベーススナップショットは、文が最初のローの読み込みを行った時点で作成される。

独立性レベル	特性
readonly-statement-snapshot[1]	●読み込みロックは適用されない。 ●すべてのローで読み込みが許可される。 ●コミットされたデータのデータベーススナップショットは、読み込み専用の文が最初のローの読み込みを行った時点で作成される。 ●更新可能な文の updatable_statement_isolation オプションで指定された独立性レベル (0、1、2、または 3) を使用する。

[1] データベースで allow_snapshot_isolation が On に設定され、そのデータベースでスナップショットアイソレーションが有効である必要があります。

デフォルトの独立性レベルは 0 です。ただし、Open Client、jConnect、TDS の各接続におけるデフォルトの独立性レベルは 1 です。

ロックベースの独立性レベルは、一部またはすべての干渉を防ぎます。レベル 3 は最も高いレベルの独立性を提供します。2 以下のレベルでは、一貫性のレベルは低くなりますが、パフォーマンスは一般にレベル 3 より高くなります。デフォルトでは、レベルは 0 (コミットされない読み出し) に設定されています。

スナップショットアイソレーションのレベルは、読み込みと書き込み間の干渉を防ぎます。書き込みは相互に干渉する可能性があります。一貫性のない動作が多少生じる可能性がありますが、競合のパフォーマンスは独立性レベルを 0 に設定した場合と同じです。ローバージョンを保存して使用する必要があるため、競合以外のパフォーマンスは低下します。

一般的に、各独立性レベルは、必要なロックの種類や、他のトランザクションが保持するロックをどのように扱うかによって特徴づけられます。独立性レベルが 0 の場合、データベースサーバは書き込みロックだけを必要とします。データベースサーバは、これらのロックを使用して、2 つのトランザクションが競合する修正を行わないようにします。たとえば、レベル 0 のトランザクションは、ローの更新や削除をする前に書き込みロックをかけ、すでに書き込みロックがかかっている新しいローを挿入します。

レベル 0 のトランザクションは、読み込み中のローはチェックしません。たとえば、レベル 0 のトランザクションがローを読み込むときは、他のトランザクションがそのローにどのようなロックをかけているかをチェックしません。チェックが不要のため、レベル 0 のトランザクション処理は速くなります。この速度は一貫性を犠牲にして得られたものです。別のトランザクションが書き込みロックをかけているローを読むと、ダーティデータを返す危険性があります。レベル 1 では、トランザクションはローを読む前に書き込みロックがかかっていないかをチェックします。操作は 1 つ増えますが、このトランザクションでは読み込むデータはすべてコミット済みであることが保証されます。

注意
どの独立性レベルにおいても、各トランザクションが完全に実行されるかまったく実行されないこと、および更新内容が失われないことが保証されます。

独立性レベルはトランザクション間にのみ当てはまります。同じトランザクション内の複数のカーソルが相互に干渉することはありません。

参照

- 「FROM 句」『SQL Anywhere サーバ SQL リファレンス』
- 「Mobile Link 独立性レベル」『Mobile Link サーバ管理』
- 「スナップショットアイソレーションを有効化する方法」871 ページ

スナップショットアイソレーション

複数のユーザが同じデータに対して同時に読み込みと書き込みを行うと、ブロックやデッドロックが発生することがあります。スナップショットアイソレーションは、さまざまなバージョンのデータを管理することにより同時実行性と一貫性を向上させる目的で設計されています。トランザクションでスナップショットアイソレーションを使用すると、データベースサーバは読み込み要求の応答としてコミットされたバージョンのデータを返します。これは読み込みロックを取得せずに行われるため、データを書き込んでいるユーザとの干渉は発生しません。

「スナップショット」とは、データベースでコミットされた一連のデータです。スナップショットアイソレーションを使用すると、トランザクション内のすべてのクエリで同じデータのセットが使用されます。データベースのテーブルでロックは取得されません。これにより、他のトランザクションはブロックせずにアクセスして修正できます。SQL Anywhere では、スナップショットを作成するときに制御できる 3 つのスナップショットアイソレーションのレベルをサポートしています。

- **snapshot**　トランザクションが最初のローの読み込み、挿入、更新、または削除を行った時点から、コミットされたデータのスナップショットを使用します。

- **statement-snapshot**　文で最初のローが読み込まれた時点から、コミットされたデータのスナップショットを使用します。トランザクション内の各文で参照されるデータのスナップショットはそれぞれ異なる時点のものになります。

- **readonly-statement-snapshot**　読み込み専用の文についてのみ、最初のローが読み込まれた時点から、コミットされたデータのスナップショットを使用します。トランザクション内の各読み込み専用文で参照されるデータのスナップショットはそれぞれ異なる時点のものになります。挿入、更新、削除の文については、updatable_statement_isolation オプションに指定された独立性レベル (0 (デフォルト)、1、2、3 のいずれか) を使用します。

BEGIN SNAPSHOT 文を使用して、トランザクションのスナップショットの開始時に指定するオプションもあります。

スナップショットアイソレーションは、主に次のような場合に便利です。

- **読み込みは多いが更新は少ないアプリケーション**　スナップショットトランザクションは、データベースを修正する文の場合のみ、書き込みロックを取得します。トランザクションが

主に読み込み操作を実行する場合、スナップショットトランザクションは、他のユーザと干渉する可能性のある読み込みロックを取得しません。

- **他のユーザがデータにアクセスする必要があるにもかかわらず、時間がかかるトランザクションを実行するアプリケーション**　スナップショットトランザクションは読み込みロックを取得しないため、他のユーザはスナップショットトランザクションの実行中にデータの読み込みや更新を実行できます。

- **データベースからの一貫したデータのセットを読み取る必要があるアプリケーション**　スナップショットはある時点からのコミット済みデータのセットを表示するため、トランザクションの実行中に他のユーザが変更を加えた場合でも、スナップショットアイソレーションを使用すれば、そのトランザクション内では変更されない一貫したデータを確認できます。

スナップショットアイソレーションは、すべてのユーザが共有するベーステーブルとグローバルテンポラリテーブルのみに影響します。その他のテーブルの種類では読み込み操作を行っても、古いバージョンのデータは表示されず、スナップショットは開始されません。別のテーブルの種類に対する更新によってスナップショットが開始されるのは、isolation_level オプションが snapshot に設定されていて、更新によりトランザクションが開始される場合だけです。

文またはトランザクションのスナップショットを使用する、WITH HOLD 句を使用して開かれたカーソルがある場合、次の文は実行できません。

- 「ALTER INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ALTER TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「DROP INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REORGANIZE TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REFRESH TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』

WITH HOLD 句でカーソルを開くと、スナップショットの開始時にコミットされたすべてのローのスナップショットが表示されます。カーソルを開いたトランザクションの開始以降の、現在の接続で完了された変更もすべて表示されます。

高速トランケーションが実行されない場合にのみ、TRUNCATE TABLE を使用できます。これは、このような場合に個別の DELETE がトランザクションログに記録されるためです。

また、これらの文のいずれかをスナップショットでないトランザクションから実行した場合、すでに実行中のスナップショットトランザクションで、それ以降にテーブルを使用しようとすると、スキーマが変更されたことを示すエラーが返されます。

トランザクションのスナップショットが開始された後でビューが再表示された場合、マテリアライズドビューマッチングではそのビューは使用されません。

スナップショットアイソレーションのレベルは、すべてのプログラミングインタフェースでサポートされています。スナップショットアイソレーションのレベルは SET OPTION 文を使用し

て設定できます。スナップショットアイソレーションの使用については、「スナップショットアイソレーション」868 ページを参照してください。

- 「isolation_level オプション」『SQL Anywhere サーバ データベース管理』
- ADO と OLE DB： 「ADO のトランザクション」『SQL Anywhere サーバ プログラミング』
- ADO.NET： SATransaction.IsolationLevel プロパティ [SQL Anywhere .NET]『SQL Anywhere サーバ プログラミング』

ローバージョン

スナップショットアイソレーションがデータベースで有効になると、ローが更新されるたびに、データベースサーバは元のローのコピーをテンポラリファイルに格納されたバージョンに追加します。元のローバージョンのエントリは、元のロー値にアクセスする必要がある可能性のある、すべてのアクティブなスナップショットトランザクションが完了するまで格納されます。スナップショットアイソレーションを使用するトランザクションは、コミット済みの値だけを確認できます。そのため、スナップショットトランザクションの開始前にローの更新がコミットされなかったかロールバックされた場合、スナップショットトランザクションは元のロー値にアクセスできる必要があります。これにより、スナップショットアイソレーションを使用するトランザクションは、基本となるテーブルにロックを設定せずにデータを閲覧できます。

VersionStorePages データベースプロパティは、バージョンストア用に現在使用されているテンポラリファイル内のページ数を返します。この値を取得するには、次のクエリを実行します。

```
SELECT DB_PROPERTY ( 'VersionStorePages' );
```

古いローバージョンのエントリは、不要になると削除されます。BLOB の古いバージョンは、不要になるまで、テンポラリファイルではなく元のテーブルに格納されています。古いローバージョンのインデックスエントリは、不要になるまで元のインデックスに格納されています。

テンポラリファイル内の空き領域のサイズは、sa_disk_free_space システムプロシージャを使用すると取得できます。

ロー値を更新するトリガが起動されると、それらのローの元の値もテンポラリファイルに格納されます。

短いトランザクションと短いスナップショットを使用するようにアプリケーションを設定すると、必要なテンポラリファイルの領域は減少します。

テンポラリファイルの増大に関心がある場合は、テンポラリファイルが特定のサイズに達したときに実行するアクションを指定する GrowTemp システムイベントを設定できます。

スナップショットトランザクションの知識

スナップショットトランザクションは、更新時に書き込みロックを取得しますが、スナップショットを使用するトランザクションや文では読み込みロックを取得しません。その結果、読み込み処理は書き込み処理をブロックせず、書き込み処理は読み込み処理をブロックしません。ただし、書き込み処理は、同じローを更新しようとする他の書き込み処理をブロックすることがあります。

スナップショットアイソレーションを使用すると、トランザクションは BEGIN TRANSACTION 文で開始されません。トランザクションで使用されるスナップショットアイソレーションのレ

ベルに応じて、トランザクション内で最初の読み込み、挿入、更新、または削除時にトランザクションが開始します。次の例は、スナップショットアイソレーションでトランザクションが開始するタイミングを示します。

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';
      SET TEMPORARY OPTION isolation_level = 'snapshot';
  SELECT * FROM Products; --transaction begins and the statement only
                    --sees changes that are already committed
  INSERT INTO Products
      SELECT ID + 30, Name, Description,
      'Extra large', Color, 50, UnitPrice, NULL
      FROM Products
      WHERE Name = 'Tee Shirt';
COMMIT; --transaction ends
```

参照

- 「BEGIN SNAPSHOT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「TRUNCATE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_disk_free_space システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「システムイベント」『SQL Anywhere サーバ データベース管理』

スナップショットアイソレーションを有効化する方法

データベースのスナップショットアイソレーションは、allow_snapshot_isolation オプションを使用して有効または無効にします。オプションを On にすると、ローバージョンがテンポラリファイル内で管理され、接続で任意のスナップショットアイソレーションのレベルを使用できます。オプションを Off にすると、スナップショットアイソレーションを使用しようとすると、エラーが発生します。

データベースがスナップショットアイソレーションを使用できるようにすると、パフォーマンスに影響を与える可能性があります。これは、スナップショットアイソレーションを使用するトランザクションの数に関係なく、修正されたすべてのローのコピーを保持する必要があるからです。

次の文は、データベースのスナップショットアイソレーションを有効にします。

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';
```

allow_snapshot_isolation オプションの設定は、ユーザがデータベースに接続している場合でも変更できます。このオプションの設定を Off から On に変更した場合、新しいトランザクションがスナップショットアイソレーションを使用するには、現在のすべてのトランザクションが完了する必要があります。このオプションの設定を On から Off に変更した場合、データベースサーバがローバージョン情報の管理を停止するには、スナップショットアイソレーションを使用するすべての未処理のトランザクションが完了する必要があります。

特定のデータベースについて現在のスナップショットアイソレーションの設定を確認するには、SnapshotIsolationState データベースプロパティの値を問い合わせます。

```
SELECT DB_PROPERTY ( 'SnapshotIsolationState' );
```

SnapshotIsolationState プロパティの値は、次のいずれかです。

- **On**　データベースでスナップショットアイソレーションが有効になっている。
- **Off**　データベースでスナップショットアイソレーションが無効になっている。
- **in_transition_to_on**　現在のトランザクションが完了するとスナップショットアイソレーションが有効になります。
- **in_transition_to_off**　現在のトランザクションが完了するとスナップショットアイソレーションが無効になります。

スナップショットアイソレーションがデータベースで有効になると、スナップショットが使用されていない場合でも、トランザクションがコミットまたはロールバックしないかぎり、ローバージョンはトランザクションで管理される必要があります。そのため、スナップショットアイソレーションを使用しない場合は、allow_snapshot_isolation オプションを Off にすることをおすすめします。

参照

- 「カーソルの感知性と独立性レベル」『SQL Anywhere サーバ プログラミング』

スナップショットアイソレーションの例

次の例では、SQL Anywhere サンプルデータベースへの接続 2 つを使用し、スナップショットアイソレーションによってブロックなしで一貫性を維持する方法を示します。

例：スナップショットアイソレーションの使用

1. 次のコマンドを実行して、SQL Anywhere サンプルデータベースへの Interactive SQL 接続 (Connection1) を作成します。

   ```
   dbisql -c "DSN=SQL Anywhere 16 Demo;ConnectionName=Connection1"
   ```

2. 次のコマンドを実行して、SQL Anywhere サンプルデータベースへの Interactive SQL 接続 (Connection2) を作成します。

   ```
   dbisql -c "DSN=SQL Anywhere 16 Demo;ConnectionName=Connection2"
   ```

3. Connection1 で、次の文を実行して独立性レベルを 1 (コミットされた読み出し) に設定します。

   ```
   SET OPTION isolation_level = 1;
   ```

4. Connection1 で、次の文を実行します。

   ```
   SELECT * FROM Products;
   ```

ID	Name	Description	Size	Color	Quantity	...
300	Tee Shirt	Tank Top	Small	White	28	...

ID	Name	Description	Size	Color	Quantity	...
301	Tee Shirt	V-neck	Medium	Orange	54	...
302	Tee Shirt	Crew Neck	One size fits all	Black	75	...
400	Baseball Cap	Cotton Cap	One size fits all	Black	112	...
...	...	...	...	...	...	...

5. Connection2 で、次の文を実行します。

```
UPDATE Products
SET Name = 'New Tee Shirt'
WHERE ID = 302;
```

6. Connection1 で、もう一度 SELECT 文を実行します。

```
SELECT * FROM Products;
```

Connection2 の UPDATE 文がコミットまたはロールバックされていないため、SELECT 文がブロックされ (**[停止]** ボタンのみが選択可能)、先に進むことはできません。SELECT 文は、Connection2 のトランザクションが完了して処理できるようになるまで待機する必要があります。これにより、SELECT 文はコミットされていないデータを結果に読み込みません。

7. Connection2 で、次の文を実行します。

```
ROLLBACK;
```

Connection2 のトランザクションが完了し、Connection1 の SELECT 文が処理されます。

独立性レベル statement snapshot を使用することで、ブロックなしで独立性レベル 1 と同じ同時実行性を実現します。

8. Connection1 で、次の文を実行してスナップショットアイソレーションを許可します。

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';
```

9. Connection1 で、次の文を実行し、独立性レベルを statement snapshot に変更します。

```
SET TEMPORARY OPTION isolation_level = 'statement-snapshot';
```

10. Connection1 で、次の文を実行します。

```
SELECT * FROM Products;
```

11. Connection2 で、次の文を実行します。

```
UPDATE Products
SET Name = 'New Tee Shirt'
WHERE ID = 302;
```

12. Connection1 で、もう一度 SELECT 文を実行します。

```
SELECT * FROM Products;
```

SELECT 文は、ブロックされずに実行されますが、Connection2 によって実行された UPDATE 文からのデータは含まれません。

13. Connection2 で、次の文を実行してトランザクションを終了します。

```
COMMIT;
```

14. Connection1 で、トランザクション (Products テーブルに対するクエリ) を終了し、もう一度 SELECT 文を実行して更新されたデータを確認します。

```
COMMIT;
SELECT * FROM Products;
```

ID	Name	Description	Size	Color	Quantity	...
300	Tee Shirt	Tank Top	Small	White	28	...
301	Tee Shirt	V-neck	Medium	Orange	54	...
302	New Tee Shirt	Crew Neck	One size fits all	Black	75	...
400	Baseball Cap	Cotton Cap	One size fits all	Black	112	...
...	...	...	...	...	...	...

15. 次の文を実行し、SQL Anywhere サンプルデータベースに対する変更を取り消します。

```
UPDATE Products
SET Name = 'Tee Shirt'
WHERE id = 302;
COMMIT;
```

参照

- タスク910 ページ
- タスク916 ページ
- タスク921 ページ

更新の競合とスナップショットアイソレーション

スナップショットアイソレーションを使用する場合、トランザクションがローの古いバージョンを認識し、更新または削除しようとすると、更新の競合が発生することがあります。このような状況では、競合が検出されるとサーバでエラーが発生します。コミットされた変更の場合、これは更新または削除が試みられた時になります。コミットされていない変更の場合、更新または削除はブロックされ、変更がコミットされるときにサーバがエラーを返します。

readonly-statement-snapshot を使用すると、更新可能な文は、スナップショットアイソレーションではなく実行し、常に最新バージョンのデータベースを認識するため、更新の競合は発生しません。そのため、readonly-statement-snapshot 独立性レベルには、スナップショットアイソレーションの多くの利点があり、元々別の独立性レベルで実行するように設計されたアプリケーションを大きく変更する必要はありません。readonly-statement-snapshot 独立性レベルを使用するときは、次のようになります。

●読み込み専用文では、読み込みロックは取得されない。

●読み込み専用文は、データベースのコミットされた状態を常に認識する。

典型的な矛盾のケース

トランザクションの同時実行中に発生する可能性のある典型的な矛盾には 3 つのタイプがあります。(他のタイプの矛盾も発生し得るため、この 3 つがすべてというわけではありません)。これらの 3 つのタイプは ISO SQL/2008 標準に記載されており、低い独立性レベルで発生する可能性のある動作という観点で定義されています。

- **ダーティリード**　トランザクション A がローを修正し、変更のコミットもロールバックもしないとします。その修正がコミットまたはロールバックされる前に、トランザクション B がそのローを読みます。その後、COMMIT が実行される前に、トランザクション A がさらにそのローを変更するか、またはその修正をロールバックしたとします。いずれの場合も、トランザクション B はコミットされなかった状態でローを読んでしまったことになります。

- **繰り返し不可能読み出し**　トランザクション A がローを読みます。次にトランザクション B がそのローを修正または削除して、COMMIT を実行します。トランザクション A がもう一度そのローを読もうとしたときには、ローは修正されているか、削除されてしまっています。

- **幻ロー**　トランザクション A が、一定の条件を満たすローのセットを読みます。次に、トランザクション B は、前にトランザクション A の条件を満たさなかったローで INSERT または UPDATE を実行します。トランザクション B はこれらの変更をコミットします。新しくコミットされたローはトランザクション A の条件を満たします。トランザクション A がもう一度データを読み込むと、更新されたローのセットを取得します。

独立性レベルとダーティリード、繰り返し不可能読み出し、幻ロー

SQL Anywhere では、使用する独立性レベルに応じて、ダーティリード、繰り返し不可能読み出し、幻ローを許可します。次の表で X は、その動作がその独立性レベルで許可されていることを表します。

独立性レベル	ダーティリード	繰り返し不可能読み出し	幻ロー
0 - コミットされない読み出し	X	X	X
readonly-statement-snapshot	X[1]	X[2]	X[3]

独立性レベル	ダーティリード	繰り返し不可能読み出し	幻ロー
1 - コミットされた読み出し		X	X
statement-snapshot		X[2]	X[3]
2 - 繰り返し可能読み出し			X
3 - 直列化可能			
snapshot			

[1] ダーティリードは、updatable_statement_isolation オプションで指定された独立性レベルがその発生を妨げない場合に、トランザクション内の更新可能な文で発生することがあります。

[2] 繰り返し不可能読み出しは、updatable_statement_isolation オプションで指定された独立性レベルがその発生を妨げない場合に、トランザクション内の文で発生することがあります。繰り返し不可能読み出しは、各文によって新しいスナップショットが開始する場合に発生することがあります。そのため、ある文が認識しない変更を別の文が認識する可能性があります。

[3] 幻ローは、updatable_statement_isolation オプションで指定された独立性レベルがその発生を妨げない場合に、トランザクション内の文で発生することがあります。幻ローは、各文によって新しいスナップショットが開始する場合に発生することがあります。そのため、ある文が認識しない変更を別の文が認識する可能性があります。

この表から、以下の 2 つのことがわかります。

- 各独立性レベルは、3 つの典型的な矛盾のケースのうち 1 つを防止します。
- 各レベルは、それ以下のレベルで防止される矛盾のケースも防止します。
- statement snapshot 独立性レベルでは、繰り返し不可能読み出しと幻ローは、トランザクション内で発生する可能性がありますが、トランザクションの単一文内では発生しません。

ODBC では、独立性レベルに異なる名前が付いています。これらの名前は、それぞれのレベルで防止される矛盾の名前に基づいて付けられています。

参照

- 「ValuePtr パラメータ」879 ページ
- 「チュートリアル：ダーティリードの知識」907 ページ
- 「チュートリアル：繰り返し不可能読み出しの知識」912 ページ
- 「チュートリアル：幻ローの知識」917 ページ

カーソル不安定性

もう 1 つの重要な矛盾は「カーソル不安定性」です。この矛盾が起きると、あるトランザクションのカーソルが参照しているローを、他のトランザクションが修正できてしまいます。カーソルを使用するアプリケーションでは、カーソル安定性があれば、データベース内のデータに対する矛盾を確実に回避できます。

例

トランザクション A はカーソルを使用してローを読み込みます。トランザクション B が、そのローを修正し、コミットします。修正されたことに気づかず、トランザクション A がそのローを修正します。

カーソル不安定性への対処

SQL Anywhere は、独立性レベル 1 から 3 で「カーソル安定性」を提供しています。カーソル安定性により、カーソルの現在のローに含まれる情報を、他のトランザクションが修正できなくなります。カーソルのローの情報は、特定のテーブルに含まれる情報のコピーや、複数テーブルの異なるローにあるデータを組み合わせたものです。SELECT 文内でジョインまたはサブ選択を使用する場合、複数のテーブルが関係する可能性があります。

カーソルは、他のアプリケーションを介して SQL Anywhere を使っているときのみに使用します。

基本となるデータに加えた変更がカーソルを使用するアプリケーションから参照できるかどうかは、カーソルを使用するアプリケーションと関連はしていますが、別個の問題です。変更がアプリケーションから参照できるかどうかは、カーソルの sensitivity を指定して制御できます。

参照

- 「ストアドプロシージャ、トリガ、バッチ、ユーザ定義関数」77 ページ
- 「SQL を使用したアプリケーション開発」『SQL Anywhere サーバ プログラミング』
- 「SQL Anywhere のカーソル」『SQL Anywhere サーバ プログラミング』

独立性レベルの設定方法

データベースへの各接続は独自の独立性レベルを持ちます。さらに、データベースはユーザやユーザ拡張ロールごとにデフォルトの独立性レベルを保存できます。isolation_level データベースオプションで PUBLIC を設定すると、デフォルトの独立性レベルを設定できます。

テーブルヒントを使用して独立性レベルを設定することもできますが、これは個別の文に独立性レベルを設定する高度な機能です。

接続の独立性レベルや、ユーザ ID に設定されたデフォルトのレベルは、SET OPTION 文を使用して変更できます。他のユーザまたはグループの独立性レベルを変更することができます。

デフォルトの独立性レベル

データベースに接続すると、データベースサーバは次のように最初の独立性レベルを決定します。

1. デフォルトの独立性レベルは、ユーザやロールごとに設定できます。レベルがユーザ ID のデータベースに保存されている場合、データベースサーバはそのレベルを使用します。

2. レベルが保存されていない場合、データベースサーバはレベルが見つかるまでユーザが属しているグループをチェックします。すべてのユーザは、PUBLIC ロールの被付与者となります。最初に他の設定が見つからない場合、SQL Anywhere は PUBLIC に割り当てられているレベルを使用します。

注意
スナップショットアイソレーションを使用する場合は、先にデータベースでスナップショットアイソレーションを有効にする必要があります。

例

- **例 1:現在のユーザの独立性レベルの設定**　SET OPTION 文を実行します。たとえば次の文は、現在のユーザに独立性レベル 3 を設定します。

```
SET OPTION isolation_level = 3;
```

- **例 2 : ユーザまたは PUBLIC ロールの独立性レベルの設定**

  1. データベースに接続します。

  2. isolation_level の前に被付与者名とピリオドを付加し、SET OPTION 文を実行します。たとえば次の文は、PUBLIC ロールのデフォルトの独立性レベルを 3 に設定します。

```
SET OPTION PUBLIC.isolation_level = 3;
```

- **例 3 : 現在の接続の独立性レベルの設定**　TEMPORARY キーワードを使用して SET OPTION 文を実行します。たとえば次の文は、現在の接続に対して独立性レベル 3 を設定します。

```
SET TEMPORARY OPTION isolation_level = 3;
```

参照

- 「ユーザのセキュリティ (ロールと権限)」『SQL Anywhere サーバ データベース管理』
- 「SET OPTION 文」『SQL Anywhere サーバ SQL リファレンス』
- WITH table-hint 句、FROM 句『SQL Anywhere サーバ SQL リファレンス』
- 「スナップショットアイソレーションを有効化する方法」871 ページ
- 「トランザクション内の独立性レベルの変更」880 ページ

ODBC 実行可能アプリケーションでの独立性レベル

ODBC アプリケーションは、SQLSetConnectAttr を呼び出すときに Attribute を SQL_ATTR_TXN_ISOLATION に、ValuePtr を対応する独立性レベルに設定します。

ValuePtr パラメータ

ValuePtr	独立性レベル
SQL_TXN_READ_UNCOMMITTED	0
SQL_TXN_READ_COMMITTED	1
SQL_TXN_REPEATABLE_READ	2
SQL_TXN_SERIALIZABLE	3
SA_SQL_TXN_SNAPSHOT	snapshot
SA_SQL_TXN_STATEMENT_SNAPSHOT	statement-snapshot
SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT	readonly-statement-snapshot

ODBC を介した独立性レベルの変更

ODBC を介して接続の独立性レベルを変更するには、*ODBC32.dll* ライブラリの SQLSetConnectAttr 関数を使用します。

SQLSetConnectAttr 関数は、次の 4 つのパラメータを取ります。ODBC コネクションハンドルの値、独立性レベルの設定要求、設定する独立性レベル、ゼロに対応する値です。独立性レベルに対応する値が、下のテーブルに表示されます。

文字列	値
SQL_TXN_ISOLATION	108
SQL_TXN_READ_UNCOMMITTED	1
SQL_TXN_READ_COMMITTED	2
SQL_TXN_REPEATABLE_READ	4
SQL_TXN_SERIALIZABLE	8
SA_SQL_TXN_SNAPSHOT	32
SA_SQL_TXN_STATEMENT_SNAPSHOT	64
SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT	128

ODBC アプリケーションから SET OPTION 文を使用して、独立性レベルを変更しないでください。ODBC ドライバは SET OPTION 文を解析しないため、ODBC で文を実行しても ODBC ドライバは認識しません。これによって、ロックが予期しない動作をする場合があります。

例

次の関数は、statement-snapshot に独立性レベルを設定します。

```
SQLSetConnectAttr (dbc, SA_SQL_ATTR_TXN_ISOLATION, (SQLPOINTER*)
SA_SQL_TXN_STATEMENT_SNAPSHOT, 0);
```

ODBC は、独立性機能を使用して各種のデータベースロックオプションをサポートします。たとえば PowerBuilder では、データベースに接続するときに、トランザクションオブジェクトの Lock 属性を使用して独立性レベルを設定できます。Lock 属性は文字列で、次のように設定されます。

```
SQLCA.lock = "RU"
```

Lock オプションは、CONNECT が発生したときだけ有効になります。CONNECT の後に Lock 属性を変更しても、接続には影響しません。

トランザクション内の独立性レベルの変更

1 つのトランザクションの中の異なる部分に、別々の独立性レベルを設定したい場合、SQL Anywhere では、使用しているデータベースの独立性レベルをトランザクション中に変更できます。

トランザクション中に isolation_level オプションを変更すると、新しい設定は次の項目だけに反映されます。

●変更後に表示されたカーソル

●変更後に実行された文

トランザクションの途中で独立性レベルを変更し、トランザクションが実施するロック数を制御する必要がある場合があります。トランザクションで大きなテーブルを読み込む必要があるが、詳細な作業をするのは一部のローだけという場合もあります。矛盾が生じてもトランザクションには重大な影響を及ぼさない場合は、その大きなテーブルをスキャンする間は独立性レベルを低レベルに設定し、他の処理が遅れないようにしてください。

たとえば、1 つのテーブルまたはテーブルグループだけがシリアルアクセスを要求している場合などは、トランザクション中に独立性レベルを変更できます。

トランザクション中に独立性レベルを変更する例については、「チュートリアル : 幻ローの知識」917 ページを参照してください。

> **注意**
> テーブルヒントを使用して独立性レベルを設定することもできますが (レベル 0 ～ -3 のみ)、これは高度な機能であるため必要な場合にのみ使用してください。詳細については、「FROM 句」『SQL Anywhere サーバ SQL リファレンス』の WITH *テーブルヒント*に関する項を参照してください。

スナップショットアイソレーションを使用している場合の独立性レベルの変更

スナップショットアイソレーションを使用するときは、トランザクション内で独立性レベルを変更できます。これを行うには、isolation_level オプションの設定を変更するか、クエリ内の独立

性レベルに影響するテーブルヒントを使用します。いつでも statement-snapshot、readonly-statement-snapshot、独立性レベル 0 ～ 3 を使用できます。ただし、snapshot 以外の独立性レベルでトランザクションを開始した場合は、そのトランザクションで snapshot 独立性レベルを使用できません。トランザクションは、更新によって開始され、次の COMMIT または ROLLBACK まで続行します。最初の更新が snapshot 以外の独立性レベルで開始した場合、トランザクションがコミットかロールバックする前に snapshot 独立性レベルを使用しようとする文を実行すると、エラー -1065 (SQLE_NON_SNAPSHOT_TRANSACTION) が返されます。次に例を示します。

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';

BEGIN TRANSACTION
   SET OPTION isolation_level = 3;
   INSERT INTO Departments
      ( DepartmentID, DepartmentName, DepartmentHeadID )
      VALUES( 700, 'Foreign Sales', 129 );
   SET TEMPORARY OPTION isolation_level = 'snapshot';
   SELECT * FROM Departments;
```

独立性レベルの参照

CONNECTION_PROPERTY 関数を使用して、現在の接続の独立性レベルを表示できます。

前提条件

データベースに接続されている必要があります。

♦ タスク

● 次の文を実行します。

```
SELECT CONNECTION_PROPERTY( 'isolation_level' );
```

結果

現在の接続の独立性レベルが返されます。

参照

- 「CONNECTION_PROPERTY 関数 [システム]」『SQL Anywhere サーバ SQL リファレンス』
- isolation_level 接続プロパティ『SQL Anywhere サーバ データベース管理』
- 「isolation_level オプション」『SQL Anywhere サーバ データベース管理』

トランザクションのブロックとデッドロック

トランザクションの実行中、データベースサーバはローにロックをかけ、処理中のローが他のトランザクションからの干渉を受けないようにします。「ロック」は、許可する干渉の量とタイプを制御します。

SQL Anywhere では、「トランザクションブロック」の使用により、干渉をまったくなくすか制限して、トランザクションを同時に実行できます。トランザクションはロックを取得し、同時に実行されている他のトランザクションが特定のローを修正したり、アクセスしたりすることを防止

できます。このトランザクションブロックスキームは、いくつかのタイプの干渉を常に防止します。たとえば、テーブル内の特定ローを更新するトランザクションは、そのローのロックを取得し、他のトランザクションが同じローを同時に更新または削除できないようにします。

参照

- 「blocking_others_timeout オプション」『SQL Anywhere サーバ データベース管理』
- 「blocking_timeout オプション」『SQL Anywhere サーバ データベース管理』
- 「blocking オプション」『SQL Anywhere サーバ データベース管理』

トランザクションのブロック

あるトランザクションが操作を実行しようとしたにもかかわらず、他のトランザクションが保持するロックによって妨げられた場合、競合が発生します。この場合、操作を実行しようとしたトランザクションの進行は妨げられます。

ひとまとまりになったトランザクションが、どれも進行できない状態になることもあります。

参照

- 「デッドロック」883 ページ
- 「blocking_others_timeout オプション」『SQL Anywhere サーバ データベース管理』
- 「blocking_timeout オプション」『SQL Anywhere サーバ データベース管理』
- 「blocking オプション」『SQL Anywhere サーバ データベース管理』

ブロックオプション

2 つのトランザクションが、ある 1 つのローに対してそれぞれ読み込みロックをかけている場合、一方がローを変更しようとしたときにどうなるかは、データベースのブロックオプションの設定によって異なります。ローを修正するトランザクションは、他方のトランザクションをブロックしなければなりませんが、他方のトランザクションにブロックされている間はそれができません。

- ブロックオプションが On (デフォルト) の場合、書き込みをしようとするトランザクションは、もう一方のトランザクションが読み込みロックを解放するまで待機します。解放されると、書き込みが実行されます。

- ブロックオプションが Off の場合、書き込みをしようとする文はエラーを受け取ります。

ブロックオプションが Off の場合、文は待機せず終了し、行った部分的な変更はロールバックされます。この場合は、あとでもう一度トランザクションの実行を試みます。

ブロックは、独立性レベルが高くなると起こりやすくなります。ロックもチェックの回数も独立性とともに増えるからです。独立性レベルが高いと、通常は同時実行性が低下します。低下の度合いは、同時に実行するトランザクションによって異なります。

参照

- 「blocking オプション」『SQL Anywhere サーバ データベース管理』
- 「blocking_others_timeout オプション」『SQL Anywhere サーバ データベース管理』
- 「blocking_timeout オプション」『SQL Anywhere サーバ データベース管理』

デッドロック

トランザクションのブロックによって、「デッドロック」が起こる可能性があります。デッドロックとは、トランザクションのまとまりで、そのどれもが処理を進行できない状態をいいます。

デッドロックの理由

デッドロックが発生する理由は次の 2 つです。

- **環状ブロックの競合**　トランザクション A がトランザクション B にブロックされ、トランザクション B がトランザクション A にブロックされている状態。この状態から脱け出すには、どちらかのトランザクションをキャンセルします。同様の状況は 3 つ以上のトランザクションが環状にブロックされた場合にも発生します。

  トランザクションのデッドロックを排除するために、SQL Anywhere はデッドロックに関わっている接続を選択し、その接続でアクティブなトランザクションの変更をロールバックして、エラーを返します。SQL Anywhere は内部のヒューリスティックを使用してロールバックする接続を選択し、blocking_timeout によって決められた残りのブロックの待機時間が最も短い接続を優先します。すべての接続が永久に待機するように設定されている場合は、サーバによってデッドロックが検出された接続を、犠牲にする接続として選択します。

- **すべてのワーカがブロックされています。**　トランザクションがブロックされても、ワーカは放棄されたわけではありません。たとえば、データベースサーバが 3 つのワーカに設定されており、トランザクション A、B、C が現在要求を実行していないトランザクション D によってブロックされると、これ以上使用できるワーカがなくなるため、デッドロック状態が発生します。この状況は、スレッドデッドロックと呼ばれます。

  データベースサーバに *n* ワーカを設定したと想定します。*n*-1 の数のワーカがブロックされているときに最後のワーカをブロックしようとすると、スレッドデッドロックが発生します。データベースサーバのカーネルは、最後のワーカをブロックすることを許可できません。ブロックすることによって、すべてのワーカがブロックされ、データベースサーバがハングするためです。代わりに、データベースサーバは最後のワーカをブロックしようとしたタスクを終了し、その接続でアクティブなトランザクションの変更をロールバックして、エラー (SQLCODE -307、SQLSTATE 40W06) を返します。

  数十または数百の接続のあるデータベースサーバでは、データベースのサイズまたはブロックが原因で、多数の要求の実行時間が長くなる場合、スレッドデッドロックが発生することがあります。この場合、データベースサーバのマルチプログラミングレベルを高くすることが、適切な解決法となることがあります。また、アプリケーションの設計によって、過度な競合または意図しない競合が発生し、スレッドデッドロックの原因となる場合もあります。この場合、アプリケーションでより大きなデータセットを使用するようにすると、問題が悪化することがあります。データベースサーバのマルチプログラミングレベルを高くしても、問題は解決されないことがあります。

サーバが使用するデータベーススレッドの数は、個々のデータベースの設定によって異なります。

参照

- 「log_deadlocks オプション」『SQL Anywhere サーバ データベース管理』
- 「blocking_timeout オプション」『SQL Anywhere サーバ データベース管理』
- 「sa_report_deadlocks システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「スレッド動作」『SQL Anywhere サーバ データベース管理』
- 「すべてのスレッドがブロックされています。」『エラーメッセージ』
- 「マルチプログラミングレベルのデータベースサーバ設定」『SQL Anywhere サーバ データベース管理』

デッドロックでブロックされるユーザを決定する方法

sa_conn_info システムプロシージャを使用して、どの接続がデッドロックでブロックされているかを判別できます。このプロシージャは、接続ごとに 1 つのローで構成される結果セットを返します。結果セットのカラムの 1 つには、接続がブロックされているかどうか、およびブロックされている場合は、どの接続でブロックされているかがリストされます。結果セットは、接続がブロックされているかどうかと、それをブロックしている接続を示します。

デッドロックが発生した場合は、デッドロックイベントを使用してアクションを実行することもできます。イベントハンドラでは、sa_report_deadlocks プロシージャを使用して、デッドロックが発生するに至った状況に関する情報を取得できます。データベースサーバからデッドロックの詳細を取り出すには、log_deadlocks オプションを使用して RememberLastStatement 機能を有効にします。

次の例では、デッドロックが発生した場合にその情報を取得するために使用できるテーブルとシステムイベントの設定方法を示します。アプリケーションで頻繁にデッドロックが発生する場合は、アプリケーションプロファイリングを使用するとデッドロックの原因の究明に役立ちます。

例：デッドロックの発生時のアクションの実行

1. 次のように、sa_report_deadlocks システムプロシージャが返すデータを格納するテーブルを作成します。

```
CREATE TABLE DeadlockDetails(
  deadlockId INT PRIMARY KEY DEFAULT AUTOINCREMENT,
  snapshotId BIGINT,
  snapshotAt TIMESTAMP,
  waiter INTEGER,
  who VARCHAR(128),
  what LONG VARCHAR,
  object_id UNSIGNED BIGINT,
  record_id BIGINT,
  owner INTEGER,
  is_victim BIT,
  rollback_operation_count UNSIGNED INTEGER );
```

2. デッドロックが発生したときに起動するイベントを作成します。

このイベントにより、sa_report_deadlocks システムプロシージャの結果がテーブルにコピーされ、管理者にデッドロックが通知されます。

```
CREATE EVENT DeadlockNotification
TYPE Deadlock
HANDLER
BEGIN
 INSERT INTO DeadlockDetails WITH AUTO NAME
 SELECT snapshotId, snapshotAt, waiter, who, what, object_id, record_id,
       owner, is_victim, rollback_operation_count
   FROM sa_report_deadlocks ();
 COMMIT;
 CALL xp_startmail ( mail_user ='George Smith',
                    mail_password ='mypwd' );
 CALL xp_sendmail( recipient='DBAdmin',
                  subject='Deadlock details added to the DeadlockDetails table.' );
 CALL xp_stopmail ( );
END;
```

3. log_deadlocks オプションを On に設定します。

```
SET OPTION PUBLIC.log_deadlocks = 'On';
```

4. 最後に実行された文のロギングを有効にします。

```
CALL sa_server_option( 'RememberLastStatement', 'YES' );
```

参照

- 「log_deadlocks オプション」『SQL Anywhere サーバ データベース管理』
- 「sa_report_deadlocks システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_server_option システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_conn_info システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE EVENT 文」『SQL Anywhere サーバ SQL リファレンス』
- 「チュートリアル：デッドロックの診断」256 ページ

Sybase Central からデッドロックの表示

log_deadlocks オプションが ON に設定されているため、データベースで発生したデッドロックの図を表示することもできます。

前提条件

MONITOR と SET ANY SYSTEM OPTION のシステム権限が必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. Sybase Central の左ウィンドウ枠でデータベースを選択し、**[ファイル]** » **[オプション]** をクリックします。
3. log_deadlocks オプションをオンにします。
   a. **[オプション]** リストで **log_deadlocks** をクリックします。

b. **[値]** フィールドで **On** をクリックします。
c. **[恒久的な設定を行う]** をクリックします。
d. **[閉じる]** をクリックします。

4. 右ウィンドウ枠で、**[デッドロック]** タブをクリックします。

結果

発生したデッドロックを示すデッドロック図が表示されます。デッドロック図の各ノードは接続を表し、デッドロックが発生した接続の詳細、ユーザ名、デッドロックの発生時に接続が実行しようとした SQL 文が示されます。デッドロックには、接続デッドロックとスレッドデッドロックの 2 種類があります。接続デッドロックの特徴として、ノードの循環依存性があります。スレッドデッドロックは、循環依存性で接続されていないノードによって示され、ノード数は、データベースのスレッド上限値に 1 を足した数になります。

次の手順

アプリケーションプロファイリングを使用して、デッドロックの原因を診断できます。

参照

- 「log_deadlocks オプション」『SQL Anywhere サーバ データベース管理』
- 「トランザクションのブロックとデッドロック」881 ページ
- 「デッドロックでブロックされるユーザを決定する方法」884 ページ

ロックの仕組み

ロックは、複数のトランザクションを同時に実行しているときにデータの整合性を保護する同時制御メカニズムです。SQL Anywhere では、2 つの接続によって同じデータが同時に変更されないようにするために、また変更処理の最中に他の接続によってデータが読み込まれないようにするために、自動的にロックが適用されます。ロックによって、更新中の情報が保護され、クエリ結果の一貫性が高められます。

データベースサーバは自動的にこれらのロックを設定するので、明示的な指示は必要ありません。あるトランザクションによって獲得されたすべてのロックは、たとえば COMMIT 文または ROLLBACK 文によってそのトランザクションが完了するまで、データベースサーバで保持されます。このルールには 1 つのみ例外があります。

ローにアクセスしているトランザクションは、ロックを保持しているといえます。ロックの種類により、他のトランザクションのそのローへのアクセスは限定されるか、まったくできなくなります。

参照

- 「ロック期間」902 ページ

ロックの種類

データベースの一貫性を実現し、トランザクション間で適切な独立性レベルをサポートするために、SQL Anywhere では次の種類のロックを使用します。

- **スキーマロック**　スキーマを変更できる機能を制御します。たとえば、テーブルの構造を変更するために新しいカラムを挿入するトランザクションはテーブルをロックして、他のトランザクションがスキーマの変更による影響を受けないようにできます。このような場合は、他のトランザクションのアクセスを制限してエラーを回避することが不可欠です。

- **ローロック**　ローレベルで同時実行しているトランザクション間で一貫性を実現するために使用されます。たとえば、トランザクションは特定のローをロックして、他のトランザクションがそのローを変更しないようにできます。また、ローを修正するためにロックするのであれば、そのローに書き込みロックを設定する必要があります。同時実行性を最大化するために、ローのキーとキーではない部分を別々にロックできます。ローのキーではないカラムを更新しても、そのローを参照する外部ローの挿入と削除は妨げられません。

- **テーブルロック**　テーブルレベルで同時実行しているトランザクション間で一貫性を実現するために使用されます。たとえば、LOCK TABLE ... IN EXCLUSIVE MODE または REFRESH MATERIALIZED VIEW ... WITH EXCLUSIVE MODE 文を実行すると、排他テーブルロックを取得します。

- **位置ロック**　テーブルの逐次スキャンまたはインデックススキャンにおける一貫性を実現するために使用されます。通常、トランザクションは、インデックスによって指定された順序に従ってローをスキャンするか、ローを逐次スキャンします。いずれの場合も、スキャン位置にロックをかけることができます。たとえば、インデックスにロックをかけると、別のトランザクションが特定の値や値の範囲を持つローを挿入しないようにできます。

スキーマロックには、スキーマの変更によって、実行中のトランザクションに不注意に影響を与えないようにするメカニズムが備わっています。ローロック、テーブルロック、位置ロックのそれぞれには独自の目的がありますが、これらは相互に機能します。それぞれの種類のロックは、特定の矛盾を防ぎます。選択した独立性レベルに応じて、データベースサーバはこれらのロックのいくつかまたはすべての種類を使用して必要な一貫性レベルを維持します。

ロック期間

ロックのクラスが異なると、保持される期間も異なることがあります。

- **位置**　特定のローにおける読み込みロックのような短期間のロックで、独立性レベル 1 でカーソルの安定性を実装するために使用される

- **トランザクション**　トランザクションの終了まで保持されるローロック、テーブルロック、位置ロック

- **接続**　WITH HOLD カーソルの使用時に作成されるスキーマロックのように、トランザクションが終了しても保持されるスキーマロック

ロック情報の取得方法

データベースのロックの問題を診断するために、ロックされているローの内容を調べると役に立つ場合があります。データベースで現在保持されているロックを確認するには、sa_locks システムプロシージャを使用します。または、Sybase Central の **[ロック]** タブを使用します。どちらの方法でも、ロックを保持している接続、ロックの期間、ロックの種類などの必要な情報を得ることができます。

> **注意**
> データベースのロックは一時的なものであるため、Sybase Central で参照できるローや sa_locks システムプロシージャによって返されるローは、クエリの完了時にはすでに存在しない場合があります。

Sybase Central を使用したロックの表示

ロックは Sybase Central で表示できます。左ウィンドウ枠のデータベースを選択し、右ウィンドウ枠の **[ロック]** タブをクリックします。このタブでは、それぞれのロックに対する接続 ID、ユーザ ID、テーブル名、ロックタイプ、ロック名が表示されます。

sa_locks システムプロシージャを使用したロックの表示

sa_locks システムプロシージャの結果セットには、ロックが参照するテーブルのローを識別できる row_identifier カラムが含まれます。ロックされたローに格納されている実際の値を判断するために、ジョイン述部でテーブルの ROWID を使用して、sa_locks システムプロシージャの結果を特定のテーブルにジョインできます。次に例を示します。

```
SELECT S.conn_id, S.user_id, S.lock_class, S.lock_type, E.*
  FROM sa_locks() S JOIN Employees E WITH( NOLOCK )
    ON ROWID(E) = S.row_identifier
  WHERE S.table_name = 'Employees';
```

> **注意**
> NOLOCK テーブルヒントを指定する必要はない場合があります。ただしクエリが 0 以外の独立性レベルで発行された場合、ロックが解放されるまでこのクエリがブロックされることがあり、この確認方法の便利さは低下してしまいます。

参照

- 「sa_locks システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』
- 「FROM 句」『SQL Anywhere サーバ SQL リファレンス』
- 「ROWID 関数 [その他]」『SQL Anywhere サーバ SQL リファレンス』

スキーマロック

スキーマロックは、データベーススキーマへの変更を直列化し、またテーブルを使用するトランザクションがスキーマの変更による影響を受けないようにします。たとえば共有スキーマロックを使用すると、別の接続上のオープンカーソルでテーブルを読み取り中に、ALTER TABLE 文によってそのテーブルからカラムが削除されるのを防ぐことができます。

スキーマロックには、共有と排他の 2 つのクラスがあります。

共有ロック

共有スキーマロックは、トランザクションがデータベース内のテーブルを直接的または間接的に参照するときに取得されます。共有スキーマロックは他の共有スキーマロックと競合しません。同時に同じテーブルで共有スキーマロックを取得できるトランザクションの数に制限はありません。共有スキーマロックは、トランザクションが COMMIT または ROLLBACK で完了するまで保持されます。

共有スキーマロックを保持する接続は、変更が他の接続と競合しない場合に、テーブルデータを変更できます。テーブルスキーマは共有 (読み込み) モードでロックされる。

排他ロック

排他スキーマロックは、テーブルのスキーマが修正されるときに取得されます。スキーマは、通常は DDL 文を使用して修正されます。修正前にテーブルで排他スキーマロックを取得する DDL 文には、ALTER TABLE 文などがあります。あるテーブルに対して一度に 1 つの接続だけが、排他スキーマロックを取得できます。他のすべての接続は、テーブルのスキーマをロック (共有または排他) しようとしても、ブロックされるかエラーで失敗します。独立性レベル 0 (最も制限が少ない独立性レベル) で実行している接続は、スキーマが排他モードでロックされたテーブルからローを読み取ろうとするとブロックされます。

テーブルスキーマの排他ロックを保持する接続のみがテーブルデータを変更できます。テーブルスキーマは、単一接続の排他的使用のためにロックされる。

ローロック

ローロックは、更新内容の消失のようなトランザクションの矛盾を防ぎます。ローロックでは、暗黙的または明示的な COMMIT 文を発行して変更をコミットするか、ROLLBACK 文で変更をアボートすることによりトランザクションが完了するまで、そのトランザクションによって修正されるローは別のトランザクションによって修正されません。

ローロックには、読み込み (共有) ロック、書き込み (排他) ロック、意図的ロックという 3 つのクラスがあります。データベースサーバは、トランザクションごとにこれらのロックを自動的に取得します。

読み込みロック

トランザクションがローを読み込むと、トランザクションのアイソレーションレベルは読み込みロックが取得されているかどうかを判断します。一度読み込みロックのかかったローに対しては、他のどのトランザクションも書き込みロックを取得できません。読み込みロックが取得されると、ローの読み込み中は、別のトランザクションはそのローを修正または削除しません。任意のローに同時に取得できる読み込みロックの数に制限はありません。そのため、読み込みロックは、共有ロックまたは非排他ロックと呼ばれることもあります。

読み込みロックは、保持される期間が異なることがあります。独立性レベル 2 と 3 では、トランザクションが取得したどの読み込みロックも、トランザクションが COMMIT または ROLLBACK によって完了するまで保持されます。これらの読み込みロックは、長期間の読み込みロックと呼ばれます。

独立性レベル 1 で実行されるトランザクションの場合、データベースサーバはカーソルが位置するローで短期間の読み込みロックを取得します。アプリケーションがカーソルをスクロールすると、カーソルが直前に位置していたローで短期間の読み込みロックは解放され、新しい短期間の読み込みロックがその次のローで取得されます。この技術は「カーソルの安定性」と呼ばれます。アプリケーションは現在のローで読み込みロックを保持するため、アプリケーションがそのローからカーソルを移動するまで、別のトランザクションがそのローに対して変更を加えることができません。カーソルが複数のテーブルを伴うクエリに対する場合は、複数のロックを取得できます。短期間の読み込みロックは、カーソル内の位置が要求 (通常は、これらの要求はアプリケーションによって発行される FETCH 文) 間で維持される必要がある場合だけ取得されます。たとえば SELECT COUNT(*) クエリの処理時は短期間の読み込みロックは取得されません。この文で開かれているカーソルが、ベーステーブルの特定ローに位置することがないためです。この場合、データベースサーバは、コミットされた読み出しのセマンティック、つまりこの文で処理されるローは他のトランザクションによってコミットされたことを保証すれば良いことになります。

独立性レベル 0 (コミットされない読み出し) で実行されるトランザクションの場合は、長期間または短期間の読み込みロックを取得しないため、他のトランザクションと競合しません (排他スキーマロックの場合を除く)。ただし、独立性レベル 0 のトランザクションは、同時に実行している他のトランザクションによって加えられたコミットされていない変更を処理することがあります。コミットされていない変更を処理しないようにするには、スナップショットアイソレーションを使用します。

書き込みロック

トランザクションがローを追加、更新、削除するときには、「書き込みロック」が設定されます。この動作は、独立性レベル 0 やスナップショットアイソレーションのレベルを含む、どの独立性レベルのトランザクションでも当てはまります。書き込みロックが設定されると、他のトランザクションはそのローに対して読み込みロック、意図的ロック、書き込みロックのいずれも取得できません。あるローに対して排他的なロックを保持できるトランザクションは一度に 1 つだけであるため、書き込みロックは排他ロックとも呼ばれます。他のトランザクションが同じローに何らかの種類のロックをかけている間は、書き込みロックを取得することはできません。同様に、トランザクションが書き込みロックを取得すると、他のトランザクションによるそのローへのロック要求は拒否されます。

意図的ロック

意図的ロックは更新を意図したロックとも呼ばれ、特定のローを修正する意図を表します。意図的ロックは、トランザクションが次の場合に取得されます。

- FETCH FOR UPDATE 文を発行する。
- SELECT...FOR UPDATE BY LOCK 文を発行する。
- ODBC アプリケーションで SQL_CONCUR_LOCK を同時実行性の基準として使用する (SQLSetStmtAttr ODBC API 呼び出しの SQL_ATTR_CONCURRENCY パラメータを使用して設定する)。
- SELECT...FROM T WITH (UPDLOCK) 文の発行

意図的ロックは、読み込みロックと競合しないため、意図的ロックを取得しても、他のトランザクションが同じローを読み込むことをブロックしません。ただし、意図的ロックは、他のトラン

ザクションが同じローで意図的ロックまたは書き込みロックを取得することを妨げるため、更新に先立って他のトランザクションによってローが変更されることはありません。

スナップショットアイソレーションを使用するトランザクションによって意図的ロックが取得される場合とは、そのローがデータベースで未修正のローであり、かつすべての同時実行トランザクションに共通である場合に限られます。ただし、ローがスナップショットのコピーである場合は、元のローが別のトランザクションによってすでに修正されているため、意図的ロックは取得されません。スナップショットトランザクションによってそのローを更新しようとしても失敗となり、スナップショットの更新競合エラーが返されます。

参照

● 「スナップショットアイソレーション」868 ページ

テーブルロック

ローのロックのほかに、SQL Anywhere ではテーブルのロックもサポートしています。テーブルロックは、テーブルのスキーマにロックをかけるスキーマロックと異なり、テーブル内のすべてのローに対してロックをかけます。テーブルロックには、共有、書き込みを意図、排他という 3 つの種類があります。

テーブルロックは、トランザクションが COMMIT または ROLLBACK で終了したときに解放されます。

次の表に、競合するテーブルロックの組み合わせを示します。

	共有	意図的	排他
共有		競合	競合
意図的	競合		競合
排他	競合	競合	競合

共有テーブルロック

共有テーブルロックでは、複数のトランザクションがベーステーブルのデータを読み込めます。ベーステーブルに共有テーブルロックを持つトランザクションは、他のトランザクションが修正中のローにロックをかけていない場合に、テーブルを変更できます。

共有テーブルロックは、たとえば LOCK TABLE...IN SHARED MODE 文を実行することで取得できます。REFRESH MATERIALIZED VIEW や REFRESH TEXT INDEX 文も WITH SHARE MODE 句をサポートするため、再表示の操作中に基本となるテーブルの共有テーブルロックを作成する場合に使用できます。

書き込みを意図したテーブルロック

書き込みを意図したテーブルロックは意図的テーブルロックとも呼ばれます。意図的テーブルロックは、トランザクションによってローの書き込みロックが最初に取得されるときに、暗黙的に取得されます。つまり、意図的テーブルロックは、ローの更新、挿入または削除時に取得され

ます。共有テーブルロックと同様に、意図的テーブルロックは、トランザクションが COMMIT または ROLLBACK で完了するまで保持されます。意図的テーブルロックは、共有テーブルロックや排他テーブルロックと競合しますが、他の意図的テーブルロックとは競合しません。

排他テーブルロック

排他テーブルロックは、新しいデータの挿入など、他のトランザクションがテーブルのスキーマまたはデータを変更するのを防ぎます。排他スキーマロックとは異なり、独立性レベル 0 で実行しているトランザクションは、テーブルロックが排他的に設定されているテーブルのローを読み込むことはできます。一度に 1 つのトランザクションだけが、テーブルに対して排他ロックをかけられます。排他テーブルロックは、その他のすべてのテーブルロックとローロックとの間で競合します。

LOAD TABLE 文を使用すると、排他テーブルロックを暗黙的に取得します。

排他テーブルロックは、LOCK TABLE...IN EXCLUSIVE MODE 文を使用することで、明示的に取得します。REFRESH MATERIALIZED VIEW や REFRESH TEXT INDEX 文も WITH EXCLUSIVE MODE 句を提供するため、再表示の操作中に基本となるテーブルの排他テーブルロックを指定する場合に使用できます。

参照

- 「スキーマロック」888 ページ
- 「LOCK TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ SQL リファレンス』
- 「REFRESH TEXT INDEX 文」『SQL Anywhere サーバ SQL リファレンス』

位置ロック

ローロックのほかに、SQL Anywhere では幻または幻ローが存在するための異常事態を避けるために設計された、キー範囲のロック形式である位置ロックを実装しています。位置ロックが関係するのは、独立性レベル 3 で実行しているトランザクションをデータベースサーバが処理しているときです。

独立性レベル 3 で実行しているトランザクションは直列化可能です。独立性レベル 3 のトランザクションの動作は、他のトランザクションによって同時に発生する更新アクティビティの影響を受けないはずです。実際に独立性レベル 3 のトランザクションは、計算結果に影響を与える可能性のあるローが取り込まれる INSERT または UPDATE (つまり幻) による影響を受けることはありません。SQL Anywhere では位置ロックを使用して、そのような更新が発生することを防ぎます。位置ロックは、独立性レベル 2 (繰り返し可能読み出し) と独立性レベル 3 とを区別する追加のロック処理です。

幻ローが作成されないように、テーブルの物理スキャン内で位置のロックが取得されます。逐次スキャンの場合は、現在のローのロー識別子に基づいてスキャン位置が決定されます。インデックススキャンの場合は、現在のローのインデックスキー値に基づいてスキャン位置が決定されます (インデックスキー値は、ユニークな場合もそうでない場合もある)。トランザクションはスキャン位置をロックすることによって、他のトランザクションが、ローの順序付けに使用される特定範囲の値に関する挿入を行うことを防ぐことができます。この動作は、INSERT 文やインデックス付き属性の値を変更する UPDATE 文に適用されます。スキャン位置がロックされた場

合、UPDATE 文は、直後に INSERT 要求が続く、インデックスエントリに対する DELETE 要求であると見なされます。

SQL Anywhere でサポートされる位置ロックには、幻ロックと挿入の 2 種類があります。どちらの種類も共有ロックであり、複数のトランザクションが同じローで同じ種類のロックを取得できます。ただし、幻ロックと対幻ロックは競合します。

幻ロック

幻ロックは対挿入ロックとも呼ばれ、その後で他のトランザクションによって幻ローが作成されないように、スキャン位置に設定されます。幻ロックが取得されると、テーブル内で、対挿入ロックがかかっているローの直前に、他のトランザクションがローを挿入することを防止します。幻ロックは長期間のロックで、トランザクションの終了まで保持されます。

幻ロックは、独立性レベル 3 で実行しているトランザクションのみが取得できます。独立性レベル 3 は、幻に関して一貫性を保証する唯一の独立性レベルです。

インデックススキャンでは、幻ロックはインデックスを介して読み込まれる各ローで取得されます。また、条件を満たすインデックス範囲の最後にインデックスが挿入されることを防ぐため、インデックススキャンの最後に幻ロックが 1 つ追加取得されます。インデックススキャンで幻ロックを使用すると、テーブルに新しいローが挿入されたり、インデックス付きの値 (幻ロックの対象となるポイントにインデックスエントリが作成される原因となる) が更新されたりすることが原因で、幻が作成されないようになります。

逐次スキャンでは、挿入処理によって結果セットが変更されないように、幻ロックはテーブルの行ごとに取得されます。独立性レベル 3 のスキャンにより、データベースの同時実行性が悪影響を受けることがあります。1 つ以上の幻ロックは挿入ロックと競合し、1 つ以上の読み込みロックは書き込みロックと競合しますが、幻ロック／挿入ロックと読み込みロック／書き込みロックの間に相互作用はありません。たとえば、書き込みロックは読み込みロックのかかったローにかけることはできませんが、幻ロックだけがかかったローにはかけることができます。この柔軟性のため、データベースサーバでは多くのオプションを利用できます。しかしそのために、幻ロックをかける場合は、読み込みロックの設定に特別な注意が要求されます。これを怠ると、他のトランザクションがローを削除してしまう可能性があります。

挿入ロック

挿入ロックは対幻ロックとも呼ばれ、ローを挿入する権利を確保するためにスキャン位置に設定される短期間のロックです。ロックは、その挿入の期間だけ保持されます。ローがデータベースページ内に正しく挿入されると、一貫性を確保するためにそのローは書き込みロックがかけられ、挿入ロックは解放されます。トランザクションがあるローに対して挿入ロックを設定すると、他のトランザクションはそのローに幻ロックを設定できません。データベースサーバは、新しい要求によって発生する可能性のある、アクティブな接続による独立性レベル 3 のスキャン操作を予期する必要があるため、挿入ロックが必要です。幻ロックと挿入ロックは、同じトランザクションによって保持される場合は相互に競合しません。

ロックの競合

SQL Anywhere は、スキーマロック、ローロック、テーブルロック、位置ロックを必要に応じて使用し、必要な一貫性レベルを確保します。特定のロックの使用を明示的に要求する必要はあり

ません。ただし、要件に最も合う独立性レベルを選択することで維持される一貫性レベルを管理する必要があります。ロックの種類を知っておくと、独立性レベルの選択、および各レベルのパフォーマンスへの影響を理解する上で便利です。1 つのトランザクションがロックを取得することで自分自身をブロックすることはできないことに注意してください。ロックの競合は、2 つ以上のトランザクション間でのみ発生します。

ロックの競合が発生するのは、一般に、あるトランザクションが、別のトランザクションがロックをかけているローに排他ロックをかけようとした場合や、別のトランザクションが排他ロックをかけているローに共有ロックをかけようとした場合です。このような場合、その「別のトランザクション」の完了を待たなければなりません。待機しなければならないトランザクションは、もう一方のトランザクションに「「ブロック」」されます。

データベースサーバが、トランザクションの即時処理を禁止するロックの競合を認識すると、トランザクションの実行を一時停止するか、またはトランザクションを終了し、変更をロールバックし、エラーを返すことができます。ブロックオプションを設定して、その手段を制御します。ブロックを **On** に設定すると、2 番目のトランザクションが待機します。

どのロックが競合するか

4 つのロックはそれぞれ特定の目的がありますが、すべての種類が干渉し合うため、トランザクション間でロックの競合が発生する原因となります。データベースの一貫性を確保するために、一度に 1 つのトランザクションだけが 1 つのローを変更できます。2 つのトランザクションが同時に 1 つの値を変更できてしまうと、1 つの値が 2 つの異なる値に変更されることになります。このため、ローの書き込みロックは排他ロックであることが重要です。これに対して、複数のトランザクションが 1 つのローを読んでも問題は生じません。ローを変更するわけではないので、競合することはありません。このため、ローの読み込みロックは多くの接続間で共有されていても構いません。

次の表に、競合するロックの組み合わせを示します。スキーマロックはローに適用されないため、含めてありません。

ローロック	readpk	read	intent	writenopk	write
「readpk」					競合
「read」				競合	競合
「intent」			競合	競合	競合
「writenopk」		競合	競合	競合	競合
「write」	競合	競合	競合	競合	競合

テーブルロック	共有ロック	意図的ロック	排他ロック
「共有ロック」		競合	競合
「意図的ロック」	競合		競合

テーブルロック	共有ロック	意図的ロック	排他ロック
「排他ロック」	競合	競合	競合

位置ロック	幻ロック	意図的ロック
「幻ロック」		競合
「意図的ロック」	競合	

参照

- 「sa_locks システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

クエリ時のロック

ユーザが SELECT 文を入力したときに SQL Anywhere が使用するロックは、トランザクションの独立性レベルによって異なります。すべての SELECT 文は、独立性レベルに関係なく、参照先テーブルで共有スキーマロックを取得します。

独立性レベル 0 の SELECT 文

独立性レベル 0 で SELECT 文を実行するときは、ロックオペレーションは必要ありません。各トランザクションは他のトランザクションによる変更から保護されません。プログラマーまたはデータベースユーザは、この制限を念頭においてこのようなクエリの結果を解釈する責任があります。

独立性レベル 1 の SELECT 文

独立性レベル 1 でトランザクションを実行する場合、SQL Anywhere は独立性レベル 0 で実行するときほど多くのロックを使用しません。それぞれのレベルでデータベースサーバのオペレーションが異なる点は、2 つしかありません。

オペレーションの最初の違いはロックの設定とは無関係で、むしろロックへの配慮に関するものです。独立性レベル 0 では、別のトランザクションが書き込みロックを取得しても、トランザクションはすべてのローを読み込むことができます。一方独立性レベル 1 のトランザクションは、各ローを読み込む前に書き込みロックがかかっているかをチェックします。書き込みロックがかかっているローは読み込むことができません。この場合、ダーティデータを読み込むことになるからです。READPAST ヒントを使用すると、サーバは書き込みロックがかかっているローを無視できます。ただし、トランザクションのブロックがなくなると、READPAST ヒントのセマンティックは独立性レベル 1 のセマンティックと一致しなくなります。

オペレーションの 2 番目の違いは、カーソル安定性に影響します。カーソル安定性は、カーソルの現在のローに短期間の読み込みロックを設定して達成されます。この読み込みロックはカーソルを移動すると解放されます。カーソルの内容がジョインの結果を示している場合は、複数のローが影響を受けます。この場合、データベースサーバはカーソルの現在のローに情報を提供したすべてのローに短期間の読み込みロックをかけ、カーソルの別のローが現在のローになるとこれらのロックを解放します。

独立性レベル 2 の SELECT 文

独立性レベル 2 では、データベースサーバはそのオペレーションを修正し、繰り返し可能読み出しのセマンティックを保証します。SELECT 文がテーブルのすべてのローから値を返す場合、データベースサーバはローを読み込むときに各ローに読み込みロックをかけます。SELECT に WHERE 句や結果におけるローを制限する他の条件が含まれている場合は、データベースサーバは各ローを読み込み、ローの値が条件を満たしているかをテストし、条件を満たすローに読み込みロックをかけます。取得された読み込みロックは、長期間の読み込みロックであり、暗黙的または明示的な COMMIT 文または ROLLBACK 文によってトランザクションが完了するまで保持されます。独立性レベル 1 と同じように、独立性レベル 2 ではカーソル安定性が保証され、ダーティリードは許可されません。

独立性レベル 3 の SELECT 文

独立性レベル 3 では、データベースサーバはすべてのトランザクションが直列化可能であることを確認する必要があります。特に、独立性レベル 2 での要件に加えて、同じ文を再実行するとすべての環境で同じ結果を返すことが保証されるように、幻ローを防ぐ必要があります。

この要件を満たすために、データベースサーバは読み込みロックと幻ロックを使用します。独立性レベル 3 で SELECT 文を実行すると、データベースサーバは結果セットの計算で処理される各ローで読み込みロックを取得します。こうすることで、そのトランザクションが完了するまで他のトランザクションがそれらのローを修正できないようにします。

この要件は、データベースサーバが独立性レベル 2 で実行するオペレーションと似ていますが、これらのローが SELECT 文の WHERE 句、ON 句、または HAVING 句の述部を満たすかどうかに関係なく、読み込まれた各ローにロックをかけなければならない点が異なります。たとえば、販売部のすべての従業員名を選択する場合、サーバはトランザクションが独立性レベル 2 または 3 のどちらで実行されているかに関係なく、販売部の従業員に関する情報が含まれているすべてのローにロックをかける必要があります。ただし、独立性レベル 3 では、販売部に所属しない従業員のローにも読み込みロックをかける必要があります。そうでない場合、最初のトランザクションが実行されている間に、別のトランザクションが別の従業員を販売部に移動する可能性があります。

読み込まれた各ローに読み込みロックをかける必要がある場合、次の 2 つの影響があります。

- データベースサーバは、独立性レベル 2 で必要とされるロックよりも多くのロックをかけなければならない場合がある。取得される幻ロックの数は、スキャンのために取得される読み込みロックの数よりも 1 つ多くなります。倍増したロックのオーバヘッドは、要求の実行時間に追加されます。

- 各ローの読み込みで読み込みロックを取得すると、同じテーブルに対するデータベース更新オペレーションの同時実行性に悪影響がある。

データベースサーバが取得する幻ロックの数には大きな幅があり、クエリオプティマイザによって選択された実行方式によって異なります。SQL Anywhere クエリオプティマイザは、システム全体の同時実行性に悪影響を与える可能性があるため、独立性レベル 3 での逐次スキャンを回避しようとします。しかし、このようなオプティマイザの機能は、文の述部と、参照先テーブルで利用できる適切なインデックスに依存します。

たとえば、Employee ID 123 の従業員に関する情報を選択したいとします。Employee ID は従業員テーブルのプライマリキーであるため、ローを効率的に検索するために、クエリオプティマイザ

がプライマリキーインデックスを使用するインデックス方式を選択しようとするのはほぼ確実です。さらに、プライマリキーの値はユニークであるため、別のトランザクションが他の EmployeeID を 123 に変更する危険性もありません。サーバは、従業員 123 に関する情報を含むローに読み込みロックをかけるだけで、別の従業員にその ID 番号が割り当てられることを防止できます。

一方、販売部の全従業員を選択する場合は、読み込みロック以外のロックもかける必要があります。適切なインデックスがないため、データベースサーバは従業員テーブルの各ローを読み込み、各従業員が販売部に所属するかどうかをテストする必要があります。この場合は、テーブルの各ローに読み込みロックと幻ロックの両方を設定する必要があります。

SELECT 文とスナップショットアイソレーション

snapshot、statement-snapshot、または readonly-statement-snapshot で実行される SELECT 文では、読み込みロックを取得しません。これは、各スナップショットトランザクション (または文) は、以前のある時点における、コミットされた状態のデータベースのスナップショットを認識するためです。この特定の時点は、3 種類あるスナップショットアイソレーションのレベルのうちどれが文で使用されるかによって決まります。つまり、読み込みトランザクションが更新トランザクションをブロックしたり、更新トランザクションが読み込みトランザクションをブロックしたりすることはありません。そのため、スナップショットアイソレーションを使用すると、一貫性という明白な長所だけでなく、同時実行性という重要な長所を得ることができます。ただし、トレードオフとして、スナップショットアイソレーションは非常にコストがかかることがあります。これは、スナップショットアイソレーションの一貫性保証では、同時に実行される他のトランザクションのために、変更されたローのコピーを保存、追跡、(最終的に) 削除する必要があるためです。

参照

- 「FROM 句」『SQL Anywhere サーバ SQL リファレンス』

挿入時のロック

INSERT オペレーションは新しいローを作成します。SQL Anywhere では、データ整合性を確保するために、挿入時に各種のロックを利用します。どの独立性レベルであっても実行している INSERT 文では、次の操作手順が発生します。

1. テーブルで共有スキーマロックを保持していない場合は、取得します。
2. テーブルで書き込みを意図したテーブルロックを保持していない場合は、取得します。
3. 新しいローを格納するために、ページでロックされていない位置を検索します。ロック競合を最小限に抑えるために、データベースサーバは、削除された (しかし、まだコミットされていない) ローによって利用可能になった領域をただちに再利用しません。新しいローを確保するために、新しいページがテーブルに割り当てられることがあります。また、データベースファイルのサイズが増大することがあります。
4. 新しいローに値を入れます。

5. ローを追加するテーブルに挿入ロックをかけます。挿入ロックは排他ロックであるため、一度挿入ロックをかけると、独立性レベル 3 の他のトランザクションは、幻ロックをかけて挿入をブロックすることができません。

6. 新しいローに書き込みロックをかけます。書き込みロックが取得されると、挿入ロックが解放されます。

7. テーブルにローを挿入します。ここで、独立性レベル 3 の他のトランザクションは初めて新しいローの存在に気が付きます。ただし、すでに書き込みロックがかかっているため、これらのトランザクションはそのローの修正や削除はできません。

8. 影響を受けるすべてのインデックスを更新し、必要に応じてユニークであることを確認します。プライマリキーの値はユニークである必要があります。他のカラムもユニークな値だけを含むように定義される場合があります。このようなカラムが存在する場合は、ユニーク性が検証されます。

9. テーブルが外部テーブルである場合は、プライマリテーブルの共有スキーマロックを保持していなければ取得し、挿入される外部キーカラムの値が NULL でない場合は、プライマリテーブルの一致するプライマリローで読み込みロックを取得します。データベースサーバは、挿入トランザクションで COMMIT が実行されたときにプライマリローが存在していることを保証する必要があります。この確認は、プライマリローの読み込みロックを取得して行います。読み込みロックをかけても他のトランザクションは自由にそのローを読むことができますが、削除や更新はできません。

   対応するプライマリローが存在しない場合は、参照整合性制約違反が発生します。

最後の手順の後、テーブルで定義された AFTER INSERT トリガが起動します。トリガ内の処理におけるロック動作は、アプリケーションの場合と同じです。トランザクションがコミット (すべての参照整合性制約が満たされる) またはロールバックされると、すべての長期間ロックが解放されます。

ユニーク性

特定のカラムまたはカラムの組み合わせに設定される値のすべてをユニークにすることができます。ユーザがあえて作成しなくても、データベースサーバがそのユニークなカラムに対するインデックスを作成し、それによって値のユニーク性を保証しています。

特に、プライマリキーの値はすべてユニークである必要があります。データベースサーバは、すべてのテーブルのプライマリキーに対するインデックスを自動的に作成します。プライマリキーに対するインデックスを作成するようデータベースサーバに要求しないでください。これをすると、重複するインデックスが作成されてしまいます。

オーファンと参照整合性

外部キーは、通常別のテーブルにあるプライマリキーまたは一意性制約を参照します。そのプライマリキーが存在しない場合、対応する外部キーは「オーファン」と呼ばれます。SQL Anywhere は、データベースに参照整合性に違反するローがないかを自動的に確認します。このプロセスを「参照整合性の検証」と呼びます。データベースサーバは、オーファン数をカウントし参照整合性を調べます。

wait_for_commit

データベースサーバに対し、トランザクションの終了まで参照整合性の検証を遅延するように指示できます。このモードでは、外部キーを含む 1 つのローを挿入し、次にプライマリキーを持たないプライマリローを挿入できます。この両方のオペレーションは同じトランザクションで実行する必要があります。

データベースサーバがコミット時間まで参照整合性の検証を遅延するように要求するには、wait_for_commit オプションを On に設定します。デフォルトでは、このオプションは Off に設定されます。ON にするには、次の文を実行します。

```
SET OPTION wait_for_commit = On;
```

新しい外部キー値が挿入されるときにサーバが一致するプライマリローを見つけられず、wait_for_commit が On の場合、サーバはオーファンとして挿入を許可します。孤立した外部ローの場合は、挿入時に次の手順が発生します。

- サーバは、プライマリテーブルで共有スキーマロックを保持していない場合は取得します。また、プライマリテーブルで書き込みを意図したロックを取得します。
- サーバは、プライマリテーブルに代理ローを挿入します。実際のローはプライマリテーブルに挿入されません。ただし、サーバはロックをかけるためにそのローのユニークなロー識別子を作成し、この代理ローで書き込みロックを取得します。次に、サーバはプライマリテーブルのプライマリキーインデックスに適切な値を挿入します。

トランザクションをコミットする前に、データベースサーバはトランザクションが作成したオーファン数をチェックし参照整合性が維持されているかを調べます。各トランザクションの終了時に、この数は 0 になっていなければなりません。

更新時のロック

データベースサーバは次の手順を使用して特定のレコードに含まれる情報を修正します。挿入時と同様に、独立性レベルに関係なくすべてのトランザクションで次の操作手順が発生します。

1. テーブルで共有スキーマロックを保持していない場合は、取得します。
2. 更新される各テーブルで書き込みを意図したテーブルロックを保持していない場合は、取得します。
   a. 更新されるテーブルごとに、テーブルにトリガがある場合は、必要に応じて OLD 値と NEW 値のテンポラリテーブルを作成します。
   b. 更新の候補となるローを識別します。ローがスキャンされている間は、ローはロックされます。

      独立性レベル 2 と 3 ではデフォルトのロック動作とは異なる次のような違いが発生します。読み込みロックではなく書き込みを意図したローレベルのロックが取得されます。また、書き込みを意図したロックは、更新の候補としては最終的には拒否されたローで取得される場合があります。

c. 手順 2.a で識別された候補となる各ローは、残りのシーケンスに従います。

3. 影響を受けるローに書き込みロックをかけます。

4. UPDATE 文により、影響を受けるそれぞれのカラムの値を更新します。

5. インデックス付きの値を変更した場合は、新しいインデックスエントリを追加します。ローの元のインデックスエントリは残りますが、削除済みのマークが付けられます。短期間の挿入ロックが保持されている間に、新しい値の新しいインデックスエントリが挿入されます。サーバは、必要に応じてインデックスのユニーク性を検証します。

6. 一意性違反が発生した場合は、ローの古い値と新しい値を格納するためにテンポラリ「保持」テーブルが作成されます。古い値と新しい値は保持テーブルにコピーされ、ベーステーブルのローは削除されます。DELETE トリガは起動されません。ローごとの処理が終わるまで、手順 7 ～ 9 は行いません。

7. ローの外部キー値が変更された場合、プライマリテーブルで共有スキーマロックを取得し、新しい外部キー値を挿入する手順に従います。

   同様に、必要に応じて wait_for_commit の手順に従います。

8. テーブルが参照整合性関係においてプライマリテーブルであり、かつ関係の UPDATE アクションが RESTRICT でない場合、外部テーブル (間) で共有スキーマロックを、各外部テーブルで書き込みを意図したテーブルロックをそれぞれ取得することで、外部テーブル内で影響のあるローを特定し、次に影響のあるすべてのローで書き込みロックを取得して、必要に応じてそれぞれのローを修正します。このプロセスは、参照整合性制約のネストした階層でカスケードされることがあります。

9. 必要に応じて AFTER ROW トリガを起動します。

保持テンポラリテーブルが必要だった場合には、最後の手順の後で、保持テンポラリテーブル内の各ローはベーステーブルに挿入されます (ただし、INSERT トリガは起動されません)。ローの挿入が成功した場合は、上記の手順 7 ～ 9 が実行され、古いロー値と新しいロー値は OLD および NEW テンポラリテーブルにコピーされて、AFTER STATEMENT UPDATE トリガによって、変更されたすべてのローが正しく処理されます。保持されたローがすべて処理された後、AFTER STATEMENT UPDATE トリガが順番に起動されます。COMMIT の実行時、サーバはこのトランザクションで生成されたオーファンの数が 0 であることを確認することで参照整合性を検証し、次にすべてのロックを解放します。

カラムの値を変更するために、多くの操作が必要になることがあります。データベースサーバが実行する作業量は、修正するカラムがプライマリキーまたは外部キーの一部でない場合は大幅に少なくなります。カラムをユニークと宣言したために、変更する値が明示的または暗黙的にインデックスに含まれていない場合も作業量は少なくて済みます。

UPDATE オペレーション中に参照整合性を確認するオペレーションは、INSERT 中に確認する場合よりも複雑になります。実際、プライマリキーの値を変更すると、オーファンが作成されることがあります。置換値を挿入すると、データベースサーバはもう一度オーファンをチェックしなければなりません。

参照

- 「挿入時のロック」897 ページ
- 「独立性レベルと一貫性」864 ページ

削除時のロック

DELETE オペレーションは、INSERT オペレーションとほとんど同じ手順を実行しますが、その順序は反対になります。挿入時や更新時と同様に、独立性レベルに関係なくすべてのトランザクションで次の操作手順が発生します。

1. テーブルで共有スキーマロックを保持していない場合は、取得します。

2. テーブルで書き込みを意図したテーブルロックを保持していない場合は、取得します。

   a. 更新の候補となるローを識別します。ローがスキャンされている間は、ローはロックされます。

      独立性レベル 2 と 3 ではデフォルトのロック動作とは異なる次のような違いが発生します。読み込みロックではなく書き込みを意図したローレベルのロックが取得されます。また、書き込みを意図したロックは、更新の候補としては最終的には拒否されたローで取得される場合があります。

   b. 手順 2.a で識別された候補となる各ローは、残りのシーケンスに従います。

3. 削除するローに書き込みロックをかけます。

4. 他のトランザクションから見えなくなるように、ローをテーブルから削除します。ローは、トランザクションがコミットされるまで破壊できません。ローを破壊すると、トランザクションをロールバックするオプションが削除されてしまうからです。削除されたローのインデックスエントリはトランザクションの完了まで残りますが、削除済みのマークが付けられます。これにより、他のトランザクションは同じローを再挿入できません。

5. テーブルが参照整合性関係においてプライマリテーブルであり、かつ関係の DELETE アクションが RESTRICT でない場合、外部テーブル (間) で共有スキーマロックを、各外部テーブルで書き込みを意図したテーブルロックをそれぞれ取得することで、外部テーブル内で影響のあるローを特定し、次に影響のあるすべてのローで書き込みロックを取得して、必要に応じてそれぞれのローを修正します。このプロセスは、参照整合性制約のネストした階層でカスケードされることがあります。

参照整合性に違反しない場合は、トランザクションをコミットできます。参照整合性を調べるために、データベースサーバは削除によって作成されたオーファンを追跡します。COMMIT の実行時、サーバはオペレーションをトランザクションログファイルに記録し、すべてのロックを解放します。

参照

- 「独立性レベルと一貫性」864 ページ

ロック期間

ロックは、通常、トランザクションが完了するまでトランザクションによって保持されます。この動作によって、他のトランザクションが変更を加えたことによって、元のトランザクションがロールバックできなくなることが回避されます。独立性レベル 3 では、トランザクションの直列可能性を保障するために、トランザクションが終了するまですべてのロックが保持されます。

トランザクションの終了まで保持されないロックは、カーソル安定性ロックのみです。このローロックは、対象のローがカーソルの現在のローである間、保持されます。多くの場合、カーソル安定性ロックの継続時間はトランザクションの存続時間よりも短くなりますが、WITH HOLD カーソルの場合、接続が存続する間、カーソル安定性ロックが保持されることがあります。

参照

- 「LOCK TABLE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「OPEN 文 [ESQL] [SP]」『SQL Anywhere サーバ SQL リファレンス』

独立性レベル選択のガイドライン

独立性レベルの選択は、アプリケーションが実行するタスクの種類によって異なります。この項では、選択のガイドラインを示します。

適切な独立性レベルを選択するには、一貫性と正当性のニーズと、同時に実行するトランザクションが妨げられずに処理を行うためのニーズのバランスを取る必要があります。トランザクションが 1 つのテーブルで 1 つまたは 2 つの特定の値しか使用しない場合は、数多くの大きなテーブルを検索したり、多くのローまたはテーブル全体をロックしたりするような、処理に非常に時間がかかるプロセスに比べると、トランザクションが妨げられる可能性はかなり少なくなります。

たとえば、トランザクションで銀行口座間の資金移動を行う場合、確実に正確な情報が返されるようにする必要があるでしょう。ただし、休止中の口座の割合の概算を計算する場合は、そのトランザクションが他のトランザクションを待つかどうかを配慮しません。また、データベースの他のユーザを妨げないようにするために、多少の精度を犠牲にします。

さらに、金銭の振り込みは、2 つの口座の残高を含む 2 つのローだけに影響するのに対して、概算を計算するためにはすべての口座を読み込む必要があります。このため、金銭の振り込みによって他のトランザクションを遅らせる可能性は少なくなります。

SQL Anywhere では、0、1、2、3 の 4 つの独立性レベルがあります。レベル 3 は完全な独立性を提供し、トランザクションはスケジュールが直列化可能となる方法でインターリーブされます。

データベースでスナップショットアイソレーションを有効にした場合は、さらに snapshot、statement-snapshot、readonly-statement-snapshot という 3 つの独立性レベルが利用可能になります。

スナップショットアイソレーションのレベルの選択

スナップショットアイソレーションには、同時実行性と一貫性の両方に利点があります。スナップショットアイソレーションを使用すると、古いバージョンのローは実行中のトランザクションで必要とされる可能性があるかぎり保存されるため、コストがかかってしまいます。そのため、スナップショットを長期間実行すると、大量の古いローバージョンを格納する必要がある可能性

があります。通常、statement-snapshot で使用されるスナップショットは、snapshot で使用されるスナップショットほど長くは存続しません。そのため、statement-snapshot の方が、snapshoto よりも一貫性は低くなりますが (トランザクション内の文ごとに、異なる時点のデータベースを認識する)、使用する領域の点で強みがあります。

ほとんどの場合は、snapshot 独立性レベルをおすすめします。これにより、トランザクション全体のデータベースに関する単一ビューが表示されるためです。

statement-snapshot 独立性レベルを使用すると、データの一貫性は低くなりますが、トランザクションを長時間実行したためにバージョン情報を格納するテンポラリファイルのサイズが大きくなりすぎる場合には有益です。

readonly-statement-snapshot 独立性レベルを使用すると、statement-snapshot よりも一貫性は低くなりますが、更新の競合が発生する可能性はなくなります。このため、この属性は元々異なる独立性レベルで実行することを想定していたアプリケーションを移植するのに最も適しています。

参照

- 「スナップショットアイソレーション」868 ページ
- 「カーソルの感知性と独立性レベル」『SQL Anywhere サーバ プログラミング』

直列化可能なスケジュール

トランザクションを同時に処理するには、データベースサーバは 1 つのトランザクションでいくつかのコンポーネント文を実行し、次に他のトランザクションのコンポーネント文を実行してから、最初のトランザクションのオペレーション処理を続行する必要があります。各種トランザクションのコンポーネントオペレーションをインターリーブする順序を、「スケジュール」と呼びます。

この方法でトランザクションを同時に適用すると、前の項で説明した一貫性が失われる 3 つのケースを含む、さまざまな結果が生じる可能性があります。トランザクションが順次実行された場合、つまり 1 つのトランザクションが完全に終了した後、次のトランザクションが開始された場合、データベースの最終状態が達成されることがあります。トランザクションをある順序で順次実行した結果、データベースが実際のスケジュールと同じ状態になった場合、そのスケジュールは「直列化可能」であるといいます。

直列化可能性は、一般に正当性の基準として受け入れられています。直列化可能なスケジュールは、データベースがトランザクションの同時実行により影響を受けないため、正しいスケジュールとして受け入れられます。

トランザクションの直列化可能性は、独立性レベルの影響を受けます。独立性レベル 3 では、すべてのスケジュールは直列化可能です。デフォルト設定値は 0 です。

直列化可能とは、同時実行性が影響を与えないということ

トランザクションが順次実行されるときでも、データベースの最終状態はこれらのトランザクションが実行される順序によって異なります。たとえば、1 つのトランザクションが特定のセルに値 5 を設定し、別のトランザクションが同じセルに 6 を設定すると、そのセルの最終値は最後に実行されたトランザクションによって決まります。

スケジュールが直列化可能であることがわかっても、トランザクションを最も効率よく実行する順序が決まるわけではありません。同時実行性の影響がないというだけです。トランザクションセットをある順序で順次実行することによって達成される結果は、すべて正しいと見なされます。

直列化不可能なスケジュールは矛盾を生じさせる

矛盾は、スケジュールが直列化可能でないときに生じる典型的な問題です。いずれのケースでも、すべてのトランザクションが順次実行された場合には起こり得ないような結果を生じさせる方法で文がインターリーブされたため、一貫性が失われました。たとえば、あるトランザクションがあるローにデータを挿入または更新しているときに、別のトランザクションがそのローを選択できる場合、ダーティリードが発生します。

参照

- 「典型的な矛盾のケース」875 ページ

各種独立性レベルでの典型的なトランザクション

各種の独立性レベルは、それぞれ適するタスクが異なります。以下の情報を参考にして、特定のオペレーションに最も合うレベルを判断してください。

典型的なレベル 0 トランザクション

データのブラウズや入力を伴うトランザクションは、終了するのに何分もかかり、相当数のローを読み込みます。独立性レベルが 2 または 3 の場合、同時実行性が犠牲になります。この種のトランザクションには、レベル 0 または 1 が使われます。

たとえば、データベースから大量のデータを読み込んで統計的にまとめる作業を行う意思決定支援アプリケーションは、後で修正されるローを多少読み込んでも大きな影響は受けません。このようなアプリケーションに上位レベルの独立性を要求すると、大量のデータに読み込みロックをかけてしまい、他のアプリケーションが書き込みアクセスできなくなります。

典型的なレベル 1 トランザクション

独立性レベル 1 は、カーソルとともに使用すると便利です。この 2 つを組み合わせると、ロック要件をそれほど増大せずにカーソルの安定性を保てます。SQL Anywhere は、カーソルの現在ローにかけられた読み込みロックを早期に解放してこれを達成します。読み込みロックは、レベル 2 や 3 では繰り返し可能読み出しを保証するためにトランザクションが終了するまで維持する必要があります。

たとえば、カーソルを使用して在庫レベルを更新するトランザクションには独立性レベル 1 が適しています。なぜなら、品目の入荷や販売があるたびに在庫レベルを調整した内容が失われることがなく、このように頻繁に調整しても、他のトランザクションへの影響が最小限で済むからです。

典型的なレベル 2 トランザクション

独立性レベル 2 では、条件を満たすローは他のトランザクションが変更することはできません。このレベルは、ローを複数回読み込み、最初の結果セットに含まれるローが変更されないことを保証する必要があるときに使用します。

比較的多数の読み込みロックが必要となるため、この独立性レベルの使用には注意を要します。レベル 3 のトランザクションと同様に、データベースとインデックスを慎重に設計することで、ロック数も減り、データベースのパフォーマンスを向上させることができます。

典型的なレベル 3 トランザクション

独立性レベル 3 はセキュリティを最も重んじるトランザクションに適しています。幻ローを防ぐと、新しいローがオペレーションの途中で追加されて結果が壊される心配をせずに、マルチステップオペレーションを実行できます。

独立性レベル 3 がいかに高度の整合性を提供するとしても、同時に多数のトランザクションの実行をサポートする必要がある大きなシステムでは使用を控える必要があります。SQL Anywhere はこのレベルでは他のレベルよりも多くのロックを設定するため、1 つのトランザクションが他の多くのトランザクションの処理を妨げる可能性があります。

独立性レベル 2 と 3 での同時実行性の改善

独立性レベル 2 と 3 は多くのロックを使用するため、これらのレベルを常用するデータベースでは、精密な設計が特に重要です。直列化可能なトランザクションを利用する必要がある場合、データベース、特にインデックスに関しては、プロジェクトのビジネスルールを念頭において設計することが重要です。大きなトランザクションを小さく分割すると、ローがロックされる時間も短縮し、パフォーマンスが向上します。

直列化可能なトランザクションは、他のトランザクションをブロックする可能性が最も大きくなりますが、必ずしも効率が低下するわけではありません。これらのトランザクションを処理する場合、SQL Anywhere では、ロック数が増加してもパフォーマンスは向上するという最適化を実行できます。たとえば、探索条件と一致するかどうかに関係なく、すべてのローには読み込みロックがかけられるので、データベースサーバはローの読み込みとロックの設定オペレーションを自由に実行することができます。

ロックの影響を削減するヒント

同時に実行される他のトランザクションに影響を及ぼす可能性のある多数のロックを設定しないで済むようにするには、トランザクションを独立性レベル 3 で実行しないことをおすすめします。

オペレーションの性質上、独立性レベル 3 で実行する必要がある場合は、読み込むローやインデックスエントリの数をできるだけ少なくするようにクエリを設計し、同時実行性への影響を減らすことができます。これによって、レベル 3 のトランザクションの処理速度が増し、さらに重要なこととして、設置するロック数を減らすことができます。

独立性レベル 3 で実行するオペレーションが 1 つでもある場合は、インデックスを追加することでトランザクションの速度が向上する場合があります。インデックスには次の 2 つの利点があります。

- インデックスの使用により、ローを効率良く見つけることができる。
- 検索にインデックスを使用するとロック数が少なくて済む。

参照

- 「ロックの仕組み」886 ページ
- 「パフォーマンスのモニタリングと診断のツール」145 ページ

独立性レベルのチュートリアル

それぞれの独立性レベルは異なる動作を実行し、データベースとそこで実行する操作に応じて、使用するレベルを決定する必要があります。以下のチュートリアルは、それぞれのタスクにどの独立性レベルを設定するかを決定するのに役立ちます。

参照

- 「独立性レベルと一貫性」864 ページ
- 「スナップショットトランザクションの知識」870 ページ
- 「典型的な矛盾のケース」875 ページ
- 「クエリ時のロック」895 ページ

チュートリアル：独立性レベルのチュートリアルのシナリオ設定

独立性レベルのすべてのチュートリアルは、Sales Manager と Accountant が同じ情報に同時にアクセスして変更するという、架空のシナリオを使用します。Sales Manager と Accountant として動作する 2 つの Interactive SQL ウィンドウを開いて、独立性レベルのチュートリアルのデータベースを設定します。

前提条件

この作業を実行するための前提条件はありません。

♦ タスク

1. Interactive SQL を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Interactive SQL]** をクリックします。
2. **[接続]** ウィンドウで、Sales Manager として SQL Anywhere サンプルデータベースに接続します。
   a. **[アクション]** ドロップダウンリストで、**[ODBC データソースを使用した接続]** をクリックします。
   b. **[ODBC データソース名]** をクリックし、その下のフィールドに **SQL Anywhere 16 Demo** と入力します。
   c. **[詳細オプション]** タブをクリックし、**[ConnectionName]** フィールドに **Sales Manager** と入力します。
   d. **[接続]** をクリックします。
3. Interactive SQL をもう 1 つ起動します。

4. **[接続]** ウィンドウで、Accountant として SQL Anywhere サンプルデータベースに接続します。
   a. **[アクション]** ドロップダウンリストで、**[ODBC データソースを使用した接続]** をクリックします。
   b. **[ODBC データソース名]** をクリックし、その下のフィールドに **SQL Anywhere 16 Demo** と入力します。
   c. **[詳細オプション]** タブをクリックし、**[ConnectionName]** フィールドに **Accountant** と入力します。
   d. **[接続]** をクリックします。

結果

Sales Manager と the Accountant としてのサンプルデータベースに接続されます。

次の手順

独立性レベルのチュートリアルの 1 つを実行します。

参照

- 「チュートリアル：ダーティリードの知識」907 ページ
- 「チュートリアル：繰り返し不可能読み出しの知識」912 ページ
- 「チュートリアル：幻ローの知識」917 ページ
- 「チュートリアル：幻ロックの知識」924 ページ

チュートリアル：ダーティリードの知識

このチュートリアルでは、複数のトランザクションを同時に実行するときに発生する可能性のある矛盾である、ダーティリードを再現してみます。このシナリオでは、小規模商社の 2 人の従業員が会社のデータベースに同時にアクセスします。1 人は会社の Sales Manager で、もう 1 人は Accountant です。

Sales Manager は、会社で販売している T シャツの価格を 0.95 ドル上げようとしていますが、SQL 言語の構文に少し問題があります。それと同時に、Sales Manager が知らないうちに、Accountant が現在の在庫の小売り価格を計算し、次の管理ミーティングに必要なレポートに記載しようとしています。

> **注意**
> このチュートリアルを正常に機能させるには、Interactive SQL の **[データベースロックの自動解放]** オプションをオフにする必要があります。このオプションの設定を確認するには、**[ツール]** » **[オプション]** をクリックし、左ウィンドウ枠の **[SQL Anywhere]** をクリックします。

権限

SELECT ANY TABLE、UPDATE ANY TABLE、SET ANY SYSTEM OPTION のシステム権限が必要です。

このチュートリアルでは、Sales Manager および Accountant としてサンプルデータベースに接続していると仮定します。「チュートリアル：独立性レベルのチュートリアルのシナリオ設定」906 ページを参照してください。

参照

- 「独立性レベルと一貫性」864 ページ
- 「スナップショットトランザクションの知識」870 ページ
- 「典型的な矛盾のケース」875 ページ
- 「クエリ時のロック」895 ページ

レッスン 1：ダーティリードの作成

Sales Manager が価格更新のプロセスを実行しているときに Accountant が計算を行う、ダーティリードを作成します。Accountant は、Sales Manager が入力後に訂正を加えている過程の間違った情報を使用して計算しました。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：ダーティリードの知識」

♦ タスク

1. Sales Manager として、次の文を実行してすべての T シャツの価格を 0.95 ドル上げます。

```
UPDATE GROUPO.Products
  SET UnitPrice = UnitPrice + 95
  WHERE Name = 'Tee Shirt';
SELECT ID, Name, UnitPrice
  FROM GROUPO.Products;
```

次の結果セットが返されます。

ID	Name	UnitPrice
300	Tee Shirt	104.00
301	Tee Shirt	109.00
302	Tee Shirt	109.00
400	Baseball Cap	9.00
...	...	...

ここで、Sales Manager 95 ではなく 0.95 を入力しなくてはならなかったことに気が付きます。しかし、間違いを訂正する前に、Accountant が別のオフィスからそのデータベースにアクセスしてきました。

2. Accountant は、在庫額が多いことを懸念しています。Accountant として次の文を実行し、全在庫品の小売り価格の合計を計算します。

```
SELECT SUM( Quantity * UnitPrice )
 AS Inventory
   FROM GROUPO.Products;
```

次の結果が返されます。

Inventory
21453.00

残念ながら、この計算は正確ではありません。Sales Manager が誤って T シャツの価格を 95 ドル上げてしまったため、合計に誤りがあります。このような誤りは、「ダーティリード」と呼ばれる典型的な矛盾の例です。Accountant であるあなたは、Sales Manager が入力したデータにアクセスしますが、このデータはまだコミットされていません。

3. Sales Manager として、最初の変更をロールバックし、正しい UPDATE 文を入力して間違いを訂正します。新しく入力した値が正しいかをチェックします。

```
ROLLBACK;
UPDATE GROUPO.Products
SET UnitPrice = UnitPrice + 0.95
WHERE NAME = 'Tee Shirt';
```

次の結果セットが返されます。

ID	Name	UnitPrice
300	Tee Shirt	9.95
301	Tee Shirt	14.95
302	Tee Shirt	14.95
400	Baseball Cap	9.00
...	...	...

4. Accountant は、計算した値に誤りがあったことに気づきません。Accountant のウィンドウでもう一度 SELECT 文を実行すると、正しい値が表示されます。

```
SELECT SUM( Quantity * UnitPrice )
 AS Inventory
   FROM GROUPO.Products;
```

Inventory
6687.15

5. Sales Manager のウィンドウでのトランザクションを終了します。Sales Manager は COMMIT 文を入力して変更を確定できますが、ここでは、ROLLBACK 文を実行して、SQL Anywhere サンプルデータベースのローカルコピーが変更されないようにします。

```
ROLLBACK;
```

結果

Accountant は、データベースサーバが Sales Manager と Accountant の作業を同時に処理しているため、知らない間に間違った情報を受け取っています。

次の手順

「レッスン 2：スナップショットアイソレーションを使用してダーティリードを避ける」910 ページに進みます。

レッスン 2：スナップショットアイソレーションを使用してダーティリードを避ける

スナップショットアイソレーションは、他のデータベースがクエリに対応してコミットされたデータだけに接続できるようにして、ダーティリードの発生を防ぎます。Accountant は、スナップショットアイソレーションを使用して、コミットされたデータがクエリに影響を与えないようにすることができます。

前提条件

このレッスンは、受講者がこれまでのすべてのレッスンを終了していることを前提としています。「レッスン 1：ダーティリードの作成」908 ページを参照してください。

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：ダーティリードの知識」

♦ タスク

1. Sales Manager として次の文を実行し、データベースのスナップショットアイソレーションを有効にします。

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'ON';
```

2. Sales Manager として、すべての T シャツの価格を 0.95 ドル上げます。
   a. 次の文を実行して価格を更新します。

```
UPDATE GROUPO.Products
SET UnitPrice = UnitPrice + 0.95
WHERE Name = 'Tee Shirt';
```

   b. Sales Manager 用の新しい T シャツ価格を使用して、全在庫品の小売り価格の合計を計算します。

```
SELECT SUM( Quantity * UnitPrice )
 AS Inventory
   FROM GROUPO.Products;
```

次の結果が返されます。

Inventory
6687.15

3. Accountant として次の文を実行し、全在庫品の小売り価格の合計を計算します。このトランザクションは snapshot 独立性レベルを使用するため、データベースにコミットされたデータのみで結果が計算されます。

```
SET OPTION isolation_level = 'Snapshot';
SELECT SUM( Quantity * UnitPrice )
 AS Inventory
   FROM GROUPO.Products;
```

次の結果が返されます。

Inventory
6538.00

4. Sales Manager として次の文を実行し、データベースに対する変更をコミットします。

```
COMMIT;
```

5. Accountant として次の文を実行し、現在の在庫の更新後の小売り価格を表示します。

```
COMMIT;
SELECT SUM( Quantity * UnitPrice )
 AS Inventory
   FROM GROUPO.Products;
```

次の結果が返されます。

Inventory
6687.15

Accountant のトランザクションで使用されるスナップショットは最初の読み込み操作で開始するため、COMMIT を実行してトランザクションを終了し、スナップショットトランザクションの開始後に加えられたデータの変更を Accountant が確認できるようにする必要があります。

6. Sales Manager として次の文を実行し、T シャツの価格の変更を取り消し、SQL Anywhere サンプルデータベースを元の状態に復元します。

```
UPDATE GROUPO.Products
SET UnitPrice = UnitPrice - 0.95
WHERE Name = 'Tee Shirt';
COMMIT;
```

結果

Accountant は、スナップショットアイソレーションを有効にして、ダーティリードを防ぐことができました。

次の手順

(省略可) サンプルデータベース (*demo.db*) を元の状態にリストアします。「サンプルデータベースの再作成 (demo.db)」『SQL Anywhere 16 紹介』を参照してください。

チュートリアル：繰り返し不可能読み出しの知識

このチュートリアルでは、複数のトランザクションを同時に実行するときに発生する可能性のある矛盾である、繰り返し不可能読み出しを再現してみます。このシナリオでは、小規模商社の 2 人の従業員が会社のデータベースに同時にアクセスします。1 人は会社の Sales Manager で、もう 1 人は Accountant です。

Sales Manager はプラスチックバイザーに新しい価格を設定すると仮定します。Accountant は最近注文があったいくつかの品目の価格を調べるとします。

この例では、独立性レベル 0 ではなく 1 を使って、両方の接続を開始します。独立性レベル 0 は SQL Anywhere サンプルデータベースのデフォルト値です。独立性レベルを 1 に設定することで、ダーティリードの可能性を減らします。

> **注意**
> このチュートリアルを正常に機能させるには、Interactive SQL の **[データベースロックの自動解放]** オプションをオフにする必要があります。このオプションの設定を確認するには、**[ツール]** » **[オプション]** をクリックし、左ウィンドウ枠の **[SQL Anywhere]** をクリックします。

権限

SELECT ANY TABLE、UPDATE ANY TABLE、SET ANY SYSTEM OPTION のシステム権限が必要です。

このチュートリアルでは、Sales Manager および Accountant としてサンプルデータベースに接続していると仮定します。「チュートリアル：独立性レベルのチュートリアルのシナリオ設定」906 ページを参照してください。

参照

- 「独立性レベルと一貫性」864 ページ
- 「スナップショットトランザクションの知識」870 ページ
- 「典型的な矛盾のケース」875 ページ
- 「クエリ時のロック」895 ページ

レッスン 1：繰り返し不可能読み出しの作成

Sales Manager によって変更され、同じトランザクション中に 2 つの異なる結果を取得するローを Accountant が読み出す、繰り返し不可能読み出しを作成します。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：繰り返し不可能読み出しの知識」

♦ タスク

1. 次の文を実行して、Accountant の接続の独立性レベルを 1 に設定します。

```
SET TEMPORARY OPTION isolation_level = 1;
```

2. 次の文を実行して、Sales Manager のウィンドウに独立性レベル 1 を設定します。

```
SET TEMPORARY OPTION isolation_level = 1;
```

3. Accountant として、次の文を実行し、サンバイザーの価格のリストを表示します。

```
SELECT ID, Name, UnitPrice
FROM GROUPO.Products;
```

ID	Name	UnitPrice
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
401	Baseball Cap	10.00
500	Visor	7.00
501	Visor	7.00
...	...	...

4. Sales Manager として、次の文を実行し、プラスチック製サンバイザーの新しい販売価格を導入します。

```
SELECT ID, Name, UnitPrice FROM GROUPO.Products
WHERE Name = 'Visor';
UPDATE GROUPO.Products
SET UnitPrice = 5.95 WHERE ID = 501;
COMMIT;
SELECT ID, Name, UnitPrice FROM GROUPO.Products
WHERE Name = 'Visor';
```

ID	Name	UnitPrice
500	Visor	7.00
501	Visor	5.95

5. Sales Manager ウィンドウでのバイザーの価格と Accountant ウィンドウでの価格を比較してみてください。Accountant として SELECT 文をもう一度実行すると、Sales Manager の新価格が表示されます。

```
SELECT ID, Name, UnitPrice
FROM GROUPO.Products;
```

ID	Name	UnitPrice
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
401	Baseball Cap	10.00
500	Visor	7.00
501	Visor	5.95
...	...	...

この矛盾を「「繰り返し不可能読み出し」」と呼んでいます。Accountant が 2 回目も**同じトランザクション**で同じ SELECT 文を実行した後で、同じ結果が得られないためです。

もちろん、Accountant が SELECT を再度使用する前に、COMMIT や ROLLBACK 文などを発行してトランザクションを終了した場合には、状況は違ってきます。データベースは複数のユーザが同時に使用でき、Accountant のトランザクションの前後に他の人が値を変更できます。他の人が行った変更によって矛盾が生じるのは、Accountant のトランザクションの途中で変更が行われるためです。そうしたイベントにより、スケジュールは直列化不可能となります。

6. Accountant はこれに気づき、以降は価格参照中にほかからの変更を防ぐことにします。繰り返し不可読み出しは、独立性レベル 2 で削除されます。Accountant として、次の文を実行します。

```
SET TEMPORARY OPTION isolation_level = 2;
SELECT ID, Name, UnitPrice
FROM GROUPO.Products;
```

7. Sales Manager は、明日受注するはずの大口注文に値下げを適用しなくて済むように、プラスチックバイザーの安売りを来週に延期することを決定します。Sales Manager として、次の文を実行します。文の実行が始まりますが、ウィンドウは実行中のまま終了しません。

   ```
   UPDATE GROUPO.Products
   SET UnitPrice = 7.00
   WHERE ID = 501;
   ```

   データベースサーバは独立性レベル 2 では繰り返し可能読み出しを保証する必要があります。Accountant は独立性レベル 2 を使用するため、データベースサーバは Accountant が読み込む Products テーブルの各ローに読み込みロックをかけます。Sales Manager が価格を元に戻そうとすると、そのトランザクションは、Products テーブルのプラスチックバイザーのローに書き込みロックをかける必要があります。書き込みロックは排他ロックであるため、Accountant のトランザクションが読み込みロックを解放するまで待機しなければなりません。

8. Accountant が価格の閲覧を終了しました。データベースを誤って変更するのを防ぐために、ROLLBACK 文でトランザクションを完了します。

   ```
   ROLLBACK;
   ```

   データベースサーバがこの文を実行すると、Sales Manager のトランザクションが完了します。

ID	Name	UnitPrice
500	Visor	7.00
501	Visor	7.00

9. Sales Manager もトランザクションを終了できるようになりました。Sales Manager は、変更をコミットして、元の価格をリストアします。

   ```
   COMMIT;
   ```

結果

Accountant は、同じトランザクション中に異なる結果を受け取ります。したがって、スナップショット独立性レベル 2 を有効にして、繰り返し不可能読み出しを防ぎます。Accountant がデータベースを変更すると、Sales Manager はデータベースを変更できなくなります。

Accountant の独立性レベルを 1 から 2 に更新したときに、データベースサーバは、前に誰もロックをかけたことのない場所で読み込みロックを使用しました。それ以降、選択内容に適合するローごとに読み込みロックをかけました。

前述のチュートリアルでは、UPDATE 文の実行中に Sales Manager のウィンドウがフリーズしました。データベースサーバは UPDATE 文の実行を開始し、Sales Manager が変更を必要としているローに Accountant のトランザクションが読み込みロックをかけていることを発見しました。この時点で、データベースサーバは UPDATE の実行を一時停止します。Accountant が ROLLBACK でトランザクションを終了すると、データベースサーバは自動的にロックを解放します。妨げがなくなると、データベースサーバが Sales Manager の UPDATE を最後まで処理します。

次の手順

「レッスン 2 : スナップショットアイソレーションを使用して繰り返し不可能読み出しを避ける」916 ページに進みます。

参照

- 「レッスン 2 : スナップショットアイソレーションを使用して繰り返し不可能読み出しを避ける」916 ページ
- 「ブロックオプション」882 ページ

レッスン 2 : スナップショットアイソレーションを使用して繰り返し不可能読み出しを避ける

スナップショットアイソレーションを使用してブロックを回避することもできます。スナップショットアイソレーションを使用するトランザクションはコミットされたデータだけを認識するため、Accountant のトランザクションは、Sales Manager のトランザクションをブロックしません。

前提条件

このレッスンは、受講者がこれまでのすべてのレッスンを終了していることを前提としています。「レッスン 1 : 繰り返し不可能読み出しの作成」912 ページを参照してください。

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル : 繰り返し不可能読み出しの知識」

♦ タスク

1. Accountant として次の文を実行して、データベースのスナップショットアイソレーションを有効にし、snapshot 独立性レベルを使用することを指定します。

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';
SET TEMPORARY OPTION isolation_level = 'snapshot';
```

2. Accountant として、次の文を実行し、サンバイザーの価格のリストを表示します。

```
SELECT ID, Name, UnitPrice
FROM GROUPO.Products
ORDER BY ID;
```

ID	Name	UnitPrice
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00

ID	Name	UnitPrice
400	Baseball Cap	9.00
401	Baseball Cap	10.00
500	Visor	7.00
501	Visor	7.00
...	...	...

3. Sales Manager として、次の文を実行し、プラスチック製サンバイザーの新しい販売価格を導入します。

```
UPDATE GROUPO.Products
SET UnitPrice = 5.95 WHERE ID = 501;
COMMIT;
SELECT ID, Name, UnitPrice FROM GROUPO.Products
WHERE Name = 'Visor';
```

4. Accountant はクエリをもう一度実行しますが、最初の読み込み時にコミットされたデータがトランザクションで使用されるため、価格の変更を認識しません。

```
SELECT ID, Name, UnitPrice
FROM GROUPO.Products;
```

5. Sales Manager として、プラスチックバイザーを元の価格に戻します。

```
UPDATE GROUPO.Products
SET UnitPrice = 7.00
WHERE ID = 501;
COMMIT;
```

データベースサーバは、Accountant が読み込み中の Products テーブルのローに読み込みロックをかけません。これは Sales Manager が Products テーブルに変更を加える前に作成された、コミットされたデータのスナップショットを Accountant が閲覧しているためです。

6. Accountant が価格の閲覧を終了しました。データベースを誤って変更するのを防ぐために、ROLLBACK 文でトランザクションを完了します。

```
ROLLBACK;
```

結果

スナップショットアイソレーションを使用して、繰り返し不可能読み出しを防ぐことができました。

チュートリアル：幻ローの知識

このチュートリアルでは、複数のトランザクションを同時に実行するときに発生する可能性のある矛盾である、幻ローを再現してみます。このシナリオでは、小規模商社の 2 人の従業員が会社

のデータベースに同時にアクセスします。1 人は会社の Sales Manager で、もう 1 人は Accountant です。

Sales Manager は、新しく Foreign Sales と Major Account Sales の部署を作りたいと考えています。Accountant は、会社に存在するすべての部署を確認したいと考えています。

この例では、独立性レベル 0 ではなく 2 を使って、両方の接続を開始します。独立性レベル 0 は SQL Anywhere サンプルデータベースのデフォルト値です。独立性レベルを 2 に設定することで、ダーティリードと繰り返し不可能読み出しの可能性を減らします。

注意
このチュートリアルを正常に機能させるには、Interactive SQL の **[データベースロックの自動解放]** オプションをオフにする必要があります。このオプションの設定を確認するには、**[ツール]** » **[オプション]** をクリックし、左ウィンドウ枠の **[SQL Anywhere]** をクリックします。

権限

SELECT ANY TABLE、INSERT ANY TABLE、DELETE ANY TABLE、SET ANY SYSTEM OPTION のシステム権限が必要です。

このチュートリアルでは、Sales Manager および Accountant としてサンプルデータベースに接続していると仮定します。「チュートリアル：独立性レベルのチュートリアルのシナリオ設定」906 ページを参照してください。

参照

- 「独立性レベルと一貫性」864 ページ
- 「スナップショットトランザクションの知識」870 ページ
- 「典型的な矛盾のケース」875 ページ
- 「クエリ時のロック」895 ページ

レッスン 1：幻ローの作成

Sales Manager がローを挿入し、Accountant が隣接するローを読み出すことで、幻ローを作成します。このアクションにより、新しいローが幻として表示されます。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：幻ローの知識」

♦ タスク

1. 次の文を実行して、Sales Manager と Accountant のウィンドウにそれぞれ独立性レベル 2 を設定します。

   ```
   SET TEMPORARY OPTION isolation_level = 2;
   ```

2. Accountant として次の文を実行し、すべての部署のリストを表示します。

```
SELECT * FROM GROUPO.Departments
ORDER BY DepartmentID;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703

3. Sales Manager は、主に外国市場を担当する新しい部署の設置を決めました。EmployeeID 129 の Philip Chin を新しい部署の責任者にします。Sales Manager として、次の文を実行し、新しい部署用のエントリを作成します。この新しいエントリは、Sales Manager のウィンドウのテーブル下部に新しいローとして表示されます。

```
INSERT INTO GROUPO.Departments
  ( DepartmentID, DepartmentName, DepartmentHeadID )
  VALUES( 600, 'Foreign Sales', 129 );
  COMMIT;
```

4. Sales Manager として次の文を実行し、すべての部署のリストを表示します。

```
SELECT * FROM GROUPO.Departments
ORDER BY DepartmentID;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

5. しかし Accountant は、新しい部署が作成されたことに気づきません。独立性レベル 2 で、データベースサーバはローを変更しないようにロックをかけますが、他のトランザクションが新しいローを挿入するのを防止するロックはかけていません。

   Accountant は SELECT 文を再実行した場合にだけ、新しいローを発見できます。Accountant として、SELECT 文を実行して、テーブルに追加された新しいローをもう一度表示します。

```
SELECT * FROM GROUPO.Departments
ORDER BY DepartmentID;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

新しく追加されたローは、「幻ロー」と呼ばれます。これは、Accountant の観点から見ると、このローがどこからともなく出現した幻のように映るためです。Accountant は独立性レベル 2 で接続されます。このレベルでは、サーバは使用中のローにだけロックをかけます。他のローにはロックがかけられないため、Sales Manager が新しいローを挿入するのを妨げるものはありません。

6. Accountant は今後そうしたことが起こらないように、現在のトランザクションの独立性レベルを 3 に上げることにします。次の文を Accountant として実行します。

```
SET TEMPORARY OPTION isolation_level = 3;
SELECT * FROM GROUPO.Departments
ORDER BY DepartmentID;
```

7. Sales Manager は、大企業に対して営業活動を行う新しい部署を追加したいと考えています。Sales Manager として、次の文を実行します。

```
INSERT INTO GROUPO.Departments
 ( DepartmentID, DepartmentName, DepartmentHeadID )
  VALUES( 700, 'Major Account Sales', 902 );
```

Accountant のロックが文をブロックするため、Sales Manager のウィンドウは実行中に一時停止します。ツールバーで、**[停止]** をクリックして、このエントリを中止します。

Accountant が独立性レベルを 3 に上げ、Departments テーブルのすべてのローを再度選択した場合、データベースサーバはテーブルの各ローに対挿入ロックを設定し、新しいローの挿入を防止するためにテーブルの最後にもう 1 つ幻ロックを設定します。Sales Manager がテーブルの最後に新しいローを挿入しようとすると、その文はこの最後のロックによってブロックされます。

Sales Manager は独立性レベル 2 で接続されているにもかかわらず、Sales Manager の文はブロックされました。データベースサーバは独立性レベルと各トランザクション文の要求に応じ、読み込みロックと同様に幻ロックを設定します。一度対挿入ロックが設定されると、このロックは同時に実行される他のすべてのトランザクションに適用されます。

8. SQL Anywhere サンプルデータベースが変更されないようにするため、Major Account Sales 部署のローを挿入する未完了トランザクションをロールバックし、2 つ目のトランザクションを使用して Foreign Sales 部署を削除してください。

a. Accountant として、次の文を実行して独立性レベルを引き下げ、Sales Manager がデータベースに対する変更を取り消すことができるようにします。

```
SET TEMPORARY OPTION isolation_level=3;
```

b. Sales Manager として、次の文を実行して現在のトランザクションをロールバックし、以前挿入されたローを削除し、この操作をコミットします。

```
ROLLBACK;
DELETE FROM GROUPO.Departments
WHERE DepartmentID = 600;
COMMIT;
```

結果

Accountant は、SELECT 文が実行されるたびに異なる結果を受け取ります。したがって、スナップショット独立性レベル 3 を有効にして、幻ローを防ぎます。Accountant がデータベースを変更すると、Sales Manager はデータベースを変更できなくなります。

次の手順

「レッスン 2：スナップショットアイソレーションを使用して幻ローを防ぐ」921 ページに進みます。

参照

- 「レッスン 2：スナップショットアイソレーションを使用して幻ローを防ぐ」921 ページ
- 「ロックの仕組み」886 ページ

レッスン 2：スナップショットアイソレーションを使用して幻ローを防ぐ

snapshot 独立性レベルを使用すると、独立性レベル 3 と同じレベルで一貫性を維持することができ、ブロックが発生しません。Sales Manager の文はブロックされず、Accountant は幻ローを認識しません。

前提条件

このレッスンは、受講者がこれまでのすべてのレッスンを終了していることを前提としています。「レッスン 1：幻ローの作成」918 ページを参照してください。

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：幻ローの知識」

♦ タスク

1. Accountant として、次の文を実行してスナップショットアイソレーションを有効にします。

```
SET OPTION PUBLIC. allow_snapshot_isolation = 'On';
SET TEMPORARY OPTION isolation_level = 'snapshot';
```

2. 次の文を実行して、すべての部署をリストします。

```
SELECT * FROM GROUPO.Departments
ORDER BY DepartmentID;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703

3. Sales Manager は、主に外国市場を担当する新しい部署の設置を決めました。EmployeeID 129 の Philip Chin を新しい部署の責任者にします。Sales Manager として、次の文を実行し、新しい部署用のエントリを作成します。この新しいエントリは、Sales Manager のウィンドウのテーブル下部に新しいローとして表示されます。

```
INSERT INTO GROUPO.Departments
  ( DepartmentID, DepartmentName, DepartmentHeadID )
  VALUES( 600, 'Foreign Sales', 129 );
COMMIT;
```

4. Sales Manager として次の文を実行し、すべての部署のリストを表示します。

```
SELECT * FROM GROUPO.Departments
ORDER BY DepartmentID;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

5. Accountant はクエリをもう一度実行できます。また、トランザクションがコミットされていないため、新しいローを認識しません。

```
SELECT * FROM GROUPO.Departments
ORDER BY DepartmentID;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703

6. Sales Manager は、大企業に対して営業活動を行う新しい部署を追加したいと考えています。Sales Manager として、次の文を実行します。

```
INSERT INTO GROUPO.Departments
 ( DepartmentID, DepartmentName, DepartmentHeadID )
   VALUES( 700, 'Major Account Sales', 902 );
```

Accountant がスナップショットアイソレーションを使用しているため、Sales Manager による変更はブロックされません。

7. Sales Manager がデータベースにコミットした変更を認識するために、Accountant はスナップショットアイソレーションを終了する必要があります。

```
COMMIT;
  SELECT * FROM GROUPO.Departments
  ORDER BY DepartmentID;
```

Accountant は Foreign Sales 部署を認識するようになりました。ただし、Major Account Sales 部署は認識しません。

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

8. SQL Anywhere サンプルデータベースが変更されないようにするため、Major Account Sales 部署のローを挿入する未完了トランザクションをロールバックし、2 つ目のトランザクションを使用して Foreign Sales 部署を削除してください。

   a. Sales Manager として、次の文を実行して現在のトランザクションをロールバックし、以前挿入されたローを削除し、この操作をコミットします。

```
ROLLBACK;
DELETE FROM GROUPO.Departments
WHERE DepartmentID = 600;
COMMIT;
```

結果

スナップショットアイソレーションを使用して、幻ローを防ぐことができました。

チュートリアル：幻ロックの知識

このチュートリアルでは、Accountant と Sales Manager はどちらも SalesOrder テーブルと SalesOrderItems テーブルに関わるタスクを行います。Accountant は、販売担当者に支払ったコミッションの小切手額を確認する必要があります。Sales Manager は、失われた注文がいくつかあることに気づき、それを追加したいと思っています。

前提条件

SELECT ANY TABLE、INSERT ANY TABLE、DELETE ANY SYSTEM のシステム権限が必要です。

このチュートリアルでは、Sales Manager および Accountant としてサンプルデータベースに接続していると仮定します。「チュートリアル：独立性レベルのチュートリアルのシナリオ設定」906 ページを参照してください。

> **注意**
> このチュートリアルを正常に機能させるには、Interactive SQL の **[データベースロックの自動解放]** オプションをオフにする必要があります。このオプションの設定を確認するには、**[ツール]** » **[オプション]** をクリックし、左ウィンドウ枠の **[SQL Anywhere]** をクリックします。

内容と備考

このチュートリアルは、幻ロックを示します。「幻ロック」は幻ローを防ぐためにインデックススキャン位置に設定される共有ロックです。独立性レベル 3 のトランザクションが指定された基準を満たすローを選択すると、データベースサーバは対挿入ロックを設定し、他のトランザクションが基準を満たすローを挿入することを禁止します。設置するロック数は検索基準やデータベースの設計によって異なります。

♦ タスク

1. 次の文を実行して、Sales Manager と Accountant のウィンドウにそれぞれ独立性レベル 2 を設定します。

   ```
   SET TEMPORARY OPTION isolation_level = 2;
   ```

2. 毎月、販売担当者には、その月の各人の売り上げ高に対し、一定の割合のコミッションが支払われます。Accountant は、2001 年 4 月分のコミッションの小切手を準備しています。彼の最初のタスクは、その月の各担当者の売り上げ合計額を計算することです。価格、注文情報、従業員データがそれぞれ別のテーブルに保存されます。外部キー関係を使用してこれらのテーブルをジョインすることにより、必要な情報を結合します。

Accountant として、次の文を実行します。

```
SELECT EmployeeID, GivenName, Surname,
  SUM( SalesOrderItems.Quantity * UnitPrice )
    AS "April sales"
FROM GROUPO.Employees
  KEY JOIN GROUPO.SalesOrders
  KEY JOIN GROUPO.SalesOrderItems
  KEY JOIN GROUPO.Products
WHERE '2001-04-01' <= OrderDate
  AND OrderDate < '2001-05-01'
GROUP BY  EmployeeID, GivenName, Surname
ORDER BY EmployeeID;
```

EmployeeID	GivenName	Surname	April sales
129	Philip	Chin	2160.00
195	Marc	Dill	2568.00
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
...	...	...	...

3. Sales Manager は、Philip Chin の多額の売り上げがデータベースに入力されていないことに気づきました。Philip はコミッションの迅速な支払いを希望しているため、Sales Manager は 4 月 25 日の彼の漏れていた注文を入力します。

   Sales Manager として、次の文を実行します。1 つの注文に多くの品目を含めることができるため、受注と品目は別のテーブルに入力されます。品目を追加する前に、売り上げ注文にエントリを作成します。参照整合性を維持するために、データベースサーバは注文がすでに存在する場合にかぎり、トランザクションが品目を注文に追加することを許可します。

```
INSERT into GROUPO.SalesOrders
VALUES ( 2653, 174, '2001-04-22', 'r1',
     'Central', 129 );
INSERT into GROUPO.SalesOrderItems
VALUES ( 2653, 1, 601, 100, '2001-04-25' );
COMMIT;
```

4. Accountant は、Sales Manager が新しい注文を追加したことを知りません。新しい注文がもっと前に入力された場合は、Philip Chin の 4 月の売り上げ計算に含まれます。

   Accountant のウィンドウで、4 月の売り上げ合計を再計算します。同じ文を使用しますが、Philip Chin の 4 月の売り上げ額が $4560.00 ドルに変更されていることに注意してください。

EmployeeID	GivenName	Surname	April sales
129	Philip	Chin	4560.00
195	Marc	Dill	2568.00

EmployeeID	GivenName	Surname	April sales
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
...	...	...	...

ここで、Accountant は 4 月の注文すべてにマークを付けて、コミッションが支払い済みであることを示します。Sales Manager が入力したばかりの注文が 2 度目の検索で見つかりましたが、たとえ Philip の 4 月の売り上げ合計に含まれていなかったとしても、支払い済みのマークが付いています。

5. 独立性レベル 3 では、データベースサーバは対挿入ロックを設定し、検索または選択条件に合うローを他のトランザクションが追加できないようにします。

   Sales Manager として次の文を実行し、新しい注文を削除します。

```
DELETE
FROM GROUPO.SalesOrderItems
WHERE ID = 2653;
DELETE
FROM GROUPO.SalesOrders
WHERE ID = 2653;
COMMIT;
```

6. Accountant として、次の文を実行します。

```
ROLLBACK;
SET TEMPORARY OPTION isolation_level = 3;
```

7. 次のクエリを実行します。

```
SELECT EmployeeID, GivenName, Surname,
  SUM( SalesOrderItems.Quantity * UnitPrice )
    AS "April sales"
FROM GROUPO.Employees
  KEY JOIN GROUPO.SalesOrders
  KEY JOIN GROUPO.SalesOrderItems
  KEY JOIN GROUPO.Products
WHERE '2001-04-01' <= OrderDate
  AND OrderDate < '2001-05-01'
GROUP BY  EmployeeID, GivenName, Surname;
```

   独立性レベル 3 を設定したため、データベースサーバは自動的に対挿入ロックを設定し、Accountant がトランザクションを終了するまで、Sales Manager が 4 月の注文品目を挿入できないようにします。

8. Sales Manager として次の文を実行して、Philip Chin の漏れていた注文の入力を試みます。

```
INSERT INTO GROUPO.SalesOrders
VALUES ( 2653, 174, '2001-04-22',
       'r1','Central', 129 );
```

   Sales Manager のウィンドウは応答を停止し、操作は完了しません。ツールバーで、**[停止]** をクリックして、このエントリを中止します。

9. Sales Manager は 4 月の注文を入力できませんが、5 月分には入力できると考えます。

   文の日付を 5 月 5 日に変更し、再実行します。

   ```
   INSERT INTO GROUPO.SalesOrders
   VALUES ( 2653, 174, '2001-05-05', 'r1',
       'Central', 129 );
   ```

   Sales Manager のウィンドウは再度応答を停止します。ツールバーで、**[停止]** をクリックして、このエントリを中止します。データベースサーバでは挿入を防止するために必要となるロック以外は設定しませんが、これらのロックによって、多数のトランザクションの処理が妨げられる可能性があります。

   データベースサーバはテーブルインデックスにロックをかけます。たとえば、インデックスに幻ロックを設定し、インデックスの直前に新しいローを追加できないようにします。ただし、適切なインデックスが存在しない場合、テーブルのすべてのローにロックをかける必要があります。ある状況では、対挿入ロックによって特定テーブルへの挿入のみをブロックできます。

10. SQL Anywhere サンプルデータベースを防ぐために、SalesOrders テーブルに対する変更をロールバックする必要があります。[Sales Manager] ウィンドウと [Accountant] ウィンドウで、次の文を実行します。

    ```
    ROLLBACK;
    ```

11. Interactive SQL の両方のインスタンスを終了します。

結果

幻ロックの動作方法を理解するチュートリアルを完了しました。

参照

- 「独立性レベルと一貫性」864 ページ
- 「スナップショットトランザクションの知識」870 ページ
- 「典型的な矛盾のケース」875 ページ
- 「クエリ時のロック」895 ページ

プライマリキーの生成と同時実行性

状況によっては、データベースに自動的にユニークな番号を生成させたいという場合があります。たとえば、商品の送り状を格納するテーブルを作成する場合、販売スタッフではなくデータベースが自動的にユニークな送り状番号を割り当てることができます。

例

たとえば、商品の送り状番号は、1 つ前の送り状番号に 1 を加えて作成できます。しかし、複数の人間がデータベースに送り状番号を入力するときは、この方法は使えません。2 人の従業員が同じ番号を選択する可能性があるからです。

この問題を解決するには、次に示すように方法がいくつかあります。

- 送り状番号を入力するユーザごとに数字の範囲を設定する。

  このスキームは、カラム user name と invoice number を持つテーブルを作成することで実装します。ローの 1 つは、送り状番号を入力するユーザを記録するのに使います。ユーザが送り状を追加するごとに、テーブル内の数字は 1 増えて新しい送り状に使われます。データベースのすべてのテーブルを処理するには、テーブルに 3 つのカラム (テーブル名、ユーザ名、最後のキー値) が必要です。各ユーザに十分な数字が確保されているかどうかを定期的に確認する必要があります。

- 2 つのカラム table name と last key を持つテーブルを作成する。

  このテーブルには、最後に使用した送り状番号が格納されるローが 1 つあります。送り状番号は、ユーザが送り状の追加、新しい接続の確立、送り状番号の増分、または変更の即時コミットを行うたびに、自動的に増分されます。ローは別のトランザクションによって直ちに更新されるため、他のユーザは新しい送り状番号にアクセスできます。

- NEWID のデフォルト値のあるカラムを UNIQUEIDENTIFIER バイナリデータ型と組み合わせて使用して、完全にユニークな識別子を生成する。

  UUID 値と GUID 値を使用すると、テーブルローをユニークに識別できます。あるコンピュータで生成された値は別のコンピュータで生成された値と一致しないため、レプリケーションと同期の環境では、UUID 値と GUID 値をキーとして使用できます。

- AUTOINCREMENT のデフォルト値のカラムを使用する。次に例を示します。

```
CREATE TABLE Orders (
  OrderID INTEGER NOT NULL DEFAULT AUTOINCREMENT,
  OrderDate DATE,
  primary key( OrderID )
);
```

  テーブルに挿入するときに、AUTOINCREMENT カラムに対して値を指定しないと、ユニークな値が生成されます。値を指定すると、その指定した値が使われます。値がカラムの現在の最大値より大きい場合は、その後の挿入開始ポイントとしてこの値が使われます。AUTOINCREMENT カラムに最後に追加されたローの値は、グローバル変数 @@identity で取得できます。

参照

- 「NEWID デフォルト」841 ページ

ユニークな値を生成するためのシーケンスの使用

「シーケンス」を使用すると、複数のテーブル間でユニークな値、つまり一連の自然数と異なる値を生成できます。シーケンスは、CREATE SEQUENCE 文を使用して作成します。シーケンス値は BIGINT 値として返されます。

各接続で、最後に使用された「次の値」が現在の値として保存されます。

シーケンスを作成すると、データベースサーバがメモリに保持しているシーケンス値の数が定義に含められます。このキャッシュが不足すると、シーケンスキャッシュが再移植されます。データベースサーバで障害が発生した場合、キャッシュに保持されていたシーケンス値がスキップされることがあります。

シーケンス値の取得

シーケンスの現在の値を取得するには、次の文を使用します。

```
SELECT sequence-name.CURRVAL;
```

シーケンスの次の値を取得するには、次の文を使用します。

```
SELECT sequence-name.NEXTVAL;
```

シーケンス値と AUTOINCREMENT 値からの選択

AUTOINCREMENT の動作	シーケンスの動作
テーブルの単一のカラムに定義される	データベースオブジェクトとして格納され、式が許される場所であればどこでも使用できる
カラムのデータ型は整数データ型または真数値データ型	値は式が使用できる場所であればどこからでも参照でき、カラムのデフォルト値に適合しなくてもよい
値は 1 つのテーブルの単一のカラムでのみ使用できる	値は複数のテーブルで使用できる
値は一連の自然数 (1、2、3、...) の一部である	一連の自然数ではない値を生成できる
値は増分される	値は増分または減分できる
カラムの以前の最大値より 1 大きいユニークな値がデフォルトで生成される sa_reset_identity システムプロシージャを使用して、次に挿入するローの AUTOINCREMENT 値を変更できる	増分の単位を指定できる
生成される次の値がカラムに格納できる最大値を超えている場合、NULL が返される	最大値または最小値に達した後に、生成された値を許可するかどうか選択できる。または NO CYCLE を指定してエラーを返す

シーケンスの例

顧客ホットラインのインシデント番号を生成するために使用されるシーケンスの場合を考えてみます。顧客からの苦情電話には 2 種類のタイプ (請求が間違っている、または荷物が行方不明) があると想定します。

```
CREATE SEQUENCE incidentSequence
  MINVALUE 1000
  MAXVALUE 100000;

CREATE TABLE reportedBillingMistake(
  incidentID INT PRIMARY KEY DEFAULT (incidentSequence.nextval),
  billNumber INT,
  valueOnBill NUMERIC(10,2),
```

```
    expectedValue NUMERIC(10,2),
    comments LONG VARCHAR );

  CREATE TABLE reportedMissingShipment(
    incidentID INT PRIMARY KEY DEFAULT(incidentSequence.nextval),
    orderNumber INT,
    comments LONG VARCHAR );
```

incidentSequence.nextval を incidentID カラムに使用することによって、incidentID が 2 つのテーブルでユニークであることが保障されます。顧客から再度問い合わせの電話があり、インシデント値が提供されたときに、請求の問題か荷物の出荷の問題かわからなくなることはありません。

請求の間違いを挿入する場合、次の文は同義です。

```
INSERT INTO reportedBillingMistake VALUES( DEFAULT, 12345, 100.00, 75.00, 'Bad bill' );

INSERT INTO reportedBillingMistake
  SELECT incidentSequence.nextval, 12345, 100.00, 75.00, 'Bad bill';
```

直前に挿入された incidentID を見つけるには、挿入を実行した (上の 2 つの文のいずれかを使用) 接続で次の文を実行できます。

```
SELECT incidentSequence.currval;
```

参照

- 「CREATE SEQUENCE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「オートインクリメントデフォルト」839 ページ
- 「GLOBAL AUTOINCREMENT デフォルト」840 ページ
- sequence-expression 句、SELECT 文『SQL Anywhere サーバ SQL リファレンス』

シーケンスの作成

Sybase Central を使用して、シーケンスを作成できます。

前提条件

CREATE ANY SEQUENCE または CREATE ANY OBJECT のシステム権限が必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で **[シーケンスジェネレータ]** を右クリックし、**[新規]** » **[シーケンスジェネレータ]** をクリックします。
3. **[シーケンスジェネレータの作成ウィザード]** の指示に従います。

結果

シーケンスを作成しています。

参照

- 「CREATE SEQUENCE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「GRANT USAGE ON SEQUENCE 文」『SQL Anywhere サーバ SQL リファレンス』
- 「ユニークな値を生成するためのシーケンスの使用」928 ページ
- 「シーケンス値と AUTOINCREMENT 値からの選択」929 ページ

シーケンスの変更

Sybase Central を使用して、シーケンスを変更できます。

前提条件

そのシーケンスの所有者であるか、または次のいずれかの権限を持っていることが必要です。

- ALTER ANY SEQUENCE システム権限
- ALTER ANY OBJECT システム権限

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. シーケンスジェネレータを右クリックして、**[プロパティ]** をクリックします。

   [一般] タブで、シーケンスの設定を変更できます。**[すぐに再起動]** をクリックすると、ALTER SEQUENCE...RESTART WITH *n* 文が実行されます。ここで、*n* は **[開始値]** フィールドの値に対応しています。

結果

変更は直ちに有効になります。

参照

- 「ALTER SEQUENCE 文」『SQL Anywhere サーバ SQL リファレンス』

シーケンスの削除

Sybase Central を使用して、シーケンスを削除できます。

前提条件

そのシーケンスの所有者であるか、または次のいずれかの権限を持っていることが必要です。

- DROP ANY SEQUENCE システム権限
- DROP ANY OBJECT システム権限

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。

2. シーケンスジェネレータを右クリックして、**[削除]** をクリックします。

結果

シーケンスはデータベースから削除されます。シーケンスを削除すると、そのシーケンス名のすべての同義語がデータベースサーバによって自動的に削除されます。

参照

- 「DROP SEQUENCE 文」『SQL Anywhere サーバ SQL リファレンス』

データ定義文と同時実行性

CREATE INDEX、ALTER TABLE、TRUNCATE TABLE のように、テーブル全体を変更するようなデータ定義文は、その文が動作しているテーブルがその時点で他の接続に使われている場合は処理されません。これらのデータ定義文は時間がかかり、データベースサーバは文の実行中は対象となるテーブルの参照要求を処理しません。

CREATE TABLE 文は同時実行性の問題は起こしません。

GRANT 文、REVOKE 文、SET OPTION 文も同時実行性の問題を起こしません。これらの文は、データベースサーバに送られる新しい SQL 文には影響しますが、未処理の文には影響しません。

データベースに接続しているユーザに対しては、GRANT と REVOKE は許可されません。

データ定義文と同期されたデータベース
同期を使用するデータベースでデータ定義文を使用する場合は特に注意が必要です。Mobile Link サーバ管理を参照してください。

参照

- 「データ定義文」『SQL Remote』

まとめ

トランザクションとロックは、テーブル間の関係にとっては 2 番目に重要な要素にすぎません。データベースの整合性とパフォーマンスは、ロックを効率的に使用したりトランザクションを注意深く構成することで向上します。いずれも、多数の文を同時実行する必要があるデータベースを作成する上で欠かせないことです。

トランザクションは、SQL 文をいくつかの作業の論理単位にグループ分けします。トランザクションを完了するには、すべての変更をロールバックするか、変更をコミットして永続的なものにします。

システム障害が発生したときにデータリカバリを行うためには、トランザクションが必要です。トランザクションは、同時に実行されるトランザクションの文を編成する上で中心的な役割を果たします。

パフォーマンスを向上させるには、複数のトランザクションを同時に実行する必要があります。各トランザクションは、コンポーネント SQL 文で構成されています。複数のトランザクションを同時に実行する場合、データベースサーバは個々の文の実行をスケジュールします。トランザクションを同時に実行すると、順次実行の場合とは異なり、矛盾が生じる可能性があります。

独立性レベルの定義には、次の 4 種類の矛盾が使用されます。

- **ダーティリード**　あるトランザクションがデータを修正した後、コミットする前に別のトランザクションがそのデータを読み込んでしまうこと。
- **繰り返し不可能読み出し**　あるトランザクションが同じローを 2 度読み込んだ場合に、得られる値が異なること。
- **幻ロー**　トランザクションが特定の基準に従ってローを 2 回選択したときに、2 回目の結果セットに新しいローが含まれること。
- **更新内容の消失**　トランザクションがローに対して行った変更が、別のトランザクションが前のデータを基にして更新内容を保存することを許可されたため、完全に消失してしまうこと。

スケジュールに従って文を実行した結果が、各トランザクションを順次実行した結果と同じ場合、そのスケジュールは直列化可能であるといいます。スケジュールが直列化可能である場合、それは「正しいスケジュール」と言えます。直列化可能なスケジュールは上記のような矛盾を引き起こしません。

ロックは、許可する干渉の量とタイプを制御します。SQL Anywhere では、独立性レベル 0、1、2、3 の 4 つのロックレベルを使用できます。最も高い独立性レベル 3 では、SQL Anywhere はスケジュールが直列化可能であること、つまり、すべてのトランザクションを実行した結果とそれらを順次実行した結果が同じになることを保証します。

残念なことに、あるトランザクションが設定したロックが他のトランザクションの進行を妨げることがあります。この問題を解決するには、矛盾が許されるかぎり、低い独立性レベルを使用するのが得策です。独立性レベルが高いほどデータの一貫性は向上しますが、同時実行性は低下し、データベースがトランザクションを同時に実行する効率も低下します。オペレーションごとに最適な独立性を決定するには、一貫性とパフォーマンス向上のバランスを取る必要があります。

異なるトランザクション間でロックの競合が起きると、ブロックまたはデッドロックとなります。SQL Anywhere には、この両方を扱うメカニズムが含まれており、それらを制御するオプションを備えています。

しかし、独立性レベルの高いトランザクションが**必ずしも**同時実行性に影響を与えるわけではありません。他のトランザクションは、ロックされたローにアクセスする場合にだけ妨げられます。データベースとトランザクションを注意深く設計することで、同時実行性を向上させることができます。たとえば、1 つのトランザクションを 2 つの短いトランザクションに分割してロックが保持される時間を短縮したり、インデックスを追加して、独立性レベルの高いトランザクションを少ないロックで実行できます。

LDAP ユーザ認証

Lightweight Directory Access Protocol (LDAP) は、ディレクトリサービスにアクセスするための業界標準です。LDAP ユーザ認証では、クライアントアプリケーションがユーザ ID とパスワード情報をデータベースサーバに送信して、カタログを利用しないで LDAP サーバによって認証を行うことができます。LDAP サーバを使用した認証によって、組織規模のパスワード管理が可能になります。

LDAP ユーザ認証は、既存のコンピューティング環境でユーザ管理を簡素化して集中管理したい組織や、新しいコンピューティング環境で異なるアプリケーションやデータベースのユーザを管理する際の無用な複雑さを回避したいユーザに最適です。

login_mode データベースオプションには LDAPUA モードが含まれています。これによって、認証メカニズムの候補が拡張され、LDAP ユーザ認証が含まれるようになります。SQL Anywhere LDAP ユーザ認証サポートにより、SQL Anywhere は、LDAP に基づいた既存の全社的ディレクトリアクセスフレームワークに統合することができます。

LDAP ユーザ認証の仕組み

各ユーザはログインポリシーに関連付けられます。ログインポリシーは、必要に応じてプライマリ LDAP サーバオブジェクト、セカンダリ LDAP サーバオブジェクト、またはそれら両方をユーザ認証のために参照できます。異なるオプションを使用して複数のログインポリシーを定義することにより、複数のドメインを指定できます。

LDAP サーバオブジェクトを指定するログインポリシーを備えたユーザを認証するには、そのユーザの「識別名 (DN)」 と、DN に関連付けられたパスワードを使用して、信頼できる LDAP ディレクトリサーバに接続します。

クライアントアプリケーションは、通常はその LDAP DN を認識しないか、データベースサーバに送信しません。その代わり、データベースのユーザ ID とパスワードを認識してデータベースサーバに送信します。データベースのユーザ ID とその DN の関連付けは、定義済みの検索文字列を使用して、LDAP サーバ上の特定のユーザ ID の DN を検索することにより行われます。LDAP サーバでの検索からデータベースサーバに返された DN は、ISYSUSER システムテーブル (user_dn カラム) に、現在の UTC 時刻 (user_dn_cached_at カラム) とともに格納されます。ALTER LOGIN POLICY...LDAP_REFRESH_DN=NOW 文を使用して関連するログインポリシーが更新されるまで、格納された DN 値が継続して使用されます。

LDAP サーバ設定オブジェクト

LDAP サーバに接続するための設定情報は、「LDAP サーバ設定オブジェクト」 に格納されます。ISYSLDAPSERVER システムテーブルの各ローは、LDAP サーバ設定オブジェクトの設定を反映します。それぞれの設定オブジェクトには、特定のユーザ ID の DN を取得する際に使用される LDAP 検索文字列など、接続のための一連の属性が含まれます。これらの属性や LDAP サーバのステータスは、SYSLDAPSERVER ビューを照会して調べることができます。

高可用性と LDAP ユーザ認証

ユーザのログインポリシーを介して、特定ユーザのプライマリサーバまたはセカンダリサーバとして LDAP サーバを指定することにより、LDAP サーバの高可用性が実現されます。何らかの理由 (LDAP サーバのメンテナンス、ネットワークの障害、LDAP サーバのクラッシュなど) によっ

てプライマリ LDAP サーバにアクセスできない場合は、セカンダリ LDAP サーバがユーザ認証に使用されます。プライマリからセカンダリにフェイルオーバした後にセカンダリからプライマリにフェールバックする機能は、SQL 文を使用して手動で管理したり、データベースサーバによって切り替えが適切であると判断されたときに自動的に実行することができます。

LDAP ログインポリシーオプションの継承

デフォルトのルートログインポリシーからの継承は、LDAP ポリシーオプションに関してグループで行われます。たとえば、ルートログインポリシーでプライマリ LDAP サーバを定義し、LDAP ポリシーオプションを指定しないログインポリシー policy_X が定義された場合、policy_X に関連付られたユーザは、ルートポリシーからプライマリ LDAP サーバと LDAP ポリシーオプションを継承します。

別のログインポリシー policy_Y がセカンダリ LDAP サーバのみを定義している場合、policy_Y に関連付られたユーザは、そのポリシーからセカンダリ LDAP サーバとその関連する設定のみを使用し、ルートポリシーからの LDAPUA ポリシーオプションは何も継承されません。

参照

- 「login_mode オプション」『SQL Anywhere サーバ データベース管理』
- 「ルートログインポリシー」『SQL Anywhere サーバ データベース管理』
- 「ログインポリシー」『SQL Anywhere サーバ データベース管理』
- 「ALTER LOGIN POLICY 文」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSLDAPSERVER システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSUSER システムビュー」『SQL Anywhere サーバ SQL リファレンス』

LDAP ユーザ認証環境の作成 (SQL の場合)

LDAP ユーザ認証環境を作成するには、LDAP サーバ設定オブジェクトを作成して、LDAP サーバを使用するログインポリシーを作成し、そのログインポリシーを使用して LDAP サーバに認証されるユーザを作成します。

前提条件

LDAP サーバが必要です。

次のシステム権限が必要です。

- SET ANY SECURITY OPTION
- MANAGE ANY LDAP SERVER
- MANAGE ANY LOGIN POLICY
- MANAGE ANY USER

♦ タスク

1. (省略可) LDAP サーバとの通信で TLS 暗号化を有効にするには、使用するサーバの証明書ファイルを指定します。次の文は、証明書ファイルを指定する方法の例です。

```
SET OPTION PUBLIC.trusted_certificates_file='c:¥¥certificates¥¥trusted.txt';
```

以降のステップで TLS 暗号化を有効にするには、それぞれの例で出現する TLS OFF 句を TLS ON 句に変更する必要があります。

2. VALIDATE LDAP SERVER 文を実行して、LDAP サーバへの接続をテストします。

   たとえば、次の文は既存の LDAP サーバの接続属性を検証します。データベースサーバは、指定されたクレデンシャルを使用して LDAP サーバに接続します。

```
VALIDATE LDAP SERVER
   SEARCH DN
      URL 'ldap://iq10web:389/dc=sybase,dc=com?dn?sub?uid=*'
      ACCESS ACCOUNT 'cn=Manager,dc=sybase,dc=com'
      IDENTIFIED BY 'Not4YourEyes'
   AUTHENTICATION URL 'ldap://iq10web:389/'
   CONNECTION TIMEOUT 1000
   CONNECTION RETRIES 3
   TLS OFF;
```

   この LDAP サーバは、ポート 389 を使用して通信します。この例では TLS 暗号化を有効にしません (TLS OFF)。次の手順に進む前に、VALIDATE LDAP SERVER 文がエラーなしで実行される必要があります。

3. CREATE LDAP SERVER 文を実行して、LDAP サーバ設定オブジェクトを作成します。

   たとえば、次の文はユーザ認証に使用できる LDAP サーバ設定オブジェクトを定義します。

```
CREATE LDAP SERVER prim_ldap
   SEARCH DN
      URL 'ldap://iq10web:389/dc=sybase,dc=com?dn?sub?uid=*'
      ACCESS ACCOUNT 'cn=Manager,dc=sybase,dc=com'
      IDENTIFIED BY 'Not4YourEyes'
   AUTHENTICATION URL 'ldap://iq10web:389/'
   CONNECTION TIMEOUT 1000
   CONNECTION RETRIES 3
   TLS OFF;
```

   VALIDATE LDAP SERVER 文とは異なり、CREATE LDAP SERVER 文は LDAP サーバへの接続を試行しません。

4. CREATE LDAP SERVER 文を実行して、フェイルオーバに使用する 2 つ目の LDAP サーバ設定オブジェクトを作成します。この手順は省略できますが、以降の手順で必要になります。

   たとえば、次の文はユーザ認証のフェイルオーバとして使用できる LDAP サーバ設定オブジェクトを定義します。

```
CREATE LDAP SERVER sec_ldap
   SEARCH DN
      URL 'ldap://iq10web:390/dc=sybase,dc=com?dn?sub?uid=*'
      ACCESS ACCOUNT 'cn=Manager,dc=sybase,dc=com'
      IDENTIFIED BY 'Not4YourEyes'
   AUTHENTICATION URL 'ldap://iq10web:390/'
   CONNECTION TIMEOUT 1000
   CONNECTION RETRIES 3
   TLS OFF;
```

   この LDAP サーバは、ポート 390 を使用して通信します。

5. SET OPTION 文を実行してログインモードを変更し、LDAP ユーザ認証を有効にします。

   たとえば、次の文は標準ユーザ認証と LDAP ユーザ認証の両方を有効にします。

   ```
   SET OPTION PUBLIC.login_mode='Standard,LDAPUA';
   ```

   login_mode オプションの現在の設定は次のようにして照会できます。

   ```
   SELECT connection_property('login_mode');
   ```

6. CREATE LOGIN POLICY 文を実行して新しいログインポリシーを作成します。

   たとえば、次の文はユーザの認証に使用できる新しいログインポリシーを作成します。

   ```
   CREATE LOGIN POLICY ldap_policy_both
     LDAP_PRIMARY_SERVER=prim_ldap
     LDAP_SECONDARY_SERVER=sec_ldap
     LDAP_FAILOVER_TO_STD=ON;
   ```

   プライマリ LDAP サーバとセカンダリ LDAP サーバの名前は、このログインポリシーで指定されます。このログインポリシーに関連付けられているユーザの認証が失敗した場合は、標準認証が試行されます (ログインモードによって許可されている場合)。ログインポリシーの現在の設定は次のようにして照会できます。

   ```
   SELECT lpo.* FROM sysloginpolicyoption AS lpo, sysloginpolicy AS lp
     WHERE lpo.login_policy_id = lp.login_policy_id
       AND lp.login_policy_name = 'ldap_policy_both';
   ```

7. CREATE USER 文を実行して、前述の手順で定義された LDAP ログインポリシーを備えた新しいユーザ ID を作成します。

   たとえば、次の文はプライマリ LDAP サーバまたはセカンダリ LDAP サーバに対して認証される新しいユーザ ID を作成します。

   ```
   CREATE USER ldap_user01 LOGIN POLICY ldap_policy_both;
   ```

   IDENTIFIED BY 句は省略します。IDENTIFIED BY 句は、標準認証に使用するパスワードを指定するために使用します。この句を使用してパスワードを指定する場合は、LDAP サーバによって認証されるパスワードと同じにする必要はありません。このパスワードは、ユーザが初めて LDAP に正しく認証されると置き換えられるため、ここでは IDENTIFIED BY 句を省略します。

8. LDAP サーバをアクティブにしてすぐに使用できるようにします。次の文はプライマリ LDAP サーバとセカンダリ LDAP サーバをアクティブにします。

   ```
   ALTER LDAP SERVER prim_ldap WITH ACTIVATE;
   ALTER LDAP SERVER sec_ldap WITH ACTIVATE;
   ```

   すべての LDAP サーバの現在のステータスは、次のクエリを使用して判別できます。

   ```
   call sa_get_ldapserver_status;
   ```

   結果セットの ldsrv_state カラムは、2 つの LDAP サーバのステータスが READY であることを示しています。

9. Interactive SQL の CONNECT 文を実行し、LDAP ユーザ認証を使用してデータベースに接続します。

   たとえば、次の文は指定されたユーザ ID とパスワードを使用してサンプルデータベースに接続します。

   ```
   CONNECT DATABASE demo USER ldap_user01 IDENTIFIED BY 'abcd1234';
   ```

   LDAP サーバがユーザ認証に失敗すると、標準認証が試行されます。標準パスワードが LDAP 認証の際に更新されていないなどの理由で、標準パスワードが指定のパスワードと一致しない場合、標準認証は失敗します。

結果

ユーザが LDAP ユーザ認証を介して SQL Anywhere データベースに接続されます。

参照

- 「VALIDATE LDAP SERVER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE LDAP SERVER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「login_mode オプション」『SQL Anywhere サーバ データベース管理』
- 「trusted_certificates_file オプション」『SQL Anywhere サーバ データベース管理』
- 「CREATE LOGIN POLICY 文」『SQL Anywhere サーバ SQL リファレンス』
- 「CREATE USER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「sa_get_ldapserver_status システムプロシージャ」『SQL Anywhere サーバ SQL リファレンス』

LDAP ユーザ認証環境の作成 (Sybase Central の場合)

LDAP ユーザ認証環境を設定するには、次のタスクを実行します。

1. LDAP サーバ設定オブジェクトを作成します。
2. LDAP サーバを使用するログインポリシーを作成します。
3. 定義したログインポリシーを使用して LDAP サーバに認証されるユーザを作成します。

これらのタスクを実行するには、次のシステム権限が必要です。

- SET ANY SECURITY OPTION
- MANAGE ANY LDAP SERVER
- MANAGE ANY LOGIN POLICY
- MANAGE ANY USER

参照

- 「LDAP サーバ設定オブジェクトの作成」940 ページ
- 「LDAP サーバを使用するログインポリシーの作成」941 ページ
- 「LDAP サーバに認証されるユーザの作成」942 ページ

LDAP サーバ設定オブジェクトの作成

Sybase Central を使用して 2 つの LDAP サーバ設定オブジェクトを作成します。

前提条件

MANAGE ANY LDAP SERVER および SET ANY SECURITY OPTION システム権限が必要です。

内容と備考

各 LDAP サーバには TCP/IP を使用してアクセスします。LDAP サーバの定義は、ユーザ認証に使用する LDAP サーバごとに必要です。次に示す応答は、デモンストレーションのみを目的としたものです。適切な同等の応答を指定してください。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で **[LDAP サーバ]** を右クリックし、**[新規]** » **[LDAP サーバ]** をクリックします。
3. LDAP サーバの名前では、サーバ名 (**prim_ldap** など) を入力して、**[次へ]** をクリックします。
4. ネットワーク暗号化のタイプを選択します (**暗号化なし (ldap:)** など)。
5. LDAP サーバのホスト名とポート番号では、サーバ名とポートを入力します (**iq10web** と **389** など)。
6. **[認証 URL]** に URL が表示されます (**ldap://iq10web:389/** など)。**[次へ]** をクリックします。
7. 検索 URL では、この URL を直接 **[検索 URL]** フィールドに入力できます (**ldap://iq10web:389/dc=sybase,dc=com?dn?sub?uid=*** など)。
8. **[識別名]** に識別名を入力します (**cn=Manager,dc=sybase,dc=com** など)。
9. LDAP に認証される際に必要なパスワードが存在する場合は、そのパスワードを **[パスワード]** フィールドに入力します (**Not4YourEyes** など)。
10. **[次へ]** をクリックします。
11. 接続タイムアウトでは、接続タイムアウトの間隔を選択します (**1** 秒など)。
12. 接続リトライ回数では、回数 (**4** など) を選択して、**[次へ]** をクリックします。
13. LDAP サーバをアクティブにするには、**[この LDAP サーバを今すぐアクティブ化]** を選択します。
14. **[ログインモード]** を選択します (少なくとも **[標準]** と **[LDAPUA]** は選択する必要があります)。

15. **[接続テスト]** をクリックして、LDAP 接続パラメータを検証します。たとえば、ダイアログが開いたら **[接続テスト]** をクリックします。接続パラメータを正しく入力していた場合は、接続に成功します。**[閉じる]** をクリックしてから、**[次へ]** をクリックします。

16. (省略可) コメントを追加します。**[次へ]** をクリックします。

17. LDAP サーバ設定オブジェクトを作成するために実行される SQL が表示されます。**[完了]** をクリックします。

結果

LDAP サーバ設定オブジェクトが作成され、そのオブジェクトへの参照が ISYSLDAPSERVER システムテーブルに追加されます。

次の手順

(省略可) フェイルオーバに使用される 2 つ目の LDAP サーバ設定オブジェクトを作成します。前述の同じ手順に従いますが、サーバ名として **sec_ldap** を使用し、ポートとして 390 を使用します。

このタスクを完了した後に、LDAP サーバを使用するログインポリシーを作成します。

参照

- 「LDAP サーバを使用するログインポリシーの作成」941 ページ
- 「login_mode オプション」『SQL Anywhere サーバ データベース管理』
- 「trusted_certificates_file オプション」『SQL Anywhere サーバ データベース管理』
- 「SYSLDAPSERVER システムビュー」『SQL Anywhere サーバ SQL リファレンス』

LDAP サーバを使用するログインポリシーの作成

Sybase Central を使用して、LDAP サーバを使用するログインポリシーを作成します。

前提条件

MANAGE ANY LOGIN POLICY システム権限が必要です。

内容と備考

LDAP サーバを使用してユーザを認証する場合は、ログインポリシーが必要です。次に示す応答は、デモンストレーションのみを目的としたものです。適切な同等の応答を指定してください。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。

2. 左ウィンドウ枠で **[ログインポリシー]** を右クリックし、**[新規]** » **[ログインポリシー]** をクリックします。

3. ログインポリシーの名前 (**ldap_policy_both** など) を入力して、**[次へ]** をクリックします。
4. **[LDAP プライマリサーバ]** を選択します (**prim_ldap** など)。
5. (省略可) **[LDAP セカンダリサーバ]** を選択します (**sec_ldap** など)。
6. (省略可だが推奨) **[標準認証への LDAP フェイルオーバ]** を選択します (**[On]** など)。
7. **[次へ]** をクリックします。
8. (省略可) コメントを追加します。**[次へ]** をクリックします。
9. ログインポリシーを作成するために実行される SQL が表示されます。**[完了]** をクリックします。

結果

ユーザ認証に LDAP サーバを使用するログインポリシーが作成され、その定義が SYSLOGINPOLICY システムテーブルと SYSLOGINPOLICYOPTION システムテーブルに追加されます。

次の手順

LDAP サーバに認証されるユーザを作成します。

参照

- 「LDAP サーバ設定オブジェクトの作成」940 ページ
- 「LDAP サーバに認証されるユーザの作成」942 ページ
- 「CREATE LOGIN POLICY 文」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSLOGINPOLICY システムビュー」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSLOGINPOLICYOPTION システムビュー」『SQL Anywhere サーバ SQL リファレンス』

LDAP サーバに認証されるユーザの作成

Sybase Central を使用して、LDAP サーバに認証されるユーザを作成します。

前提条件

MANAGE ANY USER システム権限が必要です。

内容と備考

LDAP サーバに認証されるユーザがデータベースに特定される必要があります。次に示す応答は、デモンストレーションのみを目的としたものです。適切な同等の応答を指定してください。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。

2. 左ウィンドウ枠で **[ユーザ]** を右クリックし、**[新規]** » **[ユーザ]** をクリックします。
3. 新しいユーザの名前 (**ldap_user01** など) を入力して、**[次へ]** をクリックします。
4. **[このユーザにパスワードを割り当てる]** が選択されていないことを確認します。
5. **[ログインポリシー]** の項目で、ログインポリシーを選択します (**ldap_policy_both** など)。
6. **[次へ]** をクリックします。
7. (省略可) コメントを追加します。**[次へ]** をクリックします。
8. ユーザを作成するために実行される SQL が表示されます。**[完了]** をクリックします。

結果

LDAP サーバを使用して認証されるデータベースユーザが作成されます。LDAP サーバを使用できない場合は標準認証が行われます。

次の手順

ユーザ ID と LDAP パスワードを使用してデータベースに接続します。

参照

- 「LDAP サーバ設定オブジェクトの作成」940 ページ
- 「LDAP サーバを使用するログインポリシーの作成」941 ページ
- 「CREATE USER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSUSER システムビュー」『SQL Anywhere サーバ SQL リファレンス』

LDAP サーバ設定オブジェクトのアクティブ化または中断

LDAP サーバで使用する LDAP サーバ設定オブジェクトをアクティブにしたり、中断したりします。

前提条件

MANAGE ANY LDAP SERVER システム権限が必要です。

内容と備考

LDAP サーバ設定オブジェクトを修正した後は、そのオブジェクトを再び有効にする必要があり、LDAP サーバ設定オブジェクトを修正したり削除するには、そのオブジェクトを無効にする必要があります。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[LDAP サーバ]** をクリックします。

3. 右ウィンドウ枠で、LDAP サーバ設定を右クリックし、**[アクティブ化]** または **[中断]** をクリックします。

4. **[OK]** をクリックします。

結果

LDAP サーバ設定オブジェクトのステータスが更新されます。

参照

- 「ALTER LDAP SERVER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSLDAPSERVER システムビュー」『SQL Anywhere サーバ SQL リファレンス』

LDAP サーバ設定オブジェクトの変更

LDAP ユーザ認証用の LDAP サーバに接続するための LDAP サーバ設定オブジェクトを変更します。

前提条件

MANAGE ANY LDAP SERVER システム権限が必要です。

内容と備考

Sybase Central は、LDAP サーバ設定オブジェクトを変更する前に無効にして、操作が完了したら再び有効にします。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。

2. 左ウィンドウ枠で、**[LDAP サーバ]** をクリックします。

3. 右ウィンドウ枠で、変更する LDAP サーバ設定を右クリックし、**[プロパティ]** をクリックします。

4. **[一般]** タブと **[設定]** タブのプロパティを必要に応じて編集して、**[OK]** をクリックします。

結果

ISYSLDAPSERVER システムテーブルでの LDAP サーバ設定オブジェクトの設定が更新されます。

参照

- 「LDAP サーバ設定オブジェクトのアクティブ化または中断」943 ページ
- 「ALTER LDAP SERVER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSLDAPSERVER システムビュー」『SQL Anywhere サーバ SQL リファレンス』

LDAP サーバ設定オブジェクトの検証

LDAP サーバに接続して、LDAP 設定オブジェクトが正しく設定されていることを確認します。

前提条件

MANAGE ANY LDAP SERVER システム権限が必要です。

LDAP サーバ設定オブジェクトに関連付けられている LDAP サーバにアクセスできる必要があります。

内容と備考

LDAP 設定オブジェクトの設定を使用してデータベースサーバから LDAP サーバに接続し、それらが有効であることを確認します。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[LDAP サーバ]** をクリックします。
3. 右ウィンドウ枠で、LDAP サーバ設定を右クリックし、**[接続テスト]** をクリックします。

結果

データベースサーバによって LDAP サーバ設定オブジェクトが検証されます。検証中に問題が発生した場合は、データベースサーバから警告が返されます。

参照

- 「VALIDATE LDAP SERVER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSLDAPSERVER システムビュー」『SQL Anywhere サーバ SQL リファレンス』

LDAP サーバ設定オブジェクトの削除

LDAP サーバへの接続に使用される LDAP サーバ設定オブジェクトを削除します。

前提条件

MANAGE ANY LDAP SERVER システム権限が必要です。

内容と備考

Sybase Central によって、ログインポリシーから LDAP サーバ設定オブジェクトのすべての参照が削除されてから、LDAP サーバ設定オブジェクトが削除されます。

Sybase Central では、LDAP サーバ設定オブジェクトが自動的に停止されてから削除されます。

♦ **タスク**

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. 左ウィンドウ枠で、**[LDAP サーバ]** をクリックします。
3. 右ウィンドウ枠で、LDAP サーバ設定を右クリックし、**[削除]** をクリックします。
4. **[はい]** をクリックして削除を確定します。

結果

データベースサーバによって LDAP サーバ設定オブジェクトが削除され、その参照が ISYSLDAPSERVER システムテーブルから削除されます。

参照

- 「DROP LDAP SERVER 文」『SQL Anywhere サーバ SQL リファレンス』
- 「SYSLDAPSERVER システムビュー」『SQL Anywhere サーバ SQL リファレンス』

SQL Anywhere のデバッガ

SQL Anywhere のデバッガは、作成した SQL のストアドプロシージャ、トリガ、イベントハンドラ、ユーザ定義関数をデバッグするために使用できます。

デバッガは以下の作業に使用することもできます。

- **イベントハンドラのデバッグ**　イベントハンドラは SQL ストアドプロシージャの拡張機能です。この項でのストアドプロシージャのデバッグに関する説明は、イベントハンドラのデバッグにも同様に当てはまります。
- **ストアドプロシージャとクラスのブラウズ**　SQL プロシージャのソースコードをブラウズできます。
- **実行のトレース**　ストアドプロシージャのコードを 1 行ずつ実行できます。呼び出された関数のスタックを前後に検索することもできます。
- **ブレークポイントの設定**　ブレークポイントまでコードを実行して停止します。
- **ブレーク条件の設定**　ブレークポイントには複数行のコードが含まれますが、コードをブレークする場合、条件も指定できます。たとえば、ある行を 10 回実行された時点で停止させたり、変数が特定の値を持つ場合にだけ停止させたりできます。
- **ローカル変数の検査と修正**　実行がブレークポイントで停止したときに、ローカル変数の値を検査して変更できます。
- **式を検査してブレークする**　実行がブレークポイントで停止したときに、さまざまな式の値を検査できます。
- **ロー変数の検査と修正**　ロー変数はローレベルトリガの OLD と NEW の値です。これらの値を検査して修正できます。
- **クエリの実行**　SQL プロシージャのブレークポイントで実行が停止したときに、クエリを実行できます。これによって、テンポラリテーブルに保持される中間結果を参照したり、ベーステーブルの値をチェックしたり、クエリ実行プランを参照したりできます。

Java デバッガの稼働条件

1 つのデータベースは、一度に 1 ユーザだけがデバッグできます。

HTTP/SOAP 接続を介してデバッガを使用する場合は、サーバのポートのタイムアウトオプションを変更してください。たとえば、-xs http{TO=600;KTO=0;PORT=8081) では、ポート 8081 のタイムアウトが 10 分に設定され、キープアライブのタイムアウトがオフになります。Timeout (TO) は受信パケットと次の受信パケットの間に経過する時間であることに注意してください。キープアライブタイムアウト (KTO) は、接続が実行を許可される時間の合計時間です。KTO を 0 に設定することは、タイムアウトしないように設定することと同じです。

SQL Anywhere の HTTP/SOAP クライアントプロシージャを使用して、デバッグしている SQL Anywhere HTTP/SOAP サービスを呼び出す場合、クライアントの remote_idle_timeout データベー

スオプションを 150 などの大きい値 (デフォルトは 15 秒) に設定して、デバッグセッション中のタイムアウトを回避します。

参照

- 「-xs データベースサーバオプション」『SQL Anywhere サーバ データベース管理』
- 「remote_idle_timeout オプション」『SQL Anywhere サーバ データベース管理』
- 「KeepaliveTimeout (KTO) プロトコルオプション」『SQL Anywhere サーバ データベース管理』
- 「Timeout (TO) プロトコルオプション」『SQL Anywhere サーバ データベース管理』

チュートリアル：デバッガの使用開始

このチュートリアルでは、デバッガを使用してストアドプロシージャのエラーを識別する方法を説明します。

SQL Anywhere のサンプルデータベース、*demo.db* には、debugger_tutorial という名前の意図的にエラーが含まれたストアドプロシージャがあります。debugger_tutorial システムプロシージャは、発注額の最も多い会社名とその発注額を含む結果セットを返します。プロシージャは、会社と発注をリストするクエリの結果セットをループしてこれらの値を計算します。(この結果は、プロシージャにこの論理を追加しなくとも、SELECT FIRST クエリを使用して得ることができます。このプロシージャは説明を目的としたものです)。ただし、debugger_tutorial システムプロシージャにはバグが含まれているためエラーになり、正しい結果セットが返されません。

権限

SA_DEBUG システムロールが必要です。

さらに、EXECUTE ANY PROCEDURE システム権限か、システムプロシージャ debugger_tutorial に対する EXECUTE 権限が必要です。また、ALTER ANY PROCEDURE システム権限または ALTER ANY OBJECT システム権限も必要です。

参照

- 「SQL Anywhere のデバッガ」947 ページ
- 「Java デバッガの稼働条件」947 ページ

レッスン 1：デバッガの起動とバグの検査

デバッガを起動して debugger_tutorial ストアドプロシージャを実行し、バグを検査します。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：デバッガの使用開始」

♦ タスク

1. このチュートリアルで使用するサンプルデータベースのコピーを作成します。

a. データベースを保持するディレクトリ (*c:¥demodb* など) を作成します。

b. 次のコマンドを実行して、データベースを作成します。

```
newdemo c:¥demodb¥demo.db
```

2. Sybase Central を起動します。**[スタート]** » **[プログラム]** » **[SQL Anywhere 16]** » **[管理ツール]** » **[Sybase Central]** をクリックします。

3. Sybase Central で、次のように *demo.db* に接続します。

   a. **[接続]** » **[SQL Anywhere 16 に接続]** をクリックします。

   b. **[接続]** ウィンドウで、データベースに接続するための情報を以下のフィールドに入力します。

      i. **[ユーザ ID]** フィールドに、**DBA** と入力します。

      ii. **[パスワード]** フィールドに、**sql** と入力します。

      iii. **[アクション]** ドロップダウンリストで、**[このコンピュータのデータベースを起動して接続]** をクリックします。

      iv. **[データベースファイル]** フィールドに、**c:¥demodb¥demo.db** と入力します。

      v. **[サーバ名]** フィールドに、**demo_server** と入力します。

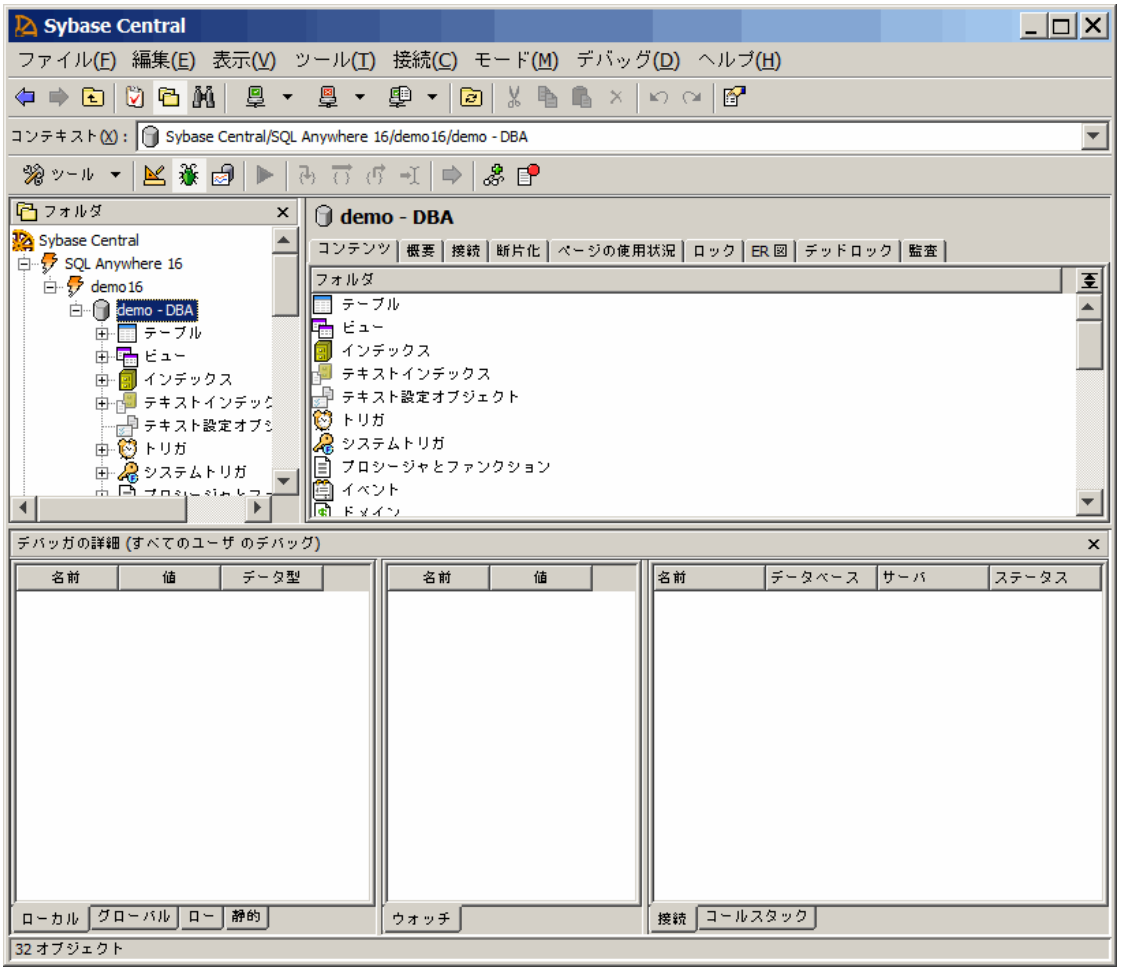
   c. **[接続]** をクリックします。

4. **[モード]** » **[デバッグ]** をクリックします。

5. **[デバッグするユーザを指定してください]** フィールドに ***** と入力し、**[OK]** をクリックします。

   Sybase Central の最下部に **[デバッガの詳細]** ウィンドウ枠が表示され、Sybase Central のツールバーには一連のデバッガツールが表示されます。

   ***** を指定すると、すべてのユーザをデバッグできます。デバッグ対象のユーザを変更するには、いったんデバッグモードを終了して再びデバッグモードに入ります。ユーザ ID を入力すると、そのユーザ ID を使用した接続に関する情報が取得され、**[接続]** タブに表示されます。



6. Sybase Central の左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックします。

7. **[debugger_tutorial (GROUPO)]** を右クリックし、**[Interactive SQL から実行]** をクリックします。

   Interactive SQL が開き、次の結果セットが表示されます。

top_company	top_value
(NULL)	(NULL)

   この結果セットは間違っています。チュートリアルの続きで、この結果をもたらしたエラーを診断します。

8. [Interactive SQL] ウィンドウが開いている場合は閉じます。

結果

デバッガが起動して、debugger_tutorial ストアドプロシージャでバグが検出されます。

次の手順

「レッスン 2：バグの診断」951 ページに進みます。

参照

- 「SQL Anywhere のデバッガ」947 ページ
- 「サンプルデータベースの再作成 (demo.db)」『SQL Anywhere 16 紹介』
- 「チュートリアル：サンプルデータベースへの接続」『SQL Anywhere サーバ データベース管理』

レッスン 2：バグの診断

debugger_tutorial ストアドプロシージャでバグを診断するには、ブレークポイントを設定してコードをステップスルーし、プロシージャの実行にともなって変数の値を監視します。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：デバッガの使用開始」

このレッスンは、受講者がこれまでのすべてのレッスンを終了していることを前提としています。「レッスン 1：デバッガの起動とバグの検査」948 ページを参照してください。

♦ タスク

1. 右ウィンドウ枠で、**[debugger_tutorial (GROUPO)]** の **[SQL]** タブに次の文があることを確認します。

   ```
   OPEN cursor_this_customer;
   ```

2. ブレークポイントを追加するには、文の左側にあるグレーの縦線の領域をクリックします。ブレークポイントは、赤い円で表示されます。

3. 左ウィンドウ枠で、**[debugger_tutorial (GROUPO)]** を右クリックし、**[Interactive SQL から実行]** をクリックします。

   Sybase Central の右枠で、ブレークポイントの上部に黄色の矢印が表示されます。

4. **[デバッガの詳細]** ウィンドウで **[ローカル]** タブをクリックし、プロシージャ内のローカル変数とその現在の値およびデータ型のリストを表示します。**Top_Company**、**Top_Value**、**This_Value**、**This_Company** の各変数は、すべて初期化されていないため、NULL を示しています。

5. [F11] キーを押して、プロシージャをスクロールします。次の行に到達すると、変数の値が変化します。

```
IF SQLSTATE = error_not_found THEN
```

6. [F11] キーをもう 2 回押して、分岐を確認します。黄色の矢印が、次のテキストに戻ります。

```
customer_loop: loop
```

IF テストは TRUE を返しませんでした。テストが失敗したのは、NULL とどのような値を比較しても NULL が返されるためです。NULL 値によりテストは失敗し、IF...END IF 文内のコードは実行されませんでした。

このことから、**Top_Value** が初期化されていないことが原因とわかります。

7. プロシージャコードを変更しないで、**Top_Value** が初期化されていないことが問題であるとする仮説をテストします。
   a. **[デバッガの詳細]** ウィンドウで、**[ローカル]** タブをクリックします。
   b. **[Top_Value]** 変数をクリックし、**[値]** フィールドに **3000** と入力し、[Enter] を押します。
   c. **[This_Value]** 変数の **[値]** フィールドが 3000 よりも大きな値になるまで [F11] キーを繰り返し押します。
   d. ブレークポイントをクリックして、グレーにします。
   e. [F5] キーを押して、プロシージャを実行します。

      [Interactive SQL] ウィンドウが再び表示されて、正しい結果が表示されます。

top_company	top_value
Chadwicks	8076

   f. [Interactive SQL] ウィンドウが開いている場合は閉じます。

結果

仮説が正しいことが確認されました。**Top_Value** 変数が初期化されていないことが原因でした。

次の手順

「レッスン 3：バグの修正」952 ページに進みます。

参照

- 「ブレークポイント」953 ページ
- 「ブレークポイントの設定」954 ページ

レッスン 3：バグの修正

前述のレッスンで特定されたバグを修正するには、**Top_Value** 変数を初期化します。

前提条件

このレッスンでは、このチュートリアルの開始時に、権限のセクションで一覧されているロールと権限を持っていることを前提としています。 「チュートリアル：デバッガの使用開始」

このレッスンは、受講者がこれまでのすべてのレッスンを終了していることを前提としています。「レッスン 1：デバッガの起動とバグの検査」948 ページを参照してください。

♦ タスク

1. **[モード]** » **[設計]** をクリックします。
2. 右ウィンドウ枠で、次の文を検索します。

   ```
   OPEN cursor_this_customer;
   ```

3. **Top_Value** 変数を初期化する次の行を下に入力します。

   ```
   SET top_value = 0;
   ```

4. **[ファイル]** » **[保存]** をクリックします。
5. プロシージャを再度実行し、Interactive SQL に正しい結果が表示されることを確認します。
6. [Interactive SQL] ウィンドウが開いている場合は閉じます。

結果

バグが修正され、プロシージャが期待どおりに実行されます。 これでデバッグに関するチュートリアルは終了です。

次の手順

このチュートリアルで使用したサンプルデータベースのコピーが含まれているディレクトリ (*c:¥demodb* など) を削除します。

参照

- 「変数」957 ページ
- 「変数値の表示」957 ページ

ブレークポイント

ブレークポイントでは、デバッガがソースコードの実行を中断するタイミングを制御します。

[デバッグ] モードで実行し、接続がブレークポイントに到達した場合、その動作は選択した接続によって変わります。

- 接続を選択していない場合は、接続は自動的に選択され、プロシージャのソースコードが表示されてデバッグされます。

- 接続が選択済みで、その接続がブレークポイントに到達した接続と同じである場合は、プロシージャのソースコードが表示されます。
- 接続は選択済みであるが、その接続がブレークポイントに到達した接続とは異なる場合、ウィンドウが開き、ブレークポイントに到達した接続に変更するように求めるメッセージが表示されます。

ブレークポイントの設定

管理者は、Sybase Central にブレークポイントを設定できます。ブレークポイントは、指定した行で実行を中断するようデバッガに指示します。デフォルトでは、ブレークポイントはすべての接続に適用されます。

前提条件

SA_DEBUG システムロールが必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[モード]** » **[デバッグ]** をクリックします。
3. **[デバッグするユーザを指定してください]** フィールドに * を入力してすべてのユーザをデバッグするか、デバッグ対象となるデータベースユーザの名前を入力します。

オプション	アクション
Sybase Central 右ウィンドウ枠	a. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックして、プロシージャを選択します。 b. 右ウィンドウ枠で、ブレークポイントを挿入する行をクリックします。 クリックした行にカーソルが表示されます。 c. [F9] キーを押します。 コード行の左側に赤い円が表示されます。

オプション	アクション
[デバッグ] メニュー	a. **[デバッグ]** » **[ブレークポイント]** をクリックします。 b. **[新規]** をクリックします。 c. **[プロシージャ]** リストからプロシージャを選択します。 d. 必要に応じて、**[条件]** フィールドと **[カウント]** フィールドにも入力します。 条件とは、ブレークポイントが処理を中断するために TRUE に評価されなければならない SQL 式です。 カウントとは、実行が停止するまでの、ブレークポイントのヒット数です。0 を指定した場合、そのブレークポイントで常に実行が停止されます。 e. **[OK]** をクリックします。ブレークポイントが、プロシージャ内の最初の実行可能な文に設定されます。

結果

ブレークポイントが設定されます。

例

特定のユーザによって作成された接続に適用されるブレークポイントを設定できます。それには、次のような条件を入力します。

```
CURRENT USER = 'user-name'
```

ブレークポイントのステータスの変更

Sybase Central でブレークポイントのステータスを変更します。

前提条件

SA_DEBUG システムロールが必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[モード]** » **[デバッグ]** をクリックします。

3. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックして、プロシージャを選択します。

オプション	アクション
Sybase Central 右ウィンドウ枠	右ウィンドウ枠で、編集する行の左側にあるブレークポイントインジケータをクリックします。ブレークポイントがアクティブから非アクティブに変わります。
[ブレークポイント] ウィンドウ	a. **[デバッグ]** » **[ブレークポイント]** をクリックします。 b. ブレークポイントを選択し、**[編集]**、**[無効にする]**、または **[削除]** をクリックします。 c. **[閉じる]** をクリックします。

結果

ブレークポイントのステータスが変更されます。

ブレークポイント条件の編集

ブレークポイントに条件を追加して、特定の条件または回数を満たす場合だけそのブレークポイントで実行を中断するよう、デバッガに指示を出します。プロシージャとトリガについては、SQL 探索条件にします。

前提条件

SA_DEBUG システムロールが必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[モード]** » **[デバッグ]** をクリックします。
3. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックして、プロシージャを選択します。
4. **[デバッグ]** » **[ブレークポイント]** をクリックします。
5. 編集するブレークポイントを選択し、**[編集]** をクリックします。
6. **[条件]** リストで、条件をクリックします。たとえば、特定のユーザ ID からの接続にのみ適用するようにブレークポイントを設定するには、次の条件を入力します。

   ```
   CURRENT USER='user-name'
   ```

user-name は、ブレークポイントをアクティブにする対象のユーザ ID です。

7. **[OK]** をクリックしてから、**[閉じる]** をクリックします。

結果

条件がブレークポイントに設定されます。

変数

デバッガでは、コードをステップしながら変数の動作を表示して編集できます。デバッガには、ストアドプロシージャで使用するさまざまな種類の変数を表示する **[デバッガの詳細]** ウィンドウ枠が用意されています。**[デバッガの詳細]** ウィンドウ枠は、Sybase Central をデバッグモードで実行しているときに、Sybase Central の最下部に表示されます。

変数値の表示

Sybase Central で変数の値を表示します。

前提条件

SA_DEBUG システムロールが必要です。

さらに、EXECUTE ANY PROCEDURE システム権限、またはそのプロシージャに対する EXECUTE 権限が必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。
2. **[モード]** » **[デバッグ]** をクリックします。
3. **[デバッグするユーザを指定してください]** フィールドに、* を入力してすべてのユーザをデバッグするか、デバッグ対象となるデータベースユーザの名前を入力します。
4. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックして、プロシージャを選択します。
5. 右ウィンドウ枠で、ブレークポイントを挿入する行をクリックします。

   クリックした行にカーソルが表示されます。
6. [F9] キーを押します。

   コード行の左側に赤い円が表示されます。
7. **[デバッガの詳細]** ウィンドウ枠で、**[ローカル]** タブをクリックします。

8. 左ウィンドウ枠で、プロシージャを右クリックし、**[Interactive SQL から実行]** をクリックします。

9. **[ローカル]** タブをクリックします。

結果

変数は、値と共に表示されます。

グローバル変数

グローバル変数は SQL Anywhere によって定義され、現在の接続、データベース、その他の設定に関する情報がグローバル変数に格納されます。グローバル変数は、**[グローバル]** タブの **[デバッガの詳細]** ウィンドウ枠に表示されます。

ロー変数は、トリガで、トリガ元の文が対象とするローの値を保持するために使用します。ロー変数は、**[ロー]** タブの **[デバッガの詳細]** ウィンドウ枠に表示されます。

静的変数は Java クラスで使用します。静的変数は **[静的]** タブに表示されます。

参照

- 「グローバル変数」『SQL Anywhere サーバ SQL リファレンス』
- 「トリガ」94 ページ

呼び出しスタックの表示

ネストされたプロシージャをデバッグするときに、行われた呼び出しの順番を検査できます。プロシージャのリストは、**[コールスタック]** タブで参照できます。

前提条件

SA_DEBUG システムロールが必要です。

さらに、EXECUTE ANY PROCEDURE システム権限、またはそのプロシージャに対する EXECUTE 権限が必要です。

♦ タスク

1. Sybase Central で、**[SQL Anywhere 16]** プラグインを使用してデータベースに接続します。

2. **[モード]** » **[デバッグ]** をクリックします。

3. **[デバッグするユーザを指定してください]** フィールドに、* を入力してすべてのユーザをデバッグするか、デバッグ対象となるデータベースユーザの名前を入力します。

4. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックして、プロシージャを選択します。

5. 右ウィンドウ枠で、ブレークポイントを挿入する行をクリックします。

   クリックした行にカーソルが表示されます。

6. [F9] キーを押します。

   コード行の左側に赤い円が表示されます。

7. **[デバッガの詳細]** ウィンドウ枠で、**[ローカル]** タブをクリックします。

8. 左ウィンドウ枠で、プロシージャを右クリックし、**[Interactive SQL から実行]** をクリックします。

9. **[デバッガの詳細]** ウィンドウ枠で、**[コールスタック]** タブをクリックします。

結果

[コールスタック] タブにプロシージャの名前が表示されます。リストの一番上に現在のプロシージャが表示されます。そのプロシージャを呼び出したプロシージャがそのすぐ下に表示されます。

接続デバッグ

[接続] タブには、データベースへの接続が表示されます。常に複数の接続が実行されています。あるものはブレークポイントで停止し、あるものは停止していません。

接続を切り替えるには、**[接続]** タブで接続をダブルクリックします。

ブレークポイントを設定して、単一のユーザ ID に対して実行を中断するようにすると便利です。このためには、次の形式でブレークポイント条件を設定します。

```
CURRENT USER = 'user-name'
```

SQL の特別値、CURRENT USER には接続のユーザ ID が保持されます。

参照

- 「ブレークポイント条件の編集」956 ページ
- 「CURRENT USER 特別値」『SQL Anywhere サーバ SQL リファレンス』

索引

記号

@@error グローバル変数
　戻り値, 664
@@identity グローバル変数
　IDENTITY カラム, 654
*=
　Transact-SQL 外部ジョイン, 505
* (アスタリスク)
　SELECT 文, 286
<
　WHERE 句での比較演算子, 299
=*
　Transact-SQL 外部ジョイン, 505
>
　WHERE 句での比較演算子, 299
-im オプション
　パフォーマンス向上のためのヒント, 238
権限
　トリガ, 104

A

a_init_pre_filter
　事前フィルタエントリポイント関数, 466
a_init_pre_filter 構造体
　説明, 452
a_init_term_breaker
　単語区切りエントリポイント関数, 467
a_init_term_breaker 構造体
　説明, 457
a_server_context 構造体
　説明, 450
a_term_breaker_for 列挙
　説明, 459
a_term 構造体
　説明, 463
a_text_source インタフェース
　説明, 454
a_word_source インタフェース
　説明, 459
access_date_time
　ディレクトリアクセスサーバ, 781
Adaptive Server Enterprise
　SQL Anywhere への移行, 761
　アーキテクチャ, 642
　エミュレート, 648, 649
　オブジェクト名の互換性, 652
　互換性, 640
　サーバクラス, 814
　データ型の変換, 815
　データのインポート／エクスポートの互換性, 770
　特殊な IDENTITY カラム, 654
Adaptive Server Enterprise の互換性
　説明, 770
ADSODBC サーバクラス
　説明, 812
Advantage Database Server
　ODBC サーバクラス, 812
ALL
　キーワードと UNION 句, 483
　サブクエリのテスト, 608
allow_nulls_by_default オプション
　Transact-SQL 互換性の設定, 650
allow_snapshot_isolation オプション
　使用, 871
All-rows 最適化ゴール
　オプティマイザの目標の選択, 220
　パフォーマンス, 230
ALL 演算子
　サブクエリのテスト, 607
　説明, 607
　注意, 608
ALTER INDEX 文
　スナップショットアイソレーションでは使用不可, 869
ALTER SERVER 文
　リモートサーバの変更, 779
ALTER TABLE 文
　外部キー, 23
　検査制約, 843
　スナップショットアイソレーションでは使用不可, 869
　同時実行性, 932
　プライマリキー, 19
ALTER TRIGGER 文
　使用, 99
ALTER 文
　オートコミット, 861
AND
　論理演算子の使用, 308
ANSI
　ANSI 以外のジョイン, 496

SQL 標準と矛盾, 875
ANSI update constraints
実行プラン, 360
ANSI 以外のジョイン
説明, 496
ANY 演算子
サブクエリのテスト, 605
説明, 605
問題, 606
ASEODBC サーバクラス
説明, 814
AS キーワード
SQL Anywhere 実装, 639
AS キーワード
エイリアス, 288
auto_commit オプション
Interactive SQL での変更のグループ分け, 862
AUTOINCREMENT
IDENTITY カラム, 654
Ultra Light アプリケーション, 839
使用する場合, 928
シーケンスとの相違点, 929
符号付データ型, 839
負の値, 839
automatic_timestamp オプション
Transact-SQL 互換性の設定, 650
AUTO REFRESH 句
説明, 410
AUTO モード
使用, 682
AvgDiskReads
アクセスプラン内の推定, 359
AvgDiskReadTime
アクセスプラン内の推定, 359
AvgDiskWrites
アクセスプラン内の推定, 359
AvgRowCount
アクセスプラン内の推定, 359
AvgRunTime
アクセスプラン内の推定, 359
AVG 関数
使い方, 566
等価な数式, 593

B

BCP フォーマット
ASE でのインポートとエクスポート, 770
BEGIN TRANSACTION 文
トランザクション管理の制限, 803
リモートデータアクセス, 803
BEGIN 文
バッチ, 107
BETWEEN キーワード
範囲クエリ, 300
BETWEEN 検索条件
範囲クエリ, 300
BINARY データ型
SQL Anywhere 実装, 637
BLOB
挿入, 623

C

CacheHits プロパティ
アクセスプラン内の統計, 358
[ノード統計] フィールドの説明, 355
CacheReadIndLeaf プロパティ
アクセスプラン内の統計, 358
CacheReadTable プロパティ
アクセスプラン内の統計, 358
CacheRead プロパティ
アクセスプラン内の統計, 358
[ノード統計] フィールドの説明, 355
CALL 文
制御文, 112
パラメータ, 111
プロシージャでの使用, 77
例, 87
CASCADE アクション
説明, 856
CASE 文
制御文, 112
CBSIZE 接続パラメータ
パフォーマンス, 255
cdata ディレクティブ
使用, 693
CHAR データ型
SQL Anywhere 実装, 637
CHECK 条件
Transact-SQL, 643
CLOSE 文
カーソル管理手順, 122
ClusteredHashGroupBy プラン項目
プランでの省略形, 342
CommBufferSize 接続パラメータ

パフォーマンス, 255
COMMENT 文
オートコミット, 861
COMMIT TRANSACTION 文
トランザクション管理の制限, 803
COMMIT 文
Ultra Light での使用, 618
参照整合性の検証, 898
トランザクション, 861
複合文, 114
プロシージャとトリガ, 138
リモートデータアクセス, 803
COMPUTE 句
CREATE TABLE, 13
Transact-SQL SELECT 文のサポートされない構文, 656
CONNECTION_PROPERTY 関数
説明, 192
CONTAINS 探索条件
単語の削除, 441
conversion_error オプション
テキストインデックスへの影響, 402
COUNT 関数
NULL, 471
グループ分けされたデータに対する集合関数の適用, 314
説明, 470
COVAR_POP 関数
等価な数式, 593
COVAR_SAMP 関数
等価な数式, 593
CPUTime
[ノード統計] フィールドの説明, 355
create_date_time
ディレクトリアクセスサーバ, 781
CREATE DATABASE 文
Transact-SQL と互換性のあるデータベースの作成, 649
CREATE DEFAULT 文
サポートされない, 643
CREATE DOMAIN 文
Transact-SQL との互換性, 643
ドメインの使用, 848
CREATE EXISTING TABLE 文
使用, 792
ディレクトリアクセスサーバのプロキシテーブルの作成, 781
プロキシテーブルのロケーションの指定, 791
CREATE EXTERNLOGIN 文
使用, 789
ディレクトリアクセスサーバの外部ログインの作成, 781
CREATE FUNCTION 文
使用, 89
CREATE INDEX 文
スナップショットアイソレーションでは使用不可, 869
同時実行性, 932
CREATE LDAP SERVER 文
ユーザ認証に LDAP サーバを使用, 936
CREATE PROCEDURE 文
使用, 799
パラメータ, 110
例, 85
CREATE RULE 文
サポートされない, 643
CREATE SERVER 文
IBM DB2 のデータ型変換, 817
Microsoft SQL Server のデータ型変換, 823
ODBC と ASE のデータ型の変換, 815
Oracle Database のデータ型変換, 829
SAP HANA のデータ型変換, 819
SQL Anywhere データベースのマッピング, 810
ディレクトリアクセスサーバの作成, 781
リモートサーバの作成, 773
CREATE TABLE 文
Transact-SQL と互換性のあるテーブルの作成, 655
外部キー, 23
ディレクトリアクセスサーバのプロキシテーブルの作成, 781
同時実行性, 932
プライマリキー, 19
プロキシテーブル, 793
プロキシテーブルのロケーションの指定, 791
CREATE TEXT CONFIGURATION 文
使用, 396
CREATE TRIGGER 文
使用, 96
CREATE VIEW 文
WITH CHECK OPTION 句, 45
CROSS APPLY 句
説明, 514
例, 514
CSV ファイル
インポート, 716

CUBE 句
GROUPING SETS のショートカットとしての使用, 553
説明, 555
CUME_DIST 関数
使い方, 590
等価な数式, 593
CurrentCacheSize プロパティ
[オプティマイザ統計] フィールドの説明, 350

D

DataSet
XML をインポートするために使用, 675
リレーショナルデータを XML としてエクスポートするために使用, 669
date_format オプション
テキストインデックスへの影響, 402
テキスト設定オブジェクトの変更, 369
DATE フォーマット
テキストインデックス, 401
DB_PROPERTY 関数
説明, 192
DB2ODBC サーバクラス
説明, 817
dbinit ユーティリティ
Transact-SQL と互換性のあるデータベースの作成, 649
dbisql ユーティリティ
データベースの再構築, 749
dbo ユーザ
Adaptive Server Enterprise, 644
dbunload ユーティリティ
再構築ツール, 749
使用, 744
データのエクスポート, 737
テーブルスキーマのエクスポート, 758
テーブルデータのエクスポート, 757
dbxtract ユーティリティ
データの抽出, 760
DB 領域
管理, 643
DDL
オートコミット, 861
スナップショットアイソレーショントランザクションでは使用できない文, 869
説明, 1
同時実行性, 932
Deadlock システムイベント
使用, 884
DECLARE 文
カーソル管理手順, 122
複合文, 114
プロシージャ, 126
default_char
テキスト設定オブジェクト, 395
デフォルトの CHAR テキスト設定オブジェクト, 403
default_nchar
テキスト設定オブジェクト, 395
デフォルトの NCHAR テキスト設定オブジェクト, 403
delayed_commits オプション
パフォーマンス向上のためのヒント, 239
DELETE 文
DELETE 文の参照整合性検査, 858
エラー, 858
使用, 630
ロック, 901
DELETE 文または UPDATE 文でのエラー
説明, 857
demo.db
スキーマ, 490
DENSE_RANK 関数
使い方, 587
等価な数式, 593
DerivedTable プラン項目
プランでの省略形, 342
DiskReadIndInt プロパティ
アクセスプラン内の統計, 358
DiskReadIndLeaf プロパティ
アクセスプラン内の統計, 358
DiskReadTable プロパティ
アクセスプラン内の統計, 358
DiskReadTime
[ノード統計] フィールドの説明, 355
DiskRead プロパティ
アクセスプラン内の統計, 358
[ノード統計] フィールドの説明, 355
DiskWriteTime
[ノード統計] フィールドの説明, 355
DiskWrite プロパティ
アクセスプラン内の統計, 358
[ノード統計] フィールドの説明, 355
DISK 文
サポートされない, 643

DistH プラン項目
プランでの省略形, 342
DISTINCT キーワード
集合関数, 470
DISTINCT 句
重複した結果の削除, 293
不要な DISTINCT の削除, 231
DISTINCT の削除
説明, 231
DistO プラン項目
プランでの省略形, 342
DML
クエリで使用, 296
権限, 617
説明, 617
DML から選択
使用, 296
DN
識別名、LDAP ユーザ認証, 935
DROP DATABASE 文
Adaptive Server Enterprise, 643
DROP EXTERNLOGIN 文
使用, 790
DROP INDEX 文
スナップショットアイソレーションでは使用不可, 869
DROP PROCEDURE 文
使用, 801, 802
DROP SERVER 文
ディレクトリアクセスサーバの削除, 787
リモートサーバの削除, 777
DROP TABLE 文
ディレクトリアクセスサーバのプロキシテーブルの削除, 787
DROP TRIGGER 文
使用, 101
DROP 文
オートコミット, 861
同時実行性, 932
DT プラン項目
プランでの省略形, 342
DUMP DATABASE 文
サポートされない, 643
DUMP TRANSACTION 文
サポートされない, 643

E

EAH プラン項目
プランでの省略形, 342
EAM プラン項目
プランでの省略形, 342
EH プラン項目
プランでの省略形, 342
element ディレクティブ
使用, 690
EM プラン項目
プランでの省略形, 342
EstCpuTime
アクセスプラン内の推定, 359
EstDiskReads
アクセスプラン内の推定, 359
EstDiskReadTime
アクセスプラン内の推定, 359
EstDiskWrites
アクセスプラン内の推定, 359
EstRowCount
アクセスプラン内の推定, 359
EstRunTime
アクセスプラン内の推定, 359
Excel
アーキテクチャの不一致, 733
Excel
SQL Anywhere データベースへのデータのインポート, 716
SQL Anywhere データベースへのデータのエクスポート, 733, 734
リモートデータアクセス, 827
Excel ファイル
外部事前フィルタと単語区切りライブラリのサポート, 442
EXCEPT 句
NULL, 486
Transact-SQL 互換性, 656
クエリの結合, 483
使用, 483
ルール, 484
Exchange プラン項目
プランでの省略形, 342
EXECUTE IMMEDIATE 文
プロシージャ, 135
EXISTS 演算子
説明, 608
EXPLICIT モード

cdata ディレクティブの使用, 693
element ディレクティブの使用, 690
hide ディレクティブの使用, 691
xml ディレクティブの使用, 692
クエリの記述, 687
構文, 685
使用, 685

F

FALSE 条件
NULL, 307
FASTFIRSTROW テーブルヒント
オプティマイザの目標の選択, 220
fetchtst
説明, 237
FETCH 文
カーソル管理手順, 122
file_name
ディレクトリアクセスサーバ, 781
Filter プラン項目
プランでの省略形, 342
FIRST_VALUE 関数
使い方, 566
例, 577
FirstRowRunTime
[ノード統計] フィールドの説明, 355
First-row 最適化ゴール
オプティマイザの目標の選択, 220
FIRST 句
使用, 480
FOR BROWSE 句
Transact-SQL SELECT 文のサポートされない構文, 656
FORCE NO OPTIMIZATION 句
クエリ処理のフェーズをスキップするための条件, 320
FORCE OPTIMIZATION 句
クエリ処理のフェーズをスキップするための条件, 320
FOR JSON AUTO
構文, 704
使用, 704
FOR JSON EXPLICIT
構文, 705
使用, 705
FOR JSON RAW
構文, 703
使用, 703
FOR JSON 句
説明, 703
FOR READ ONLY 句
無視, 657
FORWARD TO 文
ネイティブ文, 798
ネイティブ文のリモートサーバへの送信, 798
FOR XML AUTO
使用, 682
FOR XML EXPLICIT
cdata ディレクティブの使用, 693
element ディレクティブの使用, 690
hide ディレクティブの使用, 691
xml ディレクティブの使用, 692
構文, 685
使用, 685
FOR XML RAW
使用, 680
FOR XML 句
AUTO モードの使用, 682
BINARY データ型, 678
EXPLICIT モードの構文, 685
EXPLICIT モードの使用, 685
IMAGE データ型, 678
Interactive SQL での表示, 694
LONG BINARY データ型, 678
RAW モードの使用, 680
VARBINARY データ型, 678
クエリ結果を XML として取得, 677
使用法, 678
制限, 678
FOR 句
FOR XML AUTO の使用, 682
FOR XML EXPLICIT の使用, 685
FOR XML RAW の使用, 680
クエリ結果を XML として取得, 677
FOR 文
制御文, 112
FoxPro
リモートデータアクセス, 828
FROM 句
FROM 句内のストアドプロシージャ, 295
FROM 句内の派生テーブル, 295
概要, 294
独立性レベル, 864
FullCompare プロパティ
アクセスプラン内の統計, 358

FullOuterHashJoin プラン項目
プランでの省略形, 342

G

GENERIC 単語区切りアルゴリズム
テキスト設定オブジェクトの設定, 396
GENERIC テキストインデックス
全文検索のチュートリアル, 411
プレフィクス検索, 385
GLOBAL AUTOINCREMENT
シーケンスとの相違点, 929
デフォルト, 840
go
バッチ文デリミタ, 107
GRANT 文
Transact-SQL, 646
同時実行性, 932
GrByHClust プラン項目
プランでの省略形, 342
GrByHSets プラン項目
プランでの省略形, 342
GrByH プラン項目
プランでの省略形, 342
GrByOSets プラン項目
プランでの省略形, 342
GrByO プラン項目
プランでの省略形, 342
GrBySSets プラン項目
プランでの省略形, 342
GrByS プラン項目
プランでの省略形, 342
GROUP BY ALL 句
Transact-SQL SELECT 文のサポートされない構文, 656
GROUP BY 句
HAVING 句と WHERE 句との使用, 471
order by, 482
SQL/2008 標準, 475
SQL 標準準拠, 475
WHERE 句, 474
WHERE 句と HAVING 句との使用, 471
エラー, 315
拡張, 549
グループ分けされたデータに対する集合関数の適用, 314
実行, 471
集合関数, 474
パフォーマンス, 224
GROUP BY 句の概要
説明, 471
GROUP BY リスト
実行プラン内の項目, 363
GROUPING 関数
CUBE 句と使用 (OLAP), 555
NULL プレースホルダの検出, 557
ROLLUP 句と使用 (OLAP), 553
GUID
グローバルオートインクリメントとの比較, 841
生成, 928
デフォルトのカラム値, 841

H

HANA
SAP HANA へのリモートデータアクセス, 819
HANAODBC サーバクラス
説明, 819
HANA リモートデータアクセス
説明, 819
データ型変換, 819
HashAntisemijoin プラン項目
プランでの省略形, 342
HashDistinct プラン項目
プランでの省略形, 342
HashExceptAll プラン項目
プランでの省略形, 342
HashExcept プラン項目
プランでの省略形, 342
HashFilter プラン項目
プランでの省略形, 342
HashGroupBySets プラン項目
プランでの省略形, 342
HashGroupBy プラン項目
プランでの省略形, 342
HashIntersectAll プラン項目
プランでの省略形, 342
HashIntersect プラン項目
プランでの省略形, 342
HashJoin プラン項目
プランでの省略形, 342
HashSemijoin プラン項目
プランでの省略形, 342
HashTableScan プラン項目
プランでの省略形, 342

HAVING 句
GROUP BY 句との使用, 471, 477
WHERE 句, 316
サブクエリ, 601
集合関数を持つ場合と持たない場合, 477
データグループの選択, 477
パフォーマンス, 282
論理演算子, 478
HFP プラン項目
プランでの省略形, 342
HF プラン項目
プランでの省略形, 342
hide ディレクティブ
使用, 691
HOLDLOCK キーワード
Transact-SQL, 658
HTS プラン項目
プランでの省略形, 342
HTTP 関数
デバッグ, 947
HTTP サービス
デバッグ, 947

I

I/O
ビットマップのスキャン, 250
IAH プラン項目
プランでの省略形, 342
IAM プラン項目
プランでの省略形, 342
IBM DB2
IBM DB2 へのリモートデータアクセス, 817
SQL Anywhere への移行, 761
データ型変換, 817
IBM DB2 リモートデータアクセス
説明, 817
id
メタプロパティ名, 672
IDENTITY カラム
値の検索, 654
特殊な IDENTITY, 654
IF 文
制御文, 112
IGNORE NULLS 句
LAST_VALUE 関数での使用, 578
IH プラン項目
プランでの省略形, 342
IMMEDIATE REFRESH 句
説明, 409
IM プラン項目
プランでの省略形, 342
IndAdd プロパティ
アクセスプラン内の統計, 358
IndexOnlyScan プラン項目
プランでの省略形, 342
IndexScan プラン項目
プランでの省略形, 342
IndLookup プロパティ
アクセスプラン内の統計, 358
InList プラン項目
プランでの省略形, 342
INOUT パラメータ
定義, 110
INPUT 文
Excel, 716
使用, 716
テキストインデックス, 715
マテリアライズドビュー, 715
INSERT トリガ
INPUT 文による起動, 709
INSERT 文
INSERT 文の参照整合性検査, 857
SELECT, 619
指定カラムへのデータの挿入, 621
使用, 719
全カラムへのデータの挿入, 620
重複データ, 857
テキストインデックスに関する考慮事項, 719
データの追加, 619
データの変更, 630
マテリアライズドビューに関する考慮事項, 719
ロック, 897
INSTEAD OF トリガ
再帰, 105
説明, 105
ビューの更新に使用, 105
instest
説明, 237
Interactive SQL
インデックスコンサルタント, 157
クエリ結果のエクスポート, 731, 740, 741
グラフィカルなプランの表示, 341
終了, 862
スクリプトの実行, 767

スクリプトのロード, 769
スクリプトファイル, 766
データベースの再構築, 749
テーブルのインポート, 726
テーブルのデータの表示, 10
テーブルリストの表示, 489
トランザクションによる変更のグループ分け, 862
バッチ処理, 768
バッチモード, 768
複数の結果セットの表示, 119
文デリミタ, 138
リレーショナルデータを XML としてエクスポート, 668
INTERSECT 句
NULL, 486
Transact-SQL 互換性, 656
クエリの結合, 483
使用, 483
ルール, 484
INTO CLIENT FILE 句
クライアントコンピュータとのインポートとエクスポート, 747
INTO VARIABLE 句
クライアントコンピュータとのインポートとエクスポート, 748
INTO 句
使用, 116
Invocations
アクセスプラン内の統計, 358
[ノード統計] フィールドの説明, 355
IN キーワード
マッチングリスト, 301
IN 条件
サブクエリ, 604
IN パラメータ
定義, 110
IN プラン項目
プランでの省略形, 342
IN リスト
最適化, 231
実行プラン内の項目, 364
IO プラン項目
プランでの省略形, 342
IQODBC サーバクラス
説明, 820
ISNULL 関数
説明, 307
IS NULL キーワード
説明, 307
isolation_level オプション
[オプティマイザ統計] フィールドの説明, 350
ISOLATION 句
Transact-SQL SELECT 文のサポートされない構文, 656
ISO SQL 標準
典型的な矛盾, 875
同時実行性, 875
ISYSFKEY
システムテーブルの使用法, 36
ISYSIDX
インデックスの共有, 36
システムテーブルの使用法, 36
ISYSIDXCOL
システムテーブルの使用法, 36
ISYSPHYSIDX
インデックスの共有, 36
システムテーブルの使用法, 36

J

JDBC
マテリアライズドビューの候補, 228
JHAP プラン項目
プランでの省略形, 342
JHA プラン項目
プランでの省略形, 342
JHFO プラン項目
プランでの省略形, 342
JHO プラン項目
プランでの省略形, 342
JHPO プラン項目
プランでの省略形, 342
JHRO プラン項目
プランでの省略形, 342
JHR プラン項目
プランでの省略形, 342
JHSP プラン項目
プランでの省略形, 342
JHS プラン項目
プランでの省略形, 342
JH プラン項目
プランでの省略形, 342
JMFO プラン項目
プランでの省略形, 342
JMO プラン項目

プランでの省略形, 342
JM プラン項目
プランでの省略形, 342
JNLA プラン項目
プランでの省略形, 342
JNLFO プラン項目
プランでの省略形, 342
JNLO プラン項目
プランでの省略形, 342
JNLS プラン項目
プランでの省略形, 342
JNL プラン項目
プランでの省略形, 342
JSON フォーマット
クエリ結果の取得, 703
説明, 703

L

LAST_VALUE 関数
使い方, 566
例, 577
LDAP
LDAP サーバ設定オブジェクトのアクティブ化, 943
LDAP サーバ設定オブジェクトの中断, 943
ユーザ認証, 935
LDAP サーバ設定オブジェクト
検証, 945
削除, 945
作成, 940
変更, 944
LDAP ユーザ認証
SQL を使用して設定, 936
Sybase Central での LDAP サーバ設定の検証, 945
Sybase Central での LDAP サーバ設定の削除, 945
Sybase Central での LDAP サーバ設定の作成, 940
Sybase Central でのユーザの作成, 942
Sybase Central でのログインポリシーの作成, 941
Sybase Central による LDAP サーバ設定の変更, 943, 944
Sybase Central を使用して設定, 939
サーバ設定オブジェクト, 935
説明, 935
LEAVE 文
制御文, 112
LeftOuterHashJoin プラン項目
プランでの省略形, 342
LIKE 探索条件
概要, 301
ワイルドカード, 303
LIMIT 句
使用, 480
LOAD DATABASE 文
サポートされない, 643
LOAD TABLE 文
BCP フォーマットデータのインポート, 770
考慮事項, 717
使用, 727
テキストインデックスに関する考慮事項, 718
マテリアライズドビューに関する考慮事項, 717
LOAD TRANSACTION 文
サポートされない, 643
localname
メタプロパティ名, 672
locked tables
アクセスプラン内の項目, 361
login_mode オプション
LDAP ユーザ認証の有効化, 936
LONG VARCHAR データ型
XML の格納, 667
LOOP 文
制御文, 112
プロシージャ, 122
Lotus Notes
リモートデータアクセス, 828

M

Mac OS X
Mac OS X 上の Ultra Light でサポートされないリモートサーバ, 812
MANUAL REFRESH 句
説明, 410
master データベース
サポートされない, 642
materialized_view_optimization オプション
使用, 75
max_query_tasks オプション
[オプティマイザ統計] フィールドの説明, 350
クエリ内並列処理の制御, 364

MAXIMUM TERM LENGTH 設定
N-gram の推奨サイズ, 399
定義, 399
MAX 関数
使い方, 566
等価な数式, 593
MergeExceptAll プラン項目
プランでの省略形, 342
MergeExcept プラン項目
プランでの省略形, 342
MergeIntersectAll プラン項目
プランでの省略形, 342
MergeIntersect プラン項目
プランでの省略形, 342
MergeJoin プラン項目
プランでの省略形, 342
MERGE 文
RAISERROR アクションの使用, 724
使用, 720
テキストインデックスに関する考慮事項, 722
マテリアライズドビューに関する考慮事項, 721
MESSAGE 文
プロシージャ, 126
Microsoft Access
SQL Anywhere への移行, 761
リモートデータアクセス, 821
Microsoft Excel
SQL Anywhere データベースへのデータのインポート, 716
SQL Anywhere データベースへのデータのエクスポート, 733, 734
アーキテクチャの不一致, 733
リモートデータアクセス, 827
Microsoft FoxPro
リモートデータアクセス, 828
Microsoft SQL Server
SQL Anywhere への移行, 761
MINIMUM TERM LENGTH 設定
定義, 398
MIN 関数
等価な数式, 593
MIRROR サーバクラス
説明, 811
Mobile Link
データベースの再構築, 754, 755
modified_date_time
ディレクトリアクセスサーバ, 781
MSACCESSODBC サーバクラス
説明, 821
MSSODBC サーバクラス
説明, 823
MultIdx プラン項目
プランでの省略形, 342
MultipleIndexScan プラン項目
プランでの省略形, 342
MySQL
ODBC サーバクラス, 825
MYSQLODBC サーバクラス
説明, 825

N

NCHAR データ型
SQL Anywhere 実装, 637
NestedLoopsAntisemijoin プラン項目
プランでの省略形, 342
NestedLoopsJoin プラン項目
プランでの省略形, 342
NestedLoopsSemijoin プラン項目
プランでの省略形, 342
NEWID 関数
使用する場合, 928
デフォルトのカラム値, 841
N-gram
N-gram の生成方法, 402
推奨サイズ, 399
生成のための 2 段階の処理, 402
単語の分割方法を理解する, 396
定義, 396
NGRAM テキストインデックス
あいまい全文検索のチュートリアル, 423
非あいまい全文検索のチュートリアル, 427
プレフィクス検索, 385
NOHOLDLOCK キーワード
無視, 658
NOT
論理演算子の使用, 308
NOT BETWEEN キーワード
範囲クエリ, 300
Notes とリモートデータアクセス
説明, 828
NOT NULL 制約
データの整合性を維持するためのツール, 835
NOT キーワード
例, 300

NULL
0 やブランクと異なる, 305
DISTINCT 句を使用して重複した NULL 値を除外する, 294
EXCEPT 句, 486
INTERSECT 句, 486
OLAP のプレースホルダ, 557
Transact-SQL との互換性, 655
UNION 句, 486
カラム定義, 308
カラムのデフォルト, 650
集合関数, 471
出力, 742
セット演算と NULL, 486
ソート順, 480
デフォルト, 842
デフォルトパラメータ, 306
比較, 307
比較に使用した結果は UNKNOWN, 306
不定の値と WHERE 句, 305
プロパティ, 307
NULL 値
Transact-SQL 外部ジョイン, 507
挿入, 620
データのインポート, 728
変換エラーの無視, 726
NULL 入力テーブル
外部ジョイン, 501
NULL の出力
説明, 742
NULL のプロパティ
説明, 307
NULL への値の代入
説明, 307

O

ODBC
アプリケーションとロック, 878
外部サーバ, 809
独立性レベルの設定, 878
マテリアライズドビューの候補, 228
odbcfet
説明, 237
ODBC サーバクラス
Adaptive Server Enterprise, 814
Advantage Database Server, 812
IBM DB2, 817
IQODBC, 820
Lotus Notes SQL, 828
Microsoft Access, 821
Microsoft Excel, 827
Microsoft FoxPro, 828
MySQL, 825
Oracle Database, 829
SAP HANA, 819
SQL Anywhere, 811
SQL Server, 823
Ultra Light, 812
サーバクラス ODBC, 827
説明, 809
OLAP
CUBE 句, 555
GROUP BY 句の拡張, 549
OLAP パフォーマンスの向上, 548
ROLLUP 句, 553
Window 関数, 565
Window 集合関数, 565
Window ランキング関数, 584
WITH CUBE 句, 556
WITH ROLLUP 句, 554
概要, 547
基本集合関数, 566
説明, 547
線形回帰関数, 583
相関関数, 583
標準偏差関数, 579
平方偏差関数, 579
ロー番号付け関数, 592
OLAP 関数
式, 593
ON EXCEPTION RESUME 句
Transact-SQL, 666
エラー処理, 126
ストアドプロシージャ, 125
例外処理, 130
ON 句
概要, 496
ジョイン, 496
テーブルの参照, 496
OpenString プラン項目
プランでの省略形, 342
OPENXML 演算子
xp_read_file での使用, 673
使用, 669
OPEN 文

カーソル管理手順, 122
optimization_goal オプション
[オプティマイザ統計] フィールドの説明, 350
optimization_level オプション
[オプティマイザ統計] フィールドの説明, 350
optimization_workload オプション
[オプティマイザ統計] フィールドの説明, 350
使用, 224
OR
論理演算子の使用, 308
Oracle
SQL Anywhere への移行, 761
Oracle Database
データ型変換, 829
Oracle Database とリモートデータアクセス
説明, 829
ORAODBC サーバクラス
説明, 829
Order-by
実行プラン内の項目, 364
ORDER BY 句
GROUP BY, 482
クエリ結果のソート, 478
結果の制限, 480
通常のビュー定義の制限, 44
パフォーマンス, 224
パフォーマンス改善のためのインデックス使用, 312
複合インデックス, 29
部分的に定義されたウィンドウへの影響 (OLAP), 561
マテリアライズドビュー定義内, 59
例, 310
ローを常に同じ順序で表示するために必要, 312
ORDER BY と GROUP BY
説明, 482
OrderedDistinct プラン項目
プランでの省略形, 342
OrderedGroupBySets プラン項目
プランでの省略形, 342
OrderedGroupBy プラン項目
プランでの省略形, 342
OUTER APPLY 句
説明, 514
例, 514
OUTPUT 文
Excel, 733, 734
クエリ結果のエクスポート, 731, 740, 741
使用, 733
データを XML としてエクスポートするために使用, 668
OUT パラメータ
定義, 110
OVER 句
Window 関数として使用される関数での使用法, 563

P

ParallelHashAntisemijoin プラン項目
プランでの省略形, 342
ParallelHashFilter プラン項目
プランでの省略形, 342
ParallelHashSemijoin プラン項目
プランでの省略形, 342
ParallelIndexScan プラン項目
プランでの省略形, 342
ParallelLeftOuterHashJoin プラン項目
プランでの省略形, 342
ParallelTableScan プラン項目
プランでの省略形, 342
PARTITION キーワード
Transact-SQL SELECT 文のサポートされない構文, 656
PCTFREE 設定
テーブルの断片化の削減, 241
PC プラン項目
プランでの省略形, 342
PDF ファイル
外部事前フィルタと単語区切りライブラリのサポート, 442
PERCENT_RANK 関数
使い方, 590
等価な数式, 593
PercentTotalCost
[ノード統計] フィールドの説明, 355
PerformanceFetch
説明, 237
PerformanceInsert
説明, 237
PerformanceTraceTime
説明, 237
PerformanceTransaction
説明, 237
PLAN 句

Transact-SQL SELECT 文のサポートされない構文, 656
PreFilter プラン項目
プランでの省略形, 342
PREPARE TRANSACTION 文
リモートデータアクセス, 803
PREPARE 文
リモートデータアクセス, 803
ProcCall プラン項目
プランでの省略形, 342
PROPERTY 関数
説明, 192

Q

QueryMemActiveEst プロパティ
[オプティマイザ統計] フィールドの説明, 350
QueryMemActiveMax プロパティ
[オプティマイザ統計] フィールドの説明, 350
QueryMemLikelyGrant
[オプティマイザ統計] フィールドの説明, 350
QueryMemMaxUseful
[ノード統計] フィールドの説明, 355
QueryMemMaxUseful プロパティ
[オプティマイザ統計] フィールドの説明, 350
QueryMemNeedsGrant
[オプティマイザ統計] フィールドの説明, 350
QueryMemPages プロパティ
[オプティマイザ統計] フィールドの説明, 350
quoted_identifier オプション
Transact-SQL 互換性の設定, 650
説明, 305

R

RAISERROR アクション
マージ操作での使用, 724
RAISERROR 文
ON EXCEPTION RESUME の使用, 666
Transact-SQL, 665
RANGE 句
ウィンドウが部分的に定義されている場合のデフォルト, 561
使用, 561
RANK 関数
使い方, 585
等価な数式, 593
RAW モード
使用, 680
READ_CLIENT_FILE 関数
クライアントコンピュータとのインポートとエクスポート, 747
READCLIENTFILE システム権限
クライアントコンピュータからのインポート, 747
readonly-statement-snapshot 独立性レベル
SELECT 文のロック, 897
使用, 903
説明, 864
ReadPK ロック
競合, 894
READ 文
SQL スクリプトファイルの実行, 768
ReceivingTracingFrom
トレーシングの設定, 164
RecursiveHashJoin プラン項目
プランでの省略形, 342
RecursiveLeftOuterHashJoin プラン項目
プランでの省略形, 342
RecursiveTable プラン項目
プランでの省略形, 342
RecursiveUnion プラン項目
プランでの省略形, 342
REFRESH MATERIALIZED VIEW 文
スナップショットアイソレーションでは使用不可, 869
REGR_AVGX 関数
等価な数式, 593
REGR_AVGY 関数
等価な数式, 593
REGR_COUNT 関数
等価な数式, 593
REGR_INTERCEPT 関数
等価な数式, 593
REGR_R2 関数
等価な数式, 593
REGR_SLOPE 関数
等価な数式, 593
REGR_SXX 関数
等価な数式, 593
REGR_SXY 関数
等価な数式, 593
REGR_SYY 関数
等価な数式, 593
reload.sql
説明, 750
データベースの再構築, 750

データベースの再ロード, 758
テーブルデータのエクスポート, 757, 758
テーブルのエクスポート, 745
リモートデータベースの再構築, 749
REORGANIZE TABLE 文
スナップショットアイソレーションでは使用不可, 869
RESIGNAL 文
使用, 130
RESTRICT アクション
説明, 856
RETURN 文
使用, 115
REVOKE 文
Transact-SQL, 646
同時実行性, 932
RI 制約
説明, 834
データの整合性を維持するためのツール, 834
RL プラン項目
プランでの省略形, 342
ROLLBACK 文
Ultra Light での使用, 618
トランザクション, 861
トリガ, 661
複合文, 114
プロシージャとトリガ, 138
ROLLUP 演算
GROUP BY 句の概要, 472
ROLLUP 句
GROUPING SETS のショートカットとしての使用, 553
説明, 553
ROW_NUMBER 関数
使い方, 592
RowConstructor プラン項目
プランでの省略形, 342
ROWID
例, 888
RowIdScan プラン項目
プランでの省略形, 342
ROWID プラン項目
プランでの省略形, 342
RowLimit プラン項目
プランでの省略形, 342
RowReplicate プラン項目
プランでの省略形, 342
RowsReturned
アクセスプラン内の統計, 358
[ノード統計] フィールドの説明, 355
ROWS 句
ウィンドウが部分的に定義されている場合のデフォルト, 561
使用, 561
ROWS プラン項目
プランでの省略形, 342
RR プラン項目
プランでの省略形, 342
RT プラン項目
プランでの省略形, 342
RunTime
アクセスプラン内の統計, 358
[ノード統計] フィールドの説明, 355
RU プラン項目
プランでの省略形, 342

S

sa_ansi_standard_packages システムプロシージャ
SQL Flagger の使用, 634
sa_audit_string システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_clean_database システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_column_stats システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_conn_activity システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_conn_compression_info システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_conn_info システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_conn_list システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_conn_options システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_conn_properties システムプロシージャ

データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_db_info システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_db_list システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_db_properties システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
SA_DEBUG ロール
デバッガ, 947
sa_dependent_views システムプロシージャ
使用, 43
sa_disable_auditing_type システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_disk_free_space システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_enable_auditing_type システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_external_library_unload システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_flush_cache システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_flush_statistics システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_get_histogram システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_get_request_profile システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_get_request_times システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_get_table_definition システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_index_density システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_index_levels システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_install_feature システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_java_loaded_classes システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_load_cost_model システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_locks システムプロシージャ
使用, 888
sa_make_object システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_materialized_view_can_be_immediate システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_migrate_create_fks システムプロシージャ
使用, 764
sa_migrate_create_remote_fks_list システムプロシージャ
使用, 764
sa_migrate_create_remote_table_list システムプロシージャ
使用, 764
sa_migrate_create_tables システムプロシージャ
使用, 764
sa_migrate_data システムプロシージャ
使用, 764
sa_migrate_drop_proxy_tables システムプロシージャ
使用, 764
sa_migrate システムプロシージャ
使用, 763
sa_procedure_profile_summary システムプロシージャ
概要プロファイリング情報の取得, 190
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_procedure_profile システムプロシージャ
詳細プロファイリング情報の取得, 190
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_recompile_views システムプロシージャ

データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_refresh_materialized_views システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_refresh_text_indexes システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_remove_tracing_data システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_report_deadlocks システムプロシージャ
使用, 884
sa_reset_identity システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_save_trace_data システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_send_udp システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_server_option システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
プロシージャプロファイリングのフィルタの設定, 188
プロシージャプロファイリングの無効化, 189
プロシージャプロファイリングの有効化, 188
プロシージャプロファイリングのリセット, 189
sa_set_tracing_level システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT
ODBC 独立性レベル, 878
SA_SQL_TXN_SNAPSHOT
ODBC 独立性レベル, 878
SA_SQL_TXN_STATEMENT_SNAPSHOT
ODBC 独立性レベル, 878
sa_table_fragmentation システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_table_page_usage システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_table_stats システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_text_index_vocab_nchar システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_unload_cost_model システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_user_defined_counter_add システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_user_defined_counter_set システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_validate システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sa_verify_password システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
SAODBC サーバクラス
説明, 811
saplan ファイル
説明, 333
SELECT 文
DML 文から選択する, 296
INSERT, 619
INTO 句, 116
Transact-SQL との互換性, 656
エイリアス, 288
エイリアスの SQL との互換性, 639
カラムの順序, 287
カラムの見出し, 288
カーソル, 122
キーとクエリアクセス, 249
サブクエリ, 594
使用, 281
通常のビューでの制限, 44
表示文字列, 290
変数, 658
文字データ, 305
ローの指定, 298
SELECT リスト
EXCEPT 文, 483
INTERSECT 文, 483
UNION 句, 483

UNION 文, 483
エイリアス, 288
エイリアスの SQL との互換性, 639
カラムの順序は結果の順序に影響する, 287
計算カラム, 291
実行プラン, 360
説明, 285
self_recursion オプション
Adaptive Server Enterprise, 661
SendingTracingTo
トレーシングの設定, 164
seq プラン項目
プランでの省略形, 342
SET DEFAULT アクション
説明, 856
SET HIDDEN 句
使用, 141
SET NULL アクション
説明, 856
SET OPTION 文
SQL Flagger で無視, 635
Transact-SQL, 650
SET 句
UPDATE 文, 626
SHARED キーワード
Transact-SQL SELECT 文のサポートされない構文, 656
SIGNAL 文
Transact-SQL, 665
プロシージャ, 126
SingleRowGroupBy プラン項目
プランでの省略形, 342
SnapshotIsolationState プロパティ
使用, 871
snapshot 独立性レベル
使用, 903
説明, 864
SOAP 関数
デバッグ, 947
SOAP サービス
デバッグ, 947
SortedGroupBySets プラン項目
プランでの省略形, 342
SortTopN プラン項目
プランでの省略形, 342
Sort プラン項目
プランでの省略形, 342
SOUNDEX 関数
説明, 310
sp_addgroup システムプロシージャ
Transact-SQL, 646
sp_addlogin システムプロシージャ
Transact-SQL, 646
サポート, 642
sp_adduser システムプロシージャ
Transact-SQL, 646
sp_bindefault プロシージャ
Transact-SQL, 644
sp_bindrule プロシージャ
Transact-SQL, 644
sp_changegroup システムプロシージャ
Transact-SQL, 646
sp_copy_directory システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sp_copy_file システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sp_create_directory システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sp_dboption システムプロシージャ
Transact-SQL, 650
sp_delete_directory システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sp_delete_file システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sp_dropgroup システムプロシージャ
Transact-SQL, 646
sp_droplogin システムプロシージャ
Transact-SQL, 646
sp_dropuser システムプロシージャ
Transact-SQL, 646
sp_forward_to_remote_server システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sp_get_last_synchronize_result システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sp_list_directory システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81

sp_move_directory システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sp_move_file システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sp_remote_columns システムプロシージャ
使用, 795
データベースの呼び出し側／定義者設定の影響を受ける, 81
sp_remote_exported_keys システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sp_remote_imported_keys システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sp_remote_primary_keys システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sp_remote_procedures システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
sp_remote_tables システムプロシージャ
使用, 779
データベースの呼び出し側／定義者設定の影響を受ける, 81
sp_servercaps システムプロシージャ
使用, 780
SQL
他の SQL ダイアレクトとの違い, 635
入力, 284
sql_flagger_error_level オプション
SQL Flagger の使用, 634
sql_flagger_warning_level オプション
SQL Flagger の使用, 634
SQL_TXN_ISOLATION
説明, 879
SQL_TXN_READ_COMMITTED
ODBC 独立性レベル, 878
SQL_TXN_READ_UNCOMMITTED
ODBC 独立性レベル, 878
SQL_TXN_REPEATABLE_READ
ODBC 独立性レベル, 878
SQL_TXT_SERIALIZABLE
ODBC 独立性レベル, 878
SQL/2003
SQL 文の準拠のテスト, 633
SQL/2008
SQL Anywhere の特殊な機能, 635
SQL/XML
説明, 676
SQL Anywhere
XML サポート, 667
他の SQL ダイアレクトとの違い, 635
SQLCA.lock
独立性レベル, 875
独立性レベルの選択, 878
SQLCODE 変数
概要, 125
SQL Flagger
Ultra Light との SQL 準拠のテスト, 633
起動, 634
説明, 633
標準と互換性, 635
SQLFLAGGER 関数
SQL Flagger の使用, 634
SQL Remote
データベースの再構築, 754, 755
リモートデータに対してサポートされない機能, 807
SQL SECURITY 句
使用法, 78
SQL Server
データ型変換, 823
リモートデータアクセス, 823
SQLSetConnectOption
説明, 879
SQLSTATE 変数
概要, 125
SQL クエリ
説明, 284
SQL スクリプト
Interactive SQL でのロード, 769
SQL スクリプトファイル
[SQL 文] ウィンドウ枠, 766
作成, 766
実行, 767
出力の書き込み, 769
説明, 766
SQL スクリプトファイルの実行
説明, 767
SQL スクリプトファイルを実行
ロードしないで, 767
SQL 標準
ANSI 以外のジョイン, 496
GROUP BY 句, 475

SQL Anywhere の特殊な機能, 635
SQL 文の準拠のテスト, 633
空間データ, 633
準拠, 633
説明, 635
SQL プリプロセッサユーティリティ (sqlpp)
SQL Flagger の使用, 634
SQL 文
Interactive SQL での実行, 767
互換性のある SQL 文の記述方法, 654
スクリプトファイルの作成, 766
スナップショットアイソレーショントランザクションでは使用不可, 869
SrtN プラン項目
プランでの省略形, 342
st_geometry_predefined_srs システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
st_geometry_predefined_uom システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
statement-snapshot 独立性レベル
SELECT 文のロック, 897
使用, 903
説明, 864
STDDEV_POP 関数
使い方, 580
等価な数式, 593
例, 580
STDDEV_SAMP 関数
使い方, 581
等価な数式, 593
例, 581
STDDEV 関数
等価な数式, 593
STOPLIST 設定
定義, 400
SUM 関数
使い方, 566
等価な数式, 593
Sybase Central
アプリケーションプロファイリング, 148
システムオブジェクトの内容の表示, 3
システムオブジェクトの表示, 3
データベースのアンロード, 737
テーブル制約, 845
テーブルの作成, 5
プロシージャの変換, 662
Sybase IQ
リモートデータアクセス, 820
SYSCOLSTAT
システムビュー、カラム統計の更新, 221
SYSCOLUMNS
Transact-SQL の名前の重複, 648
SYSINDEXES
Transact-SQL の名前の重複, 648
SYSSERVER
リモートサーバのシステムビュー, 773
SYSTEM PROCEDURE AS INVOKER 句
使用法, 78

T

TableScan プラン項目
プランでの省略形, 342
TCP/IP
パフォーマンス, 255
TERM BREAKER 設定
定義, 396
time_format オプション
テキストインデックスへの影響, 402
テキスト設定オブジェクトの変更, 369
timestamp_format オプション
テキストインデックスへの影響, 402
テキスト設定オブジェクトの変更, 369
timestamp_with_time_zone_format オプション
テキスト設定オブジェクトの変更, 369
TIMESTAMP データ型
Transact-SQL, 652
TIMESTAMP フォーマット
テキストインデックス, 401
TIME フォーマット
テキストインデックス, 401
TOP 句
使用, 480
TRACEBACK 関数
説明, 126
Transact-SQL
Adaptive Server Enterprise のエミュレート, 648, 649
IDENTITY カラム, 654
NULL, 655
NULL 値とジョイン, 507
Transact-SQL との互換性を意識したデータベースの設定, 647

Transact-SQL との互換性を維持するためのオプション設定, 650
Transact-SQL の特殊な TIMESTAMP カラムとデータ型, 652
Transact-SQL プロシージャから返される結果セット, 663
Transact-SQL プロシージャでのエラー処理, 664
WITH ROLLUP の使用, 554
移植可能な SQL の記述, 654
外部ジョイン, 505
外部ジョインとビュー, 507
外部ジョインの制限事項, 506
結果セット, 663
後続ブランク, 647
互換性のある SQL 文の記述方法, 654
互換性の概要, 640
サポートされないファイル操作文, 643
ジョイン, 659
ストアドプロシージャの概要, 660
データベースの作成, 648, 649
トリガ, 660
バッチ, 661
バッチの概要, 661
プロシージャ, 660
プロシージャ言語の概要, 660
プロシージャ内での RAISERROR 文の使用, 665
プロシージャの変換, 662
変数, 663
Transact-SQL 互換性
データベースオプションの設定, 650
Transact-SQL との互換性
SELECT 文, 656
データベース, 651
trantest
説明, 237
TRUNCATE TABLE 文
使用, 632
スナップショットアイソレーションでの使用, 869
trusted_certificates_file オプション
LDAP ユーザ認証の有効化, 936

U

UA プラン項目
プランでの省略形, 342
ULODBC サーバクラス
説明, 812
Ultra Light
SQL 文の準拠のテスト, 633
サーバクラス, 812
Ultra Light SQL
SQL Anywhere の文が Ultra Light SQL に準拠するかどうかのテスト, 633
UnionAll プラン項目
プランでの省略形, 342
UNION 句
NULL, 486
Transact-SQL 互換性, 656
クエリの結合, 483
ルール, 484
Union リスト
実行プラン内の項目, 363
UNIX
最小キャッシュサイズ, 215
最大キャッシュサイズ, 215
初期キャッシュサイズ, 215
UNKNOWN
NULL, 307
UNLOAD TABLE 文
使用, 735
UNLOAD 文
使用, 736
UPDATE の競合
スナップショットアイソレーション, 874
UPDATE 文
SQL Anywhere 実装, 637
エラー, 858
ジョインの更新, 627
使用, 625
制約違反, 629
トリガの起動, 627
例, 858
ロック, 899
user_estimates オプション
[オプティマイザ統計] フィールドの説明, 350
USING CLIENT FILE 句
クライアントコンピュータとのインポートとエクスポート, 747
USING VALUE 句
クライアントコンピュータとのインポートとエクスポート, 747
UUID

グローバルオートインクリメントとの比較, 841
生成, 928
デフォルトのカラム値, 841

V

VALIDATE LDAP SERVER 文
ユーザ認証に LDAP サーバを使用, 936
ValuePtr パラメータ
説明, 879
VAR_POP 関数
使い方, 582
等価な数式, 593
例, 582
VAR_SAMP 関数
使い方, 582
等価な数式, 593
例, 582
VARIANCE 関数
等価な数式, 593
VersionStorePages プロパティ
使用, 870

W

wait_for_commit オプション
使用, 899
制約違反がある UPDATE 文の処理, 629
Watcom SQL
互換性のある SQL 文の記述方法, 654
説明, 639
ダイアレクト, 640
プロシージャの変換, 662
WHERE 句
GROUP BY 句, 474
GROUP BY 句との使用, 471
HAVING 句, 316
HAVING との比較, 477
NULL 値, 307
サブクエリ, 600
ジョイン, 499
説明, 298
テーブル内のローの変更, 625
パターン一致, 301
パフォーマンス, 282
日付の比較の概要, 309
文字列比較, 302
WHERE 句のサブクエリのジョインへの変換
説明, 610
WHILE 文
制御文, 112
Windows
最小キャッシュサイズ, 215
最大キャッシュサイズ, 215
初期キャッシュサイズ, 215
Windows Mobile
キャッシュサイズとページサイズの考慮事項, 251
Windows パフォーマンスモニタ
起動, 195
説明, 195
複数コピーの実行, 195
Window 関数
集合関数のリスト, 565
説明, 558
ランキング関数のリスト, 584
ローの番号付け, 592
WINDOW 句
SELECT 文での使用, 559
インラインおよび WINDOW 句, 563
Window 集合関数
OLAP, 565
サポートされる関数のリスト, 565
説明, 565
Window プラン項目
プランでの省略形, 342
WITH CHECK OPTION 句
CREATE VIEW 文での使用, 45
WITH CUBE 句
説明, 556
WITH EXPRESS CHECK
パフォーマンス, 229
WITH HOLD カーソル
カーソル安定性, 902
WITH HOLD 句
カーソル安定性, 902
EXECUTE IMMEDIATE 句
WITH RESULT SET 句, 135
プロシージャでの EXECUTE IMMEDIATE の使用, 135
WITH ROLLUP 句
説明, 554
WITH 句
共通テーブル式, 532
Word ファイル

外部事前フィルタと単語区切りライブラリのサポート, 442
WRITE_CLIENT_FILE 関数
クライアントコンピュータとのインポートとエクスポート, 747
WRITE CLIENT FILE システム権限
クライアントコンピュータへのエクスポート, 747
WriteNoPK ロック
競合, 894

X

XML
DataSet オブジェクトを使用したインポート, 675
DataSet オブジェクトを使用してデータを XML としてエクスポート, 669
FOR XML AUTO の使用, 682
FOR XML EXPLICIT の使用, 685
FOR XML RAW の使用, 680
Interactive SQL からデータを XML としてエクスポート, 668
OPENXML 演算子を使用したインポート, 669
SQL Anywhere データベースでの使用, 667
エンコード, 667
クエリ結果を XML として取得, 677
説明, 667
定義, 667
デフォルトのネームスペース, 675
リレーショナルデータからクエリ結果を XML として取得, 676
リレーショナルデータとしてインポート, 669
リレーショナルデータベースにおける格納, 667
リレーショナルデータを XML としてエクスポート, 668
XMLAGG 関数
使用, 695
XMLCONCAT 関数
使用, 696
XMLELEMENT 関数
使用, 697
XMLFOREST 関数
使用, 699
XMLGEN 関数
使用, 700
xml ディレクティブ
using, 692
XML データ型
使用, 667
XML のインポート
DataSet オブジェクトを使用, 675
OPENXML 演算子を使用, 669
説明, 669
XML 文書
整形式, 694
xp_cmdshell システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
xp_getenv システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
xp_read_file システムプロシージャ
XML のインポート, 673
データベースの呼び出し側／定義者設定の影響を受ける, 81
xp_sendmail システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
xp_startmail システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
xp_startsmtp システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
xp_stopmail システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
xp_stopsmtp システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
xp_write_file システムプロシージャ
データベースの呼び出し側／定義者設定の影響を受ける, 81
XPath
使用, 669

あ

アイドルアクティブ／秒の統計値
説明, 198
アイドル書き込み／秒の統計値
説明, 198
アイドルチェックポイント時間の統計値
説明, 198

アイドルチェックポイント／秒の統計値
説明, 198
あいまい
説明, 392
全文検索のチュートリアル, 423
データベースサーバによる、あいまい検索の解釈方法, 402
アクション
CASCADE, 856
RESTRICT, 856
SET DEFAULT, 856
SET NULL, 856
アクセスプラン
説明, 321
統計の説明, 358
アスタリスク
SELECT 文, 286
全文検索のプレフィクス検索で使用, 383
値の格納
共通テーブル式, 537
圧縮
接続パケットに対する推奨内容, 254
データベースまたはログファイルの圧縮に対する警告, 254
アップグレード
データベースファイルフォーマット, 751
アトミックなトランザクション
説明, 861
アトミックな複合文
説明, 114
アプリケーションプロファイリング
CPU が制限要因であるかどうかの判断, 181
I/O 帯域幅が制限要因であるかどうかの判断, 181
インデックスコンサルタント, 156
運用データベース, 163
説明, 148
チュートリアル, 255
トレーシングセッションの作成, 176
トレーシングデータベース, 163
プロシージャプロファイリング, 150
メモリが制限要因であるかどうかの判断, 181
要求トレース分析, 182
アプリケーションプロファイリングウィザード
起動, 149
自動起動の有効化と無効化, 149
説明, 149
チュートリアル, 255
アプリケーションプロファイリングモード
使用, 148
アプリケーション論理のデバッグ
説明, 182
アポストロフィ
文字列, 305
アルファベット順
ORDER BY 句, 310
暗号化
Sybase Central を使用したマテリアライズドビューの暗号化, 68
オブジェクトを隠す, 141
キャッシュサイズ, 212
マテリアライズドビュー, 68
アンロード
説明, 749
アンロードツール
説明, 730
[データのアンロード] ウィンドウ, 739
データベースアンロードウィザード, 737
アンロードと再ロード
データベース, 759
同期に関連しないデータベース, 752
同期に関連するデータベース, 754, 755
アーキテクチャ
Adaptive Server Enterprise, 642

い

以下
WHERE 句での比較演算子, 299
意思決定支援
独立性レベル, 904
以上
WHERE 句での比較演算子, 299
移植可能な SQL
記述, 654
依存性
ビューの依存性, 41
一意性
インデックスを使用した強化, 39
一意性制約作成ウィザード
アクセス, 846
一意性制約
生成されたインデックス, 26
説明, 846
一意性制約の設定
説明, 846

位置付け更新
例, 124
位置ロック
期間, 887
説明, 892
挿入ロック, 893
幻ロック, 893
一貫性
ISO SQL 標準, 875
繰り返し可能読み出し, 875
繰り返し可能読み出しとロック, 895
繰り返し不可能読み出しのチュートリアル, 912
スナップショットアイソレーション, 897
正当性とスケジューリング, 903
説明, 861
ダーティリード, 875
ダーティリードとロック, 895
ダーティリードのチュートリアル, 907
直列化不可能なスケジュールの影響, 904
典型的なトランザクション, 904
独立性レベル, 864, 875
独立性レベル 0, 895
トランザクション中, 875
幻ロックのチュートリアル, 924
幻ロー, 875
幻ローとロック, 896
幻ローのチュートリアル, 917
ロックを使用して保証, 886
意図的テーブルロック
説明, 891
意図的ロック
競合, 894
スナップショットアイソレーション, 891
イベント
許可される文, 140
プロファイリング結果の生成と確認, 150
イメージ
挿入, 623
インデックス
HAVING 句のパフォーマンス, 282
SQL Anywhere 実装, 637
Sybase Central を使用した検証, 33
Sybase Central を使用した再構築, 33
Transact-SQL の互換性と名前, 652
WHERE 句のパフォーマンス, 282
インデックス間の相関関係, 161
インデックスコンサルタントの使用, 156
インデックスコンサルタントの推奨内容の解釈, 158
インデックスの操作, 26
カタログ, 35
カラムの順序, 28
共有される物理インデックスの特定, 37
クラスタリング, 30
計算カラム, 31
検索引数可能な述部, 282
構造, 38
候補, 159
最適化, 26
削除, 34
作成, 31
作成するインデックスの決定, 26
述部の分析, 282
述部を満たすために使用, 223
使用する状況, 26
スキュー, 245
制限事項と考慮事項, 26
説明, 26
断片化, 245
断片化のチュートリアル, 268
テキストインデックス, 408
テキストインデックスを使用したビューの問い合わせ, 392
テンポラリテーブル, 27
統計値のリスト, 204
パフォーマンスの向上, 224
費用対効果, 156
頻繁に検索するカラムでの使用, 26
ファンアウトとページサイズ, 252
複合, 28
物理, 36
ページサイズ, 252
未使用, 160
利点とロック, 905
リーフページ, 38
論理, 36
インデックス関数
ローの番号付け, 592
インデックスコンサルタント
MANAGE PROFILING システム権限が必要, 156
概要, 26
クエリに対する推奨内容の確認, 157
クエリのための使用, 157
結果の解釈, 159

結果の実装, 161
結果の評価, 161
サーバの状態, 161
推奨内容の解釈, 158
接続の状態, 161
説明, 156
データベースに対する推奨内容の確認, 157
データベースのための使用, 157
インデックス作成ウィザード
使用, 31
インデックスの完全比較／秒の統計値
説明, 204
インデックスの共有
説明, 36
インデックスの選択性
説明, 38
インデックスの断片化
説明, 245
チュートリアル, 268
インデックスの追加／秒の統計値
説明, 204
インデックスのルックアップ／秒の統計値
説明, 204
インデックスヒント
クラスタドインデックスの使用, 30
インデックスファンアウト
説明, 38
インデックス名
実行プラン内の項目, 362
インポート
ASE 互換性, 770
説明, 711
ツール, 711
テンポラリテーブルの使用, 10
インポートウィザード
使用, 726
説明, 712
インポート中の変換エラー
説明, 726
インポートツール
INPUT 文, 714
INSERT 文, 719
Interactive SQL のインポートウィザード, 712
Interactive SQL のエクスポートウィザード, 731
LOAD TABLE 文, 717
MERGE 文, 720
説明, 711
プロキシテーブル, 725
インポート用のテーブル構造
説明, 728
インメモリモード
パフォーマンス向上のためのヒント, 238
引用符
Adaptive Server Enterprise, 305
文字列, 305
インライン化
単純なストアドプロシージャ, 231
ユーザ定義関数, 231

う

ウィンドウ (OLAP)
ORDER BY 句のデフォルトへの影響, 561
RANGE 句を使用したサイズ変更, 561
ROWS 句を使用したサイズ変更, 561
SELECT 文の WINDOW 句, 559
インラインおよび WINDOW 句, 563
インラインのウィンドウの定義, 559
ウィンドウが部分的に定義されている場合のデフォルト, 561
句の評価順, 559
サイズ, 561
ウィンドウでのランキング関数
説明, 584
ウィンドウでのロー番号付け関数
Windows ランキング関数, 584
説明, 592
運用データベース
説明, 163

え

エイリアス
SQL Anywhere 実装, 639
計算カラム, 291
説明, 288
相関名, 294
エクスポート
ASE 互換性, 770
NULL, 742
NULL 値, 742
クエリ結果, 731, 740, 741
スキーマ, 758
説明, 730
テーブル, 745
リレーショナルデータを XML としてエクスポートする, 669

エクスポートウィザード
使用, 731
エクスポートツール
dbunload ユーティリティ, 737
OUTPUT 文, 733
UNLOAD TABLE 文, 735
UNLOAD 文, 736
説明, 730
エラー
Transact-SQL, 664
Watcom SQL を使用した Transact-SQL 動作の模倣, 666
プロシージャとトリガ, 125
変換, 726
エラー処理
ON EXCEPTION RESUME, 126
プロシージャとトリガ, 125
エンコード
XML, 667
演算子
NOT キーワード, 300
算術, 292
条件の接続, 308
優先度, 292
エンティティ
整合性の確保, 850
エンティティ整合性
UPDATE 文, 629
クライアントアプリケーションによる違反, 851
プライマリキー, 636
エントリポイント
外部事前フィルタ, 466
外部単語区切り, 467

お

大文字と小文字の区別
ASE 互換データベースの作成, 647
SQL, 285
Transact-SQL との互換性, 651
識別子, 652
ソート順, 480
データ, 651
データベース, 651
テーブル名, 285
ドメイン, 652
パスワード, 652
ユーザ ID, 652
リモートデータアクセス, 807
オブジェクト
修飾された名前, 1
定義を隠す, 141
ロック可能なオブジェクト, 887
オプション
blocking, 882
DEFAULTS, 728
isolation_level, 877
オプションの外部キー
説明, 852
オプティマイザ
クエリ処理のフェーズ, 317
述部の分析, 282
説明, 320
セマンティック変形, 327
選択性推定ソース, 324
バイパス, 319
パフォーマンス改善のための最適化のリスト, 327
マテリアライズドビューの使用, 69
オプティマイザの推定
説明, 322
オートインクリメント
デフォルト, 839
オートコミット
ALTER 文, 861
COMMENT 文, 861
DROP 文, 861
データ定義文, 861
トランザクション, 862
パフォーマンス, 238
オートメーション
ユニークキーの生成, 927
オーバーフローエラー
算術演算, 292
オーファンと参照整合性
説明, 898

か

改行
SQL, 284
開始
トランザクション, 861
解析ツリー
クエリ処理, 317

階層データ構造
階層データ構造の探索, 538
構成部品の問題, 542
外部キー
MATCH 句, 636
SQL を使用した作成, 23
SQL を使用した変更, 23
Sybase Central で作成, 22
Sybase Central で表示, 22
インデックス, 636
オーファン, 20
管理, 20
キージョイン, 520
参照循環性, 853
参照整合性, 855
整合性, 636
生成されたインデックス, 26
挿入, 857
ソート順序, 636
パフォーマンス, 249
必須／オプション, 852
複合外部キー, 20
役割名, 521
用語, 20
外部キー作成ウィザード
使用, 22
外部参照
HAVING 句, 602
説明, 597
定義, 597
外部サーバ
ODBC, 809
外部事前フィルタアルゴリズム
テキスト設定オブジェクトの設定, 401
外部ジョイン
Transact-SQL, 505
Transact-SQL との互換性, 659
Transact-SQL とビュー, 507
Transact-SQL の制限事項, 506
ジョイン条件, 502
ジョインの削除によるリライト最適化, 231
スタージョインの例, 511
制限, 506
説明, 501
ビューと派生テーブル, 504
複雑, 503
外部単語区切りライブラリ
テキスト設定オブジェクトの設定, 396
外部ログイン
削除, 790
作成, 789
説明, 789
リモートサーバ, 789
外部ログイン作成ウィザード
使用, 789
外部ロード
説明, 709
[概要] タブ
インデックスコンサルタントの結果, 159
カウンタ 1 統計情報
説明, 209
カウンタ 2 統計情報
説明, 209
カウンタ 3 統計情報
説明, 209
カウンタ 4 統計情報
説明, 209
カウンタ 5 統計情報
説明, 209
書き込みロック
競合, 894
説明, 890
書き込みを意図したテーブルロック
説明, 891
隠す
マテリアライズドビュー, 66
仮想インデックス
インデックスコンサルタント, 159
説明, 159
仮想メモリ
貴重なリソース, 212
カタログ
Adaptive Server Enterprise の互換性, 644
依存性情報の検索, 43
インデックス情報, 35
カッコ
UNION 演算子, 483
算術文内, 292
稼働条件
SQL Anywhere のデバッガ, 947
カラム
@ で始まる変数として定義された値, 836
GROUP BY 句, 471
IDENTITY, 654
SELECT 文, 287
SELECT リスト, 287

Sybase Central を使用した変更, 6
timestamp, 652
カラム制約の管理, 845
計算, 291
検査制約, 847
デフォルト, 836
デフォルト値の操作, 837
データ型とドメインの割り当て, 848
データの挿入, 620, 621
ドメインの使用, 849
幅の広い, 250
カラム一意性の強化
説明, 39
カラム検査制約作成ウィザード
アクセス, 845
カラム制約
一意性, 846
データの整合性を維持するためのツール, 834
カラム属性
AUTOINCREMENT, 928
NEWID, 928
デフォルト値の生成, 928
カラムデフォルト
@ で始まる変数として定義された値, 836
変更と削除, 837
カラム統計
更新, 221
説明, 322
カラム統計の更新
説明, 221
カラムの順序
結果は SELECT リストの順序を反映する, 287
テーブル, 250
複合インデックス, 28
環状ブロックの競合
説明, 883
関数
SOUNDEX 関数, 310
TRACEBACK, 126
TSEQUAL, 653
Window ランキング関数, 584
インライン化, 231
ウィンドウ, 565
キャッシュ, 327
プロファイリング結果の生成と確認, 150
べき等または決定性, 327
ユーザ定義、処理, 89
間接参照
データベースオブジェクト, 43
完全比較
説明, 38
カンマ
スタージョイン, 509
テーブル式のジョイン時, 524
テーブル式リスト, 500
管理者の役割
Adaptive Server Enterprise, 644
カーソル
LOOP 文, 122
SELECT 文, 122
SQL Anywhere 実装, 638
安定性, 877
ジョインでの更新, 495
接続制限, 253
不安定性, 877
プロシージャ, 122
プロシージャ、トリガ、ユーザ定義関数, 121
カーソル安定性
説明, 877
カーソル安定性ロック
WITH HOLD カーソル, 902
説明, 902
カーソルの統計値
説明, 208
カーソル不安定性
説明, 877
カーディナリティ
実行プラン内の項目, 362

き

期間
ロック, 887
記号
文字列比較, 302
基本集合関数
OLAP, 566
キャッシュ
UNIX, 217
暗号化データベースには大きいキャッシュが必要, 212
クエリの最適化をバイパスする文, 325
サイズのモニタリング, 218
サブクエリ, 326
実行プラン, 325
準備, 219

初期サイズ、最小サイズ、最大サイズ, 215
ストアドプロシージャの文, 325
動的サイズ決定, 216
パフォーマンス向上のためのキャッシュの使用, 212
文レベルのキャッシュ, 325
ユーザ定義関数, 327
読み込みヒット率, 336
キャッシュウォーミング
説明, 219
キャッシュサイズ
UNIX, 217
Windows, 216
Windows Mobile, 216
Windows Mobile の考慮事項, 251
初期サイズ、最小サイズ、最大サイズ, 215
パフォーマンスの考慮事項, 250
ページサイズ, 250
モニタリング, 218
キャッシュサイズ決定
パフォーマンス, 216
キャッシュサイズの現在の統計値
説明, 197
キャッシュサイズの最小の統計値
説明, 197
キャッシュサイズの最大の統計値
説明, 197
キャッシュサイズのピーク値の統計値
説明, 197
キャッシュサイズのモニタリング
説明, 218
キャッシュされたプラン
オプティマイザバイパス, 319
キャッシュ置換：合計ページ数／秒の統計値
説明, 205
キャッシュのスカベンジアクセスの統計値
説明, 205
キャッシュのスカベンジの統計値
説明, 205
キャッシュの統計値
リスト, 197
キャッシュのパニックの統計値
説明, 205
キャッシュヒット／秒の統計値
説明, 197
キャッシュページの空きの統計値
説明, 205
キャッシュページの固定の統計値
説明, 205
キャッシュページのファイルダーティの統計値
説明, 205
キャッシュページのファイルの統計値
説明, 205
キャッシュページの割り当て構造体の統計値
説明, 205
キャッシュ読み込みのインデックス内部／秒の統計値
説明, 197
キャッシュ読み込みのインデックスリーフ／秒の統計値
説明, 197
キャッシュ読み込みの合計ページ数／秒の統計値
説明, 197
キャッシュ読み込みのテーブル／秒の統計値
説明, 197
キャッシュ読み込みのワークテーブル
説明, 197
競合
環状ブロッキング, 883
スナップショットアイソレーション, 874
テーブルロック, 891
トランザクションブロック, 882
ロック, 893
競合するトリガ
実行順序, 104
共通テーブル式
一般的な使用例, 535
階層データ構造の探索, 538
構成部品の問題, 542
再帰, 538
再帰でのデータ型, 540
再帰に関する制限, 538
最短距離の問題, 545
使用できる条件, 534
説明, 532
定数セットの格納, 537
複数の集合レベル, 536
例, 532
共有テーブルロック
説明, 891
共有ロック
説明, 889
排他, 889
近接検索
全文検索, 386

キー
シーケンスを使用した生成, 928
パフォーマンス, 249
プライマリ, 17
キージョイン
2 つ以上の外部キー, 521
ON 句, 497
ON 句の使用, 521
SQL Anywhere 実装, 637
カンマを含まないテーブル式, 524
カンマを含まないテーブル式とリスト, 527
規則, 530
説明, 520
テーブル式, 524
テーブル式リスト, 525
ビューと派生テーブル, 529
キータイプ
実行プラン内の項目, 362
キー値
実行プラン内の項目, 362
キーワード
HOLDLOCK, 658
NOHOLDLOCK, 658
リモートサーバ, 807

く

句
COMPUTE, 656
FOR BROWSE, 656
FOR READ ONLY, 657
GROUP BY ALL, 656
INTO, 116
ISOLATION, 656
PLAN, 656
説明, 281
クエリ
SELECT 文, 281
Transact-SQL と互換性のあるクエリの記述, 656
エクスポート, 731, 740, 741
大文字と小文字の不要な変換の排除, 231
オプティマイザが適用できる最適化のリスト, 327
オプティマイザバイパス, 319
共通テーブル式, 532
最適化, 320
実行時間が長いクエリを診断するチュートリアル, 263
実行しないで最適化, 329
実行プラン, 328
集合操作, 483
処理時間が長い、クエリのパフォーマンスのモニタ, 237
処理時間が長い、パフォーマンスの問題のトラブルシューティング, 180
処理のフェーズ, 317
説明, 281
セマンティック変形, 327
定義されたバイパスクエリ, 319
ディレクトリアクセスプロキシテーブル, 785
テーブルからのデータの選択, 281
不要な内部ジョインと外部ジョインの削除, 231
並列処理, 366
クエリアルゴリズム
実行プランに使用される省略形, 342
クエリオプティマイザ
説明, 320
クエリ間並列処理
クエリ内並列処理とクエリ間並列処理, 364
説明, 364
クエリ結果
エクスポート, 731, 740, 741
クエリ結果のエクスポート
説明, 731, 740, 741
クエリ結果の編成
グループへ, 471
クエリ最適化
IN リスト述部, 231
クエリ実行
説明, 364
ビューマッチング, 225
並列処理, 364
クエリ上でブロックされるクエリ
リモートデータアクセス, 808
クエリ処理
フェーズ, 317
クエリ処理のフェーズ
説明, 317
クエリ処理のフェーズをスキップできるクエリ
説明, 319
クエリセマンティック変形フェーズ
クエリ処理, 317
クエリ内並列処理

クエリ内並列処理とクエリ間並列処理, 364
交換アルゴリズム, 364
説明, 364
クエリに関する一般的な問題
リモートデータアクセス, 807
クエリの解析
リモートデータアクセス, 804
クエリの最適化
オプティマイザバイパス, 319
実行プランの解釈, 328
説明, 320
フェーズ, 317
クエリの処理時間が長い
クエリのパフォーマンスのモニタ, 237
クエリの正規化
リモートデータアクセス, 804
クエリのパフォーマンス
実行プランの解釈, 328
クエリのプランキャッシュページの統計値
説明, 210
クエリの前処理
リモートデータアクセス, 804
クエリのメモリ不足時方式の統計値
説明, 210
クエリのローの実体化／秒の統計値
説明, 210
クエリパフォーマンス
RowsReturned 統計値, 335
キャッシュの読み込み数とヒット数, 336
述部の選択性, 335
推定ソース, 335
選択性統計値, 335
データの断片化の問題の識別, 336
有効なインデックスの不足, 336
クエリ変形
単純なストアドプロシージャのインライン化, 231
ユーザ定義関数のインライン化, 231
クエリメモリ
説明, 213
クライアント側データ
クライアントからロードされたデータの損失の防止, 749
クライアント側ロード
説明, 709
クライアントコンピュータ上のデータへのアクセス
説明, 747
クライアントでの文のキャッシュ
要求ロギングでの使用, 187
クライアントファイル
クライアントコンピュータとのインポートとエクスポート, 747
クラス
リモートサーバ, 808
クラスタドインデックス
インデックスコンサルタントの結果, 159
インデックスコンサルタントの結果の実装, 161
使用, 30
宣言, 31
グラフィカルなプラン
Interactive SQL で表示, 341
SQL 関数を使用したアクセス, 341
[オプティマイザ統計] フィールドの説明, 350
クエリを実行しないで表示, 329
コンテキスト別のヘルプ, 337
最適化のバイパス, 337
実行プランの解釈, 333
述部, 339
省略形, 342
説明, 333
統計情報, 335
[ノード統計] フィールドの説明, 355
ノードの詳細情報の表示, 337
バイパスクエリ, 337
ローカルフィールドまたはグローバルフィールドの説明, 350
グラフィカルなプランのカスタマイズ
説明, 333
グラフ表示
パフォーマンスモニタの使用, 193
繰り返し可能読み出し
ODBC 用の設定, 878
SELECT 文, 895
概要, 864
同時実行性の改善, 905
矛盾のケース, 875
繰り返し可能読み出し独立性レベル
説明, 864
繰り返し不可能読み出し
説明, 875
チュートリアル, 912
独立性レベル, 875
繰り返し不可能読み出しによる矛盾
説明, 895

グループ化
全文検索, 390
複数のカラムの使用, 473
グループ読み込み
テーブル, 250
グループ分けされたデータ
説明, 313
クロスジョイン
説明, 499
グローバルオートインクリメント
GUID および UUID との比較, 841
グローバルテンポラリテーブル
共有, 10
説明, 10
テーブル構造のマージ, 729
非共有, 10
グローバルテンポラリテーブル作成ウィザード
アクセス, 11
グローバル変数
デバッガ, 958

け

警告
プロシージャとトリガ, 128
計算カラム
インデックス, 31
計算カラム式の変更, 14
計算カラムの使用, 13
検索引数可能な関数を使用したクエリ, 283
再計算, 16
制限事項, 16
挿入と更新, 15
トリガ, 15
非計算カラムへの変換, 14
計算値
GROUP BY 句, 471
説明, 468
結果
インデックスコンサルタントの解釈, 159
結果セット
Transact-SQL, 663
トラブルシューティング, 312
ファイルへの保存, 769
複数, 119
複数回のクエリの実行, 312
プロシージャの結果, 118
プロシージャの複数の結果, 118
変数, 121
リモートプロシージャ, 799
ロー数の制限, 480
結果セットの実体化
クエリ処理, 230
結果の小計
CUBE 句, 555
ROLLUP 句, 553
WITH CUBE 句, 556
WITH ROLLUP 句, 554
決定性関数
定義, 327
副次的影響, 327
権限
デバッグ, 947
データ操作, 617
ユーザ定義関数, 93
現在アクティブの統計値
説明, 205
検索
全文検索, 367
中国語、日本語、韓国語 (CJK) データ, 367
検索引数可能な述部
説明, 282
検査制約
カラム, 843
削除, 847
データの整合性を維持するためのツール, 835
テーブル, 844
ドメイン, 845
変更, 847
検証
WITH EXPRESS CHECK を使用したテーブル, 229
XML, 693
インデックス, 33
限定比較テスト
サブクエリ, 603
説明, 612

こ

降順
ORDER BY 句, 478
更新
ジョインに基づく, 627
更新可能なビュー
説明, 45

[更新] タブ
インデックスコンサルタントの結果, 160
構成部品の問題
説明, 542
後続ブランク
Transact-SQL, 647
データベースの作成, 647
比較, 299
構築値
実行プラン内の項目, 363
構文に依存しない最適化
説明, 321
候補インデックス
インデックスコンサルタント, 159
説明, 159
効率
改善とロック, 905
データのインポート時の時間の節約, 711
互換性
Adaptive Server Enterprise と Transact SQL, 640
ANSI 以外のジョイン, 496
ASE でのインポートとエクスポート, 770
GROUP BY 句, 475
NULL の出力, 742
SQL Anywhere の Transact-SQL との互換性, 640
Transact-SQL, 640
Transact-SQL との互換性を意識したデータベースの設定, 647
Transact-SQL との互換性を維持するためのオプション設定, 650
Transact-SQL のジョイン, 659
大文字と小文字の区別, 651
互換性のある SQL 文の記述方法, 654
サーバとデータベース, 642
ストアドプロシージャの自動変換, 662
コスト
インデックスコンサルタントの結果, 160
コストの高いトリガを置き換える
パフォーマンス向上のためのヒント, 250
コストベースの最適化
説明, 321
バイパス, 319
コストモデル
説明, 321
コスト利益の合計
インデックスコンサルタントの結果, 160
コピー
INSERT を使用したデータ, 623
プロシージャ, 88
コマンドデリミタ
設定, 138
コミット
wait_for_commit, 899
コミットされた読み出し
ODBC 用の設定, 878
SELECT 文, 895
概要, 864
矛盾のケース, 875
コミットされた読み出し独立性レベル
説明, 864
コミットされない読み出し
ODBC 用の設定, 878
SELECT 文, 895
概要, 864
矛盾のケース, 875
コミットされない読み出し独立性レベル
説明, 864
コミット時の参照整合性の検査
説明, 898
コミットの遅延
パフォーマンス向上のためのヒント, 239
コメント
Sybase Central を使用したプロシージャの変更, 86
コンテキスト
全文検索の外部ライブラリ, 450

さ

再帰クエリ
制限, 538
再帰サブクエリ
構成部品の問題, 542
最短距離の問題, 545
説明, 538
データ型宣言, 540
複数の集合レベル, 536
再起動
シーケンス, 931
再計算
計算カラム, 16
再構築
インデックス, 33
ダウン時間の最短化, 759
ツール, 749

データベース, 749
目的, 750
再構築ツール
dbisql ユーティリティ, 752
dbunload ユーティリティ, 752
UNLOAD TABLE 文, 756
説明, 749
最高のパフォーマンスに関するヒント
リスト, 211
最小キャッシュサイズ
説明, 215
サイズ
ディレクトリアクセスサーバ, 781
最大
キャッシュサイズ, 215
最短距離の問題
説明, 545
最適化
オプティマイザが適用する変形のリスト, 327
コストベース, 321
実行プランの解釈, 328
説明, 320
最適化ゴール
実行プラン, 360
最適化時間
[オプティマイザ統計] フィールドの説明, 350
最適化のステップ
説明, 317
最適化のバイパス
バイパスクエリ, 319
最適化フェーズ
クエリ処理, 318
最適化方法
[オプティマイザ統計] フィールドの説明, 350
最適化前フェーズ
クエリ処理, 318
最適化をバイパスするクエリ
クエリ処理のフェーズをスキップするための条件, 319
説明, 319
再表示
手動ビュー, 63
テキストインデックス, 372, 409
テキストインデックスの再表示タイプの選択, 409
再表示タイプ
手動ビューと即時ビュー, 56
テキストインデックス, 409
テキストインデックスの変更, 374
マテリアライズドビューの変更, 71
削除
CONTAINS クエリからの単語, 441
外部ログイン, 790
カラムデフォルト, 837
検査制約, 847
シーケンス, 931
ディレクトリアクセスサーバ, 787
テキストインデックスからの単語, 441
データ型, 850
ドメイン, 850
トリガ, 101
プロシージャ, 88
ユーザ定義データ型, 850
リモートサーバ, 777
リモートプロシージャ, 801, 802
作成
SQL を使用したデータ型の作成, 848
SQL を使用したドメインの作成, 848
Sybase Central を使用したデータ型の作成, 848
Sybase Central を使用したドメインの作成, 848
Transact-SQL と互換性のあるテーブル, 655
外部トレーシングデータベース, 183
外部ログイン, 789
カラムデフォルト, 836
診断トレーシングセッション, 176
シーケンス, 930
ディレクトリアクセスサーバ, 781
テンポラリプロシージャ, 85
トリガ, 96
トリガ (Sybase Central の場合), 96
プロキシテーブル、Sybase Central を使用, 792
プロシージャ, 85
ユーザ定義関数, 89
リモートサーバ, 773
リモートプロシージャ, 800
サブクエリ
ALL テスト, 607
ANY 演算子, 606
ANY テスト, 605
GROUP BY, 601
HAVING 句, 601
IN キーワード, 301
WHERE 句, 600
WHERE 句とオプティマイザの動作, 610
演算子のタイプ, 603
外部参照, 602

概要, 594
キャッシュ, 326
限定比較テスト, 603
ジョイン, 599
ジョインとして書き換え, 610
ジョインへの変換, 610
セットメンバーシップテスト, 604
説明, 594
相関, 597
相関サブクエリ, 597
存在テスト, 608
単一ローのサブクエリ, 594
探索条件, 603
ネスト, 598
ネスト解除, 231
比較演算子, 611
比較テスト, 603
複数ローのサブクエリ, 594
分類, 594
ローグループ選択, 601
ロー選択, 600
サブクエリとジョイン
説明, 610
サブクエリのテスト
説明, 603
サブクエリのネスト解除
説明, 231
サブトランザクション
セーブポイント, 864
プロシージャとトリガ, 138
算術演算子
オーバーフローエラー, 292
クエリ結果の要約, 468
式と演算子の優先度, 292
参照
別のテーブルからの参照を表示, 22
参照先オブジェクト
説明, 41
参照整合性
INSERT 文の実行時に行われる検査, 857, 858
UPDATE 文, 629
アクション, 855
オーファン, 898
外部キー, 855
確保, 851
カラムデフォルト, 836
クライアントアプリケーションによる違反, 855
検査, 856
検査制約, 843
コミット時の検証, 898
システムテーブル内の情報, 859
システムトリガ, 855
制約, 835
説明, 833
喪失, 855
データの整合性を維持するためのツール, 834
プライマリキー, 636
参照整合性アクション
システムトリガで実装, 856
参照整合性検証の遅延
説明, 898
参照整合性制約
データの整合性を維持するためのツール, 834
参照整合性の確保
説明, 851
参照元オブジェクト
説明, 41
サンプルデータベース
demo.db のスキーマ, 490
サーバ
パフォーマンスモニタでのグラフ表示, 193
リモートサーバの使用, 772
サーバ側ロード
説明, 709
サーバクラス
Advantage Database Server, 812
ASEODBC, 814
DB2ODBC, 817
DIRECTORY, 781
HANAODBC, 819
IQODBC, 820
MIRROR, 811
MSACCESSODBC, 821
MSSODBC, 823
MySQL, 825
ODBC, 827
ODBC ベース, 809
ORAODBC, 829
SAODBC, 811
ULODBC, 812
説明, 773
定義, 772
サーバ設定オブジェクト
LDAP, 935
LDAP、検証, 945

LDAP、削除, 945
LDAP、作成, 940
LDAP、変更, 944
サーバとデータベース
互換性, 642
サーバの機能
リモートデータアクセス, 804
サーバの状態
インデックスコンサルタント, 161

し

時間
プロシージャとトリガ, 139
時間節約の方式
データのインポート, 711
式
NULL 値, 308
OLAP 集合関数, 593
適用式, 514
式 SQL
実行プラン内の項目, 364
識別子
大文字と小文字の区別, 652
修飾, 285
ユニーク, 652
識別名 (DN)
LDAP ユーザ認証, 935
時刻
SQL Anywhere 実装, 635
システムオブジェクト
Interactive SQL でのオブジェクトのリストの表示, 3
Interactive SQL でのシステムオブジェクトのリストの表示, 3
所有者ごとのシステムオブジェクトのリストのクエリ, 3
データベースのシステムオブジェクトの表示, 3
内容の表示, 3
システムカタログ
Adaptive Server Enterprise, 644
システム関数
TSEQUAL, 653
システム管理者
Adaptive Server Enterprise, 644
システム障害
トランザクション, 618
システムセキュリティ担当者
Adaptive Server Enterprise, 644
システムテーブル
Adaptive Server Enterprise, 644
Transact-SQL の名前の重複, 648
参照整合性についての情報, 859
所有者, 644
所有者ごとのシステムテーブルのリストのクエリ, 3
内容の表示, 3
システムトリガ
参照整合性アクションの実装, 856
参照整合性の実行, 855
プロファイリング結果の生成と確認, 150
システムビュー
インデックス, 35
参照整合性についての情報, 859
所有者ごとのビューテーブルのリストのクエリ, 3
システムプロシージャ
システムプロシージャを使用したプロシージャプロファイリング, 188
プロファイリング結果の生成と確認, 150
事前フィルタ
外部事前フィルタサンプル, 444
外部事前フィルタライブラリの論理フロー, 445
全文検索、外部事前フィルタライブラリの定義, 444
テキスト設定オブジェクトの設定, 401
事前フィルタライブラリ
外部ライブラリからのコールバック, 450
事前フィルタリング
テキスト設定オブジェクトの設定, 401
実行
Interactive SQL での SQL スクリプト, 767
SQL スクリプト, 766
トリガ, 99
複数回のクエリ, 312
実行時間が長いクエリ
チュートリアル, 263
実行フェーズ
クエリ処理, 318
実行プラン
解釈, 328
キャッシュ, 325
クエリを実行しないで表示, 329
グラフィカルなプラン, 333

コンテキスト別のヘルプ, 337
省略形, 342
長いテキストプラン, 330
ビューマッチングの結果, 360
短いテキストプラン, 329
実行プランに使用される省略形
説明, 342
実行プランの解釈
説明, 328
自動ジョイン
外部キー, 637
射影
説明, 287
集合関数
DISTINCT キーワード, 470
GROUP BY 句, 474
NULL, 471
OLAP, 566
OLAP に等価な式, 593
ORDER BY と GROUP BY, 482
ウィンドウ (OLAP), 565
概要, 313
グループ分けされたデータに対する適用, 314
スカラー集合, 469
説明, 468
データ型, 470
複数レベル, 536
集合操作
NULL, 486
説明, 483
ルール, 484
修飾
説明, 298
修飾された名前
データベースオブジェクト, 1, 285
修正
カラムデフォルト, 837
集約
実行プラン内の項目, 363
述部
IN リストの最適化, 231
オプティマイザ, 282
実行プランでの解釈, 339
実行プラン内の項目, 363
使用方法, 298
パフォーマンス, 282
述部の分析
説明, 282
出力リダイレクション
説明, 731, 740, 741
手動ビュー
Sybase Central を使用した作成, 61
再表示, 63
手動再表示タイプのマテリアライズドビュー, 56
手動ビューに変換する際の制限, 59
説明, 56
即時ビューへの変更, 71
古さ, 56
準備
キャッシュ, 219
準備文
接続制限, 253
ジョイン
2 つのテーブル, 494
3 つ以上のテーブル, 495
ANSI 以外のジョイン, 496
CROSS APPLY ジョインと OUTER APPLY ジョイン, 514
DELETE 文、UPDATE 文、INSERT 文, 495
NULL 入力テーブル, 501
ON 句, 496
Transact-SQL 外部ジョインと NULL 値, 507
Transact-SQL 外部ジョインとビュー, 507
Transact-SQL 外部ジョインの制限, 506
Transact-SQL との互換性, 659
WHERE 句, 499
外部, 501
カンマ, 500
カーソルの更新, 495
キー, 637
キージョイン, 520
クロスジョイン, 499
サブクエリ, 599
サブクエリからジョインへの変換, 610
サブクエリの変換, 610
自動, 637
ジョインしたテーブル, 493
ジョイン条件, 492
ジョインの削除によるリライト最適化, 231
スタージョイン, 509
説明, 488
セルフジョイン, 507
全外部ジョイン, 637
探索条件, 498
重複する相関名, 509

直積, 499
適用式から生成, 514
デフォルトは KEY JOIN, 493
データ型変換, 495
テーブル式, 495
等価ジョイン, 498
動作, 492
内部, 501
内部ジョインの計算方法, 494
内部と外部, 500
ナチュラル, 637
ナチュラルジョイン, 516
ネスト, 495
派生テーブル, 513
左外部ジョイン, 637
複数テーブルからのデータ検索, 488
複数のローカルデータベースのテーブルのジョイン, 797
保護されたテーブル, 501
右外部ジョイン, 637
リモートテーブルのジョイン, 796
ジョイン演算子
Transact-SQL, 659
ジョイン条件
種類, 498
説明, 492
ジョイン方式の区切りにコロンを使用
説明, 329
使用可能 IO の統計値
説明, 210
条件
GROUP BY 句, 316
パターン一致, 301
ブレークポイントの編集, 956
論理演算子との接続, 308
昇順
ORDER BY 句, 478
初期化
マテリアライズドビュー, 62
初期化ユーティリティ (dbinit)
Transact-SQL と互換性のあるデータベースの作成, 649
初期キャッシュサイズ
説明, 215
所有者
ディレクトリアクセスサーバ, 781
テーブルの変更, 6
処理時間が長いクエリ
パフォーマンスの問題のトラブルシューティング, 180
診断トレーシング
運用データベース, 163
外部トレーシングデータベースの作成, 183
情報の解釈, 179
設定, 164
説明, 162
トレーシングセッション中のトレーシングの設定の変更, 176
トレーシングセッションの作成, 176
トレーシングタイプ, 168
トレーシングデータベース, 163
トレーシングに関連するデータベースのプロパティ, 164
トレーシングの条件, 172
トレーシングのスコープ, 166
トレーシングの設定, 175
トレーシングの設定の確認, 173
トレーシングレベル, 164
診断トレーシング条件
説明, 172
診断トレーシングセッション
作成, 176
診断トレーシングタイプ
OPTIMIZATION_LOGGING, 168
OPTIMIZATION_LOGGING_WITH_PLANS, 168
診断トレーシングのスコープ
説明, 166
診断トレーシングの設定
説明, 164
診断トレーシングのタイプ
説明, 168
診断トレーシングのレベル
カスタマイズ, 167
説明, 164
診断トレーシングレベル
設定, 175
シーケンス
オートインクリメントとの相違点, 929
現在の値または次の値の取得, 928
再起動, 931
削除, 931
作成, 930
シーケンスの作成と使用の例, 929
説明, 928
変更, 931

ユニークキーの生成, 928
シーケンスジェネレータ
オートインクリメントとの相違点, 929
削除, 931
作成, 930
シーケンスを使用したユニークな値の生成, 928
説明, 928
変更, 931
シーケンスジェネレータの作成ウィザード
使用, 930

す

[推奨インデックス] タブ
インデックスコンサルタントの結果, 159
推定ソース
オプティマイザ選択性推定ソース, 324
スカラー集合
説明, 469
スカラー集合関数
定義, 474
スキーマ
エクスポート, 758
ロック, 888
スキーマロック
共有, 889
説明, 888
スクリプト
Interactive SQL で実行, 767
SQL スクリプトファイルの作成, 766
SQL スクリプトファイルのロード, 769
SQL ファイルの説明, 766
スクリプトファイル
SQL の実行, 767
スケジュール
直列化可能, 903
直列化可能性の影響, 903
直列化不可能なスケジュールの影響, 904
ロック期間, 902
スコア
全文検索, 393
スコープ
診断トレーシング, 166
スタージョイン
説明, 509
ステータス
通常のビュー, 46
ストアドプロシージャ
FROM 句内での使用, 295
Sybase Central を使用したストアドプロシージャの変換, 662
Transact-SQL ストアドプロシージャの概要, 660
共通テーブル式を含む, 536
デバッグのチュートリアル, 948
バッチとの比較, 107
プロファイリング結果の生成と確認, 150
文のキャッシュ, 325
ストアドプロシージャ言語
概要, 660
ストップリスト
使用の際の注意, 400
ストップリストの単語を検索する場合の動作, 441
説明, 396
全文検索, 396
スナップショットアイソレーション
SELECT 文のロック, 897
SQL Anywhere 実装, 639
意図的ロック, 891
更新の競合の回避, 874
説明, 868
トランザクション, 870
トランザクション内のレベルの変更, 880
パフォーマンス上の重要事項, 902
マテリアライズドビューマッチング, 869
有効化, 871
レベルの選択, 902
ローバージョン, 870
スナップショットアイソレーションの有効化
説明, 871
スナップショット数の統計値
説明, 208
スレッド
使用できるスレッドがない場合のデッドロック, 883
スレッド安全性
ユーザ定義関数, 89
スレッドデッドロック
説明, 883
スワップ領域
データベースキャッシュ, 217

せ

正規化
パフォーマンス向上, 246
制御文
バッチ内, 107
リスト, 112
制限
手動ビューから即時ビューへの変更, 59
説明, 287
リモートデータアクセス, 807
リモートデータアクセスの文字セット変換, 773
整合性
確保, 850
カラムデフォルト, 836
検査, 856
システムテーブル内の情報, 859
整合性制約の実装, 835
説明, 833
喪失, 855
データの整合性を維持するためのツール, 834
整合性検査
NOT NULL 制約, 835
RI 制約, 834
検査制約, 835
トリガ, 835
整合性検証
テーブルとカラム制約, 834
生成
ユニークキー, 927
生成されたジョイン条件
説明, 492
制約
Sybase Central, 845
一意性制約, 846
検査制約, 845
参照整合性, 843
整合性, 843
説明, 843
データ整合性, 843
セキュリティ
オブジェクトを隠す, 141
クライアントコンピュータとのインポートとエクスポート, 748
要求ログ, 186
セキュリティモデル
16.0 より前のバージョンのシステムプロシージャを呼び出し側または定義者として実行する, 79
定義者, 79
データベースが使用するセキュリティモデルの決定, 84
呼び出し側, 79
接続
デバッガ, 959
独立性レベルの決定, 881
マテリアライズドビューの候補, 228
リモート, 803
ループバック, 797
接続オプション
マテリアライズドビューに対する影響, 58
接続数の統計値
説明, 210
接続の削除
リモートデータアクセス, 808
接続の問題点
リモートデータアクセス, 807
接続ロック
期間, 887
設定
診断トレーシング, 164
診断トレーシングの設定, 175
診断トレーシングレベル, 175
セットメンバーシップテスト
=ANY, 605
説明, 614
否定, 605
セマンティック変形
リスト, 327
セミコロン
文デリミタ, 138
セルフジョイン
説明, 507
全外部ジョイン
SQL Anywhere 実装, 637
説明, 501
線形回帰関数
OLAP, 583
選択性
オプティマイザ推定ソース, 324
実行プランでの解釈, 339
実行プラン内の項目, 362
実行プランの解釈, 335
選択性推定

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01001

実行プランでの解釈, 339
部分インデックススキャンの使用, 39
選択性統計値
説明, 335
全比較
アクセスプラン内の統計, 358
全文検索
GENERIC チュートリアル, 411
Word、PDF、HTML などのファイルのインデックス付け, 442
あいまいのチュートリアル, 423
インデックスが作成されていない単語の検索, 441
外部事前フィルタライブラリの宣言, 444
外部単語区切りライブラリの宣言, 447
外部ライブラリからのコールバック, 450
禁止キーワードとワイルドカード, 377
近接検索, 386
検索結果のスコアの取得, 393
事前フィルタエントリポイント関数, 466
ストップリスト, 396
説明, 367
全文クエリの形成, 368
全文検索のタイプ, 377
単語区切りアルゴリズム, 396
単語区切りエントリポイント関数, 467
単語と式のグループ化, 377
単語とフレーズの検索, 377
単語の最小長, 396
単語の最大長, 396
中国語、日本語、韓国語 (CJK) データ, 367
テキストインデックス, 408
テキストインデックスの管理, 408
テキストインデックスの変更, 369
テキストインデックスのリスト, 375
テキストインデックスのリスト (SQL の場合), 376
テキストインデックスへのデータベースオプションの影響, 402
テキスト設定オブジェクト, 395
テキスト設定オブジェクトのリスト, 370
テキスト設定オブジェクトの例, 403
ドキュメント, 443
非あいまいのチュートリアル, 427
複数カラムに渡る検索, 391
複数カラムの検索, 377
プレフィクス検索, 383
フレーズ検索, 381
ブール検索, 388
文字列解釈の例, 405
全文検索のタイプ
説明, 377
セーブポイント
トランザクション内, 864
ネスト, 864
プロシージャとトリガ, 138
命名, 864
セーブポイントのネスト
説明, 864
セーブポイントの命名
説明, 864

そ

相関関数
OLAP, 583
相関サブクエリ
外部参照, 597
説明, 597
定義, 597
相関名
共通テーブル式での使用, 533
スタージョイン, 509
制限, 294
説明, 521
セルフジョイン, 507
テーブル名, 294
相対利益
インデックスコンサルタントの結果, 160
挿入
指定されていないカラムでの NULL 動作, 620
挿入ロック
説明, 893
総利益
インデックスコンサルタントの結果, 160
即時ビュー
再表示中に更新されたローのみを変更, 56
作成, 71
作成時の制限, 59
手動ビューへの変更, 71
説明, 56
即時再表示タイプのマテリアライズドビュー, 56
属性
SQLCA.lock, 878
クエリ結果を XML として取得, 677

Copyright © 2013, SAP AG or an SAP affiliate company.

測定値
実行プラン内の項目, 363
その他の統計値
リスト, 210
存在テスト
説明, 608
否定, 609
ソースコード
ブレークポイントの設定, 954
ソート
インデックスの使用, 224
クエリの結果, 310
ソート順
ORDER BY 句, 478
比較, 299

た

待機
参照整合性の検証, 898
待機中の要求数の統計値
説明, 205
タイミングユーティリティ
説明, 191
タイムスタンプ
SQL Anywhere 実装, 635
代理ロー
説明, 899
単一ローのサブクエリ
説明, 594
単語
全文検索を使用したデータベースの検索, 367
単語区切り
GENERIC 単語区切りアルゴリズム, 396
外部単語区切りサンプル, 444
外部単語区切りライブラリ, 396
外部単語区切りライブラリの論理フロー, 448
全文検索, 396
全文検索、外部単語区切りライブラリの定義, 447
単語区切りライブラリ
外部ライブラリからのコールバック, 450
単語、全文検索
説明, 377
単語長
全文検索, 396
テキストインデックスの単語長の設定, 396
単語とフレーズの検索
全文検索, 377
探索条件
GROUP BY 句, 316
NOT キーワードを使用した例, 300
サブクエリ, 594
使用方法, 298
パターン一致, 301
日付の比較, 309
単純なクエリ
説明, 319
断片化
インデックス, 245
インデックスのチュートリアル, 268
説明, 240
[断片化] タブ, 242
テーブル, 241
テーブルでの削減, 241
テーブルの断片化, 241
テーブルのチュートリアル, 271
ファイル, 240
ダーティリード
クエリ中のロック, 895
チュートリアル, 907
独立性レベル, 875
矛盾, 875

ち

チェックポイントの緊急度の統計値
説明, 198
チェックポイントの統計値
リスト, 198
チェックポイント／秒の統計値
説明, 198
チェックポイントフラッシュ／秒の統計値
説明, 198
チェックポイントログ
パフォーマンス, 247
チェックポイントログの書き込み／秒の統計値
説明, 198
チェックポイントログの使用ページ数の統計値
説明, 198
チェックポイントログのディスクへのコミット／秒の統計値
説明, 198
チェックポイントログの統計値
説明, 198

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01003

チェックポイントログのビットマップサイズの統計値
説明, 198
チェックポイントログのビットマップへの書き込み／秒の統計値
説明, 198
チェックポイントログのプレイメージ保存／秒の統計値
説明, 198
チェックポイントログのページ書き込み／秒の統計値
説明, 198
チェックポイントログのページ再配置の統計値
説明, 198
チェックポイントログの保存ページイメージ／秒の統計値
説明, 198
チェックポイントログのログサイズの統計値
説明, 198
逐次スキャン
ディスク割り付けとパフォーマンス, 624
逐次テーブルスキャン
ディスク割り付けとパフォーマンス, 624
注釈フェーズ
クエリ処理, 317
抽出
SQL Remote のデータベース, 760
チュートリアル
GENERIC 全文検索, 411
あいまい全文検索, 423
アプリケーションプロファイリング, 255
インデックスの断片化の診断, 268
繰り返し不可能読み出し, 912
速度が遅い文の診断, 263
ダーティリード, 907
デッドロック, 256
デバッガ, 948
テーブルの断片化の診断, 271
独立性レベル, 906
非あいまい全文検索, 427
プロシージャプロファイリングをベースラインとして使用, 274
幻ロック, 924
幻ロー, 917
長期間の読み込みロック
説明, 889
重複結果
削除, 293
重複するロー
UNION による削除, 483
直積
説明, 499
直接依存性
説明, 43
直接参照
データベースオブジェクト, 43
直列化可能
ODBC 用の設定, 878
SELECT 文, 895
概要, 864
スケジュール, 903
同時実行性の改善, 905
矛盾のケース, 875
直列化可能独立性レベル
説明, 864
直列化可能なスケジュール
影響, 903
説明, 903
ロックの解放, 902
直列化不可能なトランザクションのスケジューリング
影響, 903

つ

追加
データベースへのデータの追加, 711
追加利用可能の統計値
説明, 205
通常のビュー
SQL を使用した通常のビューの無効化, 51
SQL を使用した通常のビューの有効化, 51
Sybase Central を使用した削除, 49
Sybase Central を使用した通常のビューの作成, 47
Sybase Central を使用した通常のビューの無効化, 50
Sybase Central を使用した通常のビューの有効化, 50
Sybase Central を使用した変更, 48
説明, 44
ビュー内のデータのブラウズ, 53
マテリアライズドビューとテーブルとの簡単な比較, 40
通信の空きバッファの統計値
説明, 200

Copyright © 2013, SAP AG or an SAP affiliate company.

通信の失敗した送信／秒の統計値
説明, 200
通信の受信されるマルチパケット数／秒の統計値
説明, 200
通信の受信バイト数／秒の統計値
説明, 200
通信の受信パケット数／秒の統計値
説明, 200
通信の受信要求数の統計値
説明, 200
通信の送信されるマルチパケット数／秒の統計値
説明, 200
通信の送信バイト数／秒の統計値
説明, 200
通信の送信パケット数／秒の統計値
説明, 200
通信の総バッファ数の統計値
説明, 200
通信の統計値
リスト, 200
通信の無圧縮受信バイト数／秒の統計値
説明, 200
通信の無圧縮受信パケット数／秒の統計値
説明, 200
通信の無圧縮送信バイト数／秒の統計値
説明, 200
通信の無圧縮送信パケット数／秒の統計値
説明, 200
通信のユニークなクライアントアドレスの統計値
説明, 200
ツール
データのアンロード, 730
データのインポート, 711
データのエクスポート, 730
データの再ロード, 749
データベースの再構築, 749

て

定義者
16.0 より前のバージョンのシステムプロシージャを呼び出し側または定義者として実行する, 79
データベースが使用するセキュリティモデルの決定, 84
有効なユーザとログインユーザ, 78
呼び出し側の権限でプロシージャとファンクションを実行する, 78
定数式デフォルト
説明, 842
ディスク I/O の統計値
リスト, 202
ディスクアクセスのコストモデル
説明, 321
ディスク書き込みのアクティブの最大値の統計値
説明, 203
ディスク書き込みのアクティブの統計値
説明, 203
ディスク書き込みのコミットファイル／秒の統計値
説明, 203
ディスク書き込みのテンポラリ拡張／秒の統計値
説明, 203
ディスク書き込みのデータベース拡張／秒の統計値
説明, 203
ディスク書き込みの統計値
リスト, 203
ディスク書き込みのトランザクションログ／秒の統計値
説明, 203
ディスク書き込みのページ／秒の統計値
説明, 203
ディスクのアクティブ I/O の統計値
説明, 202
ディスクの最大 I/O の統計値
説明, 202
ディスク読み込みのアクティブの最大値の統計値
説明, 202
ディスク読み込みのアクティブの統計値
説明, 202
ディスク読み込みのインデックス内部／秒の統計値
説明, 202
ディスク読み込みのインデックスリーフ／秒の統計値
説明, 202
ディスク読み込みの合計ページ数／秒の統計値
説明, 202
ディスク読み込みのテーブル／秒の統計値

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01005

説明, 202
ディスク読み込みの統計値
リスト, 202
ディスク読み込みのワークテーブルの統計値
説明, 202
ディレクトリアクセスサーバ
結果セット, 781
削除, 787
作成, 781
説明, 781
デリミタ, 786
プロキシテーブルの削除, 787
プロキシテーブルの問い合わせ, 785
変更, 787
ディレクトリアクセスサーバの作成ウィザード
使用, 781
テキストインデックス
GENERIC 全文検索のチュートリアル, 411
あいまい全文検索のチュートリアル, 423
記憶領域が必要, 367
基本となるテキスト設定オブジェクトの設定, 396
再表示, 372
再表示タイプの選択, 409
再表示タイプの変更, 373
作成, 371
作成と再表示へのデータベースオプションの影響, 402
説明, 408
全文検索, 408
単語と設定の表示 (SQL の場合), 376
単語と設定の表示 (Sybase Central の場合), 375
テキスト設定オブジェクトを変更できない, 369
名前変更, 374
非あいまいのチュートリアル, 427
ビューの問い合わせ, 392
ビューまたはテンポラリテーブルで使用できない, 408
古さと再表示, 409
変更, 369, 374
テキストインデックス作成ウィザード
使用, 371
テキストインデックスの管理
説明, 409
テキスト設定オブジェクト
default_char 設定, 395
default_char と default_nchar の設定, 403
default_nchar 設定, 395
作成, 369
テキストインデックスで使用するかどうかの判断, 370
テキスト設定オブジェクトの設定の表示, 370
日付、時刻、タイムスタンプ形式の変更, 369
変更, 369
例, 403
テキスト設定オブジェクト作成ウィザード
使用, 369
定義する設定, 396
テキスト設定オブジェクトの例
全文検索, 403
テキストプラン
解釈, 329
適用
CROSS APPLY ジョインと OUTER APPLY ジョイン, 514
適用式
説明, 514
例, 514
デッドロック
Sybase Central での表示, 885
診断, 884
説明, 881
チュートリアル, 256
トランザクションのブロック, 883
理由, 883
レポート, 884
デッドロックレポート
説明, 884
デバイス
管理, 643
デバッガ
HTTP 関数, 947
SOAP 関数, 947
稼働条件, 947
機能, 947
接続, 959
説明, 947
チュートリアル, 948
ブレークポイント, 953
ブレークポイントの条件の編集, 956
ブレークポイントのステータスの変更, 955
ブレークポイントの設定, 954
変数の検査, 957
変数の表示, 957
呼び出しスタックの表示, 958

Copyright © 2013, SAP AG or an SAP affiliate company.

デバッグ
HTTP プロシージャ, 947
SQL Anywhere のデバッガの使用, 947
稼働条件, 947
権限, 947
説明, 947
チュートリアル, 948
デバッグモード
使用, 947
デフォルト
@ で始まる変数として定義された値, 836
GLOBAL AUTOINCREMENT, 840
INSERT 文, 620
NEWID, 841
NULL, 842
Sybase Central で作成, 837
Transact-SQL, 643
オートインクリメント, 839
カラム, 836
現在の日付と時刻, 838
作成, 836
定数式, 842
データ整合性のために使用する, 834
トランザクションとロック, 928
文字列と数値, 842
ユーザ ID, 839
デフラグメンテーション
説明, 240
データベース内の個々のテーブル, 241
データベース内のすべてのテーブル, 241
ハードディスク, 240
デリミタ
設定, 138
ディレクトリアクセスプロキシテーブルを問い合わせるときの一貫性, 786
テンポラリテーブル
Sybase Central に作成, 11
Transact-SQL との互換性, 656
インデックス, 27
クエリ処理中のワークテーブル, 230
作成, 5
説明, 10
テンポラリテーブルの操作, 10
データのインポート, 728
テーブル構造のマージ, 729
非トランザクション指向にする, 10
非トランザクション指向の利点, 10
プロシージャ内からの参照時の考慮事項, 12
ローカルとグローバル, 10
テンポラリテーブルページの統計
説明, 210
テンポラリファイル
ワークテーブル, 247
テンポラリプロシージャ
作成, 85
データ
SQL を使用したテーブルのデータの表示, 10
Sybase Central を使用したテーブルとデータの表示, 9
XML としてエクスポート, 668
一貫性, 875
インポート, 711
インポートとエクスポート, 709
エクスポート, 730
エクスポートウィザードを使用したエクスポート, 731
エクスポートツール, 730
大文字と小文字の区別, 651
検索, 26
手動ビューの再表示, 63
整合性と正当性, 903
追加、変更、削除, 617
データの修正に必要な権限, 617
マテリアライズドビューに設定, 62
無効, 833
データ一貫性
スナップショットアイソレーション, 897
独立性レベル 0, 895
データ型
EXCEPT 句, 483
INTERSECT 句, 483
SQL を使用した作成, 848
Sybase Central を使用した作成, 848
Transact-SQL の timestamp, 652
UNION 句, 483
カラムへの割り当て, 848
再帰サブクエリ, 540
削除, 850
集合関数, 470
ユーザ定義, 848
リモートプロシージャ, 799
データ型の変換
ODBC と ASE, 815
データ型変換
IBM DB2, 817
Microsoft SQL Server, 823

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01007

Oracle Database, 829
SAP HANA, 819
データキューブ
説明, 555
データ整合性
確保, 833, 850
カラムデフォルト, 836
検査, 856
システムテーブル内の情報, 859
制約, 835
説明, 833
喪失, 855
直列化不可能なスケジュールの影響, 904
データの整合性を維持するためのツール, 834
データ整合性の確保
説明, 833
データ操作文
説明, 617
データソース
外部サーバ, 809
[データ] タブ
SQL Anywhere 16 プラグイン, 9
データ入力
独立性レベル, 904
[データのアンロード] ウィンドウ
使用, 739
データの一貫性
ISO SQL 標準, 875
繰り返し可能読み出し, 875
繰り返し可能読み出しとロック, 895
繰り返し不可能読み出しのチュートリアル, 912
正当性, 903
ダーティリード, 875
ダーティリードとロック, 895
ダーティリードのチュートリアル, 907
幻ロックのチュートリアル, 924
幻ロー, 875
幻ローとロック, 896
幻ローのチュートリアル, 917
ロックを使用して保証, 886
データの移動
インポート, 711
エクスポート, 730
データのインポート
DEFAULTS オプション, 728
INPUT 文, 714
INSERT 文, 719
INSERT 文の使用, 619
LOAD TABLE 文, 717
LOAD TABLE 文の例, 727
MERGE 文, 720
NULL 値, 728
XML、DataSet オブジェクトを使用, 675
XML、OPENXML 演算子を使用, 669
XML、xp_read_file システムプロシージャを使用, 673
XML 文書, 669
xp_read_file システムプロシージャ, 673
一致しないテーブル構造, 728
インポートウィザード, 712, 726
インポート／エクスポートの状況, 709
考慮事項, 711
説明, 709
他のデータベースから, 725
ツール, 711
テンポラリテーブル, 728
データベースのバックアップ, 710
データベースへのデータのインポート, 711
テーブル, 726, 727
パフォーマンス, 709
パフォーマンスに関するヒント, 711
プロキシテーブル, 725
変換エラー, 726
データのインポートとエクスポート
説明, 709
データのエクスポート
dbunload ユーティリティ, 737
Interactive SQL のエクスポートウィザード, 731
OUTPUT 文, 733
UNLOAD TABLE 文, 735
UNLOAD 文, 736
UNLOAD 文を使用してファイルに, 736
XML, 668
クエリ結果, 731, 740, 741
結果セット, 769
考慮事項, 730
スキーマ, 757
説明, 730
ツール, 730
データベースのバックアップ, 710
データの削除
DELETE 文, 630
TRUNCATE TABLE 文, 632
データの選択
サブクエリを使用, 594

Copyright © 2013, SAP AG or an SAP affiliate company.

データの挿入
BLOB, 623
INPUT 文, 714
INSERT の使用, 619
INSERT 文, 719
MERGE 文, 720
SELECT の使用, 622
カラムデータ INSERT 文, 620, 621
指定カラム, 621
指定されていないカラムの動作, 620
制約, 620
全カラム, 620
デフォルト, 620
データの追加
INSERT の使用, 619
説明, 619
データの変更
INSERT 文, 630
UPDATE 文, 625
権限, 617
制約違反がある UPDATE 文の処理, 629
複数のテーブルを使用したデータの更新, 627
データのロード
変換エラー, 726
データベース
ASE 互換データベースでの大文字と小文字の区別, 647
Interactive SQL でのシステムオブジェクトのリストの表示, 3
SQL Anywhere への移行, 761
SQL Remote の抽出, 760
Transact-SQL との互換性, 647
XML サポート, 667
XML のインポート, 669
XML の格納, 667
アンロード, 743
アンロードと再ロード, 759
エクスポート, 743
大文字と小文字の区別, 651
オブジェクトを使用した作業, 1
再ロード, 758
システムオブジェクトの表示, 3
データベースのファイルフォーマットのアップグレード, 750
データベースファイルの圧縮に対する警告, 254
同期に関連しないデータベースのアンロードと再ロード, 752
同期に関連しないデータベースの再構築, 752
同期に関連するデータベースのアンロードと再ロード, 754, 755
複数のデータベースのテーブルのジョイン, 797
データベースアンロードウィザード
使用, 737
データベース移行ウィザード
使用, 761
説明, 761
データベースオブジェクト
間接参照, 43
直接参照, 43
データベースオブジェクトの使用, 1
データベースオプション
Transact-SQL 互換性のための設定, 650
インデックスコンサルタント, 161
テキストインデックスへの影響, 402
テキスト設定オブジェクトの設定, 402
マテリアライズドビューに対する影響, 58
データベース作成ウィザード
Transact-SQL と互換性のあるデータベースの作成, 648
データベーススレッド
ブロック, 883
データベース抽出ウィザード
SQL Remote, 760
データベーストレーシングウィザード
使用, 176
チュートリアル, 255
データベースのアップグレード
説明, 751
データベースのアンロード
Sybase Central, 737
エクスポート, 743
カンマ区切りのフォーマット, 752
説明, 749
データベースのアンロードユーティリティ (dbunload)
テーブルデータのエクスポート, 757, 758
データベースの移行
sa_migrate システムプロシージャの使用, 762
説明, 761
データベース移行ウィザード, 761
データベースのエクスポート
Sybase Central, 743
コマンドライン, 744
データベースの更新

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01009

概要, 617
データベースの再構築
Mobile Link, 754, 755
SQL Remote, 754, 755
UNLOAD TABLE 文, 756
エクスポートのコマンド, 750
考慮事項, 749
コマンドライン, 758
説明, 749
ツール, 749
テーブルの断片化の削減, 241
同期に関連するデータベースでの dbunload の使用, 754, 755
パフォーマンス向上のためのヒント, 248
非レプリケートデータベース, 752
理由, 751
レプリケートデータベース, 754, 755
データベースの再ロード
コマンドライン, 758
説明, 749
データベースの作成
Transact-SQL と互換性のあるデータベース, 648, 649
外部トレーシング, 183
データベースのブラウズ
独立性レベル, 904
データベースファイル
断片化, 240
パフォーマンス, 247
ファイルの断片化, 240
データベースページ
インデックスコンサルタントの結果, 159
[データ] メニュー
クエリ結果のエクスポート, 731
データリカバリ
インポートとエクスポート, 710
トランザクション, 618
テーブル
CREATE EXISTING TABLE 文を使用したプロキシテーブルの作成, 792
CREATE TABLE 文を使用したプロキシテーブルの作成, 793
SQL を使用した外部キーの管理, 23
SQL を使用した外部キーの追加, 23
SQL を使用したデータの表示と編集, 10
SQL を使用したプライマリキーの管理, 19
SQL を使用したプライマリキーの追加, 19
Sybase Central からのアンロード, 739
Sybase Central からのプロキシテーブルの作成, 792
Sybase Central での外部キーの作成, 22
Sybase Central でのデータの表示と編集, 9
Sybase Central でのプライマリキーの管理, 18
Sybase Central でのプライマリキーの追加, 18
Sybase Central でのプライマリキーの表示, 18
Sybase Central を使用した削除, 8
Sybase Central を使用した変更, 6
Transact-SQL と互換性のあるテーブルの作成, 655
位置ロック, 892
インポート, 726, 727
エクスポート, 745
外部キーの管理, 20
書き込みを意図したロック, 891
共有ロック, 891
クエリでの命名, 294
グループ読み込み, 250
検査制約, 847
作成, 5
サーバ上のリモートテーブルのリスト, 779
システムテーブルの内容の表示, 3
修飾された名前, 1
処理, 4
相関名, 294
挿入ロック, 893
断片化, 241
断片化のチュートリアル, 271
通常のビューとマテリアライズドビューとの簡単な比較, 40
デフラグメンテーション, 241
テンポラリテーブルの作成, 11
データのエクスポート, 757
テーブル制約の管理, 845
排他ロック, 892
ビットマップ, 250
ビューの依存性, 8
複数のデータベースからのジョイン, 797
プライマリキーの管理, 17
プロキシテーブルの操作, 790
別のテーブルからの参照を表示, 22
変更時の考慮事項, 5
マテリアライズドビューに参照された場合に変更, 42
幻ロック, 893
リモートデータアクセス, 771
ロック, 891

Copyright © 2013, SAP AG or an SAP affiliate company.

ローのコピー, 623
ワークテーブル, 230
テーブル関数
SQL Anywhere 実装, 638
テーブル検査制約作成ウィザード
アクセス, 845
テーブル構造のマージ
説明, 729
テーブルサイズ
説明, 250
パフォーマンスの考慮事項, 250
テーブル作成ウィザード
アクセス, 5
テーブル式
キージョイン, 524
ジョイン方法, 495
テーブルスキャン
ディスク割り付けとパフォーマンス, 624
テーブル制約
一意性, 846
データの整合性を維持するためのツール, 834
テーブルのインポート
DEFAULTS オプション, 728
NULL 値, 728
一致しないテーブル構造, 728
説明, 726, 727
テンポラリテーブル, 728
テーブル構造のマージ, 729
テーブルのエクスポート
dbunload ユーティリティ, 746
UNLOAD 文, 745
スキーマ, 758
テーブルの再編成
テーブルの断片化の削減, 241
テーブルのジョイン
2 つのテーブル, 494
3 つ以上のテーブル, 495
テーブルの断片化
説明, 241
[断片化] タブ, 242
チュートリアル, 271
テーブルのロード
データベースミラーリングの考慮事項, 718
テーブルヒント
対応する独立性レベル, 864
テーブル名
識別, 285
プロシージャでの完全修飾, 139
プロシージャとトリガ, 139
[テーブル名のルックアップ] ウィンドウ
テーブルリストの表示, 489
テーブルロック
位置, 892
書き込みを意図した, 891
競合, 891
共有, 891
説明, 891
挿入, 893
排他, 892
幻, 893
[テーブルロック] タブ
Sybase Central, 888

と

等価ジョイン
説明, 498
同期
データベースの再構築, 754, 755
統計
アクセスプラン, 358
インデックス, 204
カラム統計の更新, 221
キャッシュ, 197
実行プラン, 328
その他, 210
チェックポイントとリカバリ, 198
通信, 200
ディスク I/O, 202
ディスク書き込み, 203
ディスク読み込み, 202
統計ガバナー, 222
統計クリーナー, 222
パフォーマンスモニタ, 197
パフォーマンスモニタからの削除, 194
パフォーマンスモニタに追加, 194
パフォーマンスモニタを使用したモニタリング, 193
メモリページ, 207
モニタリング, 191
統計ガバナー
説明, 222
統計クリーナー
説明, 222
統計情報
ユーザ定義, 209

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01011

ユーザ定義統計情報のアルファベット順リスト, 209
統計値
インデックスの統計値のアルファベット順リスト, 204
キャッシュの統計値のアルファベット順リスト, 197
その他の統計値のアルファベット順リスト, 210
チェックポイントとリカバリの統計値のアルファベット順リスト, 198
通信の統計値のアルファベット順リスト, 200
ディスク I/O の統計値のアルファベット順リスト, 202
ディスク書き込みの統計値のアルファベット順リスト, 203
ディスク読み込みの統計値のアルファベット順リスト, 202
メモリ診断の統計値のアルファベット順リスト, 205
メモリページの統計値のアルファベット順リスト, 207
要求の統計値のアルファベット順リスト, 208
リスト, 197
統計値の削除
パフォーマンスモニタ, 194
統計値の追加
パフォーマンスモニタ, 194
等号演算子
WHERE 句での比較演算子, 299
同時実行性
DDL 文, 932
ISO SQL 標準, 875
一貫性, 875
インデックスの使用による改善, 905
改善, 905
説明, 863
パフォーマンス, 864
プライマリキー, 927
矛盾, 875
利点, 863
ロックの仕組み, 886
同時トランザクション
ブロック, 882
動的キャッシュサイズ決定
UNIX, 217
Windows, 216
Windows Mobile, 216
説明, 216
パフォーマンス向上のためのヒント, 216
ドキュメント
全文検索, 442
全文検索のワークフロー, 443
挿入, 623
特殊なジョイン
説明, 507
独立性レベル
ODBC, 878
各レベルの典型的なトランザクション, 904
参照, 881
スナップショットアイソレーションのレベルの選択, 902
設定, 877
説明, 864
選択, 902
チュートリアル, 906
典型的なトランザクション, 904
トランザクション内の変更, 880
矛盾のケース, 875
レベル 0 で実装, 895
レベル 1 で実装, 895
レベル 2 で実装, 896
レベル 2 と 3 での同時実行性の改善, 905
レベル 3 で実装, 896
独立性レベル 0
SELECT 文のロック, 895
説明, 864
ダーティリードのチュートリアル, 907
独立性レベル 1
SELECT 文のロック, 895
繰り返し不可能読み出しのチュートリアル, 912
説明, 864
独立性レベル 2
SELECT 文のロック, 896
説明, 864
幻ロックのチュートリアル, 924
幻のローのチュートリアル, 917
独立性レベル 3
SELECT 文のロック, 896
説明, 864
独立性レベルと一貫性
説明, 864
独立性レベルの参照
説明, 881
独立性レベルの選択

説明, 902
独立性レベルの変更
説明, 877
ドメイン
SQL を使用した作成, 848
Sybase Central を使用した作成, 848
大文字と小文字の区別, 652
カラムへの適用, 849
カラムへの割り当て, 848
検査制約, 845
削除, 850
使用, 848
使用例, 848
ドメイン作成ウィザード
使用, 848
ドメインのカラム検査制約
継承, 845
トラブルシューティング
ANY 演算子, 606
GROUP BY 句, 315
アプリケーションプロファイリング, 180
結果セットが変わったように見える, 312
デッドロック, 884
ナチュラルジョイン, 517
パフォーマンス, 211
リモートデータアクセス, 807
トランザクション
開始, 861
干渉, 882
完了, 861
サブトランザクションとセーブポイント, 864
使用, 861
スナップショットアイソレーションでの開始, 870
説明, 861
セーブポイント, 864
デッドロック, 883
典型的な独立性レベル, 904
データ操作, 617
データリカバリ, 618
同時実行性, 863
独立性レベルの変更, 880
トランザクション管理の制限, 803
複数, 863
プロシージャとトリガ, 138
ブロック, 882
ブロックのチュートリアル, 912
リモートデータアクセス, 803
トランザクション間の干渉
説明, 882
トランザクション結果の保存
説明, 861
トランザクション処理
スケジューリング, 903
スケジューリングの影響, 903
直列化可能なスケジューリング, 903
データリカバリ, 618
パフォーマンス, 864
ブロック, 882
トランザクションスケジューリング
影響, 903
トランザクションと独立性レベル
説明, 861
トランザクションによる変更のグループ分け
説明, 861
トランザクションのインターリーブ
説明, 903
トランザクションの管理とリモートデータ
説明, 802
トランザクションの完了
説明, 861
トランザクションのコミットの統計値
説明, 208
トランザクションの終了
説明, 861
トランザクションの順序付け
説明, 903
トランザクションのスケジューリング
説明, 903
トランザクションのロールバックの統計値
説明, 208
トランザクションブロック
説明, 881
トランザクションログ
dbmlsync, 754, 755
データリカバリ, 710
パフォーマンス向上のためのヒント, 239
パフォーマンスに関するヒント, 239
レプリケーション, 754, 755
トランザクションログループのコミットの統計値
説明, 203
トランザクションロック
期間, 887
トリガ
AFTER トリガ, 94

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01013

BEFORE トリガ, 94
INPUT 文による INSERT トリガの起動, 709
INSTEAD OF トリガ, 105
ROLLBACK 文, 661
Transact-SQL, 660
Transact-SQL の互換性と名前, 652
UPDATE 文, 627
エラー処理, 125
概要, 77
カーソル, 121
許可される文, 140
警告, 128
構造, 110
再帰, 661
削除, 101
作成, 96
作成 (Sybase Central の場合), 96
作成のヒント, 138
時間, 139
実行, 99
実行権限, 104
実行順序, 104
使用, 94
説明, 77
セーブポイント, 138
操作の一時的な無効化, 102
タイプ, 94
データの整合性を維持するためのツール, 835
トリガ作成ウィザード, 96
トリガの実行順序, 104
日付, 139
プロファイリング結果の生成と確認, 150
文デリミタ, 138
文レベル, 660
変更, 99
利点, 77
例外ハンドラ, 129
トリガ作成ウィザード
使用, 96
トリガの条件
トリガの実行順序, 104
トレーシング
説明, 162
データベーストレーシングを使用したアプリケーションプロファイリング, 162
トレーシングデータベース, 163
トレーシングセッション
説明, 163
トレーシングセッション実行中の診断トレーシングの設定の変更
説明, 176
トレーシングデータ
アンロード操作の一部としてアンロードされない, 163
説明, 163
トレーシングデータベース
説明, 163
トークン
単語区切りと全文検索, 442

な

内部オペレーション
リモートデータアクセス, 803
内部ジョイン
SQL Anywhere 実装, 637
ジョインの削除によるリライト最適化, 231
説明, 501
内部ジョインと外部ジョイン
説明, 500
内部ロード
説明, 709
内容
ディレクトリアクセスサーバ, 781
長いテキストプラン
SQL 関数を使用して表示, 332
説明, 330
ナチュラルジョイン
ON 句の使用, 518
SQL Anywhere 実装, 637
SQL 言語, 637
エラー, 517
説明, 516
テーブル式, 518
ビューと派生テーブル, 519

に

二重引用符
文字列, 305
認証
LDAP ユーザ認証、説明, 935

ね

ネイティブ文
リモートサーバへの送信, 798
ネスト

外部ジョイン, 503
ジョイン, 495
ジョインでの派生テーブル, 513
ネストされたサブクエリ
説明, 598
ネストされた複合文と例外処理
説明, 131
ネームスペース
XML での定義, 675
インデックス, 652
トリガ, 652

は

排他テーブルロック
説明, 892
排他リスト
実行プラン内の項目, 364
バイナリラージオブジェクト
挿入, 623
バイパスクエリ
オプティマイザのバイパス, 319
グラフィカルなプランに表示されない, 337
定義, 319
バケット
ヒストグラム, 322
パケットサイズ
パフォーマンス, 255
パスワード
大文字と小文字の区別, 652
派生テーブル
DML 文から選択する, 296
外部ジョイン, 504
キージョイン, 529
ジョイン, 513
説明, 295
ナチュラルジョイン, 519
パターン一致
概要, 301
バッチ
OUTPUT 文, 109
SELECT 文の使用, 140
Transact-SQL, 661
許可される SQL 文, 140
許可される文, 140
作成, 107
作成のヒント, 138
ストアドプロシージャとの比較, 107
制御文, 107
説明, 107
複合文, 108
バッチ処理
Interactive SQL, 768
バッチに使用できる文
説明, 140
バッチモード
Interactive SQL, 768
幅の広いカラム
説明, 250
パフォーマンス
All-rows 最適化ゴール, 230
Windows パフォーマンスモニタの統計, 195
Windows パフォーマンスモニタを使用したモニタリング, 195
WITH EXPRESS CHECK, 229
アプリケーションプロファイリング, 148
インデックス, 26
インデックスに関する考慮事項, 223
インデックスの使用, 26
オプティマイザによる推定と実際の統計の比較, 335
オプティマイザの負荷, 220
改善, 30
改善とロック, 905
カスケードされた参照アクションを最小限に抑える, 247
キャッシュ読み込みヒット率, 336
キー, 249
クエリ速度の測定, 237
向上のためのヒントのリスト, 211
実行時間の実際値と推定値, 336
実行プランの解釈, 328
述部の分析, 282
詳細なアプリケーションプロファイリング, 162
推定ソース, 336
説明, 211
選択性, 335
データベースの再構築, 248
テーブルとページのサイズ, 250
パケットサイズ, 255
パフォーマンスのモニタリングと改善のためのツール, 145
パフォーマンスモニタの統計値, 197
パフォーマンスモニタを使用したモニタリング, 193

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01015

バルクロード, 709
ファイルの断片化, 240
分散読み込み, 252
ページサイズ, 250
ページサイズの推奨値, 250
マテリアライズドビューを使用した向上, 225
モニタリング, 191
ワークテーブル, 230
パフォーマンス向上のためのヒント
クエリのパフォーマンスのモニタ, 237
断片化の削減, 240
テーブルの断片化の削減, 241
パフォーマンスツール
グラフィカルなプラン, 333
タイミングユーティリティ, 191
プロシージャプロファイリング用システムプロシージャ, 188
パフォーマンス統計
モニタリング, 191
パフォーマンスの向上
インデックス, 224
オプティマイザの目標の選択, 220
異なるファイルの異なるデバイスへの配置, 247
制約の宣言, 247
説明, 211
小さいテーブルに関する統計収集の検討, 240
テーブル内のカラムの順序, 250
同時実行性の問題点の確認, 219
トランザクションログ, 239
バルクオペレーション, 709
プライマリキー幅の縮小, 249
パフォーマンスのモニタ
実行プランに使用される省略形, 342
実行プランの解釈, 328
パフォーマンスのモニタリング
クエリ測定ツール, 237
パフォーマンスのモニタリングと改善
説明, 145
パフォーマンスモニタ
Sybase Central, 193
Sybase Central で開く, 194
Windows パフォーマンスモニタ, 195
概要, 193
サポートされている統計値のリスト, 197
説明, 193
統計値の追加と削除, 194
パフォーマンスモニタの統計値, 197
パブリケーション
テーブルの削除, 8
パラメータ
関数, 313
パラメータを渡す
関数, 111
プロシージャ, 111
バルクオペレーション
説明, 709
データのリカバリの問題, 710
パフォーマンス向上のためのヒント, 253
バルクオペレーション
パフォーマンス, 709
パフォーマンスの側面, 709
範囲クエリ
説明, 300
範囲バウンド
実行プラン内の項目, 362
バージョンストアページの統計値
説明, 210
パーミッション
Adaptive Server Enterprise, 645
ディレクトリアクセスサーバ, 781

ひ

比較
NULL 値, 307
概要, 309
後続ブランク, 299
ソート順, 299
比較演算子
NULL 値, 306
サブクエリ, 611
比較テスト
サブクエリ, 603
非決定性関数
副次的影響, 327
ビジネスルール
説明, 833
ヒストグラム
更新, 221
説明, 322
非相関サブクエリ
説明, 597
左外部ジョイン
SQL Anywhere 実装, 637
説明, 501

Copyright © 2013, SAP AG or an SAP affiliate company.

日付
SQL Anywhere 実装, 635
検索, 305
探索条件の概要, 309
入力規則, 305
プロシージャとトリガ, 139
日付／時刻デフォルト
説明, 838
必須
外部キー, 852
ビット
実行プラン内の項目, 363
ビットマップ
スキャン, 250
等しくない
WHERE 句での比較演算子, 299
ビュー
FROM 句, 294
INSTEAD OF トリガを使用した更新, 105
SQL を使用した通常のビューの無効化, 51
SQL を使用した通常のビューの有効化, 51
Sybase Central を使用した削除, 49
Sybase Central を使用した通常のビューの作成, 47
Sybase Central を使用した通常のビューの変更, 48
Sybase Central を使用した通常のビューの無効化, 50
Sybase Central を使用した通常のビューの有効化, 50
エクスポート, 733
外部ジョイン, 504
共通テーブル式, 532
キージョイン, 529
更新, 45
チェックオプションと通常のビュー, 45
通常のビュー内のデータのブラウズ, 53
通常のビューの DISABLED ステータス, 46
通常のビューの INVALID ステータス, 46
通常のビューの SELECT 文の制限, 44
通常のビューの VAID ステータス, 46
通常のビューの使用, 44
通常のビューのステータス, 46
通常のビューの変更時の考慮事項, 48
テキストインデックスを使用した問い合わせ, 392
ナチュラルジョイン, 519
ビューの依存性の処理, 41
ビューの操作, 39
プログラム変数を参照, 536
変更とビューの依存性, 48
ビュー作成ウィザード
使用, 47
ビューの依存性
依存性情報の検索, 43
カタログ内の情報, 43
スキーマ変更, 42
説明, 41
通常のビューのステータス, 46
ビューのエクスポート
説明, 745
ビューのステータス
DISABLED, 46
INVALID, 46
VALID, 46
説明, 46
通常のビュー, 46
特定, 46
マテリアライズドビューのステータス, 72
ビューマッチング
アルゴリズムの要件, 226
クエリ検査, 226
クエリ実行, 225
実行プランの結果, 360
スナップショットアイソレーションでの使用, 869
説明, 226
ビューマッチングアルゴリズム、実行プランの結果, 360
ビューマッチングアルゴリズム、説明, 226
ヒューリスティック
クエリの最適化, 323
表示
通常のビューのデータ, 53
プロシージャプロファイリングの結果, 154
標準出力
ファイルへのリダイレクト, 731, 740, 741
標準偏差関数
OLAP, 579
開いているカーソルの統計値
説明, 208
ヒント
インデックスヒント, 26
クラスタドインデックスの使用, 30
パフォーマンス向上, 212
ヒープのカーバーの統計値

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01017

説明, 205
ヒープのクエリ処理の統計値
説明, 205
ヒープの再配置可能の統計値
説明, 205
ヒープの再配置可能ロックの統計値
説明, 205

ふ

ファイル
グラフィカルなプラン, 333
断片化, 240
ファイルの断片化
説明, 240
ファンアウト
インデックス, 38
ファンクション
ファンクション作成ウィザード, 89
ファンクション作成ウィザード
アクセス, 89
フェーズ
クエリ処理のフェーズ, 317
深さ
実行プラン内の項目, 362
複合インデックス
ORDER BY 句, 29
カラムの順序の効果, 28
説明, 28
復号化
Sybase Central を使用したマテリアライズド
ビューの復号化, 68
複合文
アトミック, 114
使用, 113
宣言, 114
複雑な外部ジョイン
説明, 503
複数のカラムを使用したグループ化
説明, 473
複数の結果セット
Interactive SQL の表示, 119
複数のデータベース
ジョイン, 797
複数のトランザクション
同時実行性, 863
複数ローのサブクエリ
説明, 594
不正な XML 名のエンコーディング
説明, 679
物理インデックス
共有される物理インデックスの特定, 37
説明, 36
不定の値
説明, 305
不等号
不等のテスト, 309
部分インデックススキャン
説明, 39
不要な DISTINCT の削除
説明, 231
付与されたページの統計値
説明, 205
付与失敗数の統計値
説明, 205
付与待機数の統計値
説明, 205
付与要求数の統計値
説明, 205
プライマリキー
GLOBAL AUTOINCREMENT, 840
NEWID を使用して UUID を作成, 841
SQL を使用した削除, 19
SQL を使用した作成, 19
SQL を使用した変更, 19
Sybase Central で削除, 18
Sybase Central で作成, 18
Sybase Central で変更, 18
エンティティ整合性, 851
オートインクリメント, 839
管理, 17
シーケンスを使用した生成, 928
整合性, 636
生成, 927
生成されたインデックス, 26
説明, 17
ソート順序, 636
データの整合性を維持するためのツール, 834
同時実行性, 927
パフォーマンス, 249
例, 17
プライマリキーカラム
実行プラン内の項目, 362
プライマリキー設定ウィザード
アクセス, 18
プライマリキーテーブル

Copyright © 2013, SAP AG or an SAP affiliate company.

実行プラン内の項目, 362
プライマリキーテーブルの予測ロー数
実行プラン内の項目, 362
ブラウズ
通常のビュー, 53
プラス演算子
NULL 値, 308
プラン
解釈, 328
キャッシュ, 325
クエリを実行しないで表示, 329
グラフィカルなプラン, 333
コンテキスト別のヘルプ, 337
使用される省略形, 342
長いテキストプラン, 330
短いテキストプラン, 329
プラン構築フェーズ
クエリ処理, 318
プラン内の選択性
説明, 339
プランに使用される一般的な統計
説明, 358
プランのキャッシュ
説明, 325
プランビューア
アクセス, 341
[オプティマイザ統計] フィールドの説明, 350
[ノード統計] フィールドの説明, 355
ローカルフィールドまたはグローバルフィールドの説明, 350
古いデータ
手動ビューでの再表示, 57
古さ
手動ビュー, 56
マテリアライズドビューの設定, 75
プレフィクス検索
GENERIC テキストインデックス, 385
NGRAM テキストインデックス, 385
N-gram テキストインデックスでの予期しない結果, 383
全文検索, 383
プレフィクス、全文検索
説明, 377
プレフィクス単語
説明, 383
ブレーク条件
設定, 954
ブレークポイント
カウント, 956
個々の接続, 956
個々のユーザ, 956
条件, 956
ステータス, 955
設定, 954
説明, 953
無効化, 955
有効化, 955
ブレークポイントの設定
デバッガ, 954
ブレークポイントの無効化
説明, 955
有効化, 955
ブレークポイントの有効化
説明, 955
フレーズ
全文検索, 381
全文検索での特殊文字, 381
フレーズ、全文検索
説明, 377
プロキシテーブル
CREATE EXISTING TABLE 文を使用して作成, 792
CREATE TABLE 文を使用した作成, 793
SQL を使用した作成, 792
Sybase Central からの作成, 792
説明, 790
ディレクトリアクセスサーバからの削除, 787
ディレクトリアクセスプロキシテーブルの問い合わせ, 785
デリミタとディレクトリアクセス, 786
データのインポート, 725
プロキシテーブルのロケーションの指定, 791
プロキシテーブル作成ウィザード
使用, 792
プログラム変数
共通テーブル式, 536
プロシージャ
EXECUTE IMMEDIATE 文, 135
FROM 句内での使用, 295
Sybase Central を使用した変更, 86
Transact-SQL, 662
Transact-SQL でのエラー処理, 664
Transact-SQL の概要, 660
Watcom SQL を使用した Transact-SQL のエラー処理の模倣, 666
WITH RESULT SET 句, 135

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01019

エラー処理, 125
概要, 77
カーソル, 121
カーソルの使用, 122
許可される文, 140
クエリ変形の一環としてインライン化, 231
警告, 128
結果セット, 118
結果を返す, 115
結果を結果セットとして返す, 118
構造, 110
コピー, 88
削除, 88
作成, 85
作成のヒント, 138
時間, 139
使用, 78
所有者または呼び出し側の権限でプロシージャとファンクションを実行する, 78
説明, 77
セーブポイント, 138
デフォルトのエラー処理, 125
テンポラリテーブル参照時の考慮事項, 12
テーブル名, 139
統計, 322
入力検証, 139
パラメータ, 111
パラメータの宣言, 110
日付, 139
複数の結果セット, 119
プロシージャ作成ウィザード, 85
プロシージャ、トリガ、ユーザ定義関数、バッチを作成するときのヒント, 138
プロファイリング結果の生成と確認, 150
文デリミタ, 138
文のキャッシュ, 325
変換, 662
変数結果セット, 121
戻り値, 665
呼び出し, 87
利点, 77
リモートプロシージャの削除, 801, 802
リモートプロシージャの作成, 800
例外ハンドラ, 129
プロシージャから返される結果
説明, 115
プロシージャ言語
概要, 660
プロシージャ作成ウィザード
使用, 85
プロシージャとトリガでのデフォルトの警告処理
説明, 128
プロシージャの変更
説明, 86
プロシージャの呼び出し
説明, 87
プロシージャプロファイリング
sa_server_option を使用したプロファイリングのフィルタの設定, 188
sa_server_option を使用したプロファイリングのリセット, 189
sa_server_option を使用した無効化, 189
Sybase Central, 150
システムプロシージャを使用したプロファイリングデータの取り出し, 190
システムプロシージャを使用して実行, 188
プロファイリング結果の解釈, 154
プロファイリング結果の分析, 155
プロファイリングできるオブジェクト, 154
ベースラインのチュートリアル, 274
無効化, 153
有効化, 151, 188
リセット, 152
プロシージャプロファイリングの結果の分析
説明, 154
プロシージャプロファイリングの無効化
説明, 153
プロシージャプロファイリングのリセット
説明, 152
プロシージャを作成するときのヒント
プロシージャ内のテーブルでの完全に修飾された名前の使用, 139
プロシージャの中で文を区切る, 139
ブロック
説明, 882
チュートリアル, 912
デッドロック, 883
トラブルシューティング, 884
トランザクション, 881
ブロックオプション
使用, 882
プロファイリングデータベース
内部作成と外部作成, 163
文
最適化, 321

Copyright © 2013, SAP AG or an SAP affiliate company.

サポートされない Transact-SQL 文, 643
速度が遅い文を診断するチュートリアル, 263
デリミタ, 138
複合, 113
文キャッシュヒットの統計値
説明, 208
文キャッシュミスの統計値
説明, 208
分散読み込み
パフォーマンス, 252
文の完全なパススルー
リモートデータアクセス, 805
文の準備の統計値
説明, 208
文の統計値
説明, 208
文の部分的なパススルー
リモートデータアクセス, 805
文レベルのトリガ
Transact-SQL, 660
ブール検索
全文検索, 388

へ

平方偏差関数
OLAP, 579
並列処理
クエリ内, 366
説明, 364
べき等関数
定義, 327
ベクトル集合関数
説明, 474
変換
プロシージャ, 662
変形
リライト最適化のリスト, 327
変更
Sybase Central を使用したプロシージャの変更, 86
検査制約, 847
シーケンス, 931
テキストインデックス, 369
トリガ, 99
リモートサーバ, 779
変更の確定
説明, 618
変更のキャンセル
説明, 618
変数
SELECT 文, 658
SET 文, 658
Transact-SQL, 663
デバッガでの値の表示, 957
ローカル, 658
割り当て, 658
変数の検査
デバッガ, 957
ページ
実行プラン内の項目, 363
挿入されたローに対するディスク割り付け, 624
ページサイズ
Windows Mobile の考慮事項, 251
インデックス, 252
説明, 250
挿入されたローに対するディスク割り付け, 624
パフォーマンス, 250
パフォーマンスの考慮事項, 250
ページマップ
実行プラン内の項目, 361
スキャン, 250
ベーステーブル
作成, 5
説明, 5
ベースライン
プロシージャプロファイリングのチュートリアル, 274

ほ

方向
実行プラン内の項目, 362
保護されたテーブル
外部ジョイン, 501
ホスト変数
バッチ内, 107
保存
結果セット, 769
トランザクション結果, 861

ま

マップ物理メモリ／秒の統計値
説明, 205

マテリアライズドビュー
COSTED ビューマッチングの原因, 360
SQL Anywhere 実装, 638
Sybase Central を使用した暗号化, 68
Sybase Central を使用した最適化での使用の無効化, 69
Sybase Central を使用した最適化での使用の有効化, 69
Sybase Central を使用した再表示, 63
Sybase Central を使用した削除, 67
Sybase Central を使用した作成, 61
Sybase Central を使用した初期化, 62
Sybase Central を使用したデータの設定, 62
Sybase Central を使用した復号化, 68
オプティマイザが検討したかどうかを判断する, 228
オプティマイザでのマテリアライズドビューに対する古さのしきい値の設定, 75
オプティマイザの考慮事項, 55
隠す, 66
カラム統計, 53
再表示タイプの変更, 71
手動と即時、比較, 56
手動ビューから即時ビューへの変更, 71
使用するかどうかの評価, 225
ステータスとプロパティの図, 74
ステータス, 72
スナップショットアイソレーションでのビューマッチングの使用, 869
接続とオプションの不一致, 228
接続の候補リストの決定, 228
説明, 53
即時ビューから手動ビューへの変更, 71
即時ビューの作成, 71
即時ビューを作成する際の制限, 59
通常のビューとテーブルとの簡単な比較, 40
ディスク領域の考慮事項, 53
データの最新性と一貫性, 53
データベースオプションの考慮事項, 58
テーブルの変更をブロックする依存性, 42
パフォーマンスに関するヒント, 225
パフォーマンスの向上のための使用, 54
ビューの依存性, 42
ビューマッチングアルゴリズムによる検査, 226
プランのキャッシュ, 325
古さ, 57
プロパティの概要, 73
保守コスト, 53
マテリアライズドビュー作成オプションの取得方法, 70
マテリアライズドビューに関する情報の取得, 70
マテリアライズドビューによるパフォーマンスの向上, 225
マテリアライズドビューを管理する際の制限, 58
マテリアライズドビューを使用する状況の判断, 54
有効化と無効化, 64
マテリアライズドビュー作成ウィザード
アクセス, 61
マテリアライズドビューのステータスとプロパティ
説明, 72
幻ロック
説明, 893
チュートリアル, 924
幻ロー
チュートリアル, 917
データの矛盾, 875
独立性レベル, 875
独立性レベル 2 で防止, 896
マルチページ割り当ての統計値
説明, 205
マージ
トリガの動作, 721
マージ動作の定義
説明, 720

み

右外部ジョイン
SQL Anywhere 実装, 637
説明, 501
短いテキストプラン
SQL 関数を使用して表示, 331
説明, 329
[未使用のインデックス] タブ
インデックスコンサルタントの結果, 160
未スケジュールの要求の統計値
説明, 208
見積り済み最良プラン
[オプティマイザ統計] フィールドの説明, 350
見積り済みプラン
[オプティマイザ統計] フィールドの説明, 350

Copyright © 2013, SAP AG or an SAP affiliate company.

む

無効化
SQL を使用した通常のビューの無効化, 51
Sybase Central を使用した通常のビューの無効化, 50
トリガ操作, 102
マテリアライズドビュー, 64
マテリアライズドビューのステータス, 72
無効なデータ
説明, 833
矛盾
ISO SQL 標準, 875
繰り返し不可能読み出し, 875
繰り返し不可能読み出しのチュートリアル, 912
ダーティリード, 875
ダーティリードとロック, 895
ダーティリードのチュートリアル, 907
直列化不可能なスケジュールの影響, 904
幻ロックのチュートリアル, 924
幻ロー, 875
幻ローとロック, 896
幻ローのチュートリアル, 917
ロックを使用して防止, 886

め

明示的ジョイン条件
説明, 492
命令グラフ
説明, 545
メインヒープバイトの統計値
説明, 210
メタプロパティ名
id, 672
localname, 672
メモリ
接続制限, 253
メモリガバナー
説明, 213
メモリページのカーバーの統計値
説明, 205
メモリページのクエリ処理の統計値
説明, 205
メモリページの固定カーソルの統計値
説明, 205
メモリページの再配置可能の統計値
説明, 207
メモリページの再配置／秒の統計値
説明, 207
メモリページの統計値
リスト, 207
メモリページのトリガ定義の統計値
説明, 207
メモリページのビュー定義の統計値
説明, 207
メモリページのプロシージャ定義の統計値
説明, 207
メモリページのマップページの統計値
説明, 207
メモリページのメインヒープの統計値
説明, 207
メモリページのロックテーブルの統計値
説明, 207
メモリページのロックヒープの統計値
説明, 207
メモリページのロールバックログの統計値
説明, 207
メンテナンス
パフォーマンス, 211

も

文字セット変換
リモートデータアクセス, 773
文字データ
検索, 305
文字列
quoted_identifier と引用符, 305
SELECT リストの使用, 291
引用符, 305
引用符と文字列, 305
使用, 305
全文検索を使用したデータベースの検索, 367
マッチング, 301
文字列解釈の例
全文検索, 405
文字列と数値のデフォルト
説明, 842
文字列の連結
NULL, 308
戻り値
プロシージャ, 665
役割
Adaptive Server Enterprise, 644
役割名

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01023

説明, 521

ゆ

有効化
SQL を使用した通常のビューの有効化, 51
Sybase Central でのプロシージャプロファイリング, 151
Sybase Central を使用した通常のビューの有効化, 50
マテリアライズドビュー, 64
マテリアライズドビューのステータス, 72
有効なユーザ
ログインユーザとの比較, 78
ユニークキー
生成と同時実行性, 927
ユニークな結果
制限, 293
ユニークな識別子
テーブル, 17
ユーザ ID
Adaptive Server Enterprise, 645
大文字と小文字の区別, 652
デフォルト, 839
ユーザ定義関数
キャッシュ, 327
クエリ変形の一環としてインライン化, 231
削除, 92
作成, 89
作成のヒント, 138
実行権限, 93
スレッド安全性, 89
説明, 89
パラメータ, 111
プロファイリング結果の生成と確認, 150
呼び出し, 90
ユーザ定義データ型
SQL を使用した作成, 848
カラムへの適用, 849
検査制約, 845
削除, 850
作成, 848
ユーザ定義統計情報
リスト, 209
ユーザ認証
LDAP サーバの使用, 935

よ

要求
数の削減, 254
[要求] タブ
インデックスコンサルタントの結果, 160
要求トレース分析
実行, 182
説明, 182
要求の GET DATA／秒の統計値
説明, 210
要求のアクティブの統計値
説明, 208
要求のキャンセル
リモートデータアクセス, 808
要求の交換の統計値
説明, 208
要求の統計値
説明, 208
要求ロギング
クライアントでの文のキャッシュでの使用, 187
説明, 185
要求ログ
セキュリティ, 186
説明, 185
要素
クエリ結果を XML として取得, 677
データベースへの XML の格納, 667
リレーショナルデータからの XML の生成, 668
容量の計画
説明, 181
予測アクティブの統計値
説明, 205
予測キャッシュページ数
[オプティマイザ統計] フィールドの説明, 350
予測ページ数
実行プラン内の項目, 361
予測リーフページ数
実行プラン内の項目, 362
予測ローサイズ
実行プラン内の項目, 361
予測ロー数
実行プラン内の項目, 361
呼び出し側

Copyright © 2013, SAP AG or an SAP affiliate company.

16.0 より前のバージョンのシステムプロシージャを呼び出し側または定義者として実行する, 79
データベースが使用するセキュリティモデルの決定, 84
有効なユーザとログインユーザ, 78
呼び出し側の権限でプロシージャとファンクションを実行する, 78
呼び出しスタック
デバッガでの表示, 958
読み込みロック
競合, 894
説明, 889
長期間, 889
予約語
リモートサーバ, 807
より大きい
WHERE 句での比較演算子, 299
範囲指定, 300
より大きくない
WHERE 句での比較演算子, 299
より小さい
WHERE 句での比較演算子, 299
範囲指定, 300
より小さくない
WHERE 句での比較演算子, 299

ら

ランキング
集合で使用, 588
ランキング関数
最上位と最下位の百分位数の検索, 592
例, 584
ランダム変換
実行プラン内の項目, 362

り

利益
インデックスコンサルタントの結果, 160
リカバリ
インポート／エクスポート, 710
クライアント側データのロード, 749
リカバリ I/O 予測の統計値
説明, 198
リカバリの緊急度の統計値
説明, 198
リカバリの統計値
リスト, 198
リソースガバナ
定義, 253
リダイレクト
ファイルへの出力, 731, 740, 741
リモートサーバ
Lotus Notes SQL, 828
リモートサーバ
Advantage Database Server, 812
ASE ODBC, 814
IBM DB2, 817
Mac OS X 上の Ultra Light でサポートされない, 812
Microsoft Access, 821
Microsoft Excel, 827
Microsoft FoxPro, 828
MIRROR, 811
MySQL, 825
ODBC, 827
Oracle Database, 829
SAP HANA, 819
SQL Anywhere SAODBC, 811
SQL Server, 823
Sybase Central での作成, 775
Sybase IQ IQODBC, 820
Ultra Light, 812
外部ログイン, 789
クラス, 808
削除, 777
作成, 773
トランザクション管理, 802
ネイティブ文の送信, 798
プロパティのリスト, 780
変更, 779
リモートサーバ作成ウィザードを使用した作成, 775
リモートサーバ上のテーブルのリスト, 779
リモートサーバの機能のリスト, 780
リモートサーバの使用, 772
リモートサーバ作成ウィザード
使用, 775
リモートデータ
アクセス, 771
サポートされない機能, 807
プロキシテーブルのロケーションの指定, 791
リモートデータに対してサポートされない機能, 807
リモートテーブルのマッピング, 772

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01025

リモートデータアクセス
Lotus Notes SQL, 828
Microsoft Access, 821
Microsoft Excel, 827
Microsoft FoxPro, 828
Sybase IQ, 820
大文字と小文字の区別, 807
概要, 771
クエリ上でブロックされるクエリ, 808
クエリに関する一般的な問題, 807
クエリの解析, 804
クエリの正規化, 804
クエリの前処理, 804
サーバの機能, 804
接続の問題点, 807
接続名, 808
トラブルシューティング, 807
内部オペレーション, 803
パススルーモード, 798
パフォーマンスの制限, 771
文の完全なパススルー, 805
文の部分的なパススルー, 805
文字セット変換の制限, 773
リモートサーバ, 772
リモートデータアクセスの基本概念
概要, 771
リモートデータアクセスの接続の管理
説明, 808
リモートデータへのアクセス
基本概念, 771
説明, 771
リモートテーブル
アクセス, 771
カラムのリスト, 795
サーバ上のリモートテーブルのリスト, 779
ジョイン, 796
説明, 772
リモートトランザクション管理
概要, 803
リモートファンクション
データ型, 799
リモート プロシージャ
コール, 799
削除, 801, 802
作成, 800
データ型, 799
リモートプロシージャコール
説明, 799
リモートプロシージャ作成ウィザード
使用, 800
リライト最適化
リスト, 327
リレーショナルデータ
XML としてエクスポート, 668
リレーショナルデータを XML としてエクスポート
説明, 668
リーフページ
説明, 38

る

ループバック接続
説明, 797
ルール
Transact-SQL, 643

れ

例外
宣言, 126
例外ハンドラ
エラーログ, 134
ネストされた複合文, 131
プロシージャとトリガ, 129
例, 134
列挙フェーズ
クエリ処理, 318
レプリケーション
データベースの再構築, 754, 755
同期に関連するデータベースの再構築, 754, 755
連続変換
実行プラン内の項目, 362

ろ

ロードしないで
SQL スクリプトファイルを実行, 767
ログ
ロールバックログ, 864
ログインユーザ
有効なユーザとの比較, 78
[ログ] タブ
インデックスコンサルタントの結果, 160
ログファイル
ログファイルの圧縮に対する警告, 254
ロック

sa_locks システムプロシージャを使用したロックの表示, 888
Sybase Central でのデッドロックの表示, 885
Sybase Central での表示, 888
WITH HOLD の影響, 902
位置, 892
意図的ロー, 890
意図的ローロック, 890
インデックスによって影響を削減, 905
インデックスによる減少, 39
オーファンと参照整合性, 898
書き込み, 890
書き込みを意図したテーブル, 891
書き込みを意図したテーブルロック, 891
カーソル安定性, 902
期間, 887
競合, 893
競合処理, 882
競合のタイプ, 891
共有スキーマ, 889
共有テーブル, 891
共有テーブルロック, 891
クエリ時, 895
更新時, 899
更新手順, 899
削除時, 901
削除手順, 901
種類, 887
情報の表示, 888
スキーマ, 888
説明, 886
早期解放, 904
挿入, 893
挿入時, 897
挿入手順, 897
挿入ロック, 893
デッドロック, 883
典型的なトランザクションと独立性レベル, 904
テーブル, 891
独立性レベル, 864
独立性レベル 0 で実装, 895, 896
独立性レベルのチュートリアル, 912
トランザクションのブロックとデッドロック, 881
排他スキーマ, 889
排他テーブルロック, 892
排他ロック, 892
ブロック, 882
幻, 893
幻ロック, 893
幻ロックのチュートリアル, 924
矛盾と典型的な独立性レベル, 875
ロー, 889
ローの読み込み, 889
ローロックの書き込み, 890
ローロックの読み込み, 889
ロックされたローへのアクセス待機
デッドロック, 882
ロック数の統計値
説明, 208
ロックする
位置ロック, 892
ロックの解放
例外, 902
ロックの早期解放
トランザクション, 904
論理インデックス
共有される物理インデックスの特定, 37
説明, 36
論理演算子
HAVING 句, 478
条件の接続, 308
ロー
INSERT を使用したコピー, 623
意図的ロック, 890
削除, 630
削除の影響, 625
指定カラムへのデータの挿入, 621
全カラムへのデータの挿入, 620
選択, 298
ロック, 889
ロックの書き込み, 890
ローカルテンポラリテーブル
説明, 10
命名, 10
ローカル変数
デバッガ, 957
ロー制限数
実行プラン内の項目, 364
ロー制限句
使用, 480
ロード
Interactive SQL での SQL スクリプト, 769
データベースリカバリの考慮事項, 718
同期の考慮事項, 718

Copyright © 2013, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.01027

ローの制限
FIRST 句, 480
TOP 句, 480
ローの連続記憶領域
説明, 624
ローバージョン
説明, 870
ロールバックログ
セーブポイント, 864
データリカバリ, 710
ローロック
意図的, 890
書き込み, 890
種類, 889
説明, 889
読み込み, 889
ロックの読み込み, 889

わ

ワイルドカード
パターン一致, 301
文字列比較, 302
割り当て
カラムのデータ型, 848
カラムのドメイン, 848
ワークテーブル
クエリ処理, 230
説明, 230
パフォーマンスに関するヒント, 247

Copyright © 2013, SAP AG or an SAP affiliate company.