SQL Anywhere サーバ
SQL の使用法

改訂 2007 年 3 月

版権と商標

Copyright (c) 2007 iAnywhere Solutions, Inc. Portions copyright (c) 2007 Sybase, Inc. All rights reserved.

iAnywhere Solutions, Inc. は Sybase, Inc. の関連会社です。

iAnywhere は、(1) 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、(2) マニュアルの偽装表示をしない、(3) マニュアルに変更を加えないことが遵守されるかぎり、このマニュアルをご自身の情報収集、教育、その他の非営利の目的で使用することを許可します。このマニュアルまたはその一部を、iAnywhere の書面による事前の許可なく発行または配布することは禁じられています。

このマニュアルは、iAnywhere が何らかの行動を行う、または行わない責任を表明するものではありません。このマニュアルは、iAnywhere の判断で予告なく内容が変更される場合があります。iAnywhere との間に書面による合意がないかぎり、このマニュアルは「現状のまま」提供されるものであり、その使用または記載内容の誤りに対して iAnywhere は一切の責任を負いません。

iAnywhere (R)、Sybase (R)、http://www.ianywhere.com/trademarks に示す商標は Sybase, Inc. またはその関連会社の商標です。(R) は米国での登録商標を示します。

Java および Java 関連のすべての商標は、米国またはその他の国での Sun Microsystems, Inc. の商標または登録商標です。

このマニュアルに記載されているその他の会社名と製品名は各社の商標である場合があります。

目次

はじめに ix

SQL Anywhere のマニュアル x
表記の規則 xiii
詳細情報の検索／フィードバックの提供 xvii

I. データベースの設計と作成 1

データベースの設計 3
データベースの設計の概要 4
データベース設計の概念 5
設計のプロセス 11
テーブル・プロパティの設計 24
データベース・オブジェクトの使用 29
データベース・オブジェクトを使用した作業の概要 30
データベースの編集 31
テーブルの編集 45
ビューの編集 60
インデックスの編集 87
テンポラリ・テーブルの編集 94
データ整合性の確保 95
データ整合性の概要 96
カラム・デフォルトの使い方 100
テーブル制約とカラム制約の使い方 106
ドメインの使い方 110
エンティティ整合性と参照整合性の確保 113
システム・テーブルの整合性ルール 117
トランザクションと独立性レベル 119
トランザクションの概要 120
独立性レベルと一貫性 125
トランザクションのブロックとデッドロック 140
独立性レベルの選択 143
独立性レベルのチュートリアル 147

Copyright © 2007, iAnywhere Solutions, Inc.

ロックの仕組み 166
特殊な同時性の問題 181
まとめ 183
チュートリアル：SQL Anywhere データベースの作成 185
SQL Anywhere データベース作成のチュートリアルの概要 186
レッスン 1：データベース・ファイルの作成 187
レッスン 2：データベースへの接続 188
レッスン 3：テーブルの作成 190
レッスン 4：カラムのプロパティ設定 192
レッスン 5：外部キーを使用したテーブル間の関係作成 194
まとめ 195

II. データベース・パフォーマンスのモニタリングと改善 197

パフォーマンスのモニタリングと改善 199
パフォーマンスのモニタリングと改善の概要 200
アプリケーション・プロファイリング 202
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 216
その他の診断ツールと方法 235
データベースのパフォーマンスのモニタリング 243
パフォーマンス向上のためのヒント 257

III. データのクエリと変更 281

クエリ：テーブルからのデータの選択 283
クエリの概要 284
SELECT リスト：カラムの指定 287
FROM 句：テーブルの指定 295
WHERE 句：ローの指定 296
ORDER BY 句：結果の順序付け 308
データの要約 311
クエリ結果の要約、グループ化、ソート 315
集合関数を使用したクエリ結果の要約 316
GROUP BY 句：クエリ結果のグループへの編成 321
GROUP BY 句の概要 323
HAVING 句：データ・グループの選択 327

Copyright © 2007, iAnywhere Solutions, Inc.

ORDER BY 句：クエリ結果のソート 329
UNION、INTERSECT、EXCEPT を使用した、クエリ結果に対する集合操作の実行 332
標準と互換性 338
ジョイン：複数テーブルからのデータ検索 341
概要 342
サンプル・データベース・スキーマ 344
ジョイン操作 346
ジョインの概要 347
明示的なジョイン条件 (ON 句) 352
クロス・ジョイン 356
内部ジョインと外部ジョイン 358
特殊なジョイン 365
ナチュラル・ジョイン 373
キー・ジョイン 377
共通テーブル式 391
共通テーブル式の概要 392
共通テーブル式の一般的な使用例 395
再帰共通テーブル式 398
構成部品の問題 400
再帰共通テーブル式でのデータ型宣言 404
最短距離の問題 406
複数の再帰共通テーブル式の使用 410
OLAP のサポート 413
OLAP 機能の概要 414
GROUP BY 句の拡張 416
Window 関数 425
サブクエリの使用 461
サブクエリの概要 462
WHERE 句でのサブクエリの使用 468
HAVING 句でのサブクエリ 469
サブクエリ比較テスト 471
ANY と ALL を使用した限定比較テスト 472
IN 条件によるセット・メンバシップのテスト 475
存在テスト 477

Copyright © 2007, iAnywhere Solutions, Inc.

外部参照 479
サブクエリとジョイン 480
ネストされたサブクエリ 482
サブクエリ操作 484
データの追加、変更、削除 493
データ修正文 494
INSERT によるデータの追加 499
UPDATE によるデータの変更 504
INSERT によるデータの変更 506
DELETE によるデータの削除 507
クエリの最適化と実行 509
クエリ処理のフェーズ 510
セマンティック・クエリ変形 512
オプティマイザの仕組み 523
クエリ実行アルゴリズム 546
実行プランの解釈 571
物理データの編成とアクセス 590

IV. SQL のダイアレクトと互換性 601

他の SQL ダイアレクト 603
SQL Anywhere の準拠の概要 604
SQL FLAGGER を使用した SQL 準拠のテスト 605
他の SQL ソフトウェアにはない機能 608
Transact-SQL との互換性 610
Adaptive Server のアーキテクチャ 613
Transact-SQL との互換性を意識したデータベースの設定 619
互換性のある SQL 文の記述方法 626
Transact-SQL のプロシージャ言語の概要 631
ストアド・プロシージャの自動変換 634
Transact-SQL プロシージャから返される結果セット 635
Transact-SQL プロシージャの中の変数 636
Transact-SQL プロシージャでのエラー処理 637

V. データベースにおける XML 639

Copyright © 2007, iAnywhere Solutions, Inc.

データベースにおける XML の使用 641
XML の概要 642
リレーショナル・データベースにおける XML 文書の格納 643
リレーショナル・データを XML としてエクスポートする 644
XML 文書をリレーショナル・データとしてインポートする 645
クエリ結果を XML として取得する 652
SQL/XML を使用してクエリ結果を XML として取得する 670

VI. リモート・データとバルク・オペレーション 679

データのインポートとエクスポート 681
データベースとの間でのデータの転送 682
データのインポート 684
データベースからのデータのエクスポート 694
データベースの再構築 706
データベースの抽出 715
SQL Anywhere へのデータベースの移行 716
SQL コマンド・ファイルの使用 720
Adaptive Server Enterprise の互換性 723
リモート・データへのアクセス 725
リモート・データ・アクセスの概要 726
リモート・データ・アクセスの基本概念 727
リモート・サーバの使用 729
ディレクトリ・アクセス・サーバの使用 734
外部ログインの使用 737
プロキシ・テーブルの編集 739
リモート・テーブルのジョイン 743
複数のローカル・データベースのテーブルのジョイン 745
ネイティブ文のリモート・サーバへの送信 746
リモート・プロシージャ・コール (RPC) の使用 747
トランザクションの管理とリモート・データ 750
内部オペレーション 752
リモート・データ・アクセスのトラブルシューティング 756
リモート・データ・アクセスのサーバ・クラス 759
リモート・データ・アクセスのサーバ・クラスの概要 760

JDBC ベースのサーバ・クラス 761
ODBC ベースのサーバ・クラス 764

VII. ストアド・プロシージャとトリガ 775

プロシージャ、トリガ、バッチの使用 777
プロシージャとトリガの概要 778
プロシージャとトリガの利点 779
プロシージャの概要 780
ユーザ定義関数の概要 787
トリガの概要 790
バッチの概要 799
制御文 802
プロシージャとトリガの構造 805
プロシージャから返される結果 808
プロシージャとトリガでのカーソルの使用 813
プロシージャとトリガでのエラーと警告 817
プロシージャでの EXECUTE IMMEDIATE 文の使用 825
プロシージャとトリガでのトランザクションとセーブポイント 826
プロシージャを作成するときのヒント 827
プロシージャ、トリガ、イベント、バッチで使用できる文 829
プロシージャからの外部ライブラリの呼び出し 830
プロシージャ、関数、トリガ、イベントのデバッグ 837
SQL Anywhere のデバッガの概要 838
チュートリアル：デバッガの使用開始 840
ブレークポイントの活用 845
変数の編集 847
接続の活用 848
索引 849

Copyright © 2007, iAnywhere Solutions, Inc.

はじめに

このマニュアルの内容

このマニュアルでは、データベースの設計と作成の方法、データのインポート・エクスポート・変更の方法、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。

対象読者

このマニュアルは、SQL Anywhere のすべてのユーザを対象としています。

始める前に

このマニュアルは、読者にデータベース管理システムや SQL Anywhere についての基礎知識があることを前提としています。慣れていない方は、『*SQL Anywhere 10 - 紹介*』を読んでから、このマニュアルを読んでください。

SQL Anywhere のマニュアル

このマニュアルは、SQL Anywhere のマニュアル・セットの一部です。この項では、マニュアル・セットに含まれる各マニュアルと使用法について説明します。

SQL Anywhere のマニュアル

SQL Anywhere の完全なマニュアルは、各マニュアルをまとめたオンライン形式とマニュアル別の PDF ファイルで提供されます。いずれの形式のマニュアルも、同じ情報が含まれ、次のマニュアルから構成されます。

- **『SQL Anywhere 10 - 紹介』**　このマニュアルでは、データの管理および交換機能を提供する包括的なパッケージである SQL Anywhere 10 について説明します。SQL Anywhere を使用すると、サーバ環境、デスクトップ環境、モバイル環境、リモート・オフィス環境に適したデータベース・ベースのアプリケーションを迅速に開発できるようになります。
- **『SQL Anywhere 10 - 変更点とアップグレード』**　このマニュアルでは、SQL Anywhere 10 とそれ以前のバージョンに含まれる新機能について説明します。
- **『SQL Anywhere サーバ - データベース管理』**　このマニュアルでは、SQL Anywhere データベースの実行、管理、設定について説明します。管理ユーティリティとオプションのほか、データベース接続、データベース・サーバ、データベース・ファイル、バックアップ・プロシージャ、セキュリティ、高可用性、Replication Server を使用したレプリケーションについて説明します。
- **『SQL Anywhere サーバ - SQL の使用法』**　このマニュアルでは、データベースの設計と作成の方法、データのインポート・エクスポート・変更の方法、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。
- **『SQL Anywhere サーバ - SQL リファレンス』**　このマニュアルは、SQL Anywhere で使用する SQL 言語の完全なリファレンスです。また、SQL Anywhere のシステム・ビューとシステム・プロシージャについても説明しています。
- **『SQL Anywhere サーバ - プログラミング』**　このマニュアルでは、C、C++、Java プログラミング言語、Visual Studio .NET を使用してデータベース・アプリケーションを構築、配備する方法について説明します。Visual Basic や PowerBuilder などのツールのユーザは、それらのツールのプログラミング・インタフェースを使用できます。
- **『SQL Anywhere 10 - エラー・メッセージ』**　このマニュアルでは、SQL Anywhere エラー・メッセージの完全なリストを、その診断情報とともに説明します。
- **『Mobile Link - クイック・スタート』**　このマニュアルでは、セッションベースのリレーショナル・データベース同期システムである Mobile Link について説明します。Mobile Link テクノロジは、双方向レプリケーションを可能にし、モバイル・コンピューティング環境に非常に適しています。
- **『Mobile Link - サーバ管理』**　このマニュアルでは、Mobile Link アプリケーションを設定して管理する方法について説明します。

Copyright © 2007, iAnywhere Solutions, Inc.

- **『Mobile Link - クライアント管理』** このマニュアルでは、Mobile Link クライアントを設定、構成、同期する方法について説明します。Mobile Link クライアントには、SQL Anywhere または Ultra Light のいずれかのデータベースを使用できます。
- **『Mobile Link - サーバ起動同期』** このマニュアルでは、Mobile Link のサーバによって開始される同期について説明します。サーバによって開始される同期とは、統合データベースから同期またはその他のリモート・アクションの開始を可能にする Mobile Link の機能です。
- **『QAnywhere』** このマニュアルでは QAnywhere について説明します。QAnywhere は、従来のデスクトップ・クライアントやラップトップ・クライアント用のメッセージング・プラットフォームであるほか、モバイル・クライアントや無線クライアント用のメッセージング・プラットフォームでもあります。
- **『SQL Remote』** このマニュアルでは、モバイル・コンピューティング用の SQL Remote データ・レプリケーション・システムについて説明します。このシステムによって、SQL Anywhere の統合データベースと複数の SQL Anywhere リモート・データベースの間で、電子メールやファイル転送などの間接的リンクを使用したデータ共有が可能になります。
- **『SQL Anywhere 10 - コンテキスト別ヘルプ』** このマニュアルには、[接続] ダイアログ、クエリ・エディタ、Mobile Link モニタ、SQL Anywhere コンソール・ユーティリティ、インデックス・コンサルタント、Interactive SQL のコンテキスト別のヘルプが収録されています。
- **『Ultra Light - データベース管理とリファレンス』** このマニュアルでは、小型デバイス用 Ultra Light データベース・システムの概要を説明します。
- **『Ultra Light - AppForge プログラミング』** このマニュアルでは、Ultra Light for AppForge について説明します。Ultra Light for AppForge を使用すると、Palm OS、Symbian OS、または Windows CE を搭載しているハンドヘルド、モバイル、または埋め込みデバイスに対してデータベース・アプリケーションを開発、配備できます。
- **『Ultra Light - .NET プログラミング』** このマニュアルでは、Ultra Light.NET について説明します。Ultra Light.NET を使用すると、PC、ハンドヘルド、モバイル、埋め込みデバイスのデータベース・アプリケーションを開発し、これらのデバイスに配備できます。
- **『Ultra Light - M-Business Anywhere プログラミング』** このマニュアルは、Ultra Light for M-Business Anywhere について説明します。Ultra Light for M-Business Anywhere を使用すると、Palm OS、Windows CE、または Windows XP を搭載しているハンドヘルド、モバイル、または埋め込みデバイスに対して Web ベースのデータベース・アプリケーションを開発、配備できます。
- **『Ultra Light - C/C++ プログラミング』** このマニュアルでは、Ultra Light C および Ultra Light C++ のプログラミング・インタフェースについて説明します。Ultra Light を使用すると、ハンドヘルド、モバイル、埋め込みデバイスに対してデータベース・アプリケーションを開発、配備できます。

マニュアルの形式

SQL Anywhere のマニュアルは、次の形式で提供されています。

- **オンライン・マニュアル** オンライン・マニュアルには、SQL Anywhere の完全なマニュアルがあり、SQL Anywhere ツールに関する印刷マニュアルとコンテキスト別のヘルプの両方が含

まれています。オンライン・マニュアルは、製品のメンテナンス・リリースごとに更新されます。これは、最新の情報を含む最も完全なマニュアルです。

Windows オペレーティング・システムでオンライン・マニュアルにアクセスするには、[スタート] − [プログラム] − [SQL Anywhere 10] − [オンライン・マニュアル] を選択します。オンライン・マニュアルをナビゲートするには、左ウィンドウ枠で HTML ヘルプの目次、索引、検索機能を使用し、右ウィンドウ枠でリンク情報とメニューを使用します。

UNIX オペレーティング・システムでオンライン・マニュアルにアクセスするには、SQL Anywhere のインストール・ディレクトリまたはインストール CD に保存されている HTML マニュアルを参照してください。

- **PDF ファイル**　SQL Anywhere の完全なマニュアル・セットは、Adobe Reader で表示できる Adobe Portable Document Format (pdf) 形式のファイルとして提供されています。

  Windows では、PDF 形式のマニュアルはオンライン・マニュアルの各ページ上部にある PDF のリンクから、または Windows の [スタート] メニュー ([スタート] − [プログラム] − [SQL Anywhere 10] − [オンライン・マニュアル - PDF フォーマット]) からアクセスできます。

  UNIX では、PDF 形式のマニュアルはインストール CD にあります。

Copyright © 2007, iAnywhere Solutions, Inc.

表記の規則

この項では、このマニュアルで使用されている書体およびグラフィック表現の規則について説明します。

SQL 構文の表記規則

SQL 構文の表記には、次の規則が適用されます。

- **キーワード**　SQL キーワードはすべて次の例に示す ALTER TABLE のように大文字で表記します。

  ALTER TABLE [*owner*.]*table-name*

- **プレースホルダ**　適切な識別子または式で置き換えられる項目は、次の例に示す *owner* や *table-name* のように表記します。

  ALTER TABLE [*owner*.]*table-name*

- **繰り返し項目**　繰り返し項目のリストは、次の例に示す *column-constraint* のように、リストの要素の後ろに省略記号 (ピリオド 3 つ …) を付けて表します。

  ADD *column-definition* [*column-constraint*, …]

  複数の要素を指定できます。複数の要素を指定する場合は、各要素間をカンマで区切る必要があります。

- **オプション部分**　文のオプション部分は角カッコで囲みます。

  RELEASE SAVEPOINT [*savepoint-name*]

  この例では、角カッコで囲まれた *savepoint-name* がオプション部分です。角カッコは入力しないでください。

- **オプション**　項目リストから 1 つだけ選択する場合や、何も選択しなくてもよい場合は、項目間を縦線で区切り、リスト全体を角カッコで囲みます。

  [**ASC** | **DESC**]

  この例では、ASC と DESC のどちらか 1 つを選択しても、選択しなくてもかまいません。角カッコは入力しないでください。

- **選択肢**　オプションの中の 1 つを必ず選択しなければならない場合は、選択肢を中カッコで囲み、縦棒で区切ります。

  [**QUOTES** { **ON** | **OFF** }]

  QUOTES オプションを使用する場合は、ON または OFF のどちらかを選択する必要があります。角カッコと中カッコは入力しないでください。

Copyright © 2007, iAnywhere Solutions, Inc.

オペレーティング・システムの表記規則

- **Windows**　デスクトップおよびラップトップ・コンピュータ用の Microsoft Windows オペレーティング・システムのファミリのことです。Windows ファミリには Windows Vista や Windows XP も含まれます。
- **Windows CE**　Microsoft Windows CE モジュラ・オペレーティング・システムに基づいて構築されたプラットフォームです。Windows Mobile や Windows Embedded CE などのプラットフォームが含まれます。

  Windows Mobile は Windows CE 上に構築されています。これにより、Windows のユーザ・インタフェースや、Word や Excel といったアプリケーションの小規模バージョンなどの追加機能が実現されています。Windows Mobile は、モバイル・デバイスで最も広く使用されています。

  SQL Anywhere の制限事項や相違点は、基盤となっているオペレーティング・システム (Windows CE) に由来しており、使用しているプラットフォーム (Windows Mobile など) に依存していることはほとんどありません。
- **UNIX**　特に記述がないかぎり、UNIX は Linux プラットフォームと UNIX プラットフォームの両方のことです。

ファイルの命名規則

マニュアルでは、パス名やファイル名などのオペレーティング・システムに依存するタスクと機能を表すときは、通常 Windows の表記規則が使用されます。ほとんどの場合、他のオペレーティング・システムで使用される構文に簡単に変換できます。

- **ディレクトリ名とパス名**　マニュアルでは、ドライブを示すコロンや、ディレクトリの区切り文字として使用する円記号など、Windows の表記規則を使用して、ディレクトリ・パスのリストを示します。次に例を示します。

  MobiLink¥redirector

  UNIX、Linux、Mac OS X では、代わりにスラッシュを使用してください。次に例を示します。

  MobiLink/redirector

  SQL Anywhere がマルチプラットフォーム環境で使用されている場合、プラットフォーム間でのパス名の違いに注意する必要があります。
- **実行ファイル**　マニュアルでは、実行ファイルの名前は、Windows の表記規則が使用され、拡張子 *.exe* が付きます。UNIX、Linux、Mac OS X では、実行ファイルの名前には拡張子は付きません。NetWare では、実行ファイルの名前には、拡張子 *.nlm* が付きます。

  たとえば、Windows では、ネットワーク・データベース・サーバは *dbsrv10.exe* です。UNIX、Linux、Mac OS X では、*dbsrv10* になります。NetWare では、*dbsrv10.nlm* になります。
- **install-dir**　インストール・プロセスでは、SQL Anywhere をインストールするロケーションを選択できます。マニュアルでは、このロケーションは *install-dir* という表記で示されます。

Copyright © 2007, iAnywhere Solutions, Inc.

インストールが完了すると、環境変数 SQLANY10 によって SQL Anywhere コンポーネントがあるインストール・ディレクトリのロケーション (*install-dir*) が指定されます。SQLANYSH10 は、SQL Anywhere が他の Sybase アプリケーションと共有しているコンポーネントがあるディレクトリのロケーションを指定します。

オペレーティング・システム別の *install-dir* のデフォルト・ロケーションの詳細については、「SQLANY10 環境変数」『SQL Anywhere サーバ - データベース管理』を参照してください。

- **samples-dir**　インストール・プロセスでは、SQL Anywhere に含まれるサンプルをインストールするロケーションを選択できます。マニュアルでは、このロケーションは *samples-dir* という表記で示されます。

  インストールが完了すると、環境変数 SQLANYSAMP10 によってサンプルがあるディレクトリのロケーション (*samples-dir*) が指定されます。Windows の [スタート] メニューから、[プログラム] − [SQL Anywhere 10] − [サンプル・アプリケーションおよびプロジェクト] を選択すると、このディレクトリで [Windows エクスプローラ] ウィンドウが表示されます。

  オペレーティング・システム別の *samples-dir* のデフォルト・ロケーションの詳細については、「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。

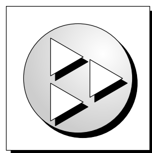
- **環境変数**　マニュアルでは、環境変数設定が引用されます。Windows では、環境変数を参照するのに、構文 *%envvar%* が使用されます。UNIX、Linux、Mac OS X では、環境変数を参照するのに、構文 *$envvar* または *${envvar}* が使用されます。

  UNIX、Linux、Mac OS X 環境変数は、*.cshrc* や *.tcshrc* などのシェルとログイン・スタートアップ・ファイルに格納されます。

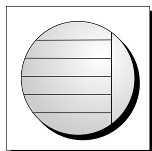
グラフィック・アイコン

このマニュアルでは、次のアイコンを使用します。

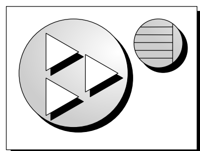
- クライアント・アプリケーション



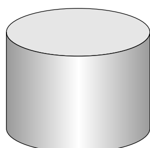
- SQL Anywhere などのデータベース・サーバ



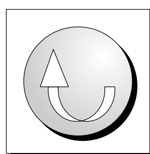
- Ultra Light アプリケーション



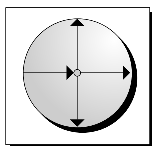
- データベース。高度な図では、データベースとデータベースを管理するデータ・サーバの両方をこのアイコンで表します。



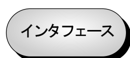
- レプリケーションまたは同期のミドルウェア。ソフトウェアのこれらの部分は、データベース間のデータ共有を支援します。たとえば、Mobile Link サーバ、SQL Remote Message Agent などが挙げられます。



- Sybase Replication Server



- プログラミング・インタフェース



Copyright © 2007, iAnywhere Solutions, Inc.

詳細情報の検索／フィードバックの提供

詳細情報の検索

詳しい情報やリソース (コード交換など) については、iAnywhere Developer Network (http://www.ianywhere.com/developer/) を参照してください。

ご質問がある場合や支援が必要な場合は、次に示す Sybase iAnywhere ニュースグループのいずれかにメッセージをお寄せください。

ニュースグループにメッセージをお送りいただく際には、ご使用の SQL Anywhere バージョンのビルド番号を明記し、現在発生している問題について詳しくお知らせくださいますようお願いいたします。バージョン情報は、コマンド・プロンプトで **dbeng10 -v** と入力して確認できます。

ニュースグループは、ニュース・サーバ *forums.sybase.com* にあります (ニュースグループにおけるサービスは英語でのみの提供となります)。以下のニュースグループがあります。

- sybase.public.sqlanywhere.general
- sybase.public.sqlanywhere.linux
- sybase.public.sqlanywhere.mobilink
- sybase.public.sqlanywhere.product_futures_discussion
- sybase.public.sqlanywhere.replication
- sybase.public.sqlanywhere.ultralite
- ianywhere.public.sqlanywhere.qanywhere

ニュースグループに関するお断り
iAnywhere Solutions は、ニュースグループ上に解決策、情報、または意見を提供する義務を負うものではありません。また、システム・オペレータ以外のスタッフにこのサービスを監視させて、操作状況や可用性を保証する義務もありません。
iAnywhere のテクニカル・アドバイザとその他のスタッフは、時間のある場合にかぎりニュースグループでの支援を行います。こうした支援は基本的にボランティアで行われるため、解決策や情報を定期的に提供できるとはかぎりません。支援できるかどうかは、スタッフの仕事量に左右されます。

フィードバック

このマニュアルに関するご意見、ご提案、フィードバックをお寄せください。

マニュアルに関するご意見、ご提案は、SQL Anywhere ドキュメンテーション・チームの iasdoc@ianywhere.com 宛てに電子メールでお寄せください。このアドレスに送信された電子メールに返信はいたしませんが、お寄せいただいたご意見、ご提案は必ず読ませていただきます。

マニュアルまたはソフトウェアについてのフィードバックは、上記のニュースグループを通してお寄せいただいてもかまいません。

Copyright © 2007, iAnywhere Solutions, Inc.

パート I. データベースの設計と作成

パート I では、データベースの設計と構築に関わる主要な概念と方式について説明します。データベースの設計、およびテーブル、ビュー、インデックスを扱う技術を取り上げます。また、参照整合性やトランザクションについても説明します。

第 1 章

データベースの設計

目次

データベースの設計の概要 4
データベース設計の概念 5
設計のプロセス 11
テーブル・プロパティの設計 24

データベースの設計の概要

中規模までのデータベースの設計は、難しくはありませんが、非常に重要な作業です。設計が良くないと、データベース・システムは非効率的で信頼性の低いものになってしまいます。クライアント・アプリケーションはデータベースの特定の部分に対して処理を行うように構築されており、データベースの設計に依存しています。このため、設計のよくないデータベースをあとから修正するのは困難です。

詳細については、データベース設計の入門書も参考にしてください。

 Copyright © 2007, iAnywhere Solutions, Inc.

データベース設計の概念

データベースを設計する上で、何についての情報を格納するか、それぞれについてどの情報を保持するかを計画します。また、これらの情報がどのように関連しているかを調べます。データベース設計の一般的な用語では、この手順で作成するものを「**概念データベース・モデル**」といいます。

エンティティと関係

情報を格納する対象となる、識別可能なオブジェクトまたは物事を「**エンティティ**」と呼びます。エンティティの間の結合は「**関係**」と呼びます。データベース記述言語では、エンティティを名詞、関係を動詞と捉えることができます。

概念モデルは、エンティティと関係の間の識別を明確にするので便利です。これらのモデルは、特定のデータベース管理システムに設計を実装するときに伴う細部が表示されないようにします。これによって、ユーザは基本的なデータベース構造に焦点をしぼることができます。また、概念モデルは、データベース設計のための共通言語を形成します。

ER (実体関連) 図

概念データベース・モデルの主なコンポーネントは、エンティティとの関係を表す図です。この図は一般に「**ER (実体関連) 図**」と呼ばれています。多くの人が概念データベース・モデルを作成する作業のことを ER モデリングと呼びます。

概念データベース設計はトップ・ダウン設計方法です。Sybase PowerDesigner などのツールは、この方法を実行するのに役立ちます。この章では紹介にとどめますが、実際には簡単なデータベースを設計するのに必要な情報が含まれています。

エンティティ

「**エンティティ**」は、データベースにおける名詞の役割を果たします。従業員、注文品目、部署、製品などの個別オブジェクトはすべてエンティティの例です。データベースでは、テーブルが各エンティティを表します。データベースに構築されるエンティティは、客先訪問の記録や、従業員データの管理など、データベースを使用する活動 (アクティビティ) から生じるものです。

属性

各エンティティには多くの「**属性**」があります。属性は、格納する対象に固有の特性です。たとえば、従業員エンティティでは、従業員 ID、姓と名前、住所、その他の特定の従業員に関する情報などを格納します。属性はプロパティともいいます。

エンティティは四角いボックスで表します。中には、エンティティに関連する属性をリストします。

Employee
Employee Number First Name Last Name Address

「**識別子 (identifier)**」 は他のすべての属性が依存する 1 つ以上の属性です。エンティティの中の項目をユニークに識別します。識別子となる属性の名前に下線を引きます。

図の Employee エンティティでは、従業員番号 (Employee Number) が従業員をユニークに識別します。他の属性は、すべてある従業員にだけ関係する情報を格納しています。たとえば、従業員番号は従業員の名前 (Name) や住所 (Address)をユニークに決定します。同じ名前や同じ住所の従業員が 2 人いるかもしれませんが、同じ従業員番号ではありません。識別子であることを示すために、Employee Number に下線を引きます。

各エンティティの識別子を作成すると実用的です。あとで説明しますが、これらの識別子はテーブルのプライマリ・キーになります。プライマリ・キーの値はユニークである必要があり、NULL または未定義にはできません。プライマリ・キーはテーブルの各ローをユニークに識別し、データベース・サーバのパフォーマンスを改善します。

関係

エンティティ間の「**関係**」は、データベースにおける動詞の役割を果たします。従業員は部署のメンバであり、事業所は都市にあります。後述するとおり、データベース内の関係はテーブル間の外部キー関係として表示されることも、それ自体が独立したテーブルとして表示されることもあります。

データベースの関係は、エンティティのデータを管理する規則や習慣の符号化です。各部署に部長が 1 人いる場合、部長を識別する、部署と従業員の間の 1 対 1 の関係を作成できます。

一度データベースの構造に関係が組み込まれると、例外はあり得ません。部長をもう 1 人入力することはできません。部署のエントリの重複は、識別子である部署 ID の重複にも関係します。識別子の重複は許可されません。

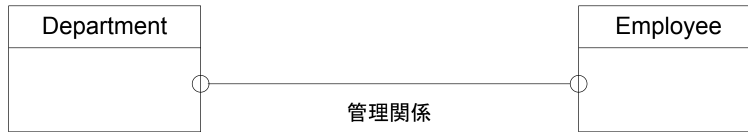
ヒント
厳密なデータベース構造は、部署に 2 人マネージャがいるなどの矛盾を排除できるので便利です。一方で、設計者としては、予測していない使用があった場合のために、多少の拡張ができる程度に柔軟に設計してください。設計が良くできているデータベースを拡張するのは、通常それほど難しくありませんが、既存のテーブル構造を変更すると、データベース全体およびそのクライアント・アプリケーションが使えなくなるおそれがあります。

関係のカーディナリティ

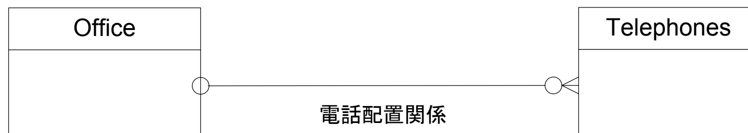
テーブルの間には 3 種類の関係があります。これは関係に関わるエンティティの「**カーディナリティ (数)**」に相当します。

- **1 対 1 の関係**　2 つのエンティティを線で結んで関係を示します。線上には 2 つの小さい円などの他のマークがあります。あとの項でこれらのマークの目的を説明します。次の図では、1 人の従業員が 1 つの部署を管理しています。

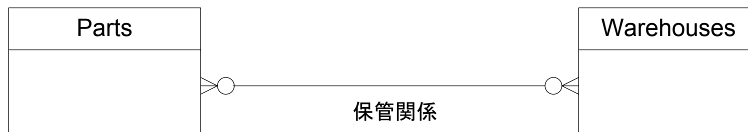
Copyright © 2007, iAnywhere Solutions, Inc.



- **1 対多の関係**　エンティティ 2 と結ばれた複数の線は、エンティティ 1 の 1 つの項目がエンティティ 2 の複数のエンティティに関連付けられるということを示しています。次の図では、1 つの事業所に多数の電話があります。



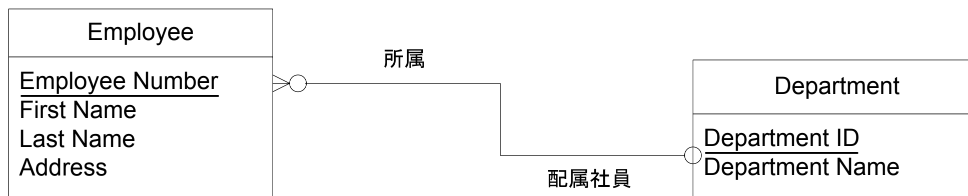
- **多対多の関係**　この関係では、両方のエンティティへの接続を示す複数の線を引きます。つまり、1 つの倉庫に各種部品を保管でき、複数の倉庫に 1 種類の部品を保管できます。



役割

各関係は 2 つの「**役割**」を使用して表すことができます。役割はそれぞれの視点から関係を説明する動詞または句です。たとえば、従業員と部署の間の関係は次の 2 つの役割で説明できます。

1. 従業員は部署の*メンバ*です。
2. 部署は従業員を*含みます*。



役割は非常に重要です。これは作業を検証する便利で効率的な手段を提供するためです。

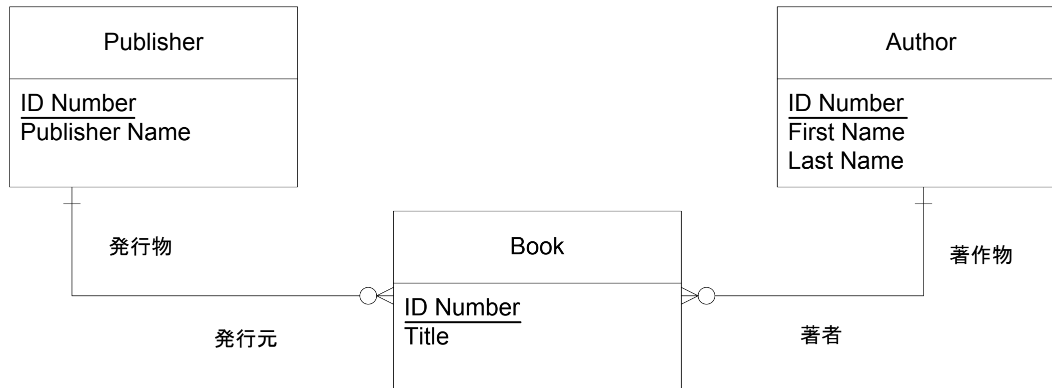
Copyright © 2007, iAnywhere Solutions, Inc.

ヒント
左から右に読む場合も、右から左に読む場合も、次のルールに従うことでこれらの図を簡単に読むことができます。最初のエンティティの名前を読み込み、最初のエンティティの横にある役割を読み込み、2 つめのエンティティへの接続のカーディナリティを読み込んで、2 つめのエンティティの名前を読み込みます。
上の図で、左から右に読み込むと、各従業員は 1 つの部門の構成メンバになります。右から左に読み込むと、1 つの部門には複数の従業員が含まれます。

必須要素

関係を示す線の端のすぐ手前にある小さい円は、重要な目的を示すものです。円は対応する要素がもう一方のエンティティになくても、あるエンティティ内で要素が存在できることを意味しています。

円の場所に横棒がある場合は、エンティティにはもう一方のエンティティの要素ごとに*少なくとも* 1 つの要素があります。例でこれらの文について説明します。



この図は次の 4 つの文に対応しています。

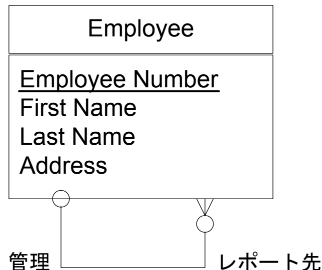
1. 1 つの出版社は *0 または 1 冊以上の*本を出版します。
2. 1 冊の本は *1 つの*出版社のみから出版されます。
3. 1 冊の本は *1 人または複数の*作家によって書かれます。
4. 1 人の作家は *0 または 1 冊以上の*本を書きます。

ヒント
小さい円を数字の 0、横棒を数字の 1 と考えます。円は*最低 0* を表します。横棒は*最低 1* を表します。

 Copyright © 2007, iAnywhere Solutions, Inc.

再帰関係

場合によっては、単一のエンティティ内のエントリ間で関係が存在することがあります。このような場合、関係を「**再帰**」と呼びます。関係の両端は単一のエンティティに付加されます。



この図は次の 2 つの文に対応します。

1. 1 人の従業員は最大 1 人の他の従業員にレポートします。
2. 1 人の従業員は 0 人以上の従業員を管理します。

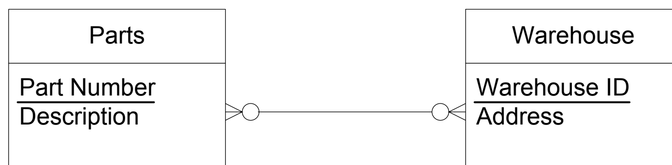
この関係の場合、両方向において関係が必然的に任意になることに注意してください。マネージャでない従業員もいます。同様に、少なくとも 1 人の従業員は組織の長で、だれの監督下にもありません。

当然、従業員は自分自身のマネージャにはなれません。この制限は一種の *ビジネス・ルール* です。ビジネス・ルールについては、後述の「設計のプロセス」 11 ページを参照してください。

多対多の関係をエンティティに変更

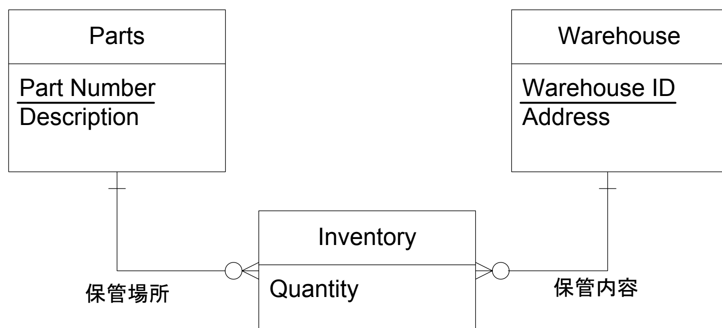
エンティティにではなく*関係*に関連する属性がある場合は、関係をエンティティに変更できます。このような状況は、関係に特有の属性があり、それをどちらのエンティティにも合理的に追加できない場合に、多対多の関係でしばしば起こります。

部品の在庫が複数の倉庫に置かれているとします。この場合、次のような図を描くことができます。



しかし、それぞれの場所に格納されている各部品数を記録したいとします。この属性は関係にだけ関連付けできます。各部品数は、関係する部品と倉庫の両方に依存します。この状況を表すには、次のように図を再作成します。

Copyright © 2007, iAnywhere Solutions, Inc.



変換に関して次の点に注意してください。

1. 2 つの新しい関係は、関係エンティティと 2 つの元のエンティティのそれぞれをジョインします。これらは元の関係の 2 つの役割から名前を継承して、それぞれ*保管場所*と*保管内容*にします。

2. Inventory エンティティの各エントリには、Part エンティティの必須エントリが 1 つと Warehouse エンティティの必須エントリが 1 つ必要です。在庫関係は特定の部品と特定の倉庫と関連付けられなければ意味がないため、これらの関係が必須になります。

3. 新しいエンティティは Part エンティティと Warehouse エンティティの両方に依存します。これは、新しいエンティティが両方のエンティティの識別子によって識別されるということです。この新しい図では、Part エンティティからのある識別子と Warehouse エンティティからのある識別子が Inventory エンティティのエントリをユニークに識別します。2 つの新しい関係を新しい Inventory エンティティにジョインする、円と複数の線の間に表示される三角は依存性を表します。

Inventory エンティティに Part Number 属性、または Warehouse ID 属性を追加しないでください。Inventory エンティティの各エントリは、特定の部品と特定の倉庫の両方に依存していますが、三角はこの依存をより明確に表しています。

Copyright © 2007, iAnywhere Solutions, Inc.

設計のプロセス

設計には主に次に示す 5 つの手順があります。

- 「手順 1：エンティティと関係を決定する」 11 ページ
- 「手順 2：必要なデータを決定する」 13 ページ
- 「手順 3：データを正規化する」 15 ページ
- 「手順 4：関係を解析する」 19 ページ
- 「手順 5：設計を検証する」 22 ページ

データベース設計の実装の詳細については、「データベース・オブジェクトの使用」 29 ページを参照してください。

手順 1：エンティティと関係を決定する

- **設計のエンティティとそれらの間の関係を決定するには、次の手順に従います。**

1. **上位レベルのアクティビティの定義**　このデータベースを使用する一般的なアクティビティを決定します。たとえば、従業員に関する情報を追跡して、変更があればそれをすぐ利用できるように記録しておくことです。
2. **エンティティを決定する**　作成したアクティビティ・リストから、情報の管理に必要なサブジェクト領域を決定します。これらのサブジェクトがエンティティになります。たとえば*従業員*の採用、*部署*への配属、*スキル*・レベルの決定などです。
3. **関係を決定する**　アクティビティを検討し、エンティティ間の関係がどのようなものか決定します。たとえば、部品と倉庫の間には関係があります。各関係を表す 2 つの役割を定義します。
4. **アクティビティを分解する**　ここでは上位アクティビティを定義しました。ここでは上位レベルのアクティビティが、下位レベルのアクティビティに分解できるかどうかを検討します。たとえば、*従業員データを管理する*という上位レベルのアクティビティは、次に示す下位レベルのアクティビティに分解できます。

    - 新しい従業員の追加
    - 現在の従業員データの更新
    - 退社した従業員データの削除

5. **ビジネス・ルールの決定**　業務内容を検討し、どのようなルールに従っているかを調べます。たとえば、各部署は 1 人の長を持つ、ただし複数の長は持たないというのは 1 つのルールです。これらのルールはデータベースの構造に組み込まれます。

エンティティと関係の例

ACME Corporation は 5 か所に事業所を持つ中規模な企業です。現在従業員数は 75 名です。この企業は急速な成長に備えており、それぞれ長を持つ 9 つの部署があります。

新しい従業員の採用のため、人事部はこれからの同社に必要となる 68 のスキルを決定しました。新規に従業員が採用されたときには、その従業員のスキルごとのスキル・レベルが識別されます。

上位レベルのアクティビティの定義

次に ACME Corporation の上位レベルのアクティビティをいくつか示します。

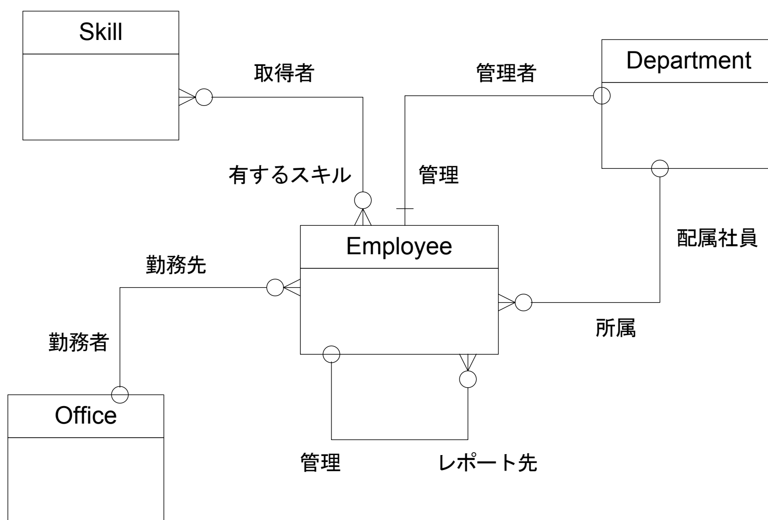
- 従業員の採用
- 従業員の退職の処理
- 個人別の従業員情報の管理
- 企業に必要なスキルの管理
- 各従業員がどのスキルを持っているかの管理
- 部署のデータの管理
- 事業所のデータの管理

エンティティと関係の決定

エンティティ (サブジェクト) とそれらを接続する関係 (役割) を決定します。各サブジェクトの記述と上位レベルのアクティビティから図を作成します。

エンティティは四角で表し、関係は線で表します。2 つの役割を使用して各関係にラベルを付けます。関係が 1 対多、1 対 1、多対多のどれにあたるかも適切な注釈を使って決定します。

次に示すのは、大まかな ER 図です。この例を通じて徐々に詳細なものになります。



 Copyright © 2007, iAnywhere Solutions, Inc.

上位レベルのアクティビティの分解

前述の上位レベルのアクティビティをもとに分解した下位レベルのアクティビティを次に示します。

- 従業員の追加または削除
- 事業所の追加または削除
- 部署の従業員のリスト
- 新しいスキルのスキル・リストへの追加
- 従業員のスキルの決定
- 各スキルに対する従業員のスキル・レベルの決定
- 特定のスキルに関して同一スキル・レベルにある従業員すべての決定
- 従業員のスキル・レベルの変更

これらの下位レベルのアクティビティは、新しくテーブルや関係が必要となるかどうかを決定するのに使用できます。

ビジネス・ルールの決定

ビジネス・ルールは、1 対多、1 対 1、多対多の関係を決定します。

データベースに関係する可能性があるビジネス・ルールを次に示します。

- 現在 5 つの事業所があり、拡張計画は最大 10 の事業所まで可能
- 従業員は部署または事業所を移動できる
- 各部署には 1 人の長がいる
- 各事業所の電話番号は最大 3 つある
- 各電話番号には 1 つまたは複数の内線番号がある
- 新しく従業員が採用された場合には、スキルごとのスキル・レベルが決定される
- 各従業員は 3 から 20 のスキルを持つ
- 従業員は事業所に属すことも属さないこともある

手順 2：必要なデータを決定する

- **必要なデータを決定するには、次の手順に従います。**

1. 基本となるデータを決定します。
2. 追跡する必要があるデータをすべてリストします。
3. 各エンティティのデータを準備します。

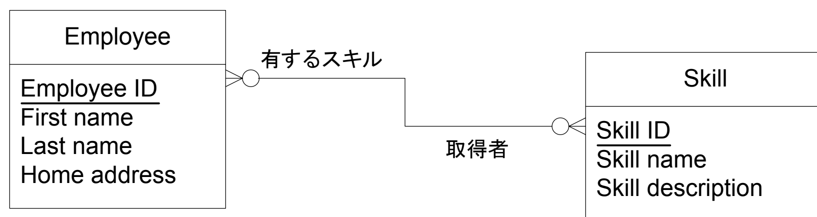
4. 各エンティティで使用可能なデータをリストします。エンティティ (サブジェクト) を記述するデータは、誰が、何を、どこで、いつ、なぜ、の基準を満たすようにします。
5. 各関係 (動詞) に必要なデータをリストします。
6. 各関係のデータがある場合はリストします。

基礎となるデータの決定

決定する基礎データはエンティティの属性の名前になります。たとえば、Employee エンティティ、Skill エンティティ、Expert In 関係に適用されるデータを次に示します。

Employee	**Skill**	**Expert In**
Employee ID	Skill ID	Skill level
Employee first name	Skill name	Date skill was acquired
Employee last name	Description of skill	
Employee department		
Employee office		
Employee address		

この図は、次のようになります。



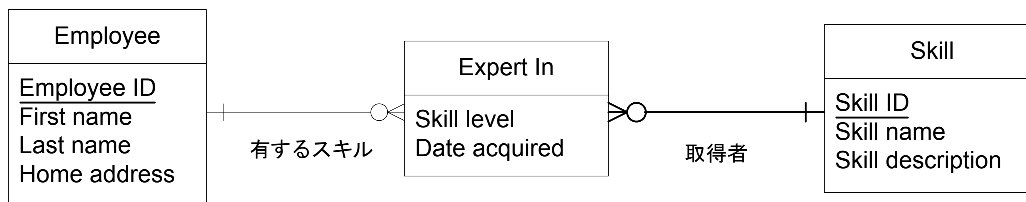
この図には、リストした属性すべてが表示されているわけではない点に注意してください。足りない項目は 2 種類のカテゴリに分かれます。

1. 他の関係に暗黙的に含まれている場合。たとえば、Employee department と Employee office は Department と Office エンティティの関係によってそれぞれ示されます。
2. どのエンティティとも関連しておらず、エンティティ間の関係と関連しているため存在しない場合。この図は不適当です。

項目の 1 つ目のカテゴリは ER 図全体を作成すれば自然に収まります。

この多対多の関係をエンティティに変換すると、2 つ目のカテゴリを追加できます。

Copyright © 2007, iAnywhere Solutions, Inc.



新しいエンティティは、Employee エンティティと Skill エンティティの両方に依存します。このエンティティは両方のエンティティに依存するため、これらのエンティティから識別子を借ります。

注意

- 基礎となるデータを確認するときは、先に確認したアクティビティを参照してデータへのアクセス方法を考慮します。

  たとえば、従業員を姓や名前でソートする必要があるとします。そのためには、データを First Name (名前) 属性と Last Name (姓) 属性に分け、氏名 (姓と名前の両方を含む) という 1 つの属性とはしません。姓と名前に分けることで、各作業に適した 2 つのインデックスを後で作成できます。

- 名前には一貫性を持たせます。一貫性があればデータベースのメンテナンスが簡単で、レポートや出力ウィンドウの内容も読みやすくなります。

  たとえば、従業員の状況属性に Emp_status などの省略名を使用する場合は、別の属性で、Employee_ID などの完全な名前は使用しないようにします。この場合は Emp_status と Emp_ID、としてください。

- この段階では、データはエンティティに最初から正しく関連している必要はありません。適切と思われる入力をしておきます。次の項で、それが正しかったかどうかを確認します。

手順 3：データを正規化する

正規化とはデータの冗長性を排除し、データが正しくエンティティまたは関係に属していることを確認する一連のテストのことです。テストは 5 つあります。この項では、その内の最初の 3 つを説明します。この 3 つのテストは最も重要なので、頻繁に使用されています。

> **正規化が必要な理由**
> 正規化の目的は、冗長性を排除し一貫性を高めることです。たとえば、顧客の住所を複数のロケーションに格納すると、変更があった場合に正しくすべてを更新するのが難しくなります。

正規化テストの詳細については、データベース設計に関連する書籍を参照してください。

正規形

データの正規化には、複数のテストがあります。データが最初のテストに通ればそのデータは第 1 正規形と呼ばれます。2 番目のテストに通れば第 2 正規形、3 番目のテストに通れば第 3 正規形と呼ばれます。

Copyright © 2007, iAnywhere Solutions, Inc.

♦ **データベースのデータを正規化するには、次の手順に従います。**

1. データをリストします。

   ♦ 各エンティティが少なくとも 1 つのキーを持つことを確認します。各エンティティは識別子を持つ必要があります。

   ♦ 関係のキーを確認します。関係のキーとはジョインしている 2 つのエンティティからのキーです。

   ♦ 基本となるデータのリストの中に計算されたデータがないか確認します。通常、計算されたデータはリレーショナル・データベースには格納されません。

2. データを第 1 正規形にします。

   ♦ 属性が同じエントリに対して複数の値を持つ可能性がある場合は、繰り返されるデータを削除します。

   ♦ 削除したデータで 1 つまたは複数のエンティティか関係を作成します。

3. データを第 2 正規形にします。

   ♦ 複数の属性からなるキーを持つエンティティと関係を確認します。

   ♦ 複数のキーのうち、一部にのみ依存するデータを削除します。

   ♦ 削除したデータで 1 つまたは複数のエンティティと関係を作成します。

4. データを第 3 正規形にします。

   ♦ エンティティと関係のキー以外のデータに依存するデータを削除します。

   ♦ 削除したデータで 1 つまたは複数のエンティティと関係を作成します。

データと識別子

データをリストしてテーブルごとの固有の識別子を確認してから、データの正規化 (設計のテスト) を開始します。識別子は 1 つのデータ (属性) または複数のデータ (複合識別子) から構成されます。

識別子はエンティティの中の各ローをユニークに識別する一連の属性です。たとえば、Employee エンティティの識別子は、Employee ID 属性です。Works In 関係の識別子は、Office Code と Employee ID 属性から構成されています。

接続する各エンティティから識別子を取得してデータベースの各関係の識別子を作成できます。次の表は、アスタリスクがついている属性がエンティティまたは関係用の識別子を示しています。

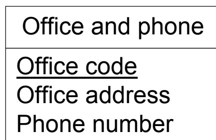
Copyright © 2007, iAnywhere Solutions, Inc.

エンティティまたは*関係*	属性
Office	*Office code Office address Phone number
Works in	*Office code *Employee ID
Department	*Department ID Department name
Heads	*Department ID *Employee ID
Member of	*Department ID *Employee ID
Skill	*Skill ID Skill name Skill description
Expert in	*Skill ID *Employee ID Skill level Date acquired
Employee	*Employee ID Last name First name Social security number Address Phone number Date of birth

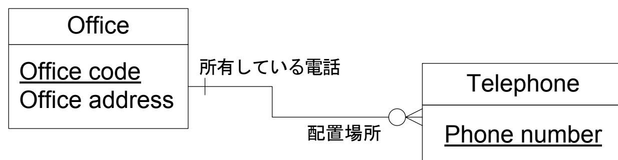
データを第 1 正規形にする

- 第 1 正規形をテストするには、繰り返される値を持つ属性を探します。
- 複数の値が単一の項目に適用できる場合は属性を削除します。これらの繰り返される属性を新しいエンティティに入れます。

次に示す例では、電話番号が繰り返されます。事業所には複数の電話番号があるからです。



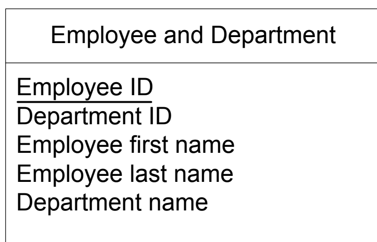
繰り返される属性を削除し、Telephone というエンティティを新しく作成します。Telephone と Office の間に関係を作成します。



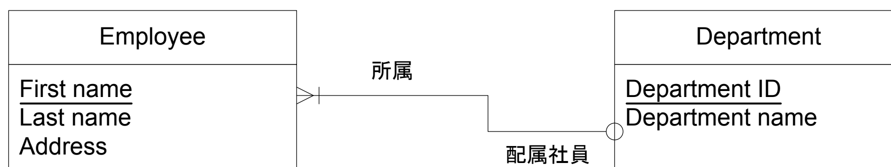
データを第 2 正規形にする

- キー全体に依存しないデータを削除します。
- 複数の属性からなる識別子を持つエンティティと関係にのみ注目します。第 2 正規形をテストするには、識別子全体に依存しないデータを削除します。各属性は、識別子を構成するすべての属性に依存するようにします。

次に示す例では、Employee and Department エンティティは 2 つの属性からなる識別子を持っています。データによっては両方の識別子属性に依存していません。たとえば、department name はその属性のうちの 1 つ Department ID にのみ依存しています。また Employee first name は Employee ID にのみ依存しています。



他の従業員データが依存しない識別子 Department ID を Department というエンティティに移動します。また、その識別子に依存するすべての属性も移動します。Employee と Department の間に関係を作成します。

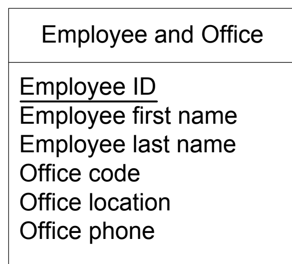


 Copyright © 2007, iAnywhere Solutions, Inc.

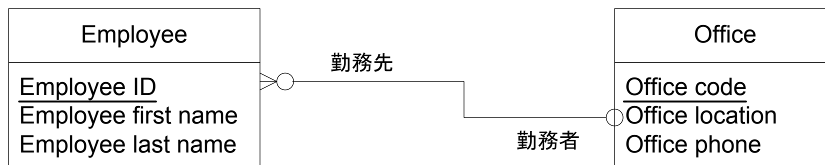
データを第 3 正規形にする

- キーに直接依存しないデータを削除します。
- 第 3 正規形をテストするには、識別子に直接ではなく他の属性に依存する属性を削除します。

この例では、Employee and Office エンティティには識別子 Employee ID に依存する属性がいくつかあります。しかし、Office location や Office phone などの属性は、別の属性である Office code に依存しています。識別子である Employee ID には直接依存しません。



Office code とそれに依存する属性を削除します。Office という新しいエンティティを作成します。そして Employee と Office を接続する関係を作成します。



手順 4：関係を解析する

正規化が終了したら、データベースの設計はほとんど完成です。次に行うことは、概念データ・モデルに対応する「**物理データ・モデル**」の生成です。この作業の大部分が概念データ・モデルの関係を、対応するテーブルと外部キー関係に変換することであるため、関係の解析とも言われます。

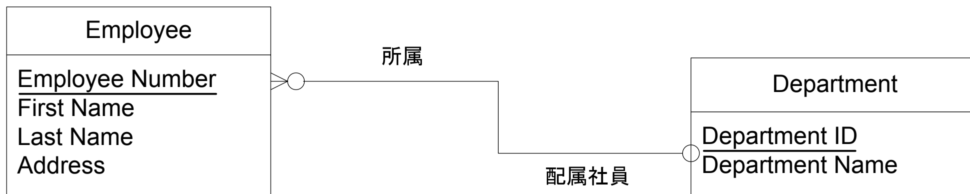
概念データ・モデルは実装作業とはほとんど関係ありませんが、物理データ・モデルはテーブル構造や特定のデータベース・アプリケーションで利用できるオプションと密接に関係しています。ここでのアプリケーションとは SQL Anywhere を指します。

データを含まない関係の解析

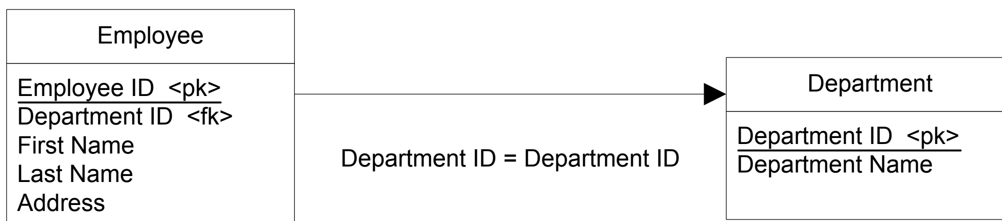
データを含まない関係を実装するには、外部キーを設定します。「**外部キー**」とは、他のテーブルからのプライマリ・キー値を含むカラムまたはカラム・セットです。外部キーを使用すると、複数のテーブルのデータに同時にアクセスできます。

PowerDesigner などのデータベース設計ツールでは、独自の物理データ・モデルを生成できます。ただし、自分で行う場合は、何をキーとするかを決定するのに役立つ基本ルールがあります。

- **1 対多**　1 対多の関係では、常に 1 つのエンティティと 1 つの外部キーの関係になります。

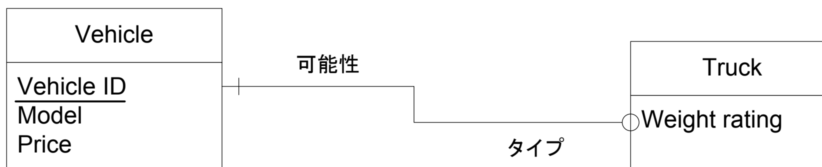


エンティティはテーブルになる点に注意してください。エンティティの識別子はテーブルのプライマリ・キー (または少なくともその一部) になります。属性はカラムになります。1 対多の関係では、*1* つのエンティティの識別子が*複数*のテーブルの新しい外部キー・カラムとして表示されます。

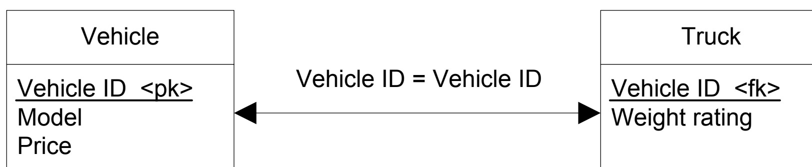


この例では、Employee *エンティティ*が Employees *テーブル*になります。同様に、Department エンティティが Departments テーブルになります。Department ID と呼ばれる外部キーが Employee テーブルに表示されます。

- **1 対 1**　1 対 1 の関係では、外部キーはどちらのテーブルにも方向付けできます。関係が一方では必須で、他方で任意であれば、任意の側に方向付けされます。この例では、乗り物が必ずしもトラックである必要はないので、Truck テーブルに外部キー (Vehicle ID) を置きます。

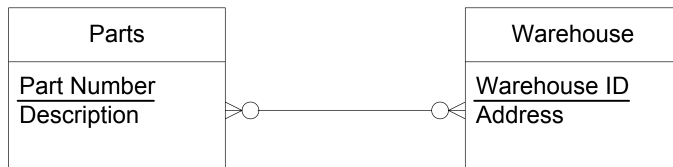


この ER モデルは次のデータベース構造に解析されます。

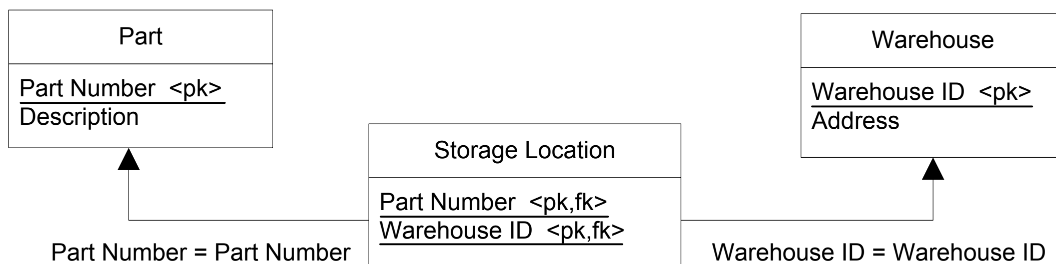


- **多対多**　多対多の関係では、新しいテーブルと 2 つの外部キーが作成されます。この方法は効率的なデータベースを作成するためには欠かせません。

 Copyright © 2007, iAnywhere Solutions, Inc.

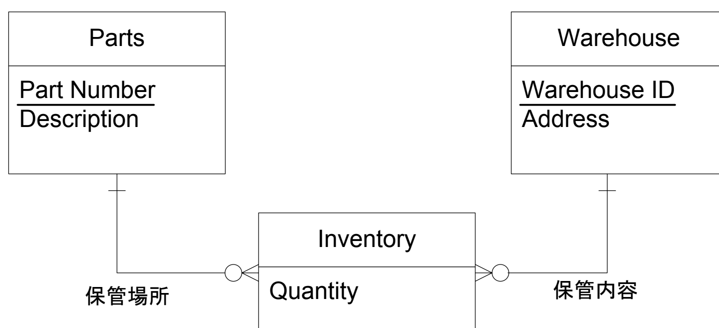


新しい Storage Location テーブルは Part テーブルと Warehouse テーブルを関連付けます。

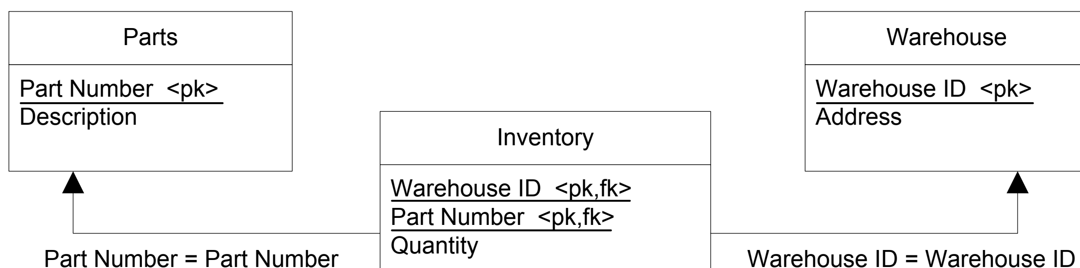


データを含む関係の解析

次の図に示すように、関係がデータを含むことがあります。これは、多対多の関係でしばしば起こります。



このような場合は、各エンティティをテーブルに解析します。各役割が別のテーブルを示す外部キーとなります。



Copyright © 2007, iAnywhere Solutions, Inc.

Inventory エンティティは Part テーブルと Warehouse テーブルの両方に依存しているため、それぞれから識別子を借りています。解析すると、これらの借りた識別子は Inventory テーブルのプライマリ・キーになります。

ヒント
概念データ・モデルは多くの詳細を隠すため、設計プロセスを簡略化します。たとえば、多対多の関係は必ず新しいテーブルと 2 つの外部キー参照を生成します。概念データ・モデルでは、通常これらの構造すべてを単一の接続で示すことができます。

手順 5：設計を検証する

要求がサポートされていることを確認してから、作成した設計を実装します。設計の最初に確認したアクティビティを検査して、それに必要なすべてのデータにアクセスできることを確認します。

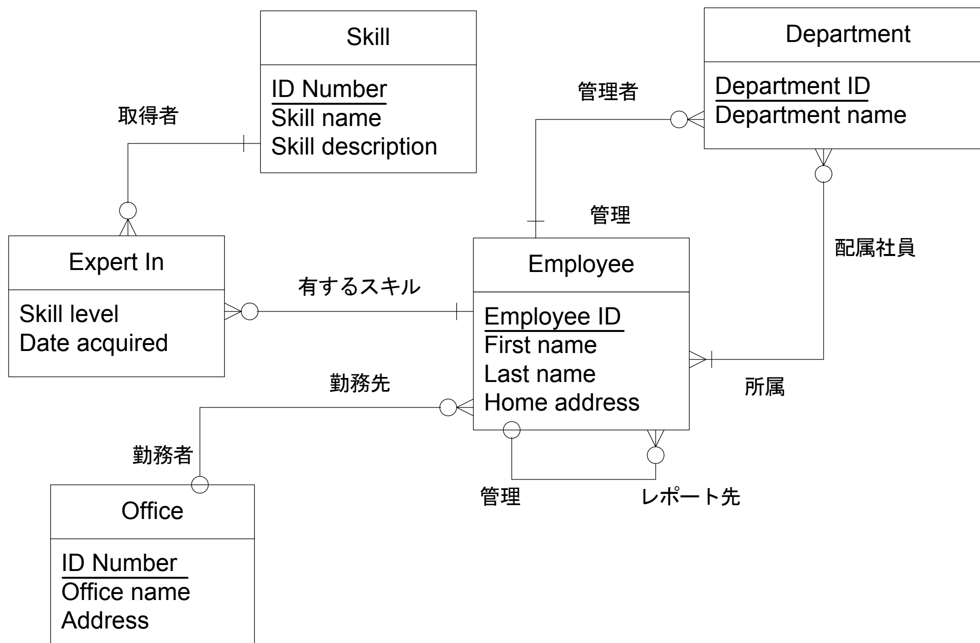
- 必要な情報を取得するパスが検索できるか？
- 設計はユーザの要求を満たしているか？
- 必要なすべてのデータは使用可能か？

これらすべての条件を満たしていれば、作成した設計を実装できます。

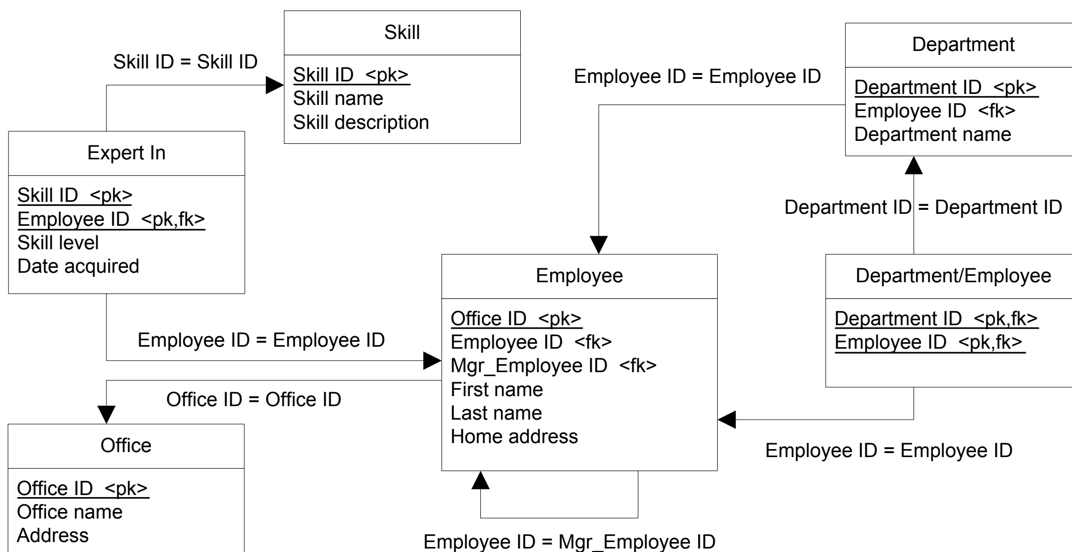
最終設計

小規模な会社用のデータベースに手順 1 から手順 3 までを適用すると、次の ER 図が作成されます。このデータベースは現在第 3 正規形です。

Copyright © 2007, iAnywhere Solutions, Inc.



対応する物理データ・モデルは次のようになります。



テーブル・プロパティの設計

データベース設計では、テーブルと各テーブルに含まれるカラムを指定します。この項では、各カラムのプロパティを指定する方法を説明します。

カラムごとに、カラム名、データ型とサイズ、NULL 値を許容するかどうか、カラム内の値をデータベースに自動で制限させるかどうかなどを決定します。

カラム名の選択

カラム名には、英字、数字、記号を自由に組み合わせて使うことができます。ただし、カラム名に英字、数字、アンダースコア以外のものが含まれていたり、カラム名が英字で始まっていなかったり、それがキーワードと同じであったりする場合、カラム名を二重引用符 (" ") で囲みます。

キーワードのリストについては、「予約語」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

カラムのデータ型の選択

SQL Anywhere では、次のデータ型を使用できます。

- integer データ型
- decimal データ型
- 浮動小数点データ型
- 文字データ型
- バイナリ・データ型
- datetime データ型
- ドメイン (ユーザ定義データ型)

データ型の詳細については、「SQL データ型」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

CHAR、VARCHAR、LONG VARCHAR、NCHAR、BINARY、VARBINARY などの文字列やバイナリ文字列のデータ型を使用して、イメージ、ワープロ文書、音声ファイルなどのラージ・オブジェクトを格納できます。

BLOB ストレージの詳細については、「データベースへの BLOB の格納」 25 ページを参照してください。

NULL と NOT NULL

カラム値がローに対して必須である場合、そのカラムを NOT NULL になるように定義します。そうしないと、カラムには NULL 値が許可され、値を表示しません。SQL Anywhere のデフォルトは NULL 値を許容しますが、NULL 値を許容する正当な理由がないかぎり、カラムには NOT NULL を明示的に宣言してください。

Copyright © 2007, iAnywhere Solutions, Inc.

SQL Anywhere サンプル・データベースには Departments という名前のテーブルがあり、そのテーブルには DepartmentID、DepartmentName、DepartmentHeadID カラムがあります。カラムの定義は次のとおりです。

カラム	データ型	サイズ	NULL/NOT NULL	制約
DepartmentID	integer	–	NOT NULL	なし
DepartmentName	char	40	NOT NULL	なし
DepartmentHeadID	integer	–	NULL	なし

NOT NULL を指定する場合は、テーブルのどのローにも値がなければなりません。

NULL 値の詳細については、「NULL 値」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。NULL 値を比較で使用する方法については、「探索条件」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

データベースへの BLOB の格納

BLOB は、未解釈のバイト文字列または文字列で、値としてカラムに格納されます。BLOB の一般的な例としては、画像や音声ファイルがあります。BLOB は大きなものが多く、CHAR、VARCHAR、NCHAR、BINARY、VARBINARY などの文字列やバイナリ文字列のデータ型で格納できます。格納する BLOB の内容や長さに応じて、データ型や長さを選択してください。

> **注意**
> キャラクタ・ラージ・オブジェクトは一般に CLOB と呼ばれるのに対し、バイナリ・ラージ・オブジェクトは BLOB と呼ばれ、両方を組み合わせたものは LOB と呼ばれますが、このマニュアルでは BLOB の略語のみ使用します。

BLOB ストレージ

BLOB 値を格納するカラムを作成する場合は、ストレージの特性を制御できます。たとえば、指定したサイズ以下の BLOB をロー (インライン) に格納し、指定したサイズを超える BLOB をローの外側にあるテーブルの拡張ページに格納するよう指定できます。また、ローの外側に格納した BLOB については、プレフィクスとも呼ばれる先頭の *n* バイトをロー内に複製するよう指定できます。このようなストレージ特性は、CREATE TABLE や ALTER TABLE 文で指定された INLINE や PREFIX 設定によって制御されます。これらの設定で指定した値が、パフォーマンスやディスク記憶領域の要件に予期しない影響を与える場合があります。

INLINE と PREFIX のどちらも指定しない場合、または USE DEFAULT を指定した場合、デフォルト値は次のように適用されます。

- CHAR、NCHAR、LONG VARCHAR、XML などの文字データ型のカラムの場合、INLINE のデフォルト値は 256 で、PREFIX のデフォルト値は 8 です。

- BINARY、LONG BINARY、VARBINARY、BIT、VARBIT、LONG VARBIT、BIT VARYING、UUID などのバイナリ・データ型のカラムの場合、INLINE のデフォルト値は 256 で、PREFIX のデフォルト値は 0 です。

デフォルト値では対応できない特定の要件がある場合を除き、INLINE と PREFIX の値は設定しないようにしてください。デフォルト値は、パフォーマンスとディスク領域の均衡が保たれるよう設定されています。たとえば、INLINE に大きい値を設定すると、すべての BLOB はインラインに格納されるため、ロー・プロセスのパフォーマンスが低下する場合があります。また、PREFIX の値が大きすぎると、プレフィクス・データは BLOB の一部を複製するため、BLOB の格納に必要なディスク領域が増えることになります。

INLINE や PREFIX の値を設定する場合は、INLINE の長さがカラム長を超えないようにしてください。同様に、PREFIX の長さも INLINE の長さを超えないようにしてください。

INLINE 句と PREFIX 句のデフォルト値については、「CREATE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

圧縮されたカラムの場合は、INLINE や PREFIX の設定に関係なく、INLINE と PREFIX の値に 0 が設定されたように動作します。したがって、プレフィクスは格納されず、BLOB はテーブル拡張ページに格納され、INDEX が指定されていれば (デフォルト設定)、BLOB のインデックスも作成されます。圧縮されたカラムを展開すると、INLINE や PREFIX で指定された有効な設定はすべて復元されます。

BLOB 共有

BLOB のサイズがインライン・サイズを超えて、BLOB の格納に 2 ページ以上のデータベース・ページが必要になった場合、データベース・サーバは同じテーブルの別のローから参照できるよう BLOB を格納します。これは BLOB 共有と呼ばれます。BLOB 共有は内部で処理されます。BLOB 共有はデータベース内で不要に BLOB が複製されないようにするためのものです。

BLOB 共有は、特定のカラムに別のカラムと同じ値を設定する場合にのみ発生します。たとえば、UPDATE t column1=column2; が該当します。この例では、column2 に BLOB が含まれている場合に、column1 に BLOB を複製する代わりに、column2 の値を示すように設定します。

BLOB を共有すると、データベース・サーバは BLOB を参照している数を追跡します。データベース・サーバがテーブル内で参照されている BLOB がないと判断すると、BLOB は削除されます。

圧縮されていない 2 つのカラムで BLOB を共有している場合に、一方のカラムを圧縮すると、BLOB は共有でなくなります。

カラムを圧縮するかどうかの選択

CHAR、VARCHAR、BINARY カラムを圧縮して、ディスク領域を節約できます。たとえば、BMP や TIFF などの大きい BLOB ファイルを格納するカラムを圧縮できます。圧縮には deflate 圧縮アルゴリズムが使用されます。これは、COMPRESS 関数で使用される圧縮方式と同じであり、Windows ZIP ファイルで使用されるアルゴリズムです。

 Copyright © 2007, iAnywhere Solutions, Inc.

カラムを圧縮すると、インデックス、データの比較、統計の生成などのデータベース・サーバのアクティビティの処理速度が、圧縮したカラムを対象とする場合に、遅くなる可能性があります。これは、値を書き込むときに圧縮し、読み込むときに解凍する必要があるからです。

圧縮されたカラムは暗号化されたテーブルの内側に格納できます。この場合、データは最初に圧縮されてから暗号化されます。

カラムに 130 バイト未満の値が設定されていたり、JPG ファイルなどの圧縮形式がすでに設定されている場合は、カラムの圧縮は意味がありません。実際には、圧縮された値を含むカラムを圧縮すると、カラムに必要な記憶領域のサイズが大きくなる可能性があります。

カラムを圧縮するには、CREATE TABLE と ALTER TABLE 文の COMPRESS 句を使用します。

カラムの圧縮による効果を確認するには、sa_column_stats システム・プロシージャを使用します。

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』
- 「sa_column_stats システム・プロシージャ」『SQL Anywhere サーバ - SQL リファレンス』
- 「テーブル暗号化」『SQL Anywhere サーバ - データベース管理』

制約の選択

カラムのデータ型は許可されるデータ値を制限しますが (たとえば数字のみ、日付のみなど)、データ値を更に制限したい場合もあります。

カラム中のデータ値は検査制約を設定して制限できます。WHERE 句に表示される有効な条件を使用して、許可される値を制限できます。通常、検査制約では BETWEEN または IN 条件を使用します。

有効な条件の詳細については、「探索条件」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。テーブルとカラムへの制約の割り当てについては、「データ整合性の確保」 95 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

第 2 章

データベース・オブジェクトの使用

目次

データベース・オブジェクトを使用した作業の概要 30
データベースの編集 31
テーブルの編集 45
ビューの編集 60
インデックスの編集 87
テンポラリ・テーブルの編集 94

データベース・オブジェクトを使用した作業の概要

SQL Anywhere ツールを使用すると、データを格納するデータベース・ファイルを作成できます。このファイルを作成すれば、データベースの管理を始められます。たとえば、テーブルやユーザなどのデータベース・オブジェクトを追加したり、データベース全体のプロパティを設定したりできます。

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。スキーマを空のデータベースとみなすことができます。

プロシージャとトリガもデータベース・オブジェクトですが、これらについては、「プロシージャ、トリガ、バッチの使用」 777 ページを参照してください。

この章を開始するにあたり、概念的な情報が必要な場合は、次の項を参照してください。

- 「データベースの設計」 3 ページ
- 「データ整合性の確保」 95 ページ
- 「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』

Copyright © 2007, iAnywhere Solutions, Inc.

データベースの編集

この項では、データベースを作成、編集する方法について説明します。この項を読むときは、次の簡単な概念に留意してください。

- 作成できるデータベース (リレーショナル・データベースと呼ぶ) は、プライマリ・キーと外部キーで関連付けされたテーブルの集合です。これらのテーブルは、データベース内の情報を保持します。また、データベースの構造は、テーブルとキーによって定義されます。データベースは、1 つまたは複数のデバイス上の、1 つまたは複数のデータベース・ファイルに格納できます。
- データベース・ファイルには、データベースを構築する場合のスキーマ定義を保持する、システム・テーブルも含まれています。

データベースの作成

SQL Anywhere では、データベースの作成方法が数種類あります。これらは、Sybase Central、Interactive SQL における作成方法や、コマンド・ラインを使用する方法です。データベースを作成することをデータベースの**初期化**と呼びます。データベースを作成すると、それに接続して、データベースで必要なテーブルやその他のオブジェクトを構築できます。

他のアプリケーション設計システムには、Sybase PowerDesigner Physical Model のようにデータベース・オブジェクト作成ツールを備えたものがあります。これらのツールは SQL 文を作成して、通常は ODBC インタフェースを通してデータベース・サーバに送られます。このようなツールを使えば、テーブルの作成、パーミッションの割り当てなどを行う SQL 文を構築する必要はありません。「PowerDesigner Physical Data Model について」『SQL Anywhere 10 - 紹介』を参照してください。

この章では、データベース・オブジェクトを定義する SQL 文について説明します。Interactive SQL のようなツールを使用してデータベースを構築する場合は、直接 SQL 文を使用できます。アプリケーション設計ツールを使っている場合でも、設計ツールでサポートされない機能をデータベースに追加する場合は SQL 文を使用します。

データベース設計の詳細については、「データベースの設計」 3 ページを参照してください。

トランザクション・ログ

データベースを作成するには、トランザクション・ログを配置する場所を決定しなければなりません。このログには、データベースへ加えられた変更内容が、変更された順に格納されます。データベース・ファイルでメディア障害が発生した場合、トランザクション・ログはデータベースのリカバリに重要な役割を果たします。また、トランザクション・ログを使用するとより効率的に作業できます。デフォルトでは、トランザクション・ログはデータベース・ファイルと同じディレクトリに配置されますが、この方法での運用はおすすめしません。

トランザクション・ログの配置の詳細については、「データ保護を目的としたデータベースの構成」『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・ファイルの互換性

SQL Anywhere データベースはオペレーティング・システム・ファイルです。このデータベースは、他のファイルと同様に別のロケーションにコピーできます。

大きなファイルに対するサイズが制限される場合や、SQL Anywhere がサポートする場合以外は、すべてのオペレーティング・システム間でデータベース・ファイルには互換性があります。「SQL Anywhere のサイズと数の制限」『SQL Anywhere サーバ - データベース管理』を参照してください。

オペレーティング・システム間の互換は、データベース・ファイルをコピーすれば可能です。同様に、パーソナル・データベース・サーバで作成されたデータベースは、ネットワーク・データベース・サーバで使用できます。SQL Anywhere データベース・サーバは、ソフトウェアの旧バージョンで作成されたデータベースを管理できますが、旧バージョンのサーバは新バージョンのデータベースを管理できません。

データベースの作成 (Sybase Central の場合)

Sybase Central では、[データベース作成] ウィザードを使用してデータベースを作成できます。

詳細については、「データベースの作成 (SQL の場合)」 33 ページ と「データベースの作成 (コマンド・ラインの場合)」 34 ページ を参照してください。

◆ 新しいデータベースを作成するには、次の手順に従います (Sybase Central の場合)。

1. [ツール] メニューから、[SQL Anywhere 10] – [データベースの作成] を選択します。

   [データベースの作成] ウィザードが表示されます。

2. ウィザードの指示に従います。

> **ヒント**
> Sybase Central では、次の方法で [データベース作成] ウィザードを利用することもできます。
>
> ◆ サーバを選択し、[ファイル] メニューから [データベースの作成] を選択する。
> ◆ サーバを右クリックし、ポップアップ・メニューから [データベースの作成] を選択する。

Windows CE 用データベースの作成

SQL Anywhere のデータベース・ファイルをデバイスにコピーすると、Windows CE 用のデータベースを作成できます。

Sybase Central には、Windows CE データベース用に簡単にデータベースを作成する機能があります。Windows のデスクトップに Windows CE サービスをインストールしてあれば、Sybase Central からデータベースを作成するときに、オプションで Windows CE データベースを作成できます。Sybase Central は、Windows CE データベースの一部の機能を省略します。また、オプションで、結果として生成されるデータベース・ファイルを Windows CE デバイスにコピーします。

 Copyright © 2007, iAnywhere Solutions, Inc.

Windows CE のチェックサムの有効化
Windows CE 上に配備するデータベースを作成する場合は、チェックサムを有効にする必要があります。これによって、データベース・ファイルの破損を速やかに検出できます。

詳細については、「Windows CE データベースの作成」『SQL Anywhere サーバ - データベース管理』を参照してください。

データベースの作成 (SQL の場合)

Interactive SQL では、CREATE DATABASE 文を使用してデータベースを作成します。既存のデータベースに接続してから、この文を使用する必要があります。

♦ 新しいデータベースを作成するには、次の手順に従います (SQL の場合)。

1. 「sample」という名前のデータベース・サーバを起動します。

   ```
   dbeng10 -n sample
   ```

2. Interactive SQL を起動します。
3. 既存のデータベースに接続します。データベースが存在しない場合は、ユーティリティ・データベース utility_db に接続できます。「ユーティリティ・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。
4. CREATE DATABASE 文を実行します。

詳細については、「CREATE DATABASE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

例

c:¥temp ディレクトリにファイル名 *temp.db* のデータベースを作成します。

```
CREATE DATABASE 'c:¥¥temp¥¥temp.db';
```

ディレクトリ・パスはデータベース・サーバと関連します。この文を実行するのに必要なパーミッションは、サーバ・コマンド・ラインで -gu オプションを使用して設定します。デフォルトの設定は、DBA 権限を必要とします。

円記号 (¥) は SQL のエスケープ文字であるため、場合によって 2 つ付けます。¥x と ¥n の各シーケンスを使用して、16 進数で文字を指定したり、改行文字を指定したりできます。n と x 以外の文字は、前に円記号が付いていても特別な意味はありません。このことが重要な場合の例を示します。

```
CREATE DATABASE 'c:¥¥temp¥¥¥x41¥x42¥x43xyz.db';
```

最初の ¥¥ シーケンスは円記号を表します。 ¥x シーケンスは、それぞれ文字 A、B、C を表します。このファイル名は ABCxyz.db です。

```
CREATE DATABASE 'c:¥temp¥¥nest.db';
```

¥n シーケンスが改行文字として解釈されないように、円記号を 2 つ連続で使用します。

詳細については、「文字列」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

データベースの作成 (コマンド・ラインの場合)

初期化ユーティリティ (dbinit) を使用して、コマンド・ラインからデータベースを作成できます。このユーティリティを使用すると、コマンド・ライン・オプションを含めて別のデータベース設定を指定できます。

♦ 新しいデータベースを作成するには、次の手順に従います (コマンド・ラインの場合)。

1. コマンド・プロンプトを開きます。
2. dbinit ユーティリティを実行します。必要なパラメータをすべて含めます。

   たとえば、ページのサイズが 4 KB のデータベース *company.db* を作成するには、次のように入力します。

   ```
   dbinit -p 4k company.db
   ```

詳細については、「初期化ユーティリティ (dbinit)」 『SQL Anywhere サーバ - データベース管理』を参照してください。

データベースの起動

Sybase Central と Interactive SQL のどちらの場合も、アプリケーションに接続しないでデータベースを起動できます。

♦ 接続しないでサーバ上でデータベースを起動するには、次の手順に従います (Sybase Central の場合)。

1. インデックスを選択し、[ファイル] – [データベースの開始] を選択します。
2. [データベースの開始] ダイアログに必要な値を入力します。

   データベースが、データベース・サーバの下に未接続のデータベースとして表示されます。

♦ 接続しないでサーバ上でデータベースを起動するには、次の手順に従います (SQL の場合)。

- START DATABASE 文を実行します。

詳細については、「START DATABASE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

例

sample データベース・サーバ上で、データベース・ファイル *c:¥temp¥temp.db* を起動します。

```
START DATABASE 'c:¥¥temp¥¥temp.db'
AS tempdb ON 'sample'
AUTOSTOP OFF;
```


Copyright © 2007, iAnywhere Solutions, Inc.

別のデータベースを起動するには、データベースに接続されている必要があります。

AUTOSTOP OFF を使用すると、データベースはすべての接続が終了しても自動的には停止されません。ここで AUTOSTOP OFF は、後述するポイントを示すために使用しています。

データベースへの接続

データベースの編集を開始する前に、データベースに接続する必要があります。

データベース・サーバが起動すると、ユニークな接続 ID がデータベース・サーバへの各新規接続に割り当てられます。CONNECTION_PROPERTY 関数を使って接続番号を要求し、ユーザの *connection-id* を取得します。次の文は、現在の接続の接続 ID を返します。

```
SELECT CONNECTION_PROPERTY( 'number' );
```

♦ データベースに接続するには、次の手順に従います (Sybase Central の場合)。

1. [接続] − [SQL Anywhere 10 に接続] を選択します。

   [接続] ダイアログが表示されます。

2. [接続] ダイアログで、データベースの接続に使用する接続情報または ODBC データ・ソース名を指定し、[OK] を選択します。

♦ データベースに接続するには、次の手順に従います (SQL の場合)。

- CONNECT 文を実行します。

詳細については、「CONNECT 文 [ESQL] [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

例 1

次の文は、Interactive SQL から CONNECT を使用して、起動済みのデータベースに接続する方法を示します。

```
CONNECT to 'sample' DATABASE tempdb
AS conn1
USER 'DBA'
IDENTIFIED BY 'sql';
```

この例では、sample はサーバ名、tempdb はデータベース名、conn1 は接続識別子、DBA はユーザ名、sql はパスワードです。

例 2

次の文は、Embedded SQL 内の CONNECT の使用法を示します。

```
EXEC SQL CONNECT "DBA"
IDENTIFIED BY "sql";
```

Copyright © 2007, iAnywhere Solutions, Inc.

データベース・オブジェクトのプロパティの設定

ほとんどのデータベース・オブジェクト (データベース自身を含む) には、表示または設定が可能なプロパティがあります。中には、設定できないプロパティや、データベースまたはオブジェクトの作成時に選択した設定を反映するプロパティもあります。それ以外はすべて設定可能なプロパティです。

プロパティを表示、変更する最も良い方法は、Sybase Central でプロパティ・シートを使用することです。

Sybase Central を使用しない場合は、CREATE 文を使用してオブジェクトの作成時にプロパティを指定できます。オブジェクトがすでに存在する場合は、ALTER 文を使用してオプションを変更できます。

♦ データベース・オブジェクトのプロパティを表示／編集するには、次の手順に従います (Sybase Central の場合)。

1. Sybase Central で、対象のオブジェクトがあるフォルダを開きます。
2. オブジェクトを選択します。オブジェクトのプロパティが Sybase Central の右ウィンドウ枠に表示されます。
3. 右ウィンドウ枠で、対象プロパティを編集できる該当タブをクリックします。

   オブジェクトのプロパティ・シートでも、プロパティを表示、編集できます。プロパティ・シートを表示するには、オブジェクトを選択し、[ファイル] – [プロパティ] を選択します。

データベース・オプションの設定

データベース・オプションを設定して、データベースの動作方法や実行方法を変更できます。Sybase Central では、これらのオプションがすべて [データベース・オプション] ダイアログでグループ分けされています。Interactive SQL では、SET OPTION 文を使用してオプション設定を変更できます。

♦ データベースのオプションを設定するには、次の手順に従います (Sybase Central の場合)。

1. 対象のサーバを開きます。
2. 目的のデータベースを右クリックし、ポップアップ・メニューで [オプション] を選択します。
3. 値を編集します。

♦ データベースのオプションを設定するには、次の手順に従います (SQL の場合)。

- SET OPTION 文で対象の値を指定します。

Copyright © 2007, iAnywhere Solutions, Inc.

ヒント
[データベースのオプション] ダイアログからも、指定のユーザとグループに対してデータベース・オプションを設定できます (あるユーザまたはグループに対してこのダイアログを開いた場合、それぞれを [ユーザのオプション] ダイアログまたは [グループのオプション] ダイアログと呼びます)。
データベース自体のオプションを設定する場合は、実際にデータベースの PUBLIC グループのオプションを設定します。これは、すべてのユーザとグループが、PUBLIC からオプションの設定を継承するためです。

オプションの詳細については、「データベース・オプションの概要」『SQL Anywhere サーバ - データベース管理』を参照してください。

統合データベースの指定

Sybase Central では、SQL Remote レプリケーションに対して統合データベースを指定できます。統合データベースとは、レプリケーションの設定時に master データベースとして機能するデータベースのことです。統合データベースにはレプリケートされるデータがすべて含まれていますが、そのリモート・データベースにはデータのサブセットだけが含まれることがあります。競合や不一致が発生した場合、すべてのデータのプライマリ・コピーは統合データベースにあるとみなされます。

詳細については、「統合リモート・データベース」『SQL Anywhere 10 - 紹介』を参照してください。

♦ **統合データベースを設定するには、次の手順に従います (Sybase Central の場合)。**

1. データベースを選択し、[ファイル] – [プロパティ] を選択します。
2. [SQL Remote] タブをクリックします。
3. [このリモート・データベースに対応する統合データベースが存在する] オプションを選択します。
4. 必要な設定を実行します。

データベースのシステム・オブジェクトの表示

データベース内のテーブル、ビュー、ストアド・プロシージャ、またはドメインは、システム・オブジェクトです。「**システム・テーブル**」には、データベースのスキーマやデータベース自体の情報が格納されます。システム・ビュー、プロシージャ、ドメインは Sybase Transact-SQL の広範囲の互換性をサポートしています。

データベースのシステム・オブジェクトに関するすべての情報は、システム・テーブルに表示されます。この情報はいくつかのテーブルに分散しています。

Copyright © 2007, iAnywhere Solutions, Inc.

♦ データベースのシステム・オブジェクトを表示するには、次の手順に従います (Sybase Central の場合)。

1. 対象のデータベース・サーバを開きます。
2. 接続済みのデータベースを選択し、[ファイル] – [所有者別にオブジェクトをフィルタ] を選択します。
3. **SYS** と **dbo** を選択して、[OK] をクリックします。

   システム・テーブル、システム・ビュー、システム・プロシージャがそれぞれのフォルダに表示されます。たとえば、システム・テーブルは、[テーブル] フォルダの通常のテーブルの横に表示されます。

♦ システム・テーブルをブラウズするには、次の手順に従います (SQL の場合)。

1. 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。

例

次の SELECT 文は、SYSOBJECT システム・ビューに対してクエリを行い、データベース内にあるすべてのオブジェクト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,
 a.object_type, a.status
  FROM ( SYSOBJECT a JOIN SYSTAB b
   ON a.object_id = b.object_id )
    JOIN SYSUSER c
  ORDER BY table_name;
```

SYSOBJECT、SYSTAB、SYSUSER の各システム・ビューの詳細については、「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』を参照してください。

SQL 文のロギング

Sybase Central でデータベースを編集する場合、アクションに応じて自動的に SQL 文が生成されます。[サーバ・メッセージと実行された SQL] と呼ばれる別のウィンドウ枠で、これらの文を追跡できます。または情報をファイルに保存できます。[サーバ・メッセージと実行された SQL] ウィンドウ枠には、データベースとデータベース・サーバごとにタブがあります。データベース・サーバのタブには、メッセージ・ウィンドウと同じ情報が表示されます。

Interactive SQL の場合、自ら実行した文のログをとることもできます。

詳細については、「コマンドのロギング」『SQL Anywhere サーバ - データベース管理』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

♦ **Sybase Central によって生成される SQL 文のログをとるには、次の手順に従います。**

1. [ビュー] メニューから、[サーバ・メッセージと実行された SQL] を選択します。

   Sybase Central のウィンドウの下部に、[サーバ・メッセージと実行された SQL] ウィンドウ枠が表示されます。

2. [サーバ・メッセージと実行された SQL] ウィンドウ枠で右クリックし、ポップアップ・メニューで [オプション] を選択します。

   [オプション] ダイアログが表示されます。

3. 表示されるダイアログで、必要な設定を指定します。ログ情報をファイルに保存する場合は、[保存] をクリックします。

データベースのアップグレード

> **注意**
> [データベース・アップグレード] ウィザードでは、バージョン 9.0.2 以前のデータベースをバージョン 10 にアップグレードすることはできません。既存のデータベースをバージョン 10 にアップグレードするには、dbunload または [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し、再ロードする必要があります。「SQL Anywhere のアップグレード」『SQL Anywhere 10 - 変更点とアップグレード』を参照してください。

[データベース・アップグレード] ウィザードは、バージョン 10.0.0 のデータベースをソフトウェアの新バージョンにアップグレードするときに役立ちます。また、データベース・オプション、システム・テーブルやビュー、およびシステム・プロシージャなどのデフォルト設定をリストアするときにも利用できます。

♦ **データベースをアップグレードするには、次の手順に従います (Sybase Central の場合)。**

1. データベースに接続します。
2. 左ウィンドウ枠で、SQL Anywhere プラグインを選択します。
3. 右ウィンドウ枠で、[ユーティリティ] タブをクリックします。
4. 右ウィンドウ枠で、[データベースのアップグレード] をダブルクリックします。

   [データベース・アップグレード] ウィザードが表示されます。

5. ウィザードの指示に従います。

Copyright © 2007, iAnywhere Solutions, Inc.

ヒント

[データベース・アップグレード] ウィザードは、次の方法でもアクセスできます。

- データベースを右クリックし、ポップアップ・メニューから [データベースのアップグレード] を選択する。
- データベースを選択し、[ファイル] メニューから [データベースのアップグレード] を選択します。

参照

- 「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』

jConnect メタデータ・サポートを既存のデータベースに追加する

jConnect メタデータ・サポートなしでデータベースを作成した場合は、Sybase Central を使用して後からインストールできます。

♦ jConnect メタデータ・サポートを既存のデータベースへ追加するには、次の手順に従います (Sybase Central の場合)。

1. アップグレードするデータベースに接続します。
2. [ツール] メニューから、[SQL Anywhere 10] – [データベースのアップグレード] を選択します。

   [データベース・アップグレード] ウィザードが表示されます。
3. ウィザードの概要ページで [次へ] をクリックします。
4. アップグレードするデータベースをリストから選択します。[次へ] をクリックします。
5. 必要であれば、データベースのバックアップ作成も選択できます。[次へ] をクリックします。
6. [jConnect メタ情報サポートをインストール] オプションを選択します。
7. ウィザードの指示に従います。

データベースとの接続の切断

データベースの編集が終了すれば、接続を切断できます。SQL Anywhere では、別のユーザとデータベースの接続を切断することもできます。Sybase Central でのこの作業の詳細については、「接続されたユーザの管理」『SQL Anywhere サーバ - データベース管理』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

データベース・サーバが起動すると、ユニークな接続 ID がデータベース・サーバへの各新規接続に割り当てられます。CONNECTION_PROPERTY 関数を使って接続番号を要求し、ユーザの *connection-id* を取得します。次の文は、現在の接続の接続 ID を返します。

```
SELECT CONNECTION_PROPERTY( 'number' );
```

♦ **データベースの接続を切断するには、次の手順に従います (Sybase Central の場合)。**

1. データベースを選択します。
2. [ファイル] – [切断] を選択します。

♦ **データベースの接続を切断するには、次の手順に従います (SQL の場合)。**

・ DISCONNECT 文を実行します。

詳細については、「DISCONNECT 文 [ESQL] [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』と「DROP CONNECTION 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

例 1

次の文は、Interactive SQL で DISCONNECT 文を使用して、現在の接続 conn1 を切断する方法を示します。

```
DISCONNECT conn1;
```

例 2

次の文は、Embedded SQL 内の DISCONNECT の使用法を示します。

```
EXEC SQL DISCONNECT :conn-name
```

♦ **別のユーザとデータベースの接続を切断するには、次の手順に従います (SQL の場合)。**

1. DBA ユーザとしてデータベースに接続します。
2. sa_conn_info システム・プロシージャを使用して、接続するユーザの接続 ID を決定します。
3. DROP CONNECTION 文を実行します。

例

次の文は、接続番号 4 を削除します。

```
DROP CONNECTION 4;
```

データベースの停止

Sybase Central と Interactive SQL のどちらの場合も、データベース・サーバを実行したままデータベースを停止できます。現在接続しているデータベースは停止できません。先にデータベースから切断してから、停止する必要があります。データベースを停止するには、同じデータベース・サーバ上の別のデータベースに接続している必要があります。

Copyright © 2007, iAnywhere Solutions, Inc.

♦ 切断後にサーバ上のデータベースを停止するには、次の手順に従います (Sybase Central の場合)。

1. 同じデータベース・サーバにある 1 つ以上の別のデータベースに接続していることを確認します。サーバ上で実行しているデータベースがほかにない場合は、ユーティリティ・データベースに接続できます。
2. 停止するデータベースを選択し、[ファイル] – [データベースの停止] を選択します。

データベースから切断すると、そのデータベースが左ウィンドウ枠に表示されなくなることがあります。現在の接続が残っている唯一の接続で、かつデータベースの起動時に AUTOSTOP を指定した場合は、このような状況になります。AUTOSTOP を指定すると、最後の接続が終了するときに、データベースが自動的に停止します。

♦ 切断後にサーバ上のデータベースを停止するには、次の手順に従います (SQL の場合)。

1. サーバ上のどのデータベースにも接続していない場合は、ユーティリティ・データベースなどのデータベースに接続します。
2. STOP DATABASE 文を実行します。

詳細については、「STOP DATABASE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ユーティリティ・データベースへの接続方法については、「ユーティリティ・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。

例

次の文は、ユーティリティ・データベースに接続して、tempdb データベースを停止します。

```
CONNECT to 'TestEng' DATABASE utility_db
AS conn2
USER 'DBA'
IDENTIFIED BY 'sql';
STOP DATABASE tempdb;
```

別のデータベースを停止するには、データベースに接続されている必要があります。

データベースの消去

データベースを消去すると、データベースへの変更を記録したトランザクション・ログを含むすべてのテーブルとデータがディスクから削除されます。データベース・ファイルはすべて、誤ってファイルを変更したり削除したりするのを防ぐために読み込み専用となっています。データベースを消去するには、デフォルトで DBA 権限が必要です。データベース・サーバの -gu オプションを使用すると、必要なパーミッションを変更できます。「-gu 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。

Sybase Central では、[データベース消去] ウィザードを使用してデータベースを消去できます。このウィザードにアクセスするには、データベースのどれかに接続する必要がありますが、[デー

 Copyright © 2007, iAnywhere Solutions, Inc.

タベース消去] ウィザードからは他のデータベースも消去できます。実行していないデータベースを消去するには、そのデータベース・サーバを起動しておきます。

Interactive SQL では、DROP DATABASE 文を使用してデータベースを消去できます。

dberase ユーティリティを使用して、コマンド・ラインからデータベースを消去する方法もあります。ただし、dberase ユーティリティは DB 領域を消去しません。DB 領域を消去したい場合、DROP DATABASE 文を使用するか、または Sybase Central の [データベース消去] ウィザードを使用します。

dberase ユーティリティ、[データベース消去] ウィザード、DROP DATABASE 文を使用する場合、消去するデータベースは実行中にしないでください。別のデータベースを切断するには、データベースに接続されている必要があります。

ユーティリティ・データベースへの接続方法については、「ユーティリティ・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。

Windows CE のデータベースは手動で消去する必要があります。「Windows CE データベースの消去」『SQL Anywhere サーバ - データベース管理』を参照してください。

♦ **データベースを消去するには、次の手順に従います (Sybase Central の場合)。**

1. [ツール] メニューから [SQL Anywhere 10] – [データベースの消去] を選択します。
2. ウィザードの指示に従います。

ヒント
Sybase Central では、次の方法で [データベース消去] ウィザードを利用することもできます。

♦ データベース・サーバを選択し、[ファイル] メニューから [データベースの消去] を選択する。

♦ サーバを右クリックし、ポップアップ・メニューから [データベースの消去] を選択する。

♦ **データベースを消去するには、次の手順に従います (SQL の場合)。**

1. 消去対象でないデータベースに接続します。たとえば、ユーティリティ・データベースに接続します。
2. DROP DATABASE 文を実行します。

   たとえば、次の DROP DATABASE 文は、temp というデータベースを消去します。

   ```
   DROP DATABASE 'c:¥¥temp¥¥temp.db';
   ```

詳細については、「DROP 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

♦ データベースを消去するには、次の手順に従います (コマンド・ラインの場合)。

- コマンド・ラインから、dberase ユーティリティを実行します。

  たとえば、次のコマンドは、temp データベースを削除します。

```
dberase c:¥temp¥temp.db
```

詳細については、「消去ユーティリティ (dberase)」『SQL Anywhere サーバ - データベース管理』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

テーブルの編集

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。

この項では、テーブルを作成、変更、削除する方法について説明します。例は InteractiveSQL で実行できますが、SQL 文は使用する管理ツールには依存しません。「Interactive SQL での結果セットの編集」『SQL Anywhere サーバ - データベース管理』を参照してください。

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、コマンド・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。コマンド・ファイルには、CREATETABLE 文と ALTERTABLE 文を含めてください。

グループ、テーブル、別のユーザとしての接続の詳細については、「グループが所有するテーブルの参照」『SQL Anywhere サーバ - データベース管理』と「データベース・オブジェクトの名前とプレフィクス」『SQL Anywhere サーバ - データベース管理』を参照してください。

テーブルの作成

データベースが最初に作成されたときのデータベース内のテーブルは、データベース・スキーマの入ったシステム・テーブルだけです。Interactive SQL の SQL 文または Sybase Central のいずれかを使用して、実際のデータを保持する新しいテーブルを作成できます。

作成できるテーブルのタイプは次の 2 つです。

- **ベース・テーブル**　永続的なデータを保管するテーブルです。テーブルとそのデータは、それらを明示的に削除しないかぎり存在し続けます。テンポラリ・テーブルやビューと区別するため、これをベース・テーブルと呼びます。
- **テンポラリ・テーブル**　テンポラリ・テーブルのデータは、単一の接続に対してだけ保持されます。グローバル・テンポラリ・テーブルの定義は、それが削除されないかぎりデータベースに保持されます (データはこのかぎりではありません)。ローカル・テンポラリ・テーブルの定義とデータは、単一の接続の間だけ存在します。テンポラリ・テーブルの詳細については、「テンポラリ・テーブルの編集」94 ページを参照してください。

テーブルはローとカラムで構成されます。各カラムは電話番号や名前など情報の種類を特定し、各ローはデータのエントリを保持します。

♦ テーブルを作成するには、次の手順に従います (Sybase Central の場合)。

1. データベースに接続します。
2. [テーブル] フォルダを開きます。
3. [ファイル] メニューから [新規] – [テーブル] を選択します。

   [テーブル作成] ウィザードが表示されます。
4. ウィザードの指示に従います。

新しいテーブルは、[テーブル] フォルダに表示されます。

5. 右ウィンドウ枠の [カラム] タブで、テーブルにカラムを追加できます。
6. 終了したら、[ファイル] – [保存] を選択します。

♦ テーブルを作成するには、次の手順に従います (SQL の場合)。

1. DBA ユーザとしてデータベースに接続します。
2. CREATE TABLE 文を実行します。

例

次の文は、会社内の従業員のスキル・レベルを記述するテーブルを新しく作成します。テーブルには、各スキルの ID 番号、名前、種別 (technical または administrative) のカラムが作成されます。

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

詳細については、「CREATE TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

テーブルの変更

この項では、テーブルの構造やカラムの定義を変更する方法について説明します。たとえば、カラムの追加、さまざまなカラムの属性の変更、カラム全体の削除を実行できます。

Sybase Central の右ウィンドウ枠の [SQL] タブで、テーブルの変更タスクを実行できます。Interactive SQL では、ALTER TABLE 文を使用してこれらのタスクを実行できます。

データベース・オブジェクト・プロパティの変更については、「データベース・オブジェクトのプロパティの設定」 36 ページを参照してください。

テーブル・パーミッションの付与と取り消しについては、「テーブルに対するパーミッションの付与」 『SQL Anywhere サーバ - データベース管理』と「ユーザ・パーミッションの取り消し」 『SQL Anywhere サーバ - データベース管理』を参照してください。

テーブルの変更とビューの依存性

従属ビューでテーブルのスキーマを変更するときは、以降の項で説明するように、追加の手順がある場合があります。

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。従属した非実体化ビュー (Non-materialized View) 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・サーバが従属した非実体化ビュー (Non-materialized View) 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。「ビューの変更」 63 ページを参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

従属した実体化ビュー (Materialized View) 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、従属した非実体化ビュー (Non-materialized View) をもう一度有効にできない場合は、変更によって実体化ビュー (Materialized View) 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、実体化ビュー (Materialized View) 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、実体化ビュー (Materialized View) をもう一度有効にしてみてください。「実体化ビュー (Materialized View) の作成」 75 ページを参照してください。

sa_dependent_views システム・プロシージャを使用すると、テーブルを変更する前に、テーブルにビューの依存性があるかどうかを判断できます。「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』を参照してください。

データベース・オブジェクトを変更することでビューの依存性にどのような影響があるかについては、「ビューの依存性」 67 ページを参照してください。

テーブルの変更 (Sybase Central の場合)

Sybase Central では、右ウィンドウ枠の [カラム] タブで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、カラムの定義の変更、テーブルまたはカラムのプロパティの変更を実行できます。依存している実体化ビュー (Materialized View) がある場合はテーブルの変更は失敗します。あらかじめ依存している実体化ビュー (Materialized View) を無効にしておく必要があります。テーブルの変更が完了したら、依存している実体化ビュー (Materialized View) をもう一度有効にする必要があります。

sa_dependent_views システム・プロシージャを使用して、依存している実体化ビュー (Materialized View) があるかどうかを決定します。「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』を参照してください。

ビューの依存性の詳細については、「ビューの依存性」 67 ページを参照してください。

♦ 既存のテーブルを変更するには、次の手順に従います (Sybase Central の場合)。

1. DBA またはテーブルの所有者としてデータベースに接続します。
2. スキーマを変更していて、テーブルに実体化ビュー (Materialized View) の依存性が存在する場合は、次のように各ビューを無効にします。
   a. [ビュー] フォルダを開き、実体化ビュー (Materialized View) を選択します。
   b. [ファイル] – [無効にする] を選択します。
3. [テーブル] フォルダを開き、変更するテーブルを選択します。
4. 右ウィンドウ枠で [カラム] タブをクリックし、必要な変更を行います。
5. [ファイル] – [保存] を選択します。

   テーブルの変更が保存されます。

6. 実体化ビュー (Materialized View) を無効にしたら、次のように各ビューをもう一度有効にし、初期化します。
   a. [ビュー] フォルダを開き、実体化ビュー (Materialized View) を選択します。
   b. [ファイル] – [再コンパイルして有効にする] を選択します。
   c. [ファイル] – [データの再表示] を選択します。

ヒント
テーブルの [カラム] タブを選択し、[ファイル] – [カラムを追加] を選択して、カラムを追加する方法もあります。
カラムを削除するには、[カラム] タブでカラムを選択し、[編集] – [削除] を選択します。
カラムをテーブルにコピーするには、右ウィンドウ枠の [カラム] タブでカラムを選択し、[コピー] をクリックします。対象のテーブルを選択し、右ウィンドウ枠の [カラム] タブをクリックして、次に [貼り付け] をクリックします。
また、[保存] をクリックするか、[ファイル] – [保存] を選択する必要があります。これを実行するまで、テーブルへの変更は行われません。

参照

- 「実体化ビュー (Materialized View) の有効化と無効化」 80 ページ
- 「データ整合性の確保」 95 ページ
- 「ビューの依存性」 67 ページ

テーブルの変更 (SQL の場合)

Interactive SQL では、ALTER TABLE 文を使用してテーブルを変更できます。依存している実体化ビュー (Materialized View) がある場合はテーブルの変更は失敗します。あらかじめ依存している実体化ビュー (Materialized View) を無効にしておく必要があります。テーブルの変更が完了したら、依存している実体化ビュー (Materialized View) をもう一度有効にする必要があります。

sa_dependent_views システム・プロシージャを使用して、依存している実体化ビュー (Materialized View) があるかどうかを決定します。「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』を参照してください。

ビューの依存性の詳細については、「ビューの依存性」 67 ページを参照してください。

◆ **既存のテーブルを変更するには、次の手順に従います (SQL の場合)。**

1. DBA ユーザとしてデータベースに接続します。
2. スキーマ変更操作を実行していて、従属した実体化ビュー (Materialized View) が存在する場合は、従属した各実体化ビュー (Materialized View) について ALTER MATERIALIZED VIEW ... DISABLE 文を使用して無効にします。従属した非実体化ビュー (Non-materialized View) を無効にする必要はありません。
3. ALTER TABLE 文を実行して、テーブルの変更を実行します。

 Copyright © 2007, iAnywhere Solutions, Inc.

データベース内のテーブルの定義が更新されます。

4. 実体化ビュー (Materialized View) をすべて無効にしたら、ALTER MATERIALIZED VIEW ... ENABLE 文を使用してもう一度有効にします。

例

これらの例は、データベースの構造を変更する方法を示しています。ALTER TABLE 文は、テーブルに関するほとんどすべての内容を変更できます。これによって、外部キーの追加と削除、カラム型の変更などを実行できます。このような場合には、一度変更を加えると、このテーブルを参照するストアド・プロシージャ、ビュー、その他のアイテムは機能しなくなる可能性があります。

次に示すコマンドは、スキルの説明を自由に書き込むカラムを Skills テーブルに追加します。

```
ALTER TABLE Skills
ADD SkillDescription CHAR( 254 );
```

ALTER TABLE 文を使用して、カラムの属性を変更することもできます。次の文は、SkillDescription カラムを最大 254 文字から最大 80 文字に変更します。

```
ALTER TABLE Skills
ALTER SkillDescription CHAR( 80 );
```

デフォルトで、80 文字より長いエントリが存在する場合は、エラーが発生します。string_rtruncation オプションを使用すると、この動作を変更できます。「string_rtruncation オプション [互換性]」『SQL Anywhere サーバ - データベース管理』を参照してください。

次の文は、SkillType カラムの名前を Classification に変更します。

```
ALTER TABLE Skills
RENAME SkillType TO Classification;
```

次の文は、Classification カラムを削除します。

```
ALTER TABLE Skills
DROP Classification;
```

次に示すコマンドは、テーブル全体の名前を変更します。

```
ALTER TABLE Skills
RENAME Qualification;
```

参照

- 「ALTER TABLE 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』
- 「ビューの変更」 63 ページ
- 「実体化ビュー (Materialized View) の有効化と無効化」 80 ページ
- 「ALTER MATERIALIZED VIEW 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「データ整合性の確保」 95 ページ
- 「ビューの依存性」 67 ページ

Copyright © 2007, iAnywhere Solutions, Inc.

テーブルの削除

この項では、データベースからテーブルを削除する方法について説明します。Sybase Central または Interactive SQL のいずれかを使用して、このタスクを実行できます。Interactive SQL では、テーブルの削除をテーブルのドロップとも呼びます。

SQL Remote 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、従属ビューを持つテーブルを削除するときは、以降の項で説明するように、追加の手順がある場合があります。

テーブルの削除とビューの依存性

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。従属した非実体化ビュー (Non-materialized View) 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。「ビューの変更」 63 ページを参照してください。

従属した実体化ビュー (Materialized View) が存在する場合、その定義は有効でなくなっているため、以降のリフレッシュは失敗します。この場合は、実体化ビュー (Materialized View) 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。「実体化ビュー (Materialized View) の作成」 75 ページを参照してください。

sa_dependent_views システム・プロシージャを使用すると、テーブルを変更する前に、テーブルにビューの依存性があるかどうかを判断できます。「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』を参照してください。

テーブルの削除によってビューの依存性にどのような影響があるかについては、「ビューの依存性」 67 ページを参照してください。

♦ **テーブルを削除するには、次の手順に従います (Sybase Central の場合)。**

1. DBA ユーザとして、またはテーブルの所有者としてデータベースに接続します。
2. 実体化ビュー (Materialized View) が依存するテーブルを削除している場合は、次のように各実体化ビュー (Materialized View) を無効にします。
   a. [ビュー] フォルダを開きます。
   b. 左ウィンドウ枠で、実体化ビュー (Materialized View) を選択します。
   c. [ファイル] – [無効にする] を選択します。
3. データベースの [テーブル] フォルダを開きます。
4. テーブルを選択し、[編集] – [削除] を選択します。

♦ **テーブルを削除するには、次の手順に従います (SQL の場合)。**

1. DBA ユーザとして、またはテーブルの所有者としてデータベースに接続します。

 Copyright © 2007, iAnywhere Solutions, Inc.

2. 実体化ビュー (Materialized View) が依存するテーブルを削除している場合は、ALTER MATERIALIZED VIEW ... DISABLE 文を使用して各実体化ビュー (Materialized View) を無効にします。
3. DROP TABLE 文を実行します。

例

次に示す DROP TABLE コマンドは、Skills テーブルからすべてのレコードを削除し、データベースから Skills テーブルの定義を削除します。

```
DROP TABLE Skills;
```

CREATE 文と同様、DROP 文はテーブルを削除する前と後に COMMIT 文を自動的に実行します。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。DROP 文では、テーブル上のインデックスもすべて削除されます。

詳細については、「DROP 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

テーブル内のデータのブラウズ

Sybase Central または InteractiveSQL を使用して、データベースのテーブルに保持されているデータをブラウズできます。

Sybase Central で作業している場合、テーブルのデータを表示するには、テーブルを選択し、右ウィンドウ枠の [データ] タブをクリックします。

Interactive SQL で作業している場合は、次の文を実行します。

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

Interactive SQL の [結果] タブ、または Sybase Central でテーブルの [データ] タブから、テーブル内のデータを編集できます。

プライマリ・キーの管理

ユニークな識別子である「**プライマリ・キー**」は、単一のカラム、またはカラムの組み合わせで構成され、ローのデータが存在するかぎりその値が変更されることはありません。良いデータベースを設計するにはこのユニーク性が重要であるため、テーブルを定義するときにプライマリ・キーを指定するのが最良の方法です。

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。

この項では、使用するデータベースでプライマリ・キーを作成したり編集したりする方法について説明します。Sybase Central または Interactive SQL のいずれかを使用して、これらのタスクを実行できます。

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 文のプライマリ・キー句や外部キー句によって決まります。カラムの順序は、CREATE TABLE 文のプライマリ・キー宣言で指定したカラム順序を基にしていません。

プライマリ・キーの管理 (Sybase Central の場合)

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、テーブル内のローをユニークに識別します。通常、プライマリ・キーは、テーブルの作成時に作成されます。ただし、後で修正することができます。Sybase Central では、次のいずれかの方法でテーブルのプライマリ・キーにアクセスします。

- テーブルを右クリックして、[プライマリ・キーの設定] を選択します。[プライマリ・キーの設定] ウィザードが開始されます。[プライマリ・キーの設定] ウィザードを使用すると、既存プライマリ・キーのカラムの順序を変更することもできます。
- 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] タブを選択します。

♦ 外部キーを設定するには、次の手順に従います (Sybase Central の場合)。

1. 左ウィンドウ枠で、[テーブル] フォルダを開きます。
2. プライマリ・キーを作成または修正するテーブルを選択し、[ファイル] – [プライマリ・キーの設定] を選択します。

   [プライマリ・キーの設定] ウィザードが表示されます。
3. ウィザードの説明に従って、プライマリ・キーを作成または修正します。
4. [プライマリ・キーの設定] ウィザードでプライマリ・キーを設定したら、[完了] を選択します。

   テーブルの新しいプライマリ・キー情報が自動的に保存されます。

プライマリ・キーの管理 (SQL の場合)

Interactive SQL では、CREATE TABLE 文と ALTER TABLE 文を使用して、プライマリ・キーを作成し、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。

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。このため、プライマリ・キーのカラムには、NOT NULL を指定します。

♦ プライマリ・キーを追加するには、次の手順に従います (SQL の場合)。

1. DBA ユーザとしてデータベースに接続します。
2. プライマリ・キーを設定するテーブルに対して、ALTER TABLE 文を実行します。

 Copyright © 2007, iAnywhere Solutions, Inc.

♦ プライマリ・キーを修正するには、次の手順に従います (SQL の場合)。

1. DBA ユーザとしてデータベースに接続します。
2. ALTER TABLE 文を実行して、既存のプライマリ・キーを削除します。
3. ALTER TABLE 文を実行して、プライマリ・キーを追加します。

♦ プライマリ・キーを削除するには、次の手順に従います (SQL の場合)。

1. DBA ユーザとしてデータベースに接続します。
2. DELETE PRIMARY KEY 句を使用した ALTER TABLE 文を実行します。

例 1

次の文は、前の例と同様に Skills テーブルを作成します。ただし、今回は SkillID カラムの値を使用してプライマリ・キーが追加されます。

```
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・キーを追加する必要があります。

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

詳細については、「ALTER TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』と「プライマリ・キーの管理 (Sybase Central の場合)」 52 ページを参照してください。

外部キーの管理

この項では、使用するデータベースで外部キーを作成したり編集したりする方法について説明します。Sybase Central または Interactive SQL のいずれかを使用して、これらのタスクを実行できます。

外部キーは、子テーブル (外部テーブル) の値と親テーブル (プライマリ・テーブル) の値の関連付けに使用されます。さまざまなタイプの情報とリンクした、複数の親テーブルを参照する複数の外部キーを、1 つのテーブルに入れることができます。

例

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。

この例では、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。

外部キーの管理 (Sybase Central の場合)

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。

外部キーの関係は、子テーブルの作成時 (つまり子テーブルにデータを挿入する前) に作成します。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。

外部キーを作成すると、右ウィンドウ枠の各テーブルの [制約] タブで追跡できます。このタブには、現在選択しているテーブルを参照する外部テーブルがすべて表示されます。

◆ **新しい外部キーを作成するには、次の手順に従います (Sybase Central の場合)。**

1. 外部キーを作成するテーブルを選択します。
2. 右ウィンドウ枠の [制約] タブをクリックします。
3. [ファイル] メニューから、[新規] – [外部キー] を選択します。

   [外部キー作成] ウィザードが表示されます。
4. ウィザードの指示に従います。

◆ **外部キーを削除するには、次の手順に従います (Sybase Central の場合)。**

1. 外部キーを削除するテーブルを選択します。

 Copyright © 2007, iAnywhere Solutions, Inc.

2. 右ウィンドウ枠の [制約] タブをクリックします。
3. 削除する外部キーを選択し、[編集] – [削除] を選択します。

指定したテーブルでは、外部キーを使用してそのテーブルを参照するテーブルのリストを表示することもできます。

♦ 指定されたテーブルを参照するテーブルのリストを表示するには、次の手順に従います (Sybase Central の場合)。

1. 左ウィンドウ枠で対象のテーブルを選択します。
2. 右ウィンドウ枠で、[参照元制約] タブをクリックします。

> **ヒント**
> 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] を選択します。
> [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] を選択すると、参照元外部キーのプロパティを表示できます。

外部キーの管理 (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。

テーブルには、プライマリ・キーは 1 つしか定義できませんが、外部キーは必要に応じていくつでも定義できます。

♦ 既存のテーブルの外部キーを変更するには、次の手順に従います (SQL の場合)。

1. DBA ユーザとしてデータベースに接続します。
2. ALTER TABLE 文を実行します。

例 1

この例では、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。

```
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 REFERENCES Skills ( Id )
);
```

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。

例 2

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 );
```

例 3

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、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;
```

詳細については、「ALTER TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Sybase Central では、[外部キー作成] ウィザード、または外部キーのプロパティ・シートで、プロパティを指定することもできます。「外部キーの管理 (Sybase Central の場合)」 54 ページを参照してください。

計算カラムの使用

計算カラムは、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。一致が起こらない特別値のリストについては、「特別値」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

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 ー 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、このクエリで 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。「述部の分析」 529 ページを参照してください。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。これは、次のように Shipments テーブルの計算カラム AverageCost を作成することで可能です。

```
ALTER TABLE Shipments
  ADD AverageCost DECIMAL(30,22)
  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、使用可能な 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 **NUMERIC(30,22)** 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。

計算カラム式の変更

ALTER TABLE 文を使用して、計算カラムで使用されている式を変更できます。次の文は、計算カラムに基づいている式を変更します。

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

カラムは、この文が実行されたときに再計算されます。新しい式が無効な場合は、ALTER TABLE 文は失敗します。

次の文を実行すると、カラムは計算カラムではなくなります。

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

この文を実行したとき、カラムの既存の値は変更されません。ただし、これ以降、自動的には更新されなくなります。

計算カラムの挿入と更新

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**　計算カラムに値を挿入するために INSERT 文や UPDATE 文を使用しないでください。そのような値は、意図した計算内容を反映しない可能性があります。また、計算カラムで手動で挿入または更新したデータは、カラムの再計算時に変更される可能性があります。
- **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。

INSERT、UPDATE、または LOAD TABLE 文を使用して計算カラムに値を挿入できますが、そのようなアプリケーションは想定しておらず、推奨できません。LOAD TABLE 文では、計算カラムの*オプション*計算が可能です。これは、複雑なアンロード／再ロード手順で DBA を支援するもので、COMPUTE 式が CURRENT TIMESTAMP などの非確定値を参照する場合に計算カラムが一定であるようなときに重要です。

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が、計算カラムは他と同様にベース・カラムなので、機能する場合に述部や式で直接参照される場合があります。

計算カラムは、問い合わせ時には再計算*されません*。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

Copyright © 2007, iAnywhere Solutions, Inc.

- テーブルの名前が変更された場合
- いずれかのカラムのデータ型、または COMPUTE 句が修正された場合
- ローが挿入された場合
- ローが更新された場合

時間に依存する式や、その他の方法でデータベースの状態に依存する式を使用している場合、計算カラムは適切な結果を返さない場合があります。

データベース内またはデータベース間でのテーブルまたはカラムのコピー

Sybase Central を使用すると、既存のテーブルまたはカラムをコピーし、同じデータベース内の別のロケーションか、まったく別のデータベースへそれらを挿入できます。

Sybase Central でのデータベース・オブジェクトのコピーについては、「SQL Anywhere プラグインのデータベース・オブジェクトのコピー」 『SQL Anywhere サーバ - データベース管理』を参照してください。

Sybase Central を使用しない場合は、次のいずれかを参照してください。

- 指定されたロケーションへ SELECT 文の結果を挿入するには、「SELECT 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
- 1 つのロー、またはデータベースのある場所から選択したローをテーブルに挿入するには、「INSERT 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ビューの編集

ビューは計算テーブルです。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、非実体化ビュー (Non-materialized View) (単にビューとも呼ばれる) と実体化ビュー (Materialized View) という 2 種類のビューをサポートします。これら 2 種類のビューは、非実体化ビュー (Non-materialized View) は呼び出しのたびに再計算されますが、実体化ビュー (Materialized View) は計算されてディスク上のベース・テーブルなどに格納され、定期的にデータをリフレッシュする必要があるという点が異なります。

実体化ビュー (Materialized View) の格納や管理はわずかに異なるため、そのタスクは別途説明します。実体化ビュー (Materialized View) については、「実体化ビュー (Materialized View) の編集」 72 ページを参照してください。

ビューには対応するステータスがあり、含まれるデータが最新であるかどうか、それらのデータはクエリへの応答に利用できるかどうかを示します。「ビューのステータス」 68 ページを参照してください。

実体化ビュー (Materialized View)、非実体化ビュー (Non-materialized View)、ベース・テーブルの比較

次の表に、ビューやテーブルで可能／不可能な操作を示します。

機能	実体化ビュー (Materialized View)	非実体化ビュー (Non-materialized View)	ベース・テーブル
アクセス・パーミッションの許可	○	○	○
自分に対する SELECT の許可	○	○	○
UPDATE の許可	×	一部	○
INSERT の許可	×	一部	○
DELETE の許可	×	一部	○
従属ビューの許可	○	○	○
インデックスの許可	○	×	○
整合性制約の許可	×	×	○
キーの許可	×	×	○

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** 非実体化ビュー (Non-materialized View) 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。実体化ビュー

 Copyright © 2007, iAnywhere Solutions, Inc.

(Materialized View) では、ディスク領域が必要ですが、呼び出しごとに再計算する必要はありません。これにより、特にデータベースが大きく、かつ同じテーブルをジョインする要求が頻繁に繰り返し発生し、データベース・サーバがその要求を処理するような環境では、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。

ビューの作成

データをブラウズするとき、SELECT 文は 1 つ以上のテーブルで動作して、結果セット、つまり同様にテーブルを作成します。ベース・テーブルと同様に、SELECT 文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。

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;
```

Sybase Central または Interactive SQL のいずれかを使用して、この文の結果を含んだビューを作成できます。

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。つまり、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。

♦ **新しいビューを作成するには、次の手順に従います (Sybase Central の場合)。**

1. データベースに接続します。
2. データベースの [ビュー] フォルダを開きます。
3. [ファイル] – [新規] – [ビュー] を選択します。

   [ビュー作成] ウィザードが表示されます。
4. ウィザードの指示に従います。

   新しいビューが [ビュー] フォルダに表示されます。
5. ウィザードの終了後、右ウィンドウ枠の [SQL] タブでビュー定義を編集できます。編集する場合は、変更を保存する必要があります。[ファイル] – [保存] を選択します。

♦ **新しいビューを作成するには、次の手順に従います (SQL の場合)。**

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

例

この項のはじめに示した、SELECT 文の結果を含む DepartmentSize ビューを作成します。

```
CREATE VIEW DepartmentSize AS
  SELECT DepartmentID, COUNT(*)
  FROM Employees
  GROUP BY DepartmentID;
```

詳細については、「CREATE VIEW 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ビューの使い方

SELECT 文に対する制限

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;
```

ビューの更新

UPDATE、INSERT、DELETE は、非実体化ビュー (Non-materialized View) を作成する SELECT 文の内容によっては、そのビューに対して使用できます。

COUNT(*) などの集合関数を含むビューは更新できません。SELECT 文で GROUP BY 句を含むビューと同じく、UNION 文を含むビューも更新できません。これらの場合には、UPDATE コマンドから基本となるテーブルに対する作業を行えないからです。

ビューのコピー

Sybase Central では、データベース間でビューをコピーできます。この作業を実行するには、Sybase Central の右ウィンドウ枠でビューを選択し、別の接続済みデータベースの [ビュー] フォルダへそれをドラッグします。新しいビューが作成されて、元のビューの定義がそこにコピーされます。新しいビューにコピーされるのは、ビューの定義だけであることに注意してください。パーミッションなど、その他のビューのプロパティはコピーされません。

 Copyright © 2007, iAnywhere Solutions, Inc.

WITH CHECK OPTION 句の使い方

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を制御するときに役立ちます。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
…	…	…	…

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・サーバがエラー「ベース・テーブル 'Employees' の挿入／更新に対して WITH CHECK OPTION が違反です。」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。

検査オプションの継承

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。

ビューの変更

ビューは Sybase Central または Interactive SQL を使用して変更できます。

Sybase Central では、ビュー、プロシージャ、関数の定義は、右ウィンドウ枠の各オブジェクトの [SQL] タブで変更できます。別のウィンドウでビューを編集するには、ビューを選択し、[ファイル] – [新しいウィンドウで編集] を選択します。Interactive SQL では、ALTER VIEW 文を使用して、ビューを変更できます。ALTER VIEW 文はビューの定義を新しい定義に置き換えますが、ビュー上のパーミッションはそのまま保持されます。

既存のビュー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。

データベース・オブジェクトの変更については、「データベース・オブジェクトのプロパティの設定」 36 ページを参照してください。

パーミッションの設定については、「テーブルに対するパーミッションの付与」『SQL Anywhere サーバ - データベース管理』と「ビューに対するパーミッションの付与」『SQL Anywhere サーバ - データベース管理』を参照してください。

パーミッションの取り消しについては、「ユーザ・パーミッションの取り消し」『SQL Anywhere サーバ - データベース管理』を参照してください。

ビューの変更とビューの依存性

ビューの定義を変更する場合、ビューに他のビューの依存性が存在するときは、ビューの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、ビューの定義と従属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。

ビューにビューの依存性が存在するかどうかを特定するには、sa_dependent_views システム・プロシージャを使用します。「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』を参照してください。

ビューの変更がビューの依存性に与える影響については、「ビューの依存性」 67 ページを参照してください。

♦ ビュー定義を変更するには、次の手順に従います (Sybase Central の場合)。

1. DBA またはビューの所有者としてデータベースに接続します。

2. [ビュー] フォルダを開きます。

3. 対象の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。

> **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。ビューを選択し、[ファイル] – [新しいウィンドウで編集] を選択して、別のウィンドウを開くことができます。

5. [ファイル] – [保存] を選択します。

 Copyright © 2007, iAnywhere Solutions, Inc.

♦ **ビュー定義を変更するには、次の手順に従います (SQL の場合)。**

1. DBA ユーザとして、またはビューの所有者としてデータベースに接続します。
2. ALTER VIEW 文を実行します。

例

この例では、ビューのスキーマ関連に変更を加える場合は、実際にはビューの定義を置き換えていることを示します。ここでは、ビュー定義のカラム名をよりわかりやすい名前に変更します。

```
CREATE VIEW DepartmentSize ( col1, col2 ) AS
  SELECT DepartmentID, COUNT( * )
  FROM Employees GROUP BY DepartmentID;
ALTER VIEW DepartmentSize ( DepartmentNumber, NumberOfEmployees ) AS
  SELECT DepartmentID, COUNT( * )
  FROM Employees GROUP BY DepartmentID;
```

次の例では、ビューの属性だけを変更する場合は、ビューを再定義する必要がないことを示します。ここでは、定義が非表示になるようにビューを設定します。

```
ALTER VIEW DepartmentSize SET HIDDEN;
```

ビューの変更の詳細については、「ALTER VIEW 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ビューの削除

Sybase Central と Interactive SQL のどちらを使用してもビューを削除できます。

ビューの削除とビューの依存性

従属ビューを持つ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、使用可能になりません。「ビューの変更」63 ページを参照してください。

ビューに依存するビューが存在するかどうかを特定するには、sa_dependent_views システム・プロシージャを使用します。「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』を参照してください。

ビューの依存性がビューの変更に与える影響については、「ビューの依存性」67 ページを参照してください。

♦ **ビューを削除するには、次の手順に従います (Sybase Central の場合)。**

1. DBA ユーザとして、またはビューの所有者としてデータベースに接続します。
2. [ビュー] フォルダを開きます。
3. ビューを選択し、[編集] − [削除] を選択します。

Copyright © 2007, iAnywhere Solutions, Inc.

♦ **ビューを削除するには、次の手順に従います (SQL の場合)。**

1. DBA またはビューの所有者としてデータベースに接続します。
2. DROP VIEW 文を実行します。

例

DepartmentSize ビューを削除します。

```
DROP VIEW DepartmentSize;
```

詳細については、「DROP 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ビューの有効化と無効化

この項では、非実体化ビュー (Non-materialized View) の有効化と無効化について説明します。実体化ビュー (Materialized View) の有効化と無効化については、「実体化ビュー (Materialized View) の有効化と無効化」 80 ページを参照してください。

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。ビューを無効にすると、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、ビューは使用できない状態になります。クエリが無効な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。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して起動できるように、明示的にもう一度有効にする必要があります。

ビューを無効にすると、そのビューを直接または間接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、ビューをもう一度有効にしたら、無効にした時点でそのビューに依存していたその他のビューもすべて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。「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』を参照してください。

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。失敗した場合、ビューは変更されないままです。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相互に一貫性があるように変更してから、ビューを有効にする必要があります。

> **注意**
> ビューを有効にする前に、そのビューが参照しているその他のビューが無効であればもう一度有効にする必要があります。

♦ **ビューを有効にするには、次の手順に従います (Sybase Central の場合)。**

1. DBA ユーザとして、またはビューの所有者としてデータベースに接続します。

 Copyright © 2007, iAnywhere Solutions, Inc.

2. データベースの [ビュー] フォルダを開きます。
3. 有効にするビューを選択します。
4. [ファイル] − [再コンパイルして有効にする] を選択します。

♦ ビューを無効にするには、次の手順に従います (Sybase Central の場合)。

1. DBA ユーザとして、またはビューの所有者としてデータベースに接続します。
2. データベースの [ビュー] フォルダを開きます。
3. 無効にするビューを選択します。
4. [ファイル] − [無効にする] を選択します。

♦ ビューを有効にするには、次の手順に従います (SQL の場合)。

1. DBA ユーザとして、またはビューの所有者としてデータベースに接続します。
2. ALTER VIEW ... ENABLE 文を実行します。

♦ ビューを無効にするには、次の手順に従います (SQL の場合)。

1. DBA ユーザとして、またはビューの所有者としてデータベースに接続します。
2. ALTER VIEW ... DISABLE 文を実行します。

例

次の例では、ビュー ViewSalesOrders が無効になります。

```
ALTER VIEW ViewSalesOrders DISABLE;
```

次の例では、ビュー ViewSalesOrders がもう一度有効になります。

```
ALTER VIEW 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』

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「**依存している**」と見なされます。

あるビューの参照先オブジェクトのセットには、そのビューが直接または間接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、他のビューやテーブルを参照している別のビューを参照できます。

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・サーバは任意の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無効な状態で参照元ビューに残らないようにします。これを実現するために、非実体化ビュー (Non-materialized View) の場合は、参照先オブジェクトのスキーマ変更が発生すると、データベース・サーバによってすべての参照元ビューが自動的に再コンパイルされます。実体化ビュー (Materialized View) の場合に、オブジェクトに有効な実体化ビュー (Materialized View) が存在すると、参照先オブジェクトのスキーマ変更が許可されません。

次は、ビューの依存性に関する考慮事項のリストです。

- 非実体化ビュー (Non-materialized View) は、テーブルやビューを参照できる。
- 実体化ビュー (Materialized View) は、ベース・テーブルだけを参照する。
- テーブルの削除前、またはテーブルやビューの変更前に、すべての従属した実体化ビュー (Materialized View) を無効または削除する必要がある。
- ビューを無効または削除すると、すべての従属した非実体化ビュー (Non-materialized View) が自動的に無効になる。
- ALTER TABLE 文の DISABLE VIEW DEPENDENCIES 句は、非実体化従属ビューだけを無効にする。すべての従属した実体化ビュー (Materialized View) は、明示的に無効または削除する必要がある。
- ビューを無効にすると、もう一度有効にするには明示的に行う必要がある。
- 非実体化ビュー (Non-materialized View) は参照先オブジェクトが無効になると無効になるが、実体化ビュー (Materialized View) は明示的に無効にする必要がある。

ビューのステータス

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

 Copyright © 2007, iAnywhere Solutions, Inc.

Central の左ウィンドウ枠で [ビュー] を選択し、右ウィンドウ枠で [ステータス] カラムの値を検査します。または、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] を選択して [ステータス] の値を検査します。

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。

  VALID 実体化ビュー (Materialized View) には、データが設定されていない可能性があります。これは、実体化ビュー (Materialized View) が一度も初期化されていない場合や、初期化やリフレッシュに失敗した場合に発生することがあります。データベース・サーバが未初期化状態の実体化ビュー (Materialized View) を参照する要求を受け取ると、初期化しようとします。

  注意
  データが古い実体化ビュー (Materialized View) でも、ステータスが VALID である可能性があります。ビューの有効性は、そのスキーマが基本となるオブジェクトのスキーマと一貫しているかどうかに関係があり、データの古さは関係ありません。

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

- **INVALID**　*このステータスは実体化ビュー (Materialized View) に適用されません。*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。

  非実体化ビュー (Non-materialized View) 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 (実体化ビュー (Materialized View) または非実体化ビュー (Non-materialized View)) を無効にした場合。

Copyright © 2007, iAnywhere Solutions, Inc.

- 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。

実体化ビュー (Materialized View) は、ALTER MATERIALIZED VIEW … DISABLE 文を実行するなど、明示的に無効にした場合だけ DISABLED ステータスになります。実体化ビュー (Materialized View) を無効にすると、そのビューのデータは削除されます。

ビューの有効化と無効化については、「ビューの有効化と無効化」 66 ページと「実体化ビュー (Materialized View) の有効化と無効化」 80 ページを参照してください。

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

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、実体化ビュー (Materialized View) 、またはカラムの削除
- テーブル、ビュー、実体化ビュー (Materialized View) 、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・タイプ、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

スキーマ変更操作を試みると、次のイベントが発生します。

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または間接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実体化ビュー (Materialized View) 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。従属した実体化ビュー (Materialized View) を無効にしてから、操作を続行してください。「実体化ビュー (Materialized View) の有効化と無効化」 80 ページを参照してください。
2. データベース・サーバは、修正するテーブルやビューと、すべての従属ビューに対して、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 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、

 Copyright © 2007, iAnywhere Solutions, Inc.

コンパイルに成功するために、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。

カタログ内のビューの依存性情報

データベース・サーバは、直接依存性を追跡します。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。

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記述します。「SYSDEPENDENCY システム・ビュー」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

sa_dependent_views システム・プロシージャを使用して、当該テーブルまたはビューに依存しているビューのリストを返すこともできます。「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』を参照してください。

ビュー内のデータのブラウズ

ビュー内のデータをブラウズするため、Interactive SQL を使用できます。Interactive SQL では、クエリを実行して、表示するデータを識別します。

これらのクエリの使用の詳細については、「クエリ：テーブルからのデータの選択」 283 ページを参照してください。

Sybase Central で作業している場合は、パーミッションを所有するビューを選択し、[ファイル] − [Interactive SQL によるデータ表示] を選択します。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。そのビューをブラウズできるようにするために、Interactive SQL は SELECT * FROM *owner.view* 文を実行します。

システム・テーブル・データの表示

システム・テーブル内のデータは、ビュー (具体的にはシステム・ビュー) を使用した場合だけ表示できます。システム・テーブルを直接問い合わせることはできません。例外はいくつかありますが、システム・テーブルのほとんどに対応するビューがあります。システム・ビューにはシステム・テーブルに似た名前が付けられていますが、先頭の「I」はありません。たとえば、データベース内のすべてのビューのリストは、ISYSTAB システム・テーブルに格納されます。この情報にアクセスするには、SYSTAB システム・ビューを使用します。同じ情報を取得するために、読みやすい形式の SYSVIEWS システム・ビューを使用することもできます (推奨)。

詳細については、「SYSTAB システム・ビュー」『SQL Anywhere サーバ - SQL リファレンス』と「SYSVIEWS 統合ビュー」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

SQL Anywhere に用意されているビューのリストと、それらに格納されている情報の種類に関する説明については、「ビュー」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

システム・ビューのデータは、Sybase Central または Interactive SQL を使ってブラウズできます。

♦ システム・ビューを使用してシステム・テーブルのデータを表示するには、次の手順に従います (Sybase Central の場合)。

1. DBA としてデータベースに接続します。
2. [ビュー] フォルダを開きます。
3. 対象にするシステム・テーブルに対応するビューを選択します。
4. 右側のウィンドウ枠で、[データ] タブに切り替えます。

♦ システム・ビューを使用してシステム・テーブルのデータを表示するには、次の手順に従います (SQL の場合)。

1. DBA としてデータベースに接続します。
2. 対象にするシステム・テーブルに対応するシステム・ビューを参照する SELECT 文を実行します。

例

ISYSTAB システム・テーブル内のデータを表示したいとします。このテーブルを直接問い合わせることはできないため、次の文では、対応する SYS.SYSTAB システム・ビュー内のすべてのデータが表示されます。

```
SELECT * FROM SYS.SYSTAB;
```

システム・テーブル内に存在するカラムが、対応するシステム・ビュー内に存在しないことがあります。特定のビューの定義を含むテキスト・ファイルを抽出する文の例を次に示します。

```
SELECT viewtext
FROM SYS.SYSVIEWS
WHERE viewname = 'SYSTAB';

OUTPUT TO viewtext.sql
FORMAT ASCII
ESCAPES OFF
QUOTE '';
```

実体化ビュー (Materialized View) の編集

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、「**実体化ビュー (Materialized View)**」を使用することを検討してください。実体化ビュー (Materialized View) 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。実体化ビュー (Materialized View) は、データベースが大きく、頻繁なクエリで大量のデータに対して集約操作とジョイン操作が発生し、かつ最新のデータにアクセスすることが重大な要件ではないような環境で、パフォーマンスを向上させることを目的に設計されています。

 Copyright © 2007, iAnywhere Solutions, Inc.

たとえば、実体化ビュー (Materialized View) 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。

実体化ビュー (Materialized View) とは、ベース・テーブルとよく似ていて、結果セットが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、実体化ビュー (Materialized View) はビューでもあり (クエリ指定がある)、テーブルでもあります (永続的な実体化ローがある)。このため、テーブルに対して実行する多くの操作を実体化ビュー (Materialized View) に対しても実行できます。たとえば、実体化ビュー (Materialized View) に対して、インデックス構築やアンロードを実行できます。

実体化ビュー (Materialized View) は、基本となるクエリの実行によってデータが設定されます。また、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。

実体化ビュー (Materialized View) のカラム統計は、テーブルの場合と同じ方法で生成され、管理されます。カラム統計の詳細については、「オプティマイザの推定とカラム統計」 524 ページを参照してください。

実体化ビュー (Materialized View) 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。

実体化ビュー (Materialized View) を使用するときの考慮事項

実体化ビュー (Materialized View) を使用する前に、次の要件や設定をよく検討してください。

- **ディスク領域の要件**　実体化ビュー (Materialized View) 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、作成する実体化ビュー (Materialized View) のサイズ分の領域を追加で割り付ける必要があることがあります。得られる利点が実体化ビュー (Materialized View) 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同時実行性の要件**　実体化ビュー (Materialized View) のデータは、定期的にリフレッシュする必要があります。次のような競合要因を考慮の上、実体化ビュー (Materialized View) をリフレッシュしなければならない頻度を判断する必要があります。
  - **基本となるデータの変更頻度**　データが頻繁に変更される場合は、ビューがリフレッシュされるとすぐに実体化ビュー (Materialized View) のデータは古くなります。
  - **リフレッシュのコスト**　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、実体化ビュー (Materialized View) の計算コストは非常に高くなることがあります。ビューのように頻繁にリフレッシュされると、データベース・サーバに対して許容できないレベルの負荷がかかる可能性があります。
  - **アプリケーションのデータ同時実行性の要件**　データベース・サーバがクエリに応答するために実体化ビュー (Materialized View) を使用するということは、データベース・サーバは古いデータをアプリケーションに提供する可能性があるということです。古いデータとは、データベースの現在の状態を表さなくなったデータです。古さの程度は、実体化ビュー (Materialized View) がリフレッシュ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。実体化ビュー (Materialized View) でデータの古さを管理する詳細については、「オプティマイザでの実体化ビュー (Materialized View) の古さ閾値の設定」 83 ページを参照してください。

- **データの一貫性の要件**　実体化ビュー (Materialized View) をリフレッシュするときは、実体化ビュー (Materialized View) をリフレッシュしなければならない一貫性を判断する必要があります。「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ - SQL リファレンス』の WITH ISOLATION LEVEL 句についての説明を参照してください。

- **最適化の使用**　クエリの実行時にオプティマイザが実体化ビュー (Materialized View) 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実体化ビュー (Materialized View) 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。「実行プランの解釈」 571 ページを参照してください。

  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実体化ビュー (Materialized View) が検討されたかどうかを判断できます。「アプリケーション・プロファイリング」 202 ページを参照してください。

  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。セキュリティ機能の詳細については、「トレーシング・レベルの選択」 218 ページを参照してください。

  オプティマイザが実体化ビュー (Materialized View) を使用する方法の詳細については、「実体化ビュー (Materialized View) によるパフォーマンスの向上」 531 ページを参照してください。

実体化ビュー (Materialized View) を管理するときの制限

実体化ビュー (Materialized View) を作成、初期化、リフレッシュするときや、後述のようにビュー・マッチング中は、次の制限が適用されます。

- 実体化ビュー (Materialized View) を作成するときは、実体化ビュー (Materialized View) 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。
- 実体化ビュー (Materialized View) を作成するときは、実体化ビュー (Materialized View) 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 (実体化ビュー (Materialized View) または非実体化ビュー (Non-materialized View)) 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
  - T-SQL 外部ジョイン
  - FOR XML 句
- 次のデータベース・オプションでは、実体化ビュー (Materialized View) 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、オプティマイザによってビューが使用されるためにも必要です。

Copyright © 2007, iAnywhere Solutions, Inc.

  - ansi_integer_overflow=On
  - ansinull=On
  - conversion_error=On
  - divide_by_zero_error=On
  - float_as_double=Off
  - sort_collation=Null
  - string_rtruncation=On
- 次のデータベース・オプションは、実体化ビュー (Materialized View) を作成するときに実体化ビュー (Materialized View) ごとに保存されます。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実体化ビュー (Materialized View) で保存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
  - uuid_has_hyphens
- ビューがリフレッシュされるときは、上記のオプションのすべての接続設定が無視されます。ただし、ビューの作成時は、これらの設定の値が使用されます。

実体化ビュー (Materialized View) 定義での ORDER BY 句の指定

実体化ビュー (Materialized View) 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、実体化ビュー (Materialized View) 定義で ORDER BY 句を指定すると、ビューが実体化されるときのローの順序にどのような影響があるかはわかりません。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。

これは、ORDER BY 句によって返される結果の順序が決まる非実体化ビュー (Non-materialized View) とは異なります。

実体化ビュー (Materialized View) と、オプティマイザによるビュー・マッチングについては、「実体化ビュー (Materialized View) によるパフォーマンスの向上」 531 ページを参照してください。

実体化ビュー (Materialized View) の作成

実体化ビュー (Materialized View) の作成時は、まずビューの定義、つまりスキーマを作成してデータベースに格納する必要があります。データベース・サーバは、適切にコンパイルできるかどうか定義を検証します。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。実体

化ビュー (Materialized View) の作成に成功したら、REFRESH MATERIALIZED VIEW 文を使用してデータを移植します。データの移植は、ビューの**初期化**とも呼ばれます。

実体化ビュー (Materialized View) を作成、初期化、またはリフレッシュする前に、実体化ビュー (Materialized View) の制限をすべて満たしていることを確認してください。「実体化ビュー (Materialized View) を管理するときの制限」 74 ページを参照してください。

データベース内にあるすべての実体化ビュー (Materialized View) とそのステータスのリストを取得するには、sa_materialized_view_info システム・プロシージャを使用します。「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』を参照してください。

実体化ビュー (Materialized View) の定義を作成すると、その定義はデータベースに格納され、新しい実体化ビュー (Materialized View) が [ビュー] フォルダに表示されます。ただし、この実体化ビュー (Materialized View) にはデータがありません。実体化ビュー (Materialized View) にデータを設定するには、初期化する必要があります。「実体化ビュー (Materialized View) の初期化」 77 ページを参照してください。

♦ 新しい実体化ビュー (Materialized View) を作成するには、次の手順に従います (Sybase Central の場合)。

1. DBA 権限または RESOURCE 権限を使用してデータベースに接続します。
2. [ビュー] フォルダを開きます。
3. [ファイル] メニューから、[新規] – [実体化ビュー] を選択します。

   [実体化ビュー作成] ウィザードが表示されます。
4. ウィザードの指示に従います。
5. 実体化ビュー (Materialized View) にデータが含まれるように、すぐに初期化する必要があります。「実体化ビュー (Materialized View) の初期化」 77 ページを参照してください。

♦ 実体化ビュー (Materialized View) を作成するには、次の手順に従います (SQL の場合)。

1. DBA 権限または RESOURCE 権限を使用してデータベースに接続します。
2. CREATE MATERIALIZED VIEW 文を実行します。データベース・サーバによってビュー定義が作成されてデータベースに格納され、ビューのステータスが ENABLED に設定されます。
3. 実体化ビュー (Materialized View) にデータが含まれるように、すぐに初期化する必要があります。「実体化ビュー (Materialized View) の初期化」 77 ページを参照してください。

例

次の文は、実体化ビュー (Materialized View) EmployeeConfidential を作成します。このビューには、従業員に関する機密情報が含まれます。

```
CREATE MATERIALIZED VIEW EmployeeConfidential AS
  SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary, ManagerID,
    Departments.DepartmentName, Departments.DepartmentHeadID
  FROM Employees, Departments
```


Copyright © 2007, iAnywhere Solutions, Inc.

```
WHERE Employees.DepartmentID=Departments.DepartmentID
ORDER BY Employees.DepartmentID;
```

参照

- 「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』
- 「SQL Anywhere サンプル・データベース」『SQL Anywhere 10 - 紹介』

実体化ビュー (Materialized View) の初期化

初期化していない実体化ビュー (Materialized View) は、データがなく、データベース内の定義として存在します。実体化ビュー (Materialized View) は、作成直後または再有効化直後の未初期化状態です。リフレッシュに失敗した場合も、実体化ビュー (Materialized View) は未初期化状態に戻ります。データベース・サーバが利用できるように、実体化ビュー (Materialized View) を初期化する必要があります。

実体化ビュー (Materialized View) を作成、初期化、またはリフレッシュする前に、実体化ビュー (Materialized View) の制限をすべて満たしていることを確認してください。「実体化ビュー (Materialized View) を管理するときの制限」 74 ページを参照してください。

♦ 実体化ビュー (Materialized View) を初期化するには、次の手順に従います (Sybase Central の場合)。

1. DBA としてデータベースに接続します。または、実体化ビュー (Materialized View) で INSERT パーミッションを使用して接続します。
2. [ビュー] フォルダを開きます。
3. 実体化ビュー (Materialized View) を選択し、[ファイル] – [データの再表示] を選択します。

♦ 実体化ビュー (Materialized View) を初期化するには、次の手順に従います (SQL の場合)。

1. DBA としてデータベースに接続します。または、実体化ビュー (Materialized View) で INSERT パーミッションを使用して接続します。
2. REFRESH MATERIALIZED VIEW 文を実行します。

例

この例では、EmployeeConfidential 実体化ビュー (Materialized View) が初期化されます。

```
REFRESH MATERIALIZED VIEW EmployeeConfidential;
```

注意

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実体化ビュー (Materialized View) を 1 回で初期化することもできます。「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』を参照してください。

参照

- 「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』
- 「実体化ビュー (Materialized View) の有効化と無効化」 80 ページ

実体化ビュー (Materialized View) のリフレッシュ

実体化ビュー (Materialized View) 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。実体化ビュー (Materialized View) のデータの古さについて許容可能な程度を検討し、それに応じてリフレッシュ方式を自動化してください。リフレッシュ処理中はビューをクエリ処理で利用できないため、リフレッシュ方式でビューのリフレッシュにかかる時間を考慮する必要があります。ビューをリフレッシュする頻度が決まったら、ビューをリフレッシュするイベントをスケジュールして作成できます。

実体化ビュー (Materialized View) 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、データベース内の実体化ビュー (Materialized View) を再表示することをおすすめします。

接続の独立性レベルは、実体化ビュー (Materialized View) 内のデータをリフレッシュするときに使用されます。

materialized_view_optimization データベース・オプションを使用して、古さの閾値を設定することもできます。オプティマイザは、クエリの処理時にこの閾値を超えた実体化ビュー (Materialized View) を使用しません。「オプティマイザでの実体化ビュー (Materialized View) の古さ閾値の設定」 83 ページを参照してください。

実体化ビュー (Materialized View) を作成、初期化、またはリフレッシュする前に、実体化ビュー (Materialized View) の制限をすべて満たしていることを確認してください。「実体化ビュー (Materialized View) を管理するときの制限」 74 ページを参照してください。

♦ ビューをリフレッシュするには、次の手順に従います (Sybase Central の場合)。

1. DBA としてデータベースに接続します。または、実体化ビュー (Materialized View) で INSERT パーミッションを使用して接続します。
2. データベースの [ビュー] フォルダを開きます。
3. 実体化ビュー (Materialized View) を選択し、[ファイル] – [データの再表示] を選択します。

♦ 実体化ビュー (Materialized View) をリフレッシュするには、次の手順に従います (SQL の場合)。

1. DBA としてデータベースに接続します。または、実体化ビュー (Materialized View) で INSERT パーミッションを使用して接続します。
2. REFRESH MATERIALIZED VIEW 文を実行します。

Copyright © 2007, iAnywhere Solutions, Inc.

例

次の文は、EmployeeConfidential 実体化ビュー (Materialized View) をリフレッシュします。

```
REFRESH MATERIALIZED VIEW EmployeeConfidential;
```

参照

- 「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 [データベース]」『SQL Anywhere サーバ - データベース管理』

実体化ビュー (Materialized View) の暗号化と復号化

セキュリティを高めるために、実体化ビュー (Materialized View) 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実体化ビュー (Materialized View) に含まれる場合、その実体化ビュー (Materialized View) も暗号化する状況も考えられます。実体化ビュー (Materialized View) を暗号化するには、データベースでテーブルの暗号化が先に有効にされている必要があります。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、実体化ビュー (Materialized View) 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、実体化ビュー (Materialized View) 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。

♦ 実体化ビュー (Materialized View) を暗号化するには、次の手順に従います (Sybase Central の場合)。

1. DBA としてデータベースに接続します。
2. [ビュー] フォルダを開き、実体化ビュー (Materialized View) を右クリックします。
3. [プロパティ] を選択します。

   実体化ビュー (Materialized View) の [プロパティ] ページが表示されます。
4. [その他] タブで、[ビュー・データは暗号化済み] チェックボックスをオンにして、[OK] を選択します。

♦ 実体化ビュー (Materialized View) を復号化するには、次の手順に従います (Sybase Central の場合)。

1. DBA としてデータベースに接続します。
2. [ビュー] フォルダを開き、実体化ビュー (Materialized View) を右クリックします。
3. [プロパティ] を選択します。

Copyright © 2007, iAnywhere Solutions, Inc.

実体化ビュー (Materialized View) の [プロパティ] ページが表示されます。

4. [その他] タブで、[ビュー・データは暗号化済み] チェックボックスをオフにして、[OK] を選択します。

♦ **実体化ビュー (Materialized View) を暗号化するには、次の手順に従います (SQL の場合)。**

1. DBA または実体化ビュー (Materialized View) の所有者としてデータベースに接続します。
2. ENCRYPTED 句を使用して ALTER MATERIALIZED VIEW 文を実行します。

♦ **実体化ビュー (Materialized View) を復号化するには、次の手順に従います (SQL の場合)。**

1. DBA または実体化ビュー (Materialized View) の所有者としてデータベースに接続します。
2. NOT ENCRYPTED 句を使用して ALTER MATERIALIZED VIEW 文を実行します。

例

次の文は、EmployeeConfidential 実体化ビュー (Materialized View) を暗号化します。この文が動作するために、暗号化済みのテーブルを許可するようにデータベースを設定済みである必要があります。

```
ALTER MATERIALIZED VIEW EmployeeConfidential ENCRYPTED;
```

次の文は、EmployeeConfidential 実体化ビュー (Materialized View) を復号化します。

```
ALTER MATERIALIZED VIEW EmployeeConfidential NOT ENCRYPTED;
```

参照

♦ 「テーブル暗号化を有効にする」『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』

実体化ビュー (Materialized View) の有効化と無効化

実体化ビュー (Materialized View) 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。また、無効になった実体化ビュー (Materialized View) は、最適化時にオプティマイザによって検討されません。クエリが無効な実体化ビュー (Materialized View) 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。実体化ビュー (Materialized View) 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。実体化ビュー (Materialized View) をもう一度有効にすると未初期化状態であるため、データを移植するためにそのビューをリフレッシュするする必要があります。

実体化ビュー (Materialized View) に依存する非実体化ビュー (Non-materialized View) は、実体化ビュー (Materialized View) 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、実体化ビュー (Materialized View) 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、実体化ビュー (Materialized View) 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

 Copyright © 2007, iAnywhere Solutions, Inc.

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。

実体化ビュー (Materialized View) を無効にすると、そのビューのすべてのインデックスも削除されるため、ビューをもう一度有効にした場合は、必要に応じてインデックスを再作成してください。

♦ 実体化ビュー (Materialized View) を有効にするには、次の手順に従います (Sybase Central の場合)。

1. DBA または実体化ビュー (Materialized View) の所有者としてデータベースに接続します。
2. データベースの [ビュー] フォルダを開きます。
3. 実体化ビュー (Materialized View) を選択し、[ファイル] – [再コンパイルして有効にする] を選択します。

♦ 実体化ビュー (Materialized View) を削除するには、次の手順に従います (Sybase Central の場合)。

1. DBA または実体化ビュー (Materialized View) の所有者としてデータベースに接続します。
2. データベースの [ビュー] フォルダを開きます。
3. 実体化ビュー (Materialized View) を選択し、[ファイル] – [無効にする] を選択します。

♦ 実体化ビュー (Materialized View) を有効にするには、次の手順に従います (SQL の場合)。

1. DBA または実体化ビュー (Materialized View) の所有者としてデータベースに接続します。
2. ALTER MATERIALIZED VIEW ... ENABLE 文を実行します。
3. REFRESH MATERIALIZED VIEW を実行して、ビューを初期化し、データを移植します。

♦ 実体化ビュー (Materialized View) を無効にするには、次の手順に従います (SQL の場合)。

1. DBA または実体化ビュー (Materialized View) の所有者としてデータベースに接続します。
2. ALTER MATERIALIZED VIEW ... DISABLE 文を実行します。

例

次の例では、EmployeeConfidential 実体化ビュー (Materialized View) が無効になります。実体化ビュー (Materialized View) のデータは削除されますが、実体化ビュー (Materialized View) の定義はデータベース内に残ります。データベース・サーバは実体化ビュー (Materialized View) を使用できなくなり、従属ビューが存在する場合はそのビューも無効になります。

```
ALTER MATERIALIZED VIEW EmployeeConfidential DISABLE;
```

次の 2 つの文は、それぞれ EmployeeConfidential 実体化ビュー (Materialized View) をもう一度有効にし、データを移植します。

```
ALTER MATERIALIZED VIEW EmployeeConfidential ENABLE;
REFRESH MATERIALIZED VIEW EmployeeConfidential;
```

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 MATERIALIZED VIEW 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』
- 「ビューの依存性」67 ページ
- 「SYSDEPENDENCY システム・ビュー」『SQL Anywhere サーバ - SQL リファレンス』

オプティマイザによる実体化ビュー (Materialized View) の使用の有効化と無効化

オプティマイザは、最適化処理で使用できる実体化ビュー (Materialized View) のリストを管理します。実体化ビュー (Materialized View) の定義にオプティマイザが拒否するような特定の要素が含まれる場合、または実体化ビュー (Materialized View) が使用するには古すぎる場合、そのビューは最適化で使用される候補にはなりません。最適化処理で実体化ビュー (Materialized View) が候補と見なされる条件については、「実体化ビュー (Materialized View) によるパフォーマンスの向上」531 ページを参照してください。

デフォルトでオプティマイザは実体化ビュー (Materialized View) を使用できます。ただし、実体化ビュー (Materialized View) がクエリで明示的に参照されないかぎり、オプティマイザによる実体化ビュー (Materialized View) の使用を無効にできます。

◆ 最適化で実体化ビュー (Materialized View) の使用を有効にするには、次の手順に従います (Sybase Central の場合)。

1. DBA としてデータベースに接続します。
2. [ビュー] フォルダを開き、実体化ビュー (Materialized View) を右クリックします。
3. [プロパティ] を選択します。

   実体化ビュー (Materialized View) の [プロパティ] ページが表示されます。
4. [一般] タブで、[最適化に使用] チェックボックスをオンにして、[OK] を選択します。

◆ 最適化で実体化ビュー (Materialized View) の使用を無効にするには、次の手順に従います (Sybase Central の場合)。

1. DBA としてデータベースに接続します。
2. [ビュー] フォルダを開き、実体化ビュー (Materialized View) を右クリックします。
3. [プロパティ] を選択します。

   実体化ビュー (Materialized View) の [プロパティ] ページが表示されます。
4. [一般] タブで、[最適化に使用] チェックボックスをオフにして、[OK] を選択します。

Copyright © 2007, iAnywhere Solutions, Inc.

♦ 最適化で実体化ビュー (Materialized View) の使用を有効にするには、次の手順に従います (SQL の場合)。

1. DBA または実体化ビュー (Materialized View) の所有者としてデータベースに接続します。
2. ENABLE USE IN OPTIMIZATION 句を使用して ALTER MATERIALIZED VIEW 文を実行します。

♦ 最適化で実体化ビュー (Materialized View) の使用を無効にするには、次の手順に従います (SQL の場合)。

1. DBA または実体化ビュー (Materialized View) の所有者としてデータベースに接続します。
2. DISABLE USE IN OPTIMIZATION 句を使用して ALTER MATERIALIZED VIEW 文を実行します。

例

次の文は、最適化で EmployeeConfidential ビューの使用を有効にします。

```
ALTER MATERIALIZED VIEW EmployeeConfidential ENABLE USE IN OPTIMIZATION;
```

次の文は、最適化で EmployeeConfidential ビューの使用を無効にします。

```
ALTER MATERIALIZED VIEW EmployeeConfidential DISABLE USE IN OPTIMIZATION;
```

参照

♦ 「ALTER MATERIALIZED VIEW 文」『SQL Anywhere サーバ - SQL リファレンス』

オプティマイザでの実体化ビュー (Materialized View) の古さ閾値の設定

実体化ビュー (Materialized View) 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。materialized_view_optimization データベース・オプションを使用すると、古さの閾値を設定できます。オプティマイザは、クエリの処理時にこの閾値を超えた実体化ビュー (Materialized View) を使用しなくなります。ただし、クエリが実体化ビュー (Materialized View) を直接参照している場合は、古さに関係なくビューがクエリの処理に使用されます。また、クエリの SELECT 文には、materialized_view_optimization データベース・オプションの設定を上書きする OPTION 句が含まれることがあります。

実体化ビュー (Materialized View) 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。その場合は、ビューをリフレッシュするイベントに指定した間隔を調整してください。

> **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、トランザクションのスナップショットの開始後に実体化ビュー (Materialized View) がリフレッシュされると、オプティマイザはその実体化ビュー (Materialized View) を使用しません。

materialized_view_optimization データベース・オプションの使用方法については、「materialized_view_optimization オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

SELECT 文で OPTION を使用して materialized_view_optimization データベース・オプションを上書きする方法については、「SELECT 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

イベントとトリガの使用については、「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』を参照してください。

実体化ビュー (Materialized View) がオプティマイザによって検討されたかどうかを特定する方法については、「実行プランの解釈」571 ページと「クエリのパフォーマンスのモニタ」260 ページを参照してください。

実体化ビュー (Materialized View) を隠す

実体化ビュー (Materialized View) の定義をユーザから隠すことができます。実体化ビュー (Materialized View) を隠すときは、データベースに格納されたビュー定義を難読化します。これにより、カタログでビューが非表示になります。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。実体化ビュー (Materialized View) を隠すと、デバッガを使用したデバッグでは、ビュー定義は表示されなくなり、プロシージャのプロファイリングで定義を利用できなくなります。ただし、ビューをアンロードして他のデータベースに再ロードすることはできます。

実体化ビュー (Materialized View) を隠す操作は元に戻せず、SQL 文を使用した場合だけ実行できます。

♦ **実体化ビュー (Materialized View) を隠すには、次の手順に従います (SQL の場合)。**

1. DBA または実体化ビュー (Materialized View) の所有者としてデータベースに接続します。
2. SET HIDDEN 句を使用して ALTER MATERIALIZED VIEW 文を実行します。

例

次の文は、EmployeeConfidential 実体化ビュー (Materialized View) を隠します。

```
ALTER MATERIALIZED VIEW EmployeeConfidential SET HIDDEN;
```

参照

- 「ALTER MATERIALIZED VIEW 文」『SQL Anywhere サーバ - SQL リファレンス』

実体化ビュー (Materialized View) の削除

不要になった実体化ビュー (Materialized View) は削除できます。

 Copyright © 2007, iAnywhere Solutions, Inc.

実体化ビュー (Materialized View) の削除とビューの依存性

実体化ビュー (Materialized View) を削除すると、そのビューに依存するすべてのビューは INVALID になります。従属ビューを使用するには、従属ビューの定義を変更するか、削除した実体化ビュー (Materialized View) をもう一度作成する必要があります。

実体化ビュー (Materialized View) にビューの依存性が存在するかどうかを特定するには、sa_dependent_views システム・プロシージャを使用します。「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』を参照してください。

実体化ビュー (Materialized View) の削除がビューの変更に与える影響については、「ビューの依存性」 67 ページを参照してください。

◆ 実体化ビュー (Materialized View) を削除するには、次の手順に従います (Sybase Central の場合)。

1. DBA またはビューの所有者としてデータベースに接続します。
2. データベースの [ビュー] フォルダを開きます。
3. 実体化ビュー (Materialized View) を選択し、[ファイル] – [削除] を選択します。

◆ 実体化ビュー (Materialized View) を削除するには、次の手順に従います (SQL の場合)。

1. DBA またはビューの所有者としてデータベースに接続します。
2. DROP MATERIALIZED VIEW 文を実行します。

例

次の文は、EmployeeConfidential 実体化ビュー (Materialized View) を削除します。

```
DROP MATERIALIZED VIEW EmployeeConfidential;
```

参照

- 「DROP 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』
- 「ビューの依存性」 67 ページ

実体化ビュー (Materialized View) の情報の取得

一般的な情報

実体化ビュー (Materialized View) のステータスなどの情報を要求するには、sa_materialized_view_info システム・プロシージャを使用します。「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』を参照してください。

実体化ビュー (Materialized View) のステータスの詳細については、「ビューのステータス」 68 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

データベース・オプションの情報

クエリで実体化ビュー (Materialized View) の名前を使用すると、実体化ビュー (Materialized View) の作成時に格納されたオプションを SYSMVOPTION システム・ビューから取得できます。たとえば、EmployeeConfidential 実体化ビュー (Materialized View) のオプションを探すには、次の文を実行します。

```
SELECT option_name, option_value
FROM SYSMVOPTION JOIN SYSMVOPTIONNAME
WHERE SYSMVOPTION.view_object_id=(
  SELECT object_id FROM SYSTAB
  WHERE table_name='EmployeeConfidential' )
ORDER BY option_name;
```

依存性の情報

実体化ビュー (Materialized View) におけるビューの依存性のリストを特定するには、sa_dependent_views システム・プロシージャを使用します。「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』を参照してください。

この情報は、SYSDEPENDENCY システム・ビューで探すこともできます。「SYSDEPENDENCY システム・ビュー」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

インデックスの編集

データベースの設計と作成では、パフォーマンスは重要な要素です。インデックスを使用すると、特定のローまたはローの特定のサブセットを検索する文のパフォーマンスを、大幅に向上させることができます。一方、インデックスを作成すると別途ディスク領域が必要になります。また、挿入、更新、削除が遅くなる場合があります。

インデックス・セットの選択

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。いくつかのインデックスによって得られるパフォーマンス上の利益は重要ですが、インデックスに伴うコストも記憶領域とデータ変更時のオーバヘッドの両方においてあります。

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。

インデックス・コンサルタントの詳細については、「インデックス・コンサルタントの使用」 211 ページを参照してください。

インデックスはいつ使うか

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。たとえば、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。

適切なインデックスが存在し、使用するとパフォーマンスが向上する場合、データベース・サーバのクエリ・オプティマイザは自動的にインデックスを使用します。

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。

インデックス・コンサルタントの詳細については、「インデックス・コンサルタントの使用」 211 ページを参照してください。

頻繁に検索するカラムにインデックスを使う

インデックスには追加領域が必要です。また、インデックスによって、テーブルのデータを修正する INSERT、UPDATE、DELETE 文などのパフォーマンスが、わずかに低下することがあります。しかし、検索のパフォーマンスは大幅に向上するので、頻繁にデータを検索する場合は常にインデックスを使用することをおすすめします。

パフォーマンスの詳細については、「インデックスの使用」 271 ページを参照してください。

SQL Anywhere は、プライマリ・キー・カラムと外部キー・カラムのインデックスを自動的に作成します。したがって、自分でキー・カラムにインデックスを設定する必要はありません。カラムがキーの一部に過ぎない場合は、インデックスが有用なこともあります。

SQL Anywhere 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そのようにします。インデックスを作成したら、明示的に参照するひつようはありません。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。

インデックス・ヒント

SQL Anywhere では、ユーザがインデックス・ヒントを指定できます。これは、特定のインデックスを強制的に使用させるために、オプティマイザによるクエリ・アクセス・プランの選択を上書きするものです。この機能は、上級ユーザやデータベース管理者のみが使用してください。これにより、パフォーマンスが低下する場合もあります。

インデックス・ヒントを指定するには、FORCE INDEX または WITH INDEX 句をクエリに追加します。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。

データベース・オブジェクト・プロパティの変更については、「データベース・オブジェクトのプロパティの設定」 36 ページを参照してください。

クラスタード・インデックスの使用

インデックスを使用すると、特定のローまたはローの特定のサブセットを検索する文のパフォーマンスを大幅に向上させることができますが、インデックスに連続して現れる 2 つのローは、データベース内の同じテーブル・ページに現れるとはかぎりません。

インデックスのクラスタ化を宣言することで、インデックスの取得パフォーマンスをさらに向上させることができます。クラスタード・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。このため、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ード・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、クラスタード・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

 Copyright © 2007, iAnywhere Solutions, Inc.

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。

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、その予期した物理的な隣接性を考慮するために予期したインデックス取得コストを修正することで、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。

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、ISYSPHYIDEX システム・テーブルのクラスタード・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。データベース・サーバがテーブルのローの非クラスタ化が大幅に進行したことを検出すると、それに応じてオプティマイザは予期したインデックス取得コストを調整します。

インデックスのクラスタ化プロパティは、いつでも追加または削除できます。クラスタード・インデックスと一致するようにテーブルのローを並べ替えるには、「REORGANIZE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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、UNIQUE 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ード・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、クラスタード・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。「UNLOAD 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- LOAD TABLE 文は、クラスタード・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。「LOAD TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- INSERT 文は、新しいローを挿入するときに、クラスタード・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。「INSERT 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- REORGANIZE TABLE 文はクラスタード・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。「REORGANIZE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

インデックスの作成

インデックスは、指定したテーブルの 1 つまたは複数のカラムに対して作成できます。インデックスはベース・テーブルまたはテンポラリ・テーブルに対して作成できますが、ビューに対しては作成できません。Sybase Central または Interactive SQL のいずれかを使用して、個々のインデックスを設定できます。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。

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。

♦ 指定されたテーブルに対する新しいインデックスを設定するには、次の手順に従います (Sybase Central の場合)。

1. DBA としてデータベースに接続します。
2. [テーブル] フォルダを開き、インデックスを作成するテーブルを選択します。
3. 右側のウィンドウ枠で、[インデックス] タブをクリックします。
4. [ファイル] – [新規] – [インデックス] を選択します。

   [インデックス作成] ウィザードが表示されます。
5. ウィザードの指示に従います。

   新しいインデックスがテーブルの [インデックス] タブに表示されます。また、[インデックス] フォルダにも表示されます。

♦ 指定されたテーブルに対する新しいインデックスを設定するには、次の手順に従います (SQL の場合)。

1. DBA ユーザとして、またはインデックスを作成するテーブルの所有者として、データベースに接続します。
2. CREATE INDEX 文を実行します。

テーブルの 1 つまたは複数のカラムに対してインデックスを作成するほかに、計算カラムを使用して組み込み関数に対してインデックスを作成できます。詳細については、「CREATE INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

例

次の例では、Surname カラムと GivenName カラムを使用して、Employees テーブルに EmployeeNames というインデックスを作成します。

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

詳細については、「CREATE INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』と「パフォーマンスのモニタリングと改善」199 ページを参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

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。この検査は、VALIDATE TABLE 文によって実行される妥当性検査を補完するものです。

警告
テーブルやデータベース全体の検証は、どの接続でもデータベースの変更が行われていないときに実施します。

♦ **インデックスを検証するには、次の手順に従います (Sybase Central の場合)。**

1. DBA ユーザとして、またはインデックスを作成するテーブルの所有者として、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] フォルダを開きます。
3. インデックスを選択し、[ファイル] – [検証] を選択します。

♦ **インデックスを検証するには、次の手順に従います (SQL の場合)。**

1. DBA ユーザとして、またはインデックスを作成するテーブルの所有者として、データベースに接続します。
2. VALIDATE INDEX 文を実行します。

♦ **インデックスを検証するには、次の手順に従います (コマンド・ラインを使用する場合)。**

1. コマンド・プロンプトを開きます。
2. dbvalid ユーティリティを実行します。

例

インデックス EmployeeNames を検証します。インデックス名の代わりにテーブル名を指定すると、プライマリ・キーのインデックスが検証されます。

```
VALIDATE INDEX EmployeeNames;
```

インデックス EmployeeNames を検証します。-i オプションは、そのオブジェクト名をインデックスとして指定します。

```
dbvalid -i EmployeeNames
```

詳細については、「VALIDATE 文」 『SQL Anywhere サーバ - SQL リファレンス』と「検証ユーティリティ (dbvalid)」 『SQL Anywhere サーバ - データベース管理』を参照してください。

インデックスの再構築

インデックスの再構築が必要になることがあります。たとえば、sa_index_density システム・プロシージャによって示されるような、時間の経過に伴ってインデックスに無駄が多くなる場合です。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。同じ物理インデックスを共有するすべての論理インデックスの再構築を実行する必要はありません。論理インデックスと物理インデックスの詳細については、「論理インデックスを使用したインデックスの共有」 593 ページを参照してください。

♦ **インデックスを再構築するには、次の手順に従います (Sybase Central の場合)。**

1. DBA ユーザとして、またはインデックスを作成するテーブルの所有者として、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] フォルダを開きます。
3. インデックスを選択し、[編集] – [再構築] を選択します。

♦ **インデックスを再構築するには、次の手順に従います (SQL の場合)。**

1. DBA ユーザとして、またはインデックスが関連付けられているテーブルの所有者としてデータベースに接続します。
2. ALTER INDEX ... REBUILD 文を実行します。

例

次の文は、Customers テーブルの IX_customer_name インデックスを再構築します。

```
ALTER INDEX IX_customer_name ON Customers REBUILD;
```

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』

インデックスの削除

Sybase Central または Interactive SQL では、不要になったインデックスをデータベースから削除できます。

♦ **インデックスを削除するには、次の手順に従います (Sybase Central の場合)。**

1. DBA ユーザとして、またはインデックスを作成するテーブルの所有者として、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] フォルダを開きます。
3. インデックスを選択し、[編集] – [削除] を選択します。

 Copyright © 2007, iAnywhere Solutions, Inc.

♦ **インデックスを削除するには、次の手順に従います (SQL の場合)。**

1. DBA ユーザとして、またはインデックスが関連付けられているテーブルの所有者としてデータベースに接続します。
2. DROP INDEX 文を実行します。

例

次の文は、データベースから EmployeeNames インデックスを削除します。

```
DROP INDEX EmployeeNames;
```

詳細については、「DROP 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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・ビューにあります。Sybase Central または Interactive SQL を使用すると、テーブルのビューに含まれるデータを確認するために、ビューをブラウズできます。

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、UNIQUE 制約、セカンダリ・インデックス) を定義します。「SYSIDX システム・ビュー」 『SQL Anywhere サーバ - SQL リファレンス』と「論理インデックスを使用したインデックスの共有」 593 ページを参照してください。

♦ **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。「SYSPHYSIDX システム・ビュー」 『SQL Anywhere サーバ - SQL リファレンス』と「論理インデックスを使用したインデックスの共有」 593 ページを参照してください。

♦ **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 つを記述するように、SYSIDXCOL システム・ビューの各ローは、SYSIDX システム・ビューで記述されるインデックスのカラム 1 つを記述します。「SYSINDEXES システム・ビュー」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

♦ **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。「SYSFKEY システム・ビュー」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

テンポラリ・テーブルの編集

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。

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。「グローバル」という語は、同じまたは異なるアプリケーションからの複数の接続が同時にテーブルを使用できるという意味です。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。

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

- 「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』

Copyright © 2007, iAnywhere Solutions, Inc.

第 3 章

データ整合性の確保

目次

データ整合性の概要 96
カラム・デフォルトの使い方 100
テーブル制約とカラム制約の使い方 106
ドメインの使い方 110
エンティティ整合性と参照整合性の確保 113
システム・テーブルの整合性ルール 117

データ整合性の概要

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。整合性確保のための制約をデータベースに組み入れると、データベースの信頼性は確実に上がります。

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。

プロシージャとトリガの詳細については、「プロシージャ、トリガ、バッチの使用」 777 ページを参照してください。

データが有効でない場合

次に、適切な検査が行われないとデータが有効でなくなる場合の例をいくつか示します。それぞれのケースは、この章で説明されている機能を使って防ぐことができます。

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

- 組織データベースの Departments テーブルに、別の 2 人が新しく同じ部 (DepartmentID は 200) を追加した。

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。

データベース内の整合性制約

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。この検査と規則を合わせたものが「**制約**」です。

Copyright © 2007, iAnywhere Solutions, Inc.

データベースの整合性制約の構築

データベース自体に組み込まれた制約は、クライアント・アプリケーション内に組み込まれた制約や、データベースのユーザに説明された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。

データベースのデータが変わる場合

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・テーブル中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。

データ整合性の支援策

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の日付を記録するデフォルト値を、現在の日付にする。
- 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。

カラムに対して設定できるデフォルトの詳細については、「カラム・デフォルトの使い方」 100 ページを参照してください。

制約

カラムやテーブルごとに、データに対して適用できる制約がいくつかあります。次に例を示します。

- 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。
- 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、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。
- トリガを使うと検査条件をより高度な形で適用できる。「プロシージャ、トリガ、バッチの使用」 777 ページを参照してください。

また、カラム制約をドメインから継承させることもできます。上記の制約を含む、テーブルやカラムの制約の詳細については、「テーブル制約とカラム制約の使い方」 106 ページを参照してください。

エンティティ整合性と参照整合性

関係は、プライマリ・キーと外部キーで定義され、リレーショナル・データベース・テーブル間の情報の橋渡しをします。関係は、データベースの設計に直接組み入れてください。次に示す整合性規則を使って、データベース構造を維持できます。

- **エンティティ整合性**　エンティティ整合性はプライマリ・キーを追跡します。これはテーブルの各ローが IS NOT NULL を保証するプライマリ・キーによってユニークに識別できることを保証します。
- **参照整合性**　参照整合性はテーブル間の関係を定義する外部キーを追跡します。これは、すべての外部キーが対応するプライマリ・キーの値に一致すること (NULL 値が許可されている場合には NULL も可) を保証します。

参照整合性の確保の詳細については、「エンティティ整合性と参照整合性の確保」 113 ページを参照してください。プライマリ・キーと外部キーの関係の正しい設計に関する詳細については、「データベースの設計」 3 ページを参照してください。

高度な整合性規則のためのトリガ

データ整合性の確保にはトリガも使用できます。「**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。

トリガの詳細については、「プロシージャ、トリガ、バッチの使用」 777 ページを参照してください。

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

- **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。

これらの文の構文の詳細については、「SQL 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

カラム・デフォルトの使い方

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。

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。
- デフォルトの 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)。
- 現在の日付、時刻、タイムスタンプ。
- データベース・ユーザの現在のユーザ ID。
- 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;
```

詳細については、「ALTER TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』と「CREATE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

カラム・デフォルトの変更と削除

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;
```

Sybase Central でカラム・デフォルトを設定する

Sybase Central では、カラム・デフォルトの追加、修正、削除をカラム・プロパティ・シートの [値] タブで行います。

♦ **カラムのプロパティ・シートを表示するには、次の手順に従います。**

1. データベースに接続します。
2. データベースの [テーブル] フォルダを開きます。
3. 変更するカラムのあ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、[カラム] タブをクリックします。
5. 対象のカラムを選択します。
6. [ファイル]－[プロパティ] を選択します。

   カラムのプロパティ・シートが表示されます。

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、現在の日付、現在の時刻、現在のタイムスタンプ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。

現在の日付をデフォルトにすると便利な例

次のような場合、現在の日付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。

現在のタイムスタンプ

現在のタイムスタンプは、現在の日付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、コンタクト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。デフォルトを現在のタイムスタンプに設定すると、それぞれの接触を区別しやすくなります。

タイムスタンプは日付と時間を 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便利です。

デフォルトのタイムスタンプ

デフォルトのタイムスタンプは、テーブル内の各ローが最後に変更された日付を示します。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。「CREATE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』の DEFAULT 句を参照してください。

タイムスタンプ、時刻、日付の詳細については、「SQL データ型」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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、マルチユーザの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)。「LAST USER 特別値」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

オートインクリメント・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制約されたカラムになります (「エンティティ整合性の確保」113 ページを参照)。

 Copyright © 2007, iAnywhere Solutions, Inc.

オートインクリメント・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。詳細については、「@@identity グローバル変数」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

オートインクリメントと負の値

オートインクリメントは正の値を想定しています。

テーブルが作成されたときの初期値は 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。

オートインクリメントと 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。

IDENTITY カラムについては、「特殊な IDENTITY カラム」 624 ページを参照してください。

GLOBAL AUTOINCREMENT デフォルト

GLOBAL AUTOINCREMENT デフォルトは、SQLRemote 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 $np + 1$ ー $(n+ 1) p$ 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 ー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。

Copyright © 2007, iAnywhere Solutions, Inc.

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。「イベントの概要」 『SQL Anywhere サーバ - データベース管理』を参照してください。

グローバル・オートインクリメント・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制約されたカラムになります (「エンティティ整合性の確保」 113 ページを参照)。

これ以外のケースにもグローバル・オートインクリメント・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逆に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、オーバフローが起こって負の値になることがあります。

オートインクリメント・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。詳細については、「@@identity グローバル変数」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

参照場所

- 「グローバル・オートインクリメントの使用」 『Mobile Link - サーバ管理』
- 「グローバル・オートインクリメント・デフォルト・カラム値の使用」 『SQL Remote』
- 「CREATE TABLE 文」 『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、テーブルのローをユニークに識別できます。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

 Copyright © 2007, iAnywhere Solutions, Inc.

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。

NULL 値の詳細については、「NULL 値」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。定数式では、「*本日より 15 日後*」といったエントリを、デフォルト値としてカラムに設定できます。この場合は次のように入力します。

```
... DEFAULT ( DATEADD( day, 15, GETDATE() ) );
```

テーブル制約とカラム制約の使い方

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。

警告
テーブルを修正する際には、他のデータベース・ユーザの処理が妨げられるおそれがあります。他のユーザがデータベースに接続していても ALTER TABLE 文を使用できますが、目的のテーブルが使用されていると ALTER TABLE 文は実行できません。また、大規模なテーブルの修正には時間がかかり、その間そのテーブルを使用できません。

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の使い方

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、非常識なデータの入力を防ぐ簡単なものであったり、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。

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。
有効な条件の詳細については、「探索条件」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

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。次の例では、City カラムにある決まった値 (たとえばその会社の事業所の所在地) 以外を入力できないように制約しています。

```
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。次に、制約を使って従業員の入社日 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。

```
ALTER TABLE Employees
ALTER StartDate
CHECK ( StartDate BETWEEN '1983/06/27'
            AND CURRENT DATE );
```

日付のフォーマットはいくつか用意されています。ここで使用した YYYY/MM/DD のフォーマットは、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の使い方

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);
```

詳細については、「ALTER TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ドメインのカラム検査制約の継承

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 posint INT
CHECK ( @col > 0 );
```

posint ドメインを使用して定義されたカラムは、検査制約が明示的に指定されていないかぎり、正の整数だけを受け付けます。@ 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。

ドメインの詳細については、「ドメイン」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Sybase Central でテーブル制約およびカラム制約を編集する

Sybase Central では、カラム制約の追加、変更、削除をテーブルやカラムのプロパティ・シートの [制約] タブで行います。

♦ **制約を管理するには、次の手順に従います。**

1. [テーブル] フォルダを開きます。
2. 右ウィンドウ枠で、変更するテーブルをダブルクリックします。
3. [制約] タブで、変更が必要な制約を変更します。

   たとえば、テーブル制約またはカラム制約を追加する場合は、[検査制約] タブをクリックして、[ファイル] – [新規] – [テーブル検査制約]、または [ファイル] – [新規] – [カラム検査制約] を選びます。

検査制約の変更と削除

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。

Copyright © 2007, iAnywhere Solutions, Inc.

```
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。詳細については、「Sybase Central でテーブル制約およびカラム制約を編集する」 108 ページを参照してください。

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、テーブルにデータを挿入する場合や、単に問い合わせるだけでも「**カラムが見つかりません**」というエラー・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。

ドメインの使い方

「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。値の許容範囲の制限には、通常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。

また、さまざまな目的のために独自のドメインを定義できます。

- エラーの多くは、不適当な値の入力を制限することで防げます。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、その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。
- ドメインを使用すると、アプリケーションやデータベース構造が非常にわかりやすくなります。
- ドメインは非常に便利です。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。

ドメインの詳細については、「SQL データ型」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ドメインの作成 (Sybase Central の場合)

Sybase Central を使って、ドメインを作成したり、それをカラムに割り当てたりすることができます。

♦ 新しいドメインを作成するには、次の手順に従います (Sybase Central の場合)。

1. [ドメイン] フォルダを選択します。
2. [ファイル] – [新規] – [ドメイン] を選びます。

   [ドメイン作成] ウィザードが表示されます。
3. ウィザードの指示に従います。

Sybase Central では、すべてのドメインは [ドメイン] フォルダに表示されます。

♦ ドメインをカラムに割り当てるには、次の手順に従います (Sybase Central の場合)。

1. 右ウィンドウ枠で対象のテーブルの [カラム] タブをクリックします。
2. 目的のカラムのデータ型カラムで、次のいずれかを実行します。

 Copyright © 2007, iAnywhere Solutions, Inc.

- ドロップダウン・リストからドメインを選ぶ。
- ドロップダウン・リストの横のボタンをクリックして、プロパティ・シートにあるドメインを選ぶ。

ドメインの作成 (SQL の場合)

CREATE DOMAIN 文は、ドメインの作成と定義に使用できます。

♦ 新しいドメインを作成するには、次の手順に従います (SQL の場合)。

1. データベースに接続します。
2. CREATE DOMAIN 文を実行します。

例 1 : 簡単なドメイン

氏名と住所をそれぞれカラムに入力するデータベースがあるとします。たとえば、次のようにドメインを定義してください。

```
CREATE DOMAIN persons_name CHAR(30)
CREATE DOMAIN street_address CHAR(35);
```

こうして定義されたドメインは、既存のデータ型と同様にいくらでも使用できます。たとえば、次のようにしてテーブルを定義できます。

```
CREATE TABLE Customers (
  ID INT  DEFAULT AUTOINCREMENT  PRIMARY KEY,
  Name persons_name,
  Street street_address);
```

例 2 : デフォルト値、検査制約、識別子

前述の例では、テーブルのプライマリ・キーを整数型に指定しました。実際に、同じような識別子を各テーブルに持たせる必要がある場合が数多くあります。こうしたアプリケーションでは、整数型を指定する代わりに、識別子ドメインを作成すると大変便利です。

ドメインを作成するときにデフォルト値や検査制約を設定し、このドメインが割り当てられたカラムに不適切な値が入力されないように設定できます。

整数はテーブル識別子によく使われます。識別子をユニークにするには、正の整数を使うことをおすすめします。このような識別子はさまざまなテーブルで使用できるので、次の例に示すようなドメインを作成するとよいでしょう。

```
CREATE DOMAIN identifier UNSIGNED INT
DEFAULT AUTOINCREMENT;
```

この定義を使用して、上記の Customers テーブルの定義を書き換えることができます。

```
CREATE TABLE Customers (
  ID identifier PRIMARY KEY,
  Name persons_name,
  Street street_address
);
```

例 3：組み込みドメイン

SQL Anywhere には、定義済みドメインがいくつかあります。このような定義済みドメインは、自作のドメインと同様に自由に使用できます。たとえば、次のような通貨データ・ドメインがすでに定義されています。

```
CREATE DOMAIN MONEY NUMERIC(19,4)
NULL;
```

詳細については、「CREATE DOMAIN 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ドメインの削除

Sybase Central または DROP DOMAIN 文を使用すると、ドメインを削除できます。

削除できるのは、DBA 権限を持つユーザ、またはそのドメインの作成者だけです。また、データベース内の変数やカラムに使用されているドメインは削除できないため、それを使用しているカラムや変数をすべて削除してから、ドメインの削除を行ってください。

♦ **ドメインを削除するには、次の手順に従います (Sybase Central の場合)。**

1. [ドメイン] フォルダを開きます。
2. ドメインをクリックし、[編集] – [削除] を選択します。

♦ **ドメインを削除するには、次の手順に従います (SQL の場合)。**

1. データベースに接続します。
2. DROP DOMAIN 文を実行します。

例

次の文は、persons_name ドメインを削除します。

```
DROP DOMAIN persons_name;
```

詳細については、「DROP 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

エンティティ整合性と参照整合性の確保

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は (データベース構造に定義されている) テーブル間の関係を適切に維持できます。

エンティティ整合性の確保

ローが挿入または更新されるたびに、データベース・サーバはそのテーブルのプライマリ・キーの有効性、つまりテーブルの各ローがプライマリ・キーを使ってユニークに識別できることを保証します。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

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。

プライマリ・キー作成の詳細については、「プライマリ・キーの管理」 51 ページを参照してください。

CREATE TABLE 文の構文に関する詳細については、「CREATE TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

テーブル構造の変更については、「ALTER TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

参照整合性の確保

外部キー (特定のカラムやカラムの組み合わせで構成) は、あるテーブル (**外部**テーブル) の情報を別のテーブル (**参照**テーブルまたは**プライマリ**・テーブル) のデータと関連付けます。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。

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。このようなキーを必須外部キーといいます。

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。

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

外部キー作成の詳細については、「プライマリ・キーの管理」 51 ページを参照してください。

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。

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パーミッションを使って実行されます。つまり、特にパーミッション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。

 Copyright © 2007, iAnywhere Solutions, Inc.

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。「SYSCONSTRAINT システム・ビュー」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。 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。「SYSCHECK システム・ビュー」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
SYS.SYSCHECK	SYS.SYSCHECK システム・ビューの各ローには、SYS.SYSCONSTRAINT システム・ビューにリストされている検査制約の定義が含まれています。「SYSCHECK システム・ビュー」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
SYS.SYSFKEY	SYS.SYSFKEY システム・ビューの各ローには、キーに定義した一致タイプなどの外部キーが記述されています。「SYSFKEY システム・ビュー」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
SYS.SYSIDX	SYS.SYSIDX システム・ビューの各ローには、データベース内の 1 つのインデックスの定義が含まれています。「SYSIDX システム・ビュー」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
SYS.SYSTRIGGER	SYS.SYSTRIGGER システム・ビューの各ローには、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 1 つのトリガが記述されています。 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。「SYSTRIGGER システム・ビュー」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

第 4 章

トランザクションと独立性レベル

目次

トランザクションの概要 120
独立性レベルと一貫性 125
トランザクションのブロックとデッドロック 140
独立性レベルの選択 143
独立性レベルのチュートリアル 147
ロックの仕組み 166
特殊な同時性の問題 181
まとめ 183

トランザクションの概要

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コマンドです。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コマンド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。

データベースのバックアップとデータ・リカバリの詳細については、「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』を参照してください。

データベースの同時使用の詳細については、「同時性の概要」 122 ページを参照してください。

トランザクションの使用

SQL Anywhere では、コマンド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。データを変更しても、その変更がすぐに永続的なものになるわけではありません。変更はトランザクション・ログに記録され、COMMIT コマンドを入力すると永続的なものになります。

 Copyright © 2007, iAnywhere Solutions, Inc.

トランザクションを開始または終了するコマンドや動作を理解すると、トランザクションを有効に使用できます。

トランザクションの開始

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

トランザクションの完了

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。データ定義コマンドの ALTER、CREATE、COMMENT、DROP はすべて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。

  オートコミットの詳細については、「オートコミットまたは手動コミット・モードの設定」『SQL Anywhere サーバ - プログラミング』を参照してください。

- 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。

  詳細については、「オートコミットまたは手動コミット・モードの設定」『SQL Anywhere サーバ - プログラミング』と「chained オプション [互換性]」『SQL Anywhere サーバ - データベース管理』を参照してください。

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。「auto_commit オプション [Interactive SQL]」『SQL Anywhere サーバ - データベース管理』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

- 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。「commit_on_exit オプション [Interactive SQL]」『SQL Anywhere サーバ - データベース管理』を参照してください。

> **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、Interactive SQL の設定は ODBC の設定によって上書きされます。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 コマンドをサポートし、Sybase Adaptive Server Enterprise との互換性を保ちます。詳細については、「Transact-SQL との互換性」610 ページを参照してください。

詳細については、「Transact-SQL との互換性」610 ページを参照してください。

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
```

同時性の概要

「同時性」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。

 Copyright © 2007, iAnywhere Solutions, Inc.

例

デパートのデータベース・システムでは、多くの店員が同時に顧客アカウントを更新する必要があります。各店員は、顧客に対応するときにアカウントの状態を更新できなければなりません。データベースを使用している人が誰もいなくなるまで待つことはできません。

同時性に関する知識の必要な方

同時性は、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。

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 コマンドでは、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。

Copyright © 2007, iAnywhere Solutions, Inc.

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。

Copyright © 2007, iAnywhere Solutions, Inc.

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動作が、同時に実行される他のトランザクションの動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。「FROM 句」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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込み	♦ 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。
1−コミットされた読み込み	♦ 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。
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。
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。

Copyright © 2007, iAnywhere Solutions, Inc.

独立性レベル	特性
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。
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] データベースでこれらの独立性レベルを使用するには、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。「スナップショット・アイソレーションの有効化」 129 ページを参照してください。

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込み) 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、パフォーマンスは競合に関して独立性レベルを 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悪くなります。

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と、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あります。同じトランザクション内の複数のカーソルは相互に干渉する可能性があります。

 Copyright © 2007, iAnywhere Solutions, Inc.

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、ブロックや場合によっては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、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。

スナップショット・アイソレーションは、次のような多くの場合で役立ちます。

- **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。

次の文は、スナップショット・アイソレーション内で実行できません。

- ALTER INDEX
- ALTER TABLE
- CREATE INDEX
- DROP INDEX
- REFRESH MATERIALIZED VIEW
- REORGANIZE TABLE

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。「TRUNCATE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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更新された場合、実体化ビュー (Materialized View) 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、次の項を参照してください。

- 「isolation_level オプション [互換性]」『SQL Anywhere サーバ - データベース管理』
- ADO と OLE DB :「トランザクションの使用」『SQL Anywhere サーバ - プログラミング』
- ADO.NET :「IsolationLevel プロパティ」『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が確認できます。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、基礎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' );
```

 Copyright © 2007, iAnywhere Solutions, Inc.

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。「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』を参照してください。

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。「システム・イベントの概要」 『SQL Anywhere サーバ - データベース管理』を参照してください。

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・アイソレーションでは、BEGIN TRANSACTION 文でトランザクションが開始しません。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
```

スナップショット・アイソレーションの有効化

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。

Copyright © 2007, iAnywhere Solutions, Inc.

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。「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』を参照してください。

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の完了後にスナップショット・アイソレーションが有効となる。
- ♦ **in_transition_to_off** 現在のトランザクションの完了後にスナップショット・アイソレーションが無効となる。

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。

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。

♦ **スナップショット・アイソレーションを使用するには、次の手順に従います。**

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 10 Demo;UID=DBA;PWD=sql;ConnectionName=Connection1"
   ```

 Copyright © 2007, iAnywhere Solutions, Inc.

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 10 Demo;UID=DBA;PWD=sql;ConnectionName=Connection2"
```

3. Connection1 で、次のコマンドを実行し、独立性レベルを 1 (コミットされた読み込み) に設定します。 レベル 1 では、現在のローで読み込みロックを取得して設定します。

```
SET OPTION isolation_level = 1;
```

4. Connection1 で、次のコマンドを実行します。

```
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	Tee Shirt	Crew Neck	One size fits all	Black	75	...
400	Baseball Cap	Cotton Cap	One size fits all	Black	112	...
...	...	...	...	...	...	...

5. Connection2 で、次のコマンドを実行します。

```
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 文がコミットされていないかロールバックされていないため、この SELECT 文はブロックされて処理できません。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、次のコマンドを実行します。

```
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。

8. 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。

Connection1 で、次のコマンド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、次のコマンドを実行し、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、次のコマンド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;
```

スナップショット・アイソレーションの使用方法のその他の例については、次の項を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

- 「スナップショット・アイソレーションを使用したダーティ・リードの回避」 149 ページ
- 「スナップショット・アイソレーションを使用した繰り返し不可能読み出しの回避」 155 ページ
- 「スナップショット・アイソレーションを使用した幻ローの回避」 159 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、更新または削除が試みられるとすぐにエラーになります。

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 3 つのケースは、ISO SQL/2003 標準に記載されており、重要なものです。独立性レベルの低い動作を定義する際の基準となるものだからです。

- **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。

  ダーティ・リードの詳細については、「チュートリアル：ダーティ・リード」 147 ページを参照してください。

- **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。

  繰り返し不能読み出しの詳細については、「チュートリアル：繰り返し不可能読み出し」 151 ページを参照してください。

- **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。

  幻ローの詳細については、「チュートリアル：幻ロー」 157 ページを参照してください。

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込み	X	X	X
readonly-statement-snapshot	X[1]	X[2]	X[3]
1–コミットされた読み込み		X	X
statement-snapshot		X[2]	X[3]
2–繰り返し可能読み出し			X
3–直列可能			
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。

Copyright © 2007, iAnywhere Solutions, Inc.

- 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。「ValuePtr パラメータ」 137 ページを参照してください。

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。

SQL プロシージャとカーソルのプログラミングについては、「プロシージャ、トリガ、バッチの使用」 777 ページを参照してください。

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。詳細については、「アプリケーションでの SQL の使用」 『SQL Anywhere サーバ - プログラミング』を参照してください。

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。

カーソルの sensitivity の詳細については、「SQL Anywhere のカーソル」 『SQL Anywhere サーバ - プログラミング』を参照してください。

独立性レベルの設定

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グループごとにデフォルトの独立性レベルを保存できます。isolation_level database データベース・オプションの PUBLIC 設定によって、単一のデフォルトの独立性レベルをデータベース・グループ全体に設定できます。

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。

接続の独立性や、ユーザ ID に設定されたデフォルトのレベルは、SET OPTION コマンドを使用して変更できます。パーミッションを持っている場合は、他のユーザやグループの独立性レベルも変更できます。

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。

スナップショット・アイソレーションのレベルの有効化と設定の詳細については、「スナップショット・アイソレーションの有効化」 129 ページを参照してください。

♦ 現在のユーザに独立性レベルを設定するには、次の手順に従います。

- 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;
```

♦ ユーザまたはグループに独立性レベルを設定するには、次の手順に従います。

1. DBA 権限のあるユーザとしてデータベースに接続します。
2. isolation_level の前にグループ名とピリオドを付加し、SET OPTION 文を実行します。たとえば次のコマンドは、グループ PUBLIC 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;
```

♦ 現在の接続の独立性レベルを設定するには、次の手順に従います。

- 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;
```

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、ユーザやグループ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。すべてのユーザは特殊グループ PUBLIC のメンバです。最初に他の設定が見つからない場合、SQL Anywhere はそのグループに割り当てられているレベルを使用します。

ユーザとグループの詳細については、「ユーザ ID とパーミッションの管理」『SQL Anywhere サーバ - データベース管理』を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

SET OPTION 文の構文の詳細については、「SET OPTION 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

たとえば、1 つ以上のテーブルがシリアル・アクセスを必要とする場合、トランザクション中に独立性レベルを変更できます。トランザクション内の独立性レベルの変更については、「トランザクション内の独立性レベルの変更」138 ページを参照してください。

ODBC 実行可能アプリケーションからの独立性レベルの設定

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 SQLSetConnectOption 関数を使用します。

SQLSetConnectOption 関数は、次の 3 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に対応する値です。次のテーブルは、これらの値をまとめたものです。

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

Copyright © 2007, iAnywhere Solutions, Inc.

文字列	値
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

たとえば次の関数呼び出しは、接続 MyConnection の独立性レベルを 2 に設定します。

```
SQLSetConnectOption( MyConnection.hDbc,
            SQL_TXN_ISOLATION,
            SQL_TXN_REPEATABLE_READ )
```

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

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」 157 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 NON_SNAPSHOT_TRANSACTION を返します。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

現在の接続の独立性レベルは、CONNECTION_PROPERTY 関数を使用して調べることができます。

♦ 現在の接続の独立性レベルを参照するには、次の手順に従います。

- 次の文を実行します。

  ```
  SELECT CONNECTION_PROPERTY('isolation_level')
  ```

Copyright © 2007, iAnywhere Solutions, Inc.

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。

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。詳細については、「デッドロック」 141 ページを参照してください。

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。

ブロック・オプションの詳細については、「blocking オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

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。
- **アクティブなデータベース・スレッドがすべてブロックされている**　トランザクションがブロックされても、データベース・スレッドは放棄されたわけではありません。サーバが 3 つのスレッドで設定されていて、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 にブロックされると、これ以上使えるスレッドがなくなるためデッドロックとなります。

SQL Anywhere は、最後にブロックされた文を自動的にロールバックして、デッドロックを解放します。ロールバックされたトランザクションにはデッドロックの種類を示すエラーを返します。

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。

データベース・スレッド数の設定の詳細については、「スレッド動作の制御」『SQL Anywhere サーバ - データベース管理』を参照してください。

ブロックされているユーザの判別

sa_conn_info システム・プロシージャを使用して、どの接続がどの接続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。

詳細については、「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』を参照してください。

データベース・サーバは、log_deadlocks オプションと sa_report_deadlocks システム・プロシージャを使用して詳細なデッドロック・レポートを提供します。log_deadlocks オプションをオンにした場合、データベース・サーバは、ブロックされた接続に関する情報を内部バッファに記録します。

♦ **デッドロック・レポートを使用するには、次の手順に従います。**

1. log_deadlocks オプションをオンにします。

```
SET OPTION PUBLIC.log_deadlocks='On'
```

   データベース・サーバは、デッドロックに関する情報を内部バッファに記録します。

詳細については、「log_deadlocks オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

2. sa_report_deadlocks を使用してデッドロック情報を検索します。

```
CALL sa_report_deadlocks()
```

詳細については、「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』を参照してください。

Sybase Central からデッドロックの表示

Sybase Central でデータベースに接続している場合は、log_deadlocks オプションが On に設定されてからデータベースで発生したデッドロックの図を確認できます。

◆ **Sybase Central のデッドロック・レポートを使用するには、次の手順に従います。**

1. 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] を選択します。

   [データベースのオプション] ダイアログが表示されます。

2. log_deadlocks オプションをオンにします。

   データベース・サーバは、デッドロックに関する情報を内部バッファに記録します。

   a. [オプション] リストで log_deadlocks を選択します。

   b. 値のフィールドに On と入力します。

   c. [一時的な設定を行う] をクリックします。

   d. [閉じる] をクリックします。

   詳細については、「log_deadlocks オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

3. 右ウィンドウ枠で、[デッドロック] タブをクリックします。

   データベースにデッドロックがある場合は、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。

Copyright © 2007, iAnywhere Solutions, Inc.

独立性レベルの選択

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。一方、休止中の口座の残高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可能となる方法でインタリーブされます。

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。

スナップショット・アイソレーションを使用するうえでのパフォーマンス上の重要事項の詳細については、「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』を参照してください。

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 よりも多少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。

スナップショット・アイソレーションの詳細については、「スナップショット・アイソレーション」 127 ページを参照してください。

直列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「**直列可能**」であるといいます。

直列可能性は、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可能です。デフォルト設定値は 0 です。

直列可能とは、同時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。

直列不可能なスケジュールは矛盾を生じさせる

「典型的な矛盾のケース」 133 ページで説明した矛盾のケースは、スケジュールが直列可能でないときに生じる典型的な問題です。それぞれのケースでは、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はずの結果を生じるような方法で文がインタリーブされたため、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。

Copyright © 2007, iAnywhere Solutions, Inc.

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、特に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、ロック要件をあまり増大せずにカーソルの安定性を保てるからです。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、特に独立性レベル 1 が適しています。なぜなら、品目の入荷や販売時に在庫レベルに対して行う各調整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すむからです。

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。したがってこのレベルは、複数回ローを読み込み、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注意して設計すると、ロック数も減り、データベースのパフォーマンスを大幅に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、ローをロックする時間も短縮され、パフォーマンスが向上します。

直列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、検索基準を満たしているかどうかに関係なく、すべてのローの読み込みをロックするので、データベース・サーバはローの読み込みとロックの設定のオペレーションを自由に組み合わせることができます。

ロックの影響の削減

独立性レベル 3 でトランザクションを実行することは、できるかぎり避けてください。多数のロックを設定すると、他の同時に実行されるトランザクションの効率を妨げることになります。

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。

特に、インデックスを追加すると、トランザクションの処理効率が格段に向上することがわかります。これは、トランザクションのうちの少なくとも 1 つに独立性レベル 3 を設定して実行する場合に特に顕著です。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。

SQL Anywhere で使用されるロック方法の詳細については、「ロックの仕組み」 166 ページを参照してください。

パフォーマンスと、コマンドを実行するために SQL Anywhere が情報にアクセスする方法の詳細については、「パフォーマンスのモニタリングと改善」 199 ページを参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

独立性レベルのチュートリアル

独立性レベルの設定によって、処理は大きく異なります。また、どのレベルを設定するかは、使用するデータベースと実行する操作によって変わってきます。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。

チュートリアル：ダーティ・リード

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矛盾の 1 つを再現してみます。小規模の商品販売会社で、2 人の従業員が、会社のデータベースに同時にアクセスするとします。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提出するレポートに記載しようとしています。

ヒント
データベースを次の方法で変更するときには、UPDATE の代わりに SELECT を使用して変更内容をテストしてから変更した方が賢明です。

この例では、実際にこの 2 人が同時にサンプル・データベースを使用するケースを示します。

1. Interactive SQL を起動します。
2. Sales Manager として SQL Anywhere サンプル・データベースに接続します。
   - [接続] ダイアログで、[SQL Anywhere 10 Demo] ODBC データ・ソースを選択します。
   - ウィンドウを容易に識別できるようにするために、[詳細] タブをクリックし、[接続名] フィールドに **Sales Manager** と入力します。
   - [OK] をクリックして接続します。
3. Interactive SQL をもう 1 つ起動します。
4. Accountant として SQL Anywhere サンプル・データベースに接続します。
   - [接続] ダイアログで、[SQL Anywhere 10 Demo] ODBC データ・ソースを選択します。
   - ウィンドウを容易に識別できるようにするために、[詳細] タブをクリックし、[接続名] フィールドに **Accountant** と入力します。
   - [OK] をクリックして接続します。
5. 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。
   - [Sales Manager] ウィンドウで次のコマンドを実行します。

```
UPDATE Products
   SET UnitPrice = UnitPrice + 95
   WHERE Name = 'Tee Shirt';
SELECT ID, Name, UnitPrice
   FROM Products;
```

結果は次の表のようになります。

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、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。

6. Accountant は、在庫額が多いことを懸念しています。Accountant として次のコマンドを実行し、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 Products;
```

結果は次の表のようになります。

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。

ダーティ・リードやその他の矛盾の防止については、「独立性レベルと一貫性」 125 ページを参照してください。

7. Sales Manager として、最初の変更をロールバックし、正しい UPDATE コマンド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 Products
SET UnitPrice = UnitPrice + 0.95
WHERE NAME = 'Tee Shirt';
COMMIT;
```


Copyright © 2007, iAnywhere Solutions, Inc.

ID	Name	UnitPrice
300	Tee Shirt	9.95
301	Tee Shirt	14.95
302	Tee Shirt	14.95
400	Baseball Cap	9.00
…	…	…

8. 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 Products;
```

Inventory
6687.15

9. 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 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;
```

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。

スナップショット・アイソレーションを使用したダーティ・リードの回避

スナップショット・アイソレーションを使用すると、他のデータベース接続は、クエリの応答でコミットされたデータだけを認識します。独立性レベルを statement-snapshot または snapshot に設定すると、ダーティ・リードが発生する可能性がなくなります。Accountant は、スナップショット・アイソレーションを使用して、クエリの実行時にコミットされたデータだけが認識されるようにできます。

1. Interactive SQL を起動します。
2. Sales Manager として SQL Anywhere サンプル・データベースに接続します。
   - [接続] ダイアログで、[SQL Anywhere 10 Demo] ODBC データ・ソースを選択します。
   - ウィンドウを容易に識別できるようにするために、[詳細] タブをクリックし、[接続名] フィールドに **Sales Manager** と入力します。
   - [OK] をクリックして接続します。
3. 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';
```

4. Interactive SQL をもう 1 つ起動します。
5. Accountant として SQL Anywhere サンプル・データベースに接続します。
   - [接続] ダイアログで、[SQL Anywhere 10 Demo] ODBC データ・ソースを選択します。
   - ウィンドウを容易に識別できるようにするために、[詳細] タブをクリックし、[接続名] フィールドに **Accountant** と入力します。
   - [OK] をクリックして接続します。
6. 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。
   - [Sales Manager] ウィンドウで次のコマンドを実行します。

     ```
     UPDATE Products
     SET UnitPrice = UnitPrice + 0.95
     WHERE Name = 'Tee Shirt';
     ```

   - 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 Products;
     ```

   結果は次の表のようになります。

Inventory
6687.15

7. Accountant として次のコマンドを実行し、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 Products;
   ```

   結果は次の表のようになります。

Inventory
6538.00

8. 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;
   ```

9. 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 )
   ```


Copyright © 2007, iAnywhere Solutions, Inc.

```
AS Inventory
 FROM Products;
```

結果は次の表のようになります。

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。「スナップショット・トランザクションの知識」 129 ページを参照してください。

10. 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 Products
SET UnitPrice = UnitPrice - 0.95
WHERE Name = 'Tee Shirt';
COMMIT;
```

チュートリアル：繰り返し不可能読み出し

「チュートリアル：ダーティ・リード」 147 ページの例では、矛盾のケースとして、まずダーティ・リードを取り上げました。その中では、Sales Manager が価格を更新している間に、Accountant が計算を実行してしまいました。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。

次の例では、別のタイプの矛盾を説明します。これは、「繰り返し不可能読み出し」というものです。この例でも、同時に SQL Anywhere サンプル・データベースを使用する同じ 2 人のユーザを使います。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 に付属の 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 に設定して、前のチュートリアルで示した一貫性が失われるケース (ダーティ・リード) を防止します。

1. Interactive SQL を起動します。
2. Sales Manager として SQL Anywhere サンプル・データベースに接続します。
   - [接続] ダイアログで、[SQL Anywhere 10 Demo] ODBC データ・ソースを選択します。
   - ウィンドウを容易に識別できるようにするために、[詳細] タブをクリックし、[接続名] フィールドに **Sales Manager** と入力します。
   - [OK] をクリックして接続します。
3. Interactive SQL をもう 1 つ起動します。

Copyright © 2007, iAnywhere Solutions, Inc.

4. Accountant として SQL Anywhere サンプル・データベースに接続します。
   - [接続] ダイアログで、[SQL Anywhere 10 Demo] ODBC データ・ソースを選択します。
   - ウィンドウを容易に識別できるようにするために、[詳細] タブをクリックし、[接続名] フィールドに **Accountant** と入力します。
   - [OK] をクリックして接続します。
5. 次のコマンドを実行して、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
```

6. 次のコマンドを実行して、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
```

7. Accountant は、バイザーの価格をリストすることにします。Accountant として、次のコマンドを実行します。

```
SELECT ID, Name, UnitPrice FROM 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
...	...	...

8. Sales Manager は、プラスチック・バイザーに新価格を導入することにします。Sales Manager として、次のコマンドを実行します。

```
SELECT ID, Name, UnitPrice FROM Products
WHERE Name = 'Visor';
UPDATE Products
SET UnitPrice = 5.95 WHERE ID = 501;
COMMIT;
SELECT ID, Name, UnitPrice FROM Products
WHERE Name = 'Visor';
```

ID	Name	UnitPrice
500	Visor	7.00
501	Visor	5.95

Copyright © 2007, iAnywhere Solutions, Inc.

9. 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 が 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 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 文を実行しても、同じ結果が得られないためです。

もちろん、Accountant が SELECT コマンドを再度使用する前に、COMMIT や ROLLBACK コマンド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不可能となります。

10. 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 Products;
```

11. 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 のウィンドウで、次の文を実行します。コマンドの実行が始まりますが、ウィンドウがフリーズします。

```
UPDATE 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

Copyright © 2007, iAnywhere Solutions, Inc.

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。

12. 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すぐに、Sales Manager のトランザクションも完了します。

ID	Name	UnitPrice
500	Visor	7.00
501	Visor	7.00

13. Sales Manager も処理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;
```

ロックの種類と各種の独立性レベル

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。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著しく速くなります。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。独立性レベルを 0 ではなく 1 に設定し、最初のチュートリアルを繰り返してみます。Sales Manager が T シャツの価格を更新するトランザクションの最中は、Accountant の計算は処理を進めることができず、不完全なままになることがわかります。

Accountant が独立性レベルを 2 に上げたとき、データベース・サーバは読み込みロックの使用を開始しました。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。

Copyright © 2007, iAnywhere Solutions, Inc.

トランザクションのブロック

前述のチュートリアルでは、UPDATE コマンドの実行中に 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 コマンドの実行を開始し、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、Sales Manager の UPDATE が最後まで処理されます。

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**」されたといいます。

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・オプションが ON に設定されていると、前述のチュートリアルで説明したように 2 番目のトランザクションは待機します。

ブロック・オプションの詳細については、「ブロック・オプション」 140 ページ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た繰り返し不可能読み出しの回避

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。

1. Interactive SQL を起動します。
2. Sales Manager として SQL Anywhere サンプル・データベースに接続します。
   - [接続] ダイアログで、[SQL Anywhere 10 Demo] ODBC データ・ソースを選択します。
   - ウィンドウを容易に識別できるようにするために、[詳細] タブをクリックし、[接続名] フィールドに **Sales Manager** と入力します。
   - [OK] をクリックして接続します。
3. Interactive SQL をもう 1 つ起動します。
4. Accountant として SQL Anywhere サンプル・データベースに接続します。
   - [接続] ダイアログで、[SQL Anywhere 10 Demo] ODBC データ・ソースを選択します。
   - ウィンドウを容易に識別できるようにするために、[詳細] タブをクリックし、[接続名] フィールドに **Accountant** と入力します。
   - [OK] をクリックして接続します。

5. 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;
```

6. Accountant は、バイザーの価格をリストすることにします。Accountant として、次のコマンドを実行します。

```
SELECT *
FROM Products
ORDER BY ID;
```

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
...	...	...

7. Sales Manager は、プラスチック・バイザーに新価格を導入することにします。Sales Manager として、次のコマンドを実行します。

```
UPDATE Products
SET UnitPrice = 5.95 WHERE ID = 501;
COMMIT;
SELECT ID, Name, UnitPrice FROM Products
WHERE Name = 'Visor';
```

8. 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 Products
```

9. 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 Products
SET UnitPrice = 7.00
WHERE ID = 501;
COMMIT;
```

 Copyright © 2007, iAnywhere Solutions, Inc.

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。

10. 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;
```

チュートリアル：幻ロー

次のチュートリアルでも同じ例を使います。ここでは、Sales Manager が新しい Departments テーブルを作成している最中に、Accountant が部署テーブルを見ています。その場合、幻ローが表示されます。

まだ実行していない場合は、前のチュートリアル「チュートリアル：繰り返し不可能読み出し」 151 ページの手順 1 から 4 までを行ってください。Interactive SQL が 2 つ起動されます。

1. 次のコマンドを実行して、Sales Manager のウィンドウに独立性レベル 2 を設定します。

```
SET TEMPORARY OPTION isolation_level = 2;
```

2. 次のコマンドを実行して、Accountant のウィンドウに独立性レベル 2 を設定します。

```
SET TEMPORARY OPTION isolation_level = 2;
```

3. Accountant のウィンドウに次のコマンドを入力し、すべての部署をリストします。

```
SELECT * FROM Departments
ORDER BY DepartmentID;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703

4. Sales Manager は外国市場に焦点を当てた新しい部署を設定することを決めます。EmployeeID 129 の Philip Chin を新しい部署の責任者にします。

```
INSERT INTO Departments
  (DepartmentID, DepartmentName, DepartmentHeadID)
  VALUES(600, 'Foreign Sales', 129);
```

```
COMMIT;
```

最後のコマンドは新しい部署に新しいエントリを作成します。このエントリは、Sales Manager のウィンドウのテーブルの一番下に新しいローとして表示されます。

Sales Manager のウィンドウに次のコマンドを入力し、すべての部署をリストします。

```
SELECT *
FROM 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 コマンドを再実行した場合にだけ、新しいローを発見できます。Accountant のウィンドウで SELECT 文を再実行してください。テーブルに新しいローが追加されているのがわかります。

```
SELECT *
FROM 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。他のローはロックがかからないため、Sales Manager が新しいローを挿入するのを防止することはできません。

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。次のコマンドを Accountant として入力します。

Copyright © 2007, iAnywhere Solutions, Inc.

```
SET TEMPORARY OPTION isolation_level = 3;
SELECT *
FROM Departments
ORDER BY DepartmentID;
```

7. Sales Manager は、大企業を対象にした営業活動を行う新たな部署を追加したいと思っています。Sales Manager のウィンドウで次のコマンドを実行します。

```
INSERT INTO Departments
 (DepartmentID, DepartmentName, DepartmentHeadID)
  VALUES(700, 'Major Account Sales', 902);
```

Accountant のロックがコマンド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。ツールバーの [SQL 文の中断] ボタン (または [SQL] メニューの [中断]) をクリックしてこのエントリを一時停止します。

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. Sales Manager のウィンドウで次のコマンドを実行し、最後の未完了トランザクションをロールバックします。

   ```
   ROLLBACK
   ```

   b. Sales Manager のウィンドウで次の 2 つの文を実行し、先に挿入したローを削除し、この操作をコミットします。

   ```
   DELETE FROM Departments
   WHERE DepartmentID = 600;

   COMMIT;
   ```

説明

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、そのコマンド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 のコマンド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。

ロックの詳細については、「ロックの仕組み」 166 ページ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た幻ローの回避

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 のコマンドはブロックされず、Accountant は幻ローを認識しません。

まだ実行していない場合は、「チュートリアル：幻ロー」 157 ページの手順 1 から 4 を実行してください。Interactive SQL が 2 つ起動されます。

Copyright © 2007, iAnywhere Solutions, Inc.

1. 次のコマンドを実行し、Accountant 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';
SET TEMPORARY OPTION isolation_level = snapshot;
```

2. Accountant のウィンドウに次のコマンドを入力し、すべての部署をリストします。

```
SELECT * FROM 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 は外国市場に焦点を当てた新しい部署を設定することを決めます。EmployeeID 129 の Philip Chin を新しい部署の責任者にします。

```
INSERT INTO Departments
  (DepartmentID, DepartmentName, DepartmentHeadID)
  VALUES(600, 'Foreign Sales', 129);
COMMIT;
```

最後のコマンドは新しい部署に新しいエントリを作成します。このエントリは、Sales Manager のウィンドウのテーブルの一番下に新しいローとして表示されます。

```
SELECT * FROM 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

4. 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終了していないため、新しいローを認識しません。

Copyright © 2007, iAnywhere Solutions, Inc.

```
SELECT *
FROM Departments
ORDER BY DepartmentID;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703

5. Sales Manager は、大企業を対象にした営業活動を行う新たな部署を追加したいと思っています。Sales Manager のウィンドウで次のコマンドを実行します。

   ```
   INSERT INTO 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。

6. 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 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

7. 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。

Copyright © 2007, iAnywhere Solutions, Inc.

a. Sales Manager のウィンドウで次のコマンドを実行し、最後の未完了トランザクションをロールバックします。

```
ROLLBACK
```

b. Sales Manager のウィンドウで次の 2 つの文を実行し、先に挿入したローを削除し、この操作をコミットします。

```
DELETE FROM Departments
WHERE DepartmentID = 600;

COMMIT;
```

チュートリアル：実際のロックの意味

前のチュートリアルと同じ例を使います。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があります。Accountant は、2001 年の 4 月に売り上げを上げた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、データベースに追加されなかっ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。

彼らの操作で幻ロックについて説明します。「**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特定の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。

まだ実行していない場合は、「チュートリアル：幻ロー」 157 ページの手順 1 から 4 を実行してください。Interactive SQL が 2 つ起動されます。

1. 次のコマンドを実行して、Sales Manager のウィンドウと Accountant のウィンドウに独立性レベル 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。

   Accountant のウィンドウに次のコマンドを入力します。価格、受注情報、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。

```
SELECT EmployeeID, GivenName, Surname,
  SUM(SalesOrderItems.Quantity * UnitPrice)
    AS "April sales"
FROM Employees
  KEY JOIN SalesOrders
  KEY JOIN SalesOrderItems
  KEY JOIN Products
WHERE '2001-04-01' <= OrderDate
  AND OrderDate < '2001-05-01'
GROUP BY  EmployeeID, GivenName, Surname;
```


Copyright © 2007, iAnywhere Solutions, Inc.

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 のウィンドウに次のコマンドを入力します。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 SalesOrders
   VALUES ( 2653, 174, '2001-04-22', 'r1',
       'Central', 129);
   INSERT into 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。同じコマンドを使用しますが、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
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 のウィンドウで、次の文を実行して新しい注文を削除します。

```
DELETE
FROM SalesOrderItems
WHERE ID = 2653;
DELETE
FROM SalesOrders
WHERE ID = 2653;
COMMIT;
```

6. Accountant のウィンドウで、次の 2 つの文を実行します。

```
ROLLBACK;
SET TEMPORARY OPTION isolation_level = 3;
```

7. Accountant のウィンドウで、前と同じクエリを実行します。

```
SELECT EmployeeID, GivenName, Surname,
  SUM(SalesOrderItems.Quantity * UnitPrice)
    AS "April sales"
FROM Employees
  KEY JOIN SalesOrders
  KEY JOIN SalesOrderItems
  KEY JOIN 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 のウィンドウに戻ります。再度 Philip Chin の漏れていた注文を入力します。

```
INSERT INTO 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。ツールバーの [SQL 文の中断] ボタン (または [SQL] メニューの [中断]) をクリックしてこのエントリを一時停止します。

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。

   コマンドの日付を 5 月 5 日に変更し、再実行します。

```
INSERT INTO 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。ツールバーの [SQL 文の中断] ボタン (または [SQL] − [中断]) をクリックしてこのエントリを一時停止します。データベース・サーバは項目の挿入を防止するために必要な箇所にしかロックを設定しませんが、これらのロックは他の多数のトランザクションを妨げ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。

 Copyright © 2007, iAnywhere Solutions, Inc.

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. Sales Manager は、注文 2651 に 2 番目の品目を追加したいと考えています。次のコマンドを使用します。

    ```
    INSERT INTO SalesOrderItems
    VALUES ( 2651, 2, 302, 4, '2001-05-22' );
    ```

    Sales Manager のウィンドウは応答を停止します。ツールバーの [SQL 文の中断] ボタン (または [SQL] メニューの [中断]) をクリックしてこのエントリを一時停止します。

11. 変更を取り消して SQL Anywhere サンプル・データベースの変更を防止し、このチュートリアルを終了します。Sales Manager のウィンドウに次のコマンドを入力します。

    ```
    ROLLBACK;
    ```

    Accountant のウィンドウに同じコマンドを入力します。

    ```
    ROLLBACK;
    ```

ここで、両方のウィンドウを閉じます。

ロックの仕組み

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、テーブルのローを 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。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 つだけ例外がありますが、「読み込みロックの早期解放」 180 ページの項で説明します。

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。

ロックできるオブジェクト

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、トランザクションはテーブルのスキーマをロックして、他のトランザクションがそのテーブルの構造を変更しないようにできます。
- **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。
- **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、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。
- **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、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。

Copyright © 2007, iAnywhere Solutions, Inc.

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

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、または Sybase Central の [テーブル・ロック] 領域を使用します。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ません。

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。左ウィンドウ枠でデータベースを選択すると、右ウィンドウ枠に [テーブル・ロック] タブが表示されます。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ユニークに識別できる 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。

Copyright © 2007, iAnywhere Solutions, Inc.

参照

sa_locks システム・プロシージャの詳細については、「sa_locks 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。

NOLOCK テーブル・ヒントの詳細については、「FROM 句」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ROWID 関数の詳細については、「ROWID 関数 [その他]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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するため、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するために、使用されます。たとえばスキーマ・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、2 つのクラスがあります。

- **共有ロック**　テーブルは共有 (読み込み) モードでロックされる。
- **排他ロック**　テーブルは、単一接続の排他的使用のためにロックされる。

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。

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。つまり、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。

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ぐために使用されます。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。

Copyright © 2007, iAnywhere Solutions, Inc.

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きに、読み込みロックを取得することがあります。読み込みロックが取得されるかどうかは、トランザクションの独立性レベルに依存します。ただし、一度読み込みロックのかかったローに対しては、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、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込みのセマンティック、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込み) 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。「スナップショット・アイソレーション」 127 ページを参照してください。

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。これは、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からの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)。

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実行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。したがって、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、スナップショットの更新競合エラーで失敗します。

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 3 種類あります。

- 共有
- 意図的 (書き込むため)
- 排他

テーブル・ロックは、トランザクションが COMMIT または ROLLBACK で終了したときにのみ解放されます。

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

Copyright © 2007, iAnywhere Solutions, Inc.

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・ロックは、LOCK TABLE ... IN SHARED MODE 文を使用することで、明示的に取得できます。トランザクションが共有テーブル・ロックを取得すると、そのトランザクションが完了するまでテーブルに変更が加えられないようにできます。複数のトランザクションが同じテーブルで共有テーブル・ロックを保持できます。

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。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。

排他ロック

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。ただし、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。

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を実装しています。幻とは、データベースが動的であるために発生する、操作に関連して表示または非表示になるローのことです。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だけです。

独立性レベル 3 で実行されるトランザクションは、直列可能であると言われます。つまり、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、計算の結果に影響受ける可能性のあるローが導入される 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を元にスキャン位置が決まります。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・キー値を元にスキャン位置が決まります (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、ローを順序づける値を特定の範囲に関する、他のトランザクションによる挿入を回避できます。ロックに関するこの説明で、挿入とは INSERT 文だけを意味するのではなく、インデックス付き属性の値を変更するような UPDATE 文も含まれます。この場合の UPDATE は、INSERT の直後にインデックス・エントリの DELETE を実行することと見なすことができます。

SQL Anywhere でサポートされる位置ロックには、幻ロックと対幻ロックの 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。これは、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あるためです。

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。したがって、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。

 Copyright © 2007, iAnywhere Solutions, Inc.

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非常に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。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。

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。したがって、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。したがって、ローの読み込みロックは多くの接続による共有ロックでも構いません。

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。

	読み込み (ロー)	意図的 (ロー)	書き込み (ロー)	共有 (テーブル)	意図的 (テーブル)	排他 (テーブル)	幻 (位置)	挿入 (位置)
読み込み (ロー)			競合			競合		
意図的 (ロー)		競合	競合			競合		
書き込み (ロー)	競合	競合	競合	競合		競合		

	読み込み (ロー)	意図的 (ロー)	書き込み (ロー)	共有 (テーブル)	意図的 (テーブル)	排他 (テーブル)	幻 (位置)	挿入 (位置)
共有 (テーブル)			競合		競合	競合		
意図的 (テーブル)				競合		競合		
排他 (テーブル)	競合	競合	競合	競合	競合	競合	競合	競合
幻 (位置)						競合		競合
挿入 (位置)						競合	競合	

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 で実行するときとほとんど同じ数のロックしか使用しません。実際、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、トランザクションは、他のトランザクションが書き込みロックを設定しているかどうかに関係なく、自由にどのローでも読み込みます。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がブロックしなくなると、そ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。「FROM 句」『SQL Anywhere サーバ - SQL リファレンス』にある READPAST ヒントに関する説明を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

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すぐにこれらのロックをすべて解放します。

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。

SELECT 文の述部を満たすかどうかに関係なくすべてのローに読み込みロックをかけることは、次の 2 つの重要な事柄を含んでいます。

- データベース・サーバは、独立性レベル 2 よりも多くのロックをかける必要がある。取得される幻ローの数は、スキャンで取得される読み込みロックの数よりも 1 多くなります。倍増したロックのオーバーヘッドは、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。

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、サーバは、削除された (しかしコミットしない) 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

 Copyright © 2007, iAnywhere Solutions, Inc.

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. 該当す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。

手順 9 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。このため、ユーザは、プライマリ・キーに対するインデックスを作成するようサーバに要求してはなりません。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 UNIQUE 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、データベースにオーファン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、次のコマンドを実行します。

```
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. 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. 該当するローに書き込みロックをかけます。
4. UPDATE 文により、該当するそれぞれのカラム値を更新します。
5. インデックスが付いた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。

Copyright © 2007, iAnywhere Solutions, Inc.

6. 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、「挿入時のロック」 176 ページの説明に従って、新しい外部キー値を挿入します。同様に、必要に応じて wait_for_commit の手順に従います。
7. テーブルが参照整合性関係でプライマリ・テーブルであり、かつ関係の UPDATE アクションが RESTRICT でない場合は、テーブルで共有スキーマ・ロック、各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して外部キーで該当するローを判断し、次に該当す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。

手順 7 の後で、AFTER UPDATE トリガが起動する場合があります。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。

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。
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でプライマリ・テーブルであり、かつ関係の DELETE アクションが RESTRICT でない場合は、テーブルで共有スキーマ・ロック、各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して外部キーで該当するローを判断し、次に該当す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。

Copyright © 2007, iAnywhere Solutions, Inc.

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。

読み込みロックの早期解放

独立性レベル 3 では、読み込むすべてのローに読み込みロックをかけます。通常、処理が終了するまでロックは解放されません。実際、スケジュールを直列可能とするためには、トランザクションがロックを早期に解放しないことが重要です。

SQL Anywhere は、トランザクションが完了するまで常に書き込みロックを保持します。早期にロックを解放すると、別のトランザクションがそのローを修正でき、最初のトランザクションをロールバックできなくなる可能性があります。

読み込みロックは、ある特別な状況でのみ解放されます。独立性レベル 1 では、カーソルの現在のローにだけ読み込みロックをかけます。ただし独立性レベル 1 では、そのローが現在のローでなくなるとロックは解放されます。データベース・サーバは独立性レベル 1 で繰り返し読み出しを保証する必要がないため、この動作は問題ありません。

独立性レベルの詳細については、「独立性レベルの選択」 143 ページを参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

特殊な同時性の問題

この項では次の特殊な同時性の問題について説明します。

- 「プライマリ・キーの生成」 181 ページ
- 「データ定義文と同時性」 182 ページ

プライマリ・キーの生成

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。

番号を生成するにはいくつかの方法があります。

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、テーブル中の数字は 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。この方法では、各ユーザに十分な数字が確保されているかを定期的にチェック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、最後に使った送り状番号を入れるローが 1 つあります。新しく送り状を作成するには、データベースに接続し、テーブルの数字を 1 増やし、コミットします。1 増加した数字は新しい送り状に使います。他のユーザも送り状番号を取得できます。瞬時に終わる別のトランザクションで、送り状番号のローが更新されたからです。

- 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 値を使用してテーブル内のローをユニークに識別できます。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。

  ユニークな識別子の生成の詳細については、「NEWID デフォルト」 104 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

- 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、オートインクリメント・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。オートインクリメント・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。

データ定義文と同時性

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・サーバはコマンド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 文は同時性の問題は起こしません。

GRANT 文、REVOKE 文、SET OPTION 文も同時性の問題を起こしません。これらのコマンドは、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『Mobile Link - サーバ管理』と SQL Remote『SQL Remote』を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

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、多数のコマンド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。それぞれのトランザクションは、加えた変更をロールバックしたり、これらの変更をコミットして確定すると終了できます。

システム障害が発生したときに適切にデータ・リカバリを行うためには、トランザクションが不可欠です。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、同じトランザクションを順次実行した場合にはない、新しい矛盾した結果を生じる可能性があります。

多くの矛盾が考えられますが、次に示す 4 つのケースは ISO SQL/2003 標準にも記載され、独立性レベルもこれを基に定義されているため、特に重要です。

- ♦ **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可能であるといいます。スケジュールが直列可能である場合、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・レベルが使用できます。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。

Copyright © 2007, iAnywhere Solutions, Inc.

第 5 章

チュートリアル：SQL Anywhere データベースの作成

目次

SQL Anywhere データベース作成のチュートリアルの概要 186
レッスン 1：データベース・ファイルの作成 187
レッスン 2：データベースへの接続 188
レッスン 3：テーブルの作成 190
レッスン 4：カラムのプロパティ設定 192
レッスン 5：外部キーを使用したテーブル間の関係作成 194
まとめ 195

SQL Anywhere データベース作成のチュートリアルの概要

このチュートリアルでは、SQL Anywhere サンプル・データベースの Products、SalesOrderItems、SalesOrders、Customers の各テーブルを使用してモデル化した単純なデータベースを Sybase Central で作成する方法について説明します。

SQL Anywhere サンプル・データベースについては、「SQL Anywhere サンプル・データベース」『SQL Anywhere 10 - 紹介』を参照してください。

データベース設計の原則については、「データベースの設計」 3 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

レッスン 1：データベース・ファイルの作成

このレッスンでは、データベースを格納するデータベース・ファイルを作成します。データベース・ファイルには、システム・テーブルと、すべてのデータベースに共通するその他のシステム・オブジェクトが入ります。後のレッスンで、テーブルを追加します。

♦ 新しいデータベース・ファイルを作成するには、次の手順に従います。

1. Sybase Central を起動します。
2. [ツール] メニューから、[SQL Anywhere 10] – [データベースの作成] を選択します。

   [データベース作成] ウィザードが表示されます。
3. 概要ページに表示される情報を読んで、[次へ] をクリックします。
4. [このコンピュータにデータベースを作成] を選択し、[次へ] をクリックします。
5. データベース・ファイルを格納する場所と名前を選択します。

   ファイル名 *c:¥temp¥mysample.db* を入力します。テンポラリ・ディレクトリとして *c:¥temp* 以外のディレクトリを使用する場合は、正しいパスを指定します。
6. [完了] をクリックするとデータベースを作成します。

   その他のオプションは、データベースの作成時に利用可能ですが、通常はデフォルトの選択内容で十分です。

   [データベースの作成中] ウィンドウに、タスクの進行状況が表示されます。[完了] ステータスは、データベース・ファイルが作成されたことを示します。
7. ステータスが [完了] になったら、[閉じる] をクリックします。

レッスン 2：データベースへの接続

このレッスンでは、Sybase Central を使用して、作成したデータベース・ファイルに接続します。ただし、データベースの作成を完了したばかりの場合は、すでに接続されているため、テーブルの作成方法を学ぶ次のレッスンにスキップできます。「レッスン 3：テーブルの作成」 190 ページを参照してください。

♦ **データベースに接続するには、次の手順に従います。**

1. Sybase Central を起動します。
2. [接続] メニューから [SQL Anywhere 10 に接続] を選択します。

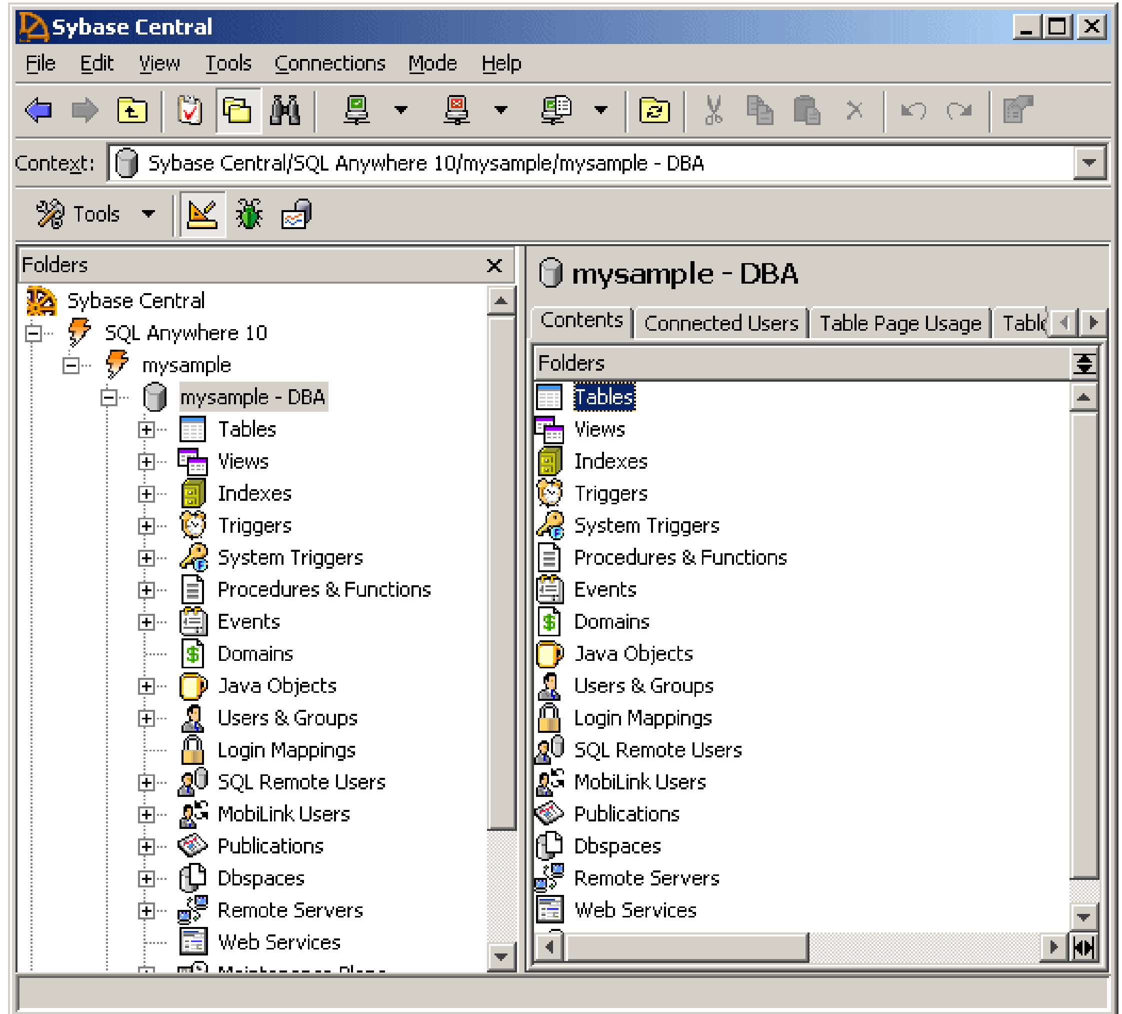
   [接続] ダイアログが表示されます。
3. ユーザ ID とパスワードを指定します。

   [ID] タブで、ユーザ ID を「**DBA**」、パスワードを「**sql**」と入力します。これらは、新しいデータベースのデフォルト値です。
4. タブの一番下にある [プロファイル] オプションで [**なし**] を選択します。
5. [データベース] タブで、[データベース・ファイル] フィールドに使用するデータベース・ファイルのフル・パスを入力します。たとえば、前のレッスンで指示のとおりにデータベース・ファイルを作成した場合は、次のように入力します。

   ```
   c:¥temp¥mysample.db
   ```
6. [OK] をクリックします。

   データベースが起動し、データベースと、そのデータベースが実行されているデータベース・サーバに関する情報が Sybase Central に表示されます。

Copyright © 2007, iAnywhere Solutions, Inc.


Sybase Central
File Edit View Tools Connections Mode Help
Context: Sybase Central/SQL Anywhere 10/mysample/mysample - DBA
Tools
Folders
Sybase Central
SQL Anywhere 10
mysample
mysample - DBA
Tables
Views
Indexes
Triggers
System Triggers
Procedures & Functions
Events
Domains
Java Objects
Users & Groups
Login Mappings
SQL Remote Users
MobiLink Users
Publications
Dbspaces
Remote Servers
Web Services
mysample - DBA
Contents
Connected Users
Table Page Usage
Folders
Tables
Views
Indexes
Triggers
System Triggers
Procedures & Functions
Events
Domains
Java Objects
Users & Groups
Login Mappings
SQL Remote Users
MobiLink Users
Publications
Dbspaces
Remote Servers
Web Services

レッスン 3：テーブルの作成

このレッスンでは、Products というテーブルを作成します。

テーブルの設計については、「テーブル・プロパティの設計」 24 ページを参照してください。

♦ **テーブルを作成するには、次の手順に従います。**

1. Sybase Central の右ウィンドウ枠で、[テーブル] フォルダをダブルクリックします。
2. [ファイル] メニューから、[新規] – [テーブル] の順に選択します。

   [テーブル作成] ウィザードが表示されます。
3. テーブルに **Products** という名前を付けます。
4. [完了] をクリックします。

   データベース・サーバによって、デフォルト設定を使用してテーブルが作成され、右ウィンドウ枠に [カラム] タブが表示されます。[カラム] タブの各ローでは、テーブルのカラムを定義します。
5. テーブルの最初のカラムを定義し、そのカラムをプライマリ・キーに設定します。

   プライマリ・キーは、連結された複数のカラムで構成することもできます。ただし、このチュートリアルでは、プライマリ・キーにはカラムが 1 つだけ含まれ、そのカラムの値は、テーブルに追加されるローごとに 1 ずつ自動的に増加します。オートインクリメント・プロパティにより、値はユニークであることが保証されます。これは、プライマリ・キーの要件です。「プライマリ・キー」 『SQL Anywhere 10 - 紹介』を参照してください。

   a. **プライマリ・キー**　[プライマリ・キー] カラムにチェックマークを付けて、カラムがテーブルのプライマリ・キーの一部であることを示します。

   b. **名前**　カラムに **ProductID** 名を指定します。

   c. **データ型**　次のようにデータ型を指定します。

   ♦ カラムを **integer** データ型と指定します。

     [データ型] カラムの右半分に省略記号が表示されます。

   ♦ 省略記号をクリックすると、カラムのプロパティ・シートが表示されます。

   ♦ [値] タブで、[デフォルト値] を選択します。

   ♦ [システム定義] を選択し、ドロップダウン・リストから [オートインクリメント] を選択します。

   d. [OK] をクリックして、[カラム] プロパティ・シートを閉じます。
6. 追加のカラムを作成します。

   [ファイル] メニューから [新規] – [カラム] を選択し、次のプロパティを持つカラムを追加します。

Copyright © 2007, iAnywhere Solutions, Inc.

- **名前**　カラムに **ProductName** 名を指定します。
- **データ型**　カラムを **char** データ型と指定します。
- **サイズ**　[サイズ] カラムに最大長を「**15**」と入力します。

7. [ファイル] – [保存] を選択します。

レッスン 4：カラムのプロパティ設定

このレッスンでは、カラムに NOT NULL 制約を追加する方法について学習します。

NULL と NOT NULL

Products テーブルの各製品には、関連付けられた製品 ID が必要です。そのため、Products テーブルの各ローでは、ID のカラムに値が含まれる必要があります。

各ローの ID のカラムに必ず値が含まれるようにするには、カラムを NOT NULL と定義します。

デフォルトではカラムに NULL 値が許可されますが、NULL 値を許可する明確な理由がないかぎり、カラムには NOT NULL を明示的に宣言してください。「NULL 値」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

♦ カラムを NOT NULL と定義するには、次の手順に従います。

1. [テーブル] フォルダを開きます。
2. 左ウィンドウ枠で Products テーブルを選択し、右ウィンドウ枠で [カラム] タブを選択します。
3. ProductID カラムを選択します。
4. [ファイル] メニューから [プロパティ] を選択して、[カラム] プロパティ・シートを開きます。
5. [制約] タブで、[NULL 値を禁止] を選択します。
6. [OK] をクリックします。

このレッスンと前のレッスンでは、データベース・テーブルを作成する際に知っておく必要のある基本概念について紹介しました。データベースにさらにテーブルを追加することで、ここで学んだことを復習できます。これらのテーブルは、この後のレッスンでも使用します。

次のテーブルをデータベースに追加します。

- **Customers**　テーブル Customers を追加し、テーブル内に以下のカラムを作成します。
  - **ID**　各顧客の ID 番号です。このカラムは **integer** データ型であり、**プライマリ・キー**です。これをオートインクリメント・カラムにします。
  - **CompanyName**　会社名です。このカラムは **char** データ型であり、最大長は **35** 文字です。
- **SalesOrders**　テーブル SalesOrders を追加し、テーブル内に以下のカラムを作成します。
  - **ID**　各注文書の ID 番号です。このカラムは **integer** データ型であり、**プライマリ・キー**です。これをオートインクリメント・カラムにします。
  - **OrderDate**　注文日です。このカラムは **date** データ型です。
  - **CustomerID**　発注した顧客の ID 番号です。このカラムは **integer** データ型です。

 Copyright © 2007, iAnywhere Solutions, Inc.

- **SalesOrderItems**　テーブル SalesOrderItems を追加し、テーブル内に以下のカラムを作成します。
  - **ID**　項目を含む注文書の ID 番号です。このカラムは **integer** データ型であり、**プライマリ・キー** カラムとして設定します。
  - **LineID**　各注文書の ID 番号です。このカラムは **integer** データ型であり、**プライマリ・キー** カラムとして設定します。
  - **ProductID**　受注製品の ID 番号です。このカラムは **integer** データ型です。

これで、データベースに 4 つのテーブルが作成されました。ただし、まだテーブル間の関連付けはされていません。次のレッスンでは、テーブルを互いに関連付ける外部キーを定義します。

レッスン 5：外部キーを使用したテーブル間の関係作成

このレッスンでは、外部キーを使用して、テーブル間の関係を作成する方法について学習します。

関係の設計については、「エンティティと関係」 5 ページを参照してください。

♦ **外部キーを作成するには、次の手順に従います。**

1. Sybase Central で [テーブル] フォルダを開きます。
2. SalesOrders テーブルを開きます。
3. Sybase Central の右ウィンドウ枠で [制約] タブをクリックします。
4. [ファイル] メニューから、[新規] – [外部キー] を選択して、[外部キーの作成] ウィザードを開きます。
5. 外部キーが参照するテーブルとして **Customers** テーブルを選択し、新しい外部キーに「**CustomerID**」と名前を付け、[次へ] をクリックします。
6. 参照する外部キーで **[プライマリ・キー]** を選択します。

   [外部カラム] ドロップダウン・リストから **[ID]** を選択し、[完了] をクリックします。
7. 1 ー 5 の手順を繰り返して、次のキーを作成します。

   - SalesOrderItems の ID カラムから外部キー。SalesOrders の ID カラムを参照します。
   - SalesOrderItems の ProductID カラムから外部キー。Products の ProductID カラムを参照します。
   - SalesOrders の CustomerID カラムから外部キー。Customers の ID カラムを参照します。

これで、リレーショナル・データベースの作成に関する一般的な説明を終わります。

データベースの設計については、「エンティティと関係」 5 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

まとめ

この章では、Sybase Central を使用して新しいデータベースを作成するために、データベース設計の原則を適用しました。

パート II. データベース・パフォーマンスのモニタリングと改善

パート II では、パフォーマンスの分析、改善、モニタリングについて説明します。

第 6 章

パフォーマンスのモニタリングと改善

目次

パフォーマンスのモニタリングと改善の概要 200
アプリケーション・プロファイリング 202
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 216
その他の診断ツールと方法 235
データベースのパフォーマンスのモニタリング 243
パフォーマンス向上のためのヒント 257

パフォーマンスのモニタリングと改善の概要

データベースのパフォーマンスを向上するには、データベースの現在のパフォーマンスを評価し、変更を加える前にまずすべての可能なオプションを検討します。SQL Anywhere のパフォーマンス機能とパフォーマンス分析ツールを使用してデータベースのスキーマとアプリケーションの設計を再評価することで、パフォーマンスの問題を診断し、訂正できます。また、データベースのパフォーマンスを最適レベルに保てます。

根本的に、データベースのパフォーマンスは、その設計に大きく左右されます。パフォーマンスを改善する最も基本的な方法の 1 つは、優れたスキーマを設計することです。データベース・スキーマは、データベースの枠組みであり、テーブル、ビュー、トリガ、およびそれらの関係などの定義が含まれます。データベース・スキーマを再評価し、少しの変更で大きな改善がもたらされる領域に留意してください。データベース・スキーマの設計の詳細については、「データベースの設計」 3 ページを参照してください。

データベースの実運用を開始したら、SQL Anywhere の高度なツールを使用して、発生するパフォーマンスの問題を検出、診断できます。これらのツールのほとんどは「**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。こ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**」などさまざまな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 では、複数の方法でパフォーマンス・データを生成、分析できます。

- **[アプリケーション・プロファイリング] ウィザードを使用したアプリケーション・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・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」 202 ページを参照してください。
- **[データベース・トレーシング] ウィザードを使用した詳細なアプリケーション・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・モードからこのウィザードを使用すると、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。したがって、注意が必要なユーザやアクティビティに注目できます。「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」 216 ページを参照してください。
- **要求トレースの分析**　この機能を使用すると、特定のユーザまたは接続から実行された要求 (文) の診断データだけを収集できます。「要求トレース分析の実行」 232 ページを参照してください。
- **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・モードからアクセスするか、スタンドアロンのツールとしてアクセスできます。「インデックス・コンサルタント」 210 ページを参照してください。
- **プロシージャのプロファイル表示**　この機能を使用すると、プロシージャ、ユーザ定義の関数、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た簡単な実装も可能です。「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」 204 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

コマンドを使用してプロシージャ・プロファイリングのタスクを実行する方法については、「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」 237 ページを参照してください。

注意
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。これは、これらの機能が基本的に同じであるからです。診断トレーシングは、アプリケーションの詳細なプロファイリングであるだけです。

アプリケーション・プロファイリング

アプリケーション・プロファイリングを使用すると、トレーシング情報を分析して、アプリケーションでデータベースがどのように操作されているかを理解できます。トレーシング・セッション中に生成されたデータから、パフォーマンスの問題を特定し、解決することもできます。プロファイリング情報は、2 つの方法で生成し、利用できます。[アプリケーション・プロファイリング] ウィザードを使用して自動的に行う方法と、Sybase Central のアプリケーション・プロファイリング・モードのさまざまなツールや機能を使用する方法です。

- **自動的なアプリケーション・プロファイリング**　この方法では、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、トレーシング・セッションを作成、分析して一般的なパフォーマンスの問題を特定します。また、データベースのパフォーマンスを向上するための具体的な推奨内容が表示されます。このウィザードでは、プロファイル情報を取得するアクティビティのタイプを指定できます。また、推奨内容はトレーシング・セッションの終了時に自動的に表示されます。[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、必要に応じてインデックスが推奨されます。

  この方法は、プロファイリング対象の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の必要がない環境に適しています。

- **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**　この方法では、コマンド・ラインから手動で作成するか、Sybase Central の設計モードで [データベース・トレーシング] ウィザードを使用して作成したトレーシング・セッション中に生成されたデータを確認します。トレーシング・データの格納場所を指定できます。また、プロファイリング対象のアクティビティも指定できるので、特定の問題に注目できます。たとえば、データベース・サーバで実行される特定の文、使用されているクエリ・プラン、デッドロック、互いにブロックする接続、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、プロファイリング対象の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。

  「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」 216 ページを参照してください。

Sybase Central のアプリケーション・プロファイリング・モードでは、前述のいずれかの方法で収集されたデータを詳しく調べることができます。たとえば、[データベース・トレーシング] ウィザードで生成されたトレーシング・セッションを開いて文を確認し、その文をブロックしたすべての文のリストを確認できます。また、[アプリケーション・プロファイリング] ウィザードで収集されたトレーシング・セッションのデータを参照して最も高コストの文やプロシージャを特定し、アプリケーションのパフォーマンスに影響を及ぼしているその他の要因に関する理解を深めることができます。

 Copyright © 2007, iAnywhere Solutions, Inc.

[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] ウィザードは、アプリケーション・プロファイリングのために診断トレーシング・セッションを自動的に行う簡単な方法です。このウィザードでは、アプリケーションでデータベースがどのように操作されているかに関する有用なデータが収集されてから、収集されたデータやインデックスに関する推奨内容が表示されます。

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・プロファイリングと診断トレーシング用に作成されるデータベース・ファイルの詳細については、「トレーシング・セッションのデータ」 216 ページを参照してください。

[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 CE 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。「トレーシング・セッションの作成」 226 ページを参照してください。

♦ 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には、次の手順に従います。

1. DBA と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] を選択します。
   - [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、ウィザードの指示に従います。
   - ウィザードが表示されなかった場合は、[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] を選択してから、ウィザードの指示に従います。

   ウィザードによって、診断トレーシング情報を格納するローカル・データベースが作成され、ネットワーク・サーバが起動され、トレーシング・セッションが開始され、プロファイリング対象のアプリケーションを実行するように指示するプロンプトが表示されます。[完了] はクリックしないでください。[完了] をクリックするとプロファイリングが終了し、ウィザードが終了します。
3. プロファイル情報を取得するアプリケーションを実行します。一般的なデータベースの使用状況のプロファイル情報を取得する場合は、データが収集される間、待ちます。終了したら、[アプリケーション・プロファイリング] ウィザードに戻り、[完了] をクリックします。ウィザードが終了したら、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。

[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の解釈」 212 ページを参照してください。

トレーシング・セッション中に収集されるプロシージャ・プロファイリング情報の詳細については、「プロシージャ・プロファイリングの結果を解釈する方法」 208 ページを参照してください。

ヒント
アプリケーション・プロファイリング・モードに切り替えたときに [アプリケーション・プロファイリング] ウィザードが自動的に起動しないようにできます。このようにするには、ウィザードの最初の画面で [今後、アプリケーション・プロファイリング・モードへの切り替え後はこのウィザードを表示しない] を選択します。また、[今後はこのページを表示しない] を選択してウィザードの最初のページが表示されないようにすることもできます。これらの設定は、[ツール] – [SQL Anywhere 10] – [ユーザ設定] を選択し、[ユーティリティ] タブを選択し、適切なオプションを選択することで変更することもできます。

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

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内に格納され、Sybase Central のアプリケーション・プロファイリング・モードからアクセスするか (推奨方法)、関数やシステム・プロシージャを使用してアクセスできます。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。この項では、アプリケーション・プロファイリング・モードでプロシージャ・プロファイリングを実行する方法について説明します。

SQL コマン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こともできます。「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」 237 ページを参照してください。

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します。

♦ **プロシージャ・プロファイリングを有効にするには、次の手順に従います。**

1. DBA としてデータベースに接続します。

Copyright © 2007, iAnywhere Solutions, Inc.

2. [モード] – [アプリケーション・プロファイリング] を選択します。

   - [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、ウィザードの指示に従います。
   - ウィザードが表示されなかった場合は、[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] を選択してから、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 [プロファイリング・オプション] 画面で [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] を選択します。その他のオプションは選択しないでください。

4. [完了] を選択します。

   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。別のモードに切り替えようとすると、プロシージャ・プロファイリングを続行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] を選択すると、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。

参照

- 「プロシージャ・プロファイリングのリセット」 205 ページ
- 「プロシージャ・プロファイリングの無効化」 206 ページ
- 「プロシージャ・プロファイリングの結果の分析」 207 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、関数、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。

◆ **プロファイリングをリセットするには、次の手順に従います。**

1. アプリケーション・プロファイリング・モードに切り替えます。[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] をクリックして終了します。

2. データベースの [データベース] プロパティ・シートを開きます。

   - プロシージャ・プロファイリングが有効になっている場合は、Sybase Central の最下部の [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て選択し、[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。
   - 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は、Sybase Central の左側の [フォルダ] ウィンドウ枠でデータベースを右クリックし、ポップアップ・メニューで [プロパティ] を選択します。

   [データベース] プロパティ・シートが表示されます。

3. [プロファイリング設定] タブで、[すぐにリセット] をクリックします。

   データベース・サーバで既存のプロファイリング情報がクリアされます。

4. [OK] をクリックしてプロパティ・シートを閉じます。

参照

- 「プロシージャ・プロファイリングの有効化」 204 ページ
- 「プロシージャ・プロファイリングの無効化」 206 ページ
- 「プロシージャ・プロファイリングの結果の分析」 207 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、関数のプロファイリング情報の取得が終了したら、プロシージャ・プロファイリングを無効にできます。プロシージャ・プロファイリングを無効にするときは、それまでに収集されたプロファイリング情報を削除するかどうかも選択できます。分析作業が完了している場合はプロファイリング情報を削除できます。

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。

♦ プロファイリング情報を削除しないでプロファイリングを無効にするには、次の手順に従います。

1. アプリケーション・プロファイリング・モードに切り替えます。[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] をクリックして終了します。

2. [アプリケーション・プロファイリングの詳細] 領域 (下のウィンドウ枠) 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情報の収集の停止] をクリックします。

   データベース・サーバでプロファイリング情報の収集が停止します。

♦ プロファイリング情報を削除してプロシージャ・プロファイリングを無効にするには、次の手順に従います。

1. アプリケーション・プロファイリング・モードに切り替えます。[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] をクリックして終了します。

2. [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。

   [データベース] プロパティ・シートが表示されます。

3. [プロファイリング設定] タブで、[すぐにクリア] をクリックします。

   プロシージャ・プロファイリングが無効になり ([このデータベースで次のプロファイリング情報を取得し表示する] チェックボックスがオフになる)、プロファイリング情報がデータベースから削除されます。

Copyright © 2007, iAnywhere Solutions, Inc.

4. [OK] をクリックしてプロパティ・シートを閉じます。

参照

- 「プロシージャ・プロファイリングの有効化」 204 ページ
- 「プロシージャ・プロファイリングのリセット」 205 ページ
- 「プロシージャ・プロファイリングの結果の分析」 207 ページ

プロシージャ・プロファイリングの結果の分析

プロシージャ・プロファイリングは、名前は「プロシージャ」ですが、実際にはデータベース内のストアド・プロシージャ、ユーザ定義関数、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。

♦ Sybase Central でのプロシージャ・プロファイリング情報を表示するには、次の手順に従います。

1. ユーザ DBA としてデータベースに接続していない場合は接続し、プロシージャ・プロファイリングを有効にします。「プロシージャ・プロファイリングの有効化」 204 ページを参照してください。

2. 左ウィンドウ枠で、プロファイリング情報を表示するオブジェクトのタイプを選択します。オブジェクトのタイプには、[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] のいずれかを選択できます。

   データベース内にあるこのタイプのオブジェクトのリストが右ウィンドウ枠に表示されます。たとえば、[プロシージャとファンクション] を選択した場合は、データベース内のすべてのプロシージャとユーザ定義関数のリストが表示されます。

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。

   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。たとえば、[プロシージャとファンクション] を選択した場合は、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すべてのプロシージャとユーザ定義関数のリストが表示されます。

   イベントなど、必要なオブジェクトが見つからない場合は、まだ実行されていない可能性があります。また、実行されたが、ビューが再表示されていない可能性があります。詳細は定期的に再表示されますが、[F5] を押していつでも再表示できます。

   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。

4. [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、行ごとの実行の詳細など、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。

   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。

Copyright © 2007, iAnywhere Solutions, Inc.

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
Name	オブジェクトの名前。
Owner	オブジェクトの所有者。
Table または Table Name	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)。
Event	オブジェクト・タイプ (プロシージャなど)。
Type	システム・トリガのタイプ。Update または Delete のいずれかです。
# Execs.	各オブジェクトが呼び出された回数。
#msecs.	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
Execs	オブジェクト内のコード行が実行された回数。
Milliseconds	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)。
Line	オブジェクト内の行番号。
Source	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、DBA 権限でデータベースに接続し、プロファイリングを有効にします。

プロファイリング・ログ・ファイルのベースラインとしての使用

Sybase Central では、プロシージャ・プロファイリングの結果をファイルに保存してベースライン情報として使用できます。たとえば、プロシージャを段階的に変更して実行を高速化できるかどうかを試している場合は、変更後にプロシージャを実行し、その結果を、ファイルに保存した結果と比較できます。

Copyright © 2007, iAnywhere Solutions, Inc.

Sybase Central でプロシージャ・プロファイリングの結果をベースラインとして使用する方法を理解するには、プロシージャ・プロファイリングの結果を解釈する方法を理解している必要があります。「プロシージャ・プロファイリングの結果の分析」 207 ページを参照してください。

♦ 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には、次の手順に従います。

1. プロファイル情報を取得するデータベースのプロシージャ・プロファイリングを有効にします。

   a. DBA としてデータベースに接続します。

   b. [モード] – [アプリケーション・プロファイリング] を選択します。

      ♦ [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、ウィザードの指示に従います。

      ♦ ウィザードが表示されなかった場合は、[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] を選択してから、ウィザードの指示に従います。

   c. [アプリケーション・プロファイリング] ウィザードの [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] を選択します。その他のオプションは選択しないでください。

   d. [完了] を選択します。

      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。別のモードに切り替えようとすると、プロシージャ・プロファイリングを続行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] を選択すると、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。

2. [プロシージャとファンクション] フォルダでプロシージャを右クリックし、[Interactive SQL から実行] を選択します。

   データベース・サーバで Interactive SQL からプロシージャが実行されます。プロシージャ・プロファイリングが有効になっているので、プロシージャの実行の詳細が取得されます。

3. Interactive SQL を閉じます。

4. 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て、[プロパティ] を選択します。

      データベースの [データベース] プロパティ・シートが表示されます。

   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選択します。

   d. [適用] 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た時点から収集されたプロシージャ・プロファイリングの情報が、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。

5. 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. [データベース] プロパティ・シートの [プロファイル設定] 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] をクリックして [データベース] プロパティ・シートを閉じます。

6. プロシージャを変更します。

   a. 左ウィンドウ枠の [プロシージャとファンクション] フォルダでプロシージャを探して選択します。

   b. 右ウィンドウ枠の [SQL] タブで、プロシージャの SQL コードを変更します。

   c. [ファイル] – [保存] を選択します。

7. プロシージャを右クリックし、[Interactive SQL から実行] を選択します。

   データベース・サーバで Interactive SQL からプロシージャが実行されます。

8. Interactive SQL を閉じます。

9. Sybase Central の右ウィンドウ枠で [プロファイリング結果] タブをクリックして実行の詳細を表示します。

   Execs. +/- と ms. +/- の 2 つの新しいカラムがあります。これらのカラムには、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の実行回数と実行時間の比較を示します。

   通常は、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す ms. +/- カラムを見ます。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、記号は表示されず、文字の色が緑になります。たとえば、-3 は、プロシージャ内のコード行の実行時間が、ベースラインよりも 3 ミリ秒短かったことを示します。

インデックス・コンサルタント

適切に設定されたインデックスを選択すると、データベースのパフォーマンスに大きな違いが出ます。適切に設定されたインデックスを選択するタスクを手助けするために、SQL Anywhere にはインデックス・コンサルタントが含まれています。インデックス・コンサルタントでは、データベースに推奨されるインデックスが表示されます。

 Copyright © 2007, iAnywhere Solutions, Inc.

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。インデックス・コンサルタントをデータベースの分析に使用すると、トレーシング・セッションを使用してデータが収集され、推奨内容が決定されます。これらのインデックスを使用してクエリ実行コストを推定し、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します。インデックス・コンサルタントは、個々のカラム・インデックスだけでなく複数のカラム・インデックスも評価します。またクラスタード・インデックスまたは非クラスタード・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て、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・コンサルタントの使用方法については、「インデックス・コンサルタントの使用」 211 ページを参照してください。

> **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、このとき、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。バージョン 9 のデータベース・サーバに接続しているときに Sybase Central でインデックス・コンサルタントを見つけ、使用する方法については、バージョン 9 のマニュアルを参照してください。

インデックス・コンサルタントの使用

インデックス・コンサルタントは、適切なインデックスを選択する手助けをします。インデックス・コンサルタントでは、単一のクエリまたはデータベース全体に推奨されるインデックスが表示されます。

クエリに推奨されるインデックスの確認

単一のクエリに推奨されるインデックスを確認するには、Interactive SQL からインデックス・コンサルタントにアクセスします。

♦ 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を確認するには、次の手順に従います。

1. InteractiveSQL で、[SQL 文] ウィンドウ枠にクエリを入力します。
2. [ツール] − [インデックス・コンサルタント] を選択します。

   インデックス・コンサルタントによって分析が行われ、推奨内容が返されます。

データベースに推奨されるインデックスの確認

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] ウィザードを使用します。これは、インデックス・コンサルタントの推奨内容を確認する最も簡単な方法です。

♦ 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を確認するには、次の手順に従います。

1. インデックス・コンサルタントで分析するデータベースに DBA として接続します。
2. [モード] – [アプリケーション・プロファイリング] を選択します。
3. [アプリケーション・プロファイリング] ウィザードが自動的に起動しない場合は、[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] を選択します。
4. ウィザードの指示に従い、アプリケーションを一定期間実行してトレーシング情報を取得します。
5. データの取得を終了したら、[アプリケーション・プロファイリング] ウィザードで [完了] 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] – [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] を選択します。

   インデックス・コンサルタントが表示されます。
7. インデックス・コンサルタントの指示に従います。

   インデックス・コンサルタントによってトレーシング・データが分析され、推奨内容が返されます。インデックス・コンサルタント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の解釈」 212 ページを参照してください。

インデックス・コンサルタントの推奨内容の解釈

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。

- **クラスタード・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、クラスタード・インデックスと非クラスタード・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、クラスタード・インデックスを適切に選択すると、非クラスタード・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、クラスタード・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。

  クラスタード・インデックスの詳細については、「クラスタード・インデックスの使用」 88 ページを参照してください。

- **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。

Copyright © 2007, iAnywhere Solutions, Inc.

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。

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。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、推奨インデックスがもたらすと考えられる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。次に示すような情報が提供されます。

- **[クラスタード]**　各テーブルは、最大 1 つのクラスタード・インデックスを持つことができます。場合によっては、クラスタード・インデックスは、非クラスタード・インデックスと比べて大きな利益をもたらします。

  クラスタード・インデックスの詳細については、「クラスタード・インデックスの使用」 88 ページを参照してください。
- **[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。

  データベース・ページ・サイズの詳細については、「初期化ユーティリティ (dbinit)」『SQL Anywhere サーバ - データベース管理』を参照してください。
- **[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、推定される利益が大きいことを示します。

Copyright © 2007, iAnywhere Solutions, Inc.

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。

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の実装」 214 ページを参照してください。

- **[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に基づいて測定されます。

  コスト・モデルの詳細については、「オプティマイザの推定とカラム統計」 524 ページを参照してください。

- **[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。ここには、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、データベースにすでに存在し、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。

インデックス・コンサルタントの推奨内容の実装

インデックス・コンサルタントにより提供された SQL スクリプトを実行して、その推奨内容を実装できます。その前に、推奨内容をデータベースに関して持っている知識に照らして評価することもできます。たとえば、提案されたインデックスの名前は、分析の名前から生成されます。データベースにインデックスを作成する前に、名前を変更できます。

Copyright © 2007, iAnywhere Solutions, Inc.

推奨内容を評価するときには、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 つのインデックスが、両方とも存在する場合のみ使用される (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) こともあります。[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。
- **クラスタード・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。** クラスタード・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ード・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

- 「SQL コマンド・ファイルの使用」 720 ページ
- 「REORGANIZE TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』

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

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る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 などのデータが含まれます。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、クエリの実行プランもあります。INSERT 文、UPDATE 文、DELETE 文については、対象のロー数もあります。診断トレーシングでは、ロックとデッドロックに関する情報を記録し、多数のパフォーマンス統計値を取得できます。

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・アクティビティを行います。

- 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
- リソースの競合 (CPU、ディスク I/O、メモリ) が発生する状況
- 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す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おり、要求ロギングで記録されるすべての情報と、用意された分析に役立つ詳細情報を記録できます。要求ロギングの詳細については、「要求トレース分析の実行」 232 ページを参照してください。

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・セッションは、次の 3 つの方法のいずれかで取得できます。

1. 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。
2. [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。
3. 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。生成するトレーシング・データの量とタイプを設定する方法の詳細については、「診断トレーシングの設定」 218 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

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、または単にプロファイリング対象のデータベースといいます。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**」といいます。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、別個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おすすめします。このようにすると、運用データベースのサイズが大きくなるのを防ぐことができます。これは重要です。データベース・ファイルは一度大きくなると小さくできません。また、特に運用データベースが大きく、使用が多い場合は、トレーシング・データを格納、管理するオーバヘッドが別のデータベースで発生すると運用データベースのパフォーマンスが高くなります。

注意
Windows CE で実行しているデータベースのトレーシング・セッションを作成する場合は、[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。[アプリケーション・プロファイリング] ウィザードは使用できません。また、Windows CE 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 CE データベースのコピーに格納する必要があります。Windows CE 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 CE 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・データが格納されるテーブルを**診断トレーシング・テーブル**といいます。これらのテーブルの所有者は dbo です。これらのテーブルの詳細については、「診断トレーシング・テーブル」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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] ウィザードの推奨内容が不要になったら、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・データを運用 (ローカル) データベースに保存するか、外部トレーシング・データベース (推奨) 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・データベースがない場合は、ウィザードを使用して作成することもできます。外部トレーシング・データベースの拡張子は *.db* です。

外部トレーシング・データベースの作成方法については、「別個のトレーシング・データベースの作成」 234 ページを参照してください。

診断トレーシングの設定

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。「sa_diagnostic_tracing_level テーブル」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

診断トレーシングの設定は、次の 2 つの方法のいずれかで行うことができます。

- Sybase Central の [データベース・トレーシング] ウィザードを使用します。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します。アプリケーション・プロファイリングの管理に使用するシステム・プロシージャの詳細については、「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』と「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』を参照してください。

注意
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が使用され、[データベース・トレーシング] ウィザードを使用するときに指定するトレーシングの設定は適用されません。

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。これらのプロパティの詳細については、「データベース・レベルのプロパティ」『SQL Anywhere サーバ - データベース管理』を参照してください。

トレーシング・レベルの選択

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を「**トレーシング・タイプ**」といいます。指定できるレベルと、それぞれに含まれるトレーシング・タイプについてこの後で説明します。ここに示すトレーシング・タイプの説明については、「診断トレーシングのタイプ」 221 ページを参照してください。

トレーシングの設定をカスタマイズすると、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と、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、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、トレーシング・レベルを 3 に設定し、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。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。

Copyright © 2007, iAnywhere Solutions, Inc.

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。「トレーシングの設定の変更」 225 ページを参照してください。

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。「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』を参照してください。

トレーシング・セッションの実行中は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。ただし、変更することはできます。「トレーシング・セッションの実行中のトレーシングの設定の変更」 226 ページを参照してください。

診断トレーシングのレベル

トレーシング・レベルと、含まれる情報 (トレーシング・タイプ) の種類について次に示します。これらのレベルは、[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。各種のトレーシング・タイプの説明については、「診断トレーシングのタイプ」 221 ページを参照してください。

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・カウンタが収集され、実行された文がサンプリングされます (5 秒に 1 回)。このレベルでは、次のトレーシング・タイプがあります。

- volatile_statistics、1 秒ごとのサンプリング
- non_volatile_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・カウンタが収集され、実行された文がすべて記録され、実行されたプランがサンプリングされます (5 秒に 1 回)。このレベルでは、次のトレーシング・タイプがあります。

- volatile_statistics、1 秒ごとのサンプリング
- non_volatile_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。このレベルでは、次のトレーシング・タイプがあります。

- volatile_statistics、1 秒ごとのサンプリング

- non_volatile_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 (通常はカラム名やデータベース名などの識別子) 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、トレースを行うユーザの名前です。
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。

 Copyright © 2007, iAnywhere Solutions, Inc.

scope カラムの値	説明
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、実体化ビュー (Materialized View)、非実体化ビュー (Non-materialized View) 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」 221 ページ
- 「診断トレーシング条件」 223 ページ

診断トレーシングのタイプ

次は、診断トレーシングに設定できる「**トレース・タイプ**」のリストです。トレース・タイプは、トレーシング・データを生成する情報のタイプを制御します。次に示すように、各トレース・タイプには対応する条件が必要です。トレース・タイプの値は、dbo.sa_diagnostic_tracing_level diagnostic 診断テーブルの trace_type カラムに格納されます。対応する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。使用可能なすべての条件のリストについては、「診断トレーシング条件」 223 ページを参照してください。

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一定の間隔で収集します。 スコープと条件：このトレース・タイプは、DATABASE スコープと SAMPLE_EVERY 条件が必要です。
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一定の間隔で収集します。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：このトレース・タイプは、DATABASE スコープが必要で、SAMPLE_EVERY 条件を指定できます。

trace_type カラムの値	説明
CONNECTION_STATISTICS	接続の統計値のサンプルを一定の間隔で収集します。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：このトレース・タイプは DATABASE、USER、CONNECTION_NUMBER、CONNECTION_NAME の各スコープで使用でき、SAMPLE_EVERY 条件を指定できます。
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：このトレース・タイプはすべてのスコープで使用でき、NONE、NULL、SAMPLE_EVERY のいずれかの条件を指定できます。
PLANS	条件とスコープに従ってクエリの実行プランを収集します。 スコープと条件：このトレース・タイプはすべてのスコープで使用でき、NONE、NULL、SAMPLE_EVERY、ABSOLUTE_COST のいずれかの条件を指定できます。
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：このトレース・タイプはすべてのスコープで使用でき、すべての条件を指定できます。
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。このトレース・タイプは、STATEMENTS_WITH_VARIABLES、PLANS、PLANS_WITH_STATISTICS、OPTIMIZATION_LOGGING、OPTIMIZATION_LOGGING_WITH_PLANS のいずれかのトレース・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：このトレース・タイプはすべてのスコープで使用でき、すべての条件を指定できます。
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：このトレース・タイプはすべてのスコープで使用でき、すべての条件を指定できます。

Copyright © 2007, iAnywhere Solutions, Inc.

trace_type カラムの値	説明
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。このトレース・タイプは、OPTIMIZATION_LOGGING_WITH_PLANS トレース・タイプを指定すると自動的に追加されます。 このトレース・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。
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 トレース・タイプは、OPTIMIZATION_LOGGING_WITH_PLANS トレース・タイプを指定すると自動的に追加されます。 このトレース・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」 220 ページ
- 「診断トレーシング条件」 223 ページ

診断トレーシング条件

次は、診断トレーシングに設定できる「**条件**」のリストです。条件は、特定のトレース・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、ミリ秒で指定した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」 220 ページ
- 「診断トレーシングのタイプ」 221 ページ

現在のトレーシングの設定の確認

現在のトレーシングの設定を表示するには、Sybase Central で [データベース・トレーシング] ウィザードを起動します。設定を確認したら、[キャンセル] をクリックしてウィザードを終了します。sa_diagnostic_tracing_level テーブルへのクエリで、現在、有効になっているトレーシングの設定を取得することもできます。

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。

♦ **現在のトレーシングの設定を確認するには、次の手順に従います (Sybase Central の場合)。**

1. DBA と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] を選択します。

   [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] をクリックして終了します。
3. データベースを右クリックし、ポップアップ・メニューで [トレーシング] を選択します。

   初めてトレーシングを使用する場合、またはトレーシングの設定をクリアした場合は、[データベース・トレーシング] ウィザードが表示されます。それ以外の場合は、[トレーシング] – [設定] を選択してトレーシングを開始します。
4. ウィザードを進み、[トレーシング・レベルの編集] ページを表示します。このページには、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。トレーシング・セッションが実行中である必要はありません。

 Copyright © 2007, iAnywhere Solutions, Inc.

5. 設定を確認したら、[キャンセル] をクリックしてウィザードを終了します。

♦ **現在のトレーシングの設定を確認するには、次の手順に従います (Interactive SQL の場合)。**

1. DBA として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。

   データベース・サーバから、現在使用中の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 diagnostic テーブルに問い合わせ、現在の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』

トレーシングの設定の変更

トレーシングの設定は、運用データベースに固有です。つまり、運用データベースに対して指定するトレーシングの設定は、別の運用データベースに対して行うトレースに影響しません。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] ウィザードを起動する方法については、「トレーシング・セッションの作成」 226 ページを参照してください。

[データベース・トレーシング] ウィザードで設定する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。

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・レベルを変更することもできます。この方法では、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、スコープ、条件、値などの他の設定が限られています。このプロシージャの詳細については、「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』を参照してください。

♦ トレーシング・レベルを変更するには、次の手順に従います (Interactive SQL の場合)。

1. DBA として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。

例

次の文は、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 1 に設定します。:

```
CALL sa_set_tracing_level( 1 );
```

既存のトレーシングの設定は、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トレーシングの設定で上書きされます。各トレーシング・レベルに関連付けられているデフォルトの設定については、「診断トレーシングのレベル」 219 ページを参照してください。

トレーシング・セッションの実行中のトレーシングの設定の変更

Sybase Central で [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・セッションの実行中にトレーシングの設定を変更できます。

♦ トレーシング・セッション中にトレーシングの設定を変更するには、次の手順に従います (Sybase Central の場合)。

1. DBA としてデータベースに接続します。
2. コンテキスト・ドロップダウン・リストからデータベースを選択します。
3. [ファイル] – [トレーシング] – [トレーシング・レベルの変更] を選択します。
4. トレーシングの設定を変更し、[OK] をクリックします。

   データベース・サーバによって、既存の診断トレーシングの設定が新しい設定で上書きされます。

トレーシング・セッションの作成

トレーシング・セッションを開始するときに、実行するトレースのタイプを設定し、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。

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を実行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。「TCP/IP プロトコルの使用」『SQL Anywhere サーバ - データベース管理』を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

注意
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。

♦ **トレーシング・セッションを作成するには、次の手順に従います (Sybase Central の場合)。**

1. プロファイル情報を取得する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] を選択します。

   [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] をクリックして終了します。
3. データベースを選択して、[ファイル] − [トレーシング] を選択します。サブメニューが表示される場合は、[トレーシングの設定と開始] を選択します。

   [データベース・トレーシング] ウィザードが表示されます。
4. [データベース・トレーシング] ウィザードの指示に従って、トレーシング・セッションを設定し、取得します。

   a. [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、必要に応じてトレーシングのスコープを小さくします。

   b. [トレーシング・レベルの編集] ページで、必要に応じてトレーシングの設定をカスタマイズします。

   c. [外部データベースの作成] ページで、次の操作を行います。

      ♦ [Create a New Tracing Database] を選択します。
      ♦ 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。
      ♦ [現在のサーバでデータベースを起動] を選択します。
      ♦ [データベースの作成] をクリックします。

        [データベース・トレーシング] ウィザードによって、運用データベースからスキーマとパーミッションの情報がアンロードされ、新たに作成されたトレーシング・データベースにロードされます。
      ♦ [次へ] をクリックします。

   d. [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] を選択します。
      ♦ トレーシング・データベース用に指定したユーザ名とパスワードを指定します。運用データベースに接続するために使用したユーザ名やパスワードと一致するはずです。

- ♦ 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指定します。次に例を示します。

  ```
  ENG=Server47;DBN=TracingDB
  ```

  e. [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。] で、保存するトレーシング・データベースのボリュームについてオプションを選択します。

  f. [完了] をクリックします。

5. しばらくの間、アプリケーションによってデータベースが操作され、データが収集されるのを待ちます。

6. 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] を選択します。

   [データベース・トレーシング] ウィザードによってトレーシング・セッションが終了され、取得されたデータが診断テーブルに格納されます。

♦ トレーシング・セッションを作成するには、次の手順に従います (Interactive SQL の場合)。

1. DBA として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。
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。

   Sybase Central のアプリケーション・プロファイリング・モードで診断トレーシングのデータを表示できます。「アプリケーション・プロファイリング」 202 ページを参照してください。

例

この例は、現在のデータベースのトレースを開始し、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。この例はすべて 1 行で記述します。

```
ATTACH TRACING TO
'UID=DBA;PWD=sql;ENG=dbsrv10;DBN=tracing;LINKS=tcpip' LIMIT HISTORY 2 HOURS;
```

この例は、現在のデータベースのトレースを開始し、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、トレースを停止し、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、トレースを停止し、診断データを保存しない方法を示します。

```
DETACH TRACING WITHOUT SAVE;
```

 Copyright © 2007, iAnywhere Solutions, Inc.

参照

- 「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、トレーシング・セッションに対して設定したトレーシング・レベルと設定に対応するすべてのアクティビティの記録です。したがって、データを分析するときは、設定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。

パフォーマンスの問題のトラブルシューティング

多くの場合、パフォーマンスの問題は複数の要因の組み合わせです。アプリケーション・プロファイリング機能を使用すると、パフォーマンスの問題が次の要因に起因するかどうかを確認できます。

- 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

パフォーマンスのトラブルシューティング・シナリオの最初のタスクは、速度が下がった一番の原因がアプリケーションであるか、データベース・サーバであるかを判断することです。アプリケーション・クライアントが受け取ったデータを処理する時間が、合計実行時間のほとんどを占めている場合は、データベース・サーバはほとんどの時間アイドル状態の可能性があります。これは、アプリケーション・プロファイリング・ツールの [詳細] ビューで確認できます (結果を単一の接続でフィルタします)。この接続からの要求の間隔が大きいかどうかを確認します。間隔が大きい場合、速度が下がった一番の原因はアプリケーション・クライアント自体にあります。

速度が下がった一番の原因がデータベース・サーバにある場合は、その原因を特定します。

速度が遅い文の検出

[概要] ビューと [詳細] ビューを使用して、データベース・サーバで処理に最も時間を要している文を特定できます。[概要] ビューでは、似ている文をグループとして、合計呼び出し回数と合計

処理時間が表示されます。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。他の文は 1 つにまとめられます。たとえば、CREATE TABLE 文は [概要] ビューで 1 つのエントリになります。文が [概要] ビューでコストが高いと表示される場合、その文は本質的に高コストの文である可能性と、その文自体のコストは低いが頻繁に実行された可能性があります。

[詳細] ビューには、トレーシング・セッション中に取得された各文とその実行時間が表示されます。表示される各文の継続時間は、データベース・サーバで要求の処理に実際に要した時間です。カーソルを長時間開いたままでも (開始時間とカーソルのクローズ時間の間隔が長い)、データベース・サーバで、このカーソルに小規模な結果セットが作成されただけの場合は、継続時間が短い場合があります。

文を右クリックすると、さらに詳細を表示できます。文の完全な SQL テキスト、文がいつ使われたか、そのコンテキストが表示されます。表示される文のテキストは、元のテキストとは一致しない場合があります。この文がデータベース・サーバによって解析された後で取得された場合や (ストアド・プロシージャやトリガなど、コンパイル済みデータベース・オブジェクトの一部であるため)、サンプリングまたはコストの条件を満たすためにトレースに含めるように選択された場合は、最初の記述と異なる場合があります。特に、ビューの定義は多くの場合、クエリに展開される (インラインで展開される) ので、ビュー以降のクエリは大きく異なる場合があります。

この文がクエリだった場合、[More Details] ダイアログには、クエリの実行に使用されたプランの詳細も表示されます。クエリ・プランのテキストは常に取得され、データベース・サーバで実際に使用されたアクセス・プランを正確に表します。プランに選択した設定によっては、表示されるグラフィカルなプランが、データベース・サーバで実際に使用されたプランではない場合があります。グラフィカルなプランの説明が「最も妥当なプラン」の場合は、データベース・サーバによって、最初に最適化されたときのデータベース・サーバの条件をできるかぎりシミュレートしてクエリが再度最適化されています。このとき、通常は同じプランが選択されます。ただし、妥当なプランが実際のテキスト・プランと一致することを確認してください。「実行プランの解釈」 571 ページを参照してください。

ハードウェア・リソースが制限要因であるかどうかの判断

データベース・システムの負荷が増えると、最終的に 1 つまたは複数のハードウェア・リソースによってパフォーマンスが制限されます。通常は、CPU サイクル、メモリ容量、またはディスク I/O 帯域幅のいずれかのリソースが不足します。この場合、アプリケーションまたはデータベース・サーバの処理の効率が悪い可能性があります。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・リソースを追加し、データベース・サーバでさらに大きな負荷を処理できるようにし、また既存の負荷のパフォーマンスを向上できます。一般的な非効率の原因とその推奨される解決方法については、「パフォーマンスの問題のトラブルシューティング」 229 ページを参照してください。

リソースを追加してもスケーラビリティの問題がすべて解決しない場合があります。また、リソースを追加しても、コンピュータの機能が線形的に向上することはまれです。たとえば、データベース・サーバに割り当てられた CPU が完全に使用されている場合、さらに大きな負荷を処理するには、CPU リソースを追加する必要があります。ただし、データベース・サーバで使用できる CPU を倍増しても、同じ時間に実行できる処理の量は 2 倍になりません。

Copyright © 2007, iAnywhere Solutions, Inc.

[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] タブを使用して複数のデータベース統計情報を検査し、ハードウェア・リソースがパフォーマンスの制限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、[統計の追加] ボタンをクリックし、[ProcessCPU] を選択します。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、[統計の追加] ボタンをクリックし、[CacheHits] と [CacheReads] を選択します。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 ー 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、[統計の追加] ボタンをクリックし、[CurrIO] を選択します。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」 246 ページ

アプリケーション論理のデバッグ

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、関数、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。

内部データベース・オブジェクト (プロシージャ、トリガなど) 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トレース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。「プロシージャ、関数、トリガ、イベントのデバッグ」 837 ページを参照してください。

デッドロックの調査

デッドロックを調べる必要もあります。デッドロックは、複数の接続がそれぞれ他方の接続が終了するのを待っていて、いずれも終了できない場合に発生します。トレーシング・セッション中にデッドロックが発生した場合は、アプリケーション・プロファイリング・モードで調べることができます。調べるには、トレーシング・セッションを開き、Sybase Central の [アプリケーション・プロファイリングの詳細] 領域 (下のウィンドウ枠) の [デッドロック] タブを表示します。

[デッドロック] タブには、トレーシング・セッション中に記録された各デッドロックがグラフィックに表示されます。関連する接続と、実行しようとしたときに各接続でブロックされた文を確認できます。ローカル・データベースまでトレースした場合は、ロックを取得できなかったローのプライマリ・キーも確認できます。

デッドロックとその原因の詳細については、「トランザクションのブロックとデッドロック」 140 ページを参照してください。

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。次に、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して、競合やボトルネックの可能性を特定します。

♦ **要求トレース分析を行うには、次の手順に従います。**

1. DBA と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] を選択します。

   [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] をクリックして終了します。
3. 分析するデータベースを選択し、[ファイル] – [トレーシング] を選択します。[データベース・トレーシング] ウィザードを初めて使用する場合、または以前にトレーシングの設定を

 Copyright © 2007, iAnywhere Solutions, Inc.

クリアした場合は、ウィザードが自動的に起動します。サブメニューが表示された場合は、[トレーシングの設定と開始] を選択します。

4. [データベース・トレーシング] ウィザードの指示に従って、トレーシング・セッションを設定、取得します。特に、スコープを、問題があるユーザか接続、またはその両方に設定します。[トレーシング・レベルの編集] ページでは、収集する診断データのすべての条件をカスタマイズできます。

5. トレースの詳細の設定を終了したら、[完了] をクリックします。

   トレーシング・セッションが開始します。しばらくの間、アプリケーションによってデータベースが操作され、データが収集されるのを待ちます。

6. 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] を選択します。

   すると、トレーシング・セッションが停止し、トレーシング・セッションの取得されたデータが格納されます。

7. 分析を実行します。

   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、[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。

      [分析を開くかトレーシング・データベースに接続] ダイアログが表示されます。

   b. [トレーシング・データベース内] を選択し、[開く] をクリックします。

      [トレーシング・データベースに接続] ダイアログが表示されます。

   c. 分析している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、[OK] 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、トレーシング・データベースと、このデータベースに格納されたトレーシング・セッションに関する情報が表示されます。

   d. [ロギング・セッション ID] ドロップダウンから、リストの最後のエントリ (最後に取得されたセッション) を選択します。

      トレーシング・セッションの詳細が [アプリケーション・プロファイリングの詳細] ウィンドウ枠に表示されます。

   e. [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。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、リスト内で検索する文字列を [SQL 文に含まれる内容] フィールドに入力します。

f. 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] を選択します。すると、[詳細] タブにさらに詳細な情報が表示されます。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。

別個のトレーシング・データベースの作成

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。ただし、データベースで同時に 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最小限に抑えるために、トレーシング・データを別個のトレーシング・データベースに格納することをおすすめします。

別個のトレーシング・データベースを作成するには、[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・セッションを開始するときに作成する方法が最も簡単です。[データベース・トレーシング] ウィザードによって、トレーシング・データベースの作成時にスキーマとパーミッションの情報が運用データベースからアンロードされます。作成するトレーシング・データベースには、後で実行するトレーシング・セッションのデータも格納できます。トレーシング・セッションの作成方法の詳細については、「トレーシング・セッションの作成」 226 ページを参照してください。

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・セッションを作成しないでトレーシング・データベースだけを作成できます。

♦ アンロード・ユーティリティ (dbunload) を使用して別個のトレーシング・データベースを作成するには、次の手順に従います。

1. DBA としてデータベースに接続します。
2. 次のような 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。

   ```
   dbunload -c "UID=DBA;PWD=sql;ENG=sample;DBN=sample" -an tracing.db -n -k
   ```

   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 *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。
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。

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』

Copyright © 2007, iAnywhere Solutions, Inc.

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。「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」 216 ページを参照してください。

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。

警告
要求ログには SQL 文の完全なテキストが含まれるので、GRANT CONNECT 文、CREATE DATABASE 文、CREATE EXTERNAL LOGIN 文などのパスワードを含む SQL 文の場合、これは機密情報になります。セキュリティが心配な場合は、要求ログ・ファイルへのアクセスを制限してください。

-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 サーバ・オプション」 『SQL Anywhere サーバ - データベース管理』
- 「-zo サーバ・オプション」 『SQL Anywhere サーバ - データベース管理』
- 「-zn サーバ・オプション」 『SQL Anywhere サーバ - データベース管理』
- 「-zs サーバ・オプション」 『SQL Anywhere サーバ - データベース管理』

Copyright © 2007, iAnywhere Solutions, Inc.

> **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 と summarize satmp_request_time を satmp_request_profile という別のグローバル・テンポラリ・テーブルに呼び出すことができます。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。

これらのシステム・プロシージャの詳細については、「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』と「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』を参照してください。

要求ログのフィルタ

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。

sa_server_option システム・プロシージャの詳細については、「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』を参照してください。

♦ 接続に基づいてフィルタするには、次の手順に従います。

- 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いてフィルタするには、次の手順に従います。

- 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。

 Copyright © 2007, iAnywhere Solutions, Inc.

♦ **フィルタをリセットするには、次の手順に従います。**

- 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出力できます。

♦ **ホスト変数の値を含めるには、次の手順に従います。**

- 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、Sybase Central で実行するか、Interactive SQL でシステム・プロシージャ呼び出しを使用して実行できます。Sybase Central の方が機能が豊富で柔軟性に優れています。このため、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してプロシージャ・プロファイリングを実行することをおすすめします。「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」 204 ページを参照してください。

sa_server_option を使用したプロファイリングの有効化

この項では、Interactive SQL で 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方法について説明します。

このシステム・プロシージャの構文と返される結果の詳細については、「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』を参照してください。

プロシージャ・プロファイリングを有効にして使用するには、DBA 権限が必要です。

Copyright © 2007, iAnywhere Solutions, Inc.

♦ Interactive SQL でプロシージャ・プロファイリングを有効にするには、次の手順に従います。

1. DBA として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、次のように入力します。

   ```
   CALL sa_server_option( 'ProcedureProfiling' , 'ON' );
   ```

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。

♦ Interactive SQL でプロシージャ・プロファイリングをユーザ別にフィルタするには、次の手順に従います。

1. DBA として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' );
   ```

   userid の値は、モニタされるユーザの名前です。

sa_server_option を使用したプロファイリングのリセット

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、関数、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。この項では、Interactive SQL で 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リセットする方法について説明します。

このシステム・プロシージャの構文と返される結果の詳細については、「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』を参照してください。

次の項の説明は、DBA としてすでにデータベースに接続していることと、プロシージャ・プロファイリングが有効になっていることを前提としています。

♦ Interactive SQL でプロファイリングをリセットするには、次の手順に従います。

- 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、次のように入力します。

  ```
  CALL sa_server_option( 'ProcedureProfiling' , 'RESET' );
  ```

 Copyright © 2007, iAnywhere Solutions, Inc.

sa_server_option を使用したプロファイリングの無効化

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。この項では、Interactive SQL で 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無効にする方法について説明します。

このシステム・プロシージャの構文と返される結果の詳細については、「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』を参照してください。

♦ プロファイリングを無効にするには、次の手順に従います (Interactive SQL の場合)。

・ 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 に設定します。

たとえば、次のように入力します。

```
CALL sa_server_option( 'ProcedureProfiling' , 'OFF' );
```

♦ プロファイリングを無効にし、既存のデータをクリアするには、次の手順に従います (Interactive SQL の場合)。

・ 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 CLEAR に設定します。

たとえば、次のように入力します。

```
CALL sa_server_option( 'ProcedureProfiling' , 'CLEAR' );
```

システム・プロシージャを使用したプロファイリング情報の取り出し

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、関数、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。DBA としてデータベースに接続する必要があります。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_server_option を使用したプロファイリングの有効化」 237 ページを参照してください。

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。

これらのシステム・プロシージャの構文と返される結果については、「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』と「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』を参照してください。

♦ プロファイリング一覧情報を表示するには、次の手順に従います (Interactive SQL の場合)。

1. sa_procedure_profile_summary システム・プロシージャを実行します。

   たとえば、次のように入力します。

   ```
   CALL sa_procedure_profile_summary;
   ```

2. [SQL] – [実行] を選択します。

   データベースにあるプロシージャすべての情報を含む結果セットが、[結果] ウィンドウ枠に表示されます。

♦ 詳細なプロファイリング情報を表示するには、次の手順に従います (Interactive SQL の場合)。

1. sa_procedure_profile システム・プロシージャを実行します。

   たとえば、次のように入力します。

   ```
   CALL sa_procedure_profile;
   ```

2. [SQL] – [実行] を選択します。

   プロファイリング情報を含む結果セットが [結果] ウィンドウ枠に表示されます。

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表示できます。この機能は、特定のクエリのパフォーマンス上の問題を診断するのに役立ちます。たとえば、プラン内の情報に基づいてデータベースのどこにインデックスを追加するかを判断できます。Interactive SQL で [ファイル] – [プランの保存] を選択すると、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。

注意
グラフィカルなプランは、Sybase Central のアプリケーション・プロファイリング・モードからも利用できます。Sybase Central のアプリケーション・プロファイリング機能の詳細については、「アプリケーション・プロファイリング」202 ページを参照してください。

グラフィカルなプランは、短いプランや長いプランよりも多くの情報を提供します。グラフィカルなプランの表示は、統計情報付きか統計情報なしかを選択できます。どちらを選択した場合

 Copyright © 2007, iAnywhere Solutions, Inc.

も、プランの中で最も高コストとして推定された部分をすぐに表示できます。統計情報付きのグラフィカルなプランの表示は高コストですが、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。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、実際の統計と推定とに違いがあると予想してください。グラフィカルなプランは、アクセス・プランのデフォルト・フォーマットです。

グラフィカルな図でノードをクリックすると、プラン内のノードに関する詳細情報を取得できます。統計情報付きのグラフィカルなプランには、グラフィカルなプランに表示されるすべての推定に加えて、文を実行したときの実際の実行時間コストが表示されます。これを行うには、文を実際に実行します。これは、コストの高いクエリの場合、プランへのアクセスに遅延が生じる可能性があることを意味します。また、削除や更新など、クエリのどのような部分も実際に実行しなければなりませんが、ロールバックを利用すればこれらの変更は取り消し可能です。

パフォーマンスに問題があり、推定されるロー・カウントや実行時間が予測とは異なる場合は、統計情報付きのグラフィカルなプランを使用してください。統計情報付きのグラフィカルなプランを使用すると、推定と実際の統計を比較できます。実際値と推定値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適切な推定を準備できない可能性があることを示す警告とみなすことができます。

次に、統計情報付きのグラフィカルなプランでチェックできる主な統計と、考えられる対応策を示します。

- ロー・カウントは、結果セット内のローを測定します。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、基本となっている述部の選択性が正しくない可能性があります。
- 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。一般的に、推定は正確ではないことがあります。ただし、極端に大きな誤差は問題がある可能性を示しています。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、最後の手段として、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。
- 実行時間は、クエリの実行所要時間を測定します。テーブル・スキャンまたはインデックス・スキャンの実行時間が正しくない場合は、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。
- 推定ソースが Guess の場合、オプティマイザが使用する情報はなく、問題を示している可能性があります。推定ソースが Index で、選択性推定が正しくない場合は、インデックスに無駄が多い可能性があります。REORGANIZE INDEX 文を使用してインデックスの断片化をデフラグすると改善できる場合があります。
- キャッシュの読み込み数とヒット数がまったく同じ場合は、データベース全体がキャッシュにあり、良好な状態です。読み込み数がヒット数よりも多い場合は、データベース・サーバ

がキャッシュにアクセスしようとしたが失敗し、ディスクから読み込むことを意味します。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・ヒット率が低いことは、パフォーマンスの問題を示す場合があり、キャッシュ・サイズを大きくすると改善できる可能性があります。

参照

- 「グラフィカルなプラン」 581 ページ
- 「実行プランへのアクセス」 588 ページ
- 「実行プランの解釈」 571 ページ

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、*samples-dir¥SQLAnywhere* にあります。*samples-dir* 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。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

データベースのパフォーマンスのモニタリング

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。これらの統計値は次の 3 つの方法でアクセスできます。

- **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。「Sybase Central パフォーマンス・モニタを使用したモニタリング」 243 ページを参照してください。
- **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。「Windows パフォーマンス・モニタを使用した統計値のモニタリング」 245 ページを参照してください。
- **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。「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』を参照してください。
- **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。「SQL 関数を使用した統計値のモニタ」 255 ページを参照してください。

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。診断トレーシングの詳細については、「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」 216 ページを参照してください。

モニタできる SQL Anywhere のすべての統計値のリストについては、「パフォーマンス・モニタの統計値」 246 ページを参照してください。

Sybase Central パフォーマンス・モニタを使用したモニタリング

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情報を詳細に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。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。「Windows パフォーマンス・モニタを使用した統計値のモニタリング」 245 ページを参照してください。

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。

モニタできる SQL Anywhere のすべての統計値のリストについては、「パフォーマンス・モニタの統計値」 246 ページを参照してください。

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。パフォーマンス・モニタのグラフの統計値の追加と削除の詳細については、「統計値の追加と削除」 244 ページを参照してください。

♦ パフォーマンス・モニタを開くには、次の手順に従います。

1. 左ウィンドウ枠で対象のサーバを選択します。

2. 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。

参照

♦ 「Windows パフォーマンス・モニタを使用した統計値のモニタリング」 245 ページ
♦ 「統計値の追加と削除」 244 ページ

統計値の追加と削除

♦ 統計値を Sybase Central パフォーマンス・モニタに追加するには、次の手順に従います。

1. Sybase Central の左ウィンドウ枠で対象のサーバを選択します。

2. 右ウィンドウ枠で、[統計情報] タブをクリックします。

3. グラフ化されていない統計値を右クリックし、ポップアップ・メニューで [パフォーマンス・モニタに追加] を選択します。

♦ 統計値を Sybase Central パフォーマンス・モニタから削除するには、次の手順に従います。

1. Sybase Central の左ウィンドウ枠で対象のサーバを選択します。

2. 右ウィンドウ枠で、[統計情報] タブをクリックします。

3. グラフ表示されている統計値を右クリックし、ポップアップ・メニューで [パフォーマンス・モニタから削除] を選択します。

ヒント
統計値の追加と削除は、統計値のプロパティ・シートにある [パフォーマンス・モニタ] からもできます。

 Copyright © 2007, iAnywhere Solutions, Inc.

モニタできる SQL Anywhere のすべての統計値のリストについては、「パフォーマンス・モニタの統計値」 246 ページを参照してください。

参照

- 「Sybase Central パフォーマンス・モニタを開く」 244 ページ
- 「Windows パフォーマンス・モニタを使用した統計値のモニタリング」 245 ページ

Windows パフォーマンス・モニタを使用した統計値のモニタリング

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。

Windows パフォーマンス・モニタは、Windows に付属します。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。

SQL Anywhere でモニタできるパフォーマンス統計値の完全なリストについては、「パフォーマンス・モニタの統計値」 246 ページを参照してください。

♦ Windows パフォーマンス・モニタを使用するには、次の手順に従います。

1. SQL Anywhere サーバが実行されている状態でパフォーマンス・モニタを起動します。

   - Windows の [コントロール パネル] で [管理ツール] を選択します。
   - [パフォーマンス] を選択して Windows パフォーマンス・モニタを起動します。

2. ツールバーのプラス記号ツール (+) を選択します。[カウンタの追加] ダイアログが表示されます。

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 10 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 10 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 10 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 接続] または [SQL Anywhere データベース] を選択した場合は、インスタンスの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。[Ctrl] キーまたは [Shift] キーを押しながらクリックすると、複数の統計値を選択できます。

5. [SQL Anywhere 10 接続] または [SQL Anywhere 10 データベース] を選択した場合は、インスタンスのボックスから、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説明を表示するには、[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. 表示したいカウンタをすべて選択したら [閉じる] をクリックします。

パフォーマンス・モニタの統計値

SQL Anywhere では、さまざまな統計値を使用できます。処理速度は毎秒レポートされます。統計値は、次のように分類されています。

- 「キャッシュの統計値」 246 ページ
- 「チェックポイントとリカバリの統計値」 247 ページ
- 「通信の統計値」 248 ページ
- 「ディスク I/O の統計値」 249 ページ
- 「ディスク読み込みの統計値」 250 ページ
- 「ディスク書き込みの統計値」 250 ページ
- 「インデックスの統計値」 251 ページ
- 「メモリ・ページの統計値」 252 ページ
- 「要求の統計値」 253 ページ
- 「その他の統計値」 254 ページ

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・ページがパージされている頻度を示す。

Copyright © 2007, iAnywhere Solutions, Inc.

統計情報	スコープ	説明
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す。
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・ログのサイズ (ページ数) を示す。
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す。

統計情報	スコープ	説明
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す。
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) を示す。

通信の統計値

これらの統計値により、クライアント／サーバ間の通信状況を分析できます。

統計情報	スコープ	説明
通信：受信バイト数／秒	接続とサーバ	ネットワーク・データが受信される速度 (バイト単位) を示す。
通信：無圧縮受信バイト数／秒	接続とサーバ	圧縮が無効になっていた場合のバイトの受信速度を示す。
通信：送信バイト数／秒	接続とサーバ	ネットワークでバイトが送信される速度を示す。
通信：無圧縮送信バイト数／秒	接続とサーバ	圧縮が無効になっていた場合のバイトの送信速度を示す。

Copyright © 2007, iAnywhere Solutions, Inc.

統計情報	スコープ	説明
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す。
通信：受信パケット数／秒	接続とサーバ	ネットワーク・パケットの受信速度を示す。
通信：無圧縮受信パケット数／秒	接続とサーバ	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す。
通信：送信パケット数／秒	接続とサーバ	ネットワーク・パケットの送信速度を示す。
通信：無圧縮送信パケット数／秒	接続とサーバ	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す。この統計は TCP/IP の場合にのみ収集されます。
通信：受信要求数	接続とサーバ	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] が到達した最大値を示す。

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す。
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す。Windows と NetWare プラットフォームではバッファなし (direct) 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す。

 Copyright © 2007, iAnywhere Solutions, Inc.

統計情報	スコープ	説明
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。

統計情報	スコープ	説明
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・ページがパージされている頻度を示します。
ヒープ：カーバ	接続とサーバ	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。
ヒープ：クエリ処理	接続とサーバ	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	接続とサーバ	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	接続とサーバ	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プされている頻度を示します。
メモリ・ページ：カーバ	接続とサーバ	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	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	接続とサーバ	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。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

統計情報	スコープ	説明
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。
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	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	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。
要求：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。

統計情報	スコープ	説明
文キャッシュ・ヒット	接続とサーバ	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	接続とサーバ	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	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	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	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	ロールバック要求が処理される頻度を示します。

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。
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。

Copyright © 2007, iAnywhere Solutions, Inc.

SQL 関数を使用した統計値のモニタ

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。「PROPERTY 関数 [システム]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
- **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。「DB_PROPERTY 関数 [システム]」『SQL Anywhere サーバ - SQL リファレンス』と「DB_EXTENDED_PROPERTY 関数 [システム]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
- **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。「CONNECTION_PROPERTY 関数 [システム]」『SQL Anywhere サーバ - SQL リファレンス』と「CONNECTION_EXTENDED_PROPERTY 関数 [文字列]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

システム関数を使って取得できるプロパティの完全なリストについては、「システム関数」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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' );
```

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。

Copyright © 2007, iAnywhere Solutions, Inc.

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 );
```

PROPERTY_NUMBER 関数の詳細については、「PROPERTY_NUMBER 関数 [システム]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

これらの統計値の多くは、Sybase Central のパフォーマンス・モニタを使って、グラフ形式で閲覧できます。

 Copyright © 2007, iAnywhere Solutions, Inc.

パフォーマンス向上のためのヒント

SQL Anywhere は、非常に優れたパフォーマンスを自動的に提供します。ここでは、この製品から最高のパフォーマンスを引き出すためのヒントを提供します。

適切なハードウェアの入手

PC で実行する場合は、CPU、メモリ、ディスクについて最低要件が満たされていることを確認してください。

- SQL Anywhere は、最低 4 MB のメモリで実行できます。Sybase Central や Interactive SQL などの管理ツールを使用する場合、SQL Anywhere には RAM が 32 MB 以上必要です。使用するコンピュータには、このメモリ容量に加えて、オペレーティング・システム用のメモリ容量が必要になります。
- データベースとログ・ファイルを保持するのに十分なディスク領域。
- これらは、最低必要容量であることに留意してください。ハードウェア要件を最低限満たしている場合で、パフォーマンスが思わしくない場合は、ハードウェアの一部または全体のアップグレードを検討してください。一般論として、データベース・サーバにかかる負荷に対してハードウェアの構成が適切かどうか評価してください。

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。

詳細については、「-fc 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。

常にトランザクション・ログを使用する

SQL Anywhere はトランザクション・ログがない方がディスクで管理する情報が少なくなるので実行が速くなる、と思っているユーザもいるかもしれません。しかし、実際はその逆です。トランザクション・ログは優れた保護を提供するだけでなく、パフォーマンスを飛躍的に改善します。

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。こうした変更の書き込みには、大量のリソースが消費されます。

しかし、トランザクション・ログを使用すると、SQL Anywhere では変更が発生したときにその詳細を示すメモを書き込めばいいだけです。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。

Copyright © 2007, iAnywhere Solutions, Inc.

ヒント
必ずトランザクション・ログを使用するようにしてください。トランザクション・ログはデータの保護を助け、しかもパフォーマンスを大いに向上させます。

トランザクション・ログを、メイン・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。

同時性の問題点の確認

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、テーブルのローを 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多数のユーザが同時にアクセスする場合、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。「sa_locks 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。

ロック状態が検出されると、関連する接続プロセス情報を AppInfo 接続プロパティを使って表示できます。「接続レベルのプロパティ」 『SQL Anywhere サーバ - データベース管理』を参照してください。

オプティマイザの優先度の選択

optimization_goal オプションは、SQL Anywhere において SQL 文を応答時間 (最初のロー) に対して最適化するか、リソースの総消費量 (すべてのロー) 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。

クエリの FROM 句の FASTFIRSTROW テーブル・ヒントを使用すると、特定のクエリの最適化ゴールを First-row に設定できます。この場合、optimization_goal の設定を変更する必要はありません。

 Copyright © 2007, iAnywhere Solutions, Inc.

このオプションを All-rows (デフォルト) に設定すると、SQL Anywhere はクエリを最適化して、予測される合計検索時間が最短のアクセス・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 [データベース]」『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』

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。「CREATE STATISTICS 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

詳細については、「テーブル制約とカラム制約の使い方」 106 ページを参照してください。

キャッシュ・サイズの拡大

SQL Anywhere では、最近使用されたページをキャッシュに格納します。そのページに複数回アクセスする要求が生じても、また別の接続から同じページが要求されても、そのページはすでにメモリに入っているため、ディスクから情報を読み込む必要がありません。暗号化されていない場合に比べて大きいキャッシュを必要とする暗号化データベースには、これが特に重要です。

キャッシュ容量が小さすぎると、SQL Anywhere はメモリにページを長く維持できず、キャッシュの利点を生かすことができません。

UNIX と Windows では、データベース・サーバによってキャッシュ・サイズが必要に応じて動的に変更されます。それでも、キャッシュは物理的に使用可能なメモリ容量と他のアプリケーションが使用する容量によって制限されます。

Copyright © 2007, iAnywhere Solutions, Inc.

Novell NetWare では、キャッシュ・サイズはデータベース・サーバ起動時に設定されます。同時に実行される他のアプリケーションやプロセスの要件を考慮に入れて、できるだけ多くのメモリをデータベースのキャッシュに割り付けるようにしてください。

> **ヒント**
> キャッシュ・サイズを拡大すると、メモリから情報を取り出す方がディスクから読み込むより数倍速いので、パフォーマンスを飛躍的に向上できます。キャッシュを拡大するために RAM をさらに追加するのも無駄ではない場合もあります。

詳細については、「パフォーマンス向上へのキャッシュの使用」 273 ページを参照してください。

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、パフォーマンスの点で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。「データ整合性の概要」 96 ページを参照してください。

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できるように多数のツールが用意されています。これらのツールは、この後に示す *samples-dir¥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。*samples-dir* 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。

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。

fetchtst

機能　結果セットの取り出しに必要な時間を決定します。

ロケーション　*samples-dir¥SQLAnywhere¥PerformanceFetch*

odbcfet

機能　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。

ロケーション　*samples-dir¥SQLAnywhere¥PerformanceFetch*

Copyright © 2007, iAnywhere Solutions, Inc.

instest

機能　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。

ロケーション　*samples-dir¥SQLAnywhere¥PerformanceInsert*

trantest

機能　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。

ロケーション　*samples-dir¥SQLAnywhere¥PerformanceTransaction*

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ます (たとえば複数のテーブルで繰り返されているカラム)。この場合は、テーブルを正規化できます。

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、会社の従業員が多数のオフィスで働いているとします。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。

ただし、正規化を過信しないように注意する必要があります。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。

データ正規化の詳細については、「手順 3：データを正規化する」 15 ページを参照してください。

テーブル内のカラムの順序の確認

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、ローの終わりのカラムにアクセスするために、SQL Anywhere は、ロー内で手前にあるカラムをスキップする必要があります。プライマリ・キー・カラムは、常にローの始めに格納されます。このため、テーブル内で小さいカラムか頻繁にアクセスされるカラム、またはその両方に当てはまるカラムが、あまりアクセスされないカラムの前にくるようにテーブルを作成することが重要です。

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。

異なる物理データベース・ファイルを異なる物理デバイスに入れると、ほとんどの場合データベースのパフォーマンスは向上します。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、別のデバイスはログ・ファイルに書き込みができます。

Copyright © 2007, iAnywhere Solutions, Inc.

このようなパフォーマンスの向上を得るためには、各デバイスを独立させます。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 では次の 4 種類のファイルが使用されます。

1. データベース (*.db*)
2. トランザクション・ログ (*.log*)
3. トランザクション・ログ・ミラー (*.mlg*)
4. テンポラリ・ファイル (*.tmp*)

「**データベース・ファイル**」は、データベースの内容全体を保持します。1 つのファイルで 1 つのデータベースを構成する方法と、最高 12 の DB 領域を追加する方法があります。DB 領域は、同じデータベースの一部を入れる追加ファイルです。必要に応じて、データベース・ファイルと DB 領域のロケーションを選択します。

「**トランザクション・ログ・ファイル**」は、障害発生時にデータベースの情報をリカバリするのに必要です。さらにデータベースを保護するために、3 種類目のファイル「**トランザクション・ログ・ミラー・ファイル**」で、トランザクション・ログの重複コピーを管理できます。SQL Anywhere では、同じ情報が同時にこれらの各ファイルに書き込まれます。

ヒント
トランザクション・ログ・ミラー・ファイルを (使用する場合に) 物理的に別個のドライブに置くと、ディスク障害からデータベースを保護できます。また、ログ・ファイルとログ・ミラー・ファイルへの書き込み効率が高まるため、SQL Anywhere が高速になります。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・ファイルのロケーションを指定するには、トランザクション・ログ (dblog) ユーティリティ、または Sybase Central の [ログ・ファイル設定の変更] ウィザードを使用します。「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』と「トランザクション・ログの場所の変更」『SQL Anywhere サーバ - データベース管理』を参照してください。

SQL Anywhere で、ソートや UNION 処理など、オペレーション用キャッシュで使用できる領域よりも多くの領域が必要な場合は、「**テンポラリ・ファイル**」が使用されます。こうした領域が必要な場合、データベース・サーバは、通常はその領域を集中的に使用します。データベース全体のパフォーマンスは、テンポラリ・ファイルを含むデバイスの速度に大きく依存します。

ヒント
テンポラリ・ファイルが、データベース・ファイルのあるデバイスとは物理的に異なる高速デバイス上にあると、通常は SQL Anywhere の実行が高速になります。これは、テンポラリ・ファイルを使用する必要がある大部分のオペレーションは、データベースから多くの情報を取り出す必要があるためです。情報を 2 つの別々のディスクに置くと、オペレーションを同時に行うことができます。

テンポラリ・ファイルのロケーションは慎重に選択します。テンポラリ・ファイルのロケーションは、データベース・サーバの起動時に -dt サーバ・オプションを使用して指定できます (UNIX 上の共有メモリ接続を除くすべての接続において)。テンポラリ・ファイルのロケーションを指

Copyright © 2007, iAnywhere Solutions, Inc.

定せずにデータベース・サーバを起動した場合は、SQL Anywhere で以下の環境変数がこの順序のとおりにチェックされます。

1. SATMP
2. TMP
3. TMPDIR
4. TEMP

環境変数が定義されていない場合、テンポラリ・ファイルは、Windows の場合は現在のディレクトリ、UNIX の場合は */tmp* ディレクトリに作成されます。

コンピュータが十分な数の高速デバイスを持っている場合、これらのファイルをそれぞれ別のデバイスに入れてパフォーマンスを改善できます。データベースを複数の DB 領域に分割し、別のデバイスへの配置もできます。その場合は、テーブルを個別の DB 領域にグループ分けして、一般的なジョイン操作によって異なる DB 領域から情報を読み込めるようにします。

すべてのテーブルをシステム DB 領域以外のロケーションに作成した場合は、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ディスクに保存する場合に便利です。このようにするには、すべての CREATE TABLE 文を変更して 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。「default_dbspace オプション [データベース]」『SQL Anywhere サーバ - データベース管理』と「CREATE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。「-fc 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。

参照

- 「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)」279 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』

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。

データベースを再構築すると、データやスキーマを含むすべての情報がデータベースから削除され、同一の形式ですべての情報が戻されるので、ディスク・ドライブの断片化デフラグメンテー

ションと同じように、スペースが埋められてパフォーマンスが改善されます。また、再構築は、特定の設定を変更する機会にもなります。「データベースの再構築」 706 ページを参照してください。

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。これ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。

- 多数のテーブルで削除／更新／挿入アクティビティを大々的に行った場合は、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

データベース・ファイルが断片化しすぎていると、パフォーマンスに影響が出ます。これはディスク断片化であり、データベースが大きくなるにつれ、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・サーバのメッセージ・ウィンドウに次の情報が表示されます。

nnn フラグメントで構成されるデータベース・ファイル mydatabase.db

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。

詳細については、「データベース・レベルのプロパティ」 『SQL Anywhere サーバ - データベース管理』を参照してください。

ファイルのフラグメンテーション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。

Copyright © 2007, iAnywhere Solutions, Inc.

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。ローがこのような状態の場合、ページ・アクセスが増えるため、パフォーマンスが低下します。テーブルの断片化は、ファイルの断片化とは異なります。

断片化がパフォーマンスに与える影響は、状況によってさまざまです。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。これとは反対に、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。その結果、分割されたローが頻繁にアクセスされ、かつ余計な I/O のコストが削減されない場合は、パフォーマンスが重大な影響を受け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は削減されますが、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。この項で後述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

sa_table_fragmentation システム・プロシージャを使用すると、データベース・テーブルの断片化レベルに関する情報を取得できます。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、このシステム・プロシージャを 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プロシージャを実行し、ベースラインとなる結果を取得してください。次に、長期間にわたってプロシージャを定期的に実行し、プロシージャの出力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。この方法では、テーブルの断片化状態がパフォーマンスに影響のあるレベルになる割合を決定でき、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。

このプロシージャを実行するには、DBA 権限が必要です。次の文は、sa_table_fragmentation システム・プロシージャを呼び出します。

```
CALL sa_table_fragmentation( [ 'table-name' [, 'owner-name' ] ] );
```

「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』を参照してください。

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 文を使用すると、特定のテーブルの断片化を解除できます。テーブルを再編成しても、データベース・アクセスは中断されません。
- **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、クラスタード・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

- 「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』
- 「再構築ユーティリティ (rebuild)」『SQL Anywhere サーバ - データベース管理』

インデックスの断片化削減

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 が実行されると、断片化することがあります。このため、インデックスが頻繁にアクセスされ、キャッシュが小さいためにインデックスがすべて収まらないと、パフォーマンス低下を招く場合があります。

sa_index_density システム・プロシージャは、データベースのインデックスの断片化レベルに関する情報を提供します。このプロシージャを実行するには、DBA 権限が必要です。次の文は、sa_index_density システム・プロシージャを呼び出します。

```
CALL sa_index_density( [ 'table-name' [, 'owner-name' ] ] );
```

インデックスが極端に断片化されている場合は、REORGANIZE TABLE を実行します。また、インデックスを削除して再作成する方法もあります。ただし、インデックスがプライマリ・キーの場合は、外部キー・インデックスの削除と再作成も必要になります。

参照

- 「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』
- 「インデックスの編集」87 ページ

Copyright © 2007, iAnywhere Solutions, Inc.

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。

テーブル幅の縮小

カラム数の多いテーブルは、ワイド・テーブルと呼ばれます。テーブル内のカラム数が多いため個々のローのサイズがデータベース・ページ・サイズを超える場合、各ローは、複数のデータベース・ページに分割されます。ローが多数のページに分割されるほど、各ローの読み込みには時間がかかります。パフォーマンスが低下している場合に、カラム数の多いテーブルがある場合は、テーブルを正規化してカラム数を減らすことを検討します。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。

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。「LazyClose 接続パラメータ [LCLOSE]」 『SQL Anywhere サーバ - データベース管理』と「PrefetchOnOpen 接続パラメータ」 『SQL Anywhere サーバ - データベース管理』を参照してください。

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。「ユーザ定義関数の概要」 787 ページを参照してください。

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な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と、パフォーマンスを改善できます。「トリガの概要」 790 ページを参照してください。

クエリ結果の効果的なソート

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。

ソートの詳細については、「ORDER BY 句：クエリ結果のソート」 329 ページまたは 「GROUP BY 句：クエリ結果のグループへの編成」 321 ページを参照してください。

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。「カーソル・タイプの選択」 『SQL Anywhere サーバ - プログラミング』を参照してください。

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、または削除されて以来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、統計をリフレッシュしてください。

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。

詳細については、「明示的な選択性推定」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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、各コマンドの最後に 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、コマンド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、コマンドの各論理グ

Copyright © 2007, iAnywhere Solutions, Inc.

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・ファイルを使用しないでオートコミット・モードを使用すると、特にコストが高くなります。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。

オートコミットの詳細については、「オートコミットまたは手動コミット・モードの設定」『SQL Anywhere サーバ - プログラミング』を参照してください。

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。どのページ・サイズを選択しても利点と欠点があります。

ページが小さいと、保持する情報が少なくなり、特にページの半分をわずかに超えるローを挿入する場合などは領域を効率良く使用できません。しかし、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 を少ないリソースで実行できます。特に、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しなければならない場合に便利です。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。

一方、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。また、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。

ただし、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。メモリやディスク領域を増設した効果がパフォーマンスに現れたことを確認してから、16 KB または 32 KB のページ・サイズを使用してください。

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 (そして適切な) ページ・サイズの選択が特に重要です。

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。「CREATE DATABASE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

大きいページ・サイズの詳細については、「最大ページ・サイズの設定」『SQL Anywhere サーバ - データベース管理』を参照してください。

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、分散読み込みが使用されません。

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 または string 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、経済的なデータ型をできるだけ選択してください。「SQL データ型」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。「CREATE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』と「ALTER TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

バルク・オペレーションのパフォーマンスの向上については、「バルク・オペレーションのパフォーマンスの側面」 682 ページを参照してください。

インデックスの有効な使用

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。インデックスは、アクセスを高速化します。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人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、たいてい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ます。インデックスの作成と管理の詳細については、「インデックスの編集」 87 ページを参照してください。

インデックスの使用

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する必要があり、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。「インデックス・コンサルタント」 210 ページを参照してください。

クラスタード・インデックス

クラスタード・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。「インデックス」 592 ページと「クラスタード・インデックスの使用」 88 ページを参照してください。

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、1,000,000 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・キーと外部キーの詳細については、「テーブル間の関係」『SQL Anywhere 10 - 紹介』を参照してください。

プライマリ・キーを使ったクエリのパフォーマンス改善

次は、プライマリ・キーを使用してパフォーマンスを改善する文です。

```
SELECT *
FROM Employees
WHERE EmployeeID = 390;
```

[プラン] タブの情報

[結果] ウィンドウ枠の [プラン] タブには、次の情報が表示されます。

```
Employees <Employees>
```

[プラン] タブの PLAN 記述のカッコ内の名前がテーブル名と同じである場合は、パフォーマンス改善のためにそのテーブルのプライマリ・キーが使われています。

外部キーを使ったクエリのパフォーマンス改善

次に示すクエリは、顧客 ID 113 を持つ顧客からの注文をリストします。

```
SELECT *
FROM SalesOrders
WHERE CustomerID = 113;
```

[プラン] タブの情報

[結果] ウィンドウ枠の [プラン] タブには、次の情報が表示されます。

```
SalesOrders <FK_CustomerID_ID>
```

FK_CustomerID_ID は、SalesOrders テーブルが Customers テーブルに対して持つ外部キーです。

 Copyright © 2007, iAnywhere Solutions, Inc.

別個のプライマリ・キー・インデックスと外部キー・インデックス

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。

パフォーマンス向上へのキャッシュの使用

データベース・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・サーバがディスクからデータを読み込まなければならない回数が減ります。ディスクからデータを読み込むオペレーションは速度が遅いので、使用できるキャッシュの容量がパフォーマンスを決める重要な要因になることがよくあります。

-c オプションを指定して、データベースの開始時にデータベース・サーバのコマンド・ラインでデータベース・キャッシュのサイズを制御できます。

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。また、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( 'CacheSize' );
```

参照

- 「-c サーバ・オプション」『SQL Anywhere サーバ - データベース管理』
- 「-ca サーバ・オプション」『SQL Anywhere サーバ - データベース管理』
- 「-ch サーバ・オプション」『SQL Anywhere サーバ - データベース管理』
- 「-cl サーバ・オプション」『SQL Anywhere サーバ - データベース管理』

キャッシュが使用するメモリの制限

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**　初期キャッシュ・サイズを変更するには、データベース・サーバの -c オプションを指定します。デフォルトの値は次のとおりです。
  - **Windows CE**　式は次のとおりです。

    ```
    max( 600 KB, min( dbsize, physical-memory ) );
    ```

    dbsize は起動したデータベース・ファイルのトータル・サイズです。*physical-memory* は、コンピュータの物理メモリの 25% です。
  - **Windows**　式は次のとおりです。

    ```
    max( 2 MB, min( dbsize, physical-memory ) );
    ```

    dbsize は起動したデータベース・ファイルのトータル・サイズです。*physical-memory* は、コンピュータの物理メモリの 25% です。

    Windows で AWE キャッシュを使用している場合、式は次のとおりです。

min(100% of available memory-128MB, *dbsize*);

この値が 2 MB より小さい場合、AWE キャッシュは使用されていません。

AWE キャッシュについては、「-cw 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。

- **NetWare**

  max(8 MB, min(*dbsize*, *physical-memory*));

  dbsize は起動したデータベース・ファイルのトータル・サイズです。*physical-memory* は、コンピュータの物理メモリの 25% です。

- **UNIX**　最小で 8 MB です。

  UNIX の初期キャッシュ・サイズについては、「動的キャッシュ・サイズ決定 (UNIX)」 276 ページを参照してください。

- **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いています。Windows CE 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、これは通常、物理メモリ量の合計の約 90% で、256 MB 以下です。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 CE 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 CE では、デフォルトの最小キャッシュ・サイズは 600 KB です。

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。

- **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。
- **CurrentCacheSize**　現在のキャッシュ・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。

サーバ・プロパティの値の取得については、「サーバ・レベルのプロパティ」 『SQL Anywhere サーバ - データベース管理』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

参照

- 「-c サーバ・オプション」『SQL Anywhere サーバ - データベース管理』
- 「-ca サーバ・オプション」『SQL Anywhere サーバ - データベース管理』
- 「-ch サーバ・オプション」『SQL Anywhere サーバ - データベース管理』
- 「-cl 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 は、データベース・キャッシュの自動サイズ変更機能を提供します。この機能は、オペレーティング・システムによって異なります。Windows と UNIX の各オペレーティング・システムでは、キャッシュは増大したり縮小したりします。詳細は、次の項で説明します。

完全な「**動的キャッシュ・サイズ決定**」によって、不適切なメモリ割り付けによるデータベース・サーバのパフォーマンスへの影響を防止できます。データベース・サーバの効率がキャッシュ・サイズを拡大することによって向上する場合は、使用可能なメモリがあるかぎりキャッシュ・サイズが拡大し、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小さくなります。この処理によって、データベース・サーバはシステム上の他のアプリケーションに過度の影響を与えないで済みます。動的キャッシュ・サイズ決定の効果は、当然、システムで使用できる物理メモリによって限られます。

通常、動的キャッシュ・サイズ決定は、約 1 分間隔でキャッシュの必要量を推定します。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、5 秒間隔で 30 秒間統計がサンプリングされ、キャッシュのサイズが変更されます。最初の 30 秒間が経過すると、サンプリング率は 1 分間隔に戻ります。データベースの起動後、またはサンプリング率の上昇をもたらした最後のファイル増加以降に、ファイルが 8 分の 1 増加した場合、大幅な増加が生じたと見なされます。サンプリング率の変化により、データベースが動的に起動された場合、または大量のデータが挿入された場合に、ただちにキャッシュ・サイズが変更され、パフォーマンスがさらに向上します。

動的キャッシュ・サイズ決定は、さまざまな状況でデータベース・キャッシュを明示的に設定する必要をなくし、SQL Anywhere をより使いやすくします。

Novell NetWare では、動的にキャッシュ・サイズが変更されません。Address Windowing Extensions (AWE) キャッシュが使用されている場合、動的キャッシュ・サイズ決定は無効になっています。Windows CE では AWE キャッシュを使用できません。

AWE キャッシュの詳細については、「-cw 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。

動的キャッシュ・サイズ決定 (Windows)

Windows と Windows CE 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。

Windows では、データベース・サーバの起動時に -cw コマンド・ライン・オプションを指定すると、Address Windowing Extensions (AWE) を使用して大きなキャッシュ・サイズをサポートできます。AWE キャッシュでは、動的なキャッシュ・サイズの変更はサポートされていません。Windows CE では、AWE キャッシュをサポートしていません。「-cw 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。

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、たいていの UNIX オペレーティング・システムにありますが、ないものもあります。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 より大きい場合は、データベース・サーバが「**メモリ不足**」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。「-c 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。

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・サーバが「**メモリ不足**」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

 Copyright © 2007, iAnywhere Solutions, Inc.

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・サイズを設定してください。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。詳細については、「-ch 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。

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。「-cl 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。

キャッシュ・サイズのモニタリング

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

注意
Windows パフォーマンス・モニタは、Windows に付属します。

これらのプロパティの詳細については、「サーバ・レベルのプロパティ」『SQL Anywhere サーバ - データベース管理』を参照してください。

パフォーマンスのモニタリングについては、「データベースのパフォーマンスのモニタリング」243 ページを参照してください。

キャッシュ・ウォーミングの使用

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・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

Copyright © 2007, iAnywhere Solutions, Inc.

シュ・ウォーミングにより、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、パフォーマンスを向上させることができます。

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、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)、収集速度が最小閾値を下回るまで、またはデータベースが停止するまで、追跡し続けます。データベース・サーバが収集最大ページ数と収集閾値を制御することに注意してください。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。

キャッシュ・ウォーミングに使用されるデータベース・サーバのオプションの詳細については、「-cc サーバ・オプション」『SQL Anywhere サーバ - データベース管理』、「-cr サーバ・オプション」『SQL Anywhere サーバ - データベース管理』、「-cv 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。

圧縮機能の使用

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。

圧縮閾値を指定すると、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。圧縮閾値の最適値は、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。

Copyright © 2007, iAnywhere Solutions, Inc.

参照

- 「パフォーマンス改善のための通信圧縮設定の調整」『SQL Anywhere サーバ - データベース管理』
- 「Compress 接続パラメータ [COMP]」『SQL Anywhere サーバ - データベース管理』
- 「CompressionThreshold 接続パラメータ [COMPTH]」『SQL Anywhere サーバ - データベース管理』

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

- 「データベース検証時のパフォーマンスの改善」『SQL Anywhere サーバ - データベース管理』
- 「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』

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・テーブルを使用した方が代替方式よりもコストが小さいと SQL Anywhere が判断すると、ワーク・テーブルが使用されます。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。デフォルトはすべてのローです。optimization_goal が最初のロー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。すべてのロー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 設定の詳細については、「optimization_goal オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

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 がすべてのロー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。最適化ゴールがすべてのローに設定されている場合、SQL Anywhere は低コストの方式を選択します。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。

Copyright © 2007, iAnywhere Solutions, Inc.

- 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移植されます。
- 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 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、コマンドを書き直して、競合とワーク・テーブルが作成されるのを防げます。これは、常に可能とはかぎりません。

Copyright © 2007, iAnywhere Solutions, Inc.

パート III. データのクエリと変更

パート III では、データのクエリ方法と変更方法、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、SQL Anywhere の OLAP 機能の利用方法の詳細についても説明します。

第 7 章

クエリ：テーブルからのデータの選択

目次

クエリの概要 284
SELECT リスト：カラムの指定 287
FROM 句：テーブルの指定 295
WHERE 句：ローの指定 296
ORDER BY 句：結果の順序付け 308
データの要約 311

クエリの概要

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、また 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できます。

クエリと SELECT 文

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に配信されます。クライアント・アプリケーションは、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 文は、複数の部分、つまり「**句**」を含めることができます。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 句には取得したいカラムを指定します。SELECT 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章では説明しません。

  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 (抽出テーブル)、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。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。
- 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)。

  GROUP BY 句、HAVING 句、ORDER BY 句については、「クエリ結果の要約、グループ化、ソート」315 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

これらの句の大半はオプションですが、SELECT 文に記述するときは、正しい順序で記述してください。

SELECT 文の構文の詳細については、「SELECT 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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、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、識別子 (テーブル名、カラム名など) 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。

データベースの作成については、「データベースの作成」 31 ページまたは「初期化ユーティリティ (dbinit)」『SQL Anywhere サーバ - データベース管理』を参照してください。

大文字と小文字の区別の詳細については、「大文字と小文字の区別」 622 ページを参照してください。

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'
```

この章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。

この章の後続の項では、SELECT 文の構文をもっと詳しく分析します。

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

SELECT リスト：カラムの指定

select リスト

select リストは、通常はカンマで間を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省略形として 1 つのアスタリスク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。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、集合関数も記述できます。集合関数については、「クエリ結果の要約、グループ化、ソート」 315 ページで説明します。

式の詳細については、「式」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

以下の項では、select リストで使用できる、さまざまな種類の式を示します。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
…	…	…

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必要な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。

結果セットのカラムのリストは、「**select リスト**」と呼ばれます。このリストはカンマで区切られています。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。この方法でカラムを制限することを、「**射影**」と呼ぶことがあります。

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
```


Copyright © 2007, iAnywhere Solutions, Inc.

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。

ジョインの使用の詳細については、「ジョイン：複数テーブルからのデータ検索」 341 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

クエリ結果にあるカラム名の変更

クエリ結果は一連のカラムで構成されます。デフォルトでは、各カラムの見出しは select リストに提供されている式です。

クエリ結果が表示されると、各カラムのデフォルトの見出しは、そのカラムの作成時に与えられた名前になります。別のカラム見出し、すなわち「**エイリアス**」を指定するには、次の方法があります。

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
…	…

エイリアスでのスペースとキーワードの使用

DepartmentID の代わりに使用されるエイリアス integer は、識別子であるため二重引用符で囲みます。キーワード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
```

Adaptive Server Enterprise との互換性を確保する場合は、30 バイト以下の、引用符で囲んだエイリアスを使用します。

クエリ結果の文字列

これまで見てきた 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した。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。

Copyright © 2007, iAnywhere Solutions, Inc.

```
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 リストの値の計算

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

Name	Sell before reorder
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
…	…

算術演算子の優先度

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;
```

誤解を避けるために、カッコを使用することをおすすめします。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、こちらの方がわかりやすいと言えます。

```
SELECT Name, ( Quantity * UnitPrice ) - 5
FROM Products;
```

演算子の優先度の詳細については、「演算子の優先度」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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/2003 準拠) 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。

 Copyright © 2007, iAnywhere Solutions, Inc.

次の例は、select リストでの文字列連結演算子の使用法を示しています。

```
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。SQL 関数については、「SQL 関数」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

  演算子の完全なリストについては、「演算子」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **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;
  ```

Copyright © 2007, iAnywhere Solutions, Inc.

ID	UCASE(Products.name)
300	TEE SHIRT
301	TEE SHIRT
302	TEE SHIRT
400	BASEBALL CAP
…	…

関数の完全なリストについては、「アルファベット順の関数リスト」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

参照

- 「不要な DISTINCT 条件の排除」 513 ページ

 Copyright © 2007, iAnywhere Solutions, Inc.

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 (抽出テーブル) へのジョインを記述できます。これらの機能については、「ジョイン：複数テーブルからのデータ検索」 341 ページを参照してください。

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、または現在の接続のユーザ ID が所属するグループ名と異なるユーザ ID によって所有されている場合だけです。

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。

FROM 句の詳細については、「FROM 句」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。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。テーブルやビューからのクエリに加えて、抽出テーブル (実体は SELECT 文) や結果セットを返すストアド・プロシージャも使用できます。

たとえば、次のクエリはストアド・プロシージャの結果セットを操作します。

```
SELECT *
FROM ShowCustomerProducts( 149 );
```

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。

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。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 (修飾または述部とも呼ばれる) 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;
```

- **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%';
```

探索条件の詳細については、「探索条件」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

WHERE 句での比較演算子の使用

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*

比較演算子の詳細については、「比較演算子」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。式を構成する各要素の説明については、「式」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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"
  ```

  照合順は Sybase Central からも確認できます。データベース・プロパティ・シートの [詳細情報] タブにあります。

- **後続ブランク**　データベースの作成時に、比較を目的とした場合に後続ブランクを無視するかどうかを指定します。

  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。Adaptive Server Enterprise データベースの場合、後続ブランクはデフォルトでは無視されます。

- **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。Adaptive Server Enterprise データベースの場合は、これがデフォルトの動作です。

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;
```

WHERE 句での範囲の使用

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。

♦ **$10 以上 $15 以下の価格の製品をすべてリストするには、次の手順に従います。**

・ 次のクエリを入力します。

```
SELECT Name, UnitPrice
  FROM Products
  WHERE UnitPrice BETWEEN 10 AND 15;
```

Name	UnitPrice
Tee Shirt	14
Tee Shirt	14
Baseball Cap	10
Shorts	15

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。

♦ **$10 未満か、$15 を超える製品をすべてリストするには、次の手順に従います。**

・ 次のクエリを実行します。

```
SELECT Name, UnitPrice
  FROM Products
  WHERE UnitPrice NOT BETWEEN 10 AND 15;
```

Name	UnitPrice
Tee Shirt	9

Copyright © 2007, iAnywhere Solutions, Inc.

Name	UnitPrice
Baseball Cap	9
Visor	7
Visor	7
…	…

WHERE 句でのリストの使用

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 句での文字列のマッチング

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フォームは、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**　セットは、任意の順序での値の個別の集合で構成されます。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 ー 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。クエリのフォームは次のとおりです。

```
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

Copyright © 2007, iAnywhere Solutions, Inc.

マッチ式	説明	検索結果
'[M-Z]owell'	M ー 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%';
```

文字列リテラルの詳細については、「バイナリ・リテラル」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

日付値と時刻値での LIKE の使用

文字データだけではなく、日時フィールドでも LIKE を使用できます。日付値や時刻値で LIKE を使用すると、日付は標準的な DATETIME フォーマットに変換され、次に VARCHAR に変換されます。

DATETIME 値を検索するときに LIKE を使用する場合の特徴は、日付エントリと時刻エントリにはさまざまな日付部分があるため、検索に成功するには等号テストを慎重に書き込まなければならないということです。

たとえは、カラム arrival_time に値 9:20 と現在の日付を入力するときに、次のように句を指定するとします。

```
WHERE arrival_time = '9:20'
```

このエントリには、時刻だけではなく日付も格納されているため、値の検索に失敗します。しかし、次の句なら 9:20 という値を検出します。

```
WHERE arrival_time LIKE '%09:20%'
```

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。

詳細については、「LIKE 探索条件」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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)、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。

♦ 現在のユーザ ID の quoted_identifier オプションを OFF に設定するには、次の手順に従います。

・ 次のコマンドを入力します。

```
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なります (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?"'
```


Copyright © 2007, iAnywhere Solutions, Inc.

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。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、create table 文に指定したように NULL 値が許可されたカラムに、次の 2 つの方法で入力できます。

- **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、NULL が入力されます。
- **明示的な入力**　NULL (引用符なし) と入力することで、NULL 値を明示的に入力できます。

  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 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 が検索された場合

NULL が検索されると、Interactive SQL によるクエリ結果表示の適切な位置に (null) が表示されます。

```
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)

NULL のためのカラムのテスト

探索条件で IS NULL 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。次の条件を含む句は、

```
WHERE column1 = column2
```

カラムが NULL を含んでいるときにはローを返しません。

次のパターンの NULL や NOT NULL もあります。

```
WHERE column_name IS [NOT] NULL
```

次に例を示します。

```
WHERE advance < $5000
OR advance IS NULL
```

詳細については、「NULL 値」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

NULL のプロパティ

次の項目で、NULL 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 の反対 ("not false") は true なので、FALSE と UNKNOWN の間には大きな論理的相違があります。次に例を示します。

  ```
  1 = 2
  ```

  上の例は false と評価されます。

Copyright © 2007, iAnywhere Solutions, Inc.

```
1 != 2
```

上の例は、false の反対なので true と評価されます。しかし、"not unknown" は依然として不定です。比較の中に null 値がある場合は、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。

- **複数の NULL 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 を 1 つの特定の値と置き換えます。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 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。たとえば次の文では、column1 が NULL であれば NULL と評価されます。

  ```
  1 + column1
  ```

- **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';
  ```

論理演算子による複数の条件の接続

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。

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。Beth Glassmann のローは検出しません。

```
SELECT *
  FROM Contacts
  WHERE GivenName = 'Beth'
    AND Surname = 'Purcell';
```

OR の使用

OR 演算子も 2 つ以上の条件を結合しますが、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';
```

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。次に例を示します。

```
SELECT *
  FROM Customers
  WHERE ( City = 'Newmarket'
     OR City = 'Forest Hill' )
  AND State = 'MN';
```

探索条件による日付の比較

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。

♦ 生年月日が 1964 年 3 月 13 日より前の従業員をすべてリストするには、次の手順に従います。

- Interactive SQL で次のクエリを実行します。

```
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
…	…

Copyright © 2007, iAnywhere Solutions, Inc.

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。次の例は、すべて SQL Anywhere で使用できます。

```
'March 13, 1964'
'1964/03/13'
'1964-03-13'
```

  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チューニングできます。*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。

  クエリで許可される日付フォーマットを制御する方法については、「date_order オプション [互換性]」『SQL Anywhere サーバ - データベース管理』と「オプションの設定」『SQL Anywhere サーバ - データベース管理』を参照してください。

- **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。

  使用できる比較演算子の完全なリストについては、「比較演算子」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。社内のどの従業員が Brown のような発音の名前を持っているでしょうか。

◆ **Brown のように聞こえる姓を持つ従業員をリストするには、次の手順に従います。**

・ Interactive SQL で次のクエリを実行します。

```
SELECT Surname, GivenName
  FROM Employees
  WHERE SOUNDEX( Surname ) = SOUNDEX( 'Brown' );
```

Surname	GivenName
Braun	Jane

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。

詳細については、「SOUNDEX 関数 [文字列]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。この場合も、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。

◆ **製品をアルファベット順にリストするには、次の手順に従います。**

・ Interactive SQL で次のクエリを実行します。

```
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
…	…	…

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

Copyright © 2007, iAnywhere Solutions, Inc.

ID	Quantity
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
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。

インデックスの使用による ORDER BY のパフォーマンス改善

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。

詳細については、「インデックスの使用」 271 ページを参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

データの要約

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。

集合関数の概要

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 句がないと、集合関数は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。

♦ 社内の従業員数をリストするには、次の手順に従います。

・ 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をリストするには、次の手順に従います。

・ 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

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」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

グループ分けされたデータに対する集合関数の適用

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。

例

♦ **営業担当者と各担当者が受注した件数をリストするには、次の手順に従います。**

・ Interactive SQL で次のクエリを実行します。

```
SELECT SalesRepresentative, count( * )
  FROM SalesOrders
  GROUP BY SalesRepresentative
  ORDER BY SalesRepresentative;
```

SalesRepresentative	count(*)
129	57
195	50
299	114
467	56
…	…

GROUP BY 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。

詳細については、「GROUP BY 句：クエリ結果のグループへの編成」 321 ページを参照してください。

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;
```

 Copyright © 2007, iAnywhere Solutions, Inc.

指定された ID を持つ従業員についての結果ローで姓がすべて同じであることを SQL Anywhere は保証できないため、「select リストの中の 'Surname' に対する関数またはカラムの参照は GROUP BY 句の中になければなりません。」というエラーがレポートされます。

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。

グループの制限

WHERE 句を使用して結果セット内のローを制限する方法については、すでに説明しました。グループ内のローを制限するには、HAVING 句を使用します。

♦ **受注数が 55 を超えるすべての営業担当者をリストするには、次の手順に従います。**

- Interactive SQL で次のクエリを実行します。

```
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

HAVING 句 の詳細については、「HAVING 句：データ・グループの選択」 327 ページを参照してください。

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。必要な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。

 Copyright © 2007, iAnywhere Solutions, Inc.

第 8 章

クエリ結果の要約、グループ化、ソート

目次

集合関数を使用したクエリ結果の要約 316
GROUP BY 句 : クエリ結果のグループへの編成 321
GROUP BY 句の概要 323
HAVING 句 : データ・グループの選択 327
ORDER BY 句 : クエリ結果のソート 329
UNION、INTERSECT、EXCEPT を使用した、クエリ結果に対する集合操作の実行 332
標準と互換性 338

集合関数を使用したクエリ結果の要約

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。

テーブル中のすべてのロー、WHERE 句によって指定されるテーブルのサブセット、またはテーブル中のローが 1 つ以上のグループに、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。

使用でき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。

集合関数の完全なリストについては、「集合関数」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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
```

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。

Copyright © 2007, iAnywhere Solutions, Inc.

集合関数を使用できる場所

前述の例のように、集合関数は select リストで使用するか、GROUP BY 句を含む SELECT 文の HAVING 句で使用します。

HAVING 句 の詳細については、「HAVING 句 : データ・グループの選択」 327 ページを参照してください。

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 文には、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。

SELECT 文に WHERE 句が入っているが GROUP BY 句は入っていない場合、集合関数は、WHERE 句が指定するローのサブセットに対して単一の値を生成します。

GROUP BY 句がない SELECT 文で集合関数を使用すると、必ず単一の値が生成されます。これは、その関数がテーブル中のすべてのローに作用するか、WHERE 句が定義したローのサブセットに作用するかに関係なく、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 文で 2 つ以上のスカラ集合関数を作成できます。

集合関数と外部参照

SQL Anywhere は、サブクエリでの集合関数の使用法を明確に規定した SQL/2003 標準に準拠しています。この変更は、Adaptive Server Anywhere の以前のバージョン用に記述された文の動作に影響を与えます。以前は正しかったクエリでエラー・メッセージが生成され、結果セットが変わる可能性があります。

集合関数がサブクエリに使用され、集合関数の参照先カラムが外部参照の場合は、集合関数全体が外部参照として扱われるようになりました。これは、集合関数がサブクエリ内ではなく外部ブロック内で計算されるようになり、サブクエリ内の定数となることを意味しています。

サブクエリに含まれる外部参照の集合関数の使用には、次の制限が適用されるようになりました。

- 外部参照の集合関数を指定できるのは、SELECT リストまたは HAVING 句にあるサブクエリの中だけです。また、これらの句は、外部ブロックの隣になくてはなりません。
- 外部参照の集合関数に指定できるのは、1 つの外部カラム参照だけです。
- ローカル・カラム参照と外部カラム参照を、同じ集合関数に混在させることはできません。

新しい標準に関連する問題は、ローカル参照しか含まないように集合関数を書き換えると回避することができます。たとえば、サブクエリ (SELECT MAX(S.y + R.y) FROM S) がローカル・カラム参照 (S.y) と外部カラム参照 (R.y) の両方で構成されていると、不正になります。このサブクエリは (SELECT MAX(S.y) + R.y FROM S) に書き換えることができます。書き換えると、集合関数はローカル・カラム参照だけを持つことになります。外部参照の集合関数が SELECT や HAVING 以外の句に指定されている場合にも、同様の書き換えを使用できます。

例

次のクエリの場合、Adaptive Server Anywhere バージョン 7 では次に示す結果が生成されました。

Copyright © 2007, iAnywhere Solutions, Inc.

```
SELECT Name,
    ( SELECT SUM( p.Quantity )
       FROM SalesOrderItems )
  FROM Products p;
```

name	sum(p.Quantity)
Tee Shirt	30,716
Tee Shirt	59,238

最新のバージョンで同じクエリを実行すると、エラー・メッセージ「**SQL Anywhere エラー -149: 'name' に対する関数またはカラムの参照も GROUP BY 句に記述する必要があります。**」が生成されます。この文が有効ではなくなったのは、外部参照の集合関数 sum(p.Quantity) が外部ブロック内で計算されるようになったためです。最新のバージョンでは、このクエリはセマンティック上は次のクエリと同じです (Z が結果セットに表示されないことを除く)。

```
SELECT Name,
     SUM( p.Quantity ) AS Z,
     ( SELECT Z
        FROM SalesOrderItems )
  FROM Products p;
```

外部ブロックが集合関数を計算するようになったため、外部ブロックはグループ化したクエリとして扱われます。また、カラム名を SELECT リストに表示するには、GROUP BY 句に指定する必要があります。

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(*) の使用

COUNT(*) 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。COUNT(*) 関数は、1 つのテーブルのローの総数を検出します。次の文は、従業員の総数を検出します。

```
SELECT COUNT(*)
  FROM Employees;
```

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。

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

```
SELECT count(*), AVG( UnitPrice )
  FROM Products
  WHERE UnitPrice > 10;
```

count(*)	AVG(Products.UnitPrice)
5	18.2

DISTINCT を伴う集合関数の使用

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、次のように入力します。

```
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 name	last name
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

Copyright © 2007, iAnywhere Solutions, Inc.

```
SELECT AVG( UnitPrice )
  FROM Products
  WHERE UnitPrice > 50;
```

AVG(Products.UnitPrice)
(NULL)

Copyright © 2007, iAnywhere Solutions, Inc.

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 つ以上のカラム名か、計算カラムの結果によってグループ化 (GROUP BY) することができます。

句の順序
GROUP BY 句は、常に HAVING 句より前に置いてください。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。

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。

集合関数に伴う GROUP BY の使用

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 文が生成する計算値 (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計算値 (スカラ集約値) 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;
```

Copyright © 2007, iAnywhere Solutions, Inc.

AVG(Products.UnitPrice)
13.3

Copyright © 2007, iAnywhere Solutions, Inc.

GROUP BY 句の概要

クエリに GROUP BY 句が含まれていると、どのクエリが有効でどのクエリが無効かを判断するのは困難です。この項では、クエリの結果と有効性をよりよく理解できるように、GROUP BY のあるクエリについて説明します。

GROUP BY のあるクエリを実行する方法

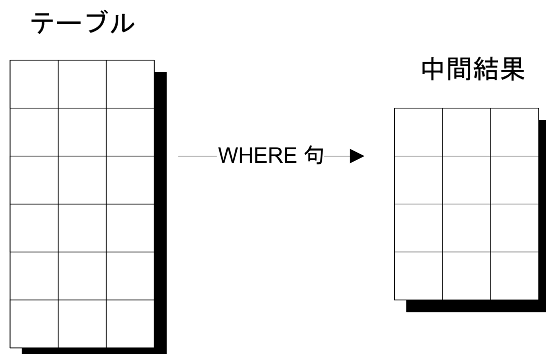
この項では、式や例で GROUP BY 句のサブ句 ROLLUP を使用します。ROLLUP 句 の詳細については、「ROLLUP と CUBE の使用」 419 ページを参照してください。

次のようなフォーム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*

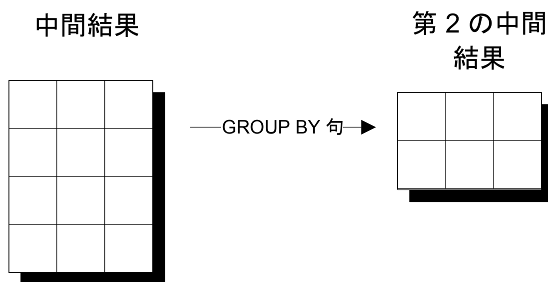
このクエリは、次のように実行されると考えられます。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

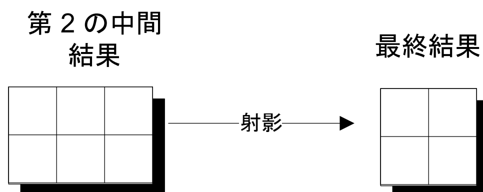


3. **ROLLUP 演算があれば適用する**　ROLLUP 演算の一部として計算された小計ローが、結果セットに追加されます。

   詳細については、「ROLLUP の使用」 420 ページを参照してください。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

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)
Tee Shirt	Small	9
Tee Shirt	Medium	14
Tee Shirt	One size fits all	14
Baseball Cap	One size fits all	9.5
…	…	…

select リストにない GROUP BY 句内のカラム

Adaptive Server Enterprise と SQL Anywhere の両方によってサポートされている SQL/92 標準に対する iAnywhere の拡張機能の 1 つは、select リストにない式を GROUP BY 句に許可することです。たとえば、次のクエリは、各都市の連絡先の数をリストします。

```
SELECT State, count( ID )
  FROM Contacts
  GROUP BY State, City;
```

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%'
```

Copyright © 2007, iAnywhere Solutions, Inc.

```
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。

Copyright © 2007, iAnywhere Solutions, Inc.

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が、HAVING 探索条件ではそれができるという点が異なります。したがって次の例は有効です。

```
HAVING AVG( UnitPrice ) > 20
```

しかし、次の例は無効です。

```
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

HAVING 句に集合関数を使用できる場合については、「集合関数を使用できる場所」 317 ページを参照してください。

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%';
```

Copyright © 2007, iAnywhere Solutions, Inc.

Name
Baseball Cap

HAVING における 2 つ以上の条件

HAVING 句では、2 つ以上の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

 Copyright © 2007, iAnywhere Solutions, Inc.

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
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

カラム名のかわりに、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。

クエリが返すロー数を明示的に制限する

FIRST キーワードまたは TOP キーワードを使用して、クエリの結果セットに含まれるロー数を制限できます。これらのキーワードは、ORDER BY 句を含むクエリで使用できます。

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、姓でソートした場合の最初の 5 人の従業員を返します。

```
SELECT TOP 5 *
  FROM Employees
  ORDER BY Surname;
```

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;
```

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

> **注意**
> START AT 値は 1 以上の値にしてください。TOP 値は、定数の場合に 1 以上の値にし、変数の場合に 0 以上にしてください。

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の実行

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。

- データが非正規化形式で格納されている場合、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。

  NULL と集合操作の詳細については、「集合演算子と NULL」 335 ページ を参照してください。

詳細については、「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』を参照してください。

UNION 文を使用してセットを結合する

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

EXCEPT と INTERSECT の使用

EXCEPT 文は、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 文は、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 文と同様に、EXCEPT と INTERSECT では ALL 修飾子を必要とします。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ることを防ぎます。

詳細については、「EXCEPT 文」 『SQL Anywhere サーバ - SQL リファレンス』と「INTERSECT 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

- **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。結果セット用に新しいカラム見出しを定義する場合は、次の例のように、最初のクエリの select リストで定義できます。

```
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 文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
```


Copyright © 2007, iAnywhere Solutions, Inc.

```
    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特に有効な結果の 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
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)

Copyright © 2007, iAnywhere Solutions, Inc.

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)

標準と互換性

この項では、SQL Anywhere の GROUP BY 句の、標準と互換性について説明します。

GROUP BY と SQL/2003 標準

GROUP BY について、SQL/2003 標準では次のように定めています。

- SELECT 句の式で使用されるカラムは、GROUP BY 句になければならない。そうでない場合、このカラムを使用する式は集合関数でなければならない。
- GROUP BY 式は、select リストからのカラム名だけを含むことができるが、ベクトル集合関数の引数としてのみ使用されるカラム名を含むことはできない。

ベクトル集合関数による標準的な GROUP BY の結果は、1 グループあたり、1 つの値を持つ 1 つのローを生成します。

SQL Anywhere と Adaptive Server Enterprise は、HAVING に select リストにも GROUP BY 句にもない集合関数を含むことができる拡張機能をサポートしています。

Adaptive Server Enterprise との互換性

Adaptive Server Enterprise は、SQL Anywhere ではサポートされていない、GROUP BY 句に対する拡張をサポートしています。拡張機能には次のようなものがあります。

- **select リスト中のグループ化されないカラム**　Adaptive Server Enterprise では、GROUP BY 句にないカラム名を select リストに入れることができます。たとえば、Adaptive Server Enterprise では次の文が有効です。

  ```
  SELECT Name, UnitPrice
    FROM Products
    GROUP BY Name;
  ```

  この構文は SQL Anywhere ではサポートされていません。

- **ネストされた集合関数**　スカラ集合関数の中にベクトル集合関数をネストする次のクエリは、Adaptive Server Enterprise では有効ですが、SQL Anywhere では無効です。

  ```
  SELECT MAX( AVG( UnitPrice ) )
    FROM Products
    GROUP BY Name;
  ```

- **GROUP BY と ALL**　SQL Anywhere では、GROUP BY 句に ALL を使用できません。
- **GROUP BY のない HAVING**　SQL Anywhere では、select 句と having 句内のすべての式が集合関数でないかぎり、GROUP BY 句のない HAVING は使用できません。たとえば、次のクエリは Adaptive Server Enterprise では有効ですが、SQL Anywhere ではサポートされていません。

Copyright © 2007, iAnywhere Solutions, Inc.

```
SELECT UnitPrice
  FROM Products
  HAVING COUNT(*) > 8;
```

ただし、MAX 関数と COUNT 関数は集合関数なので、次の文は SQL Anywhere で有効です。

```
SELECT MAX( UnitPrice )
  FROM Products
  HAVING COUNT(*) > 8;
```

- **HAVING の条件**　Adaptive Server Enterprise は、select リストにも GROUP BY 句にもない集合関数以外の関数を HAVING に入れることができる拡張機能をサポートしています。SQL Anywhere では、このタイプは集合関数だけが許可されます。
- **ORDER BY または GROUP BY を伴う DISTINCT**　Adaptive Server Enterprise では、select リストだけでなく、SELECT DISTINCT クエリにもないカラムを ORDER BY 句または GROUP BY 句の中で使用できます。そのため、SELECT DISTINCT 結果セットでは値が反復されることになります。SQL Anywhere は、この動作をサポートしていません。
- **複数の UNION の中のカラム名**　Adaptive Server Enterprise では、クエリの UNION 中の ORDER BY 句でカラムを使用できます。SQL Anywhere では、ORDER BY 句で整数を使用して、結果を順序付けするカラムをマーク付けする必要があります。

第 9 章

ジョイン：複数テーブルからのデータ検索

目次

概要 342
サンプル・データベース・スキーマ 344
ジョイン操作 346
ジョインの概要 347
明示的なジョイン条件 (ON 句) 352
クロス・ジョイン 356
内部ジョインと外部ジョイン 358
特殊なジョイン 365
ナチュラル・ジョイン 373
キー・ジョイン 377

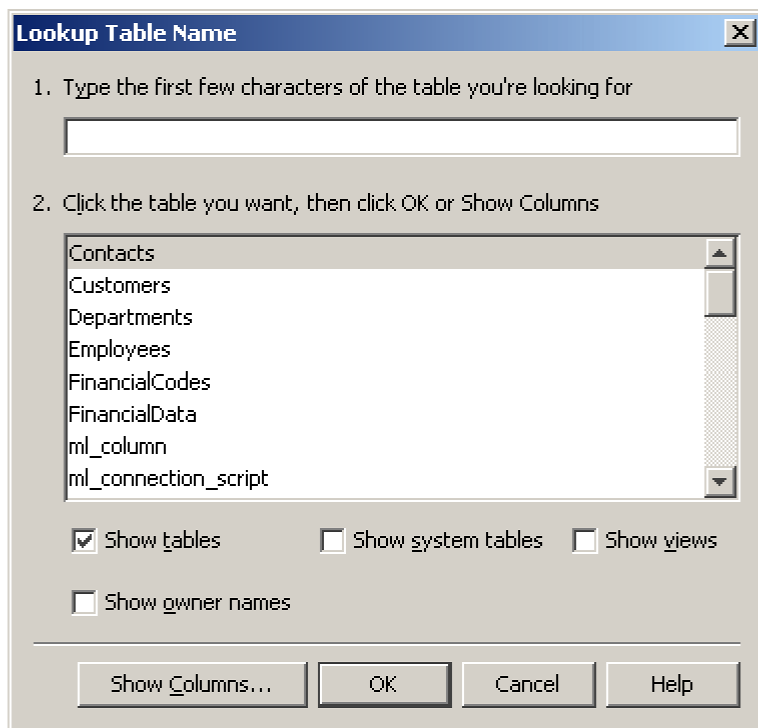
Copyright © 2007, iAnywhere Solutions, Inc.

概要

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 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。



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。[Esc] キーを押すとテーブル・リストに戻り、もう一度 [Esc] キーを押すと [SQL 文] ウィンドウ枠に戻ります。[Esc] キーの代わりに [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。

リストを終了するには、[Esc] キーを押します。

Copyright © 2007, iAnywhere Solutions, Inc.

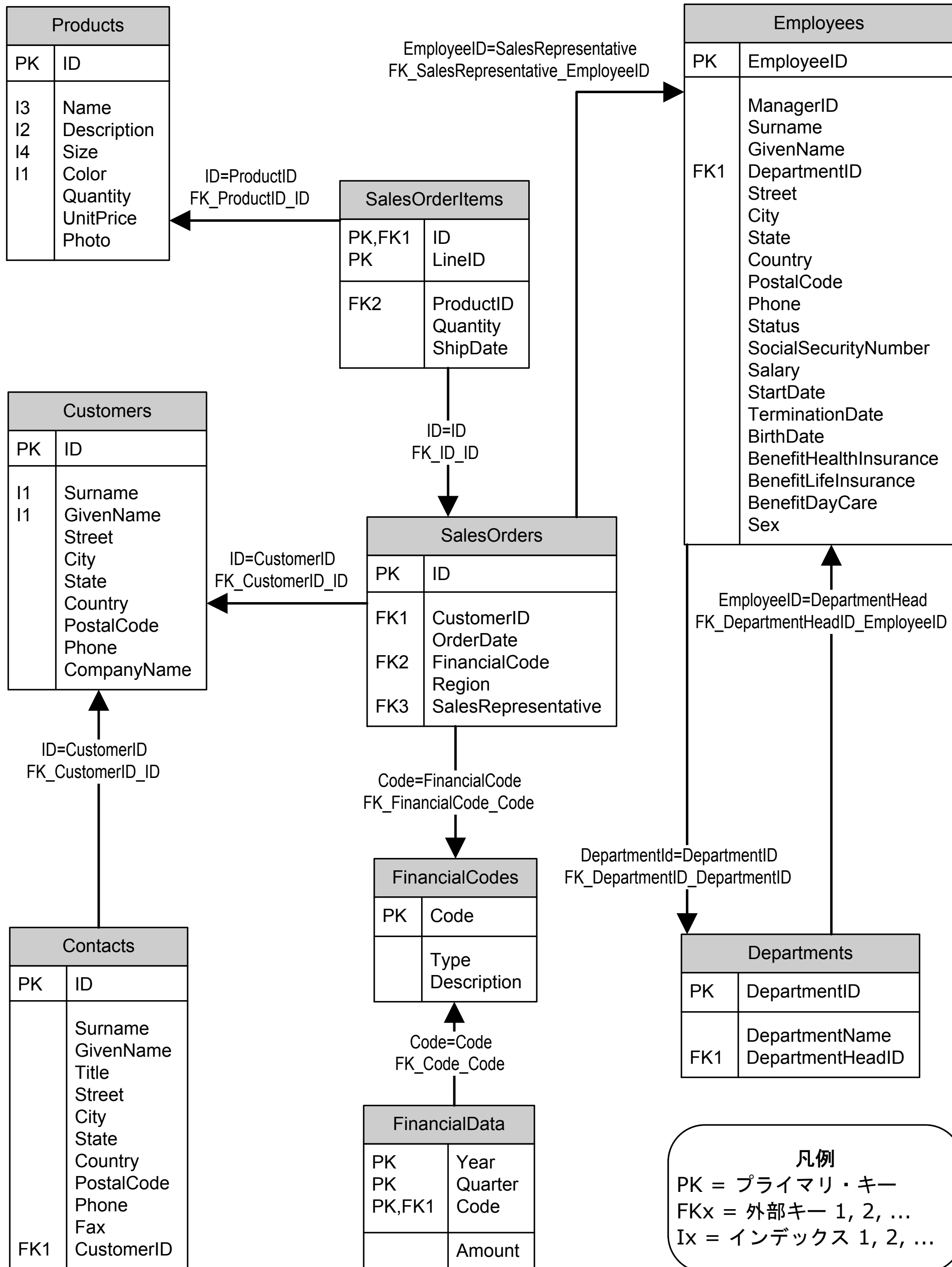
SQL Anywhere サンプル・データベース内のテーブルの詳細については、「チュートリアル：サンプル・データベースの使用」『SQL Anywhere サーバ - データベース管理』を参照してください。

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」 378 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.



Products
PK ID
I3 Name
I2 Description
I4 Size
I1 Color
Quantity
UnitPrice
Photo
ID=ProductID
FK_ProductID_ID
SalesOrderItems
PK,FK1 ID
PK LineID
FK2 ProductID
Quantity
ShipDate
EmployeeID=SalesRepresentative
FK_SalesRepresentative_EmployeeID
Employees
PK EmployeeID
ManagerID
Surname
GivenName
FK1 DepartmentID
Street
City
State
Country
PostalCode
Phone
Status
SocialSecurityNumber
Salary
StartDate
TerminationDate
BirthDate
BenefitHealthInsurance
BenefitLifeInsurance
BenefitDayCare
Sex
Customers
PK ID
I1 Surname
I1 GivenName
Street
City
State
Country
PostalCode
Phone
CompanyName
ID=ID
FK_ID_ID
ID=CustomerID
FK_CustomerID_ID
SalesOrders
PK ID
FK1 CustomerID
OrderDate
FK2 FinancialCode
Region
FK3 SalesRepresentative
EmployeeID=DepartmentHead
FK_DepartmentHeadID_EmployeeID
ID=CustomerID
FK_CustomerID_ID
Code=FinancialCode
FK_FinancialCode_Code
DepartmentId=DepartmentID
FK_DepartmentID_DepartmentID
FinancialCodes
PK Code
Type
Description
Contacts
PK ID
Surname
GivenName
Title
Street
City
State
Country
PostalCode
Phone
Fax
FK1 CustomerID
Departments
PK DepartmentID
DepartmentName
FK1 DepartmentHeadID
Code=Code
FK_Code_Code
FinancialData
PK Year
PK Quarter
PK,FK1 Code
Amount
凡例
PK = プライマリ・キー
FKx = 外部キー 1, 2, ...
Ix = インデックス 1, 2, ...

ジョイン操作

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。

さまざまなテーブルの情報を組み合わせて使用すれば、ほとんどの問い合わせに対する答えを取得できます。たとえば、「営業部を管理しているのは誰か」という問い合わせがあります。この人物の名前を見つけるには、Departments テーブルの情報から正しい人物を特定し、Employees テーブルでその名前を検索します。

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

ジョインの概要

「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。ここで説明した概念を以降の各項ではさらに詳しく説明します。

FROM 句

FROM 句を使用して、ジョインの対象となるベース・テーブル、テンポラリ・テーブル、ビュー、抽出テーブルを指定します。FROM 句は、SELECT 文または UPDATE 文で使用できます。次は、FROM 句の構文を簡略化したものです。

FROM *テーブル式 {{FROM:てーぶるしき}}*, …

文中の各項目を次に説明します。

table-expression:
table | *view* | *derived table* | *joined table* | **(** *table-expression*, … **)**

table or view:
[*userid*.] *table-or-view-name* [[**AS**] *correlation-name*]

derived table:
(*select-statement* **)** [**AS**] *correlation-name* [**(** *column-name*, … **)**]

joined table:
table-expression join-operator table-expression [**ON** *join-condition*]

join-operator:
[**KEY** | **NATURAL**] [*join-type*] **JOIN** | **CROSS JOIN**

join-type:
INNER | **FULL** [**OUTER**] | **LEFT** [**OUTER**] | **RIGHT** [**OUTER**]

注意

CROSS JOIN には ON 句を使用できません。

完全な構文については、「FROM 句」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ON 句を使った構文については、「探索条件」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

```
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、生成されたジョイン条件はテーブル間の外部キー関係に基づいています。ナチュラル・ジョインの場合には、名前が同じカラムに基づいています。

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*付けずに* JOIN キーワードを使用して ON 句内で同じジョイン条件を明示的に記述しても、同一の結果が得られます。

キー・ジョインまたはナチュラル・ジョインで 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。つまり、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。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－デフォルト)**　データベースにすでに構築されている外部キー関係に基づいて、ジョイン条件が自動的に生成されます。ジョイン・タイプを指定しないでキーワード JOIN を使用する場合や、ON 句がない場合は、キー・ジョインがデフォルトになります。
- **NATURAL JOIN (ナチュラル・ジョイン)**　名前が同じカラムに基づいて、ジョイン条件が自動的に生成されます。

Copyright © 2007, iAnywhere Solutions, Inc.

- **ON 句を使用したジョイン**　明示的ジョイン条件を指定します。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。

2 つのテーブルのジョイン

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	サイズ	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)。

Copyright © 2007, iAnywhere Solutions, Inc.

- 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)。

外部ジョインの計算方法については、「外部ジョイン」 358 ページを参照してください。

3 つ以上のテーブルのジョイン

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。したがって、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 にジョインされます。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。つまり、テーブル式 A JOIN B が C に結合されるという意味です。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。

外部ジョインの詳細については、「外部ジョイン」 358 ページを参照してください。

SQL Anywhere でテーブル式のキー・ジョインがどのように実行されるかについては、「テーブル式のキー・ジョイン」 381 ページを参照してください。

SQL Anywhere でテーブル式のナチュラル・ジョインがどのように実行されるかについては、「テーブル式のナチュラル・ジョイン」 375 ページを参照してください。

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。

ジョインでのデータ型変換の詳細については、「データ型間の比較」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

DELETE 文、UPDATE 文、INSERT 文でのジョインの使用

ジョインは、SELECT 文だけではなく DELETE 文、UPDATE 文、INSERT 文でも使用できます。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 のバージョン 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。「ansi_update_constraints オプション [互換性]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

ANSI 以外のジョイン

SQL Anywhere は、ISO/ANSI 規格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 =*)」 362 ページ
◆ 「ジョインで重複する相関名 (スター・ジョイン)」 366 ページ
◆ 「キー・ジョイン」 377 ページ
◆ 「ナチュラル・ジョイン」 373 ページ

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。

詳細については、「REWRITE 関数 [その他]」 『SQL Anywhere サーバ - SQL リファレンス』 を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

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;
```

参照先になるテーブル

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 (ここでは C JOIN D) ではありません。

ただし、ANSI/ISO 規格 SQL99 と Adaptive Server Anywhere 7.0 については、この規則は適用されません。つまり、テーブル式の間にカンマを使用すれば、ジョインの ON 条件は、構文中にその 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)
```

カンマの詳細については、「カンマ」 356 ページを参照してください。

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;
```

 Copyright © 2007, iAnywhere Solutions, Inc.

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。

詳細については、「明示的なジョイン条件 (ON 句)」 352 ページを参照してください。

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;
```

Copyright © 2007, iAnywhere Solutions, Inc.

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義です。

```
SELECT *
FROM A JOIN B
  ON A.x = B.y
  AND A.w = B.z;
```

キー・ジョインの詳細については、「キー・ジョイン」 377 ページを参照してください。

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) などの検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;
```

探索条件の詳細については、「探索条件」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ジョイン条件に対する WHERE 句の使用

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、外部ジョインの場合、つまり、WHERE 句を使用することで外部ジョインを内部ジョインに変換できる場合を除いて、結果にはなんら変化をもたらしません。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

WHERE 句と外部ジョインの詳細については、「外部ジョインとジョインの条件」 359 ページを参照してください。

- 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。

  キー・ジョインがデフォルトになるときの条件の詳細については、「キー・ジョインがデフォルトの場合」 377 ページを参照してください。

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。

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* × *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 CROSS JOIN B CROSS JOIN C
WHERE A.x = B.y;
```

と

```
SELECT *
FROM A, B, 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。

生成されたジョイン条件でカンマがどのように機能するかについては、「テーブル式のキー・ジョイン」 381 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

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。詳細については、「ジョインで重複する相関名 (スター・ジョイン)」 366 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

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。キーワード OUTER はオプションです。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。つまり、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 FROM 句を使用しても前述の例と同じ結果が得られます。

```
FROM Customers JOIN SalesOrders
```

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**」は、一方のテーブルではすべてのローを保護し、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 を入力します。「**左外部ジョイン**」は左側のテーブルのローをすべて保護し、「**右外部ジョイン**」は右側テーブルのローをすべて保護します。「**全外部ジョイン**」では、両方のテーブルのローがすべて保護されます。

Copyright © 2007, iAnywhere Solutions, Inc.

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。

Transact-SQL 構文を使用した外部ジョインの作成については、「Transact-SQL の外部ジョイン (*= or =*)」 362 ページ を参照してください。

例

たとえば次の文には、注文するかどうかにかかわらず、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、受注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。

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。通常は、WHERE 句を使って NULL 入力テーブルに制限を加えると、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。

検索条件の詳細については、「探索条件」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

複雑な外部ジョインについて

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。つまり、テーブル式 (A JOIN B) が C に結合されるという意味です。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。

Copyright © 2007, iAnywhere Solutions, Inc.

```
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。SQL Anywhere がこのタイプの結合のジョイン条件を生成する方法については、「カンマを含まないテーブル式のキー・ジョイン」 381 ページで説明しています。

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 規格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;
```

および

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = C.x, C;
```

ビューと抽出テーブルの外部ジョイン

外部ジョインは、ビューと抽出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ことによって、計算したビュー V のすべてのローが A にジョインされます。

Copyright © 2007, iAnywhere Solutions, Inc.

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 規格に基づき、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。「tsql_outer_joins オプション [互換性]」『SQL Anywhere サーバ - データベース管理』を参照してください。

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、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*』(http://www.ianywhere.jp/tech/index.html) 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)。

Copyright © 2007, iAnywhere Solutions, Inc.

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。この規則は、クエリで参照されるビューにも適用されます。

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 )
```

Transact-SQL 外部ジョインを使ったビューの使用

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。

Copyright © 2007, iAnywhere Solutions, Inc.

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、抽出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
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、管理者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
…	…	…	…

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 規格の拡張機能です。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。

Copyright © 2007, iAnywhere Solutions, Inc.

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;
```

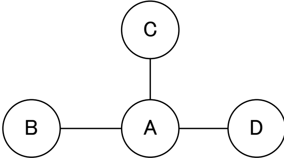
次の例では、テーブル A が 3 つのテーブル B、C、D にジョインされます。

```
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。



Copyright © 2007, iAnywhere Solutions, Inc.

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。
詳細については、「extended_join_syntax オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

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管理者は誰か」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

Copyright © 2007, iAnywhere Solutions, Inc.

ル・データベースの構造 (「サンプル・データベース・スキーマ」 344 ページを参照) 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。詳細については、「セルフジョイン」 365 ページを参照してください。

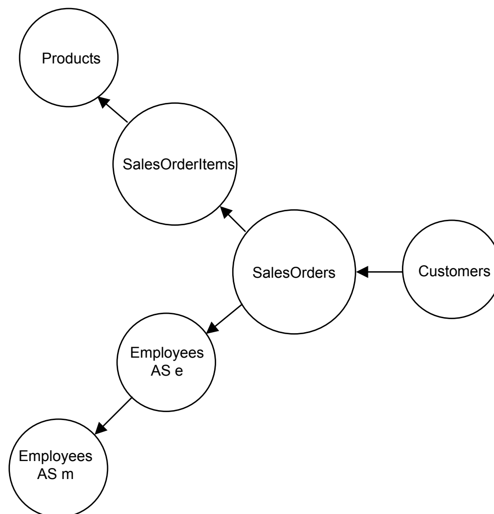
次の文は SalesOrders テーブルを中心にしてスター・ジョインを作成します。このジョインは、結果セットにすべての顧客が含まれるように、すべて外部ジョインになっています。注文しなかった顧客もいますが、そういう顧客の他の値は NULL になっています。結果セットのカラムは、Customers、Products、Quantity ordered、営業部員の管理者名です。

```
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。

Copyright © 2007, iAnywhere Solutions, Inc.


Products
SalesOrderItems
SalesOrders
Customers
Employees
AS e
Employees
AS m

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;
```

抽出テーブルに関連したジョイン

抽出テーブルでは FROM 句内にクエリをネストできます。抽出テーブルを使用すると、ビューを作成せずに、グループのグループ化を実行したり、グループとのジョインを組み立てたりできます。

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抽出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。

抽出テーブルのキー・ジョインについては、「ビューと抽出テーブルのキー・ジョイン」 385 ページを参照してください。

抽出テーブルのナチュラル・ジョインについては、「ビューと抽出テーブルのナチュラル・ジョイン」 376 ページを参照してください。

抽出テーブルの外部ジョインについては、「ビューと抽出テーブルの外部ジョイン」 361 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

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 は次のクエリと同義です。

```
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;
```

Copyright © 2007, iAnywhere Solutions, Inc.

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。

Copyright © 2007, iAnywhere Solutions, Inc.

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、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、「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);
```

これは次と同義です。

```
SELECT *
FROM (Employees KEY JOIN SalesOrders)
  JOIN (SalesOrderItems KEY JOIN Products)
    ON SalesOrders.ID = SalesOrderItems.ID;
```

ビューと抽出テーブルのナチュラル・ジョイン

ANSI/ISO SQL 規格の拡張機能では、ナチュラル・ジョインのどちらの側にもビューまたは抽出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) 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、このビューから抽出テーブルへのナチュラル・ジョインを作成します。抽出テーブルには、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);
```


Copyright © 2007, iAnywhere Solutions, Inc.

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。

KEY JOIN を指定すると、データベース内の外部キー関係に基づいてジョイン条件が生成されます。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。SQL Anywhere サンプル・データベースの詳細については、「チュートリアル：サンプル・データベースの使用」『SQL Anywhere サーバ - データベース管理』を参照してください。

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。

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;
```

Copyright © 2007, iAnywhere Solutions, Inc.

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、上の例は次のクエリと同義です。

```
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。このまとめについては、「キー・ジョイン操作規則」 387 ページを参照してください。

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。

SQL Anywhere サンプル・データベースにあるすべての外部キーの役割名を含む図については、「サンプル・データベース・スキーマ」 344 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

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。

例 1

SQL Anywhere サンプル・データベースでは、テーブル Employees と 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。

Employees		
EmployeeID	<pk>	integer
ManagerID		integer
Surname		person_name_t
GivenName		person_name_t
DepartmentID	<fk>	integer
Street		street_t
City		city_t
State		state_t
Country		country_t
PostalCode		postal_code_t
Phone		phone_number_t
Status		char(2)
SocialSecurityNumber		char(11)
Salary		numeric(20,3)
StartDate		date
TerminationDate		date
BirthDate		date
BenefitHealthInsurance		bit
BenefitLifeInsurance		bit
BenefitDayCare		bit
Sex		char(2)

DepartmentID = DepartmentID
(FK_DepartmentID_DepartmentID)

Departments		
DepartmentID	<pk>	integer
DepartmentName		char(40)
DepartmentHeadID	<fk>	integer

DeparmentHeadID = EmployeeID
(FK_DeparmentHeadID_EmployeeID)

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;
```

Copyright © 2007, iAnywhere Solutions, Inc.

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。

Copyright © 2007, iAnywhere Solutions, Inc.

```
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」 381 ページ
- 「テーブル式リストのキー・ジョイン」 382 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みなされてエラーが発行される。

Copyright © 2007, iAnywhere Solutions, Inc.

- 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み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。

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」 378 ページを参照してください。

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。

- 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;
```


Copyright © 2007, iAnywhere Solutions, Inc.

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み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み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。

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」 378 ページを参照してください。

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

Copyright © 2007, iAnywhere Solutions, Inc.

ブルの相関名と同じ名前の外部キーはほかにないため、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;
```

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 A とテーブル式 C NATURAL JOIN D のジョイン条件と、テーブル B とテーブル式 C NATURAL JOIN D のジョイン条件が生成されます。

```
SELECT *
FROM (A,B) KEY JOIN (C NATURAL JOIN D);
```

ここでは、(A,B) がテーブル式リストで、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。

 Copyright © 2007, iAnywhere Solutions, Inc.

```
((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 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・ジョインのジョイン条件が生成されます。これについては、「カンマを含まないテーブル式のキー・ジョイン」 381 ページを参照してください。

(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・ペア内には有効な外部キー関係が見つけられる必要があります。これについては、「テーブル式リストのキー・ジョイン」 382 ページを参照してください。

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 つだけ作成されます。そのために、C-E と D-E のペア内で外部キー関係が 1 つだけ見つけられます。これについては、「カンマを含まないテーブル式のキー・ジョイン」 381 ページを参照してください。

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 つだけ必要である。その外部キー関係とは、SalesOrders.SalesRepresentative = Employees.EmployeeID である。
- 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 つだけ必要である。その外部キー関係とは、FK_DepartmentID_DepartmentID.DepartmentID = Employees.DepartmentID である。

この例は次の文と同義です。次の例では、必ずしも相関名 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、2 つの外部キーのうち Employees と Departments のジョインで使用するキーを明確にするときだけです。

```
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;
```

ビューと抽出テーブルのキー・ジョイン

キー・ジョインにビューまたは抽出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。
- ビューまたは抽出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抽出テーブルが含まれる場合、ビューまたは抽出テーブル定義には 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、抽出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。

  抽出テーブルとビューの機能は同じです。抽出テーブルの場合、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。

  再帰テーブル式の詳細については、「再帰共通テーブル式」 398 ページと「再帰テーブル・アルゴリズム」 564 ページを参照してください。

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;
```

または

```
SELECT *
FROM C,D;
```

または

```
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「テーブル式のキー・ジョイン」 381 ページで説明した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。

上記のビュー定義のいずれか使用すると、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。

 Copyright © 2007, iAnywhere Solutions, Inc.

- 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み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み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 であるとすると、次に示す元のジョインを展開できます。

```
SELECT *
FROM View1 KEY JOIN B;
```

この文は次の文と同義です。

```
SELECT *
FROM View1 JOIN B ON B.y = View1.z;
```

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」 378 ページを参照してください。

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);
```

これは次と同義です。

```
SELECT *
FROM V JOIN (SalesOrders,
   Departments FK_DepartmentID_DepartmentID)
ON (V.EmployeeID = SalesOrders.SalesRepresentative
AND V.DepartmentID =
    FK_DepartmentID_DepartmentID.DepartmentID);
```

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。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。A1、B1 はカンマを含まないテーブル式を表します。

Copyright © 2007, iAnywhere Solutions, Inc.

1. テーブル式 Ai と Bj の各ペアについて、規則 1 または 2 を適用し、テーブル式 (Ai KEY JOIN Bj) にユニークな生成されたジョイン条件を見つける。テーブル式のペアの KEY JOIN のいずれかが規則 1 または 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：カンマを含む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。A1、B1 はカンマを含むテーブル式を表します。

1. テーブル式 Ai と Bj の各ペアについて、規則 1、2、または 3 を適用し、テーブル式 (Ai KEY JOIN Bj) にユニークな生成されたジョイン条件を見つける。テーブル式のペアの KEY JOIN のいずれかが規則 1、2、または 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。

Copyright © 2007, iAnywhere Solutions, Inc.

第 10 章

共通テーブル式

目次

共通テーブル式の概要 392
共通テーブル式の一般的な使用例 395
再帰共通テーブル式 398
構成部品の問題 400
再帰共通テーブル式でのデータ型宣言 404
最短距離の問題 406
複数の再帰共通テーブル式の使用 410

共通テーブル式の概要

SELECT 文で WITH プレフィクスを使用して、共通テーブル式を定義できます。共通テーブル式は、1 つの SELECT 文のスコープ内のみで認識されるテンポラリ・ビューです。共通テーブル式によって、クエリをより簡単に記述できます。また、共通テーブル式なしでは記述できないクエリもあります。

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を定義する場合に、共通テーブル式を使用すると便利です。また、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。

再帰共通テーブル式によって、たとえば社内の報告関係など、階層情報を表すテーブルに問い合わせできます。また、再帰共通テーブル式は、構成部品の問題や最短距離の問題の解決にも使用できます。

再帰クエリの詳細については、「再帰共通テーブル式」 398 ページを参照してください。

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がぎこちなくなっていることです。2 つめの欠点は、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

 Copyright © 2007, iAnywhere Solutions, Inc.

方法は合理的です。しかし、ビューが特定の SELECT 文内で 1 度しか使用しない場合は、代わりに共通テーブル式を使用することをおすすめします。

共通テーブル式の使用

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。共通テーブル式を使用して、前述のクエリを次のように記述できます。

```
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。

複数の相関名

テーブルを使用するときに、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つ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;
```

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 ),
  DeptPayroll( DepartmentID, amt ) AS
     ( SELECT DepartmentID, SUM( Salary ) AS amt
```

Copyright © 2007, iAnywhere Solutions, Inc.

```
    FROM Employees GROUP BY DepartmentID )
SELECT count.DepartmentID, count.n, pay.amt
FROM CountEmployees AS count JOIN DeptPayroll AS pay
ON count.DepartmentID = pay.DepartmentID
WHERE count.n = ( SELECT MAX( n ) FROM CountEmployees )
   OR pay.amt = ( SELECT MIN( amt ) FROM DeptPayroll );
```

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 DeptPayroll( DepartmentID, amt ) AS
  ( SELECT DepartmentID, SUM( Salary ) AS amt
    FROM Employees GROUP BY DepartmentID )
SELECT DepartmentID, amt
FROM DeptPayroll
WHERE amt = ( SELECT MAX( amt )
              FROM DeptPayroll );
```

- **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、定義内の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 ),
     DeptPayroll( DepartmentID, amt ) AS
       ( SELECT DepartmentID, SUM( Salary ) AS amt
         FROM Employees GROUP BY DepartmentID )
  SELECT count.DepartmentID, count.n, pay.amt
  FROM CountEmployees count JOIN DeptPayroll pay
  ON count.DepartmentID = pay.DepartmentID
  WHERE count.n = ( SELECT MAX( n ) FROM CountEmployees )
     OR pay.amt = ( SELECT MAX( amt ) FROM DeptPayroll );
```

- **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 DeptPayroll( DepartmentID, amt ) AS
    ( SELECT DepartmentID, SUM( Salary ) AS amt
      FROM Employees
      GROUP BY DepartmentID )
  SELECT DepartmentID, amt, CURRENT TIMESTAMP
  FROM DeptPayroll
  WHERE amt = ( SELECT MAX( amt )
                FROM DeptPayroll );
```


Copyright © 2007, iAnywhere Solutions, Inc.

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 DeptPayroll( DepartmentID, amt ) AS
  ( SELECT DepartmentID, SUM( Salary ) AS amt
    FROM Employees GROUP BY DepartmentID )
SELECT DepartmentID, amt
FROM DeptPayroll
WHERE amt = ( SELECT MAX( amt )
              FROM DeptPayroll )
```

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。そして、その顧客の購入履歴を問い合わせたいとします。何度も同様の情報にアクセスしたり、複数の集合関数を使用したりする必要がある場合は、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。

プログラム変数を参照するビューは作成できません。なぜならば、ビューの範囲をプロシージャの範囲に制限する方法がないからです。一度作成すると、ビューは他の場合にも使用できます。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できます。共通テーブル式の範囲はその文に制限されるため、変数の参照にあいまい性はありませんので、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
```

Copyright © 2007, iAnywhere Solutions, Inc.

```
            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。こ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・パラメータと宣言されたどのローカル変数も参照できます。

```
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;
```

次の文は、前述のプロシージャの呼び出し方法を示しています。

```
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、ある会社が販売担当者の成績を四半期ごとでなく、1 年を 3 期に分けて分析する方法を採用しているとします。1 年を 3 期に分けた日付セットが四半期のようには組み込まれていないため、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,
```


Copyright © 2007, iAnywhere Solutions, Inc.

```
       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;

CALL sales_by_third (2000);
```

再帰共通テーブル式

共通テーブル式は、再帰できます。WITH の直後に RECURSIVE キーワードを使用すると、共通テーブル式は再帰的になります。1 つの WITH 句には、複数の再帰式を含めることもできます。また再帰共通テーブル式と非再帰共通テーブル式の両方を含めることができます。

再帰共通テーブル式は、階層の任意の深さに対する関係を返すクエリを記述するのに便利な方法です。たとえば、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簡単に記述できます。

クエリの記述方法によって、再帰レベル数を制限することも、まったく制限しないことも可能です。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、たとえばある従業員が直接または間接的に自分自身に報告するなど、図にあらゆる循環が含まれる場合、無限の再帰となる可能性があります。この状況は、たとえば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。

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のテーブルをトラバースすることがはるかに簡単になります。再帰式を使用せずに単一の文内でそうした構造をトラバースする唯一の方法は、考えられる各レベルごとに 1 回ずつテーブルをそれ自体にジョインすることです。たとえば、報告階層に含まれるレベルが最大で 7 つある場合、Employees テーブルをそれ自体に 7 回ジョインする必要があります。会社の組織が変更され、新しい管理レベルが導入された場合は、クエリを再度記述する必要があります。

再帰共通テーブル式には、*初期サブクエリ* (シード) と、各繰り返し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。

共通テーブル式の詳細については、「共通テーブル式の概要」 392 ページを参照してください。

階層データの選択

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
```


Copyright © 2007, iAnywhere Solutions, Inc.

```
    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;
```

再帰クエリ内の条件として管理レベルを 20 未満に制限するのは、重要な予防措置です。これによって、テーブル・データに循環が含まれる場合も無限再帰を防げます。

max_recursive_iterations オプション

max_recursive_iterations オプションは、暴走する再帰クエリを捕らえることを目的として設計されています。このオプションのデフォルト値は 100 です。この再帰レベル数を超えた再帰クエリは終了し、エラーを発生させます。

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。各繰り返し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、最大に達する前にデータベースの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関する制限

再帰共通テーブル式に適用される制限を次に示します。

- 他の再帰共通テーブル式への参照は、再帰共通テーブル式の定義内に含められません。したがって、再帰共通テーブル式は相互に再帰できません。ただし、非再帰共通テーブル式には、再帰共通テーブル式への参照を含められます。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られます。
- 初期サブクエリと再帰サブクエリ間で唯一許可されている set 演算子は、UNION ALL です。他の set 演算子は許可されていません。
- 再帰サブクエリ定義内では、再帰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ます。
- 再帰サブクエリの FROM 句内に自己参照が含まれる場合、再帰テーブルへの参照は外部ジョインの NULL 入力側に含められません。
- 再帰サブクエリには、DISTINCT 句、GROUP BY 句、ORDER BY 句は含められません。
- 再帰サブクエリでは、集合関数はすべて使用できません。
- 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。

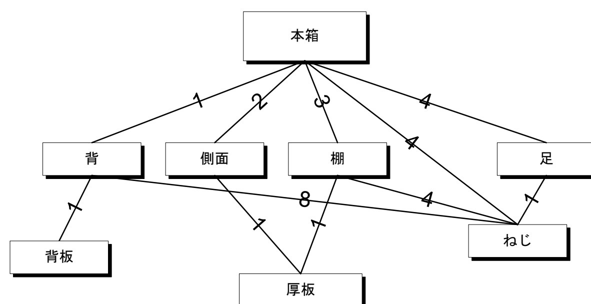
Copyright © 2007, iAnywhere Solutions, Inc.

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、次の図は単純な本棚のコンポーネントを表しています。この本棚は、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。

 Copyright © 2007, iAnywhere Solutions, Inc.


本箱
1
2
3
4
4
背
側面
棚
足
8
4
1
1
1
1
ねじ
背板
厚板

次のテーブル内の情報は、本棚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。

コンポーネント	サブコンポーネント	数量
本棚	背	1
本棚	側面	2
本棚	棚	3
本棚	脚	4
本棚	ねじ	4
背	背板	1
背	ねじ	8
側面	厚板	1
棚	厚板	1
棚	ねじ	4

次の文は、bookcase テーブルを作成し、前述のテーブルに示された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;
```

bookcase テーブルを作成したら、前述の部品テーブルを次のクエリを使用して再作成できます。

```
SELECT * FROM bookcase
ORDER BY component, subcomponent;
```

このテーブルが作成できたら、本棚の組み立てに必要な基本部品と各必要数量のリストを生成できます。

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT component, subcomponent, quantity
  FROM bookcase WHERE component = 'bookcase'
```


Copyright © 2007, iAnywhere Solutions, Inc.

```
    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。

サブコンポーネント	数量
背板	1
脚	4
厚板	5
ねじ	24

もう 1 つの選択肢として、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。

```
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;
```

このクエリの結果は、前述のクエリの結果とまったく同じです。

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、本棚の構成部品のサンプルは、初期サブクエリが bookcase テーブルからの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。

詳細については、「構成部品の問題」 400 ページを参照してください。

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 short integer 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 )
```


Copyright © 2007, iAnywhere Solutions, Inc.

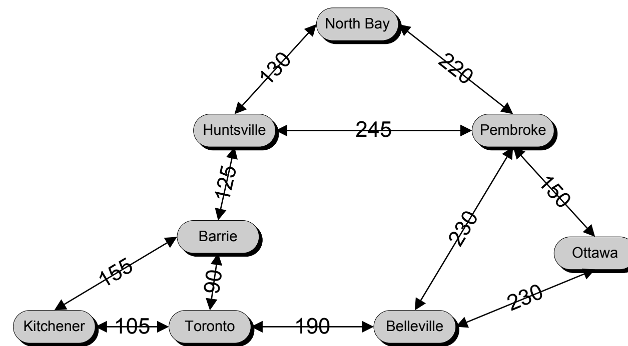
```
SELECT * FROM parts
ORDER BY component, subcomponent;
```

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。

 Copyright © 2007, iAnywhere Solutions, Inc.



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 2 方向性を持つため、逆方向を表すエッジも挿入する必要がありま

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。効率をあげるために、次の条件のいずれかに該当した場合、ルートは終端されます。

- パスが出発地に戻る場合。
- パスが直前の地点に戻る場合。
- パスが目的地に達した場合。

この例では、どのパスもキッチナー市に戻ってはならず、全てのパスはペンブローク市に達した場合、終端されなければなりません。

環状のグラフを探索するために再帰クエリを使用する場合は、再帰クエリが適切に終了するように保証することが特に重要です。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。次の再帰クエリは、どのルートのセグメント数も最大 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
```


Copyright © 2007, iAnywhere Solutions, Inc.

```
( 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。

ルート	距離	セグメント数
キッチナー、バリー、ハンツビル、ペンブローク	525	3
キッチナー、トロント、ベルビル、ペンブローク	525	3
キッチナー、トロント、バリー、ハンツビル、ペンブローク	565	4
キッチナー、バリー、ハンツビル、ノースベイ、ペンブローク	630	4
キッチナー、バリー、トロント、ベルビル、ペンブローク	665	4
キッチナー、トロント、バリー、ハンツビル、ノースベイ、ペンブローク	670	5
キッチナー、トロント、ベルビル、オタワ、ペンブローク	675	4

複数の再帰共通テーブル式の使用

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
```


Copyright © 2007, iAnywhere Solutions, Inc.

```
  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;

CALL best_routes ( 'Pembroke', 'Kitchener' );
```

第 11 章

OLAP のサポート

目次

OLAP 機能の概要 414
GROUP BY 句の拡張 416
Window 関数 425

Copyright © 2007, iAnywhere Solutions, Inc.

OLAP 機能の概要

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 Anywhere では、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 句に対する拡張では、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すべて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。「GROUP BY 句の拡張」 416 ページを参照してください。

  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 関数」 425 ページを参照してください。

- **Window 集合関数**　ほぼすべての SQL Anywhere 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」**の概念をサポートしています。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 集合関数」 432 ページを参照してください。

- **Window ランキング関数**　Windows 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。「Window ランキング関数」 450 ページを参照してください。

OLAP のパフォーマンス向上

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ード GROUP BY ハッシュ演算子を使用することを

Copyright © 2007, iAnywhere Solutions, Inc.

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、クラスタード 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 [データベース]」『SQL Anywhere サーバ - データベース管理』
- 「クラスタード・ハッシュ GROUP BY アルゴリズム」560 ページ
- 「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

Copyright © 2007, iAnywhere Solutions, Inc.

	City	State	CompanyName	Cnt
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 ー 9 は、CompanyName ごとにグループ化して生成されたローで、ロー 10 ー 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 ー 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 ー 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。「GROUPING 関数を使用したプレースホルダの NULL の検出」 423 ページを参照してください。

例

次の例では、GROUPING SETS を使用したクエリから返された結果を調整する方法と、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 ー 5 とロー 7 ー 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。「GROUPING 関数を使用したプレースホルダの NULL の検出」 423 ページを参照してください。

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 ー 3 がロー 4 ー 6 と同一であることに注意してください。

	City	Cnt
1	'Drayton'	3

 Copyright © 2007, iAnywhere Solutions, Inc.

	City	Cnt
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 ) );
```

ROLLUP と CUBE の使用

多く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。「GROUPING 関数を使用したプレースホルダの NULL の検出」 423 ページを参照してください。

ROLLUP の使用

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。SQL Anywhere では、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。

$\{(), (X_1), (X_1,X_2), (X_1,X_2,X_3), \ldots, (X_1,X_2,X_3, \ldots, X_n)\}$.

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。

Copyright © 2007, iAnywhere Solutions, Inc.

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。

ROLLUP 句の構文の詳細については、「GROUP BY 句」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

T-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;
```

CUBE の使用

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。

$$\{(), (X_1), (X_1,X_2), (X_1,X_2,X_3), ..., (X_1,X_2,X_3,...,X_n), (X_2), (X_2,X_3), (X_2,X_3,X_4), ..., (X_2,X_3,X_4,...,X_n), ..., (X_n)\}.$$

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;
```

Copyright © 2007, iAnywhere Solutions, Inc.

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
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 ー 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 ー 10 と 12 ー 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 SQL Anywhere では、GROUP BY 式が 64 個以上含まれる GROUP BY 句を許可していません。この上限を超える文は、SQLCODE -944 (SQLSTATE 42WA1) で失敗します。

 Copyright © 2007, iAnywhere Solutions, Inc.

CUBE 句の構文の詳細については、「GROUP BY 句」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

T-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;
```

GROUPING 関数を使用したプレースホルダの 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。つまり、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です。

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

	Employees	Year	Orders	GE	GY
3	102	(NULL)	0	0	0
4	390	(NULL)	0	0	0
5	1062	(NULL)	0	0	0
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。

GROUPING 関数の構文の詳細については、「GROUPING 関数 [集合]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

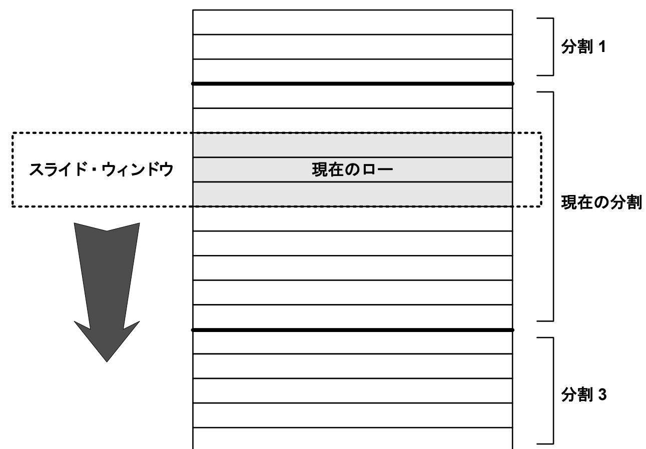
Window 関数

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。

入力ローの処理に合わせたウィンドウの移動を次の図に示します。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。

Copyright © 2007, iAnywhere Solutions, Inc.



Copyright © 2007, iAnywhere Solutions, Inc.

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と、ほぼすべての集合関数は、「**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、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。サポートされる Window 集合関数は次のとおりです。

  - AVG
  - COVAR_POP
  - COVAR_SAMP
  - CUME_DIST
  - DENSE_RANK
  - MAX
  - MIN
  - PERCENT_RANK
  - RANK
  - REGR_AVGX
  - REGR_AVGY
  - REGR_COUNT
  - REGR_INTERCEPT
  - REGR_R2
  - REGR_SLOPE
  - REGR_SXX
  - REGR_SXY
  - REGR_SYY
  - STDDEV
  - STDDEV_POP
  - STDDEV_SAMP
  - SUM
  - VAR_POP
  - VAR_SAMP
  - VARIANCE

  Window 集合関数の詳細については、「Window 集合関数」 432 ページを参照してください。

- **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 Window ランキング関数は次のとおりです。

  - CUME_DIST
  - DENSE_RANK
  - PERCENT_RANK
  - RANK

ランキング関数の詳細については、「Window ランキング関数」 450 ページを参照してください。

- **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。SQL Anywhere の ROW_NUMBER 関数は、ANSI 標準準拠の関数で、SQL Anywhere の NUMBER(*) 関数と同等の機能の多くを使用できます。

  この関数の使用の詳細については、「ROW_NUMBER 関数」 458 ページを参照してください。

ウィンドウの定義

SQL のウィンドウ拡張を使用すると、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。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、ウィンドウを使うことでもグループ化と順序付けの操作を実行できます。

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 句を使用して入力ローを分割 (またはグループ化) する方法を定義します。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**　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であると判断した方法で入力ローが処理されます。

- **境界**　現在のローの値からのオフセットでデータ値の範囲 (RANGE 句)、または現在のローからのオフセットでローの数 (ROWS 句) を使用して、ウィンドウの境界を定義します。ウィンドウ・サイズは分割の 1 つ、複数、またはすべてのローとなります。

  ROWS 句や RANGE 句では、オプションの PRECEDING、BETWEEN、FOLLOWING の各句を組み合わせることで、ウィンドウの先頭ローと末尾ローを現在のローからの相対的な値として指定します。これらの句では、式だけでなく、UNBOUNDED と CURRENT ROW の各キーワードも使用できます。

  ウィンドウの境界が指定されていない場合、ウィンドウのサイズ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 句が含まれる場合、ウィンドウの開始ポイントは UNBOUNDED PRECEDING、終了ポイントは CURRENT ROW になります。

Copyright © 2007, iAnywhere Solutions, Inc.

- ウィンドウ指定に ORDER BY 句が含まれない場合、ウィンドウの開始ポイントは UNBOUNDED PRECEDING、終了ポイントは UNBOUNDED FOLLOWING になります。

最善な方法で判断するために、常にウィンドウの境界を定義して、ウィンドウ指定を明確にしてください。

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。

ウィンドウのサイズ変更

現在のローは、ウィンドウの開始ポイントと終了ポイントを判断するための参照ポイントになります。指定するウィンドウの境界により、現在のローに相対的なウィンドウが定義されます。

境界は、ROWS 句を使用してローの正確な数として指定する方法と、RANGE 句を使用して現在のローの値からのオフセットで値の範囲として指定する方法があります。後者の場合は、ウィンドウのサイズは囲んでいるローの値に応じて変化する可能性があります。

ローの数 (ROWS 句) を指定した境界の設定

指定できるウィンドウの境界のその他の例を次に示します。

- **ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW** このウィンドウは、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) を計算するときは、この構成を使用してください。
- **ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING** この構文は、分割全体に固定ウィンドウ (現在のローとは無関係) を指定します。分割の各ローに対して集合関数の値を同一にするときは、この構成を使用してください。
- **ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING** この構文は、現在のローとその前後のローからなる固定サイズ (隣接する 3 ロー) の移動するウィンドウを指定します。たとえば、3 日間または 3 か月間の移動平均を計算するときに、この構文を使用してください。RANGE 句を基にしたウィンドウの境界では、隣接するローで範囲に差があったり重複した値のローがあったりする場合に、そのような隣接するローが自動的に処理されます。そのため、値の組が連続的でない場合は、Window 関数の入力の差に起因する問題を避けるために、ROWS 句ではなく RANGE 句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 が存在します。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ときに、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** この構文は、現在のロー 1 つからなるウィンドウを指定します。
- **ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING** この構文は、現在のローの直前のロー 1 つからなるウィンドウを指定します。この構成により、隣接するロー間のデルタ (値の差分) の計算が簡単になります。

範囲 (RANGE 句) を指定した境界の設定

RANGE 句を使用するときは、ORDER BY 句で指定したカラムの値を基にウィンドウのサイズを定義します。RANGE 句を使用するには、ORDER BY 句も指定する必要があります。また、ORDER BY 句にカラム 1 つだけが含まれ、そのカラムが番号ドメイン内にある必要があります。ウィンドウのサイズは、ORDER BY 句で指定したカラムの値と現在のローの値を比較することで決まります。

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。

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。

ウィンドウのインラインの定義、または WINDOW 句の使用

ウィンドウ定義は、Window 関数の OVER 句に配置できます。このウィンドウ定義は、インライン構文を使用してウィンドウを定義するときに参照されます。また、WINDOW 句内でウィンドウを個別に定義することもできます。この場合は、関数から参照されます。WINDOW 句を使用する利点は、ウィンドウが個別に定義されるため、複数の関数がウィンドウを参照できることと、ウィンドウでその他の計算を実行できることです。

たとえば、次の SELECT 文は、同じ名前付きウィンドウ (Qty) を参照する 3 つの異なる関数を示します。

```
SELECT Products.ID, Description, Quantity,
  RANK() OVER Qty AS Rank_quantity,
  CUME_DIST() OVER Qty AS Dist_Cume,
  ROW_NUMBER() OVER Qty AS Qty_order
FROM Products
WINDOW Qty AS ( ORDER BY Quantity ASC )
ORDER BY Qty_order;
```

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。

インラインのウィンドウ定義の例

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、SQL Anywhere 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。

Copyright © 2007, iAnywhere Solutions, Inc.

```
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
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 句によるウィンドウ定義の例

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。

Copyright © 2007, iAnywhere Solutions, Inc.

```
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 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。

この章で説明するために、Window 集合関数を次の 3 つのカテゴリに分類します。

- **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
  - AVG
  - MAX
  - MIN
  - FIRST_VALUE
  - LAST_VALUE
  - COUNT
- **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
  - STDDEV_POP
  - STDDEV_SAMP
  - VAR_POP
  - VAR_SAMP
  - VARIANCE
- **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
  - COVAR_SAMP
  - REGR_AVGX
  - REGR_AVGY
  - REGR_COUNT
  - REGR_INTERCEPT
  - REGR_R2
  - REGR_SLOPE
  - REGR_SXX
  - REGR_SXY
  - REGR_SYY

Copyright © 2007, iAnywhere Solutions, Inc.

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。

SQL Anywhere の基本集合関数は次のとおりです。

- **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、数値式または設定されたユニークな値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。
- **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

- 「Window 関数」 425 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

Copyright © 2007, iAnywhere Solutions, Inc.

	DepartmentID	Surname	StartDate	Salary	Sum_Salary
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。

SUM 関数の正確な構文の詳細については、「SUM 関数 [集合]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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)

 Copyright © 2007, iAnywhere Solutions, Inc.

	EmployeeNumber	LastName	CurrentRow	PreviousRow	Delta
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。また、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 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;
```

Copyright © 2007, iAnywhere Solutions, Inc.

demo (DBA) on demo10

File Edit SQL Data Tools Window Help

SQL Statements

```
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 = (
```

Results

Main Query

Details　Advanced Details

SELECT

```
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
```

SELECT / Work / Sort / Filter / GrByH / JH* / JH* / o / s / p

Node Statistics

Results　Messages　Plan

Line 15　Column 24　11 rows

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
6	500	949	360	2520.00

Copyright © 2007, iAnywhere Solutions, Inc.

	Products	SalesRepresentative	total_quantity	total_sales
7	501	690	360	2520.00
8	501	949	360	2520.00
9	600	299	612	14688.00
10	601	299	336	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抽出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,
```


Copyright © 2007, iAnywhere Solutions, Inc.

```
          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。

demo (DBA) on demo10

File Edit SQL Data Tools Window Help

SQL Statements

```
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
```

Results

Main Query

SELECT
Work
Filter
DT
Window
Sort
GrByH
JH*
JH* o
s p

Details Advanced Details

SELECT

```
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
```

Node Statistics

	Estimates	Actual	Description
Invocations	-	1	Number of times the result was computed
RowsReturned	54.85	11	Number of rows returned
PercentTotalCost	0	0.32467	Run time as a percent of total query time
RunTime	0	7.04e-005	Time to compute the results
CPUTime	0	-	Time required by CPU

Results Messages Plan

Line 13 Column 24 11 rows

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 SalesRepresentatives はすべてまとめられます。このランキングにより、最上位の営業担当者を特定するのに必要なのは、ランクが 1 ではない抽出テーブルのローを拒否するように抽出テーブルの結果を制限するだけです。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販売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 句を使用していることに注意してください。ROWS 句の場合のように単に隣接ローの数で計算されるのではなく、RANGE 句により月の値を基にウィンドウ境界が計算されます。たとえば、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に、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

 Copyright © 2007, iAnywhere Solutions, Inc.

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、Interactive SQL の [結果] ウィンドウ枠で [プラン] タブに表示されます。クエリ・オプティマイザがどのように、このネストされたクエリを変形して、Window 関数を含む抽出テーブル (相関名 DT) による Products テーブルと SalesOrders テーブルをジョインするかに注目してください。

demo (DBA) on demo10

File Edit SQL Data Tools Window Help

SQL Statements

```
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;
```

Results

Main Query

SELECT / Work / Sort / JNL / JNL / o / JNL / s / DT / p / GrByO / s2

Details | Advanced Details

SELECT

```
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
```

Node Statistics

	Estimates	Actual	Description
Invocations	-	1	Number of times the result was computed
RowsReturned	421.92	743	Number of rows returned
PercentTotalCost	3.75	4.3633	Run time as a percent of total query time
RunTime	0.0021096	0.0064701	Time to compute the results
CPUTime	0.0021096	-	Time required by CPU
DiskReadTime	0	-	Time to perform reads from disk
DiskWriteTime	0	-	Time to perform writes to disk
DiskRead	0	0	Disk reads

Results | Messages | Plan

Line 7 Column 21

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_qty.ID, o.OrderDate, p.*
  FROM ( SELECT s.ID, s.ProductID,
           MAX( s.Quantity ) OVER (
             PARTITION BY s.ProductID
             ORDER BY s.ProductID )
           AS max_q
           FROM SalesOrderItems s )
  AS order_qty, Products p, SalesOrders o
  WHERE p.ID = ProductID
    AND o.ID = order_qty.ID
    AND p.Quantity < max_q
  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。指定しなかった場合は、NULL であるかどうかに関係なく、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,
```

 Copyright © 2007, iAnywhere Solutions, Inc.

```
  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：データの密度を高めるときの NULL 値の挿入

FIRST_VALUE 関数と LAST_VALUE 関数は、データの密度を高めた後で、NULL の代わりに値を挿入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 週に表彰され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を作成します。また、LAST_VALUE 関数を使用して、表彰がなかった日には、結果に次の表彰者が返されるまで、rep_of_the_day の NULL 値の代わりに、最後に表彰された担当者の ID が挿入されます。

```
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。

OrderDate	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

前のクエリからの抽出テーブル v を、対象日をすべて含む抽出テーブル 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」 425 ページ

標準偏差関数と平方偏差関数

SQL Anywhere では、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 つのバージョンをサポートしています。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、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、SQL Anywhere は平均と平均からの偏差を同時に計算します。つまり、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。

SQL Anywhere で提供される標準偏差関数と平方偏差関数は、次のとおりです。

Copyright © 2007, iAnywhere Solutions, Inc.

- STDDEV 関数
- STDDEV_POP 関数
- STDDEV_SAMP 関数
- VARIANCE 関数
- VAR_POP 関数
- VAR_SAMP 関数

これらの関数が表す数学上の式を確認するには、「集合関数の数学上の式」 458 ページを参照してください。

STDDEV 関数

この関数は、STDDEV_SAMP 関数のエイリアスです。「STDDEV_SAMP 関数 [集合]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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	DepartmentID	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

Copyright © 2007, iAnywhere Solutions, Inc.

	Employee	DepartmentID	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...
...	...	...	...

この関数の構文の詳細については、「STDDEV_SAMP 関数 [集合]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

```
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...
...	...	...	...

この関数の構文の詳細については、「STDDEV_POP 関数 [集合]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

VARIANCE 関数

この関数は、VAR_SAMP 関数のエイリアスです。「VAR_SAMP 関数 [集合]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

この関数の構文の詳細については、「VAR_POP 関数 [集合]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

この関数の構文の詳細については、「VAR_SAMP 関数 [集合]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

相関関数と線形回帰関数

SQL Anywhere では、さまざまな統計関数をサポートしています。関数の結果は、線形回帰の質を分析するのに役立ちます。

これらの関数が表す数学上の式の詳細については、「集合関数の数学上の式」 458 ページを参照してください。

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。

  この関数の構文の詳細については、「COVAR_SAMP 関数 [集合]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **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。

  この関数の構文の詳細については、「COVAR_POP 関数 [集合]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **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。

 Copyright © 2007, iAnywhere Solutions, Inc.

この関数の構文の詳細については、「CORR 関数 [集合]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **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。

  この関数の構文の詳細については、「REGR_AVGX 関数 [集合]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **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。

  この関数の構文の詳細については、「REGR_AVGY 関数 [集合]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **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。

  この関数の構文の詳細については、「REGR_SLOPE 関数 [集合]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **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。

  この関数の構文の詳細については、「REGR_INTERCEPT 関数 [集合]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **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。

  この関数の構文の詳細については、「REGR_R2 関数 [集合]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **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。

  この関数の構文の詳細については、「REGR_COUNT 関数 [集合]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **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。

  この関数の構文の詳細については、「REGR_SXX 関数 [集合]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **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。

  この関数の構文の詳細については、「REGR_SYY 関数 [集合]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

- **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。

  この関数の構文の詳細については、「REGR_SXY 関数 [集合]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Window ランキング関数

ランキング関数は、入力の各ローに対して値を返します。SQL Anywhere でサポートされるランキング関数は、RANK、DENSE_RANK、PERCENT_RANK、CUME_DIST です。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 句を指定できません。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。SQL Anywhere では、NULL 値は常に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

Copyright © 2007, iAnywhere Solutions, Inc.

	Description	Quantity	UnitPrice	Rank
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。「DENSE_RANK 関数」 452 ページを参照してください。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

```
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
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

RANK 関数の構文の詳細については、「RANK 関数 [ランキング]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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)。

Copyright © 2007, iAnywhere Solutions, Inc.

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。「RANK 関数」 450 ページを参照してください。

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、集合関数の値を元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

Copyright © 2007, iAnywhere Solutions, Inc.

	SalesRepresentative	Region	total_sales	sales_rank
2	(NULL)	Canada	24768.00	1
3	1596	Canada	3564.00	2
4	856	Canada	2724.00	3
5	299	Central	32592.00	1
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の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。

DENSE_RANK 関数の構文の詳細については、「DENSE_RANK 関数 [ランキング]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

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 ー 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)。

例 1

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

例 2

CUME_DIST 関数は、値のセットの中央値を求める簡単な方法です。CUME_DIST を使用すると、同順がある場合や、入力ローの数が偶数か奇数に関係なく、中央値を計算できます。基本的に必要なことは、最初のローの CUME_DIST 値 が 0.5 以上であることを特定することだけです。

次のクエリは、単価が中央値である製品の製品情報を返します。

```
SELECT FIRST *
 FROM ( SELECT Description, Quantity, UnitPrice,
      CUME_DIST() OVER ( ORDER BY UnitPrice ASC ) AS CDist
     FROM Products ) As DT
 WHERE CDist >= 0.5
 ORDER BY CDist;
```

このクエリは、次の結果を返します。

Description	Quantity	UnitPrice	CDist
Wool cap	12	10.00	0.5

CUME_DIST 関数の構文の詳細については、「CUME_DIST 関数 [ランキング]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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)/(n-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、性別によって分割されます。

```
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。

 Copyright © 2007, iAnywhere Solutions, Inc.

	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

PERCENT_RANK 関数の構文の詳細については、「PERCENT_RANK 関数 [ランキング]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ロー番号付け関数

SQL Anywhere では 2 つのロー番号付け関数 NUMBER と ROW_NUMBER がサポートされていますが、可能なかぎり ROW_NUMBER 関数を使うことをおすすめします。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。「NUMBER 関数 [その他]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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、抽出テーブルで 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、分割全体のみ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。

ROW_NUMBER 関数の構文の詳細については、「ROW_NUMBER 関数 [その他]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

集合関数の数学上の式

情報を提供する目的で、SQL Anywhere でサポートされるすべての Window 集合関数について、等価な数学上の式を次の 2 つの表に示します。

 Copyright © 2007, iAnywhere Solutions, Inc.

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*	$\sum xy - \frac{(\sum y)(\sum x)}{n}$

第 12 章

サブクエリの使用

目次

サブクエリの概要 462
WHERE 句でのサブクエリの使用 468
HAVING 句でのサブクエリ 469
サブクエリ比較テスト 471
ANY と ALL を使用した限定比較テスト 472
IN 条件によるセット・メンバシップのテスト 475
存在テスト 477
外部参照 479
サブクエリとジョイン 480
ネストされたサブクエリ 482
サブクエリ操作 484

サブクエリの概要

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管できます。別のクエリの WHERE 句または HAVING 句に表示されるクエリである「**サブクエリ**」を使用して、関連するテーブルからデータを抽出できます。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、あるクエリの結果を別のクエリの一部として使用します。この項では、サブクエリを使用できる場合について説明します。例として、在庫が少なくなっている製品の注文項目をリストするクエリを構築します。

このリストを生成するには、2 つのクエリが必要です。この項では、最初に個々のクエリについて説明してから、同じ結果を生成する 1 つのクエリについて説明します。

たとえば、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。

一般的に、テーブルを 1 つだけ使用して回答できるのは、最も簡単な質問だけです。たとえば、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、「在庫切れ寸前の状態」が、一般顧客が注文するタイプごとに品目量が異なる場合は、数字 "50" を SalesOrderItems テーブルから取得する値に置き換える必要があります。

サブクエリの構造

サブクエリは通常のクエリのように構成され、メイン・クエリの SELECT 句、FROM 句、WHERE 句、または HAVING 句の中に置かれます。たとえば、前述の例では、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選択し、その数をメイン・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。

簡単な例

サブクエリを使用する 3 つの簡単な例を次に示します。

 Copyright © 2007, iAnywhere Solutions, Inc.

♦ 在庫数が 20 未満のすべての製品をリストするには、次の手順に従います。

- Interactive SQL で次の文を実行します。

```
SELECT ID, Description, Quantity
FROM Products
WHERE Quantity < 20;
```

ID	Description	Quantity
401	Wool cap	12

このクエリは、在庫数が少ない製品はウールの帽子だけであることを示します。

♦ ウールの帽子についてすべての注文項目をリストするには、次の手順に従います。

- Interactive SQL で次の文を実行します。

```
SELECT *
FROM SalesOrderItems
WHERE ProductID = 401
ORDER BY ShipDate DESC;
```

ID	LineID	ProductID	Quantity	ShipDate
2082	1	401	48	7/9/2001
2053	1	401	60	6/30/2001
2125	2	401	36	6/28/2001
2027	1	401	12	6/17/2001
…	…	…	…	…

在庫数が少ない項目を調べて、これらの項目に対する受注を調べるというこの 2 段階の処理は、サブクエリを使用して単一のクエリに結合できます。

♦ 在庫数が少ない製品のすべての注文項目をリストするには、次の手順に従います。

- Interactive SQL で次の文を実行します。

```
SELECT *
FROM SalesOrderItems
WHERE ProductID IN
  (  SELECT ID
     FROM Products
     WHERE Quantity < 20 )
ORDER BY ShipDate DESC;
```

ID	LineID	ProductID	Quantity	ShipDate
2082	1	401	48	7/9/2001

ID	LineID	ProductID	Quantity	ShipDate
2053	1	401	60	6/30/2001
2125	2	401	36	6/28/2001
2027	1	401	12	6/17/2001
…	…	…	…	…

SQL 文中のサブクエリは、カッコで囲まれた次のフレーズです。

```
( SELECT ID
  FROM Products
  WHERE Quantity < 20 );
```

このサブクエリは、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。

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として扱い、メイン・クエリ内の各ローがセットのメンバかどうかをテストします。

単一ローのサブクエリと複数ローのサブクエリ

サブクエリが返すことのできるローとカラムの数には制約があります。IN、ANY、または ALL を使用すると、サブクエリは複数のローを返すことができますが、返されるカラムは 1 つだけです。その他の演算子を使用すると、サブクエリは 1 つの値を返します。

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 FinancialData.Code IN
  ( SELECT FinancialCodes.Code
    FROM FinancialCodes
    WHERE type = 'revenue' );
```

Year	Quarter	Code	Amount
1999	Q1	r1	1023
1999	Q2	r1	2033
1999	Q3	r1	2998

 Copyright © 2007, iAnywhere Solutions, Inc.

Year	Quarter	Code	Amount
1999	Q4	r1	3014
2000	Q1	r1	3114

この例では、各参照の Code カラムが属するテーブルを明確に識別する修飾子を使用します。この例に関しては、修飾子を省略することもできます。

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コマンドと同じです。

```
SELECT *
FROM FinancialData
WHERE FinancialData.Code NOT IN
  (  SELECT FinancialCodes.Code
    FROM FinancialCodes
    WHERE type = 'revenue' );
```

単一ローのサブクエリ

IN 条件とともに使用されるサブクエリがローのセットを返してもよいのに対して、比較演算子とともに使用されるサブクエリはローを 1 つだけ返す必要があります。たとえば、次のコマンドを実行すると、サブクエリは 2 つのローを返すのでエラーが発生します。

```
-- this query returns an error
SELECT *
FROM FinancialData
WHERE FinancialData.Code =
   ( SELECT FinancialCodes.Code
    FROM FinancialCodes
    WHERE type = 'revenue' );
```

サブクエリの使用における一般的なエラー

通常、サブクエリの結果セットは単一カラムに制限されています。次の例は、FinancialCodes テーブルのどのカラムを FinancialData.Code カラムと比較するかを SQL Anywhere が認識しないので意味がありません。

```
-- this query returns an error
SELECT *
FROM FinancialData
WHERE FinancialData.Code IN
   ( SELECT FinancialCodes.Code, FinancialCodes.type
     FROM FinancialCodes
     WHERE type = 'revenue' );
```

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;
```

ID	OrderDate	CompanyName
2488	1/15/2001	North Land Trading
2513	2/05/2001	The Hat Company
2518	2/10/2001	Sports Replay
2049	2/17/2001	Cooper Inc.
...	...	...

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;
```


Copyright © 2007, iAnywhere Solutions, Inc.

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。これは「**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。

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;
```

Copyright © 2007, iAnywhere Solutions, Inc.

WHERE 句でのサブクエリの使用

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。WHERE 句内で使用できる簡単な探索条件については、「クエリ：テーブルからのデータの選択」 283 ページの章で説明します。

Copyright © 2007, iAnywhere Solutions, Inc.

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、当然 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
Shorts	80

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 );
```

Copyright © 2007, iAnywhere Solutions, Inc.

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。

サブクエリのテスト

HAVING 句内で使用できる簡単な探索条件については、「クエリ：テーブルからのデータの選択」 283 ページの章で説明します。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。

- **サブクエリ比較テスト**　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。
- **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。

Copyright © 2007, iAnywhere Solutions, Inc.

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
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);
```

このクエリは、「select リストの中にカラムが 2 つ以上指定されています。」というエラーを返します。

ANY と ALL を使用した限定比較テスト

限定比較テストは、ALL テストと ANY テストの 2 つのカテゴリに分類できます。

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

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 NULL 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。

ALL テスト

ANY テストと同様、ALL テストは、6 つの 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

「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 を検索する」という要求は、当然 ALL テストで処理されます。

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > ALL (
  SELECT ShipDate
  FROM SalesOrderItems
  WHERE ID=2001 );
```

ID	CustomerID
2002	102
2003	103
2004	104

ID	CustomerID
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か、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か、さらに…」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)
```

このテストの詳細については、「限定比較テスト」 486 ページを参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

IN 条件によるセット・メンバシップのテスト

サブクエリ・セット・メンバシップ・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・メンバシップ・テストの否定

サブクエリ・セット・メンバシップ・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' ) );
```

 Copyright © 2007, iAnywhere Solutions, Inc.

存在テスト

サブクエリ比較テストとセット・メンバシップ・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 日以降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
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。

詳細については、「相関サブクエリ」 484 ページを参照してください。

```
SELECT *
FROM SalesOrders
WHERE ( OrderDate > '2001-07-13' ) AND
      ( Customers.ID = SalesOrders.CustomerID )

SELECT OrderDate
FROM SalesOrders
WHERE ( OrderDate > '2001-07-13' ) AND
      ( Customers.ID = SalesOrders.CustomerID );
```

Copyright © 2007, iAnywhere Solutions, Inc.

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 というフォームで否定できます。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**」であるといいます。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。

Copyright © 2007, iAnywhere Solutions, Inc.

外部参照

通常、サブクエリの本体では、メイン・クエリのアクティブなローにあるカラムの値を参照する必要があります。次のクエリを考えてみます。

```
SELECT Name, Description
FROM Products
WHERE Quantity < 2 * (
  SELECT AVG( Quantity )
  FROM SalesOrderItems
  WHERE Products.ID = SalesOrderItems.ProductID );
```

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。SQL 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。

外部参照の説明

このサブクエリの Products.ID カラムは、外部参照の例です。外部参照を使用するサブクエリを相関サブクエリと言います。外部参照は、サブクエリの FROM 句内のどのテーブルのどのカラムも参照しないカラム名です。代わりに、このカラム名は、メイン・クエリの FROM 句に指定されるテーブルのカラムを参照します。前述の例が示すように、外部参照のカラムの値は、メイン・クエリによって現在テストされているローから抽出されます。

サブクエリとジョイン

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。

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。

ジョインによるサブクエリの置き換え

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フォームを次に示します。

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
  ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

このフォームのクエリは 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。

サブクエリとして作成できないジョイン

これらのクエリはどちらも正確な注文日と担当者を検索します。どちらかの方がより正確ということはありません。通常は、サブクエリのフォームの方が自然に感じられます。顧客 ID についての情報は要求されておらず、この問い合わせに応答するために SalesOrders テーブルと Customers テーブルをジョインするのは不自然に感じられる場合があるためです。

しかし、Customers テーブルからの情報を含むように要求が変更されると、サブクエリのフォームは機能しなくなります。たとえば、「Mrs Clarke と Suresh がいつ注文し、どの担当者が注文を

Copyright © 2007, iAnywhere Solutions, Inc.

受け、この 2 人の顧客の氏名は何か」という要求の場合は、メインの WHERE 句に Customers テーブルを含む必要があります。

```
SELECT GivenName, Surname, OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
  ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

GivenName	Surname	OrderDate	SalesRepresentative
Belinda	Clarke	2001-01-05	1596
Belinda	Clarke	2000-01-27	667
Belinda	Clarke	2000-11-11	467
Belinda	Clarke	2001-02-04	195
…	…	…	…

ジョインとして作成できないサブクエリ

同様に、サブクエリは機能するが、ジョインは機能しないという場合があります。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。

ジョインの詳細については、「ジョイン：複数テーブルからのデータ検索」 341 ページを参照してください。

ネストされたサブクエリ

これまで見てきたように、サブクエリは、通常、クエリの WHERE 句か HAVING 句内にあります。サブクエリ自体に WHERE 句か HAVING 句、またはその両方が含まれることがあり、その結果サブクエリが別のサブクエリ内にある場合があります。別のサブクエリ内のサブクエリは、「ネストされたサブクエリ」と呼ばれます。

例

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 をリストします。

```
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
2002	1
…	…

ネストされたサブクエリの説明

- この例では、最も内側のサブクエリが、"Fees" という説明がある財務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。

Copyright © 2007, iAnywhere Solutions, Inc.

```
SELECT ID, LineID
FROM SalesOrderItems
WHERE ShipDate = ANY ( subquery-expression );
```

ネストされたサブクエリのレベルは、3 つを超えることもできます。レベルの最大数はありませんが、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。

サブクエリ操作

クエリにサブクエリが含まれていると、どのクエリが有効でどのクエリが有効でないかを理解するのは難しい場合があります。同様に、マルチレベルのクエリが何を実行するかを理解するのも非常に複雑ですが、SQL Anywhere でサブクエリがどのように処理されるかを理解するうえで役に立ちます。クエリ処理の概要については、「クエリ結果の要約、グループ化、ソート」315 ページを参照してください。

相関サブクエリ

単純なクエリでは、データベース・サーバはクエリのローごとに一度、WHERE 句を評価して処理します。しかし場合によっては、サブクエリが 1 つの結果しか返さず、データベース・サーバが結果セット全体に対して 2 回以上評価する必要がないことがあります。

非相関サブクエリ

次のクエリを考えてみます。

```
SELECT Name, Description
FROM Products
WHERE Quantity <  2 * (
  SELECT AVG( Quantity )
  FROM SalesOrderItems );
```

この例では、サブクエリは 1 つの値、SalesOrderItems テーブルから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。

相関サブクエリ

サブクエリに外部参照が含まれていると、このような方法は使用できません。たとえば、次のクエリのサブクエリは、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。このようなサブクエリを相関サブクエリと呼びます。この場合、サブクエリは外部クエリのローごとに異なる値を返すことがあり、それによってデータベース・サーバが複数の評価を実行することが必要な場合があります。

```
SELECT Name, Description
FROM Products
WHERE Quantity < 2 * (
  SELECT AVG( Quantity )
  FROM SalesOrderItems
  WHERE Products.ID=SalesOrderItems.ProductID );
```

WHERE 句のサブクエリのジョインへの変換

SQL Anywhere のクエリ・オプティマイザは、一部のマルチレベルのクエリをジョインを使用するように変換します。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。

 Copyright © 2007, iAnywhere Solutions, Inc.

例

「Mrs. Clarke と Suresh がいつ注文し、どの担当者が注文を受けたか」という問い合わせは、2 つのレベルのクエリとして作成できます。

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID IN (
   SELECT ID
   FROM Customers
   WHERE Surname = 'Clarke' OR GivenName = 'Suresh' );
```

代替の、同等の方法では、ジョインを使用してクエリを作成します。

```
SELECT GivenName, Surname, OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
  ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

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*

サブクエリをジョインに変換できるかどうかは、演算子のタイプ、クエリの構造、サブクエリの構造など、さまざまな要素によって異なります。

比較演算子

比較演算子 (=、<>、<、<=、>、>=) に続くサブクエリは、ジョインに変換する場合、一定の条件を満たさなければなりません。たとえば比較演算子に続くサブクエリは、メイン・クエリのローごとに値を 1 つずつ返す場合は有効です。この基準に加えて、サブクエリが次の場合にはジョインに変換できます。

- 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

Copyright © 2007, iAnywhere Solutions, Inc.

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 );
```

限定比較テスト

キーワード ALL、ANY、SOME のいずれかに続くサブクエリは、一定の条件を満たす場合にだけ、ジョインに変換できます。

- 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、サブクエリのフォームで処理できます。

Copyright © 2007, iAnywhere Solutions, Inc.

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID = ANY (
  SELECT ID
  FROM Customers
  WHERE Surname = 'Clarke' OR GivenName = 'Suresh' );
```

または、ジョインのフォームで表現できます。

```
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、いつ注文したか」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。次のフォーム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
=	<>
<	=>
>	=<
=<	>
=>	<
<>	=

セット・メンバシップ・テスト

キーワード IN に続くサブクエリを含むクエリは、次の条件を満たす場合にだけジョインに変換されます。

Copyright © 2007, iAnywhere Solutions, Inc.

- 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

場合によっては、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、それに応じてサブクエリをジョインに変換するかどうかを決定します。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' ) );
```

存在テスト

キーワード EXISTS に続くサブクエリは、次の 2 つの条件を満たす場合にだけジョインに変換されます。

- 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

 Copyright © 2007, iAnywhere Solutions, Inc.

テーブルからのデータがあるロー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 );
```

第 13 章

データの追加、変更、削除

目次

データ修正文 494
INSERT によるデータの追加 499
UPDATE によるデータの変更 504
INSERT によるデータの変更 506
DELETE によるデータの削除 507

Copyright © 2007, iAnywhere Solutions, Inc.

データ修正文

データの追加、変更、削除に使う文を「**データ修正文**」といいます。最も一般的な文は、次のようなものです。

- **INSERT**　テーブルに新規のローを追加。
- **更新**　テーブルにある既存のローを変更。
- **DELETE**　テーブルから特定のローを削除。

単一の INSERT、UPDATE、または DELETE 文は、いずれも 1 つのテーブルまたはビューのデータだけを変更します。

一般的な文以外に、LOAD TABLE、TRUNCATE TABLE 文は、特にデータのバルク・ロードと削除に便利です。

データ修正文はまとめて、SQL の「**データ修正言語 (DML)**」部分として知られています。

データ修正のパーミッション

修正するデータベース・テーブルに適切なパーミッションがある場合だけ、データの修正文を実行できます。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、だれがどのデータ修正機能にアクセスするかを決定します。

パーミッションを個人ユーザ、グループ、または PUBLIC グループに付与できます。パーミッションの詳細については、「ユーザ ID とパーミッションの管理」『SQL Anywhere サーバ - データベース管理』を参照してください。

トランザクションとデータ修正

データを修正すると、各データ修正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。これにより、トランザクションを開始した場合、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。「トランザクションと独立性レベル」 119 ページを参照してください。

変更の確定

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。

Copyright © 2007, iAnywhere Solutions, Inc.

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。

Interactive SQL オプションの詳細については、「Interactive SQL 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。「COMMIT 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

ROLLBACK 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。「ROLLBACK 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

トランザクションとデータ・リカバリ

システム障害や停電によって突然データベース・サーバがダウンしたと仮定します。SQL Anywhere は、そのような状況でもデータベースの整合性を保護するように入念に設計されています。データベースをリストアする独自の方法がいくつも用意されています。たとえば、「**ログ・ファイル**」を別のドライブに保管し、1 つのドライブがシステム障害を起こしても、データをリストアできます。また、ログ・ファイルを使用すると、SQL Anywhere は頻繁にデータベースを更新する必要がなくなるため、データベースのパフォーマンスが向上します。

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。

データ・リカバリ・メカニズムの詳細については、「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』を参照してください。

整合性の検査

SQL Anywhere はデータ内の一般的なエラーについて自動的に検査を行います。

重複データの挿入

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。

これを行うには、次のコマンドを入力します。

```
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 カラムに基づく 1 対多です。プライマリ・テーブル (Employees) にレコードが入力されていないかぎり、外部テーブル (SalesOrders) に対応するレコードを挿入できません。

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。これは、SalesOrders テーブル内にあるそれぞれの営業担当者 ID 番号が、Employees テーブル内にある従業員の従業員 ID 番号と必ず一致することを意味します。

営業担当者 ID 番号 284 に対して受注を追加しようとすると、次のようなエラー・メッセージが表示されます。

SalesOrders テーブルの外部キー FK_SalesRepresentative_EmployeeID にはプライマリ・キー値がありません。

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。このエラーによって、有効な営業担当者 ID を持たない受注の挿入が禁止されます。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間での参照の整合性が維持されていることを検査するので、「**参照整合性**」検査と呼ばれます。

Copyright © 2007, iAnywhere Solutions, Inc.

プライマリ・キーと外部キーの詳細については、「テーブル間の関係」『SQL Anywhere 10 - 紹介』を参照してください。

DELETE 文または UPDATE 文でのエラー

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。Departments テーブルのプライマリ・キーである DepartmentID フィールドは 1 対多の関係の「1」の側を構成します (Employees テーブルの DepartmentID フィールドは対応する外部キーであり、1 対多の関係の「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。

例：DELETE エラー

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次のような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。

テーブル 'Departments' のプライマリ・キーは他のテーブルから参照されています。

R&D 部を削除するには、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 部を削除できます。

次の ROLLBACK 文を入力して、(今後の使用のために) 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;
```

最後に実行した COMMIT 文以降に行った変更は、すべて取り消されます。COMMIT 文を実行していない場合は、Interactive SQL の起動以降に行った変更がすべて取り消されます。

例：UPDATE 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;
```

「**テーブル 'Employees' の外部キー 'FK_DepartmentID_DepartmentID' に対応するプライマリ・キーの値がありません**」というエラーメッセージが表示されるのは、DepartmentID が 600 である部署が Departments テーブルにないからです。

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

前述のすべての例で、それぞれのコマンド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。「wait_for_commit オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

また、プライマリ・キーに行われ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。「データ整合性の確保」95 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

INSERT によるデータの追加

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 つのフォームがあります。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*

INSERT 文の詳細については、「INSERT 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ローの全カラムへの値の挿入

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 Departments
VALUES ( 702, 'Eastern Sales', 902 );
```

注意

- 元の CREATE TABLE 文にあるカラム名と同じように、ID 番号、名前、部長 ID の順で値を入力します。
- 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。

Copyright © 2007, iAnywhere Solutions, Inc.

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。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。

たとえば DepartmentID と DepartmentName の 2 つのカラムだけにデータを追加するには、次のような文にします。

```
INSERT INTO Departments ( DepartmentID, DepartmentName )
VALUES ( 703, 'Western Sales' );
```

ROLLBACK 文を実行して挿入を取り消します。

DepartmentHeadID カラムにはデフォルトがありませんが、NULL を使用できます。このカラムには NULL が割り当てられます。

カラム名をリストする順番は、値をリストする順番と一致させます。次の例は、前の例と同じ結果になります。

```
INSERT INTO Departments ( DepartmentName, DepartmentID )
VALUES ( 'Western Sales', 703 );
```

ROLLBACK 文を実行して挿入を取り消します。

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 入力可能	null 入力不可	null 入力可能 (DEFAULT 指定)	null 入力不可 (DEFAULT 指定)	null 入力不可 (DEFAULT AUTOINCREMENT 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。「allow_nulls_by_default オプション [互換性]」『SQL Anywhere サーバ - データベース管理』と「ALTER TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

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。制約はそのデータの種類によって、追加の可否を決定できます。

制約の詳細については、「テーブル制約とカラム制約の使い方」 106 ページを参照してください。

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。

デフォルトの詳細については、「カラム・デフォルトの使い方」 100 ページを参照してください。

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、更新を使用して他のカラムの値を追加できます。

値が挿入されていないカラムにデフォルトがあるか、または NULL が指定されているかどうかを確認してから、テーブルにある一部のカラム (すべてのカラムではない) に値を挿入します。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。

例

カラムが CREATE TABLE 文で同じ順序になっている場合、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。Products テーブルにあるのと同じフォーマットの製品情報を持つローが、NewProducts というテーブル内にいくつか含まれているとします。Products に NewProducts のすべてのローを追加するには、次の手順に従います。

```
INSERT Products
SELECT *
FROM NewProducts;
```

INSERT 文中の SELECT 文に式を使用できます。

Copyright © 2007, iAnywhere Solutions, Inc.

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、ローにある一部のカラム (すべてのカラムではない) に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';
```

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場合は、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できます。

アプリケーションへの INSERT 文の追加の詳細については、「準備文の使用方法」『SQL Anywhere サーバ - プログラミング』と「SET 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

テーブルへのファイル内容の挿入には、xp_read_file システム関数も使用できます。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、この関数を使用すると便利です。

この関数を使用するには DBA 権限が必要です。

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. いくつかの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' ) );
```

 Copyright © 2007, iAnywhere Solutions, Inc.

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』
- 「データベースへの BLOB の格納」 25 ページ
- 「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』

UPDATE によるデータの変更

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。UPDATE 文には、テーブル名やビュー名が続きます。すべてのデータ修正文と同様、一度に変更できるのは単一のテーブルまたはビュー内のデータだけです。

UPDATE 文は、変更するローまたは新しいデータを指定します。新しいデータは、指定する定数か式、または他のテーブルから引き出したデータです。

UPDATE 文が整合性制約に違反すると、更新は行われずにエラー・メッセージが表示されます。たとえば、追加された値の 1 つが誤ったデータ型であったり、カラムやデータ型のいずれかに定義された制約に違反した場合、更新は行われません。

UPDATE 構文

UPDATE 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。
詳細については、「Interactive SQL での結果セットの編集」『SQL Anywhere サーバ - データベース管理』を参照してください。

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

SET 句では、データ型が正しければどんな式でも使用できます。

WHERE 句

WHERE 句で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';
```

FROM 句

FROM 句を使用して、1 つ以上のテーブルから更新するテーブルにデータを引き出せます。

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。

詳細については、「INSERT 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

DELETE によるデータの削除

DELETE 文は、次のような単純なフォームになっています。

DELETE [**FROM**] *table-name*
WHERE *column-name* = *expression*

次のような、より複雑なフォームも使用できます。

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。

詳細については、「DELETE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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;
```

例を完了したら、変更をロールバックしてください。

Copyright © 2007, iAnywhere Solutions, Inc.

```
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。
> 詳細については、「Interactive SQL での結果セットの編集」『SQL Anywhere サーバ - データベース管理』を参照してください。

テーブルから全ローを削除

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。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

第 14 章

クエリの最適化と実行

目次

クエリ処理のフェーズ 510
セマンティック・クエリ変形 512
オプティマイザの仕組み 523
クエリ実行アルゴリズム 546
実行プランの解釈 571
物理データの編成とアクセス 590

Copyright © 2007, iAnywhere Solutions, Inc.

クエリ処理のフェーズ

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。各クエリが解析されるたびに、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。

次のリストに、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、その他の最適化手順について説明します。

- **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)、パーミッションのチェック、定義済み参照整合性制約を使用した KEY JOIN と NATURAL JOIN 変換、非実体化ビュー (Non-materialized View) 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。

- **クエリ書き換えフェーズ**　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、述部の書き換えなど) 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。

- **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。

  - **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、クエリを解析し、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実体化ビュー (Materialized View) のすべてを検出します。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、クエリの一部またはすべてを満たすために使用可能なすべての実体化ビュー (Materialized View) が決定されます。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] タブでは、プランのグラフィカルなバージョンを確認できます ([ツール] – [オプション] – [プラン] でいずれかのグラフィカルなプランが選択される必要があります)。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。「実行プランの解釈」 571 ページを参照してください。
- **実行フェーズ**　前のフェーズで構築されたクエリ実行プランを使用して、クエリの結果が計算されます。

結果セットのない文 (UPDATE 文や DELETE 文など) にも、クエリ実行プランはあります。

最適化をバイパスするクエリ

ほとんどすべての文が前述のフェーズを経由しますが、オプティマイザをバイパスする単純なクエリと、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の 2 つは例外です。「最適化をバイパスするクエリ」 511 ページを参照してください。

クエリが単純な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。最適化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。単純なクエリとは、単一のカラムに対して、サブクエリ、複数のテーブル、プロキシ・テーブル、ユーザ定義関数、NUMBER(*)、UNION、集合関数、DISTINCT、GROUP BY、または複数の述部を含まない DYNAMIC SCROLL カーソルまたは NO SCROLL カーソルです。単純なクエリに ORDER BY を使用できるのは、WHERE 句に各プライマリ・キー・カラムの条件がある場合だけです。

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。「プランのキャッシュ」 544 ページを参照してください。

SELECT 文の OPTION FORCE OPTIMIZATION 句を使用することで、データベース・サーバで単純なクエリを最適化したり、クエリを含むストアド・プロシージャやユーザ定義関数を最適化できます。この句を指定した場合、プランのキャッシュ処理は使用されません。「SELECT 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

セマンティック・クエリ変形

効率的に操作するために、通常 SQL Anywhere はクエリを書き換え、場合によってはいくつかの手順を実行して、新しいフォームにします。Adaptive Server Anywhere は、クエリの表現を変えても、その新しいバージョンで同じ結果が計算されることを保証します。つまり、SQL Anywhere は、クエリをセマンティック上は等しく、構文上は異なるフォームに書き換えます。

SQL Anywhere は、さまざまな書き換え操作を実行できます。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、オプティマイザは可能なかぎりジョインを使ってサブクエリを書き換えようとします。オプティマイザは SQL 文をより効率的にするために書き換えるという点に注意してください。

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。「REWRITE 関数 [その他]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

例

SQL 言語の定義とは異なり、AND と OR の演算の厳密な動作を要求する言語もあります。言語によっては、左側の条件が最初に評価されることを指定しているものもあります。条件全体が真であると決定されると、コンパイラは右側の条件が評価されないよう指定します。

この調整により、これ以外の場合には 2 つのネストされた IF 文を必要とする条件を、1 つに組み合わせることができます。たとえば、C でポインタが NULL であるかどうかをテストしてから、これを次のように使用できます。最初の文でネストされた条件は、次の 2 番目の文で示される構文を使用して置き換えることができます。

```
if ( X != NULL ) {
   if ( X->var != 0 ) {
      ... statements ...
   }
}

if ( X != NULL && X->var != 0 ) {
      ... statements ...
}
```

C とは異なり、SQL には実行順序に関するこのような規則がありません。SQL Anywhere は、適合すると判断した場合、条件の順序を自由に再調整できます。SQL 言語の仕様では順序の違いが区別されないため、元のフォームも順序が変わったフォームもセマンティック上は等しくなります。特に、クエリ・オプティマイザは、WHERE 句、HAVING 句、ON 句の述部の順序を完全に自由に変更できます。

セマンティック変形の種類

クエリ書き換えフェーズで SQL Anywhere は、クエリのより効率的で適切な表現方法を探すため、さまざまな変形を実行します。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、元のクエリのリテラルな解釈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。

Copyright © 2007, iAnywhere Solutions, Inc.

- 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 化されたビューと抽出テーブルでの述部のプッシュダウン
- OR 述部と IN リスト述部の最適化
- LIKE 述部の最適化
- 外部ジョインから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

不要な DISTINCT 条件の排除

DISTINCT 条件が不要な場合があります。たとえば、結果内の 1 つまたは複数のカラムがプライマリ・キーであるため、そのプロパティに UNIQUE 条件が明示的または暗黙的に含まれているような場合です。

例

次のコマンドでは、Products テーブルにプライマリ・キー p.ID があって結果セットの一部になっているので、DISTINCT キーワードは不要です。

```
SELECT DISTINCT p.ID, p.Quantity
FROM Products p;
```

Products<seq>

データベース・サーバは、セマンティック上等しいクエリを実行します。

```
SELECT p.ID, p.Quantity
FROM Products p;
```

同様に、以下のクエリの結果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。そのためデータベース・サーバは、結果セットで DISTINCT を実行せずに、このクエリを実行します。

```
SELECT DISTINCT *
FROM SalesOrders o JOIN Customers c
  ON o.CustomerID = c.ID
WHERE c.State = 'NY';
```

Work[HF[c<seq>] *JH o<seq>]

サブクエリのネスト解除

SQL 言語で適当な構文が指定されている場合、文をネストされたクエリとして表すことができます。ただし、通常、ネストされたクエリをジョインとして書き換えると、SQL Anywhere がサブクエリの WHERE 句にある高度な選択条件をうまく利用できるようになるため、文の実行がさらに効率的になり、最適化もさらに効果的なものとなります。一般に、サブクエリのネスト解除は、FROM 句で多くても 1 テーブルを使用する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・セマンティクスに基づいてサブクエリが多くてもローを 1 つ返すことが判断できる場合は、相関でないサブクエリ、または FROM 句で複数のテーブルを使用するサブクエリはフラットにされます。

例

次の例に示すサブクエリでは、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、SQL Anywhere は、これを内部ジョインに変換できるものとみなします。

```
SELECT s.*
FROM SalesOrderItems s
WHERE EXISTS
   ( SELECT *
     FROM Products p
     WHERE s.ProductID = p.ID
        AND p.ID = 300 AND p.Quantity > 20);
```

変換後、同じ文がジョイン構文を使用して、内部的に表現されます。

```
SELECT s.*
FROM Products p JOIN SalesOrderItems s
   ON p.ID = s.ProductID
WHERE p.ID = 300 AND p.Quantity > 20;
```

```
p<Products> JNL s<FK_ProductID_ID>
```

同様に、次に示すクエリのサブクエリには、結合 EXISTS 述部があります。このサブクエリでは、1 つ以上のローを一致させることができます。

```
SELECT p.*
FROM Products p
WHERE EXISTS
   ( SELECT *
     FROM SalesOrderItems s
     WHERE s.ProductID = p.ID
        AND s.ID = 2001);
```

SQL Anywhere は、SELECT リストに DISTINCT を使って、このクエリを内部ジョインに変換します。

```
SELECT DISTINCT p.*
FROM Products p JOIN SalesOrderItems s
   ON p.ID = s.ProductID
WHERE s.ID = 2001;
```

```
Work[ DistH[ s<FK_ID_ID> JNL p<Products> ] ]
```

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、SQL Anywhere は、比較しているサブクエリを削除することもできます。これは次のようなクエリで行われます。

Copyright © 2007, iAnywhere Solutions, Inc.

```
SELECT *
FROM Products p
WHERE p.ID =
   ( SELECT s.ProductID
     FROM SalesOrderItems s
     WHERE s.ID = 2001
        AND s.LineID = 1 );
```

SQL Anywhere は、このクエリを次のように書き換えます。

```
SELECT p.*
FROM Products p, SalesOrderItems s
WHERE p.ID = s.ProductID
  AND s.ID = 2001
  AND s.LineID = 1;
```

s<SalesOrderItems> JNL p<Products>

サブクエリのネストを解除するリライト最適化の実行中は、DUMMY テーブルは特殊テーブルとして扱われます。サブクエリをフラットにする処理は、それが相関サブクエリではない場合でも、常に SELECT *expression* FROM DUMMY 形式のサブクエリに対して実行されます。

UNION または GROUP 化されたビューと抽出テーブルでの述部のプッシュダウン

少数のレコードだけが返されるようにビューの結果を制限するのは、クエリでは一般的です。ビューに GROUP BY または UNION がある場合、データベース・サーバにとって望ましいのは対象のローの結果だけを計算することです。述部のプッシュダウンは、述部が単一ビューまたは抽出テーブルのカラムを排他的に参照する場合にかぎり、述部に対して実行されます。たとえばジョイン述部はビューにプッシュダウンされません。

例

たとえば、ビュー ProductSummary が次のように定義されているとします。

```
CREATE VIEW ProductSummary( ID,
      NumberOfOrders,
      TotalQuantity) AS
SELECT ProductID, count(*), sum( Quantity )
FROM SalesOrderItems
GROUP BY ProductID;
```

ProductSummary ビューは、注文があった製品ごとに、含まれる注文の件数と、すべての注文の受注数量の合計を返します。ここで、このビューに対する次のクエリを考えてみます。

```
SELECT *
FROM ProductSummary
WHERE ID = 300;
```

このクエリは、ID カラムが 300 であるローだけに出力を制限します。このクエリと、ビューの定義におけるクエリを結合して、次のようにセマンティック上は等しい SELECT 文にすることができます。

```
SELECT ProductID, COUNT(*), SUM( Quantity )
FROM SalesOrderItems
GROUP BY ProductID
HAVING ProductID = 300;
```

Copyright © 2007, iAnywhere Solutions, Inc.

このクエリに対する単純な実行プランは、各製品の集合の計算を行い、その結果を製品 ID 300 に関する 1 つのローだけに制限します。ただし、ProductID カラムの HAVING 述部はグループ化カラムなので、次のようにクエリの WHERE 句にプッシュできます。

```
SELECT ProductID, COUNT(*), SUM( Quantity )
FROM SalesOrderItems
WHERE ProductID = 300
GROUP BY ProductID;
```

この SELECT 文によって、必要な計算が大幅に減少します。この述部に十分な選択性があれば、オプティマイザは ProductID のインデックスを使用して製品 300 のローだけを取り出すことができます。SalesOrderItems テーブルの逐次スキャンは行いません。

これと同じ最適化は、UNION または UNION ALL を含むビューにも使用されます。

OR 述部と IN リスト述部の最適化

オプティマイザは、インデックス・カラム上で IN 述部を利用する特殊な最適化をサポートしています。この最適化は、同じインデックス・カラムに対して OR で結合された複数の述部にも等しく適用されます。これは、どちらもセマンティック上は等しいためです。最適化を有効にするには、IN リストに定数だけを指定します。

オプティマイザは、修飾する IN リスト述部を検出した場合に、IN リスト述部にインデックス検索を十分に検討できる選択性がある場合には、IN リスト述部をネスト・ループ・ジョインに変換します。この最適化の機能を、次の例で説明します。

たとえば次のクエリがあるとします。これは、2 人の営業担当者が扱うすべての注文をリストします。

```
SELECT *
FROM SalesOrders
WHERE SalesRepresentative = 902 OR SalesRepresentative = 195;
```

このクエリは、セマンティック上は次に示すものと同じです。

```
SELECT *
FROM SalesOrders
WHERE SalesRepresentative IN (195, 902);
```

オプティマイザは、IN リスト述部の結合選択性がインデックス検索を保証できる程度に低いものと推定します。その結果、IN リストを仮想テーブルとして扱い、この仮想テーブルを SalesRepresentative 属性で SalesOrders テーブルにジョインします。この最適化の最終的な効果はアクセス・プランに追加のジョインを組み込むことですが、クエリのジョイン数は増加しないので、最適化時間には影響を与えません。

この最適化には、2 つの主要な利点があります。第 1 の利点は、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ることです。第 2 の利点は、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ることです。

上記のクエリのアクセス・プラン (省略形) は、次のとおりです。

```
SalesOrders<FK_SalesRepresentative_EmployeeID>
```


Copyright © 2007, iAnywhere Solutions, Inc.

LIKE 述部の最適化

LIKE 述部には、ほとんどの場合にリテラル定数やホスト変数がパターンとして使用されます。パターンによっては、オプティマイザが LIKE 述部全体を書き換えたり、対応するテーブルに対するインデックス検索の実行に利用できる追加条件を追加したりする場合があります。LIKE 述部の追加条件では LIKE_PREFIX 述部が使用されます。LIKE_PREFIX 述部はクエリで直接指定することはできませんが、クエリ・オプティマイザによって最適化が適用できるときに長いプランやグラフィカルなプランに表示されます。

例

次のそれぞれの例で、LIKE 述部のパターンがリテラル定数またはホスト変数であり、X がベース・テーブルのカラムであると仮定します。

- X LIKE '%' は X IS NOT NULL と書き換えられる。
- X LIKE 'abc%' は LIKE_PREFIX 述部に拡大される。これは検索引数可能 (インデックスの検索に使用可能) な述部で、X の値が abc で始まる必要があるという条件を適用します。LIKE_PREFIX 述部では、マルチバイト文字セットと、ブランクが埋め込まれたデータベースで正しいセマンティックが適用されます。

外部ジョインから内部ジョインへの変換

オプティマイザはアクセス・プラン用に左側が深い処理ツリーを生成します。このルールの唯一の例外は、右側が深いネスト外部ジョイン式の存在です。LEFT OUTER JOIN または RIGHT OUTER JOIN の計算に使用するクエリ実行エンジンのアルゴリズムでは、どのようなジョイン方式の場合も、保護されたテーブルを NULL 入力テーブルよりも前に置きます。そこで、オプティマイザはできるだけ LEFT または RIGHT 外部ジョインを内部ジョインに変換しようとします。これは、内部ジョインは交換可能で、オプティマイザがジョイン列挙を実行するときの自由度が大きくなるためです。

NULL 入力テーブルに対する NULL を許容しない述部がクエリの WHERE 句にある場合は、LEFT OUTER JOIN または RIGHT OUTER JOIN を内部ジョインに変換できます。この述部は NULL を許容しないため、外部ジョインによって生成されるすべて NULL のローは結果から省かれます。その結果、このクエリはセマンティック上は内部ジョインと等しくなります。

例

たとえば、数量の多い製品すべてとその注文をリストするクエリがあるとします。LEFT OUTER JOIN は、注文がなくても全製品をリストするためのものです。

```
SELECT *
FROM Products p KEY LEFT OUTER JOIN SalesOrderItems s
WHERE s.Quantity > 15;
```

このクエリの問題は、s.Quantity が NULL の場合に WHERE 句の述部 s.Quantity > 15 が FALSE として解釈されるため、注文のない製品がこの述部によって結果から削除されることです。そのため、このクエリは、セマンティック上は次に示すものと同じです。

```
SELECT *
FROM Products p KEY JOIN SalesOrderItems s
WHERE s.Quantity > 15;
```

データベース・サーバが最適化するのも、クエリを書き換えたこのフォームになります。

この例で、クエリが正しく書き換えられないのはほぼ確実です。正しくは、次のようにする必要があります。

```
SELECT *
FROM Products p
   KEY LEFT OUTER JOIN SalesOrderItems s
      ON s.Quantity > 15;
```

この場合、Quantity のテストは外部ジョイン条件の一部になっています。2 つのクエリの相違を示すために、順序のない Products テーブルに新しい製品を挿入してから、クエリを再実行してみます。

```
INSERT INTO Products
SELECT ID + 10, Name, Description,
      'Extra large', Color, 50, UnitPrice, Photo
FROM Products
WHERE Name = 'Tee Shirt';
```

この最適化が単純な外部ジョイン・クエリに適用されるのはまれで、通常はクエリが外部ジョインを使用して書き換えられる 1 つ以上のビューを参照する場合に適用できます。クエリの WHERE 句には、1 つ以上のテーブル式からの NULL 入力ローをすべて削除するようにビューの出力を制限する条件を含めることができるため、この最適化が適用可能になります。

不要な内部ジョインと外部ジョインの削除

ジョインの削除によるリライト最適化では、問題がない場合はクエリからテーブルが削除され、クエリのジョイン数が減少します。通常、この最適化はプライマリ・キーから外部キーへのジョインまたはプライマリ・キーからプライマリ・キーへのジョインとして定義された内部ジョインに対して適用されます。ジョインの削除による最適化は、外部ジョインで使用されるテーブルにも適用できますが、最適化が有効となる条件はかなり複雑になります。

この最適化では、UPDATE または DELETE WHERE CURRENT を使用して更新可能なテーブルは、削除が適切である場合でも削除されません。このため、クエリのパフォーマンスに悪影響がある可能性があります。ただし、クエリが読み込み専用である場合は、SELECT 文で FOR READ ONLY を指定することで、ジョインの削除が実行されます。メインのクエリ・ブロック内のテーブルが更新可能であっても、サブクエリに現れるテーブルやネストされた抽出テーブルは、本来は更新可能ではありません。

要約すると、このリライト最適化が適用されるジョインには 3 つのメイン・カテゴリがあります。

- 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から参照される場合。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、テーブルが更新可能でないと、一方のテーブルが削除されます。
- ジョインが外部ジョインで、NULL 入力テーブル式に次の性質がある場合。

 Copyright © 2007, iAnywhere Solutions, Inc.

- 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 つのローを返す。
- 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。

例

たとえば、次のクエリで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。

```
SELECT s.ID, s.LineID, p.ID
FROM SalesOrderItems s KEY JOIN Products p
FOR READ ONLY;
```

クエリは、次のように書き換えられます。

```
SELECT s.ID, s.LineID, s.ProductID
FROM SalesOrderItems s
WHERE s.ProductID IS NOT NULL
FOR READ ONLY;
```

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。

次のクエリでは、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、かつ LEFT OUTER JOIN を超えて Products のカラムが使用されない場合に、OUTER JOIN を削除できます。

```
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;
```

クエリは、次のように書き換えられます。

```
SELECT s.ID, s.LineID
FROM SalesOrderItems s
WHERE s.Quantity > 5
FOR READ ONLY;
```

述部の推定による利用可能な条件の発見

ほとんどすべてのクエリで効率的なアクセス方式は、WHERE 句、ON 句、HAVING 句に検索引数可能な条件が存在するかどうかによります。インデックス検索は、マッチング述部に検索引数可能な条件を利用することによってのみ可能になります。また、ハッシュ、マージ、ブロックの各ネスト・ループ・ジョインを使用できるのは、等価ジョイン条件がある場合だけです。このような理由から、SQL Anywhere は、オプティマイザが利用できる簡略または暗黙的な条件を発見するために、オリジナルのクエリ・テキスト内で探索条件を詳細に分析します。

前処理として、ビューが拡張され、マージされてから、オリジナルの文の述部に対していくつかの簡略化が行われます。次に例を示します。

- X = X は、X が NULL 入力可能な場合は X IS NOT NULL に書き換えられ、それ以外の場合は述部は削除される
- ISNULL(X,X) は単に X に書き換えられる
- X+0 は、X が数値カラムの場合は X に書き換えられる
- AND 1=1 は削除される
- OR 1=0 は削除される
- 単一要素で構成される IN リスト述部は、単純な等号条件に変換される

この前処理の後に、SQL Anywhere はオリジナルの探索条件を論理積正規形 (CNF: conjunctive normal form) に正規化しようとします。式を CNF に正規化するために、式の各項が AND で結合されます。各項は、単一のアトミックな条件、または OR で結合された一連の条件で構成されます。

任意の条件を CNF に変換すると、複雑さは同じでも、かなり多数の条件セットを持つ式になります。SQL Anywhere はこの状況を認識し、条件を単純に CNF に変換するのを控えます。代わりに、オリジナルの探索条件によって暗黙的に指定された利用可能な述部のオリジナルの式を分析し、推定されたこれらの条件をクエリに AND します。完全正規化も、高コストの述部 (限定サブクエリ述部など) の重複を必要とする場合は退けられます。ただし、このアルゴリズムでは、可能な場合はいつでも IN リスト述部がマージされます。

探索条件が完全に正規化されるか、利用可能な条件が検出されると、オプティマイザは中間的分析を実行し、中間的等号条件、主として中間的ジョイン条件と定数を伴う条件を発見します。これによって、このような中間的条件では追加の代替ジョイン順が許可される場合があるため、オプティマイザはコストベースの最適化フェーズ時にジョイン列挙を実行する自由度を高めます。

例

元のクエリを次に示します。

```
SELECT e.Surname, s.ID, s.OrderDate
FROM SalesOrders s, Employees e
WHERE
  ( e.EmployeeID = s.SalesRepresentative AND
    ( s.SalesRepresentative = 142 OR
      s.SalesRepresentative = 1596 )
  ) OR (
     e.EmployeeID = s.SalesRepresentative AND
     s.CustomerID = 667 );
```

このクエリには結合等価ジョイン条件がありません。したがって、オプティマイザは述部の詳細分析を実行しなければ、効率的なアクセス・プランを発見できません。幸い、SQL Anywhere は式全体を CNF に変換して同じクエリを生成できます。

```
SELECT e.Surname, s.ID, s.OrderDate
FROM SalesOrders s, Employees e
WHERE
   e.EmployeeID = s.SalesRepresentative AND
   ( s.SalesRepresentative = 142 OR
     s.SalesRepresentative = 1596 OR
     s.CustomerID = 667 );
```

このクエリは、内部ジョイン・クエリとして効率的に最適化できます。

Copyright © 2007, iAnywhere Solutions, Inc.

大文字と小文字の不要な変換の排除

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文字列比較をサポートしています。オプティマイザは、テキスト変換が不要な場合に、ユーザが UPPER、UCASE、LOWER、LCASE 組み込み関数を使用して、このような変換を明示的に強制しているクエリを検出することがあります。SQL Anywhere は、データベースの照合順で許可されている場合は、この不要な変換を自動的に排除します。述部の大文字と小文字の変換を排除することにより、これらの述部の一部を検索引数可能な述部に変換できる利点があります。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に使用できます。

例

次のクエリを考えてみます。

```
SELECT *
FROM Customers
WHERE UPPER(Surname) = 'SMITH';
```

大文字と小文字を区別しないデータベースでは、このクエリは内部的には次のように書き換えられます。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。

```
SELECT *
FROM Customers
WHERE Surname = 'SMITH';
```

サブクエリを EXISTS 述部として書き換える

SQL Anywhere の設計の基本となる仮定条件では、メモリを大事に使用し、デフォルトでできるだけ速やかにカーソルの最初の 2、3 の結果を返すことが要求されています。SQL Anywhere は、これらの目的を遂行するため、このような書き換えがセマンティック上正しい場合は、IN、ANY、SOME 述部などの集合演算サブクエリすべてを EXISTS 述部または NOT EXISTS 述部として書き換えます。これにより、SQL Anywhere は不要なワーク・テーブルを作成しないで、テーブルにアクセスするための適切なインデックスをより簡単に識別できるようになります。

非相関サブクエリと相関サブクエリ

非相関サブクエリは、クエリの残りのさらに高いレベルに格納されている 1 つまたは複数のテーブルへの明示的な参照を持たないサブクエリです。

次は、非相関サブクエリを持つ通常のクエリを示しています。このクエリは、2001 年 1 月 1 日に注文していないすべての顧客に関する情報を選択します。

```
SELECT *
FROM Customers c
WHERE c.ID NOT IN
  ( SELECT o.CustomerID
    FROM SalesOrders o
    WHERE o.OrderDate = '2001-01-01' );
```

このクエリに対する考えられる評価方法の 1 つとしては、まず 2001 年 1 月 1 日に注文した、SalesOrders テーブル内のすべての顧客のワーク・テーブルを作成します。次に、Customers テー

ブルに問い合わせて、ワーク・テーブルにリストされている顧客ごとに 1 つのローを抽出します。

ただし、SQL Anywhere は結果をワーク・テーブルとして実体化することを避けます。また、結果の最初の 2、3 のローを最も速く返すプランを優先します。したがって、オプティマイザは、NOT EXISTS 述部を使用して、このようなクエリを書き換えます。このフォームでは、サブクエリが「相関」サブクエリになり、Customers テーブルの ID カラムへの明示的な外部参照を持つようになります。

```
SELECT *
FROM Customers c
WHERE NOT EXISTS
  ( SELECT *
    FROM SalesOrders o
    WHERE o.OrderDate = '2000-01-01'
      AND o.CustomerID = c.ID );
```

このクエリは、上記のクエリとセマンティック上は等しくなりますが、この新しい構文で表すと、次の利点が明らかになります。

1. オプティマイザは、SalesOrders テーブルの CustomerID 属性または OrderDate 属性のインデックスを選択して使用できます (ただし、SQL Anywhere サンプル・データベースでは、ID カラムと CustomerID カラムに対してのみインデックスが作成されています)。

2. オプティマイザは、中間結果をワーク・テーブルに実体化しないでサブクエリを評価することもできます。

3. データベース・サーバは、実行中に相関サブクエリの結果をキャッシュできます。これにより、以前に計算されたこの述部の値を外部参照 c.ID の同じ値で再利用できます。上記のクエリの場合、顧客の ID 番号が Customers テーブル内でユニークであるため、キャッシュは役に立ちません。そのためサブクエリは外部参照 c.ID が常に異なる値で計算されます。

サブクエリ・キャッシュの詳細については、「サブクエリと関数のキャッシュ」 566 ページを参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

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実体化ビュー (Materialized View) 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 では、[結果] ウィンドウ枠の [プラン] タブをクリックして、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。表示される詳細の程度を変更するには、([ツール] メニューから) [オプション] ダイアログを開いて、[プラン] タブの設定を変更します。「グラフィカルなプラン」 240 ページと「実行プランの解釈」 571 ページを参照してください。

コストベース

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。「ALTER DATABASE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。「optimization_goal オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

構文に依存しない

ほとんどのコマンドは、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等しくなりますが、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。クエリに非実体化ビュー (Non-materialized View) 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。

良いプランが必ずしも最高のプランではない

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。「optimization_level オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

コストがかかる複雑なクエリの場合や、高い最適化レベルを設定した場合、オプティマイザの動作時間が長くなります。非常に大きなコストがかかるクエリの場合は、はっきりとした遅延が生じるほどクエリの実行時間が長くなることがあります。

オプティマイザの推定とカラム統計

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。「カラム統計の更新」 526 ページを参照してください。

カラム統計

オプティマイザが使用するカラム統計のもっと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。

 Copyright © 2007, iAnywhere Solutions, Inc.

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。

ヒューリスティック

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。

プロシージャ統計

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、ProcCall アルゴリズムによって、ISYSPROCEDURE システム・テーブルの stats カラムに格納されます。「SYSPROCEDURE システム・ビュー」『SQL Anywhere サーバ - SQL リファレンス』と「ProcCall アルゴリズム」 569 ページを参照してください。

参照

カラム統計の詳細については、「SYSCOLSTAT システム・ビュー」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

述部の選択性とカラム値の分散を取得する方法の詳細については、次を参照してください。

- 「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』
- 「ESTIMATE 関数 [その他]」『SQL Anywhere サーバ - SQL リファレンス』

Copyright © 2007, iAnywhere Solutions, Inc.

- 「ESTIMATE_SOURCE 関数 [その他]」 『SQL Anywhere サーバ - SQL リファレンス』

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が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。これは、テーブルやカラムを参照する述部を満たす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、後続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 は古い統計だけを削除します。

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。

カラム統計の詳細については、次を参照してください。

- 「SYSCOLSTAT システム・ビュー」 『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』
- 「CREATE STATISTICS 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「update_statistics オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』
- 「collect_statistics_on_dml_updates オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』

パフォーマンスの自動チューニング

クエリで最も一般的な制約は、カラムの値の等価性です。たとえば、次に示すクエリは、Sex カラムの等号 (=) でテストします。

```
SELECT *
FROM Employees
WHERE Sex = 'f';
```

クエリは、2 回目の実行で異なった最適化を行うことがよくあります。上記の種類の制約に関して、SQL Anywhere は経験から学習し、値の異常な分散があるカラムに自動的に対処します。DROP STATISTICS コマンドを使用して明示的に削除しないかぎり、データベースにはこの情報が永続的に保存されます。後続のクエリにそのカラムへの述部があると、データベース・サーバ

 Copyright © 2007, iAnywhere Solutions, Inc.

がカラムのヒストグラムを再作成する場合があるので注意してください。「カラム統計の更新」 526 ページを参照してください。

基本となる仮定条件

SQL Anywhere クエリ・オプティマイザの設計方針と理念には、基本となるいくつかの仮定条件があります。オプティマイザの決定を理解することにより、独自のアプリケーションの質とパフォーマンスを向上させることができます。これらの仮定条件は、以降の各項で説明されている事柄を理解するための背景となります。

最小限の管理作業

従来、高性能のデータベース・サーバは、知識が豊富な専任のデータベース管理者の存在に大きく頼ってきました。このような管理者は、データベースの最適なパフォーマンスを獲得するため、あらゆる種類のデータ記憶領域やパフォーマンス制御の調整に多くの時間を割いていました。これらの制御では、通常、データベースのデータの変更に応じ、継続的な調整が必要でした。

SQL Anywhere は、データベースが成長して変化するのに応じて、学習と調整を行います。各クエリは、データベースでのデータ分散に関する知識を蓄積します。SQL Anywhere はこの情報を自動的に保管し、以降のクエリの最適化に使用します。

各クエリは、この内部知識に貢献するとともに、そこから恩恵を受けます。各ユーザは、SQL Anywhere が別のユーザのクエリを実行して得た知識の恩恵を受けることができます。

このように統計収集のメカニズムは、データベース・サーバにとって不可欠な部分であり、外部のメカニズムを必要としません。これが役立つ状況であると判断した場合、インデックス・ヒントをデータベース・サーバに提供できます。これらのヒントにより、最適化で特定のインデックスが使用され、それによって選択性推定を基にしたオプティマイザによる決定が上書きされます。これらの推定をトリガやプロシージャにエンコードする場合、必要な場合はいつでもヒントを更新してください。「カラム統計の更新」 526 ページと「インデックスの編集」 87 ページを参照してください。

最初のローの最適化または結果セット全体の最適化

optimization_goal オプションを使用すると、クエリ処理の最適化の目的が最初のローを迅速に返すことであるか、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であるか (デフォルトの動作) を指定できます。「optimization_goal オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

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

Copyright © 2007, iAnywhere Solutions, Inc.

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。

詳細については、「optimization_workload オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

統計が存在し、正確である

オプティマイザはセルフチューニングで、必要なすべての情報を内部的に格納しています。ISYSCOLSTAT システム・テーブルは、データ分散と述部選択性推定の永続的レポジトリです。各クエリが完了すると、SQL Anywhere は、クエリ実行中に収集された統計を使用して ISYSCOLSTAT を更新します。したがって、すべての後続クエリはより正確な推定にアクセスできます。

オプティマイザはこれらの統計に大きく依存しており、そのためオプティマイザが生成するアクセス・プランの質も、これらの統計に大きく依存することになります。最近になって新しいローを多数挿入した場合、これらの統計はもはやデータを正確に表していないことがあります。後続クエリの実行速度が著しく遅くなることもあります。

データを大幅に変更しており、クエリの実行が低速であることが判明した場合は、DROP STATISTICS や CREATE STATISTICS を実行します。「カラム統計の更新」 526 ページを参照してください。

インデックスを使用して述部を満たすことができる

通常、SQL Anywhere はインデックスを使用して検索引数可能な述部を評価できます。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、optimization_goal が first-row に設定されている場合、オプティマイザはインデックスを使用して ORDER BY 句と GROUP BY 句の指定を満たそうとします。「optimization_goal オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

適切なインデックスが見つからないと、オプティマイザは逐次テーブル・スキャンを行いますが、これにはコストがかかります。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ができます。一般的な要求を効率的に処理できるようになる場合には、インデックスをテーブルに追加するか、クエリを書き換えてください。

新しいインデックスを作成することでクエリのパフォーマンスを向上できるかどうかを調べるには、Sybase Central のアプリケーション・プロファイリング・モードで [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ください。

プロファイリングと述部分析の詳細については、「アプリケーション・プロファイリング」 202 ページと「述部の分析」 529 ページを参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

仮想メモリは貴重なリソースである

オペレーティング・システムと多くのアプリケーションは、一般的なコンピュータのメモリを頻繁に共有します。SQL Anywhere は、メモリを貴重なリソースとして扱います。SQL Anywhere はメモリを経済的に使用するので、比較的小さなコンピュータ上で実行できます。ポータブル・コンピュータや古い機種のコンピュータでデータベースを操作する場合に、この経済性は重要です。

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。バッファ・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。

SQL Anywhere はこの状況を認識し、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。

一方、パフォーマンスの向上のために、オプティマイザがメモリを使用するのは大切なことです。たとえば、サブクエリの結果がクエリの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。

述部の分析

「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 はこの文を効率的に評価するため、Employees テーブルのプライマリ・インデックスを使用します。

```
SELECT *
FROM Employees
WHERE Employees.EmployeeID = 102;
```

プランでは、これは Employees<Employees> のように表示されます。

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;
```

プランでは、これは Employees<seq> のように表示されます。

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';
```

関数を使用しないようにクエリを書き換えて、検索引数可能にできる場合があります。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) が整数の代わりに文字列を返した場合には、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。

計算カラムの詳細については、「計算カラムの使用」 56 ページを参照してください。

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	*x* IS NOT NULL

Copyright © 2007, iAnywhere Solutions, Inc.

検索引数可能	検索引数不可能
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

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。このフォームでは、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、たくさん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、ジョイン方式によって決まります。オプティマイザは、ジョイン方式の変更を検討する場合、マッチング述部の選択を再評価します。「述部の推定による利用可能な条件の発見」 519 ページを参照してください。

実体化ビュー (Materialized View) によるパフォーマンスの向上

実体化ビュー (Materialized View) とはビューの 1 種で、結果セットがディスクに格納される点はベース・テーブルとよく似ていて、計算される点はビューとよく似ています。概念としては、実体化ビュー (Materialized View) はビューでもあり (クエリ指定がある)、テーブルでもあります (永続的な実体化ローがある)。このため、テーブルに対して実行する多くの操作を実体化ビュー (Materialized View) に対しても実行できます。たとえば、実体化ビュー (Materialized View) に対して、インデックス構築やアンロードを実行できます。

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。実体化ビュー (Materialized View) 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。

実体化ビュー (Materialized View) の恩恵を受けるためにクエリを変更する必要はありません。たとえば実体化ビュー (Materialized View) は、基本となるデータが頻繁に変更されないようなデータ・ウェアハウス・アプリケーションでの使用に向いています。

Copyright © 2007, iAnywhere Solutions, Inc.

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実体化ビュー (Materialized View) 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実体化ビュー (Materialized View) 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) で行われれる推奨の対象にそのビューを含めます。実体化ビュー (Materialized View) 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。オプティマイザが実体化ビュー (Materialized View) 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。つまり、実体化ビュー (Materialized View) 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がありません。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。

実体化ビュー (Materialized View) 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、候補となる実体化ビュー (Materialized View) のそれぞれが検査されます。実体化ビュー (Materialized View) は、次の場合にビュー・マッチング・アルゴリズムによって使用することが検討されます。

- データベース・サーバで実体化ビュー (Materialized View) を使用できる。「実体化ビュー (Materialized View) の有効化と無効化」 80 ページを参照してください。
- 最適化で実体化ビュー (Materialized View) を使用できる。「オプティマイザによる実体化ビュー (Materialized View) の使用の有効化と無効化」 82 ページを参照してください。
- 実体化ビュー (Materialized View) が初期化されている。「実体化ビュー (Materialized View) の初期化」 77 ページを参照してください。
- 実体化ビュー (Materialized View) が検討におけるオプティマイザ要件のすべてを満たす。「ビュー・マッチング・アルゴリズムの要件」 533 ページを参照してください。
- 実体化ビュー (Materialized View) 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する。「実体化ビュー (Materialized View) を管理するときの制限」 74 ページを参照してください。
- 実体化ビュー (Materialized View) の最後のリフレッシュが、materialized_view_optimization データベース・オプションに設定された古さのスレッショルドを超えていない。「オプティマイザでの実体化ビュー (Materialized View) の古さ閾値の設定」 83 ページを参照してください。

実体化ビュー (Materialized View) 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実体化ビュー (Materialized View) を含めます。ただし、それにより最終的にその実体化ビュー (Materialized View) 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。たとえば実体化ビュー (Materialized View) を使用しない別のアクセス・プランの方がコストが安いと推定されると、クエリの結果を計算するのに適切であると考えられる実体化ビュー (Materialized View) であっても使用されない可能性があります。

現在の接続に対する実体化ビュー (Materialized View) 候補リストの決定

オプティマイザが検討する候補となるすべての実体化ビュー (Materialized View) のリストは、次のコマンドを実行することでいつでも取得できます。

```
SELECT * FROM sa_materialized_view_info( ) WHERE AvailForOptimization='Y';
```


Copyright © 2007, iAnywhere Solutions, Inc.

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実体化ビュー (Materialized View) の作成時のオプションとが一致しない場合、その実体化ビュー (Materialized View) は候補と見なされません。一致する必要のあるオプションのリストについては、「実体化ビュー (Materialized View) を管理するときの制限」 74 ページを参照してください。

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実体化ビュー (Materialized View) 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';
```

実体化ビュー (Materialized View) をオプティマイザが検討したかどうかを判断する

特定のクエリで使用される実体化ビュー (Materialized View) 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。「実行プランの解釈」 571 ページを参照してください。

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実体化ビュー (Materialized View) 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。このトレーシング・タイプの詳細については、「アプリケーション・プロファイリング」 202 ページと「トレーシング・レベルの選択」 218 ページを参照してください。

最適化の列挙フェーズの詳細については、「クエリ処理のフェーズ」 510 ページを参照してください。

> **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リフレッシュされた実体化ビュー (Materialized View) を検討しません。

ビュー・マッチング・アルゴリズムの要件

ビュー定義が次の状態である場合、ビュー・マッチング・アルゴリズムで検査される実体化ビュー (Materialized View) のセットに、オプティマイザはその実体化ビュー (Materialized View) 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
  - 抽出テーブル

- UNION
- EXCEPT
- INTERSECT
- 実体化ビュー (Materialized View)
- DISTINCT
- TOP
- FIRST
- セルフジョイン
- 再帰ジョイン
- FULL OUTER JOIN

実体化ビュー (Materialized View) の定義に GROUP BY 句や HAVING 句 (HAVING 句に subselect やサブクエリが含まれない場合) が含まれることがあります。

> **注意**
> これらの制約は、ビュー・マッチング・アルゴリズムで検討される実体化ビュー (Materialized View) のみに適用されます。クエリで実体化ビュー (Materialized View) が明示的に参照される場合、オプティマイザはそのビューをベース・テーブルであるかのように使用します。

参照

- 「実行プランの解釈」 571 ページ.
- 「クエリ処理のフェーズ」 510 ページ
- 「アプリケーション・プロファイリング」 202 ページ

ビュー・マッチング

ビュー・マッチング・アルゴリズムは、クエリを満たすために実体化ビュー (Materialized View) を使用できるかどうかを判断します。この判断は、クエリ検査ステップと実体化ビュー (Materialized View) 検査ステップの 2 つのステップで行われます。

クエリ検査ステップ

クエリ検査ステップでは、ビュー・マッチング・アルゴリズムはクエリを検査します。次のいずれかの条件が true である場合、クエリの処理に実体化ビュー (Materialized View) 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。「ビュー・マッチング」 534 ページを参照してください。

  このため、オプティマイザは本来更新可能である SELECT 文や、更新可能なカーソルで明示的に宣言された SELECT 文で、実体化ビュー (Materialized View) 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 (そのためヒューリスティックに最適化される) 単純な DML 文である。ただし、OPTION 句で FORCE OPTIMIZATION を使用することによ

 Copyright © 2007, iAnywhere Solutions, Inc.

り、任意の SELECT 文でコストベースの最適化を使用できます。「SELECT 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- ストアド・プロシージャやユーザ定義関数の内部にクエリが含まれる場合と同様に、クエリの実行プランがキャッシュされている。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。「プランのキャッシュ」 544 ページを参照してください。

実体化ビュー (Materialized View) 検査ステップ

実体化ビュー (Materialized View) 検査ステップでは、クエリの全体または一部を計算するために使用可能な既存の実体化ビュー (Materialized View) を判断します。

実体化ビュー (Materialized View) がクエリの一部と一致すると、最終クエリ実行プランでビューを使用するかどうかがコストベースで決定されます。列挙フェーズの役割とは、ビュー・マッチング・アルゴリズムで推奨されるビューが含まれるプランを生成し、プランの推定コストに基づいて最適なアクセス・プラン (実体化ビュー (Materialized View) の一部が含まれることも含まれないこともある) を選択することです。

実体化ビュー (Materialized View) がグループ化されたプロジェクト選択ジョイン・クエリ (**グループ化されたクエリ**、または GROUP BY 句を含むクエリとも呼ばれる) として定義されている場合、ビュー・マッチング・アルゴリズムはそのビューとグループ化されたクエリ・ブロックとを対応させることができます。実体化ビュー (Materialized View) がプロジェクト選択ジョイン・クエリとして定義されている場合 (つまりグループ化されたクエリでない場合)、ビュー・マッチング・アルゴリズムはそのビューと任意のタイプのクエリ・ブロックを一致させることができます。

ビュー V がクエリ Q に属するクエリ・ブロック QB の一部と一致することをビュー・マッチング・アルゴリズムが判断するために必要な条件を次に示します。一般に、V には、クエリ QB のテーブルのサブセットが含まれる必要があります。唯一の例外は、V の拡張テーブルです。V の拡張テーブルは、V の他のテーブルのロー 1 つだけとジョインするテーブルです。たとえばプライマリ・キー・テーブルは、null でない外部キー・カラムとそのプライマリ・キー・カラムとの間で等価ジョインであるのが述部だけである場合に、拡張テーブルとなります。拡張テーブルを含む実体化ビュー (Materialized View) の例については、「 例 2 : グループ化されたプロジェクト選択ジョイン・ビューの一致」 539 ページを参照してください。

- 実体化ビュー (Materialized View) V を作成するためのオプションの値が、クエリを実行する接続のオプションの値と一致する。一致する必要のあるオプションのリストについては、「実体化ビュー (Materialized View) を管理するときの制限」 74 ページを参照してください。
- 実体化ビュー (Materialized View) V の最後のリフレッシュが、materialized_view_optimization データベース・オプション、または SELECT 文の MATERIALIZED VIEW OPTIMIZATION 句 (指定されている場合) によって指定された古さのスレッショルドを超えていない。「オプティマイザでの実体化ビュー (Materialized View) の古さ閾値の設定」 83 ページを参照してください。
- V で使用されるすべてのテーブルが QB に存在する (ただし V の拡張テーブルの一部に例外がある可能性がある)。この QB の共通テーブルのセットをこれ以降 CT とします。

- CT 内のテーブルは、クエリ Q で更新可能ではない。
- CT 内のすべてのテーブルは、QB で外部ジョインの同じ側に属する (つまり、すべて外部ジョインの保護された側にあるか、すべて QB の外部ジョインの NULL 入力側にある)。
- V の述部は、CT だけを参照する QB の述部のサブセットを包含することを判断可能である。つまり、V の述部は QB の述部よりも制限が少ない場合です。V の述部と正確に一致する QB の述部を**一致述部**と呼びます。
- CT が一致述部で使用されていない場合に、この CT 内のテーブルを参照する QB の任意の式は、V の select リストに出現する必要がある。
- V と QB の両方がグループ分けされている場合、QB は CT 内のテーブルの他に余分なテーブルを含まない。また、V の GROUP BY 句内の式のセットは、QB の GROUP BY 句内の式のセットと等しいか、そのスーパーセットである必要がある。
- 式の同一セットで V と QB の両方がグループ分けされている場合、QB 内のすべての集合関数も V で計算される必要があるか、または V の集合関数から計算できる。たとえば QB が AVG(x) を含む場合、V は AVG(x) を含むか、または SUM(x) と COUNT(x) の両方を含む必要があります。
- QB の GROUP BY 句が V の GROUP BY 句のサブセットである場合、QB の単純な集合関数は V の集合関数内にある必要がある。また、QB の複合集合関数は、V の単純な集合関数から計算できる必要がある。単純な集合関数は次のとおりです。

  - BIT_AND
  - BIT_OR
  - BIT_XOR
  - COUNT
  - LIST
  - MAX
  - MIN
  - SET_BITS
  - SUM
  - XMLAGG

  単純な集合関数から計算できる複合集合関数は次のとおりです。

  - SUM(x)
  - COUNT(x)
  - SUM(CAST(x AS DOUBLE))
  - SUM(CAST(x AS DOUBLE) * CAST(x AS DOUBLE))
  - VAR_SAMP(x)
  - VAR_POP(x)
  - VARIANCE(x)
  - STDDEV_SAMP(x)
  - STDDEV_POP(x)
  - STDDEV(x)

  単純な集合関数を使用すると、次の統計集合関数を計算できます。

Copyright © 2007, iAnywhere Solutions, Inc.

- COVER_SAMP(y,x)
- COVER_POP(y,x)
- CORR(y,x)
- REGR_AVGX(y,x)
- REGR_AVGY(y,x)
- REGR_SLOPE(y,x)
- REGR_INTERCEPT(y,x)
- REGR_R2(y,x)
- REGR_COUNT(y,x)
- REGR_SXX(y,x)
- REGR_SYY(y,x)
- REGR_SXY(y,x)

計算に使用される単純な集合関数は次のとおりです。

- SUM(y1)
- SUM(x1)
- COUNT(x1)
- COUNT(y1)
- SUM(x1*y1)
- SUM(y1*x1)
- SUM(x1*x1)
- SUM(y1*y1)

x1 は CAST(IFNULL(x, x,y) AS DOUBLE))、y1 は CAST(IFNULL(y,y,x) AS DOUBLE) をそれぞれ表します。

例 1：**プロジェクト選択ジョイン・ビューの一致**

ベース・テーブルの特定部分がクエリによって頻繁にアクセスされる場合、その部分を格納する実体化ビュー (Materialized View) を定義すると便利な場合があります。たとえば、次のように定義された実体化ビュー (Materialized View) V_Canada は、Customer テーブルからのカナダに住んでいるすべての顧客を格納します。この実体化ビュー (Materialized View) は State カラムが特定の値に制限されるときに使用できるため、V_Canada 実体化ビュー (Materialized View) の State カラムでインデックス V_Canada_State を作成することをおすすめします。

```
CREATE MATERIALIZED VIEW V_Canada AS
 SELECT c.ID, c.City, c.State, c.CompanyName
  FROM Customers c
  WHERE c.State IN ( 'AB', 'BC', 'MB', 'NB', 'NL',
   'NT', 'NS', 'NU', 'ON', 'PE', 'QC', 'SK', 'YT' );
REFRESH MATERIALIZED VIEW V_Canada;
CREATE INDEX V_Canada_State on V_Canada( State );
```

カナダに住んでいる顧客のサブセットだけを必要とする任意のクエリ・ブロックは、この実体化ビュー (Materialized View) の恩恵を受ける可能性があります。たとえばオンタリオ在住のすべての顧客に対して製品の合計額を会社ごとに計算する以下のクエリ 1 では、そのアクセス・プランで V_Canada 実体化ビュー (Materialized View) を使用する可能性があります。クエリ 1 がセマンティック上同等なクエリ 1_v に書き換えられたかのように、V_Canada 実体化ビュー (Materialized View) を使用するクエリ 1 のアクセス・プランは、有効なアクセス・プランを表します。オプティマイザがクエリを書き換えないことに注意してください。理論的には、実体化ビュー

(Materialized View) を使用する生成されたアクセスプランは、書き換えられたクエリに対応しているように見えます。

クエリ 1 の実行プランは Work[GrByH[V_Canada<V_Canada_State> JNLO SalesOrders<FK_CustomerID_ID> JNLO SalesOrderItems<FK_ID_ID> JNLO Products<ProductsKey>]] で、V_Canada 実体化ビュー (Materialized View) を使用します。

クエリ 1：

```
SELECT SUM( SalesOrderItems.Quantity
   * Products.UnitPrice ) AS Value
 FROM Customers c
  LEFT OUTER JOIN SalesOrders
   ON( SalesOrders.CustomerID = c.ID )
  LEFT OUTER JOIN SalesOrderItems
   ON( SalesOrderItems.ID = SalesOrders.ID )
  LEFT OUTER JOIN Products
   ON( Products.ID = SalesOrderItems.ProductID )
 WHERE c.State = 'ON'
 GROUP BY c.CompanyName;
```

クエリ 1_v：

```
SELECT SUM( SalesOrderItems.Quantity
   * Products.UnitPrice ) AS Value
  FROM V_Canada
   LEFT OUTER JOIN SalesOrders ON( SalesOrders.CustomerID = V_Canada.ID )
   LEFT OUTER JOIN SalesOrderItems ON( SalesOrderItems.ID = SalesOrders.ID )
   LEFT OUTER JOIN Products ON( Products.ID = SalesOrderItems.ProductID )
  WHERE V_Canada.State = 'ON'
  GROUP BY V_Canada.CompanyName;
```

クエリ 2 は、このビューをメイン・クエリ・ブロックと HAVING サブクエリの両方で使用します。オプティマイザが V_Canada 実体化ビュー (Materialized View) を使用して列挙するアクセス・プランの一部は、クエリ 2_v を表します。これはセマンティック上、Customer テーブルが V_Canada ビューで置き換えられたクエリ 2 と同一です。

実行プランは Work[GrByH[V_Canada<V_Canada_State> JNLO SalesOrders<FK_CustomerID_ID> JNLO SalesOrderItems<FK_ID_ID> JNLO Products<ProductsKey>]] : GrByS[V_Canada<seq> JNLO SalesOrders<FK_CustomerID_ID> JNLO SalesOrderItems<FK_ID_ID> JNLO Products<ProductsKey> です。

クエリ 2：

```
SELECT  SUM( SalesOrderItems.Quantity
  * Products.UnitPrice ) AS Value
 FROM Customers c
  LEFT OUTER JOIN SalesOrders
   ON( SalesOrders.CustomerID = c.ID )
  LEFT OUTER JOIN SalesOrderItems
   ON( SalesOrderItems.ID = SalesOrders.ID )
  LEFT OUTER JOIN Products
   ON( Products.ID = SalesOrderItems.ProductID )
 WHERE c.State = 'ON'
 GROUP BY CompanyName
 HAVING Value >
  ( SELECT AVG( SalesOrderItems.Quantity
      * Products.UnitPrice ) AS Value
     FROM Customers c1
```


Copyright © 2007, iAnywhere Solutions, Inc.

```
LEFT OUTER JOIN SalesOrders
ON( SalesOrders.CustomerID = c1.ID )
LEFT OUTER JOIN SalesOrderItems
ON( SalesOrderItems.ID = SalesOrders.ID )
 LEFT OUTER JOIN Products
ON( Products.ID = SalesOrderItems.ProductID )
WHERE c1.State IN ('AB', 'BC', 'MB', 'NB', 'NL', 'NT', 'NS',
'NU', 'ON', 'PE', 'QC', 'SK', 'YT' ) );
```

クエリ 2_v：

```
SELECT  SUM( SalesOrderItems.Quantity
         * Products.UnitPrice ) AS Value
 FROM V_Canada
 LEFT OUTER JOIN SalesOrders
  ON( SalesOrders.CustomerID=V_Canada.ID )
 LEFT OUTER JOIN SalesOrderItems
  ON( SalesOrderItems.ID=SalesOrders.ID )
 LEFT OUTER JOIN Products
  ON( Products.ID=SalesOrderItems.ProductID )
WHERE V_Canada.State = 'ON'
GROUP BY V_Canada.CompanyName
HAVING Value >
   ( SELECT AVG( SalesOrderItems.Quantity
         * Products.UnitPrice ) AS Value
       FROM V_Canada
        LEFT OUTER JOIN SalesOrders
        ON( SalesOrders.CustomerID = V_Canada.ID )
        LEFT OUTER JOIN SalesOrderItems
        ON( SalesOrderItems.ID = SalesOrders.ID )
         LEFT OUTER JOIN Products
        ON( Products.ID = SalesOrderItems.ProductID )
        WHERE V_Canada.State IN ('AB', 'BC', 'MB',
       'NB', 'NL', 'NT', 'NS', 'NU', 'ON', 'PE', 'QC',
       'SK', 'YT' ) );
```

例 2：グループ化されたプロジェクト選択ジョイン・ビューの一致

グループ化された実体化ビュー (Materialized View) は、グループ化されたクエリに与えるパフォーマンスの影響がもっとも高い可能性があります。頻繁に実行されるクエリで類似した集約が使用される場合、これらのクエリで使用される GROUP BY 句のスーパーセットの集約前データに対して実体化ビュー (Materialized View) が定義されなければなりません。クエリの複合集合関数は、ビューで使用される単純な集約から計算できるため、実体化ビュー (Materialized View) には単純な集合関数だけを格納することをおすすめします。

以下の実体化ビュー (Materialized View) V_quantity は、毎月と毎年の製品ごとの数量の合計とカウントを事前に計算します。以下のクエリ 3 ではこのビューを使用して、2000 年の月だけを選択できます (短いプランは Work[GrByH[V_quantity<seq>]] で、クエリ 3_v に対応する)。

クエリ 4 は拡張テーブル SalesOrders を参照しませんが、V_quantity はクエリ 4 の計算に必要なすべてのデータを含むため、V_quantity を使用します (短いプランは Work[GrByH [V_quantity<seq>]] で、クエリ 4_v に対応する)。

```
CREATE MATERIALIZED VIEW V_Quantity AS
 SELECT s.ProductID,
  Month( o.OrderDate ) AS month,
  Year( o.OrderDate ) AS year,
  SUM( s.Quantity ) AS q_sum,
  COUNT( s.Quantity  ) AS q_count
 FROM SalesOrderItems s KEY JOIN SalesOrders o
```

```
 GROUP BY s.ProductID, Month( o.OrderDate ),
  Year( o.OrderDate );
REFRESH MATERIALIZED VIEW V_Quantity;
```

クエリ 3：

```
SELECT s.ProductID,
  Month( o.OrderDate ) AS month,
  AVG( s.Quantity) AS avg,
  SUM( s.Quantity ) AS q_sum,
  COUNT( s.Quantity ) AS q_count
 FROM SalesOrderItems s KEY JOIN SalesOrders o
 WHERE year( o.OrderDate ) = 2000
 GROUP BY s.ProductID, Month( o.OrderDate );
```

クエリ 3_v：

```
SELECT V_Quantity.ProductID,
  V_Quantity.month AS month,
  SUM( V_Quantity.q_sum ) / SUM( V_Quantity.q_count )
   AS avg,
  SUM( V_Quantity.q_sum ) AS q_sum,
  SUM( V_Quantity.q_count ) AS q_count
 FROM V_Quantity
 WHERE V_Quantity.year  = 2000
 GROUP BY V_Quantity.ProductID, V_Quantity.month;
```

クエリ 4：

```
SELECT s.ProductID,
  AVG( s.Quantity ) AS avg,
  SUM( s.Quantity ) AS sum
 FROM SalesOrderItems s
 WHERE s.ProductID IS NOT NULL
 GROUP BY s.ProductID;
```

クエリ 4_v：

```
SELECT V_Quantity.ProductID,
  SUM( V_Quantity.q_sum ) / SUM( V_Quantity.q_count )
   AS avg,
  SUM( V_Quantity.q_sum ) AS sum
 FROM V_Quantity
 WHERE V_Quantity.ProductID IS NOT NULL
 GROUP BY V_Quantity.ProductID;
```

例 3：複雑なクエリの一致

ビュー・マッチング・アルゴリズムは、クエリ・ブロックごとに適用されます。そのため、クエリ・ブロックごとに複数の実体化ビュー (Materialized View) を使用できるため、クエリ全体で複数の実体化ビュー (Materialized View) を使用できます。以下のクエリ 5 では、3 つの実体化ビュー (Materialized View) を使用します。V_Canada は LEFT OUTER JOIN の NULL 入力側、V_ship_date はメイン・クエリ・ブロックの保護された側 (以下の定義参照)、V_quantity はサブクエリ・ブロックです。クエリ 5_v の実行プランは Work[Window[Sort [V_ship_date<V_Ship_date_date> JNLO (so<SalesOrdersKey> JH V_Canada<V_Canada_state>)]]] : GrByS[V_quantity<seq>] です。

```
CREATE MATERIALIZED VIEW V_ship_date AS
 SELECT s.ProductID, p.Description,
  s.Quantity, s.ShipDate, s.ID
```


Copyright © 2007, iAnywhere Solutions, Inc.

```
 FROM SalesOrderItems s KEY JOIN Products p ON ( s.ProductId = p.ID )
 WHERE s.ShipDate >= '2000-01-01'
  AND s.ShipDate <= '2001-01-01';
REFRESH MATERIALIZED VIEW V_ship_date;
CREATE INDEX V_ship_date_date ON V_ship_date( ShipDate );
```

クエリ 5：

```
SELECT p.ID, p.Description, s.Quantity,
  s.ShipDate, so.CustomerID, c.CompanyName,
  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) LEFT OUTER JOIN (
   SalesOrders so JOIN Customers c
    ON ( c.ID = so.CustomerID AND c.State = 'ON' ) )
  ON (s.ID = so.ID )
 WHERE s.ShipDate >= '2000-01-01'
   AND s.ShipDate <= '2000-12-31'
   AND s.Quantity  > ( SELECT  AVG( s.Quantity ) AS avg
                    FROM SalesOrderItems s KEY JOIN SalesOrders o
                    WHERE year( o.OrderDate) = 2000 )
 FOR READ ONLY;
```

クエリ 5_v：

```
SELECT  V_ship_date.ID, V_ship_date.Description,
  V_ship_date.Quantity, V_ship_date.ShipDate,
  so.CustomerID, V_Canada.CompanyName,
  SUM( V_ship_date.Quantity ) OVER ( PARTITION BY V_ship_date.ProductID
                         ORDER BY V_ship_date.ShipDate
                         ROWS BETWEEN UNBOUNDED PRECEDING
                         AND CURRENT ROW ) AS cumulative_qty
 FROM V_ship_date
  LEFT OUTER JOIN ( SalesOrders so JOIN V_Canada
   ON ( V_Canada.ID = so.CustomerID AND V_Canada.State = 'ON' ) )
  ON ( V_ship_date.ID = so.ID )
 WHERE V_ship_date.ShipDate >= '2000-01-01'
  AND V_ship_date.ShipDate <= '2000-12-31'
  AND V_ship_date.Quantity >
   ( SELECT SUM( V_quantity.q_sum ) / SUM( V_quantity.q_count )
      FROM V_Quantity
      WHERE V_Quantity.year = 2000 )
 FOR READ ONLY;
```

例 4：実体化ビュー (Materialized View) と外部ジョインの一致

ビュー・マッチング・アルゴリズムは、ビューと内部ジョインのみを含むクエリとの場合と同様のルールを使用して、ビューと外部ジョインを含むクエリとを対応させることができます。実体化ビュー (Materialized View) で OUTER JOIN の NULL 入力側は、NULL 入力側のすべてのテーブルが拡張テーブルである場合は、表示されないことがあります。クエリには、ビューの外部ジョインと一致する内部ジョインを含めることが可能です。以下のクエリ 6_v、7_v、8_v、9_v では、定義に OUTER JOIN を含む実体化ビュー (Materialized View) の別のクエリでの使用方法を示します。

以下のクエリ 6 は、実体化ビュー (Materialized View) V_SalesOrderItems_2000 と完全に一致し、これが 6_v であるかのように評価できます。

クエリ 7 では、外部ジョインの保護された側に述部が追加されていますが、V_SalesOrderItems_2000 を使用して計算できます。NULL 入力側のテーブル Products は、ビュー V_SalesOrderItems_2000 の拡張テーブルです。したがって、このビューは、Products テーブルが含まれないクエリ 8 とも一致します。

クエリ 9 には、テーブル SalesOrderItems と Products の内部ジョインだけが含まれます。V_SalesOrderItems_2000 ビューと一致するには、このビューでテーブル Products からの NULL 入力側ローではないローだけを選択します。クエリ 9_v での追加の述部 V.Description IS NOT NULL は、NULL 入力されないローだけを選択するために使用されます。

```
CREATE MATERIALIZED VIEW V_SalesOrderItems_2000 AS
 SELECT s.ProductID, p.Description,
  s.Quantity, s.ShipDate, s.ID
 FROM SalesOrderItems s LEFT OUTER JOIN Products p
           ON ( s.ProductId = p.ID )
 WHERE s.ShipDate >= '2000-01-01'
  AND s.ShipDate <= '2001-01-01';
REFRESH MATERIALIZED VIEW V_SalesOrderItems_2000;
CREATE INDEX V_SalesOrderItems_shipdate ON V_SalesOrderItems_2000( ShipDate );
```

クエリ 6：

```
CREATE MATERIALIZED VIEW V_SalesOrderItems_2000 AS
 SELECT s.ProductID, p.Description,
  s.Quantity, s.ShipDate, s.ID
 FROM SalesOrderItems s LEFT OUTER JOIN Products p
           ON ( s.ProductId = p.ID )
 WHERE s.ShipDate >= '2000-01-01'
  AND s.ShipDate <= '2001-01-01';
REFRESH MATERIALIZED VIEW V_SalesOrderItems_2000;
CREATE INDEX V_SalesOrderItems_shipdate ON V_SalesOrderItems_2000( ShipDate );
```

クエリ 6_v：

```
SELECT V.ProductID,  V.Description, V.Quantity, V.ShipDate,
  SUM( V.Quantity ) OVER ( PARTITION BY V.ProductID
                    ORDER BY V.ShipDate
                    ROWS BETWEEN UNBOUNDED PRECEDING
                    AND CURRENT ROW ) AS cumulative_qty
 FROM V_SalesOrderItems_2000 as V
 FOR READ ONLY;
```

クエリ 7：

```
SELECT s.ProductID,  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 LEFT OUTER JOIN Products p
           ON (s.ProductID = p.ID )

 WHERE s.ShipDate >= '2000-01-01'
   AND s.ShipDate <= '2001-01-01'
   AND s.Quantity >= 50
 FOR READ ONLY;
```

クエリ 7_v：

```
SELECT V.ProductID,  V.Description, V.Quantity, V.ShipDate,
  SUM( V.Quantity ) OVER ( PARTITION BY V.ProductID
```

 Copyright © 2007, iAnywhere Solutions, Inc.

```
                    ORDER BY V.ShipDate
                    ROWS BETWEEN UNBOUNDED PRECEDING
                    AND CURRENT ROW ) AS cumulative_qty
  FROM V_SalesOrderItems_2000 as V
  WHERE V.Quantity >= 50
  FOR READ ONLY;
```

クエリ 8：

```
 SELECT s.ProductID, 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
  WHERE s.ShipDate >= '2000-01-01'
    AND s.ShipDate <= '2001-01-01'
    AND s.Quantity >= 50
  FOR READ ONLY;
```

クエリ 8_v：

```
 SELECT V.ProductID, V.Quantity, V.ShipDate,
   SUM( V.Quantity ) OVER ( PARTITION BY V.ProductID
                    ORDER BY V.ShipDate
                    ROWS BETWEEN UNBOUNDED PRECEDING
                    AND CURRENT ROW ) AS cumulative_qty
  FROM V_SalesOrderItems_2000 as V
  WHERE V.Quantity >= 50
  FOR READ ONLY;
```

クエリ 9：

```
 SELECT s.ProductID,  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 >= '2000-01-01'
    AND s.ShipDate <= '2001-01-01'
  FOR READ ONLY;
```

クエリ 9_v：

```
 SELECT V.ProductID,  V.Description, V.Quantity, V.ShipDate,
   SUM( V.Quantity ) OVER ( PARTITION BY V.ProductID
                    ORDER BY V.ShipDate
                    ROWS BETWEEN UNBOUNDED PRECEDING
                    AND CURRENT ROW ) AS cumulative_qty
  FROM V_SalesOrderItems_2000 as V
  WHERE V.Description IS NOT NULL
  FOR READ ONLY;
```

MIN 関数と MAX 関数の最適化

MIN/MAX コストベース最適化は、既存のインデックスを利用して、MAX または MIN 集合関数を伴う単純な集合クエリの結果を効率的に計算するように設計されています。この最適化の目的

は、インデックスからわずか数ローを検索することで結果を計算できるようにすることです。この最適化の候補となるクエリの条件は、次のとおりです。

- GROUP BY 句がない
- 単一テーブルを対象としている
- クエリの SELECT リストに集合関数 (MAX または MIN) が 1 つだけ指定されている

例

この最適化の例として、SalesOrderItems テーブルにインデックス prod_qty (ShipDate ASC, Quantity ASC) があるとします。次のクエリを考えてみます。

```
SELECT MIN( Quantity )
 FROM SalesOrderItems
 WHERE ShipDate = '2000-03-25';
```

このクエリは内部的に次のように書き換えられます。

```
SELECT MAX( Quantity )
 FROM ( SELECT FIRST Quantity
        FROM SalesOrderItems
        WHERE ShipDate = '2000-03-25'
          AND Quantity IS NOT NULL
 ORDER BY ShipDate ASC, Quantity ASC ) AS s(Quantity);
```

この最適化を適用したときに、集合クエリに関する NULL_VALUE_ELIMINATED 警告が生成されない場合があります。

書き換えられたクエリの実行プラン (省略形) は、次のとおりです。

```
GrByS[ RL[ SalesOrderItems<prod_qty> ] ]
```

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。これらの文が繰り返し最適化されるコストを削減するために、オプティマイザによって次のプランがキャッシュされます。

- 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クエリ、INSERT 文、UPDATE 文、DELETE 文
- 最適化を回避するための条件を満たす INSERT 文、UPDATE 文、または DELETE 文。「最適化をバイパスするクエリ」 511 ページを参照してください。

  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ユニークに識別する探索条件を含む必要があります。プランをキャッシュする場合

Copyright © 2007, iAnywhere Solutions, Inc.

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、オプティマイザによって文の再利用可能なプランが構築されます。再利用可能なプランでは、選択性推定やリライト最適化にホスト変数の値は使用されません。このため、再利用可能なプランの方が高コストになることがあります。再利用可能なプランが、文が以前に実行されたときに観察されたプランと同じ構造体を持つ場合、データベース・サーバは、再利用可能なプランをプラン・キャッシュに追加します。それ以外の場合は、最適化を避けることによる節約よりも実行のたびに最適化する利点の方が重要なので、実行プランはキャッシュされません。

文で参照されない実体化ビュー (Materialized View) 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が最適化されます。

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、これは文を最適化するよりかなり高速です。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の方がまだ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。「max_plans_cached オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

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。「接続レベルのプロパティ」 『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・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。

参照

- 「実体化ビュー (Materialized View) によるパフォーマンスの向上」 531 ページ
- 「materialized_view_optimization オプション [データベース]」 『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』

Copyright © 2007, iAnywhere Solutions, Inc.

クエリ実行アルゴリズム

オプティマイザの機能は、特定の SQL 文 (SELECT、INSERT、UPDATE、または DELETE) を、さまざまな関係代数演算子 (ジョイン、重複排除、共用体など) から構成される効率的なアクセス・プランに変換することです。アクセス・プラン内の演算子は、元の SQL 文と構造が異なる場合がありますが、その SQL 要求とセマンティック上は等しい結果が計算されます。

アクセス・プランは、3 つの関係代数演算子から構成されます。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。アクセス・プランは、わかりやすいようにグラフィカルに表示できます。「実行プランの解釈」 571 ページと「グラフィカルなプラン」 581 ページを参照してください。

SQL Anywhere では、これらの関係代数演算の複数の実装がサポートされています。たとえば、SQL Anywhere では、ネスト・ループ・ジョイン、マージ・ジョイン、ハッシュ・ジョインの 3 つの異なる実装の内部ジョインがサポートされています。これらの演算子は、それぞれ特定の条件での使用に適しています。クエリ・オプティマイザで分析されるパラメータは、キャッシュ内のテーブル・データの量、ジョイン述部の特性と選択性、ジョインへの入力とジョインからの出力のソートの程度、ジョインの実行に使用できるメモリ容量などです。

SQL Anywhere では、実行時に、オプティマイザで選択された物理的な代数演算子から、論理的に同じである別の物理的なアルゴリズムに動的に切り替わる場合があります。一般に、この別のアクセス・プランは次の 2 つの状況で使用されます。

- 文の実行に使用されるメモリの合計容量がメモリ・ガバナーのスレッショルドに近いので、実行速度は遅いが、他の演算子 (他の要求) が使用できるようにメモリ容量が解放される方式に切り替わるとき。この場合は、QueryLowMemoryStrategy プロパティが増分されます。この情報は、文のグラフィカルなプランにも表示されます。QueryLowMemoryStrategy プロパティの詳細については、「接続レベルのプロパティ」 『SQL Anywhere サーバ - データベース管理』を参照してください。

  メモリ・ガバナーの制限は、サーバのマルチプログラミング・レベルとアクティブな接続の数によって異なります。「SQL Anywhere でのスレッド」 『SQL Anywhere サーバ - データベース管理』と「データベース・サーバのマルチプログラミング・レベルの設定」 『SQL Anywhere サーバ - データベース管理』を参照してください。

- 実行の開始時に特定の演算子 (たとえばハッシュ内部ジョイン) で、入力のカーディナリティが、最適化のときにオプティマイザによって計算されたカーディナリティと異なることが検出されたとき。この場合、演算子が、実行コストが低い別の方式に切り替わる可能性があります。この別の方式では通常はインデックス・ネスト・ループ処理が使用されます。ハッシュ・ジョインの場合は、この切り替えが発生するときに QueryJHToJNLOptUsed プロパティが増分されます。ジョイン方法の切り替えは文のグラフィカルなプランにも含まれます。QueryJHToJNLOptUsed プロパティの詳細については、「接続レベルのプロパティ」 『SQL Anywhere サーバ - データベース管理』を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

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 (「交換アルゴリズム」 567 ページを参照)。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 (「max_query_tasks オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』を参照)。

クエリ内並列処理は、background_priority が on に設定されている接続には使用されません。「background_priority オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

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・サーバの使用ライセンスがあるマシンの 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。「サーバ・レベルのプロパティ」 『SQL Anywhere サーバ - データベース管理』を参照してください。

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る場合がありますが、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 (dbisql 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、抽出テーブルで 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 I/O サブシステムを調整することで、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。スピンドルが複数ある場合は、並列実行がある程度含まれる実行プランがオプティマイザで選択される可能性が高くなります。

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 (「-gn サーバ・オプション」 『SQL Anywhere サーバ - データベース管理』を参照)。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) 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) 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

- 「max_query_tasks オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』
- 「SQL Anywhere でのスレッド」 『SQL Anywhere サーバ - データベース管理』
- 「-gtc 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・サーバのマルチプログラミング・レベルの設定」 『SQL Anywhere サーバ - データベース管理』
- 「交換アルゴリズム」 567 ページ
- 「実行プランの解釈」 571 ページ
- 「ALTER DATABASE 文」 『SQL Anywhere サーバ - SQL リファレンス』

並列処理が使用される状況

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、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

Copyright © 2007, iAnywhere Solutions, Inc.

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、マシン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。
- background_priority オプションが On 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 (マシン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

- 「クエリ実行時の並列処理」 547 ページ
- 「SQL Anywhere でのスレッド」 『SQL Anywhere サーバ - データベース管理』
- 「max_query_tasks オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』
- 「background_priority オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』
- 「サーバ・レベルのプロパティ」 『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』

テーブル・アクセス・アルゴリズム

この項では、テーブルにアクセスするためのさまざまな方法について、最も一般的な方法であるテーブル・スキャンとインデックス・スキャンとともに説明します。

ハッシュ・テーブル・スキャン

ハッシュ・テーブル・スキャン演算子は、SQL Anywhere がハッシュ・ジョインの構築側をメモリ内テーブルのようにスキャンします。これを使用して、次のような構造を持つプランを変換できます。

```
table1<seq>*JH ( <operator>... ( table2<seq> ) )
```

変換後の構造は次のとおりです。idx は、ハッシュ・テーブル内に格納されているジョイン・キー値の調査に使用できるインデックスです。

```
table1<seq>*JF ( <operator>... ( HTS JNB table2<idx> ) )
```

ハッシュ・ジョインとスキャンの間で演算子が使用されると、他の演算子で処理される必要のあるローの数が減ります。

この方式は、たとえば、構築側のローの数がインデックスのカーディナリティと比べて小さい場合など、インデックス調査で選択性が高い場合に最も便利です。

Copyright © 2007, iAnywhere Solutions, Inc.

注意
ハッシュ・ジョインの構築側が大きい場合、通常の逐次スキャンの方が効果的です。

オプティマイザは、ハッシュ・ジョイン代替実行に対するスレッショルドの計算と同様の方法でスレッショルドの構築サイズを計算します。この値を超えると、実行中に (HTS JNB table<idx>) が逐次スキャン (table<seq>) として処理されます。

注意
逐次方式は、ハッシュ・テーブルの構築側がディスクに溢れるような場合に使用します。

インデックス・スキャン

インデックス・スキャンでは、探索条件を満たすローを判断するときにインデックスを使用します。読み込まれるのは、条件を満たすページだけです。インデックスによって、ローをソートして返すことができます。

インデックス・スキャンは、短いプランと長いプランに *correlation_name*<*index_name*> と表示されます。ここで、*correlation_name* は FROM 句に指定された相関名、または指定されていない場合はテーブル名です。*index_name* はインデックス名です。

インデックスは、大規模なテーブルから少数のローを効率的に読み込むメカニズムを提供します。ただし、インデックス・スキャンでは、ページがデータベースからランダムに読み込まれるため、逐次読み込みよりコストが高くなります。また、あるテーブル・ページに探索条件を満たす複数のローがあると、インデックス・スキャンはそのページを複数回参照します。インデックス・スキャンによって一致するページが少数しかないと、そのページがキャッシュに残り、複数のアクセスによる余分な I/O は生じません。ただし、多数のページが探索条件と一致すると、すべてがキャッシュに収まらないことがあります。このため、インデックス・スキャンではディスクから同じページが複数回読み込まれることになります。

optimization_goal が first-row に設定されていると、オプティマイザは逐次テーブル・スキャンよりインデックス・スキャンを優先する傾向があります。これは、テーブル・スキャンに比べると、インデックスの方がクエリの最初のローを短時間で返す傾向があるためです。

また、インデックスは、順序付けの要件を満たすためにも使用できます。これは、ORDER BY 句で明示的に定義することも、GROUP BY または DISTINCT 句で暗黙的に必要とすることもあります。順序付き GROUP BY 方式と順序付き DISTINCT 方式は、ハッシュベースのグループ化や DISTINCT より短い時間で最初のローを返すことができます。ただし、結果セット全体を返すときには低速になる場合があります。

探索条件が検索引数可能であり、オプティマイザによる探索条件の選択性推定が低いためにインデックス・スキャンの方が逐次テーブル・スキャンより低コストの場合、オプティマイザはインデックス・スキャンを使用して探索条件を満たします。

インデックス・スキャンは、ローがインデックスからフェッチされた後で検索引数不可能な探索条件を評価することもできます。インデックス・スキャンで条件を評価することは、インデックス・スキャン後にフィルタ内の条件を評価するより多少効率的です。

 Copyright © 2007, iAnywhere Solutions, Inc.

SQL Anywhere がインデックスを使用できる場合の詳細については、「述部の分析」 529 ページを参照してください。

最適化のゴールの詳細については、「optimization_goal オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

並列インデックス・スキャン

並列インデックス・スキャン演算子は交換演算子とともに使用することで、インデックス・スキャンを並列に実行できます。各並列インデックス・スキャン演算子はローが必要なため、次の未処理のリーフ・ページを使用して、ページからローを 1 度に 1 つずつ返します。この方法では、並列処理を実現するために、演算子間でページが分割されます。並列インデックス・スキャン演算子間でページがどのように分散しているかに関係なく、すべてのローがアクセスされます。

並列テーブル・スキャン

並列テーブル・スキャン演算子は交換演算子とともに使用することで、並列テーブル・スキャンを並列に実行できます。各並列テーブル・スキャン演算子はローが必要なため、次の未処理のテーブル・ページを使用して、ページからローを 1 度に 1 つずつ返します。この方法では、並列処理を実現するために、演算子間でページが分割されます。並列テーブル・スキャン演算子間でページがどのように分散しているかに関係なく、テーブル内のすべてのローがアクセスされます。

ROWID スキャン・アルゴリズム

ROWID スキャン・アルゴリズムは、ベース・テーブルまたはテンポラリ・テーブルで、ROWID 関数を使用する等号比較述部に基づいて効率よくローを検索するのに使用されます。この比較述部では定数リテラルが参照される場合もありますが、通常はシステム関数またはシステム・プロシージャの呼び出し (sa_locks など) から返されたロー識別子の値が ROWID 関数で使用されます。

ROWID のスキャンは短いプランと長いプランに <*correlation_name*><*rowID*> と表示されます。ここで *correlation_name* は FROM 句で指定された相関名、または相関名が指定されていない場合はテーブル名です。

ROWID スキャン・アルゴリズムでは、ROWID 関数で参照されているテーブルのロー識別子が無効である場合と、ロー識別子がない場合を区別できません。したがって、比較述部で指定されたロー識別子がテーブル内で見つからない場合は、空の集合が返されます。

参照

- 「ROWID 関数 [その他]」『SQL Anywhere サーバ - SQL リファレンス』
- 「sa_locks システム・プロシージャ」『SQL Anywhere サーバ - SQL リファレンス』

逐次テーブル・スキャン

逐次テーブル・スキャンでは、テーブルの全ページのローすべてが、データベースに格納された順に読み込まれます。

逐次テーブル・スキャンは、短いプランと長いプランに *correlation_name*<*seq*> と表示されます。ここで、*correlation_name* は FROM 句に指定された相関名、または指定されていない場合はテーブル名です。

この種のスキャンが使用されるのは、テーブル・ページの大多数にクエリの探索条件と一致するローがある場合、または適切なインデックスが定義されていない場合です。

逐次テーブル・スキャンではインデックス・スキャンより多数のページが読み込まれる場合があります。ただし、ディスクの連続するブロックからページが読み込まれるため、ディスク I/O はかなり低コストです (このパフォーマンスの改善は、データベース・ファイルがディスク上で断片化されていない場合には最適です)。逐次 I/O では、ディスク・ヘッドの移動と回転待ち時間は低下します。大きなテーブルの場合、逐次テーブル・スキャンでは、一度に数ページも読み込まれます。このため、逐次テーブル・スキャンはインデックス・スキャンに比べてさらに低コストになります。

多数のローを一致させる場合、逐次テーブル・スキャンの方がインデックス・スキャンより短時間で済みます。ただし、スキャンが数回実行される場合は、インデックス・スキャンほど効率的にキャッシュを利用できません。インデックス・スキャンの方がアクセスするテーブル・ページ数が少ないため、キャッシュ内のページを利用できる可能性が高くなり、その結果アクセスが短時間になります。このため、ネスト・ループ・ジョインの右側など、反復的なテーブル・アクセスにはインデックス・スキャンの方が適しています。

トランザクションが独立性レベル 3 で実行されている場合は、アクセスするローが探索条件を満たしていなくても、SQL Anywhere は各ローをロックします。この独立性レベルでは、逐次テーブル・スキャンはテーブル内のローすべてをロックしますが、インデックス・スキャンは探索条件と一致するローだけをロックします。つまり、マルチユーザ環境で逐次テーブル・スキャンを使用すると、スループットが大幅に悪くなる可能性があります。このため、独立性レベル 3 ではオプティマイザは逐次アクセスよりインデックス・アクセスを優先します。逐次スキャンでは、スキャン中にテーブル・カラムと定数の間の単純な比較述部を効率的に評価できます。スキャン中のテーブルだけを参照するその他の探索条件は、これらの単純な比較の後で評価されます。このアプローチは、逐次スキャンの後でフィルタ内の条件を評価する方法より多少効率的です。

ジョイン・アルゴリズム

SQL Anywhere では、クエリ・オプティマイザが選択できるさまざまなジョインの実装がサポートされています。ジョイン・アルゴリズムはそれぞれ特性が異なるので、適しているクエリや実行環境が異なります。

アクセス・プランでのジョインの順序は、元の SQL 文でのジョインの順序に対応している場合としていない場合があります。クエリ・オプティマイザによって、最も低い実行コストに基づいて、各クエリに最適なジョイン方式が選択されます。場合によっては、特定の文に計算されたジョイン数を増加または減少する複合文にクエリ書き換え最適化が使用されます。

Copyright © 2007, iAnywhere Solutions, Inc.

SQL Anywhere では、 3 種類のジョイン・アルゴリズムがサポートされていて、この 3 種類にはそれぞれさらに変形があります。

- **ネスト・ループ・ジョイン**　最も単純なアルゴリズムは、ネスト・ループ・ジョインです。左側のローごとに右側がスキャンされ、ジョイン条件に基づいて一致するローが検索されます。通常は、右側のローは、全体的な実行コストを削減するために、インデックスを使用してアクセスされます。このシナリオは、一般に**インデックス・ネスト・ループ・ジョイン**と言います。

  ネスト・ループ・ジョインには、LEFT OUTER ジョインと FULL OUTER ジョインをサポートする変形があります。ネスト・ループ・ジョインは、セミジョイン (EXISTS サブクエリの処理に使用) にも使用できます。

  ネスト・ループ・ジョインは、ジョイン条件の特性に関係なく使用できます。ただし、不等号条件のジョインは、計算の効率が悪い可能性があります。

  ネスト・ループ FULL OUTER ジョインは、どのサイズの入力に対して実行しても高コストなので、クエリ・オプティマイザでは、他のジョイン・アルゴリズムが不可能な場合にのみ最後の手段として選択されます。

- **マージ・ジョイン**　マージ・ジョインでは、ジョイン属性に基づいて 2 つの入力がソートされている必要があります。クエリ・オプティマイザでこの方法が選択されるには、ジョイン条件に 1 つ以上の等号述部が必要です。基本のアルゴリズムは、2 つの入力の単純なマージです。2 つのジョイン属性の値が異なる場合は、左側または右側のどちらか値が小さい方の次のローに移ります。複数のローが一致する場合は、バックトラックが必要である場合があります。

  マージ・ジョインには、LEFT OUTER ジョインと FULL OUTER ジョインをサポートする変形があります。FULL OUTER ジョインのマージ・ジョインは、ネスト・ループ FULL OUTER ジョインよりも効率的です。
  基本のマージ・ジョイン・アルゴリズムは、SQL の集合演算子 EXCEPT と INTERSECT のサポートにも使用されます。これらの変形は、アクセス・プラン内では明示的に EXCEPT アルゴリズムまたは INTERSECT アルゴリズムと呼ばれます。

- **ハッシュ・ジョイン**　ハッシュ・ジョインは、SQL Anywhere データベース・サーバでサポートされている最も多用途のジョイン方法です。ハッシュ・ジョイン・アルゴリズムは、2 つの入力のうち小さい方のメモリ内ハッシュ・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、メモリ内ハッシュ・テーブルを調査して一致するローを検索します。

  ハッシュ・ジョインには、LEFT OUTER ジョイン、FULL OUTER ジョイン、セミジョイン、非セミジョインをサポートする変形があります。また、SQL Anywhere では、再帰 UNION クエリ式が使用されているときに再帰 INNER ジョインと再帰 LEFT OUTER ジョイン用のハッシュ・ジョインの変形がサポートされます。

  ハッシュ内部ジョイン、左外部ジョイン、セミジョイン、非セミジョインのアルゴリズムは並列実行できます。

  アルゴリズムによって構築されたメモリ内のハッシュ・テーブルが使用可能なメモリに収まらない場合は、ハッシュ・ジョイン・アルゴリズムによって入力が分割され (大きな入力の場

合は再帰的に)、各分割に対して別々にジョインが実行されます。ジョイン属性が特定の値であるローの格納に十分なキャッシュ・メモリがない場合、可能であれば、各ハッシュ・ジョイン・アルゴリズムは、文のメモリ消費割り当て量を使いつくさないように中間結果を廃棄してから、インデックスベースのネスト・ループ方式に動的に切り替わります。
ハッシュ・ジョインには、SQL のクエリ式 EXCEPT と INTERSECT をサポートする変形もあります。これらの変形は、アクセス・プラン内では明示的に EXCEPT アルゴリズムまたは INTERSECT アルゴリズムと呼ばれます。

ハッシュ非セミジョイン・アルゴリズム

ハッシュ・ジョイン・アルゴリズムのハッシュ非セミジョイン変形は、左側と右側の非セミジョインを実行します。右側のローは、結果に表示される左側のローを決定するためだけに使用されます。ハッシュ非セミジョインでは、右側のローを読み込んでメモリ内ハッシュ・テーブルが作成されます。その後、左側の各ローによってこのテーブルが調査されます。右側のどのローとも一致しない左側のローだけが出力されます。ハッシュ非セミジョインは、非ジョインとして書き換えることができる限定 (NOT IN、ALL、NOT EXISTS) のネストされたクエリのテーブル式がジョインの入力に含まれる場合に使用されます。右側のインデックス検索を低コストにする適切なインデックスが存在しない場合は、ハッシュ非セミジョインの方が、限定のクエリを参照する探索条件の評価よりパフォーマンスに優れています。

ハッシュ・ジョインと同様に、操作を完了するのに十分なキャッシュ・メモリがない場合は、ハッシュ非セミジョイン・アルゴリズムがネスト・ループ方式に戻ることがあります。この場合は、パフォーマンス・カウンタの値が増分されます。このモニタ値を読むには、QueryLowMemoryStrategy データベースまたは接続プロパティ、または Windows パフォーマンス モニタの [クエリ : メモリ不足時方式] カウンタを使用します。

メモリ・ガバナーの制限は、サーバのマルチプログラミング・レベルとアクティブな接続の数によって異なります。「SQL Anywhere でのスレッド」 『SQL Anywhere サーバ - データベース管理』と「データベース・サーバのマルチプログラミング・レベルの設定」 『SQL Anywhere サーバ - データベース管理』を参照してください。

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

詳細については、「接続レベルのプロパティ」 『SQL Anywhere サーバ - データベース管理』の「QueryLowMemoryStrategy」を参照してください。

ハッシュ・ジョイン・アルゴリズム

ハッシュ・ジョイン・アルゴリズムは、2 つの入力のうち小さい方のメモリ内ハッシュ・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、メモリ内ハッシュ・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、ハッシュ・ジョイン演算子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。

 Copyright © 2007, iAnywhere Solutions, Inc.

小さい方の入力がメモリに収まる場合は、大きい方の入力のサイズに関係なく、ハッシュ・ジョイン・アルゴリズムのパフォーマンスが最高になります。通常、入力の一方が他方よりかなり小さいと予測される場合に、オプティマイザはハッシュ・ジョインを選択します。

特定のジョイン属性値を持つすべてのローを保持できるほど十分なキャッシュ・メモリがない環境でハッシュ・ジョイン・アルゴリズムを実行した場合、ハッシュ・ジョイン・アルゴリズムは完了できません。この場合、ハッシュ・ジョイン方式では中間結果が廃棄され、代わりにインデックスベースのネスト・ループ・ジョインが使用されます。小さい方のテーブルのローがすべて読み込まれ、それを使用してワーク・テーブルが調査され、一致するローが検索されます。このインデックスベースの方式は他のジョイン方式に比べてかなり低速です。また、オプティマイザはクエリの実行中にメモリ不足を検出すると、ハッシュ・ジョインを使用したアクセス・プランの生成を避けます。

メモリ・ガバナーの制限は、サーバのマルチプログラミング・レベルとアクティブな接続の数によって異なります。「SQL Anywhere でのスレッド」『SQL Anywhere サーバ - データベース管理』と「データベース・サーバのマルチプログラミング・レベルの設定」『SQL Anywhere サーバ - データベース管理』を参照してください。

メモリ不足のためにネスト・ループ方式が必要になると、パフォーマンス・カウンタの値が増分されます。このモニタ値を読むには、QueryLowMemoryStrategy データベースまたは接続プロパティ、または Windows パフォーマンス モニタの [クエリ : メモリ不足時方式] カウンタを使用します。

ハッシュ・ジョイン・アルゴリズムはメモリ不足状態の Windows CE では無効になります。

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

詳細については、「接続レベルのプロパティ」『SQL Anywhere サーバ - データベース管理』と「データベース・サーバのマルチプログラミング・レベルの設定」『SQL Anywhere サーバ - データベース管理』の「QueryLowMemoryStrategy」を参照してください。

ハッシュ・ジョイン・アルゴリズムには、次の特徴もあります。

- 結果のすべてのローを計算してから、最初のローを返す。
- ワーク・テーブルを使用するので、value-sensitive カーソルが要求されていない場合は insensitive セマンティックが提供される。
- 並列実行できる。
- 入力のローをロックしてからメモリにコピーする。

ハッシュ・セミジョイン・アルゴリズム

ハッシュ・ジョイン・アルゴリズムのハッシュ・セミジョイン変形は、左側と右側のセミジョインを実行します。前述のネスト・ループ・セミジョインと同様に、右側のローは、結果に表示される左側のローを決定するためだけに使用されます。ハッシュ・セミジョインでは、右側のローを読み込んでメモリ内ハッシュ・テーブルが作成されます。その後、左側の各ローによってこの

Copyright © 2007, iAnywhere Solutions, Inc.

テーブルが調査されます。一致するローが見つかると、左側のローが直ちに結果に出力され、左側の次のローについて、再び調査プロセスが開始されます。少なくとも 1 つの等号ジョイン条件がないと、クエリ・オプティマイザはハッシュ・セミジョインを考慮しません。ネスト・ループ・セミジョインと同様に、ジョインとして書き換えられた存在限定 (IN、SOME、ANY、EXISTS) のネストされたクエリのテーブル式がジョインの入力に含まれる場合、ハッシュ・セミジョインを使用できます。ジョイン条件に不等号が含まれる場合、または右側のインデックス検索を低コストにする適切なインデックスが存在しない場合は、ハッシュ・セミジョインの方がネスト・ループ・セミジョインよりパフォーマンスに優れています。

ハッシュ・ジョインと同様に、操作を完了するのに十分なキャッシュ・メモリがない場合は、ハッシュ・セミジョイン・アルゴリズムがネスト・ループ・セミジョイン方式に戻ることがあります。この場合は、パフォーマンス・カウンタの値が増分されます。このモニタ値を読むには、QueryLowMemoryStrategy データベースまたは接続プロパティ、または Windows パフォーマンス モニタの [クエリ : メモリ不足時方式] カウンタを使用します。

メモリ・ガバナーの制限は、サーバのマルチプログラミング・レベルとアクティブな接続の数によって異なります。「SQL Anywhere でのスレッド」 『SQL Anywhere サーバ - データベース管理』と「データベース・サーバのマルチプログラミング・レベルの設定」 『SQL Anywhere サーバ - データベース管理』を参照してください。

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

詳細については、「接続レベルのプロパティ」 『SQL Anywhere サーバ - データベース管理』の「QueryLowMemoryStrategy」を参照してください。

マージ・ジョイン・アルゴリズム

マージ・ジョイン・アルゴリズムは 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。

入力がまだジョイン属性によって順序付けされていない場合 (以前のマージ・ジョインがあるため、または探索条件を満たすためにインデックスが使用されたためなど)、オプティマイザはソートを追加して正しいロー順を生成します。このソートによってマージ・ジョインにコストが追加されます。

マージ・ジョインがハッシュ・ジョインより優れているのは、マージ・ジョインが同じ属性を対象としている場合に、ソートによるコストを複数のジョインに分散できることです。入力のサイズが同じと思われる場合、またはソートによるコストを複数の操作に分散できる場合、オプティマイザはハッシュ・ジョインよりマージ・ジョインを選択します。

Copyright © 2007, iAnywhere Solutions, Inc.

ネスト・ループ・ジョイン・アルゴリズム

ネスト・ループ・ジョインでは、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 (オプティマイザは要求内のブロックごとに適切なジョイン順を選択するため、クエリに含まれるテーブルの構文上の順序は重要ではありません)。

ジョイン条件に等号条件が含まれない場合、または文が First-Row 最適化ゴールで最適化されている場合 (optimization_goal オプションが First-Row に設定されているか、FROM 句でテーブルのヒントとして FASTFIRSTROW が指定されている場合) に、オプティマイザでネスト・ループ・ジョインが選択される可能性があります。

ネスト・ループ・ジョインは右側を何回も読み込むため、右側のコストに大きく影響されます。右側がインデックス・スキャンまたは小さなテーブルの場合は、以前の反復でキャッシュされたページを利用して右側を計算できます。これに対して、右側が逐次テーブル・スキャンまたは多数のローと一致するインデックス・スキャンの場合は、右側をディスクから何回も読み込みます。通常、ネスト・ループ・ジョインの効率は、他のジョイン方式より低くなります。ただし、ネスト・ループ・ジョインが最初の一致するローを返す速度は、結果全体を計算してから返すジョイン方式より高速です。

ジョインで構成されるクエリに sensitive セマンティックを提供できるジョイン・アルゴリズムは、ネスト・ループ・ジョイン・アルゴリズムだけです。つまり、ジョイン上の sensitive カーソルは、ネスト・ループ・ジョインを使用した場合のみ実行できます。

セミジョインは、右側から最初の一致するローだけをフェッチします。これは、より効率的なネスト・ループ・ジョインですが、EXISTS キーワード、ときには DISTINCT キーワードの使用時しか使用できません。

参照

- 「optimization_goal オプション [データベース]」『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』

ネスト・ループ・セミジョイン・アルゴリズム

前述のネスト・ループ・ジョインと同様に、ネスト・ループ・セミジョイン・アルゴリズムは、ネスト・ループ・アルゴリズムを使用して左側の各ローを右側のローとジョインします。ネスト・ループ・ジョインと同様に、右側が何度も読み込まれるため、大きい入力にはインデックス・スキャンの方が適しています。ただし、ネスト・ループ・セミジョインは 2 つの点でネスト・ループ・ジョインと異なります。第 1 に、セミジョインは左側の値のみを出力します。右側は、結果に表示される左側のローを制限するためだけに使用されます。第 2 に、ネスト・ループ・セミジョイン・アルゴリズムでは、最初に一致するローが見つかった時点ですぐに右側の検索が停止します。ジョインとして書き換えられた存在限定 (IN、SOME、ANY、EXISTS) のネストされたクエリのテーブル式がジョインの入力に含まれる場合、ネスト・ループ・セミジョインをジョイン・アルゴリズムとして使用できます。

Copyright © 2007, iAnywhere Solutions, Inc.

再帰ハッシュ・ジョイン・アルゴリズム

再帰ハッシュ・ジョインは、ハッシュ・ジョイン・アルゴリズムの変形の 1 つで、再帰的な UNION クエリで使用されます。

詳細については、「ハッシュ・ジョイン・アルゴリズム」 554 ページと「再帰共通テーブル式」 398 ページを参照してください。

再帰左外部ハッシュ・ジョイン・アルゴリズム

再帰左外部ハッシュ・ジョインは、ハッシュ・ジョイン・アルゴリズムの変形の 1 つで、特定の再帰的な UNION クエリで使用されます。

詳細については、「ハッシュ・ジョイン・アルゴリズム」 554 ページと「再帰共通テーブル式」 398 ページを参照してください。

重複排除アルゴリズム

重複排除演算子は、重複するローを含まない出力を生成します。重複排除ノードは、ネストされたクエリをジョインに変換する場合などに、オプティマイザが採用します。

詳細については、「ハッシュ DISTINCT アルゴリズム」 558 ページと「順序付き DISTINCT アルゴリズム」 559 ページを参照してください。

ハッシュ DISTINCT アルゴリズム

ハッシュ DISTINCT アルゴリズムでは、入力が読み込まれ、メモリ内ハッシュ・テーブルが構築されます。入力のローは、ハッシュ・テーブル内で見つかると無視されます。それ以外の場合は、ワーク・テーブルに書き込まれます。入力全体がメモリ内ハッシュ・テーブルに収まらない場合は、小さなワーク・テーブルに分割され、再帰的に処理されます。

ハッシュ DISTINCT アルゴリズムは、次のような性質があります。

- DISTINCT ローがメモリ内テーブルに収まる場合は、入力の合計ロー数に関係なく適切に機能します。
- ワーク・テーブルを使用するため、insensitive セマンティックまたは value sensitive セマンティックを提供できます。
- 前に返されなかったローを見つけるとすぐにそのローを返します。ただし、ハッシュ DISTINCT の結果が完全に実体化されてから、クエリがローを返さなければなりません。このため、オプティマイザは必要に応じてワーク・テーブルを実行プランに追加します。
- 入力のローをロックします。

オプティマイザはクエリの実行中にメモリ不足を検出すると、ハッシュ DISTINCT アルゴリズムを使用したアクセス・プランの生成を避けます。利用可能なキャッシュ・メモリがほとんどない

Copyright © 2007, iAnywhere Solutions, Inc.

環境でハッシュ DISTINCT アルゴリズムを実行した場合、ハッシュ DISTINCT アルゴリズムは完了できません。この場合、ハッシュ DISTINCT 方式では中間結果が廃棄され、代わりにインデックス DISTINCT アルゴリズムが使用されます。

メモリ・ガバナーの制限は、サーバのマルチプログラミング・レベルとアクティブな接続の数によって異なります。「SQL Anywhere でのスレッド」『SQL Anywhere サーバ - データベース管理』と「データベース・サーバのマルチプログラミング・レベルの設定」『SQL Anywhere サーバ - データベース管理』を参照してください。

インデックス DISTINCT アルゴリズム

インデックス DISTINCT アルゴリズムは、入力からのユニークなローのワーク・テーブルを管理します。ローが入力から読み込まれると、ワーク・テーブル上のインデックスが検索され、入力ローにこれまであった重複の有無がチェックされます。重複が見つかると、入力のローは無視されます。それ以外の場合は、ワーク・テーブルに挿入されます。SELECT リストのすべてのカラムについて、ワーク・テーブルのインデックスが作成されます。インデックスのパフォーマンスを向上させるために、最初の式としてハッシュ式が追加されます。このハッシュ式は計算値で、SELECT リストに含まれるすべてのカラムの値が埋め込まれています。

インデックス DISTINCT 方式では、DISTINCT ローが検出されるたびに返されます。このため、他の重複排除方式に比べると、最初のいくつかのローを迅速に返すことができます。インデックス DISTINCT アルゴリズムは、一度に 2 つのローだけをメモリに格納するので、メモリ容量が極端に小さい状況でも十分に機能します。ただし、DISTINCT ローが多数ある場合は、通常、インデックス DISTINCT アルゴリズムの実行コストの方がハッシュ DISTINCT より悪くなります。DISTINCT ローの格納に使用されるワーク・テーブルがキャッシュに収まらないと、ワーク・テーブルのページがランダム・アクセス・パターンで何度も再読み込みされてしまいます。

インデックス DISTINCT 方式ではワーク・テーブルが使用されるため、sensitive セマンティックを完全には提供できません。ただし、insensitive セマンティックも完全には提供できないので、insensitive カーソルには別のワーク・テーブルが必要です。また、インデックス DISTINCT 方式は入力のローをロックします。

メモリ不足のためにインデックス DISTINCT アルゴリズム方式が必要になると、パフォーマンス・カウンタの値が増分されます。この値を確認するには、QueryLowMemoryStrategy データベースまたは接続プロパティ、グラフィカルなプランの QueryLowMemoryStrategy 統計情報 (統計情報付きで実行した場合)、または Windows パフォーマンス モニタの [クエリ：メモリ不足時方式] カウンタを使用します。「パフォーマンス・モニタの統計値」246 ページを参照してください。

順序付き DISTINCT アルゴリズム

入力がすべてのカラムによって順序付けされている場合は、順序付き DISTINCT アルゴリズムを使用できます。このアルゴリズムでは、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。順序付き DISTINCT アルゴリズムが効果的なのは、ローが (インデックスまたはマージ・ジョインの可能性があるため) すでに順序付けされている場合です。入力が順序付けされていない場合、オプティマイザはソートを挿入します。ワーク・テーブルは、順序付き DISTINCT 自体では使用されませんが、挿入されたソートによって使用されます。

グループ化アルゴリズム

グループ化アルゴリズムでは、入力の合計が計算されます。グループ化アルゴリズムを適用できるのは、クエリに GROUP BY 句がある場合、またはクエリに集合関数 (SELECT COUNT(*) FROM T など) がある場合だけです。

詳細については、「ハッシュ GROUP BY アルゴリズム」 560 ページ、「順序付き GROUP BY アルゴリズム」 561 ページ、「単一の GROUP BY アルゴリズム」 562 ページを参照してください。

クラスタード・ハッシュ GROUP BY アルゴリズム

場合によっては、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ので、似たような値が互いに接近して現れます。たとえば、常に現在の日付に設定されているカラムがテーブルに含まれている場合、単一の日付を持つすべてのローがテーブル内で比較的近くなります。クラスタード・ハッシュ GROUP BY アルゴリズムは、このクラスタ化を利用します。

使用可能なメモリよりも大幅に大きいテーブルをグループ化する場合、オプティマイザはクラスタード・ハッシュ GROUP BY を使用する可能性があります。特に、HAVING 述部がローの小さい部分のみを返す場合に効果的です。

クラスタード・ハッシュ GROUP BY アルゴリズムでは、データがクエリ実行と同時に更新されるような環境で使用される場合、オプティマイザの作業の一部が大幅に無駄になる可能性があります。したがって、クラスタード・ハッシュ GROUP BY は、一時バッチ形式の更新や読み込みベースのクエリを特徴とする OLAP 負荷に最適です。optimization_workload オプションを OLAP に設定して、調査する候補にクラスタード・ハッシュ GROUP BY アルゴリズムを含めるようにオプティマイザに指示します。「optimization_workload オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

OLAP 負荷で使用できるインデックスまたは外部キーを作成する場合は、FOR OLAP WORKLOAD 句を指定してください。この句を指定すると、データベース・サーバは同じキー内の 2 つのローの最大ページ距離に関して、クラスタード GROUP BY ハッシュで使用する統計情報を管理します。「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』を参照してください。

OLAP の詳細については、「OLAP のサポート」 413 ページを参照してください。

ハッシュ GROUP BY アルゴリズム

ハッシュ GROUP BY アルゴリズムでは、グループ・ローのメモリ内ハッシュ・テーブルが作成されます。ローが入力から読み込まれると、グループ・ローが更新されます。

ハッシュ GROUP BY アルゴリズムは、結果のローをすべて計算してから最初のローを返します。また、完全な sensitive または values sensitive カーソルを満たすために使用できます。ハッシュ GROUP BY の結果が完全に実体化されてから、クエリがローを返されなければなりません。このため、オプティマイザは必要に応じてワーク・テーブルを実行プランに追加します。

 Copyright © 2007, iAnywhere Solutions, Inc.

ハッシュ GROUP BY アルゴリズムは並列実行できます。

グループがメモリに収まる場合は、入力のサイズに関係なくハッシュ GROUP BY アルゴリズムが適切に機能します。ハッシュ・テーブルがメモリに収まらない場合は、入力はメモリに収まるようになるまで小さなワーク・テーブルに再帰的に分割されます。オプティマイザはクエリの実行中にメモリ不足を検出すると、ハッシュ GROUP BY アルゴリズムを使用したアクセス・プランの生成を避けます。分割用のメモリが足りないと、オプティマイザはハッシュ GROUP BY からの中間結果を廃棄し、代わりにインデックス GROUP BY アルゴリズムを使用します。

メモリ・ガバナーの制限は、サーバのマルチプログラミング・レベルとアクティブな接続の数によって異なります。「SQL Anywhere でのスレッド」『SQL Anywhere サーバ - データベース管理』と「データベース・サーバのマルチプログラミング・レベルの設定」『SQL Anywhere サーバ - データベース管理』を参照してください。

インデックス GROUP BY アルゴリズム

インデックス GROUP BY アルゴリズムでは、グループごとに 1 つのローで構成されるワーク・テーブルが構築されます。入力のローが読み込まれると、インデックスを使用してワーク・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。

メモリ不足のためにインデックス GROUP BY アルゴリズムが必要になると、パフォーマンス・カウンタの値が増分されます。この値を確認するには、QueryLowMemoryStrategy データベースまたは接続プロパティ、グラフィカルなプランの QueryLowMemoryStrategy 統計情報 (統計情報付きで実行した場合)、または Windows パフォーマンス モニタの [クエリ : メモリ不足時方式] カウンタを使用します。「パフォーマンス・モニタの統計値」 246 ページを参照してください。

ハッシュ GROUP BY SETS アルゴリズム

GROUPING SETS クエリを実行する場合、ハッシュ GROUP BY アルゴリズムの変形が使用されます。

ハッシュ GROUP BY SETS アルゴリズムは並列実行できません。

詳細については、「ハッシュ GROUP BY アルゴリズム」 560 ページを参照してください。

順序付き GROUP BY アルゴリズム

順序付き GROUP BY アルゴリズムでは、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。

Copyright © 2007, iAnywhere Solutions, Inc.

順序付き GROUP BY SETS アルゴリズム

GROUPING SETS クエリを実行するときに使用される順序付き GROUP BY アルゴリズムの変形です。このアルゴリズムは、使用可能なインデックスが必要です。

詳細については、「順序付き GROUP BY アルゴリズム」 561 ページを参照してください。

単一の GROUP BY アルゴリズム

GROUP BY を指定しなければ、単一ローの集合が生成されます。

単一の XML GROUP BY アルゴリズム

XML ベースのクエリを処理している場合に使用するアルゴリズムです。

ソート GROUP BY SETS アルゴリズム

GROUPING SETS を含む OLAP クエリを処理する場合に使用されるアルゴリズムです。

このアルゴリズムは、インデックスがなく、順序付き GROUP BY SETS アルゴリズムを使用できない場合に使用できます。

クエリ式アルゴリズム

クエリ式アルゴリズムは、次のカテゴリに分類できます。

- EXCEPT アルゴリズム (EXCEPT マージと EXCEPT ハッシュ)
- INTERSECT アルゴリズム (INTERSECT マージと INTERSECT ハッシュ)
- UNION アルゴリズム (UNION、UNION ALL、再帰 UNION)

EXCEPT アルゴリズム

SQL Anywhere のクエリ・オプティマイザは、SQL の集合差演算子 EXCEPT のソートベースの変形「**EXCEPT マージ**」とハッシュベースの変形「**EXCEPT ハッシュ**」の 2 つの物理的な実装から選択します。

EXCEPT マージでは、マージ・ジョイン演算子を使用して、ソートされた順序でローの一致を分析して、2 つの入力の集合差が計算されます。多くの場合、2 つの入力の明示的なソートが必要です。同様に、EXCEPT ハッシュでは、ハッシュ非セミジョイン・アルゴリズムを使用して 2 つの入力の集合差が計算され、ハッシュ左外部ジョインを使用して 2 つの入力の差が計算されます (EXCEPT ALL)。

Copyright © 2007, iAnywhere Solutions, Inc.

メモリ不足が検出されると、EXCEPT ハッシュ演算子は動的にネスト・ループ方式に切り替わる場合があります。この場合は、パフォーマンス・カウンタの値が増分されます。このモニタ値を読むには、QueryLowMemoryStrategy データベースまたは接続プロパティ、グラフィカルなプランの QueryLowMemoryStrategy 統計情報 (統計情報付きで実行した場合)、または Windows パフォーマンス モニタの [クエリ : メモリ不足時方式] カウンタを使用します。

EXCEPT ハッシュ・アルゴリズムはメモリ不足状態の Windows CE では無効になります。

EXCEPT の場合、EXCEPT マージまたは EXCEPT ハッシュ・アルゴリズムは、結果に重複が含まれないように、DISTINCT アルゴリズムの 1 つと組み合わせられます。EXCEPT ALL の場合、EXCEPT アルゴリズムは、結果内の重複するローの正しい数を計算する RowReplicate アルゴリズムと組み合わせられます。

参照

- 「EXCEPT 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「UNION、INTERSECT、EXCEPT を使用した、クエリ結果に対する集合操作の実行」 332 ページ
- QueryLowMemoryStrategy 接続プロパティ :「接続レベルのプロパティ」『SQL Anywhere サーバ - データベース管理』
- QueryLowMemoryStrategy データベース・プロパティ :「データベース・レベルのプロパティ」『SQL Anywhere サーバ - データベース管理』
- クエリ : メモリ不足時方式の統計値 :「パフォーマンス・モニタの統計値」 246 ページ

INTERSECT アルゴリズム

SQL Anywhere のクエリ・オプティマイザは、SQL の集合積演算子 INTERSECT のソートベースの変形「**INTERSECT マージ**」とハッシュベースの変形「**INTERSECT ハッシュ**」の 2 つの物理的な実装から選択します。

INTERSECT マージでは、マージ・ジョイン演算子を使用して、ソートされた順序でローの一致を分析して、2 つの入力の集合積が計算されます。多くの場合、2 つの入力の明示的なソートが必要です。同様に、INTERSECT ハッシュでは、ハッシュ内部ジョイン・アルゴリズムを使用して、2 つの入力間の集合積とバッグ積が計算されます (INTERSECT と INTERSECT ALL)。

メモリ不足が検出されると、INTERSECT ハッシュ演算子は必要に応じて動的にネスト・ループ方式に切り替わります。この場合は、パフォーマンス・カウンタの値が増分されます。このモニタ値を読むには、QueryLowMemoryStrategy データベースまたは接続プロパティ、グラフィカルなプランの QueryLowMemoryStrategy 統計情報 (統計情報付きで実行した場合)、または Windows パフォーマンス モニタの [クエリ : メモリ不足時方式] カウンタを使用します。

INTERSECT ハッシュ・アルゴリズムはメモリ不足状態の Windows CE では無効になります。

INTERSECT の場合、INTERSECT マージまたは INTERSECT ハッシュ・アルゴリズムは、結果に重複が含まれないように、DISTINCT アルゴリズムの 1 つと組み合わせられます。INTERSECT ALL クエリ式の場合、INTERSECT アルゴリズムは、結果内の重複するローの正しい数を計算する RowReplicate アルゴリズムと組み合わせられます。

参照

- 「INTERSECT 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「UNION、INTERSECT、EXCEPT を使用した、クエリ結果に対する集合操作の実行」 332 ページ
- QueryLowMemoryStrategy 接続プロパティ：「接続レベルのプロパティ」『SQL Anywhere サーバ - データベース管理』
- QueryLowMemoryStrategy データベース・プロパティ：「データベース・レベルのプロパティ」『SQL Anywhere サーバ - データベース管理』
- クエリ：メモリ不足時方式の統計値：「パフォーマンス・モニタの統計値」 246 ページ

再帰テーブル・アルゴリズム

再帰テーブルは、クエリ内の WITH 句の結果構築される共通テーブル式です。WITH 句は、再帰的な UNION クエリに使用されます。共通テーブル式は、1 つの SELECT 文のスコープ内のみで認識されるテンポラリ・ビューです。

詳細については、「共通テーブル式」 391 ページを参照してください。

再帰 UNION アルゴリズム

再帰 UNION アルゴリズムは、再帰的な UNION クエリの実行中に使用されます。

詳細については、「再帰共通テーブル式」 398 ページを参照してください。

RowReplicate アルゴリズム

RowReplicate アルゴリズムは、EXCEPT ALL や INTERSECT ALL などの集合操作の実行時に使用されます。こうした操作の特徴は、結果セット内のロー数が、操作の対象となっている 2 つの集合内のロー数に明示的に関連付けられていることです。RowReplicate アルゴリズムは、結果セット内のロー数が正しいことを保証します。

詳細については、「UNION、INTERSECT、EXCEPT を使用した、クエリ結果に対する集合操作の実行」 332 ページを参照してください。

UNION ALL アルゴリズム

UNION ALL アルゴリズムでは、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 句と UNION ALL 句を実装するために使用されます。UNION の場合は、UNION ALL によって生成される重複を削除するための重複排除アルゴリズムが必要です。

Copyright © 2007, iAnywhere Solutions, Inc.

ソート・アルゴリズム

ソート・アルゴリズムは、クエリに ORDER BY 句がある場合、またはクエリの実行方式で、入力の完全なソートが必要な場合に適用されます。

詳細については、「ソート・アルゴリズム」 565 ページと「UNION ALL アルゴリズム」 564 ページを参照してください。

ソート・アルゴリズム

ソート演算子は入力をメモリに読み込み、メモリ内でソートしてからソート結果を出力します。入力全体がメモリに収まらない場合は、複数のソート済み処理が作成されてからマージされます。ソートはすべての入力ローを読み込むまで、ローを返しません。ソートは入力ローをロックします。

利用可能なキャッシュ・メモリがほとんどない環境でソート・アルゴリズムを実行すると、ソート・アルゴリズムが完了できない場合があります。この場合、ソート・アルゴリズムはインデックスベースのソート方式を使用して残りの入力を順序付けます。入力ローが読み込まれてワーク・テーブルに挿入され、ワーク・テーブルの順序付けカラムに基づいてインデックスが作成されます。この場合、ローは複合インデックス・スキャンを使用してワーク・テーブルから読み込まれます。このインデックスベースの方式は、かなり低速です。オプティマイザはクエリの実行中にメモリ不足を検出すると、ソート・アルゴリズムを使用したアクセス・プランの生成を避けます。メモリ不足のためにインデックスベースの方式が必要な場合は、パフォーマンス・カウンタの値が増分されます。このモニタを読むには、QueryLowMemoryStrategy プロパティまたは Windows パフォーマンス モニタの [クエリ：メモリ不足時方式] カウンタを使用します。

メモリ・ガバナーの制限は、サーバのマルチプログラミング・レベルとアクティブな接続の数によって異なります。「SQL Anywhere でのスレッド」 『SQL Anywhere サーバ - データベース管理』と「データベース・サーバのマルチプログラミング・レベルの設定」 『SQL Anywhere サーバ - データベース管理』を参照してください。

ソート・パフォーマンスは、ソート・キーのサイズ、ローのサイズ、入力の合計サイズの影響を受けます。多数のローの場合は、VALUES SENSITIVE カーソルを使用する方が低コストになることがあります。その場合、SELECT リストのカラムは、ソートに使用されるワーク・テーブルにはコピーされません。ソートでは、出力ローがワーク・テーブルに書き込まれませんが、ソートの結果が実体化されてから、ローがアプリケーションに返されなければなりません。このため、オプティマイザは必要に応じてワーク・テーブルを追加します。

上位 N のソート・アルゴリズム

上位 N のソート・アルゴリズムは、TOP N 句と ORDER BY 句を含むクエリで使用されます。このアルゴリズムは、結果セットに必要なローのみをソートする場合に効率的です。

詳細については、「SELECT 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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、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、NEW_ID、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。

ユーザ定義関数の詳細については、「CREATE FUNCTION 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

その他のアルゴリズム

その他に、アクセス・プランには次のアルゴリズムを使用できます。

抽出テーブル・アルゴリズム

抽出テーブルは、クエリの FROM 句に含まれている SELECT 文です。SELECT 文の結果セットは、論理的にはテーブルのように扱われます。クエリ・オプティマイザは、クエリの書き換え時、たとえば UNION、INTERSECT、または EXCEPT の操作に基づくセットを含むクエリでも抽出テーブルを生成することがあります。グラフィカル・プランには、抽出テーブルの名前と、計算されたカラムのリストが表示されます。

抽出テーブルには、アクセス・プラン内で、クエリの結果を変更しないで文のアクセス・プランの他の部分にマージ (フラット処理) できない部分が含まれます。抽出テーブルは、元の文で指定されている抽出テーブルのセマンティックを適用するために使用され、特にクエリに 1 つ以上の外部ジョインがある場合にクエリ書き換えの最適化などの理由でプランに表示されます。

抽出テーブルの詳細については、「FROM 句：テーブルの指定」 295 ページと「FROM 句」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

例

次のクエリは、グラフィカルなプランに抽出テーブルがあります。

```
SELECT EmployeeID FROM Employees
UNION ALL
SELECT DepartmentID FROM (
   SELECT TOP 5 DepartmentID
   FROM Departments
   ORDER BY DepartmentName DESC ) MyDerivedTable;
```

交換アルゴリズム

交換アルゴリズムは、SELECT 文の処理時にクエリ内並列処理を実装するために使用されます。交換演算子にはサブツリーが 2 つ以上あり、それぞれのサブツリーが並列に実行されます。各サブツリーを実行すると、交換の親演算子によって消費されるローのバッファが満杯になります。交換の結果は、その子の結果の共用体です。交換の各子は、親と同様に 1 タスクを使用します。そのため、2 つの子がある単一の交換を使用するプランでは、3 つのタスクを実行する必要があります。

交換アルゴリズムは、SELECT 文の処理時に、クエリ内並列処理が有効な場合にのみ使用されます。

並列処理の詳細については、「SQL Anywhere でのスレッド」『SQL Anywhere サーバ - データベース管理』を参照してください。

フィルタと事前フィルタ・アルゴリズム

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。探索条件は、文の WHERE 句と HAVING 句、JOIN の ON 条件の FROM 句にあります。

オプティマイザでは、探索条件内の一連の述部が任意に簡素化、変更されます。また、元の文で指定された順序とは別の順序で条件が適用されるアクセス・プランが作成される場合もあります。クエリ書き換え最適化では、プラン内で評価された一連の述部が変更される場合があります。

クエリ内に述部があっても、アクセス・プランにフィルタ・アルゴリズムがない場合もあります。たとえば、インデックス・スキャンなどのさまざまなアルゴリズムでは、明示的な演算子がなくても、述部を適用できます。具体的な例として、BETWEEN 述部に 2 つのリテラル定数があり、述部で参照されているカラムにインデックスがあるとします。この場合、BETWEEN 述部は、インデックス・スキャンの下限と上限で適用でき、クエリのプランには明示的なフィルタ・アルゴリズムは含まれません。ジョイン条件である述部も、アクセス・プランでフィルタになりません。

事前フィルタ・アルゴリズムは、事前フィルタの述部で使用される式が、クエリで参照されているテーブルまたはビューに依存しないことを除き、フィルタ・アルゴリズムと同じです。簡単な例として、WHERE 1 = 2 句の探索条件は事前フィルタに評価できます。

参照

- 「WHERE 句 : ローの指定」 296 ページ
- 「EXISTS 探索条件」 『SQL Anywhere サーバ - SQL リファレンス』

ハッシュ・フィルタ・アルゴリズム

ハッシュ・フィルタまたはハッシュ・マップは、ブルーム・フィルタとも呼ばれるデータ構造体で、単一カラムまたはカラム・セット内の値の分散を表します。ハッシュ・フィルタは、(長い) ビット文字列と考えることができます。1 ビットは特定のローが存在することを示し、0 ビットはそのビット位置にローがないことを示します。ローのセットからフィルタ内のビット位置に値をハッシュすることで、データベース・サーバで、その値の一致するローがあるかどうかを簡単に判別できます (ハッシュの衝突が存在することを条件として)。

次のプランを例にとります。

```
R<idx> *JH S<seq> JH* T<idx>
```

この例では、R を S と T とジョインします。データベース・サーバで R のローをすべて読み込んでから、T のローを読み込み、R とジョインできない T のローを直ちに拒否します。これによって、2 番目のハッシュ・ジョインで格納するローの数を減らすことができます。

ハッシュ・フィルタは、次の両方の条件を満たすクエリ内で使用できます。

- クエリ内の操作が入力全体を読み込んでから、後の操作にローを返す場合。たとえば、1 つのカラムに基づいて 2 つのテーブルのハッシュ・ジョインを行うには、一方の入力のすべての関連するローを読み込み、ジョインのハッシュ・テーブルを作成する必要があります。

Copyright © 2007, iAnywhere Solutions, Inc.

- クエリのアクセス・プラン内の後続の操作が、その操作の結果内のローを参照する場合。たとえば、最初のジョインと同じカラムに基づく 2 つめのジョインは、最初のジョインを満たすローのみを使用します。

この場合、最初のジョインの結果として構築されたハッシュ・フィルタによって、2 つめのジョインのパフォーマンスが向上します。このとき、ハッシュ・フィルタのビット文字列内でルックアサイドが行われ、最初のジョインで正常に処理されたローがあるかどうかが確認されます。正常に処理されたローがない場合は、ハッシュ・フィルタ内に 1 ビットがなければ、2 つめのジョインでハッシュ・テーブルを調査しても一致するローは見つからないことがわかっているので、調査を完全に回避できます。

IN リスト

IN リスト・アルゴリズムは、インデックスを使用して IN 述部を満たすことができる場合に使用されます。たとえば、次のクエリの場合、オプティマイザはプライマリ・キー・インデックスを使用して Employees テーブルにアクセスできることを認識します。

```
SELECT *
FROM Employees
WHERE EmployeeID IN ( 102, 105, 129 );
```

このアクセスを行うために、左側に特別な IN リスト・テーブルを指定したジョインが構築されます。ローは IN リストからフェッチされ、Employees テーブルの調査に使用されます。同じインデックスを使用して、複数の IN リストの条件を満たすことができます。オプティマイザがあるインデックスを使用しないで IN 述部を満たすことを選択した場合 (別のインデックスの方が高パフォーマンスを提供する可能性があるなどの理由で)、IN リストはフィルタ内の述部になります。

ProcCall アルゴリズム

ProcCall アルゴリズムは、FROM 句内のプロシージャに使用されるもので、プロシージャ・コールを実行し、結果セットでローを返します。後方のフェッチはできないので、カーソル・タイプで必要な場合はワーク・テーブルの下に表示されます。

ProcCall が実行されると、データベース・サーバで引数の値、返されるローの数、すべてのローのフェッチに要した合計時間が記録されます。オプティマイザでは、この情報を使用して後続のプロシージャ・コールのコストとカーディナリティが予測されます。データベース・サーバでは、プロシージャごとに返されるローの数の移動平均と合計実行時間の移動平均が保持されます。また、限られた数の特定の引数の値ごとの移動平均も保持されます。この情報は SYSPROCEDURE システム・テーブルの stats カラムに永続的に格納されます。値はバイナリ形式で、内部でのみ使用されます。

複数の結果セットに対する制限と、スキーマ一致の要件については、「FROM 句」『SQL Anywhere サーバ - SQL リファレンス』で FROM 句の procedure 句に関する説明を参照してください。

参照

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

Copyright © 2007, iAnywhere Solutions, Inc.

◆ 「プロシージャ統計」 525 ページ

ロー・コンストラクタ・アルゴリズム

ロー・コンストラクタ・アルゴリズムは、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。ロー・コンストラクタ・アルゴリズムは、次の 2 とおりの方法で使用されます。

- INSERT ... VALUES 文では、VALUES 句で参照されている式 (通常はリテラル定数かホスト変数、またはその両方) から、挿入される仮想ローが作成されます。この場合、グラフィカルなプランで INSERT の下にロー・コンストラクタが表示されます。
- システム・テーブル SYS.DUMMY への直接または間接的な参照は自動的にロー・コンストラクタ・アルゴリズムに変換され、SYS.DUMMY に対するスキャン・アルゴリズムに置き換わり、DUMMY テーブルの (単一) ページをラッチする必要がなくなります。

短いテキスト・プランまたは長いテキスト・プランの場合、テーブル・スキャンがロー・コンストラクタ・アルゴリズムに置き換わっても、プランの文字列にテーブル SYS.DUMMY への参照が含まれます。

参照

◆ 「DUMMY システム・テーブル」 『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』
◆ 「実行プランの解釈」 571 ページ

ローの制限アルゴリズム

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。

詳細については、「SELECT 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Window 関数アルゴリズム

このアルゴリズムは、Window 関数を使用する OLAP クエリを評価するのに使用します。

詳細については、「Window 関数」 425 ページを参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

実行プランの解釈

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手順のセットです。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限りません。ただし、セマンティック上は同じであり、クエリで明示的に指定したベース・テーブルの代わりに実体化ビュー (Materialized View) を使用できます。

文が実行されるまでに経由するフェーズの詳細については、「クエリ処理のフェーズ」 510 ページを参照してください。

データベース・サーバがクエリを書き換える場合に従う規則の詳細については、「セマンティック・クエリ変形」 512 ページと「実体化ビュー (Materialized View) によるパフォーマンスの向上」 531 ページを参照してください。

オプティマイザの仕事は、クエリのセマンティックを把握し、その結果を計算するプランを構築することです。アクセス・プランは使用されている構文と正確に一致しないことがあります。オプティマイザは、セマンティック上等しいフォームであれば、クエリを自由に書き換えることができます。

SQL Anywhere がクエリを書き換える場合に従う規則の詳細については、「サブクエリを EXISTS 述部として書き換える」 521 ページと「セマンティック・クエリ変形」 512 ページを参照してください。

オプティマイザがクエリの実装に使用する方式については、「クエリ実行アルゴリズム」 546 ページを参照してください。

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。

- 短いプラン
- 長いプラン
- 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)

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。「GRAPHICAL_PLAN 関数 [その他]」『SQL Anywhere サーバ - SQL リファレンス』と「EXPLANATION 関数 [その他]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

プランの保存と確認の方法の詳細については、「実行プランへのアクセス」 588 ページを参照してください。

実行プランを読み込む方法については、「テキスト・プラン」 580 ページと「グラフィカルなプラン」 581 ページを参照してください。

アクセス・プランに表示される統計とその他の項目について、次に説明します。

Copyright © 2007, iAnywhere Solutions, Inc.

プランに使用される省略形

次の表に、短いプランと、グラフィカルなプランの省略名のフォームに使用される省略形を示します。

省略形	名前
DELETE	削除
DistH	ハッシュ DISTINCT
DistO	順序付き DISTINCT
DT	抽出テーブル
EAH	ハッシュ EXCEPT ALL
EAM	マージ EXCEPT ALL
EH	ハッシュ EXCEPT
EM	マージ EXCEPT
交換	交換
フィルタ	フィルタ
FS	ファイル・スキャン
GrByH	ハッシュ GROUP BY
GrByHClust	ハッシュ Group By クラスタード
GrByHP	並列ハッシュ Group By
GrByHSets	ハッシュ Group By セット
GrByO	順序付き GROUP BY
GrByOSets	順序付けされた Group By セット
GrByS	単一ロー GROUP BY
GrBySSets	ソートされた Group By セット
HF	ハッシュ・フィルタ
HFP	並列ハッシュ・フィルタ
HTS	ハッシュ・テーブル・スキャン
IAH	ハッシュ INTERSECT ALL
IAM	マージ INTERSECT ALL

 Copyright © 2007, iAnywhere Solutions, Inc.

省略形	名前
IH	ハッシュ INTERSECT
IM	マージ INTERSECT
IN	IN リスト
INSENSITIVE	大文字小文字の区別
INSERT	挿入
IS	インデックス・スキャン 短いプランの場合は、*table-name* に続いて <*rowID*>、<*seq*>、<*rowID*> のいずれか、グラフィカルなプランの場合は *table-name*
ISP	並列インデックス・スキャン
JE	EXISTS ジョイン
JH	ハッシュ・ジョイン
JHE	Exists ハッシュ・ジョイン
JHEP	Exists 並列ハッシュ・ジョイン
JHFO	全外部ハッシュ・ジョイン
JHNE	Exists ハッシュ・ジョイン以外
JHNEP	Exists 並列ハッシュ・ジョイン以外
JHO	左外部ハッシュ・ジョイン
JHP	並列ハッシュ・ジョイン
JHPO	並列左外部ハッシュ・ジョイン
JHR	再帰ハッシュ・ジョイン
JHRO	再帰左外部ハッシュ・ジョイン
JM	マージ・ジョイン
JMFO	全外部マージ・ジョイン
JMO	左外部マージ・ジョイン
JNL	ネスト・ループ・ジョイン
JNLFO	全外部ネスト・ループ・ジョイン
JNLO	左外部ネスト・ループ・ジョイン

省略形	名前
KEYSET	キーセット
LOAD	ロード
PC	プロシージャ・コール (テーブル関数)
PreFilter	事前フィルタ
RowID Scan	ロー識別子スキャン 短いプランの場合は *table-name* <*rowID*>、グラフィカルなプランの場合は *table-name*
ROWS	ロー・コンストラクタ
RL	ロー制限
RR	ロー・レプリケート
RT	再帰テーブル
RU	再帰 UNION
SELECT	選択
Sort	ソート (インデックスまたはマージ)
SrtN	ソート・トップ N
TS	テーブル・スキャン 短いプランの場合は *table-name* <*seq*>、グラフィカルなプランの場合は *table-name*
TSP	並列テーブル・スキャン
UA	UNION ALL
UPDATE	更新
ウィンドウ	ウィンドウ
Work	ワーク・テーブル

各アルゴリズムの説明については、「クエリ実行アルゴリズム」 546 ページを参照してください。

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
Invocations	ローがサブツリーから要求された回数。

 Copyright © 2007, iAnywhere Solutions, Inc.

統計情報	説明
RowsReturned	現在のノードについて返されたローの数。
RunTime	子の時間を含めたサブツリーの実行所要時間。
CacheHits	成功したキャッシュ読み込み数。
CacheRead	キャッシュの中で検索されたデータベース・ページの数。
CacheReadTable	キャッシュから読み込まれたテーブル・ページの数。
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。
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。
DiskRead	ディスクから読み込まれたページ数。
DiskReadTable	ディスクから読み込まれたテーブル・ページの数。
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。
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。
DiskWrite	ディスクに書き込まれたページ (ワーク・テーブル・ページまたは修正されたテーブル・ページ) の数。
IndAdd	インデックスに追加されたエントリの数。
IndLookup	インデックスの中で検索されたエントリの数。
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。

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)。
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。

統計情報	説明
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_goal オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。
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。「optimization_workload オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。
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)。「ansi_update_constraints オプション [互換性]」『SQL Anywhere サーバ - データベース管理』を参照してください。
Optimization level	予約。
Select list	クエリによって選択される式のリスト。

Copyright © 2007, iAnywhere Solutions, Inc.

項目	説明
Materialized views	オプティマイザによって検討される実体化ビュー (Materialized View) のリスト。リスト内の各エントリは、*view-name* [*view-matching-outcome*] [*table-list*] というフォーマットの組です。ここで *view-matching-outcome* は実体化ビュー (Materialized View) 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 ◆ パーミッションが一致しません ◆ 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 ◆ しきい値に一致するビューに到達しました ◆ ビューは使用されています オプティマイザによる実体化ビュー (Materialized View) の使用を妨げる制約と条件の詳細については、「実体化ビュー (Materialized View) によるパフォーマンスの向上」 531 ページと「実体化ビュー (Materialized View) を管理するときの制限」 74 ページを参照してください。

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。
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。

Copyright © 2007, iAnywhere Solutions, Inc.

項目	説明
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
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。「テーブルとページのサイズ」 591 ページを参照してください。
Estimated leaf pages	リーフ・ページの推定数。
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。
Selectivity	範囲バウンドと一致する推定ロー数。
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。

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。「プラン内の選択性」 587 ページを参照してください。

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

項目	説明
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。

テキスト・プラン

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。Interactive SQL でテキスト・プランのタイプを選択するには、[ツール] メニューから [オプション] ダイアログを開き、[プラン] タブをクリックします。SQL 関数を使用してテキスト・プランにアクセスする方法については、「SQL 関数を使用した実行プランへのアクセス」 589 ページを参照してください。

プランには、グラフィカルなバージョンもあります。「グラフィカルなプラン」 581 ページを参照してください。

短いテキスト・プラン

短いプランは、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 JM 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 GivenName, Surname, OrderDate
FROM Customers JOIN SalesOrders
WHERE CustomerID < 100
ORDER BY OrderDate;
```

```
Work[ Sort[ HF[ Customers<CustomersKey> ] JM SalesOrders<FK_CustomerID_ID> ] ]
```

プランに使用されるコード・ワードの詳細については、「プランに使用される省略形」 572 ページを参照してください。

ジョイン方式の区切りにコロンを使用

次に示すコマンドには、「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 *
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

Copyright © 2007, iAnywhere Solutions, Inc.

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。

プランに使用される省略形の詳細については、「プランに使用される省略形」 572 ページを参照してください。

長いテキスト・プラン

長いプランでは、短いプランより若干多くの情報が提供されます。また、情報は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。

次の例では、長いプランの最初の行は Plan[Total Cost Estimate: 1.040539E-5] です。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。このプランは、結果がソートされ、マージ・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、TRUE という語または残りの探索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。HashFilter は、マージ・ジョインの右側にハッシュ・フィルタが構築されることを示します。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。

```
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;

( Plan [ Total Cost Estimate: 1.040539E-5 ]
  ( WorkTable
    ( Sort
      ( MergeJoin [ ( ((Customers.Country LIKE 'Canada' : 100% Computed) AND (Customers.Country =
'Canada' : 7.936508209% Statistics)) OR ((SalesOrders.Region LIKE 'Eastern' : 100% Computed) AND
(SalesOrders.Region = 'Eastern' : 5% Guess)) ) : 100% Guess ]
        ( 1 HashFilter
          ( IndexScan Customers CustomersKey )
        )
        ( IndexScan SalesOrders FK_CustomerID_ID[ hash( SalesOrders.CustomerID ) in hashmap
( Customers.ID ) : 100% Guess ] )
      )
    )
  )
)
```

プランに使用される省略形の詳細については、「プランに使用される省略形」 572 ページを参照してください。

グラフィカルなプラン

グラフィカルなプランでは、実行プランの情報が視覚的に表されます (形、色など)。グラフィカルなプランを表示するか、統計情報付きのグラフィカルなプランを表示するかを選択できます。どちらを選択した場合も、プランの中で最も高コストとして推定された部分をすぐに表示できます。統計情報付きのグラフィカルなプランの表示は高コストですが、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、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

Copyright © 2007, iAnywhere Solutions, Inc.

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、実際の統計と推定とに違いがあると予想してください。

SQL Anywhere では、デフォルトで統計情報なしのグラフィカルなプランが表示されます。グラフィカルなプランへの統計情報の表示を切り替えるには、[ツール] − [オプション] を選択し、[プラン] タブをクリックし、プランのタイプを選択します。SQL 関数を使用してプランにアクセスする方法については、「SQL 関数を使用した実行プランへのアクセス」 589 ページを参照してください。

テキスト・プランの詳細については、「テキスト・プラン」 580 ページを参照してください。

グラフィカルなプランは、主要な情報の一部を視覚的に提供することを目的として設計されています。たとえば、グラフィカルなプランでは各操作がコンテナに入っています。コンテナは、ノードとも呼ばれます。次のように、コンテナの形は操作のタイプを示します。

- 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

通常は、グラフィカルなプラン内で太い線と赤い枠線があるかどうかでクエリのパフォーマンスを確認できます。次に例を示します。

- プランでは、操作が次の操作に渡すローの数が、操作を結んでいる線の太さによって示されています。これは、クエリ内の大半のデータに対して実行される操作を視覚的に表します。
- 特に遅い操作のコンテナは、赤い枠線で表示されます。

グラフィカルなプランの表示のカスタマイズ

グラフィカルなプラン内の項目の表示はカスタマイズできます。グラフィカルなプランの表示を変更するには、Interactive SQL の左下のウィンドウ枠でプランを右クリックし、[カスタマイズ] を選択し、設定を変更します。プランの表示をカスタマイズするには、その前に実行する必要があります。変更内容は、グラフィカルなプランが次に表示されるときに反映されます。

グラフィカルなプランを右クリックし、[印刷] を選択すると、プランを印刷できます。

クエリと、対応するグラフィカルなプランを次に示します。図はツリー形式になっており、各ノードがすぐ下のノードからのローを要求することを示しています。

 Copyright © 2007, iAnywhere Solutions, Inc.

コンテキスト別のヘルプ

グラフィカルな図でノードをクリックし、右ウィンドウ枠に表示される対応する情報を読むことで、プラン内のノードに関する詳細情報を取得できます。この例では、nested loops join (JNL) ノードが選択されています。右ウィンドウ枠には、そのノードに関連する情報だけが表示されています。たとえば、述部は TRUE になっています。これは、クエリ実行のこの段階では述部が適用されていないことを示します。Customers テーブルのクエリ・ノードをクリックすると、述部の値が Customers.ID > 100 : 100% Index; true 126/126 100% のようになります。

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、ノードを選択し、右クリックして [ヘルプ] を選択します。

プランに使用される省略形の詳細については、「プランに使用される省略形」 572 ページを参照してください。

注意
クエリが単純なクエリとして認識されている場合、一部の最適化手順が省略され、クエリ・オプティマイザ・セクションと述部セクションの両方がグラフィカルなプランに表示されません。省略されるクエリの詳細については、「オプティマイザの仕組み」 523 ページを参照してください。

統計情報付きのグラフィカルなプラン

統計情報付きのグラフィカルなプランには、グラフィカルなプランに表示されるすべての推定に加えて、文を実行したときの実際の実行時間コストが表示されます。これを行うには、文を実行します。これは、コストの高いクエリの場合、プランへのアクセスに遅延が生じる可能性があることを意味します。また、削除や更新など、クエリのどのような部分も実際に実行されます。ロールバックを利用すればこれらの変更は取り消し可能です。

パフォーマンスに問題があり、推定されるロー・カウントや実行時間が予測とは異なる場合は、統計情報付きのグラフィカルなプランを使用してください。統計情報付きのグラフィカルなプランを使用すると、比較するために推定と実際の統計が提供されます。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適切なアクセス・プランを選択できないことを示す警告です。

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は、ルート演算子ノードについてのみ表示されます。

次に、統計情報付きのグラフィカルなプランでチェックできる主な統計と、考えられる対応策を示します。

- **選択性統計値**　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。選択性推定が不正確な場合、クエリ・オプティマイザは不適切なアクセス・プランしか生成できません。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、実際の選択性がかなり低い (50%) 場合は、不適切なプランを選択してしまう可能性があります。選択性推定は正確なものではありませんが、極端に大きくはずれている場合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 を使用して対象となる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他の問題が発生する可能性があります。

  選択性の詳細については、「プラン内の選択性」 587 ページを参照してください。

  統計作成の詳細については、「CREATE STATISTICS 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

  ユーザ推定の詳細については、「明示的な選択性推定」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

  選択性統計は、クエリが単純なクエリと判断された場合に表示されないため、オプティマイザは省略されます。単純なクエリの詳細については、「オプティマイザの仕組み」 523 ページを参照してください。

  次のような場合に不適切な選択性の兆候が見られます。

 Copyright © 2007, iAnywhere Solutions, Inc.

- **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。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動作していることを示す警告とみなすことができます。
- **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。述部情報の解釈については、「プラン内の選択性」 587 ページを参照してください。

  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。「CREATE STATISTICS 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

  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。

  推定ソースが「予測」の場合、オプティマイザが使用する情報がないことを示します。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。「REORGANIZE TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
  選択性推定に考えられるソースの完全なリストについては、「ESTIMATE_SOURCE 関数 [その他]」 『SQL Anywhere サーバ - SQL リファレンス』 を参照してください。

- **キャッシュの読み込み数とヒット数**　キャッシュの読み込み数とヒット数がまったく同じ場合は、クエリを実行するのに必要なすべての情報がキャッシュにあります。読み込み数がヒット数よりも多い場合は、データベース・サーバがキャッシュにアクセスしようとしたが失敗し、ディスクから読み込むことを意味します。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・ヒット率が低いことは、パフォーマンスの問題を示す場合があり、キャッシュ・サイズを大きくすると改善できる可能性があります。

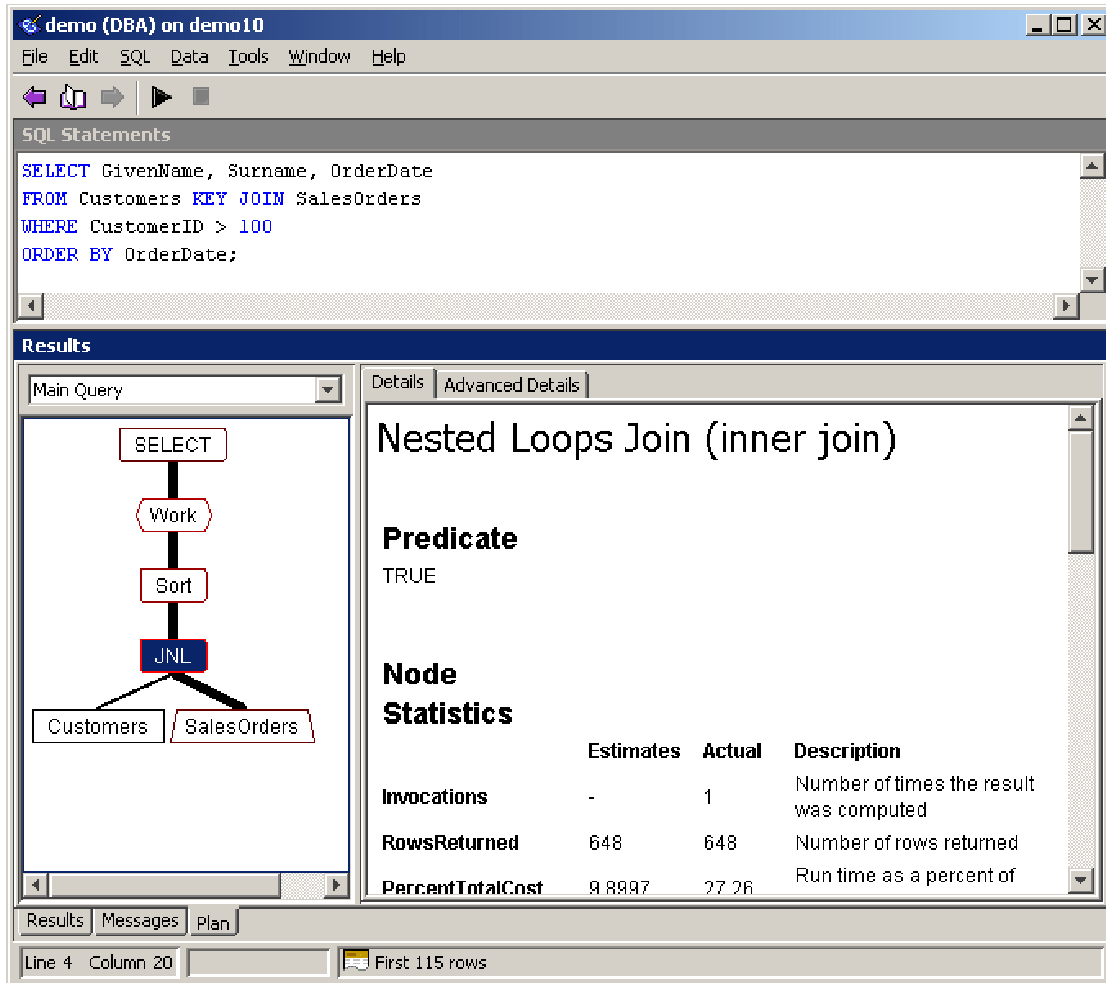
  キャッシュ管理の詳細については、「キャッシュ・サイズの拡大」 259 ページを参照してください。
- **有効なインデックスの不足**　クエリ実行プランからは、インデックスが良好なパフォーマンスの提供に役立っているかどうかが不明な場合があります。SQL Anywhere で使用されているスキャンベースのアルゴリズムの中には、インデックスを使用せずに多くのクエリに対して最高のパフォーマンスを提供するものもあります。

  インデックスとパフォーマンスの詳細については、「インデックスの有効な使用」 271 ページと「インデックス・コンサルタント」 210 ページを参照してください。
- **データの断片化の問題**　実行時間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。実行時間は、クエリの実行所要時間を測定します。テーブル・スキャンまたはインデック

ス・スキャンの実行時間が正しくない場合は、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。

詳細については、「REORGANIZE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』と「テーブルの断片化削減」265 ページを参照してください。

次の例に、統計情報付きのグラフィカルなプランを示します。この例でも、nested loops join ノードが選択されています。右ウィンドウ枠の統計は、クエリのその部分で使用されているリソースを示します。



プランに使用されるコード・ワードの詳細については、「プランに使用される省略形」572 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

プラン内の選択性

次の例では、探索条件の選択性を示す Predicate (述部) を示します。この例では、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。

最適化を省略できる単純な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。単純なクエリの詳細については、「オプティマイザの仕組み」 523 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。

demo (DBA) on demo10

File Edit SQL Data Tools Window Help

SQL Statements

```
SELECT *
FROM Departments
WHERE DepartmentName = 'Sales'
```

Results

Main Query

SELECT

Departments

Details | Advanced Details

Table Scan

Scan Departments sequentially

Table reference

Creator name	GROUPO
Table name	Departments
Estimated rows	5
Estimated pages	1
Estimated pages in cache	0
Estimated row size (bytes)	47
Page map	yes
Buffer fetch	no
Relax cursor stability	no
Lock	Isolation level 0

Predicate

DepartmentName = 'Sales' : 20% Column; true 1/5 20%

Results | Messages | Plan

Line 3 Column 31 | 1 rows

この述部記述は次のとおりです。

```
Departments.DepartmentName = 'Sales' : 20% Column; true 1/5 20%
```

Copyright © 2007, iAnywhere Solutions, Inc.

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。詳細については、「ESTIMATE 関数 [その他]」『SQL Anywhere サーバ - SQL リファレンス』 を参照してください。
- 推定ソースは「Column」です。これは、ESTIMATE_SOURCE 関数で得られる出力と同じです。選択性推定に考えられるソースの完全なリストについては、「ESTIMATE_SOURCE 関数 [その他]」『SQL Anywhere サーバ - SQL リファレンス』 を参照してください。
- 「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り、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。

実行プランへのアクセス

Interactive SQL または SQL 関数を使用して実行プランを表示できます。プランは、インデックス・コンサルタントでも表示できます。

Interactive SQL を使用した実行プランへのアクセス

Interactive SQL では、次の種類のプランを使用できます。

- 短いテキスト・プラン (「短いテキスト・プラン」 580 ページを参照)
- 長いテキスト・プラン (「長いテキスト・プラン」 581 ページを参照)
- 統計情報付きまたは統計情報なしのグラフィカルなプラン (「グラフィカルなプラン」 581 ページを参照)

別のタイプのプランを選択するには、[ツール] – [オプション] を選択し、[プラン] タブをクリックしてから、プランのタイプを選択します。

プランを表示するには、クエリを実行し、Interactive SQL ウィンドウ下部にある [プラン] タブをクリックします。

Copyright © 2007, iAnywhere Solutions, Inc.

SQL 関数を使用した実行プランへのアクセス

SQL 関数を使用して実行プランにアクセスし、出力を XML フォーマットで取り出すことができます。

- 短いプランにアクセスする方法については、「EXPLANATION 関数 [その他]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
- 長いプランにアクセスする方法については、「PLAN 関数 [その他]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
- グラフィカルなプランにアクセスする方法については、「GRAPHICAL_PLAN 関数 [その他]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

物理データの編成とアクセス

各テーブルやエントリに対する記憶領域の割り付けは、クエリの効率に大きく影響します。次に示す各項目は、クエリの実行速度に影響するため、特に重要です。

挿入されたローに対するディスク割り付け

ここでは、データベース内のローがディスクに保存される方法について説明します。

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表す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。詳細については、「CREATE TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

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、特にローの挿入時に SQL Anywhere が追加領域を許可しない点には、注意が必要です。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。

また、各ページの空き領域のレコードを保持しています。新しいローを挿入するよう要求すると、Adaptive Server Anywhere は、まず既存のページの領域のレコードを検索します。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。

ただし、多くのローを削除し、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します。

詳細については、「REORGANIZE TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

テーブルとページのサイズ

データベースに対して選択したページ・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。通常、ページ・サイズが小さいと、ランダムな場所から比較的少ない数のローを取り出す操作に有利な傾向があります。反対に、大きなページは、特にローがインデックスを使用して取り出される順序でページに保管されている場合に、逐次テーブル・スキャンを実行するクエリに有利な傾向があります。この場合、メモリに 1 ページを読み込んで 1 つのローの値を取得すると、次のいくつかのローの内容をメモリにロードできるという 2 次的な効果があります。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。

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**」とよばれるこの方法で、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。

8 KB などの大きなページ・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。たとえば、1 MB のメモリには、1 ページを 2 KB とすると 512 ページ格納できますが、1 ページ 8 KB であれば 128 ページしか格納できません。ページ・サイズのキャッシュ・サイズに対する適切なページ比は、データベースと、アプリケーションで実行されるクエリの性質によって異なります。さまざまなキャッシュ・サイズで、パフォーマンス・テストを実行できます。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。これは、Windows CE デバイスで SQL Anywhere を使用する場合に特に重要です。ページ・サイズが大きいと、内部の断片化が増える可能性があります。

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。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・サイズの選択が特に重要です。

参照

- 「適切なページ・サイズの使用」 269 ページ
- 「初期化ユーティリティ (dbinit)」 『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』

インデックス

インデックスによって、インデックス・カラムの検索パフォーマンスが大幅に向上します。ただし、インデックスはデータベース内の領域を占有し、また、挿入、更新、削除操作の速度を低下させます。この項では、インデックスの作成が必要な場合を判断し、インデックスから最大限のパフォーマンスを得る方法について説明します。

インデックスを作成すると、データベースのパフォーマンスが向上することがよくあります。インデックスは、一部またはすべてのカラムの値を基に、テーブルのローに順序を付けます。インデックスによって、SQL Anywhere はローを迅速に見つけることができます。インデックスは、アクセスされるデータベース・ページの数を制限して、同時実行性を高めます。また、インデックスは SQL Anywhere に、テーブル内のローに一意性制約を強制するための便利な手段を提供します。

Copyright © 2007, iAnywhere Solutions, Inc.

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。

インデックス・コンサルタントは、使用中のデータベースに最適なインデックス・セットを選択する手助けをするツールです。「インデックス・コンサルタント」 210 ページを参照してください。

論理インデックスを使用したインデックスの共有

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 ISYSPHYIDX 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 ISYSPHYIDX システム・テーブルに定義されます。

ISYSIDX システム・テーブルと ISYSPHYIDX システム・テーブルの詳細については、それぞれの対応するビューである「SYSIDX システム・ビュー」 『SQL Anywhere サーバ - SQL リファレンス』と「SYSPHYSIDX システム・ビュー」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

物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。

インデックスを作成する前に、そのインデックスを作成する必要があるかどうかを慎重に検討してください。「どのようなときにインデックスを作成するか」 595 ページを参照してください。

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、ISYSPHYIDX 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。

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限りません。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 [Phys. ID] 値もリスト内の別のインデックスのために存在するかどうかに注意してください。[Phys. 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。または、すべてのインデックスを削除してから再作成します。

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
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。これを行うには、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 [Phys. ID] カラムの値が同じである複数のローを探します。[Phys. 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ます。

Copyright © 2007, iAnywhere Solutions, Inc.

参照

- 「インデックスの再構築」 92 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』

どのようなときにインデックスを作成するか

インデックスの作成が必要かどうかは、単純な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。

  詳細については、「複合インデックス」 597 ページを参照してください。

- **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・コンサルタントは、使用中のデータベースに最適なインデックス・セットを選択する手助けをするツールです。「インデックス・コンサルタント」 210 ページを参照してください。

テンポラリ・テーブル

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

Copyright © 2007, iAnywhere Solutions, Inc.

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。

詳細については、「インデックスの編集」 87 ページを参照してください。

インデックスのパフォーマンス向上

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

この 2 つの措置は、次に説明するように、インデックスの選択性とインデックス・ファンアウトを高めることを目的としています。

インデックスの選択性

「**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 プロパティ関数は、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 CE 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。詳細については、「プランに使用される一般的な統計」 574 ページを参照してください。

FullCompare 関数の詳細については、「データベース・レベルのプロパティ」 『SQL Anywhere サーバ - データベース管理』を参照してください。

インデックス構造とインデックス・ファンアウト

インデックスは、ツリーのように多数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

Copyright © 2007, iAnywhere Solutions, Inc.

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。「テーブルとページのサイズ」 591 ページを参照してください。

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。「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』を参照してください。

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必要がある場合、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 コマンドを使用してインデックスを作成した場合、カラムはコマンド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;
```

 Copyright © 2007, iAnywhere Solutions, Inc.

インデックスのその他の使用法

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て、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。

  インデックスとロックの詳細については、「ロックの仕組み」 166 ページを参照してください。
- **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。

インデックスのタイプ

インデックスは、クラスタード・インデックスまたは非クラスタード・インデックスとして宣言されます。特定のテーブル上で 1 つのインデックスのみ、クラスタ化できます。インデックスがクラスタ化されることを特定した場合は、インデックスを削除して再作成する必要はありません。ALTER INDEX 文を発行することで、インデックスのクラスタリング特性が削除または追加されます。クラスタード・インデックスでは、クエリ・オプティマイザはインデックス・スキャンのコストについてより正確に判断できるため、パフォーマンスが向上する可能性があります。

SQL Anywhere では、B リンク・インデックスが実装されます。

参照

- 「クラスタード・インデックスの使用」 88 ページ
- 「ALTER INDEX 文」 『SQL Anywhere サーバ - SQL リファレンス』

B リンク・インデックス

B リンク・インデックスは、B- と B+- ツリー・インデックスの変形であり、各インデックス・ページ (非リーフとリーフ) に、右の兄弟のページ番号またはリンクが含まれます。また、インデックス・ページが親ページに表示される必要がありません。B リンク・インデックスの最大の利点は、同時実行性が向上することです。

SQL Anywhere では、ファンアウトを向上するために、インデックス値の圧縮形式が格納されます。この形式には、直前の値と共通のプレフィクスは含まれません。ページ内を検索するときの CPU 時間を短縮するため、完全なインデックス・キーの小さなルックアサイド・マップ (データ長制限の対象) も格納されます。特に、SQL Anywhere のインデックスでは、同じ (またはほぼ同じ) インデックス値が効率的に処理されます。そのため、インデックス値に含まれる共通のプレフィクスは、必要記憶域とパフォーマンスにほとんど影響しません。

パート IV. SQL のダイアレクトと互換性

パート IV 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。

第 15 章

他の SQL ダイアレクト

目次

SQL Anywhere の準拠の概要 604
SQL FLAGGER を使用した SQL 準拠のテスト 605
他の SQL ソフトウェアにはない機能 608
Transact-SQL との互換性 610
Adaptive Server のアーキテクチャ 613
Transact-SQL との互換性を意識したデータベースの設定 619
互換性のある SQL 文の記述方法 626
Transact-SQL のプロシージャ言語の概要 631
ストアド・プロシージャの自動変換 634
Transact-SQL プロシージャから返される結果セット 635
Transact-SQL プロシージャの中の変数 636
Transact-SQL プロシージャでのエラー処理 637

Copyright © 2007, iAnywhere Solutions, Inc.

SQL Anywhere の準拠の概要

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/ANSI SQL-2003 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。

 Copyright © 2007, iAnywhere Solutions, Inc.

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、SQL/1999 と SQL/2003 の ISO/ANSI SQL 標準で定められています。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を確認できます。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。

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/2003 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。

2 つのテーブル間でのプライマリ・キーと外部キーの暗黙的なジョインもベンダ拡張として通知されます。明示的な KEY JOIN 構文の使用は、構文上のベンダ拡張であり、Ultra Light でサポートされていません。また、ON 句を省略して JOIN キーワードを使用したジョインである暗黙的なキー・ジョインもベンダ拡張として通知されます。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

- 「SQL プリプロセッサの実行」『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」377 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。「SQLFLAGGER 関数 [その他]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **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/2003 または SQL/1999 国際標準のオプション・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。「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』を参照してください。

- **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_flagger_error_level オプション [互換性]」『SQL Anywhere サーバ - データベース管理』と「sql_flagger_warning_level オプション [互換性]」『SQL Anywhere サーバ - データベース管理』を参照してください。

- **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。「SQL プリプロセッサ」『SQL Anywhere サーバ - プログラミング』と「SQL プリプロセッサの実行」『SQL Anywhere サーバ - プログラミング』を参照してください。

参照

- 「バッチの概要」 799 ページ

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/2003 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 要素が次の ANSI SQL 標準に準拠するかどうかを確認できます。

- 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

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、実体化ビュー (Materialized View)、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 標準のベンダ拡張なので、準拠しないと通知されます。

Copyright © 2007, iAnywhere Solutions, Inc.

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 の SQL 要素のリファレンス」 『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 ソフトウェアにはない機能

次の SQL Anywhere の機能は、他の多くの SQL ソフトウェアには実装されていません。

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。

日付と時刻の演算に使用できる関数は、このほかにも数多くあります。詳細については、「SQL 関数」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

整合性

SQL Anywhere は、エンティティ整合性と参照整合性の両方をサポートします。これは、次に示す 2 つの拡張機能を使って、CREATE TABLE と ALTER TABLE コマンドに実装されます。

```
PRIMARY KEY ( column-name, ... )
[NOT NULL] FOREIGN KEY [role-name]
        [(column-name, ...)]
      REFERENCES table-name [(column-name, ...)]
        [ CHECK ON COMMIT ]
```

PRIMARY KEY 句では、関係のためのプライマリ・キーを宣言します。SQL Anywhere は宣言されたプライマリ・キーがユニークであり、どのカラムも NULL 値を含まないよう管理します。

FOREIGN KEY 句は、このテーブルと他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。システムは、これらのカラムが参照整合性を保つようにします。カラムが変更されたりテーブルにローが挿入されたりすると、これらのカラムを調べて、NULL のものはないか、また、値が他のテーブルの一部のローのプライマリ・キーにある対応するカラムと一致するか確認します。詳細については、「CREATE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ジョイン

SQL Anywhere は、テーブル間の**自動ジョイン**をサポートします。他のソフトウェアでもサポートされる NATURAL ジョインと OUTER ジョインの演算子に加え、SQL Anywhere では外部キー関係に基づく KEY ジョインがサポートされます。これにより、ジョイン実行時における WHERE 句の複雑さを軽減できます。

Copyright © 2007, iAnywhere Solutions, Inc.

更新

SQL Anywhere では、UPDATE コマンドを使って複数のテーブルを参照できます。複数のテーブルで定義されたビューも更新できます。多くの SQL ソフトウェアでは、ジョインされたテーブルの更新は許可されません。

テーブルの変更

ALTER TABLE コマンドが拡張されました。エンティティ整合性と参照整合性の変更に加え、次に示す変更が行えます。

```
ADD column data-type
ALTER column data-type
DELETE column
RENAME new-table-name
RENAME old-column TO new-column
```

ALTER 句は、文字カラムの最大長の変更とデータ型の変更に使用できます。詳細については、「ALTER TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

式を使用できる場所で実行するサブクエリ

SQL Anywhere では、式が使用できる場所であれば、どこでもサブクエリを使用できます。多くの SQL ソフトウェアでは、サブクエリが使用できるのは比較演算子の右側だけです。たとえば、次に示すコマンドは SQL Anywhere では有効ですが、他の多くの SQL ソフトウェアでは無効です。

```
SELECT Surname,
       BirthDate,
       ( SELECT DepartmentName
         FROM Departments
         WHERE EmployeeID = Employees.EmployeeID
         AND DepartmentID = 200 )
FROM Employees;
```

その他の関数

SQL Anywhere は、ANSI SQL 定義にはない関数をいくつかサポートします。使用できる関数のリストについては、「SQL 関数」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

カーソル

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。

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 システムにはこの機能はありません。

Copyright © 2007, iAnywhere Solutions, Inc.

Transact-SQL との互換性

SQL Anywhere は、Sybase Adaptive Server Enterprise がサポートする SQL のダイアレクトである「**Transact-SQL**」の大規模なサブセットをサポートします。この章では、SQL Anywhere と Adaptive Server Enterprise の間の 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。

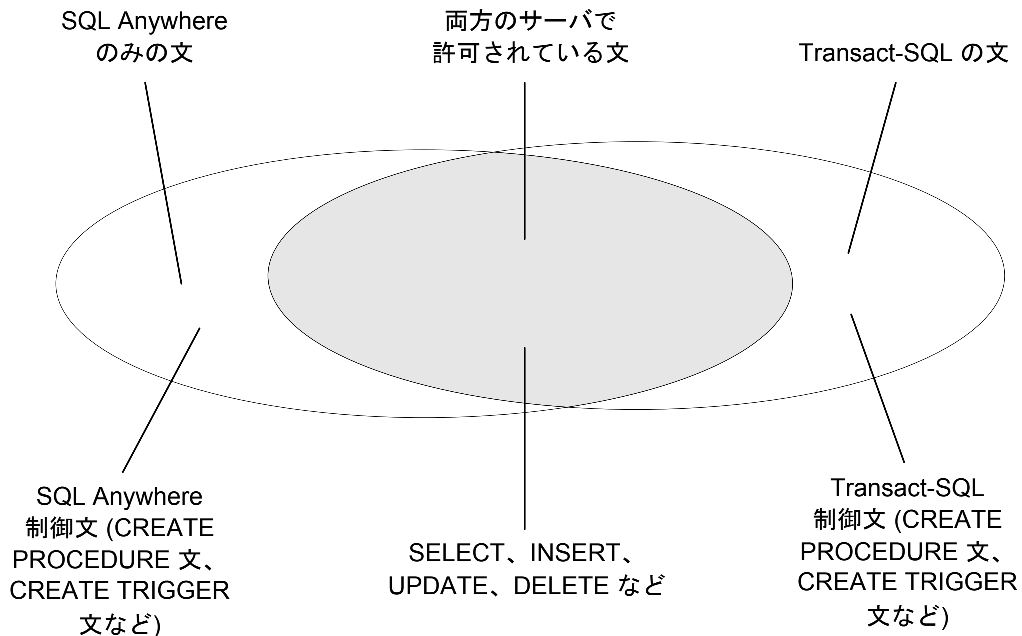
Transact-SQL のサポート方法

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。それらの文の場合には、SQL Anywhere は SQL の 2 種類の「ダイアレクト」をサポートします。この章では、それらのダイアレクトを Transact-SQL と Watcom-SQL と呼びます。
- 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中では、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。

 Copyright © 2007, iAnywhere Solutions, Inc.



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、1 つを除くすべての文字列関数、すべての集合関数、すべての日付と時間関数をサポートします。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) の大部分と、Transact-SQL データのデータ定義言語文のほとんど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。

この章では、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。

- 制御文の CASE、LOOP、FOR
- IF と WHILE の SQL Anywhere バージョン
- CALL 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

注意

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。つまり、ダイアレクトは次の条件で使用できます。

- 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。

 Copyright © 2007, iAnywhere Solutions, Inc.

Adaptive Server 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・プロシージャの実行を提供し、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。

Open Server サポートの詳細については、「Open Server としての SQL Anywhere」『SQL Anywhere サーバ - データベース管理』を参照してください。

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。

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。

ディスク管理の詳細については、「データベース・ファイルの処理」『SQL Anywhere サーバ - データベース管理』を参照してください。

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。

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。

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。

これらの文の SQL Anywhere 構文の詳細については、「SQL 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

この 2 製品のシステム・カタログの説明を含む個別の説明については、「Transact-SQL 互換のビュー」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

Adaptive Server Enterprise では、データベース所有者 (ユーザ ID **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 ID **SYS**) がシステム・テーブルを所有します。ユーザ ID 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。

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、完全なパーミッションを持ちます。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コマンドの実行を行うパーミッション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パーミッションをユーザに与え、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パーミッション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パーミッションを持ちます。

次に、SQL Anywhere で管理上の役割を持つ、データベース全体に適用されるパーミッションを示します。

- データベース管理者 (DBA 権限) 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) に対して完全なパーミッションを持ち、他のユーザにデータベース内でのオブジェクト作成とコマンド実行のパーミッションを付与できます。デフォルトのデータベース管理者は、ユーザ ID **DBA** です。
- RESOURCE パーミッションは、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パーミッション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パーミッション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、適切なパーミッション (SQL Anywhere では RESOURCE、Adaptive Server Enterprise では CREATE 文ごとのパーミッション) 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

Adaptive Server Enterprise と SQL Anywhere では、ユーザとグループのモデルに違いがあります。

Adaptive Server Enterprise では、サーバに接続する各ユーザは、サーバに対するログイン ID とパスワードと、そのサーバ上でアクセスする各データベースに対するユーザ ID が必要です。データベースの各ユーザは、1 つのグループにしか属せません。

SQL Anywhere では、ユーザは直接データベースに接続するため、データベース・サーバに対する別のログイン ID は必要ありません。代わりに、各ユーザはデータベースを使用できるようにそのデータベースに対するユーザ ID とパスワードを付与されます。ユーザは複数のグループに属することができ、グループの階層構造も許可されます。

いずれのサーバもグループをサポートしているため、複数のユーザに一度にパーミッションを付与できます。ただし、グループの詳細に関しては、サーバ間で違いがあります。たとえば、Adaptive Server Enterprise では各ユーザは 1 つのグループにしか入れませんが、SQL Anywhere では制限はありません。特定の情報については、ユーザとグループに関して両者のマニュアルを比較してください。

Adaptive Server Enterprise と SQL Anywhere のどちらにも、デフォルトのパーミッションを定義する public グループがあります。すべてのユーザは、自動的に public グループのメンバになります。

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。

各プロシージャの引数については、「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』を参照してください。

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

システム・プロシージャ	説明
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・パーミッション

個々のデータベース・オブジェクトに対するパーミッションを付与する 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 パーミッションを付与でき、データベースのテーブルの選択したカラムに対する UPDATE パーミッションを付与できます。どちらも、ストアド・プロシージャに対する EXECUTE パーミッションを付与できます。

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 文を使用できるパーミッションを 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、パーミッション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パーミッション

Adaptive Server Enterprise と SQL Anywhere は、データベース全体に適用されるユーザ・パーミッションに異なったモデルを使用します。これらのモデルについては、「ユーザとグループ」 616 ページで説明します。SQL Anywhere は DBA パーミッションを使用して、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、このパーミッション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 権限は、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。

SQL Anywhere は、RESOURCE パーミッションを使用して、データベース内にオブジェクトを作成する権限をユーザに許可します。Adaptive Server Enterprise のパーミッションで非常に近いのは、データベース所有者が使用する GRANT ALL です。

Copyright © 2007, iAnywhere Solutions, Inc.

Transact-SQL との互換性を意識したデータベースの設定

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・オプションで制御できます。

Transact-SQL と互換性のあるデータベースの作成

この項では、データベースの作成や、再構築のときに必要な選択肢について説明します。

クイック・スタート

ここでは、Transact-SQL と互換性のあるデータベースを作成するときに必要な手順を示します。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。

♦ Transact-SQL と互換性のあるデータベースを作成するには、次の手順に従います (Sybase Central の場合)。

1. Sybase Central を起動します。
2. [ツール] – [SQL Anywhere 10] – [データベースの作成] を選択します。
3. ウィザードの指示に従います。
4. [Adaptive Server Enterprise のエミュレート] ボタンが表示されたら、このボタンをクリックし、[次へ] をクリックします。
5. ウィザードの指示に従います。

♦ Transact-SQL と互換性のあるデータベースを作成するには、次の手順に従います (コマンド・ラインの場合)。

- コマンド・プロンプトで次のコマンドを入力します。

```
dbinit -b -c -k db-name.db
```

これらのオプションの詳細については、「初期化ユーティリティ (dbinit)」 『SQL Anywhere サーバ - データベース管理』を参照してください。

♦ Transact-SQL と互換性のあるデータベースを作成するには、次の手順に従います (SQL の場合)。

1. SQL Anywhere データベースに接続します。
2. 次の文を、Interactive SQL などで入力します。

```
CREATE DATABASE 'dbname.db'
ASE COMPATIBLE
CASE RESPECT
BLANK PADDING ON;
```

この文で、ASE COMPATIBLE は Adaptive Server Enterprise と互換性があるという意味です。SYS.SYSCOLUMNS と SYS.SYSINDEXES の各ビューが作成されません。

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。

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。

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みなします。

```
'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みなします。

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、次の文で「テーブル名 'SYSCOLUMNS' はあいまいです。」というエラーが発生します。

```
SELECT * FROM SYSCOLUMNS ;
```


Copyright © 2007, iAnywhere Solutions, Inc.

Transact-SQL との互換性を維持するためのオプション設定

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/2003 ISO 規格と互換性があります。

Adaptive Server Enterprise を SQL/2003 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/2003 ISO 規格に一致する 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/2003 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。

quoted_identifier オプションの詳細については、「quoted_identifier オプション [互換性]」『SQL Anywhere サーバ - データベース管理』を参照してください。

automatic_ timestamp オプションの On への設定

Transact-SQL では、timestamp カラムは特殊なプロパティで定義されています。automatic_timestamp オプションを On に設定した場合の SQL Anywhere での timestamp カラムの処理は、Adaptive Server Enterprise の動作と似ています。

SQL Anywhere で automatic_timestamp オプションが On (デフォルトは Off) に設定されていて、TIMESTAMP データ型を持つ新しいカラムに特にデフォルト値が設定されていないときは、デフォルト値に timestamp が与えられます。

timestamp カラムについては、「Transact-SQL の特殊な timestamp カラムとデータ型」 623 ページを参照してください。

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。

string_rtruncation オプションの詳細については、「string_rtruncation オプション [互換性]」『SQL Anywhere サーバ - データベース管理』を参照してください。

Transact-SQL と互換性のあるデータベース・オプションの詳細については、「互換性オプション」『SQL Anywhere サーバ - データベース管理』を参照してください。

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。データベースで使用する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。

Copyright © 2007, iAnywhere Solutions, Inc.

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 timestamp 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 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 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 ではデータ型は自分で指定します。

automatic_timestamp データベース・オプションが On に設定されている場合、デフォルト値を設定する必要はありません。TIMESTAMP データ型で作成し、明示的なデフォルト値がない新しいカラムには、デフォルト値 timestamp が与えられます。次の文は automatic_timestamp を On に設定します。

```
SET OPTION PUBLIC.automatic_timestamp='On';
```

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。

たとえば、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。

tsequal 関数を使用する UPDATE 文の一般的な例を次に示します。

```
UPDATE publishers
SET City = 'Springfield'
WHERE pub_id = '0736'
AND TSEQUAL(timestamp, '2005/10/25 11:08:34.173226');
```

tsequal 関数の最初の引数は、特殊な timestamp カラムの名前です。2 番目の引数は SELECT 文で取得した timestamp です。Embedded SQL では、2 番目の引数は、カラムで最後に FETCH から取得した TIMESTAMP を含むホスト変数であることが多くなります。

特殊な IDENTITY カラム

IDENTITY カラムを作成するには、次に示すように CREATE TABLE 構文を使います。

```
CREATE TABLE table-name (
  ...
  column-name numeric(n,0) IDENTITY NOT NULL,
  ...
)
```

n はテーブルに挿入されるローの最大数よりも大きい数です。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

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 プロパティはユーザ自身で設定してください。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 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。

@@identity の値は、文がテーブルにローを挿入するたびに変わります。

- 文が IDENTITY カラムのないテーブルに影響を及ぼすと、@@identity は 0 に設定されます。
- 文が複数のローを挿入すると、@@identity は IDENTITY カラムに最後に挿入された値を反映します。

この変更は確定的です。文にエラーが発生したり、文を含むトランザクションがロールバックされたりしても、@@identity は前の値に復元されません。

@@identity の動作の詳細については、「@@identity グローバル変数」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

互換性のある SQL 文の記述方法

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。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しないで、使用可能なオプションをすべて略さずに書きます。
- 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。

互換性のあるテーブルの作成

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 があるローと、NULL 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

 Copyright © 2007, iAnywhere Solutions, Inc.

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。

このオプションについては、「Transact-SQL との互換性を維持するためのオプション設定」 621 ページを参照してください。

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。SQL Anywhere での宣言テンポラリ・テーブルについては、「DECLARE LOCAL TEMPORARY TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

CREATE TABLE 文については、「CREATE TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

互換性のあるクエリの記述方法

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。
- SELECT 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に、SELECT 文の構文の互換性について説明します。quoted_identifier 設定値が Off であり、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。

SQL Anywhere がサポートする、Transact-SQL SELECT 文のサブセットを次に示します。

構文

```
SELECT [ ALL | DISTINCT ] select-list
…[ INTO #temporary-table-name ]
…[ FROM table-spec [ HOLDLOCK | NOHOLDLOCK ],
…  table-spec [ HOLDLOCK | NOHOLDLOCK ], … ]
…[ WHERE search-condition ]
…[ GROUP BY column-name, … ]
…[ HAVING search-condition ]
  [ ORDER BY { expression | integer }
    [ ASC | DESC ], … ]
```

パラメータ

select-list:
 *table-name.**

| *
| *expression*
| *alias-name* = *expression*
| *expression as identifier*
| *expression as string*

table-spec:
[*owner* .]*table-name*
…[[**AS**] *correlation-name*]
…[**(INDEX** *index_name* [**PREFETCH** *size*][**LRU** | **MRU**] **)**]

alias-name:
identifier | '*string*' | "*string*"

SELECT 文の詳細については、「SELECT 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

SQL Anywhere は、次に示す 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
- COMPUTE 句
- FOR BROWSE 句
- FOR UPDATE
- GROUP BY ALL 句

注意

- SQL Anywhere は、グループ作成に使用されないカラムと式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。Adaptive Server Enterprise では、この拡張はサマリレポートを生成します。

  「OLAP のサポート」 413 ページを参照してください。

- 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。このキーワードは、指定したテーブルまたはビューの共有ロックを (トランザクションが完了したかどうかに関わらず、要求されたデータ・ページが必要なくなると同時に共有ロックを解放しないで) トランザクションの完了まで保持することによって、さらに制限できます。HOLDLOCK が指定されているテーブルのために、クエリは独立性レベル 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。

 Copyright © 2007, iAnywhere Solutions, Inc.

```
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 は、次に示す SELECT 構文の句をサポートしません。
  - INTO ホスト変数リスト
  - INTO 変数リスト
  - カッコ付きのクエリ
- Adaptive Server Enterprise は、ジョイン用の FROM 句と ON 条件ではなく、WHERE 句でジョイン演算子を使用します。

ジョインの互換性

Transact-SQL では、次に示す構文を使用して、ジョインを WHERE 句に指定します。

```
start of select, update, insert, delete, or subquery
  FROM { table-list | view-list } WHERE [ NOT ]
  [ table-name.| view name.]column-name
    join-operator
  [ table-name.| view-name.]column_name
  [{ AND | OR } [ NOT ]
  [ table-name.| view-name.]column_name
    join-operator
  [ table-name.| view-name.]column-name ]…
end of select, update, insert, delete, or subquery
```

WHERE 句の *join-operator* は、比較演算子の場合もあれば、次に示す「**外部ジョイン演算子**」のいずれかの場合もあります。

- ***=**　左外部ジョイン演算子
- **=***　右外部ジョイン演算子

SQL Anywhere は、Transact-SQL 外部ジョイン演算子をネイティブの SQL/2003 構文の代わりになるものとしてサポートします。1 つのクエリの中に複数のダイアレクトを混在させることはできません。このルールはクエリによって使用されるビューにも適用されます。ビュー上の外部ジョイン・クエリは、ビュー定義クエリが使用しているダイアレクトに従ってください。

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 と ANSI／ISO SQL 規格におけるジョインの詳細については、「ジョイン：複数テーブルからのデータ検索」 341 ページと「FROM 句」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ジョインの Transact-SQL 互換性の詳細については、「Transact-SQL の外部ジョイン (*= or =*)」 362 ページを参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

Transact-SQL のプロシージャ言語の概要

「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/2003 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・プロシージャの概要

SQL Anywhere ストアド・プロシージャ言語は、ISO/ANSI 規格を基準にしており、Transact-SQL ダイアレクトとは多くの点で異なっています。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 のトリガの概要

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 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。すなわち、**文レベル**の**完了後**トリガです。SQL Anywhere は、ローが変更される前または後に実行される**ロー・レベル**・トリガと、文全体の後に実行される文レベル・トリガの両方をサポートします。

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。詳細については、「プロシージャ、トリガ、バッチの使用」 777 ページで説明します。

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。「SET 文 [T-SQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

  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。

- **トリガ中の ROLLBACK 文** 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。トリガす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。「ROLLBACK TRIGGER 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Transact-SQL のバッチの概要

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。バッチはコマンド・ファイルとして保存されます。SQL Anywhere の Interactive SQL と Adaptive Server Enterprise の Interactive SQL ユーティリティは、バッチを対話型で実行するためのよく似た機能を持っています。

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。

コマンド・ファイルに保存されているバッチでは、SQL Anywhere はコマンド・ファイルにあるパラメータの使用をサポートします。Adaptive Server Enterprise は、パラメータをサポートしません。

 Copyright © 2007, iAnywhere Solutions, Inc.

パラメータについては、「PARAMETERS 文 [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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のほかに、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。
- **WatcomSQL(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')
```

Sybase Central を使用するストアド・プロシージャの変換

Sybase Central は、プロシージャとトリガの作成、表示、変更を行うための機能を備えています。

- **Sybase Central を使ってストアド・プロシージャを変換するには、次の手順に従います。**

1. 変更するプロシージャの所有者または DBA ユーザとして、Sybase Central から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] フォルダを開きます。
3. 右ウィンドウ枠の [SQL] タブをクリックし、エディタ内をクリックします。
4. [ファイル] メニューから、使用するダイアレクトによって [Watcom-SQL に変換] コマンドまたは [Transact-SQL に変換] コマンドを選択します。

   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、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. 必要に応じて変換されなかった行を書き直します。
6. 終了したら、[ファイル] – [保存] を選択し、変換されたバージョンをデータベースに保存します。そのテキストをファイルにエクスポートして Sybase Central の外部で編集することもできます。

 Copyright © 2007, iAnywhere Solutions, Inc.

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
```

プロシージャと結果の詳細については、「プロシージャから返される結果」 808 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

Transact-SQL プロシージャの中の変数

SQL Anywhere は SET 文を使用して、プロシージャ中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。

SELECT 文の使用による変数割り当ての詳細については、「互換性のあるクエリの記述方法」 627 ページを参照してください。SET 文の使用による変数割り当ての詳細については、「SET 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

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	意味
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
−2	データ型エラー
−3	プロセスはデッドロックの対象となった
−4	パーミッション・エラー
−5	構文エラー
−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
−8	致命的でない内部の問題
−9	システムの限界に達した
−10	致命的な内部の矛盾

Copyright © 2007, iAnywhere Solutions, Inc.

値	意味
−11	致命的な内部の矛盾
−12	テーブルまたはインデックスが壊れている
−13	データベースが壊れている
−14	ハードウェア・エラー

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。

プロシージャの中での RAISERROR 文の使用

RAISERROR 文は、ユーザ定義エラーを生成するための Transact-SQL 文です。これは、SIGNAL 文と似た機能を持っています。

RAISERROR 文の詳細については、「RAISERROR 文 [T-SQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

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。

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、これらの 2 つの句は一緒に使用しないようにしてください。

 Copyright © 2007, iAnywhere Solutions, Inc.

パート V. データベースにおける XML

パート V では、データベースで XML を使用する方法について説明します。

第 16 章

データベースにおける XML の使用

目次

XML の概要 642
リレーショナル・データベースにおける XML 文書の格納 643
リレーショナル・データを XML としてエクスポートする 644
XML 文書をリレーショナル・データとしてインポートする 645
クエリ結果を XML として取得する 652
SQL/XML を使用してクエリ結果を XML として取得する 670

XML の概要

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。

XML の詳細については、http://www.w3.org/XML/ を参照してください。

XML と SQL Anywhere

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

Copyright © 2007, iAnywhere Solutions, Inc.

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、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ない点に注意してください。

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る点に注意してください。

要素の内容がエスケープされる方法の詳細については、「不正な XML 名のエンコーディング」 654 ページを参照してください。

XML データ型の詳細については、「XML データ型」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

リレーショナル・データを XML としてエクスポートする

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 からリレーショナル・データを XML としてエクスポートする

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。

OUTPUT 文を使用した XML のエクスポートの詳細については、「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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 プロシージャまたは 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。

XML のインポートの詳細については、「XML 文書をリレーショナル・データとしてインポートする」 645 ページを参照してください。

DataSet オブジェクト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

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 ADO.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 オブジェクトを使用したローの挿入、更新、削除」『SQL Anywhere サーバ - プログラミング』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

XML 文書をリレーショナル・データとしてインポートする

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 プロシージャを使用して、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 を使用した XML のインポート

クエリの FROM 句で openxml プロシージャを使用すると、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。
- **. (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。

Copyright © 2007, iAnywhere Solutions, Inc.

.

<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
```

openxml がサポートする XPath 構文の完全なリストについては、「openxml 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。

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

詳細については、「openxml 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。

openxml を使用したエッジ・テーブルの生成

openxml プロシージャを使用すると、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。

Copyright © 2007, iAnywhere Solutions, Inc.

次の SQL 文は、XML 文書を含む変数 x 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 修飾子を指定した FOR XML AUTO を使用して、Employees テーブル、SalesOrders テーブル、Customers テーブルの各カラムに対応する要素が生成されています。

XMLELEMENT 関数の詳細については、「XMLELEMENT 関数 [文字列]」 『SQL Anywhere サーバ - SQL リファレンス』 を参照してください。

FOR XML AUTO の詳細については、「FOR XML AUTO の使用」 658 ページを参照してください。

```
CREATE VARIABLE x XML;
SET x=(SELECT XMLELEMENT( NAME root,
        (SELECT * FROM Employees
        KEY JOIN SalesOrders
        KEY JOIN Customers
        FOR XML AUTO, ELEMENTS)));
SELECT x;
```

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>299</EmployeeID>
  <ManagerID>902</ManagerID>
  <Surname>Overbey</Surname>
  <GivenName>Rollin</GivenName>
  <DepartmentID>200</DepartmentID>
  <Street>191 Companion Ct.</Street>

  <City>Kanata</City>
  <State>CA</State>
  <Country>USA</Country>
  <PostalCode>94608</PostalCode>
  <Phone>5105557255</Phone>
  <Status>A</Status>
  <SocialSecurityNumber>025487133</SocialSecurityNumber>
  <Salary>39300.000</Salary>
  <StartDate>1987-02-19</StartDate>
  <BirthDate>1964-03-15</BirthDate>
  <BenefitHealthInsurance>Y</BenefitHealthInsurance>
  <BenefitLifeInsurance>Y</BenefitLifeInsurance>
  <BenefitDayCare>N</BenefitDayCare>
  <Sex>M</Sex>

  <SalesOrders>
  <ID>2001</ID>
  <CustomerID>101</CustomerID>
  <OrderDate>2000-03-16</OrderDate>
  <FinancialCode>r1</FinancialCode>
  <Region>Eastern</Region>
  <SalesRepresentative>299</SalesRepresentative>

    <Customers>
    <ID>101</ID>
    <Surname>Devlin</Surname>
    <GivenName>Michael</GivenName>
    <Street>114 Pioneer Avenue</Street>
    <City>Kingston</City>
```

Copyright © 2007, iAnywhere Solutions, Inc.

```
      <State>NJ</State>
      <PostalCode>07070</PostalCode>
      <Phone>2015558966</Phone>
      <CompanyName>The Power Group</CompanyName>
      </Customers>
    </SalesOrders>
  </Employees>
  ...
```

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。メタプロパティ名は大文字と小文字を区別します。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。

```
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	299
55	16	ManagerID	902
79	16	Surname	Overbey
...	...	...	...

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。ファイル *c:¥inventory.xml* の内容が次のようになっているとします。

```
<inventory>
  <product>Tee Shirt
    <quantity>54</quantity>
    <color>Orange</color>
  </product>
  <product>Baseball Cap
    <quantity>112</quantity>
    <color>Black</color>
  </product>
</inventory>
```


Copyright © 2007, iAnywhere Solutions, Inc.

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。

```
CREATE VARIABLE x XML;
SELECT xp_read_file( 'c:¥¥inventory.xml' )
 INTO x;
SELECT * FROM openxml( x, '//*' )
 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、ラテラル抽出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 test (ManagerID INT, Reports XML);
INSERT INTO test
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 test
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>

Copyright © 2007, iAnywhere Solutions, Inc.

ManagerID	Reports
1293	<reports> <e>148</e> <e>390</e> <e>586</e> <e>757</e> ... </reports>
...	...

次のクエリは、ラテラル抽出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 test, LATERAL( openxml( test.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
...	...

ラテラル抽出テーブルの詳細については、「FROM 句」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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 ADO.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。

 Copyright © 2007, iAnywhere Solutions, Inc.

SQL Anywhere .NET データ・プロバイダを使用して、DataSet を元に XML 文書からデータかスキーマまたはその両方を読み込む方法については、「SADataAdapter オブジェクトを使用したデータの取得」『SQL Anywhere サーバ - プログラミング』を参照してください。

クエリ結果を XML として取得する

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。

  FOR XML 句の使用については、「FOR XML 句を使用してクエリ結果を XML として取り出す」 653 ページと「SELECT 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **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 規格に基づく関数をサポートしています。

  クエリ内でこれらの関数を 1 つ以上使用する方法については、「SQL/XML を使用してクエリ結果を XML として取得する」 670 ページを参照してください。

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 とおり提供されます。多くの場合、どちらを使用しても同じ XML が生成されます。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

FOR XML 句を使用してクエリ結果を XML として取り出す

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。

XML データ型については、「XML データ型」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

FOR XML 句の使用方法の例については、「FOR XML の例」 655 ページを参照してください。

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。

  詳細については、「FOR XML RAW の使用」 656 ページを参照してください。

- **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*内のテーブルの順序に基づきます。

  詳細については、「FOR XML AUTO の使用」 658 ページを参照してください。

- **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 のフォームを制御できます。

  詳細については、「FOR XML EXPLICIT の使用」 660 ページを参照してください。

以降の項では、FOR XML 句の 3 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 句の使用方法の例を示します。

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、自動的に復号化します。

openxml の詳細については、「openxml 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。

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。

for_xml_null_treatment オプションについては、「for_xml_null_treatment オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

不正な XML 名のエンコーディング

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ユニコード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ユニコード文字の UTF-16 値は、16 ビット・ワード 1 つです。

Copyright © 2007, iAnywhere Solutions, Inc.

- 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ユニコード文字の 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、ユニコード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。

FOR XML 句の構文については、「SELECT 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

> **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。
> トランケーション長の設定については、「truncation_length オプション [Interactive SQL]」『SQL Anywhere サーバ - データベース管理』と「[オプション] ダイアログ : [結果] タブ」『SQL Anywhere 10 - コンテキスト別ヘルプ』を参照してください。

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
```

Copyright © 2007, iAnywhere Solutions, Inc.

```
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 の使用

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。

Copyright © 2007, iAnywhere Solutions, Inc.

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 句を含むクエリで NULL 値が返される方法については、「FOR XML と NULL 値」 654 ページを参照してください。

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))
```

XMLELEMENT 関数の詳細については、「XMLELEMENT 関数 [文字列]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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"/>
...
```

Copyright © 2007, iAnywhere Solutions, Inc.

FOR XML AUTO の使用

AUTO モードは、XML 文書内にネストされた要素を生成します。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 リスト内でテーブルが参照される順序に基づきます。AUTO モードを指定すると、select リストの各テーブルに対して要素が作成され、そのテーブル内の各カラムは別個の属性となります。

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。

for_xml_null_treatment オプションについては、「for_xml_null_treatment オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

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てください。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、希望する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 ) );
```

XMLELEMENT 関数の詳細については、「XMLELEMENT 関数 [文字列]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

```
<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 name="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。

FOR XML EXPLICIT の使用

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、希望する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。

FOR XML EXPLICIT を使用してクエリを記述する方法の例は、「EXPLICIT モードのクエリの記述」 662 ページを参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

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(**GivenName!1** と **ID!2** データ・カラムの目的は、次の「クエリへのデータ・カラムの追加」 661 ページで説明します)。

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。これは、両要素とも階層のトップ・レベルに生成されることを意味し、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>
```

FOR XML EXPLICIT を使用してクエリを記述する方法の例は、「EXPLICIT モードのクエリの記述」 662 ページを参照してください。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

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**　このカラムが、結果を生成する目的で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。

  hide ディレクティブの使用例については、「hide ディレクティブの使用」 667 ページを参照してください。

- **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。

  element ディレクティブの使用例については、「element ディレクティブの使用」 666 ページを参照してください。

- **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。

  xml ディレクティブの使用例については、「xml ディレクティブの使用」 668 ページを参照してください。

- **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。

  cdata ディレクティブの使用例については、「cdata ディレクティブの使用」 669 ページを参照してください。

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。

for_xml_null_treatment オプションの詳細については、「for_xml_null_treatment オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』と「FOR XML と NULL 値」 654 ページを参照してください。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

```
<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。

♦ FOR XML EXPLICIT クエリを記述するには、次の手順に従います。

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。

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;
```

2. 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
```

3. クエリに UNION ALL 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 ALL

SELECT
    2,
    1,
    EmployeeID,
    CustomerID,
    Region
FROM Employees KEY JOIN SalesOrders
```

4. 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,
```

 Copyright © 2007, iAnywhere Solutions, Inc.

```
      EmployeeID AS [employee!1!EmployeeID],
      NULL      AS [customer!2!CustomerID],
      NULL      AS [customer!2!Region]
   FROM Employees
   UNION ALL

   SELECT
      2,
      1,
      EmployeeID,
      CustomerID,
      Region
   FROM Employees KEY JOIN SalesOrders
   ORDER BY 3, 1
   FOR XML EXPLICIT;
```

EXPLICIT モードの構文については、「パラメータ」 661 ページを参照してください。

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 ALL

SELECT
   2,
   1,
   EmployeeID,
   NULL,
   OrderDate,
   NULL
FROM Employees KEY JOIN SalesOrders
UNION ALL

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 ALL
SELECT
     2,
     1,
     EmployeeID,
     NULL,
     OrderDate,
     NULL
FROM Employees KEY JOIN SalesOrders
UNION ALL

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>
```


Copyright © 2007, iAnywhere Solutions, Inc.

```
<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 ALL

SELECT
        2,
        1,
        EmployeeID,
        NULL,
        OrderDate,
        NULL
FROM Employees KEY JOIN SalesOrders
UNION ALL

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;
```

Copyright © 2007, iAnywhere Solutions, Inc.

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」 654 ページを参照)。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!CompanyName]
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。

```
<Customers CompanyName="Sterling &amp; Co.">
  <ID>115</ID>
</Customers>
```

xml ディレクティブは、生成される XML にカラム値が引用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!CompanyName!xml]
FROM Customers
WHERE ID = '115'
FOR XML EXPLICIT;
```

結果内で、アンパサンドは引用されていません。

Copyright © 2007, iAnywhere Solutions, Inc.

```
<customer>
 <ID>115</ID>
 <CompanyName>Sterling & Co.</CompanyName>
</customer>
```

この XML は、アンパサンドが含まれるため、整形式ではない点に注意してください。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* フィールドは無視され、属性ではなく要素が生成されます。

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>
...
```

SQL/XML を使用してクエリ結果を XML として取得する

SQL/XML は、XML を SQL 言語に機能統合する方法を定める、ドラフト段階の規格です。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。

無効な式のマークアップの詳細については、「不正な XML 名のエンコーディング」 654 ページを参照してください。

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」 643 ページを参照してください。

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>

 Copyright © 2007, iAnywhere Solutions, Inc.

department_list
<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>
...

XMLAGG 関数の詳細については、「XMLAGG 関数 [集合]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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>
<given_name>Julie</given_name> <surname>Jordan</surname>
...

詳細については、「XMLCONCAT 関数 [文字列]」 『SQL Anywhere サーバ - SQL リファレンス』 を参照してください。

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>`
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。

Copyright © 2007, iAnywhere Solutions, Inc.

```
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 ( UnitPrice AS
                                                                      price ),
                       Products.name
                      ) AS products
FROM Products
WHERE ID > 400;
```

詳細については、「XMLELEMENT 関数 [文字列]」 『SQL Anywhere サーバ - SQL リファレンス』 を参照してください。

例

次の例では HTTP Web サービスで XMLELEMENT を使用します。

```
ALTER 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 tabl XML;
  SET var  = NULL;
loop_h:
  LOOP
    SET var = NEXT_HTTP_HEADER( var );
    IF var IS NULL THEN LEAVE leave loop_h END IF;
    SET varval = http_header( var );
    -- ... do some action for <var,varval> pair...
      SET tabl = tabl ||
                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' ),

             tabl ) ;
    SELECT res;
END
```

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。

<table>
<tr><th>ID</th><th>product_info</th></tr>
<tr><td>401</td><td><pre><item_description>
 <name>Baseball Cap</name>
 <color>White</color>
 <price>10.00</price>
</item_description></pre></td></tr>
<tr><td>500</td><td><pre><item_description>
 <name>Visor</name>
 <color>White</color>
 <price>7.00</price>
</item_description></pre></td></tr>
<tr><td>501</td><td><pre><item_description>
 <name>Visor</name>
 <color>Black</color>
 <price>7.00</price>
</item_description></pre></td></tr>
<tr><td>...</td><td>...</td></tr>
</table>

詳細については、「XMLFOREST 関数 [文字列]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

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ている点に注意してください。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 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」 670 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 );
```

 Copyright © 2007, iAnywhere Solutions, Inc.

このクエリは、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>
```

XMLGEN 関数の詳細については、「XMLGEN 関数 [文字列]」 『SQL Anywhere サーバ - SQL リファレンス』 を参照してください。

パート VI. リモート・データとバルク・オペレーション

パート VI では、データベースのロードとアンロードの方法、およびリモートデータへのアクセス方法について説明します。

第 17 章

データのインポートとエクスポート

目次

データベースとの間でのデータの転送 682
データのインポート 684
データベースからのデータのエクスポート 694
データベースの再構築 706
データベースの抽出 715
SQL Anywhere へのデータベースの移行 716
SQL コマンド・ファイルの使用 720
Adaptive Server Enterprise の互換性 723

データベースとの間でのデータの転送

さまざまな状況下で、データをデータベースから出し入れすることが必要になる場合があります。次に例を示します。

- 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。

多くの場合、これらのタスクは、バルク・オペレーションとしてグループ化されます。これらは、DBA 権限を持つユーザによって実行されます。通常のエンドユーザ・アプリケーションの一部ではありません。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・ログに特定の要求を課す場合があり、通常は他の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のが最適です。

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。

データベース・サーバの内部：サーバ側

LOAD TABLE、UNLOAD TABLE、UNLOAD の各文はデータベース・サーバで実行されます。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になります。LOAD TABLE 文は、単一のコマンドとしてトランザクション・ログに記録されます。内部でインポートとエクスポートを行うと、テキスト形式と BCP フォーマットにだけアクセス可能ですが、この方法の方が速く処理できます。

データベース・サーバの外部：クライアント側

INPUT と OUTPUT の各コマンドは Interactive SQL で実行されます。読み込みまたは書き込み対象のファイルへのパスは、サーバではなく Interactive SQL とそれを実行しているコンピュータへの相対パスになります。INPUT は、読み込む各ローへの個別の INSERT 文としてトランザクション・ログに記録されます。このため、INPUT は LOAD TABLE よりかなり遅くなります。また、INSERT トリガが INPUT 中に起動することになります。OUTPUT 文は、SELECT 文の結果セットを多数あるファイル・フォーマットのどれか 1 つで書き出すことができるので、互換性が重要な場合に役立ちます。

Copyright © 2007, iAnywhere Solutions, Inc.

詳細については、「データのインポートのパフォーマンスのヒント」 684 ページを参照してください。

バルク・オペレーションのデータ・リカバリの問題

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。

バルク・オペレーション・モード (-b オプション) でデータベース・サーバを実行している場合、データベース・サーバは特定の重要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。

Copyright © 2007, iAnywhere Solutions, Inc.

データのインポート

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 では、次の作業が可能です。

- ASCII 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。
- DBASE や EXCEL などの他のデータベース・フォーマット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。
- スクリプトでインポート手順を自動化して、連続した複数のテーブルをインポートする。
- 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。
- 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。

データベース全体のアンロードまたは再ロードの詳細については、「データベースの再構築」 706 ページを参照してください。

インポートに関する考慮事項

データをデータベースにインポートする前に、現在あるリソースは何であるか、またデータベースへのデータのインポートによって厳密に何を行うのかを詳細に検討してください。

検討事項	参照先
使用可能なインポート・ツール	「インポート・ツール」 685 ページ
インポートがデータベース・パフォーマンスに影響を与える可能性	「バルク・オペレーションのパフォーマンスの側面」 682 ページ 「データのインポートのパフォーマンスのヒント」 684 ページ
データとインポート先テーブルとの互換性の問題	「インポート用のテーブル構造」 691 ページ

データのインポートのパフォーマンスのヒント

データを大量にインポートするには時間がかかることがありますが、時間を節約するための方法がいくつかあります。

Copyright © 2007, iAnywhere Solutions, Inc.

- ◆ 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。
- ◆ 「ALTER DBSPACE 文」『SQL Anywhere サーバ - SQL リファレンス』に説明されているように、データベースのサイズを拡張する。このコマンドを使うと、領域が必要になったときに小さいサイズで拡張する代わりに、領域が必要になる前にデータベースを大幅に拡張できます。大量データをロードするときのパフォーマンスの向上だけでなく、ファイル・システム内で DB 領域の断片化を防ぐことにもなります。
- ◆ 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 文を使用する場合、-b オプション (バルク・オペレーション・モード) を指定してサーバを起動する必要はないため、このオプションを指定しない。
- ◆ INPUT 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。

インポート・ツール

データを SQL Anywhere データベースにインポートするのに役立つさまざまなツールがあります。これらの各インポート・ツールについて次に説明します。

Interactive SQL の [インポート] ウィザードの使用

Interactive SQL の [インポート] ウィザードでは、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できます。データは、テキスト、DBASEII、DBASEIII、Excel 2.1、FOXPRO、Lotus フォーマットでインポートできます。このデータを既存のテーブルにインポートすることを選択できます。または、ウィザードを使用してまったく新しいテーブルを作成し、設定できます。

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グラフィカル・インタフェースを使用する場合。

例

この例では、Interactive SQL の [データ] メニューを使用してデータをインポートする方法を示します。

♦ データをインポートするには、次の手順に従います (Interactive SQL の [データ] メニューの場合)。

1. [データ] メニューから、[インポート] を選択します。

   [開く] ダイアログが表示されます。

2. インポートするファイルを検索し、[開く] をクリックします。

   [インポート] ウィザードが表示されます。

3. インポートするファイルにデータベースの値をどのように格納するかを指定します。

4. [既存のテーブルを使用します] オプションを選択し、既存のテーブルの名前とロケーションを入力します。[次へ] をクリックします。

   [参照] をクリックすると、データをインポートするテーブルも検索できます。

5. ASCII ファイルの場合は、次の情報を指定できます。

   ♦ ファイルのフィールド・デリミタ
   ♦ エスケープ文字
   ♦ 後続ブランクを含めるかどうか
   ♦ ファイルのエンコード

   エンコード・オプションを使用することで、ファイルの読み込みに使用するコード・ページを指定して文字を確実に正しく処理するようにできます。(デフォルト) を選択した場合、Interactive SQL を実行しているシステム用のデフォルト・エンコードが使用されます。

   詳細については、「推奨文字セットと照合」『SQL Anywhere サーバ - データベース管理』を参照してください。

6. ウィザードの指示に従います。

   インポートを行うと、新しいデータが既存のテーブルに追加されます。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ことを示すメッセージが表示されます。

次の例は、SQL Anywhere サンプル・データベースの Employees テーブルにデータを追加する方法を示しています。この例は、Julia Chan という名前の従業員に関する情報を追加します。

♦ データを SQL Anywhere サンプル・データベースにインポートするには、次の手順に従います。

1. 次の値から成るテキスト・ファイルを作成し (値は 1 行で入力します)、*import.txt* という名前で保存します。

   ```
   100,500,'Chan','Julia',100,'300 Royal Drive',
   'Springfield','OR','USA','97015','6175553985',
   'A','017239033',55700,'1984-09-29',,'1968-05-05',
   1,1,0,'F'
   ```

2. Interactive SQL の [データ] メニューから [インポート] を選択します。

 Copyright © 2007, iAnywhere Solutions, Inc.

3. *import.txt* ファイルを検索し、[開く] をクリックします。

   [インポート] ウィザードが表示されます。

4. [カンマなどのデリミタによって区切る] オプションを選択します。

5. [既存のテーブルを使用します] オプションを選択し、インポート先テーブルの名前として「**Employees**」と入力します。[次へ] をクリックします。

6. [インポート] ウィザードの残りの手順に従います。このときデフォルト値を使用します。

INPUT 文を使用したデータのインポート

INPUT 文を使用すると、1 つまたは複数のテーブルにさまざまなファイル・フォーマットでデータをインポートできます。デフォルトの入力フォーマットを選択したり、INPUT 文ごとにファイル・フォーマットを指定したりできます。Interactive SQL では、複数の INPUT 文が含まれたコマンド・ファイルを実行できます。INPUT 文は Interactive SQL コマンドなので、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。

次の場合は、INPUT 文を使用してデータをインポートします。

- 1 つまたは複数のテーブルにデータをインポートする場合。
- コマンド・ファイルを使用してインポート処理を自動化する場合。
- テキスト以外のフォーマットのデータをインポートする場合。

詳細については、「INPUT 文 [Interactive SQL]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

INPUT 文は LOAD TABLE 文よりも低速です。

データ・ファイルのフォーマットが DBASE、DBASEII、DBASEIII、FOXPRO、LOTUS のいずれかで、テーブルが存在しない場合は、テーブルが作成されます。

例

この例では、Interactive SQL の INPUT 文を使用してデータをインポートする方法を示します。

♦ **データをインポートするには、次の手順に従います (Interactive SQL の INPUT 文の場合)。**

1. 次の値から成るテキスト・ファイルを作成し (値は 1 行で入力します)、*new_employees.txt* という名前で保存します。

```
101,500,'Chan','Julia',100,'300 Royal Drive',
'Springfield','OR','USA','97015','6175553985',
```

```
'A','017239033',55700,'1984-09-29',,'1968-05-05',
1,1,0,'F'
```

2. インポート先テーブルが存在することを確認します。

3. Interactive SQL の [SQL 文] ウィンドウ枠に INPUT 文を入力します。

   次は、ASCII テキスト・ファイルからの INPUT 文の例です。

   ```
   INPUT INTO Employees
   FROM c:¥new_employees.txt
   FORMAT ASCII;
   SELECT * FROM Employees;
   ```

   この文では、Employees はインポート先テーブルの名前で、*new_employees.txt* はソース・ファイルの名前です。

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ことを示すメッセージが表示されます。

INPUT 文を使用したデータのインポートについては、「INPUT 文 [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

LOAD TABLE 文を使用したデータのインポート

LOAD TABLE 文を使用すると、データをテキストや ASCII フォーマットで効率的にテーブルにインポートできます。LOAD TABLE 文は、1 行に 1 ローずつ、値をデリミタで区切ってインポートします。

次の場合は、LOAD TABLE 文を使用します。

- テキスト・フォーマットでデータをインポートする場合。
- パフォーマンスを改善する必要があり、INPUT 文ではなく LOAD TABLE 文を使用できる場合。

詳細については、「LOAD TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

データベースに対する影響

LOAD TABLE 文を使用した場合、変更内容はトランザクション・ログには記録されず、SQL Remote や Mobile Link リモート・データベースで LOAD TABLE 文を使用できなくなります。現在のトランザクション・ログとデータベースのコピーを使用してデータをリカバリする必要がある場合、元のデータ・ファイルが存在しないとリカバリは失敗します。

LOAD TABLE 文は、INPUT 文よりもかなり高速です。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

INSERT 文を使用したデータのインポート

INSERT 文を使用して、データベースにローを追加できます。インポート先テーブルにインポートするデータを直接 INSERT 文に含めるので、この文は対話型入力とみなされます。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。

詳細については、「INSERT 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。このため、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。

詳細については、「トランザクション・ログ」 『SQL Anywhere サーバ - データベース管理』を参照してください。

例

この例では、INSERT 文を使用してデータをインポートする方法を示します。この INSERT 文では、SQL Anywhere サンプル・データベースの Departments テーブルにデータを追加します。

♦ **データをインポートするには、次の手順に従います (INSERT 文の場合)。**

1. インポート先テーブルが存在することを確認します。
2. INSERT 文を実行します。次に例を示します。

   次の例では、SQL Anywhere サンプル・データベースの Departments テーブルに新しいローを挿入します。

   ```
   INSERT
   INTO Departments ( DepartmentID, DepartmentName, DepartmentHeadID )
   VALUES ( 700, 'Training', 501);
   SELECT * FROM Departments;
   ```

   値を挿入すると、新しいデータが既存のテーブルに追加されます。

プロキシ・テーブルを使用したデータのインポート

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。

Copyright © 2007, iAnywhere Solutions, Inc.

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。このため、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。

リモート・データ・アクセスの詳細については、「リモート・データへのアクセス」 725 ページを参照してください。

INSERT 文の詳細については、「INSERT 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

インポート中の変換エラーの処理

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、カラムのデータ型に合わない日付や数値が含まれていることがあります。conversion_error データベース・オプションを使えば、無効な値を NULL 値に変換し、変換エラーを無視できます。

データベース・オプションの設定の詳細については、「SET OPTION 文」 『SQL Anywhere サーバ - SQL リファレンス』と「conversion_error オプション [互換性]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

テーブルのインポート

Interactive SQL の [データ] メニューまたは [SQL 文] ウィンドウ枠を使用して、既存のデータベースにテーブルをインポートできます。テーブル・データには、テキスト、DBASEII、Excel 2.1、FOXPRO、Lotus フォーマットを使用できます。

♦ テーブルをインポートするには、次の手順に従います (Interactive SQL の [データ] メニューの場合)。

1. データを入れるテーブルが存在することを確認します。
2. [データ] メニューから、[インポート] を選択します。

   [開く] ダイアログが表示されます。
3. インポートするファイルを検索し、[開く] をクリックします。ファイル拡張子には、[インポート] ウィザードで認識される拡張子を指定します。

 Copyright © 2007, iAnywhere Solutions, Inc.

[インポート] ウィザードが表示されます。

4. [既存のテーブルを使用します] オプションを選択し、フィールドにテーブルの名前を入力します。

5. ウィザードの残りの指示に従います。

   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と挿入されたローの数が表示されます。インポートに失敗した場合は、インポートに失敗したことを示すメッセージが表示されます。[結果] ウィンドウ枠の [結果] タブにテーブルに内容が表示されます。

♦ テーブルをインポートするには、次の手順に従います (Interactive SQL の [SQL 文] ウィンドウ枠の場合)。

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 GROUPO.Departments
(
DepartmentID        integer NOT NULL,
DepartmentName       char(40) NOT NULL,
DepartmentHeadID      integer NULL,
CONSTRAINT DepartmentsKey PRIMARY KEY (DepartmentID)
);
```

   詳細については、「CREATE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

2. LOAD TABLE 文を実行します。次に例を示します。

```
LOAD TABLE Departments
FROM '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 ON) では、データの挿入前に、値の後続ブランクを取り除きます。

   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。

   TRUNCATE TABLE 文と LOAD TABLE 文は、カスケード型削除のような参照整合性に関わるアクションを含め、トリガを起動することはありません。

LOAD TABLE 文の構文の詳細については、「LOAD TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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 入力不可能なカラムがカラム・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、Employees テーブルの City カラムのデフォルト値を定義してから、次のような LOAD TABLE 文を使用して新しいローを Employees テーブルにロードできます。

```
ALTER TABLE Employees
ALTER City DEFAULT 'Waterloo';
LOAD TABLE Employees (Surname, GivenName, EmployeeID, DepartmentID, StartDate)
FROM 'new_employees.txt'
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ることができます。

◆ グローバル・テンポラリ・テーブルを使用して構造が異なるデータをロードするには、次の手順に従います。

1. Interactive SQL ウィンドウの [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。

Copyright © 2007, iAnywhere Solutions, Inc.

詳細については、「CREATE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

バイナリ・ファイルのインポート

xp_read_file システム・プロシージャを使用して、JPEG、ビットマップ、Microsoft Word ファイルなどのバイナリ・ファイルをデータベースにインポートできます。「ドキュメントとイメージの挿入」502 ページを参照してください。

データベースからのデータのエクスポート

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。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なツールです。SQL Anywhere では、次の作業が可能です。

- 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。
- 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。

データベース全体のアンロードまたは再ロードの詳細については、「データベースの再構築」 706 ページを参照してください。

エクスポートの考慮事項

データのエクスポートを開始する前に、現在あるリソースは何であるか、またデータベースから厳密にどのようなタイプの情報をエクスポートするのかを詳細に検討してください。

検討事項	参照先
使用可能なエクスポート・ツール	「エクスポート・ツール」 694 ページ
エクスポートがデータベースのパフォーマンスに与える影響	「バルク・オペレーションのパフォーマンスの側面」 682 ページ
NULL 値の処理方法	「OUTPUT 文を使用した NULL 値の出力」 702 ページ

エクスポート・ツール

次のようなさまざまなツールを使用して、データをエクスポートできます。

Interactive SQL の [エクスポート] ウィザードの使用

Interactive SQL の [エクスポート] ウィザードを使用して、クエリの結果をエクスポートできます。

 Copyright © 2007, iAnywhere Solutions, Inc.

このエクスポート・ツールは、結果セットをファイルに保存する場合に使用します。Interactive SQL の [データ] メニューから [エクスポート] を選択します。このとき、エクスポートするクエリ結果のファイル・フォーマットを選択できます。

例

次の例は、SQL クエリからファイルに結果セットをエクスポートする方法を示しています。

♦ Interactive SQL を使用して結果セット・データをエクスポートするには、次の手順に従います。

1. SQL Anywhere のサンプル・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。

   ```
   SELECT * FROM Employees
   WHERE State = 'GA';
   ```

   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。

2. Interactive SQL の [データ] メニューから [エクスポート] を選択します。

   [エクスポート] ダイアログが表示されます。

3. エクスポートするデータの名前とロケーションを指定します。

4. ファイル・フォーマットを指定して [OK] をクリックします。

   結果セットが、指定した場所にあるファイルにエクスポートされます。

OUTPUT 文を使用したデータのエクスポート

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したりできます。Interactive SQL では、複数の OUTPUT 文が含まれたコマンド・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、[オプション] ダイアログ ([ツール] − [オプション] を選択) 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。
- ♦ コマンド・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。

詳細については、「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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、*Employees.txt* という名前の *.txt* ファイルにエクスポートします。

```
SELECT *
FROM Employees;
OUTPUT TO Employees.txt
FORMAT ASCII;
```

UNLOAD TABLE 文の使用

UNLOAD TABLE 文を使用すると、テキストまたは ASCII 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。データは、プライマリ・キー値の順にエクスポートされるため、再ロードは速くなります。

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。

データベースに対する影響

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。

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。

詳細については、「UNLOAD TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

例

SQL Anywhere のサンプル・データベースを使用し、次のコマンドを実行して Employees テーブルをテキスト・ファイル *employee_data.csv* にアンロードできます。

```
UNLOAD TABLE Employees TO 'employee_data.csv';
```

データベース・サーバでテーブルをアンロードするため、*employee_data.csv* にはデータベース・サーバ・コンピュータ上のファイルを指定します。

Copyright © 2007, iAnywhere Solutions, Inc.

UNLOAD 文の使用

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 文を使用すると、より効率的にテキストまたは ASCII フォーマットだけでデータをエクスポートできます。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・コマンドを埋め込む場合。

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。

UNLOAD 文を使用するには、文の一部として指定する SELECT の実行に必要なパーミッションが必要です。

UNLOAD 文を使用できるユーザの管理の詳細については、「-gl 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。

UNLOAD 文は現在の独立性レベルで実行されます。

詳細については、「UNLOAD 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

例

SQL Anywhere のサンプル・データベースを使用し、次のコマンドを実行して Employees テーブルのサブセットをテキスト・ファイル *employee_data.csv* にアンロードできます。

```
UNLOAD
SELECT * FROM Employees
WHERE State = 'GA'
TO 'employee_data.csv';
```

データベース・サーバで結果セットをアンロードするため、*employee_data.csv* にはデータベース・サーバ・コンピュータ上のファイルを指定します。

dbunload ユーティリティを使用したデータのエクスポート

dbunload ユーティリティでは、データベース内の 1 つ、多数、またはすべてのテーブルをエクスポートできます。テーブル・データだけでなくテーブル・スキーマも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は、dbunload ユーティリティを使用して必要なコマンド・ファイルを作成し、必要に応じて修正できます。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。

Copyright © 2007, iAnywhere Solutions, Inc.

コマンド・ファイルを使用してもしなくても、1 つまたは多くのテーブルを抽出できます。 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。

注意
dbunload ユーティリティと 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。

次の場合は、dbunload ユーティリティを使用します。

- 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。

dbunload ユーティリティの詳細については、「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』を参照してください。

アンロード・ツール

次のツールを使用して、データベースからデータをアンロードできます。

- 「dbunload ユーティリティを使用したデータのエクスポート」 697 ページ
- 「[データベース・アンロード] ウィザードの使用」 698 ページ
- 「[データのアンロード] ダイアログの使用」 699 ページ

[データベース・アンロード] ウィザードの使用

Sybase Central の [データベース・アンロード] ウィザードを使用して、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できます。

このウィザードを使用してデータベースをアンロードするときに、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。[所有者別にオブジェクトをフィルタ] ダイアログ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させたい場合、アンロードするデータベースを右クリックし、ポップアップ・メニューから [所有者別にオブジェクトをフィルタ] を選択して、表示されるダイアログから対象ユーザを選択します。

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

この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 ASCII 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 Interactive SQL コマンド・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。

dbunload ユーティリティと 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。

注意
アンロードするデータベースが実行中のときに [データベース・アンロード] ウィザードを起動すると、アンロードの前に SQL Anywhere プラグインによってデータベースが自動的に停止されます。

データベースのアンロードに関する特記事項については、「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』を参照してください。

♦ データベース・ファイルまたは実行中のデータベースをアンロードするには、次の手順に従います (Sybase Central の場合)。

1. [ツール] メニューから [SQL Anywhere 10] – [データベースのアンロード] を選択します。

   [データベース・アンロード] ウィザードが表示されます。

2. ウィザードの指示に従います。

ヒント
Sybase Central では、次の方法で [データベース・アンロード] ウィザードを利用することもできます。

- データベースを選択し、[ファイル] メニューから [データベースのアンロード] を選択します。
- データベースを右クリックし、ポップアップ・メニューから [データベースのアンロード] を選択する。

[データのアンロード] ダイアログの使用

Sybase Central の [データのアンロード] ダイアログを使用して、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、このダイアログを使用すると、[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。

♦ [データのアンロード] ダイアログを使用してテーブルをアンロードするには、次の手順に従います。

1. Sybase Central のデータベースに接続します。
2. コンテキスト・ドロップダウン・リストからデータベースを選択します。
3. [テーブル] フォルダを開きます。
4. データのエクスポート元のテーブルを選択します。

   Ctrl キーを押したままでテーブルをクリックすると複数のテーブルを選択できます。
5. [ファイル] メニューから、[データのアンロード] を選択します。

   [データのアンロード] ダイアログが表示されます。
6. 目的のオプションを選択し、[OK] をクリックしてデータをアンロードします。

クエリ結果のエクスポート

Interactive SQL から [データ] メニュー、OUTPUT 文、または UNLOAD 文を使用して、クエリ (ビューのクエリを含む) をファイルにエクスポートできます。

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。

詳細については、「Adaptive Server Enterprise の互換性」 723 ページを参照してください。

♦ クエリ結果をエクスポートするには、次の手順に従います (Interactive SQL の [データ] メニューの場合)。

1. Interactive SQL の [SQL 文] ウィンドウ枠にクエリを入力します。
2. [SQL 文の実行] をクリックして、結果セットを表示します。
3. [データ] メニューから [エクスポート] を選択します。

   [名前を付けて保存] ダイアログが表示されます。
4. エクスポートするデータの名前とロケーションを指定します。
5. ファイル・フォーマットを指定して [OK] 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。

 Copyright © 2007, iAnywhere Solutions, Inc.

♦ クエリ結果をエクスポートするには、次の手順に従います (Interactive SQL の OUTPUT 文の場合)。

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.txt* 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 'employees.txt';
   ```

3. 次のいずれかを行います。

処理	使用する句	例
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。	APPEND	SELECT * FROM Employees; OUTPUT TO 'employees.txt' 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。	VERBOSE	SELECT * FROM Employees; OUTPUT TO 'employees.txt' VERBOSE;
ASCII (デフォルト) 以外のファイル・フォーマットを指定する。	FORMAT	SELECT * FROM Employees; OUTPUT TO 'employees.dbf' FORMAT DBASEIII; *employees.dbf* は新しいファイルの名前と拡張子で、DBASEIII はファイル・フォーマットです。一重引用符または二重引用符で文字列を囲むことができますが、引用符が必要となるのは、ファイル・パスまたはファイル名にスペースが含まれる場合のみです。 FORMAT オプションを指定しない場合、デフォルトのファイル・タイプは ASCII になります。

4. SQL メニューから [実行] を選択して文を実行します。

   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。

   OUTPUT 文を使用したクエリ結果のエクスポートの詳細については、「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

ヒント
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** のように入力します。
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。現在も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、新しい Interactive SQL 文を使用すると、出力がより正確になり、コードが速く読み込まれます。
APPEND と VERBOSE の詳細については、「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

♦ クエリ結果をエクスポートするには、次の手順に従います (UNLOAD 文の場合)。

1. [SQL 文] ウィンドウ枠で、UNLOAD 文を入力します。次に例を示します。

```
UNLOAD
SELECT *
FROM Employees
TO 'employee_data.csv';
```

2. SQL メニューから [実行] を選択して UNLOAD 文を実行します。

   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。

OUTPUT 文を使用した NULL 値の出力

他のソフトウェアで使うためにデータを抽出する場合、他のソフトウェア製品では NULL 値を解釈できないことがあるため、Interactive SQL で OUTPUT 文を使用した NULL 値の指定には 2 つの方法があります。

- output_nulls オプションを使用すると、OUTPUT 文で使用する出力値を指定できます。
- IFNULL 関数を使用すると、特定のインスタンスまたはクエリに出力値を適用できます。

どちらのオプションの場合も、NULL 値の代わりに指定した値を出力できます。NULL 値をどのように出力するかを指定すると、他のソフトウェア製品との互換性を高めることができます。

♦ NULL 値の出力を指定するには、次の手順に従います (Interactive SQL の場合)。

· SET OPTION 文を実行して、output_nulls オプションの値を目的の値に変更します。

  次の例では、NULL 値として表示される値を (不明) に変更します。

```
SET OPTION output_nulls = '(unknown)';
```

 Copyright © 2007, iAnywhere Solutions, Inc.

Interactive SQL オプションの詳細については、「SET OPTION 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

> **ヒント**
> Interactive SQL の [結果] ウィンドウ枠で、NULL 値として表示される値を変更できます。[ツール] メニューから [オプション] を選択します。Interactive SQL の [オプション] ダイアログの [結果] タブで、[NULL 値の表示形式] の値を設定します。

データベースのエクスポート

> **注意**
> アンロードするデータベースが実行中のときに [データベース・アンロード] ウィザードを起動すると、アンロードの前に SQL Anywhere プラグインによってデータベースが自動的に停止されます。

♦ データベースの全部または一部をアンロードするには、次の手順に従います (Sybase Central の場合)。

1. [ツール] メニューから [SQL Anywhere 10] – [データベースのアンロード] を選択します。

   [データベース・アンロード] ウィザードが表示されます。

2. ウィザードの指示に従います。

♦ データベースの全部または一部をアンロードするには、次の手順に従います (コマンド・ラインの場合)。

1. コマンド・プロンプトで dbunload コマンドを入力し、-c オプションを使用して接続パラメータを指定します。

   たとえば、次のコマンドでは、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:¥DataFiles* にアンロードします。

   ```
   dbunload -c "DBN=demo;UID=DBA;PWD=sql" c:¥DataFiles
   ```

   スキーマの再作成と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。

2. 次のいずれかを行います。

   - データのみをエクスポートするには、-d を使用します。次に例を示します。

     ```
     dbunload -c "DBN=demo;UID=DBA;PWD=sql" -d c:¥DataFiles
     ```

     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。

   - スキーマのみをエクスポートするには、-n 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。

詳細については、「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』を参照してください。

テーブルのエクスポート

ヒント
以下の各項で説明する方法の他に、テーブルのデータをすべて選択したクエリ結果をエクスポートして、テーブルをエクスポートすることもできます。
詳細については、「クエリ結果のエクスポート」700 ページを参照してください。

ヒント
ビューのエクスポートは、テーブルのエクスポートと同じように実行できます。

♦ **テーブルをエクスポートするには、次の手順に従います (コマンド・ラインの場合)。**

・ コマンド・プロンプトで、次の dbunload コマンドを入力し、[Enter] キーを押します。

```
dbunload -c "DBN=demo;UID=DBA;PWD=sql"
-t Employees c:¥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名で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:¥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コマンド・ファイルは、デフォルト名 *reload.sql* として現在のローカル・ディレクトリに作成されます。

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。

♦ **テーブルをエクスポートするには、次の手順に従います (SQL の場合)。**

・ UNLOAD TABLE 文を実行します。次に例を示します。

```
UNLOAD TABLE Departments
TO 'departments.csv';
```

この文は、SQL Anywhere サンプル・データベースの Departments テーブルをデータベース・サーバの現在の作業ディレクトリにある departments.csv ファイルにアンロードします。ネットワーク・データベース・サーバに対して実行する場合、このコマンドはクライアント・コンピュータにではなく、サーバ上のファイルにデータをアンロードします。また、ファイル名はサーバに文字列として渡されます。ディレクトリ名やファイル名が n (¥n は改

Copyright © 2007, iAnywhere Solutions, Inc.

行文字) またはその他の特殊文字で始まる場合は、ファイル名に円記号 (エスケープ文字) を使用すれば誤った解釈を避けることができます。

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。

UNLOAD TABLE 文の構文の詳細については、「UNLOAD TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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) と抽出のプロシージャを使用すると、データベースを再構築して、既存のデータベースの一部から新しいデータベースを作成し、未使用のページを排除できます。

データベースを再構築して新しいバージョンの SQL Anywhere にアップグレードする場合は、「SQL Anywhere のアップグレード」『SQL Anywhere 10 - 変更点とアップグレード』 を参照してください。

データベースは、Sybase Central や dbunload ユーティリティで再構築できます。

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。詳細については、「バックアップとリカバリの概要」『SQL Anywhere サーバ - データベース管理』を参照してください。

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。

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、ASCII 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。

Copyright © 2007, iAnywhere Solutions, Inc.

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。「データベースの抽出」 715 ページを参照してください。

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 と Replication 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。

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。次の処理が必要な場合は、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。特定の機能を有効にするためにアンロードと再ロードが必要かどうかについては、新機能に関するマニュアルで説明します。

  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に依存する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 より前からバージョン 10 に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 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。

  データベースのアップグレードの詳細については、「SQL Anywhere のアップグレード」 『SQL Anywhere 10 - 変更点とアップグレード』を参照してください。

  SQL Anywhere のアップグレード、またはデータベースのミラーリングに使用しているデータベースの再構築については、「データベース・ミラーリング・システムでの SQL Anywhere ソフトウェアとデータベースのアップグレード」 『SQL Anywhere 10 - 変更点とアップグレード』を参照してください。

- **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。「REORGANIZE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

参照

- 「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』

再構築の考慮事項

再構築 (アンロードとロード) と抽出のプロシージャを使用すると、データベースを再構築して、古いデータベースの一部から新しいデータベースを作成できます。データベースを再構築する前に、プロセスの開始方法を検討してください。

検討事項	参照先
データベースの再構築に関連するもの	「データベースの再構築」 706 ページ 「データベースを再構築する理由」 707 ページ
再構築に使用可能なツール	「再構築ツールと再ロード・ツール」 711 ページ
再構築がデータベース・ユーザに与える影響	「再構築中のダウン時間の最短化」 714 ページ
データベースが同期またはレプリケーションに関連するかどうか	「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」 710 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」 709 ページ
データベースの照合順の変更	「データベースの照合を変更する」『SQL Anywhere サーバ - データベース管理』

Copyright © 2007, iAnywhere Solutions, Inc.

検討事項	参照先
実体化ビュー (Materialized View) を再表示する必要があるかどうか	「実体化ビュー (Materialized View) のリフレッシュ」 78 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

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。

♦ 同期やレプリケーションに関連しないデータベースを再構築するには、次の手順に従います (コマンド・ラインの場合)。

1. コマンド・プロンプトで、次のオプションのいずれかを使用して、dbunload ユーティリティを実行します。

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=NewDemo.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。-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。

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。

注意

-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。

dbunload ユーティリティでは追加オプションを使用できます。追加オプションを使用すると、アンロード作業をチューニングできます。その他、実行中または実行していないデータベースとデータベース・パラメータを指定できる接続パラメータ・オプションもあります。

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

この項は、SQL Anywhere の Mobile Link クライアント (dbmlsync を使用するクライアント) とともに、SQL Remote と Replication Agent にも適用されます。

データベースが同期またはレプリケートされている場合、データベースの再構築には特に注意が必要です。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が関係するときは、バックアップをきちんと実行することが特に重要です。

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には、2 つの方法があります。最初の方法では、dbunload ユーティリティの -ar オプションを使用して、同期やレプリケーションに干渉しない方法でアンロードと再ロードを実行します。もう 1 つの方法では、手動で同じタスクを実行します。

♦ 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するには、次の手順に従います (dbunload ユーティリティの場合)。

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. コマンド・プロンプトで、dbunload 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 DBA 権限での接続を表し、*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。

   詳細については、「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』を参照してください。
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。

   妥当性検査の詳細については、「データベースの検証」 『SQL Anywhere サーバ - データベース管理』を参照してください。
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。

注意

dbunload ユーティリティでは追加オプションを使用できます。追加オプションを使用すると、アンロード作業をチューニングできます。その他、実行中または実行していないデータベースとデータベース・パラメータを指定できる接続パラメータ・オプションもあります。「アンロー

 Copyright © 2007, iAnywhere Solutions, Inc.

ド・ユーティリティ (dbunload)」『SQL Anywhere サーバ - データベース管理』を参照してください。

前述の手順では要求に対応できない場合は、トランザクション・ログのオフセットを手動で調整できます。次の手順では、このような操作の実行方法について説明します。

♦ 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操作して再構築するには、次の手順に従います。

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。

   この手順については、「データベースの再構築」 706 ページを参照してください。
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 パラメータに指定し、相対オフセットを 0 に設定します。

   ```
   dblog -x 0 -z 137829 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。

再構築ツールと再ロード・ツール

データベースを再構築できるさまざまなツールがあります。既存のデータベースにデータを再ロードできるツールもいくつかあります。

dbunload ユーティリティを使用したデータの再構築

dbunload ユーティリティまたは dbisql ユーティリティを使用すると、データベース全体をカンマ区切りの ASCII 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 Interactive SQL コマンド・ファイルを作成できます。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。このユーティリティは、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。

> **注意**
> dbunload ユーティリティと 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。

次の場合は、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。

詳細については、次の項を参照してください。

- 「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」 709 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」 710 ページ

UNLOAD TABLE 文を使用したデータベースの再構築

UNLOAD TABLE 文を使用すると、特定の文字エ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。

詳細については、「UNLOAD TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

テーブル・データまたはテーブル・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・データやテーブル・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。

ここまでの例の dbunload コマンドは、データまたはスキーマを、SQL Anywhere サンプル・データベース・テーブル (デフォルトのデータベース名を持つ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:¥DataFiles* ディレクトリのファイルにアンロードします。スキーマの再作成と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。

 Copyright © 2007, iAnywhere Solutions, Inc.

♦ テーブル・データをエクスポートするには、次の手順に従います (コマンド・ラインの場合)。

- コマンド・プロンプトで dbunload コマンドを入力し、-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、-d オプションを指定してデータのみをアンロードするよう指定します。

  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" -d -t Employees c:¥DataFiles
  ```

  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。

♦ テーブル・スキーマをエクスポートするには、次の手順に従います (コマンド・ラインの場合)。

- コマンド・プロンプトで dbunload コマンドを入力し、-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よう指定します。

  たとえば、Employees テーブルからスキーマ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
  ```

  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。

データベースの再ロード

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。データベースは、コマンド・ラインから再ロードします。

♦ データベースを再ロードするには、次の手順に従います (コマンド・ラインの場合)。

1. コマンド・プロンプトを開きます。
2. 新しい空のデータベース・ファイルを作成します。

   たとえば、次のコマンドは *newdemo.db* という名前のファイルを作成します。

   ```
   dbinit newdemo.db
   ```

3. *reload.sql* スクリプトを実行します。

   たとえば、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=newdemo.db;UID=DBA;PWD=sql" reload.sql
   ```

Copyright © 2007, iAnywhere Solutions, Inc.

再構築中のダウン時間の最短化

次の手順を実行すると、ダウン時間を最短に抑えてデータベースを再構築できます。これは、データベースが 1 日 24 時間稼働している場合に特に役立ちます。

手順 1 ー 4 を試験的に実行し、各手順に必要な時間を判断してから実際に再構築を開始することをおすすめします。また、再構築中のさまざまな時点でファイルのコピーを保存できます。

警告
運用データベースのログ名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かどうかを確認してください。この種のバックアップが誤って発生した場合は、再構築されたデータベースに、名前が変更されたログからのトランザクションを適切な順序で適用する必要が出てきます。

♦ **再構築中のダウン時間を最小化するには、次の手順に従います。**

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ログの名前を変更します。

   詳細については、「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』を参照してください。

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 を実行し、再構築したサーバにトランザクションを適用します。

   詳細については、「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』を参照してください。

   再構築したデータベースには、手順 3 でバックアップが終了した時点までのトランザクションがすべて含まれています。

5. 運用サーバを停止し、データベースとログのコピーを作成します。

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 のログに対して dbtran を実行します。

   このファイルは比較的小さなサイズです。

8. 再構築したデータベースに対してサーバを起動します。ただし、ユーザには接続を許可しないでください。

9. 手順 8 のトランザクションを適用します。

10. ユーザに接続を許可します。

 Copyright © 2007, iAnywhere Solutions, Inc.

データベースの抽出

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。

データベースを抽出する方法の詳細については、次を参照してください。

- 「データベース抽出ユーティリティ」 『SQL Remote』
- 「抽出ユーティリティ (dbxtract) の使用」 『SQL Remote』
- 「グループの抽出」 『SQL Remote』
- 「リモート・データベースの配備」 『Mobile Link - クライアント管理』
- 「Sybase Central でのリモート・データベースの抽出」 『SQL Remote』

SQL Anywhere へのデータベースの移行

ストアド・プロシージャのセット sa_migrate または [データベース移行] ウィザードを使用して、リモートの Oracle、DB2、Microsoft SQL Server、Sybase Adaptive Server Enterprise、SQL Anywhere データベースのテーブルを SQL Anywhere にインポートできます。

[データベース移行] ウィザードやストアード・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。

データベースの作成については、「データベースの作成」 31 ページを参照してください。

Access データベースの移行
SQL Anywhere for MS Access 移行ユーティリティ (upsize ツール) を使用して、Microsoft Access データベースを SQL Anywhere に移行できます。このユーティリティは、http://www.ianywhere.com/developer/code_samples/sqlany_migration_utility.html からダウンロードできます。

[データベース移行] ウィザードの使用

[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外部ログインを作成できます。

♦ **リモート・テーブルをインポートするには、次の手順に従います (Sybase Central の場合)。**

1. Sybase Central からターゲット・データベースに接続します。
2. [ツール] – [SQL Anywhere 10] – [データベースの移行] を選択します。

   [データベース移行] ウィザードが表示されます。
3. リストからターゲット・データベースを選択し、[次へ] をクリックします。
4. 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・サーバをまだ作成していない場合は、[今すぐにリモート・サーバを作成] をクリックして [リモート・サーバ作成] ウィザードを開きます。

   リモート・サーバの作成の詳細については、「リモート・サーバの作成」 729 ページを参照してください。

   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。

Copyright © 2007, iAnywhere Solutions, Inc.

5. 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。

6. ターゲット・データベースで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。

   ユーザをまだ作成していない場合は、[今すぐにユーザを作成] をクリックして [Mobile Link ユーザ作成] ウィザードを開きます。

   詳細については、「新しいユーザの作成」 『SQL Anywhere サーバ - データベース管理』を参照してください。

7. 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。

8. [完了] をクリックします。

sa_migrate ストアド・プロシージャの使用

sa_migrate ストアド・プロシージャを使用することにより、リモート・データを移行できます。テーブルをまったく修正しないのであれば、2 ステップの方法を使用できます。それに対して、テーブルや外部キー・マッピングを削除するのであれば、拡張方法を使用します。

ストアド・プロシージャのセット sa_migrate を使用している場合は、必ず次の手順を実行してから、リモート・データベースをインポートします。

1. ターゲット・データベースを作成します。

   詳細については、「データベースの作成」 31 ページを参照してください。

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。

   詳細については、「リモート・サーバの作成」 729 ページを参照してください。

3. 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。

   詳細については、「外部ログインの作成」 737 ページを参照してください。

4. ターゲット・データベースに移行するテーブルを所有するローカル・ユーザを作成します。

   詳細については、「新しいユーザの作成」 『SQL Anywhere サーバ - データベース管理』を参照してください。

sa_migrate を使用した 2 ステップのデータベースの移行

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。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。

♦ リモート・テーブルをインポートするには、次の手順に従います (2 ステップの場合)。

1. Interactive SQL からターゲット・データベースに接続します。
2. Interactive SQL の [SQL 文] ウィンドウ枠で、sa_migrate ストアド・プロシージャを実行します。次に例を示します。

   ```
   CALL sa_migrate( 'local_a', 'ase', NULL, l_smith, NULL, 1, 1, 1 );
   ```

   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 l_smith に属するリモート・テーブルをすべて移行します。

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。

詳細については、「sa_migrate 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。

sa_migrate ストアド・プロシージャを使用した個別のテーブルの移行

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。

♦ リモート・テーブルをインポートするには、次の手順に従います (修正がある場合)。

1. Interactive SQL からターゲット・データベースに接続します。
2. sa_migrate_create_remote_table_list ストアド・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( 'ase',
       NULL, 'remote_a', 'mydb' );
   ```

   リモート・サーバの名前を指定してください。

   これにより、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。

   詳細については、「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』を参照してください。
3. sa_migrate_create_tables ストアド・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_a' );
   ```


Copyright © 2007, iAnywhere Solutions, Inc.

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。

詳細については、「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』を参照してください。

4. 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 ストアド・プロシージャを実行します。次に例を示します。

   次の文を実行します。

   ```
   CALL sa_migrate_data( 'local_a' );
   ```

   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。

   詳細については、「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』を参照してください。

   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 7 に進みます。

5. sa_migrate_create_remote_fks_list ストアド・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( 'ase' );
   ```

   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 に移植します。

   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。

   詳細については、「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』を参照してください。

6. sa_migrate_create_fks ストアド・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_a' );
   ```

   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。

   詳細については、「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』を参照してください。

7. 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 ストアド・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_a' );
   ```

   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。

   詳細については、「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』を参照してください。

SQL コマンド・ファイルの使用

この項では、一連のコマンドで構成されるファイルの処理方法について説明します。「**コマンド・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。

コマンド・ファイルの作成

コマンド・ファイルは、好みのテキスト・エディタを使用して作成できます。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。コマンド・ファイルは、一般的には、「**スクリプト**」ともいいます。

Interactive SQL で SQL コマンド・ファイルを開く

Windows プラットフォームでは、Interactive SQL を *.sql* ファイルのデフォルト・エディタにすることができます。このため、ファイルをダブルクリックすると、ファイルの内容が Interactive SQL の [SQL 文] ウィンドウ枠に表示されます。

Interactive SQL を *.sql* ファイルのデフォルト・エディタに設定する方法の詳細については、「[オプション] ダイアログ：[一般] タブ」『SQL Anywhere 10 - コンテキスト別ヘルプ』を参照してください。

SQL コマンド・ファイルの実行

コマンド・ファイルは次のいずれかの方法で実行できます。

- Interactive SQL の READ 文を使用して、コマンド・ファイルを実行する。たとえば、次の文はファイル *temp.sql* を実行します。

  ```
  READ temp.sql;
  ```

  詳細については、「READ 文 [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- コマンド・ファイルを [SQL 文] ウィンドウ枠にロードして、そこから直接実行する。

  コマンド・ファイルを [SQL 文] ウィンドウ枠にロードするには、[ファイル] – [開く] を選択します。ファイル名の入力を要求された場合は、*temp.sql* と入力します。

- Interactive SQL のコマンド・ライン引数としてコマンド・ファイルを指定する。

SQL コマンド・ファイルの実行

コマンド・ファイルは、対話型またはバッチ・モードで実行できます。

♦ **すぐにコマンド・ファイルを実行するには、次の手順に従います。**

1. Interactive SQL の [ファイル] メニューから [スクリプトの実行] を選択します。

Copyright © 2007, iAnywhere Solutions, Inc.

[開く] ダイアログが表示されます。

2. [開く] ダイアログでファイルを検索し、[開く] をクリックします。

   [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。READ 文の例は、次のとおりです。

♦ [SQL 文] ウィンドウ枠を使用してコマンド・ファイルを実行するには、次の手順に従います。

・ 次のコマンドを入力します。

```
READ 'c:¥¥filename.sql';
```

この文の *c:¥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。

♦ コマンド・ファイルをバッチ・モードで実行するには、次の手順に従います。

1. コマンド・プロンプトを開きます。
2. コマンド・ファイルを、Interactive SQL のコマンド・ライン引数として指定します。

   たとえば、次のコマンドは、SQL Anywhere サンプル・データベースに対してコマンド・ファイル *myscript.sql* を実行します。

```
dbisql -c "DSN=SQL Anywhere 10 Demo" myscript.sql
```

SQL コマンド・ファイルのロード

新しい Interactive SQL セッションを開始するとき、コマンド・ファイルの内容を [SQL 文] ウィンドウ枠にロードするか、コマンドをすぐに実行できます。

♦ ファイルのコマンドを [SQL 文] ウィンドウ枠にロードするには、次の手順に従います。

1. [ファイル] メニューで [開く] を選択します。

   [開く] ダイアログが表示されます。

2. [開く] ダイアログでファイルを検索し、選択します。
3. [開く] をクリックして、コマンド・ファイルをロードします。

ファイルへのデータベース出力の書き込み

Interactive SQL では、各コマンドの結果セット・データは次のコマンドが実行されると [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。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を *file1* と *file2* に出力します。

```
statement1; OUTPUT TO file1;
statement2; OUTPUT TO file2;
```

たとえば、次のコマンドはクエリの結果を *Employees.txt* という名前のファイルに保存します。

```
SELECT * FROM EMPLOYEE;
OUTPUT TO 'C:¥¥My Documents¥¥Employees.txt';
```

詳細については、「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

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。BCP 出力はデリミタで区切られた ASCII フォーマットです。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。

BCP と FORMAT 句の詳細については、「LOAD TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』または「UNLOAD TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

第 18 章

リモート・データへのアクセス

目次

リモート・データ・アクセスの概要 726
リモート・データ・アクセスの基本概念 727
リモート・サーバの使用 729
ディレクトリ・アクセス・サーバの使用 734
外部ログインの使用 737
プロキシ・テーブルの編集 739
リモート・テーブルのジョイン 743
複数のローカル・データベースのテーブルのジョイン 745
ネイティブ文のリモート・サーバへの送信 746
リモート・プロシージャ・コール (RPC) の使用 747
トランザクションの管理とリモート・データ 750
内部オペレーション 752
リモート・データ・アクセスのトラブルシューティング 756

Copyright © 2007, iAnywhere Solutions, Inc.

リモート・データ・アクセスの概要

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、Oracle、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、MS-Access データベース、FoxPro、テキスト・ファイルなどのデスクトップ・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その他のデータ・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 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・ソースにサポートされていない機能を SQL Anywhere が補います。
- パススルー・モードを使用して、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
- Oracle
- IBM DB2
- Microsoft SQL Server
- その他の ODBC データ・ソース

利用可能なプラットフォームについては、「プラットフォーム別 SQL Anywhere 10.0.1 コンポーネント」にある SQL Anywhere の表を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

リモート・データ・アクセスの基本概念

この項では、リモート・データのアクセスに必要な基本概念について説明します。

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・テーブルをクライアントのローカル・テーブルとして見せるには、そのリモート・データに対するローカルのプロキシ・テーブルを作成します。

♦ **プロキシ・テーブルを作成するには、次の手順に従います。**

1. リモート・データが置かれているサーバを定義します。これによってサーバのタイプとリモート・サーバの場所を指定します。「リモート・サーバの使用」 729 ページを参照してください。
2. ローカル・サーバのログイン情報とリモート・サーバのログイン情報とが異なる場合は、ローカル・ユーザのログイン情報をリモート・サーバのユーザのログイン情報にマップします。「外部ログインの使用」 737 ページを参照してください。
3. プロキシ・テーブルの定義を作成します。これによってローカルのプロキシ・テーブルからリモート・テーブルへ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。

詳細については、「プロキシ・テーブルの編集」 739 ページを参照してください。

リモート・テーブルのマッピングの管理

リモート・テーブルのマッピングとリモート・サーバの定義を管理するには、Sybase Central を使用することもできますし、Interactive SQL などのツールを使用して SQL 文を直接実行することもできます。

> **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、SQL Anywhere プラグインの [データ] タブを使用してプロキシ・テーブルの内容を表示または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、DB/2、SQL Anywhere などのその他の RDBMS ではこの制限はありません。

サーバ・クラス

「**サーバ・クラス**」は、サーバとのアクセスに使用するアクセス方法を指定します。サーバ・クラスは各リモート・サーバに割り当てられます。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・クラスには 2 つのグループがあります。1 つは ODBC ベース、もう 1 つは JDBC ベースです。

ODBC ベースのサーバ・クラスを次に示します。

- **saodbc**　SQL Anywhere
- **aseodbc**　Sybase SQL Server と Adaptive Server Enterprise (バージョン 10 以降)
- **db2odbc**　IBM DB2
- **mssodbc**　Microsoft SQL Server
- **oraodbc**　Oracle サーバ (バージョン 8.0 以降)
- **odbc**　その他の ODBC データ・ソース

> **注意**
> JDBC クラスはパフォーマンスに重大な影響を及ぼすため、NetWare のように ODBC クラスを使用できない場合にのみ使用してください。

JDBC ベースのサーバ・クラスを次に示します。

- **sajdbc**　SQL Anywhere
- **asejdbc**　Sybase SQL Server と Adaptive Server Enterprise (バージョン 10 以降)

リモート・サーバ・クラスの詳細については、「リモート・データ・アクセスのサーバ・クラス」 759 ページを参照してください。

PowerBuilder DataWindows からのリモート・データへのアクセス

接続時に DBParm Block パラメータを 1 に設定することによって、PowerBuilder DataWindow からリモート・データにアクセスできます。

- 設計環境で、[データベース プロファイル セットアップ] ダイアログの [トランザクション] タブを開いて [Retrieve ブロック化係数] を 1 に設定することによって、Block パラメータを設定できます。
- 接続文字列で、次のパラメータを使用します。

```
DBParm="Block=1"
```


Copyright © 2007, iAnywhere Solutions, Inc.

リモート・サーバの使用

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、ISYSSERVER システム・テーブルにエントリが追加されます。この項では、リモート・サーバ定義を作成、変更、削除する方法について説明します。

リモート・サーバの作成

CREATE SERVER 文を使用して、リモート・サーバ定義を設定します。この文を直接実行するか、または Sybase Central を使用します。

ODBC 接続では、各リモート・サーバは ODBC データ・ソースに対応します。SQL Anywhere を含むいくつかのシステムでは各データ・ソースがデータベースを記述するので、データベースのそれぞれに個別のリモート・サーバ定義が必要になります。

リモート・サーバを作成するには、RESOURCE 権限が必要です。

UNIX プラットフォームでは、ODBC ドライバ・マネージャも参照する必要があります。

CREATE SERVER 文の詳細については、「CREATE SERVER 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

例 1

次の文は、RemoteASE という Adaptive Server Enterprise サーバのエントリを ISYSSERVER システム・テーブルに作成します。

```
CREATE SERVER RemoteASE
CLASS 'ASEJDBC'
USING 'rimu:6666';
```

- **RemoteASE**　リモート・サーバの名前
- **ASEJDBC**　リモート・サーバが Adaptive Server Enterprise であり、そのサーバへの接続には JDBC が使われることを示すキーワード
- **rimu:6666**　リモート・サーバが置かれているマシンの名前と TCP/IP ポート番号

例 2

次の文は、RemoteSA という ODBC ベースの SQL Anywhere サーバのエントリを ISYSSERVER システム・テーブルに作成します。

```
CREATE SERVER RemoteSA
CLASS 'SAODBC'
USING 'test4';
```

- **RemoteSA**　このデータベース内で識別するリモート・サーバの名前
- **SAODBC**　サーバが SQL Anywhere であり、そのサーバへの接続に ODBC が使われることを示すキーワード

Copyright © 2007, iAnywhere Solutions, Inc.

- **test4**　ODBC データ・ソース名 (DSN)

例 3

次の文は、UNIX プラットフォームで RemoteSA という ODBC ベースの SQL Anywhere サーバのエントリを ISYSSERVER システム・テーブルに作成します。

```
CREATE SERVER RemoteSA
CLASS 'SAODBC'
USING 'driver=SQL Anywhere 10;dsn=my_sa_dsn';
```

- **RemoteSA**　このデータベース内で識別するリモート・サーバの名前
- **SAODBC**　サーバが SQL Anywhere であり、そのサーバへの接続に ODBC が使われることを示すキーワード
- **USING**　ODBC ドライバ・マネージャへの参照

例 4

UNIX プラットフォームでは、次の文を使って、ODBC ベースの Adaptive Server Enterprise サーバ RemoteASE のエントリを ISYSSERVER システム・テーブルに作成します。

```
CREATE SERVER RemoteASE
CLASS 'ASEODBC'
USING '/opt/sybase/ase_odbc_1500/DataAccess/ODBC/lib/libsybdrvodb.so;dsn=my_ase_dsn'
```

- **RemoteASE**　このデータベース内で識別するリモート・サーバの名前
- **ASEODBC**　サーバが Adaptive Server Enterprise であり、そのサーバへの接続に ODBC が使われることを示すキーワード
- **USING**　ODBC ドライバ・マネージャへの参照

Sybase Central を使用したリモート・サーバの作成

リモート・サーバを、Sybase Central のウィザードを使用して作成できます。

◆ リモート・サーバを作成するには、次の手順に従います (Sybase Central の場合)。

1. Sybase Central からホスト・データベースに接続します。
2. データベースの [リモート・サーバ] フォルダを開きます。
3. [ファイル] − [新規] − [リモート・サーバ] を選択します。

   [リモート・サーバ作成] ウィザードが表示されます。
4. ウィザードの最初のページで、リモート・サーバに使用する名前を入力します。この名前は、ローカル・データベース内でリモート・サーバを識別するためのものです。サーバが提供する名前に一致させる必要はありません。[次へ] をクリックします。
5. 該当するサーバのタイプを選択して、[次へ] をクリックします。
6. データ・アクセス・メソッド (ODBC か JDBC) を選択して接続情報を指定します。

Copyright © 2007, iAnywhere Solutions, Inc.

- ODBC の場合は、データ・ソース名を指定するか、ODBC Driver= パラメータを指定します。
- JDBC の場合は、*machine-name*:*port-number* の形式で URL を指定します。
- [次へ] をクリックします。

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。

7. 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ます。[次へ] をクリックします。
8. リモート・データベース用の外部ログインを作成します。

   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。「CREATE EXTERNLOGIN 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
9. オプションとして、リモート・サーバ定義が作成される前に [接続のテスト] をクリックして、リモート・サーバに接続できるかどうかをテストできます。

   [完了] をクリックして、リモート・サーバ定義を作成します。

リモート・サーバの削除

Sybase Central または DROP SERVER 文を使用すると、ISYSSERVER システム・テーブルからリモート・サーバを削除できます。このアクションを実行するには、このサーバ上に定義されているすべてのリモート・テーブルが、すでに削除されていなければなりません。

リモート・サーバを削除するには、DBA 権限が必要です。

♦ リモート・サーバを削除するには、次の手順に従います (Sybase Central の場合)。

1. Sybase Central からホスト・データベースに接続します。
2. [リモート・サーバ] フォルダを開きます。
3. 削除するリモート・サーバを選択し、[ファイル] – [削除] を選択します。

♦ リモート・サーバを削除するには、次の手順に従います (SQL の場合)。

1. InteractiveSQL からホスト・データベースに接続します。
2. DROP SERVER 文を実行します。

詳細については、「DROP SERVER 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

例

次の文は RemoteSA という名前のサーバを削除します。

```
DROP SERVER RemoteSA;
```

リモート・サーバの変更

Sybase Central または ALTER SERVER 文を使用すると、サーバの属性を変更できます。変更された設定は、次回リモート・サーバに接続する際に有効になります。

サーバを変更するには、RESOURCE 権限が必要です。

♦ リモート・サーバのプロパティを変更するには、次の手順に従います (Sybase Central の場合)。

1. Sybase Central からホスト・データベースに接続します。
2. データベースの [リモート・サーバ] フォルダを開きます。
3. リモート・サーバを選択し、[ファイル] – [プロパティ] を選択します。
4. サーバのプロパティを設定し、[OK] をクリックします。

♦ リモート・サーバのプロパティを変更するには、次の手順に従います (SQL の場合)。

1. InteractiveSQL からホスト・データベースに接続します。
2. ALTER SERVER 文を実行します。

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。サーバのデータ・ソース名は RemoteASE です。

```
ALTER SERVER RemoteASE
CLASS 'aseodbc';
```

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。「ALTER SERVER 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

サーバ上のリモート・テーブルのリスト

指定されたサーバで使用可能なリモート・テーブルのリストを取得するように SQL Anywhere を設定するときに、 sp_remote_tables システム・プロシージャを使用すると便利です。sp_remote_tables プロシージャは、リモート・サーバ上のテーブルのリストを返します。

```
sp_remote_tables(
@server-name
[, @table-name
[, @table-owner
[, @table-qualifier
```


Copyright © 2007, iAnywhere Solutions, Inc.

```
[, @with-table-type ] ] ] ]
)
```

table-name または *table-owner* を指定すると、テーブル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、リモート・サーバから使用可能なすべての Microsoft Excel シートの中から、excel という名前のもののリストを取得するには、次のようにします。

```
CALL sp_remote_tables  excel;
```

Adaptive Server Enterprise サーバ asetest の production データベースにある fred が所有するすべてのテーブルのリストを取得するには、次のようにします。

```
CALL sp_remote_tables  asetest, null, fred, production;
```

詳細については、「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』を参照してください。

リモート・サーバの機能のリスト

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 はこの機能情報から、どのくらいの SQL 文をリモート・サーバに渡すことができるかを判断します。

サーバの機能を保持するシステム・テーブルは、SQL Anywhere がリモート・サーバに最初に接続した後でないと、設定されません。この情報は ISYSCAPABILITY と ISYSCAPABILITYNAME システム・テーブルから取得されます。指定した *server-name* は、CREATE SERVER 文で使用する *server-name* と同じでなければなりません。

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』

Copyright © 2007, iAnywhere Solutions, Inc.

ディレクトリ・アクセス・サーバの使用

「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ディレクトリにアクセスします。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。

ディレクトリ・アクセス・サーバの作成

ディレクトリ・アクセス・サーバを作成するには、Sybase Central で CREATE SERVER 文や [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ます。

ディレクトリ・アクセス・サーバを作成すると、アクセスできるサブディレクトリ数を制限したり、既存のファイルの変更時にディレクトリ・アクセス・サーバを使用できるようになります。

ディレクトリ・アクセス・サーバの設定手順は次のとおりです。

1. ディレクトリのリモート・サーバを作成します (DBA 権限が必要)。
2. ディレクトリ・アクセス・サーバを使用するデータベース・ユーザの外部ログインを作成します (DBA 権限が必要)。
3. コンピュータ上のディレクトリにアクセスするためのプロキシ・テーブルを作成します (RESOURCE 権限が必要)。

♦ ディレクトリ・アクセス・サーバを作成して設定するには、次の手順に従います (Sybase Central の場合)。

1. 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] フォルダを開きます。
3. [ファイル] – [新規] – [ディレクトリ・アクセス・サーバ] を選択します。

   [ディレクトリ・アクセス・サーバの作成] ウィザードが表示されます。
4. ウィザードの指示に従います。

♦ ディレクトリ・アクセス・サーバを作成して設定するには、次の手順に従います (SQL の場合)。

1. CREATE SERVER 文を使用して、リモート・サーバを作成します。

   次に例を示します。

 Copyright © 2007, iAnywhere Solutions, Inc.

```
CREATE SERVER my_dir_tree
CLASS 'directory'
USING 'root=c:¥Program Files';
```

2. CREATE EXTERNLOGIN 文を使用して、外部ログインを作成します。

   次に例を示します。

```
CREATE EXTERNLOGIN DBA TO my_dir_tree;
```

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。

   次に例を示します。

```
CREATE EXISTING TABLE my_program_files AT 'my_dir_tree;;;.';
```

   この例では、my_program_files がディレクトリの名前、my_dir_tree がディレクトリ・アクセス・サーバの名前です。

例

次の文では、最大 3 レベルのサブディレクトリにアクセスできる directoryserver3 という名前のディレクトリ・アクセス・サーバを新規に作成し、DBA ユーザのディレクトリ・アクセス・サーバへの外部ログインを作成し、diskdir3 という名前のプロキシ・テーブルを作成します。

```
CREATE SERVER directoryserver3
CLASS 'DIRECTORY'
USING 'ROOT=c:¥mydir;SUBDIRS=3'
CREATE EXTERNLOGIN DBA TO directoryserver3;
CREATE EXISTING TABLE diskdir3 AT 'directoryserver3;;;.';
```

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ディレクトリを表示できます。

```
CALL sp_remote_tables( 'directoryserver3' );
```

次の SELECT 文を使用すると、*c:¥mydir¥myfile.txt* ファイルの内容を表示できます。

```
SELECT contents
FROM diskdir3
WHERE file_name = 'myfile.txt';
```

また、表示するサブディレクトリを選択することもできます。

```
-- Get the list of directories in this disk directory tree.
SELECT permissions, file_name, size
FROM diskdir3
WHERE PERMISSIONS LIKE 'd%';
-- Get the list of files.
SELECT permissions, file_name, size
FROM diskdir3
WHERE PERMISSIONS NOT LIKE 'd%';
```

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・サーバの削除

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 文を使用してディレクトリ・アクセス・サーバを削除してから、新規に作成してください。

ディレクトリ・アクセス・サーバの削除

♦ ディレクトリ・アクセス・サーバを削除するには、次の手順に従います (Sybase Central の場合)。

1. データベースの [ディレクトリ・アクセス・サーバ] フォルダを開きます。
2. ディレクトリ・アクセス・サーバを選択し、[編集] – [削除] を選択します。

♦ ディレクトリ・アクセス・サーバを削除するには、次の手順に従います (SQL の場合)。

- DROP SERVER 文を実行します。

  次に例を示します。

```
DROP SERVER my_directory_server;
```

プロキシ・テーブルの削除

DROP TABLE 文を使用して、ディレクトリ・アクセス・サーバで使用されているプロキシ・テーブルを削除します。

♦ プロキシ・テーブルを削除するには、次の手順に従います (Sybase Central の場合)。

1. データベースの [ディレクトリ・アクセス・サーバ] フォルダを開きます。
2. 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。
3. プロキシ・テーブルを選択し、[編集] – [削除] を選択します。

♦ プロキシ・テーブルを削除するには、次の手順に従います (SQL の場合)。

- DROP TABLE 文を実行します。

  次に例を示します。

```
DROP TABLE my_files;
```

参照

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

Copyright © 2007, iAnywhere Solutions, Inc.

外部ログインの使用

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。

詳細については、「統合化ログインの使用方法」 『SQL Anywhere サーバ - データベース管理』を参照してください。

外部ログインの作成

外部ログインは、Sybase Central または CREATE EXTERNLOGIN 文を使用して作成できます。

外部ログインを追加または変更できるのは、DBA または外部ログインを使用するユーザだけです。

♦ **外部ログインを作成するには、次の手順に従います (Sybase Central の場合)。**

1. Sybase Central からホスト・データベースに接続します。
2. データベースの [リモート・サーバ] フォルダを開いて、リモート・サーバを選択します。
3. [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] を選択します。

   [外部ログイン作成] ウィザードが表示されます。
5. ウィザードの指示に従います。

♦ **外部ログインを作成するには、次の手順に従います (SQL の場合)。**

1. InteractiveSQL からホスト・データベースに接続します。
2. CREATE EXTERNLOGIN 文を実行します。

例

次の文によって、ローカル・ユーザ fred は、banana というパスワードとリモート・ログイン frederick を使用してサーバ RemoteASE へアクセスします。

```
CREATE EXTERNLOGIN fred
TO RemoteASE
REMOTE LOGIN frederick
IDENTIFIED BY banana;
```

詳細については、「CREATE EXTERNLOGIN 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

外部ログインの削除

Sybase Central または DROP EXTERNLOGIN 文を使用すると、SQL Anywhere システム・テーブルから外部ログインを削除できます。

外部ログインを削除できるのは、DBA または外部ログインを使用するユーザだけです。

♦ 外部ログインを削除するには、次の手順に従います (Sybase Central の場合)。

1. Sybase Central からホスト・データベースに接続します。
2. [リモート・サーバ] フォルダを開きます。
3. 左ウィンドウ枠でリモート・サーバを選択し、次に右ウィンドウ枠で [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、[ファイル] – [削除] を選択します。

♦ 外部ログインを削除するには、次の手順に従います (SQL の場合)。

1. InteractiveSQL からホスト・データベースに接続します。
2. DROP EXTERNLOGIN 文を実行します。

例

次の文は、前述の例で作成したローカル・ユーザ fred の外部ログインを削除します。

```
DROP EXTERNLOGIN fred TO RemoteASE;
```

参照

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

Copyright © 2007, iAnywhere Solutions, Inc.

プロキシ・テーブルの編集

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、実体化ビュー (Materialized View) 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。「トランザクション内のセーブポイント」 123 ページを参照してください。

プロキシ・テーブルのロケーションの指定

AT キーワードを CREATE TABLE と CREATE EXISTING TABLE 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。場合によってはこのフィールドは適用せず、入力しないでおきます。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。

Copyright © 2007, iAnywhere Solutions, Inc.

- **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みなされます。

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：

```
'excel;d:¥pcdb¥quarter3.xls;;sheet1$'
```

- Access：

```
'access;¥¥server1¥production¥inventory.mdb;;parts'
```

プロキシ・テーブルの作成 (Sybase Central の場合)

プロキシ・テーブルは、Sybase Central または CREATE EXISTING TABLE 文を使用して作成できます。システム・テーブル用のプロキシ・テーブルは作成できません。

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 文の詳細については、「CREATE EXISTING TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

♦ **プロキシ・テーブルを作成するには、次の手順に従います (Sybase Central の場合)。**

1. Sybase Central からホスト・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] フォルダを開きます。
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. [ファイル] – [新規] – [プロキシ・テーブル] を選択します。
5. ウィザードの指示に従います。

Copyright © 2007, iAnywhere Solutions, Inc.

> **ヒント**
> 左ウィンドウ枠でリモート・サーバを選択すると、対応するプロキシ・テーブルが、右ウィンドウ枠の [プロキシ・テーブル] タブ上に表示されます。また、[テーブル] フォルダにも表示されます。

CREATE EXISTING TABLE 文を使用したプロキシ・テーブルの作成

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 TABLE 文を使用してプロキシ・テーブルを作成するには、次の手順に従います (SQL の場合)。

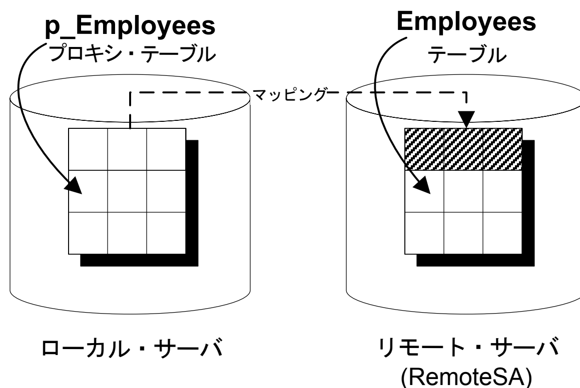
1. ホスト・データベースに接続します。
2. CREATE EXISTING TABLE 文を実行します。

詳細については、「CREATE EXISTING TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

例 1

サーバ RemoteSA のリモート・テーブル Employees に対して、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';
```



例 2

次の文は、プロキシ・テーブル a1 を Microsoft Access ファイル *mydbfile.mdb* にマップします。AT キーワードでは、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。

Copyright © 2007, iAnywhere Solutions, Inc.

```
CREATE EXISTING TABLE a1
AT 'access;d:¥mydbfile.mdb;;a1';
```

CREATE TABLE 文を使用したプロキシ・テーブルの作成

AT オプションとともに 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。

詳細については、「CREATE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』と「CREATE EXISTING TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

例

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・テーブルにマップする 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'
```

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。

```
sp_remote_columns servername, tablename [, owner ]
[, database]
```

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。

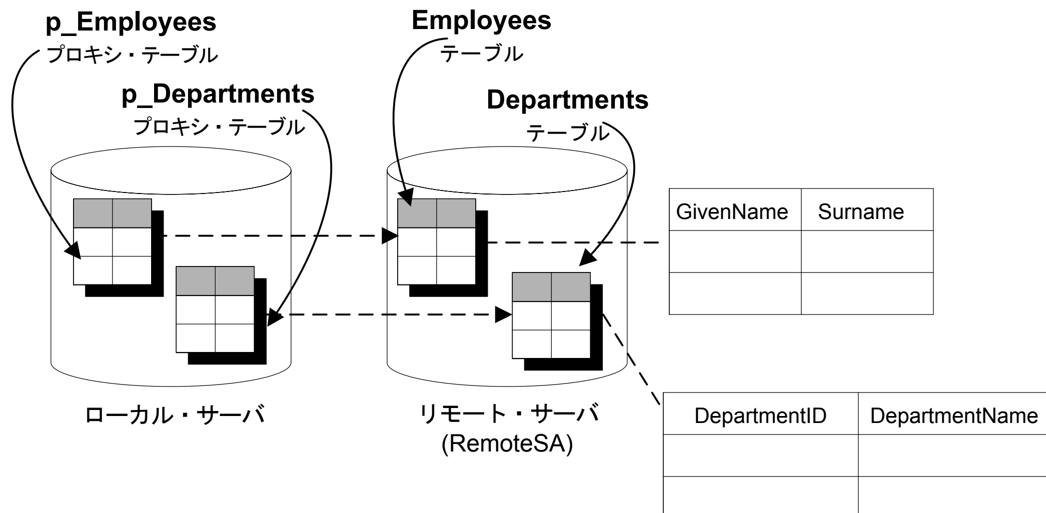
```
CALL sp_remote_columns  asetest, sysobjects, null, production;
```

詳細については、「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』を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

リモート・テーブルのジョイン

次の図は、ローカル・データベース・サーバのプロキシ・テーブルにマップされた、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 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。

♦ 2 つのリモート・テーブル間のジョインを実行するには、次の手順に従います (SQL の場合)。

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。この操作は、次のコマンド・ラインを使用して行います。

   ```
   dbeng10 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 DSN SQL Anywhere 10 Demo を参照します。

   ```
   CREATE SERVER RemoteSA
   CLASS 'saodbc'
   USING 'SQL Anywhere 10 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・サーバでデータベースに接続するときにユーザ ID とパスワードの入力が必要です。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 です。

```
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;
```

 Copyright © 2007, iAnywhere Solutions, Inc.

複数のローカル・データベースのテーブルのジョイン

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 パラメータ値」 761 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. 使用しているデータベースのいずれかに接続します。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';
```

   JDBC を使用する場合は次のようになります。

```
CREATE SERVER remote_db2
CLASS 'sajdbc'
USING 'mypc1:2638/db2';
CREATE SERVER remote_db3
CLASS 'sajdbc'
USING 'mypc1:2638/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';
```

Copyright © 2007, iAnywhere Solutions, Inc.

ネイティブ文のリモート・サーバへの送信

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。

詳細については、「FORWARD TO 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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
```

 Copyright © 2007, iAnywhere Solutions, Inc.

リモート・プロシージャ・コール (RPC) の使用

SQL Anywhere ユーザは、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。

この機能は、SQL Anywhere、Adaptive Server Enterprise、Oracle、DB2 でサポートされています。リモート・プロシージャ・コールの発行は、ローカル・プロシージャ・コールの使用と類似しています。

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、リモート関数を使用してリモート・プロシージャと関数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・プロシージャの作成

リモート・プロシージャ・コールは、Sybase Central または CREATE PROCEDURE 文を使用して発行できます。

リモート・プロシージャを作成するには、DBA 権限が必要です。

♦ リモート・プロシージャ・コールを発行するには、次の手順に従います (Sybase Central の場合)。

1. Sybase Central からホスト・データベースに接続します。
2. [リモート・サーバ] フォルダを開きます。
3. 左ウィンドウ枠で、リモート・プロシージャを作成するリモート・サーバを選択します。
4. 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。
5. [ファイル] − [新規] − [リモート・プロシージャ] を選択します。

   [リモート・プロシージャ作成] ウィザードが表示されます。
6. ウィザードの指示に従います。

♦ リモート・プロシージャ・コールを発行するには、次の手順に従います (SQL の場合)。

1. SQL Anywhere へのプロシージャを定義します。

   構文はローカル・プロシージャの定義と同じですが、SQL 文を使用してプロシージャ本体を作成するのではなく、プロシージャの実体が存在するロケーションを定義するロケーション文字列を指定する点が異なります。

   ```
   CREATE PROCEDURE remotewho()
   AT 'bostonase.master.dbo.sp_who';
   ```
2. 次のようにプロシージャを実行します。

   ```
   CALL remotewho();
   ```

詳細については、「CREATE PROCEDURE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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 uname char( 30 ) )
AT 'bostonase.master.dbo.sp_helpuser';
CALL remoteuser( 'joe' );
```

リモート・プロシージャのデータ型

次のデータ型は RPC パラメータ用のものです。

- [UNSIGNED] SMALLINT
- [UNSIGNED] INT
- [UNSIGNED] BIGINT
- TINYINT
- REAL
- DOUBLE
- CHAR
- BIT
- データ型 NUMERIC と DECIMAL は、IN パラメータには指定できますが、OUT パラメータと INOUT パラメータには指定できません。

リモート・プロシージャの削除

Sybase Central または DROP PROCEDURE 文を使用すると、リモート・プロシージャを削除できます。

リモート・プロシージャを削除するには、DBA 権限が必要です。

◆ リモート・プロシージャを削除するには、次の手順に従います (Sybase Central の場合)。

1. [リモート・サーバ] フォルダを開きます。
2. 左ウィンドウ枠でリモート・サーバを選択します。
3. 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] タブで、リモート・プロシージャを選択し、[ファイル] – [削除] を選択します。

◆ リモート・プロシージャを削除するには、次の手順に従います (SQL の場合)。

1. データベースに接続します。
2. DROP PROCEDURE 文を実行します。

Copyright © 2007, iAnywhere Solutions, Inc.

詳細については、「DROP 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

例

次の文は、remoteproc というリモート・プロシージャを削除します。

```
DROP PROCEDURE remoteproc;
```

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。これらの相違点について、次の項で説明します。

トランザクションの概要については、「トランザクションと独立性レベル」 119 ページを参照してください。

リモート・トランザクション管理の概要

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

- 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。

Copyright © 2007, iAnywhere Solutions, Inc.

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 datetime 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

前述の手順は、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。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

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から厳密にリモート・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・サーバは指定された SQL 文のすべての機能をサポートしているとはかぎらないので、SQL Anywhere 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しなければなりません。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。その結果、ユーザは SQL Anywhere がサポートする全範囲の SQL を、特定のバックエンドの機能を考慮することなく使用できます。

文の完全なパススルー

効率をあげるために、SQL Anywhere では、文の可能なかぎり多くの部分がリモート・サーバに渡されます。多くの場合、これは元々 SQL Anywhere に指定されたとおりの、完全な文になります。

SQL Anywhere は、次のような場合に完全な文を渡します。

- 文中のテーブルがどれも同じリモート・サーバに常駐する。
- リモート・サーバが文中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。

詳細については、「ALTER SERVER 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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・サーバに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、次の文中にある 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。検出される条件に合うどのローに対しても、内部テーブルとして認識されるもう 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を検出します。

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、次の文を発行します。

```
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。

 Copyright © 2007, iAnywhere Solutions, Inc.

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・データにサポートされていない機能

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・テーブルに対する 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・データに対してオープン・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。
- 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。

 Copyright © 2007, iAnywhere Solutions, Inc.

- 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、トレーシング情報がサーバ・メッセージ・ウィンドウに表示されます。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。

  cis_option オプションの詳細については、「cis_option オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

  -o サーバ・オプションの詳細については、「-o 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。

クエリに関する一般的な問題

SQL Anywhere でリモート・テーブルに対するクエリを処理できない場合は、SQL Anywhere でクエリがどのように実行されるかを理解すると役立ちます。次のようにして、クエリの実行プランだけでなく、リモート・トレーシングも表示できます。

```
SET OPTION cis_option = 7
```

リモート・トレーシングを有効にすると、トレーシング情報がサーバ・メッセージ・ウィンドウに表示されます。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。

リモート・データ・アクセスを使用しているときにクエリのデバッグで使用する cis_option オプションの詳細については、「cis_option オプション [データベース]」『SQL Anywhere サーバ - データベース管理』を参照してください。

-o サーバ・オプションの詳細については、「-o 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。

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。「トランザクションのブロックとデッドロック」 140 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

ODBC を使用したリモート・データ・アクセスの接続の管理

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。「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』を参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

第 19 章

リモート・データ・アクセスのサーバ・クラス

目次

リモート・データ・アクセスのサーバ・クラスの概要 760
JDBC ベースのサーバ・クラス 761
ODBC ベースのサーバ・クラス 764

リモート・データ・アクセスのサーバ・クラスの概要

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・クラスには、2 つのカテゴリがあります。

- JDBC ベースのサーバ・クラス
- ODBC ベースのサーバ・クラス

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、データベース管理者とプログラマがこの特性を知っておく必要があります。

この章を読むときは、サーバ・クラスのカテゴリ (JDBC ベースか ODBC ベース) 全般について述べている項と、個々のサーバ・クラスに固有の項の両方を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

JDBC ベースのサーバ・クラス

JDBC ベースのサーバ・クラスは、SQL Anywhere が内部的に Java 仮想マシンと jConnect 5.5 を使用してリモート・サーバに接続するときに使用します。JDBC ベースのサーバ・クラスを次に示します。

- **sajdbc** SQL Anywhere
- **asejdbc** Sybase SQL Server と Adaptive Server Enterprise (バージョン 10 以降)

NetWare を使用している場合、sajdbc クラスのみサポートされます。

JDBC クラスの設定上の注意

JDBC ベースのクラスで定義されたリモート・サーバにアクセスするときは、次のことに注意してください。

- 最適なパフォーマンスを得るには、iAnywhere では ODBC ベースのクラス (saodbc か aseodbc) をおすすめします。
- asejdbc サーバ・クラスまたは sajdbc サーバ・クラスを使用してアクセスするリモート・サーバは、jConnect 5.5.x ベースのクライアントを処理できるよう設定してください。jConnect 設定スクリプトは、SQL Anywhere の場合は *jcatalog.sql*、Adaptive Server Enterprise の場合は *sql_server.sql* です。これらのスクリプトを、使用するリモート・サーバに対して実行します。

サーバ・クラス sajdbc

サーバ・クラス sajdbc のサーバは、SQL Anywhere 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。

CREATE SERVER 文の USING パラメータ値

アクセスする SQL Anywhere データベースごとに、個別の CREATE SERVER を実行してください。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、次のようにローカルの SQL Anywhere を設定します。

```
CREATE SERVER TestSAdb1
CLASS 'sajdbc'
USING 'banana:2638/db1'
CREATE SERVER TestSAdb2
CLASS 'sajdbc'
USING 'banana:2638/db2'
CREATE SERVER TestSAdb3
CLASS 'sajdbc'
USING 'banana:2638/db3'
```

Copyright © 2007, iAnywhere Solutions, Inc.

/database-name 値を指定しないと、リモート接続はリモートの SQL Anywhere のデフォルト・データベースを使用します。

CREATE SERVER 文の詳細については、「CREATE SERVER 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

サーバ・クラス asejdbc

サーバ・クラス asejdbc のサーバは Sybase SQL Server と Adaptive Server Enterprise (バージョン 10 以降) のサーバです。Adaptive Server Enterprise データ・ソースの設定には、特別な要件はありません。

データ型変換：JDBC と Adaptive Server Enterprise

CREATE TABLE 文を発行するときに、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
tinyint	tinyint
smallint	smallint
int	int
integer	integer
decimal [デフォルトは p=30 s=6]	numeric(30,6)
decimal(128,128)	サポートされていない
numeric [デフォルトは p=30 s=6]	numeric(30,6)
numeric(128,128)	サポートされていない
float	real
real	real
double	float
smallmoney	numeric(10,4)
money	numeric(19,4)
date	datetime

Copyright © 2007, iAnywhere Solutions, Inc.

SQL Anywhere データ型	Adaptive Server Enterprise のデフォルト・データ型
time	datetime
timestamp	datetime
smalldatetime	datetime
datetime	datetime
char(n)	varchar(n)
character(n)	varchar(n)
varchar(n)	varchar(n)
character varying(n)	varchar(n)
long varchar	text
text	text
binary(n)	binary(n)
long binary	image
image	image
bigint	numeric(19,0)

ODBC ベースのサーバ・クラス

ODBC ベースのサーバ・クラスは以下のとおりです。

- saodbc
- aseodbc
- db2odbc
- mssodbc
- oraodbc
- odbc

ODBC 外部サーバの定義

ODBC ベースの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。これを実行するには、ODBC アドミニストレータにデータ・ソースを作成します。

データ・ソースを定義すると、CREATE SERVER 文の USING 句が ODBC データ・ソース名に一致します。

たとえば、Data Source Name も mydb2 である mydb2 という名前の 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'
```

データ・ソースの作成の詳細については、「ODBC データ・ソースの作成」『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・ソースの代わりに接続文字列を使用

データ・ソース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 データベースへの接続の例です。

```
CREATE SERVER TestSA
CLASS 'saodbc'
USING 'driver=SQL Anywhere 10;eng=TestSA;dbn=sample;links=tcpip{}'
```

この例で接続を定義する SQL Anywhere データベース・サーバは、名前が TestSA、データベースが sample、使用するプロトコルは 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」 765 ページ
- 「サーバ・クラス aseodbc」 765 ページ
- 「サーバ・クラス db2odbc」 767 ページ
- 「サーバ・クラス oraodbc」 769 ページ
- 「サーバ・クラス mssodbc」 770 ページ

Copyright © 2007, iAnywhere Solutions, Inc.

◆ 「サーバ・クラス odbc」 772 ページ

サーバ・クラス saodbc

サーバ・クラス saodbc のサーバは、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、CREATE SERVER 文を発行します。

サーバ・クラス aseodbc

サーバ・クラス aseodbc のサーバは Sybase SQL Server と Adaptive Server Enterprise (バージョン 10 以降) のデータベース・サーバです。クラスが aseodbc のリモートの Adaptive Server Enterprise サーバに接続するには、SQL Anywhere 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。しかし、パフォーマンスは asejdbc クラスよりも優れています。

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。
- [Configuration Manager] の [User Data Source] を、次の属性で設定します。
    - **[一般] 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。
    - **[詳細] タブ**　[Application Using Threads] オプションと [Enable Quoted Identifiers] オプションを選択します。
    - **[接続] 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。
    - **[パフォーマンス] 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。Sybase ではこの値を 100 に設定することをおすすめします。

      [Select Method] を **0-Cursor** に設定します。

Copyright © 2007, iAnywhere Solutions, Inc.

[Packet Size] をできるだけ大きな値に設定します。iAnywhere ではこの値を -1 に設定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 文を発行するときに、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
Tinyint	tinyint
Smallint	smallint
Int	int
Integer	integer
decimal [デフォルトは p=30 s=6]	numeric(30,6)
decimal(128,128)	サポートされていない
numeric [デフォルトは p=30 s=6]	numeric(30,6)
numeric(128,128)	サポートされていない
Float	real
Real	real
Double	float
Smallmoney	numeric(10,4)
Money	numeric(19,4)
Date	datetime
Time	datetime
Timestamp	datetime
Smalldatetime	datetime
Datetime	datetime
char(n)	varchar(n)

Copyright © 2007, iAnywhere Solutions, Inc.

SQL Anywhere データ型	Adaptive Server Enterprise のデフォルト・データ型
Character(n)	varchar(n)
varchar(n)	varchar(n)
Character varying(n)	varchar(n)
long varchar	text
Text	text
binary(n)	binary(n)
long binary	image
Image	image
Bigint	numeric(20,0)

サーバ・クラス db2odbc

サーバ・クラス db2odbc のサーバは、IBM DB2 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 構成の設定を行い、テストを実行してください。SQL Anywhere には、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 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：DB2

CREATE TABLE 文を発行するときに、SQL Anywhere は、データ型を対応する DB2 のデータ型に自動的に変換します。次の表に、SQL Anywhere から DB2 へのデータ型変換を示します。

SQL Anywhere データ型	DB2 のデフォルト・データ型
Bit	smallint
Tinyint	smallint
Smallint	smallint

SQL Anywhere データ型	DB2 のデフォルト・データ型
Int	int
Integer	int
Bigint	decimal(20,0)
char(1–254)	varchar(n)
char(255–4000)	varchar(n)
char(4001–32767)	long varchar
Character(1–254)	varchar(n)
Character(255–4000)	varchar(n)
Character(4001–32767)	long varchar
varchar(1–4000)	varchar(n)
varchar(4001–32767)	long varchar
Character varying(1–4000)	varchar(n)
Character varying(4001–32767)	long varchar
long varchar	long varchar
text	long varchar
binary(1–4000)	bit データには varchar
binary(4001–32767)	bit データには long varchar
long binary	bit データには long varchar
image	bit データには long varchar
decimal [デフォルトは p=30 s=6]	decimal(30,6)
numeric [デフォルトは p=30 s=6]	decimal(30,6)
decimal(128, 128)	サポートされていない
numeric(128, 128)	サポートされていない
real	real
float	float
double	float

Copyright © 2007, iAnywhere Solutions, Inc.

SQL Anywhere データ型	DB2 のデフォルト・データ型
smallmoney	decimal(10,4)
money	decimal(19,4)
date	date
time	time
smalldatetime	timestamp
datetime	timestamp
timestamp	timestamp

サーバ・クラス oraodbc

サーバ・クラス oraodbc のサーバは、Oracle バージョン 8.0 以降です。

注意

- iAnywhere は、バージョン 8.0.03 の Oracle 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 構成の設定を行い、テストを実行してください。
- 以下は、myora という Oracle サーバの CREATE EXISTING TABLE 文の例です。

```
CREATE EXISTING TABLE employees
AT 'myora.database.owner.employees'
```

- Oracle ODBC ドライバの制限により、システム・テーブルの CREATE EXISTING TABLE を発行することはできません。テーブルまたはカラムが見つからないことを示すメッセージが返されます。

データ型変換：Oracle

CREATE TABLE 文を発行するときに、SQL Anywhere は、データ型を対応する Oracle のデータ型に自動的に変換します。次の表に、SQL Anywhere から Oracle へのデータ型変換を示します。

SQL Anywhere データ型	Oracle のデータ型
bit	number(1,0)
tinyint	number(3,0)
smallint	number(5,0)
int	number(11,0)
bigint	number(20,0)

SQL Anywhere データ型	Oracle のデータ型
decimal(prec, scale)	number(prec, scale)
numeric(prec, scale)	number(prec, scale)
float	float
real	real
smallmoney	numeric(13,4)
money	number(19,4)
date	date
time	date
timestamp	date
smalldatetime	date
datetime	date
char(n)	n > 255 の場合は long、それ以外は varchar(n)
varchar(n)	n > 2000 の場合は long、それ以外は varchar(n)
long varchar	long または clob
binary(n)	n > 255 の場合は long raw、それ以外は raw(n)
varbinary(n)	n > 255 の場合は long raw、それ以外は raw(n)
long binary	long raw

サーバ・クラス mssodbc

サーバ・クラス mssodbc のサーバは、Microsoft SQLServer version 6.5 (Service Pack 4) です。

注意

- iAnywhere は、バージョン 3.60.0319 の Microsoft SQL Server ODBC ドライバ (MDAC 2.0 リリースに含まれる) の使用を確認しています。この製品の説明に従って、ODBC 構成の設定を行い、テストを実行してください。
- mymssql という Microsoft SQLServer サーバの CREATE EXISTING TABLE 文の例を次に示します。

```
CREATE EXISTING TABLE accounts,
AT 'mymssql.database.owner.accounts'
```


Copyright © 2007, iAnywhere Solutions, Inc.

データ型変換：Microsoft SQL Server

CREATE TABLE 文を発行するときに、SQL Anywhere は、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。次の表に、SQL Anywhere から Microsoft SQLServer へのデータ型変換を示します。

SQL Anywhere データ型	Microsoft SQLServer のデフォルト・データ型
bit	bit
tinyint	tinyint
smallint	smallint
int	int
bigint	numeric(20,0)
decimal [デフォルトは p=30 s=6]	decimal(prec, scale)
numeric [デフォルトは p=30 s=6]	numeric(prec, scale)
float	(prec) の場合は float(prec)、それ以外は float
real	real
smallmoney	smallmoney
money	money
date	datetime
time	datetime
timestamp	datetime
smalldatetime	datetime
datetime	datetime
char(n)	length > 255 の場合は text、それ以外は varchar (length)
character(n)	char(n)
varchar(n)	length > 255 の場合は text、それ以外は varchar (length)
long varchar	text
binary(n)	length > 255 の場合は image、それ以外は binary (length)
long binary	image

SQL Anywhere データ型	Microsoft SQLServer のデフォルト・データ型
double	float
uniqueidentifierstr	uniqueidentifier

Microsoft SQL Server の uniqueidentifier カラムは、SQL Anywhere の uniqueidentifierstr カラムにマップされます。マップされた結果の文字列は、STRTOUUID 関数を使用してバイナリ UUID 値に変換できます。「STRTOUUID 関数 [文字列]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。ODBC バージョン 2.0 準拠レベル 1 以降に準拠する ODBC ドライバであれば、どれでも使用できます。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)」 772 ページ
- 「Microsoft Access (Microsoft 3.51.171300)」 773 ページ
- 「Microsoft FoxPro (Microsoft 3.51.171300)」 773 ページ
- 「Lotus Notes SQL 2.0」 774 ページ

最新バージョンの Microsoft ODBC ドライバは、Microsoft Data Access Components (MDAC) とともに、http://www.microsoft.com/downloads/search.aspx?displaylang=ja からダウンロードできます。MDAC 2.0 には、以下に示すバージョンの Microsoft ドライバが含まれています。

以降の項では、これらのデータ・ソースへのアクセスについての注意点を説明します。

Microsoft Excel (Microsoft 3.51.171300)

Excel で、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でシートにマップされます。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 文を発行する場合には、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。

この例では、excel という名前の ODBC データ・ソースが作成されています。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'
```


Copyright © 2007, iAnywhere Solutions, Inc.

CREATE EXISTING 文を使用して、既存のワークシートを SQL Anywhere にインポートできます。ここでは、スプレッド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、マップ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。

Microsoft Access (Microsoft 3.51.171300)

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 つにマップします。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。

Microsoft FoxPro (Microsoft 3.51.171300)

FoxPro テーブルは 1 つの FoxPro データベース・ファイル (*.dbc*) にまとめて格納されるか、それぞれのテーブルが各自の*.dbf* ファイルに格納されます。*.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'
```

Copyright © 2007, iAnywhere Solutions, Inc.
```

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 2.0

このドライバは Lotus Web サイトから取得できます。Notes データがリレーショナル・テーブルにどのようにマップ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プできます。

Address サンプル・ファイル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。

- NotesSQL ドライバを使用して、ODBC データ・ソースを作成します。データベースは names サンプル・ファイル *c:¥notes¥data¥names.nsf* になります。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。
- SQL Anywhere にサーバを作成します。

```
CREATE SERVER names
CLASS 'odbc'
USING 'my_notes_dsn'
```

- Person フォームを SQL Anywhere テーブルにマップします。

```
CREATE EXISTING TABLE Person
AT 'names...Person'
```

- テーブルを問い合わせます。

```
SELECT * FROM Person
```

パスワードのプロンプトを表示しない

Lotus Notes は、ODBC API を介したユーザ名とパスワードの送信をサポートしません。パスワードで保護された ID を使用して Lotus Notes にアクセスしようとすると、SQL Anywhere が稼働しているコンピュータに、パスワードの入力を求めるプロンプトが表示されます。マルチユーザのサーバ環境では、このような動作が起こらないようにしてください。

パスワードの入力を求めるプロンプトを表示せずに Lotus Notes にアクセスするには、パスワードで保護されていない ID を使用してください。ID が作成されたときに Domino 管理者がパスワードを要求しなかった場合、パスワード保護をクリア ([ファイル] − [ツール] − [ユーザ ID] − [パスワードの解除] の順に選択) することによって、それを解除できます。パスワードの使用が必要とされていた場合は、クリアすることはできません。

Copyright © 2007, iAnywhere Solutions, Inc.

パート VII. ストアド・プロシージャとトリガ

パート VII では、SQL ストアド・プロシージャとトリガを使用して、データベースに論理を構築する方法について説明します。データベースに格納された論理は、自動的にすべてのアプリケーションで利用できるようになるので、一貫性、パフォーマンス、セキュリティ面で利点があります。また、あらゆる種類の論理をデバッグできる強力なツールである SQL Anywhere デバッガの使用方法についても説明します。

第 20 章

プロシージャ、トリガ、バッチの使用

目次

プロシージャとトリガの概要 778
プロシージャとトリガの利点 779
プロシージャの概要 780
ユーザ定義関数の概要 787
トリガの概要 790
バッチの概要 799
制御文 802
プロシージャとトリガの構造 805
プロシージャから返される結果 808
プロシージャとトリガでのカーソルの使用 813
プロシージャとトリガでのエラーと警告 817
プロシージャでの EXECUTE IMMEDIATE 文の使用 825
プロシージャとトリガでのトランザクションとセーブポイント 826
プロシージャを作成するときのヒント 827
プロシージャ、トリガ、イベント、バッチで使用できる文 829
プロシージャからの外部ライブラリの呼び出し 830

プロシージャとトリガの概要

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。

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し、他のプロシージャを呼び出すか、またはトリガを起動できます。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。

注意
さまざまな用途で、サーバ側 JDBC は、SQL ストアド・プロシージャよりも柔軟にデータベースにロジックを構築します。JDBC の詳細については、「SQL Anywhere JDBC API」『SQL Anywhere サーバ - プログラミング』を参照してください。

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。「プロシージャ、関数、トリガ、イベントのデバッグ」 837 ページを参照してください。

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。「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」 237 ページを参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

プロシージャとトリガの利点

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、多くの利点があります。

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

プロシージャとトリガは、テーブルのデータへのユーザのアクセスを制限し、ユーザが直接検査や修正をできないようにすることによって、セキュリティを提供しています。

たとえば、トリガは関連するテーブルの所有者のパーミッションにより実行されますが、テーブルのローを挿入、更新、または削除するパーミッションを持つユーザであれば、トリガを起動できます。同様に、プロシージャ (ユーザ定義関数を含む) はプロシージャの所有者のパーミッションにより実行されますが、パーミッションを与えられたユーザはプロシージャを呼び出すことができます。つまり、プロシージャとトリガのパーミッションはそれらを起動するユーザが持つパーミッションと異なる可能性があります。

プロシージャの概要

プロシージャを使用するには、次のことを行う方法を理解する必要があります。

- プロシージャの作成
- データベース・アプリケーションからの呼び出し
- プロシージャの削除
- パーミッションの制御

この項では、プロシージャを使用するための上記の方法とプロシージャの他の使用方法について説明します。

プロシージャの作成

SQL Anywhere では、新しいプロシージャを作成するためのツールが用意されています。

Sybase Central では、ウィザードを使用して必要な情報を指定できます。[プロシージャ作成] ウィザードには、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。

また、Interactive SQL を使用し、CREATE PROCEDURE 文を実行してプロシージャを作成することもできます。ただし、RESOURCE 権限を所有していなければなりません。

◆ 新しいプロシージャを作成するには、次の手順に従います (Sybase Central の場合)。

1. DBA 権限または RESOURCE 権限を所有するユーザとしてデータベースに接続します。
2. データベースの [プロシージャとファンクション] フォルダを開きます。
3. [ファイル] – [新規] – [プロシージャ] を選択します。

   [プロシージャ作成] ウィザードが表示されます。
4. ウィザードの指示に従います。
5. ウィザードの終了後、右ウィンドウ枠の [SQL] タブでプロシージャ・コードを完了できます。

   新しいプロシージャは、[プロシージャとファンクション] フォルダに表示されます。

接続の詳細については、「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。

例

次に、SQL Anywhere のサンプル・データベースの Departments テーブルに対して INSERT を実行して、新しい部署を作成するプロシージャ NewDepartment の簡単な例を示します。

```
CREATE PROCEDURE NewDepartment(
  IN id INT,
```

 Copyright © 2007, iAnywhere Solutions, Inc.

```
   IN name CHAR(35),
   IN head_id INT )
BEGIN
   INSERT
   INTO Departments ( DepartmentID,
      DepartmentName, DepartmentHeadID )
   VALUES ( id, name, head_id );
END;
```

プロシージャの本体は複合文です。複合文は BEGIN で始まり、END で終わります。NewDepartment では、複合文は BEGIN 文と END 文に挟まれた 1 つの INSERT 文です。

プロシージャのパラメータは IN、OUT、または INOUT のいずれかです。デフォルトでは、パラメータは INOUT パラメータです。NewDepartment プロシージャのパラメータは、プロシージャによって変更されないため、すべてが IN パラメータです。パラメータを使用して呼び出し元に値を返さない場合は、パラメータを IN に設定してください。

詳細については、「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』、「複合文の使用」 803 ページを参照してください。

プロシージャの変更

Sybase Central または Interactive SQL を使って既存のプロシージャを変更できます。DBA 権限を所有しているか、プロシージャの所有者でなければなりません。

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。

Interactive SQL では、ALTER PROCEDURE 文を実行して既存のプロシージャを修正できます。プロシージャを作成した CREATE PROCEDURE 文と同じ構文で、この文に新しいプロシージャ全体を含めます。

データベース・オブジェクトのプロパティの変更については、「データベース・オブジェクトのプロパティの設定」 36 ページを参照してください。

プロシージャのパーミッションの付与または取り消しの詳細については、「プロシージャに対するパーミッションの付与」『SQL Anywhere サーバ - データベース管理』と「ユーザ・パーミッションの取り消し」『SQL Anywhere サーバ - データベース管理』を参照してください。

♦ **プロシージャのコードを変更するには、次の手順に従います (Sybase Central の場合)。**

1. [プロシージャとファンクション] フォルダを開きます。
2. 対象のプロシージャを選択します。次のいずれかを行います。
   - 右ウィンドウ枠の [SQL] タブで、コードを直接編集する。
   - プロシージャを右クリックし、ポップアップ・メニューで [新しいウィンドウで編集] を選択して、コードを別のウィンドウで編集する。

Copyright © 2007, iAnywhere Solutions, Inc.

ヒント
プロシージャ間でコードをコピーする場合は、プロシージャごとに別のウィンドウを開きます。

プロシージャの変換については、「Sybase Central を使用するストアド・プロシージャの変換」 634 ページを参照してください。

詳細については、「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」 780 ページを参照してください。

プロシージャの呼び出し

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。

詳細については、「CALL 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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 );
```

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 テーブルを表示できます。

プロシージャの EXECUTE パーミッションを付与されたすべてのユーザは、Departments テーブルのパーミッションがなくても、NewDepartment プロシージャを呼び出すことができます。

EXECUTE パーミッションの詳細については、「GRANT 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

結果セットを返すプロシージャを呼び出すもう 1 つの方法は、クエリ内で呼び出す方法です。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。

```
SELECT t.ID, t.QuantityOrdered AS q
FROM ShowCustomerProducts( 149 ) t;
```

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。

Sybase Central におけるプロシージャのコピー

Sybase Central では、データベース間でプロシージャをコピーできます。Sybase Central の左ウィンドウ枠でプロシージャを選択し、コピー先の接続済みデータベースの [プロシージャとファンクション] フォルダへドラッグします。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。

Copyright © 2007, iAnywhere Solutions, Inc.

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あることに注意してください。他のプロシージャ・プロパティ (パーミッションなど) はコピーされません。新しい名前を付ける場合、プロシージャは同じデータベースにコピーできます。

プロシージャの削除

作成したプロシージャは、明示的に削除されるまでデータベースに存在します。プロシージャの所有者か DBA 権限を所有するユーザだけがプロシージャをデータベースから削除できます。

♦ **プロシージャを削除するには、次の手順に従います (Sybase Central の場合)。**

1. DBA ユーザとして、またはプロシージャの所有者と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] フォルダを開きます。
3. 対象のプロシージャを右クリックし、ポップアップ・メニューで [削除] を選択します。

♦ **プロシージャを削除するには、次の手順に従います (SQL の場合)。**

1. DBA 権限を所有するユーザとして、またはプロシージャの所有者としてデータベースに接続します。
2. DROP PROCEDURE 文を実行します。

例

次の文は、NewDepartment プロシージャをデータベースから削除します。

```
DROP PROCEDURE NewDepartment;
```

詳細については、「DROP 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

プロシージャの結果をパラメータとして返す

プロシージャは次のいずれかの方法で、呼び出された環境に結果を返します。

- 個別の値を OUT または INOUT パラメータとして返す。
- 結果セットを返す。
- RETURN 文を使って 1 つの結果を返す。

ここでは、パラメータとしてプロシージャから結果を返す方法について説明します。

次に示す例は、SQL Anywhere のサンプル・データベースの従業員の平均給与を OUT パラメータとして返します。

```
CREATE PROCEDURE AverageSalary( OUT avgsal NUMERIC(20,3) )
BEGIN
  SELECT AVG( Salary )
```

Copyright © 2007, iAnywhere Solutions, Inc.

```
  INTO avgsal
  FROM Employees;
END;
```

♦ **このプロシージャを実行し、その出力を表示するには、次の手順に従います (SQL の場合)。**

1. Interactive SQL を使用して、DBA として SQL Anywhere サンプル・データベースに接続します。接続の詳細については、「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。
2. [SQL 文] ウィンドウ枠に上記のプロシージャ・コードを入力します。
3. プロシージャの結果を格納する変数を作成します。ここでは、結果は小数点以下 3 桁の数字なので、変数を次のように作成します。

   ```
   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);
   ```

   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。

5. 次の文を実行して変数の値を検査します。

   ```
   SELECT Average;
   ```

   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。

プロシージャの結果を結果セットとして返す

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。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。

```
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;
```

Interactive SQL から呼び出された場合、RESULT 句での名前はクエリの結果と一致し、表示される結果のカラムの見出しに使われます。

Interactive SQL からこのプロシージャをテストするには、部署名を指定して CALL 文を使います。InteractiveSQL では、結果が [結果] ウィンドウ枠の [結果] タブに表示されます。

例

次を入力して、研究開発部 (部署 ID 100) の従業員の給与を表示します。

```
CALL SalaryList( 100 );
```

 Copyright © 2007, iAnywhere Solutions, Inc.

Employee ID	Salary
102	45700.000
105	62000.000
160	57490.000
243	72995.000
…	…

[オプション] ダイアログの [結果] タブでこのオプションを有効にした場合にのみ、Interactive SQL では複数の結果セットを返すことができます。各結果セットは [結果] ウィンドウ枠の個別のタブに表示されます。

詳細については、「プロシージャから複数の結果セットを返す」 811 ページを参照してください。

プロシージャに関する詳細情報

テンポラリ・プロシージャの作成

テンポラリ・プロシージャを作成するには、CREATE PROCEDURE 文を拡張した CREATE TEMPORARY PROCEDURE 文を使用してください。テンポラリ・プロシージャは、データベースに一時的に格納されます。テンポラリ・プロシージャは、接続の終了時または削除が指定された時点で削除されます。

テンポラリ・プロシージャの作成の詳細については、「CREATE PROCEDURE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

リモート・プロシージャの作成

リモート・プロシージャを作成するには、1 つ以上のリモート・サーバが必要です。「Sybase Central を使用したリモート・サーバの作成」 730 ページを参照してください。

♦ 新しいリモート・プロシージャを作成するには、次の手順に従います (Sybase Central の場合)。

1. DBA 権限を所有するユーザとしてデータベースに接続します。
2. データベースの [プロシージャとファンクション] フォルダを開きます。
3. [ファイル] – [新規] – [リモート・プロシージャ] を選択します。

   [リモート・プロシージャ作成] ウィザードが表示されます。
4. ウィザードの指示に従います。
5. ウィザードの終了後、右ウィンドウ枠の [SQL] タブでプロシージャ・コードを完了できます。

Copyright © 2007, iAnywhere Solutions, Inc.

新しいリモート・プロシージャは、[プロシージャとファンクション] フォルダに表示されます。

 Copyright © 2007, iAnywhere Solutions, Inc.

ユーザ定義関数の概要

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。ここでは、ユーザ定義関数の作成、使用、削除について説明します。

ユーザ定義関数の作成

CREATE FUNCTION 文を使用してユーザ定義関数を作成します。この文を実行するには、RESOURCE 権限が必要です。

次に示す例は、2 つの文字列を、間にスペースを入れた形で結合し、姓と名前から氏名を作成するのに使います。

```
CREATE FUNCTION FullName( FirstName CHAR(30),
  LastName CHAR(30) )
RETURNS CHAR(61)
BEGIN
  DECLARE name CHAR(61);
  SET name = FirstName || ' ' || LastName;
  RETURN ( name );
END;
```

詳細については、「CREATE FUNCTION 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

CREATE FUNCTION の構文は、CREATE PROCEDURE 文の構文と若干異なります。次に相違点を示します。

- 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。

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が、適切なパーミッションが必要です。

次に示す例は、姓と名前の入った 2 つのカラムから氏名を表示します。

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

Full Name
Fran Whitney

Copyright © 2007, iAnywhere Solutions, Inc.

Full Name
Matthew Cobb
Philip Chin
…

次の例は、文中に提供された姓と名前から氏名を表示します。

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

Full Name
Jane Smith

関数に対する EXECUTE パーミッションを付与されたユーザは FullName 関数を使用できます。

例

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。

Customers テーブルには、アメリカの顧客の中にカナダの顧客が混ざっています。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';
```

カーソルと例外の宣言については後で説明します。

Copyright © 2007, iAnywhere Solutions, Inc.

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、性能が非常に低くなることがあります。たとえば、テーブルに 100,000 のローがあり、その中の 10,000 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 10,000 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、100,000 回呼び出されます。

ユーザ定義関数の削除

作成したユーザ定義関数は、いずれかのユーザが明示的に削除するまでデータベースに保持されます。ユーザ定義関数の所有者または DBA 権限を所有するユーザのみが、データベースから関数を削除できます。

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;
```

ユーザ定義関数を実行するためのパーミッション

ユーザ定義関数の所有権はその関数を作成したユーザに所属し、そのユーザはパーミッションなしに実行できます。ユーザ定義関数の所有者は、GRANT EXECUTE コマンドを使って他のユーザにパーミッションを与えることができます。

たとえば、FullName 関数の作成者が別のユーザに FullName 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;
```

パーミッションを取り消す文は、次のようになります。

```
REVOKE EXECUTE ON Nationality FROM BobS;
```

関数のユーザ・パーミッションの管理の詳細については、「プロシージャに対するパーミッションの付与」『SQL Anywhere サーバ - データベース管理』を参照してください。

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「**冪等**」として扱われます。冪等関数は、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」 566 ページを参照してください。

トリガの概要

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。「データ整合性の確保」 95 ページと「CREATE TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

トリガを実行するためのパーミッション
トリガは、関連するテーブルまたはビューの所有者のパーミッション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 (実体化ビュー (Materialized View) を除く)。「INSTEAD OF トリガ」 796 ページを参照してください。

トリガを定義する構文の詳細については、「CREATE TRIGGER 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

動作	説明
INSERT	トリガの関連するテーブルの中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中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中のローが更新されたときに、トリガが起動される。

Copyright © 2007, iAnywhere Solutions, Inc.

動作	説明
UPDATE OF *column-list*	トリガの関連するテーブル中のローが、*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。「トリガ・オペレーション条件」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

トリガ実行のタイミングは柔軟に設定できるので、トリガが参照整合性、たとえばカスケード更新または削除が実行されたかどうかによって実行するかどうかを決めるような場合に、特に有効です。

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。「アトミックな複合文」 803 ページを参照してください。

トリガの作成

Sybase Central または Interactive SQL を使ってトリガを作成します。Sybase Central では、ウィザードを使用して必要な情報を指定できます。Interactive SQL では、CREATE TRIGGER 文を使用できます。いずれのツールを使用する場合でも、トリガを作成するには DBA または RESOURCE 権限が必要です。また、トリガと関連するテーブルに対して ALTER パーミッションが必要です。

トリガの本文は複合文、つまり BEGIN と END 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。

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。

♦ 指定されたテーブルに対するトリガを作成するには、次の手順に従います (Sybase Central の場合)。

1. 対象のテーブルの [トリガ] フォルダを開きます。
2. [ファイル] – [新規] – [トリガ] を選択します。

   [トリガ作成] ウィザードが表示されます。
3. ウィザードの指示に従います。
4. ウィザードの終了後、右ウィンドウ枠の [SQL] タブでトリガ・コードを完了できます。

♦ 指定されたテーブルに対するトリガを作成するには、次の手順に従います (SQL の場合)。

1. データベースに接続します。
2. CREATE TRIGGER 文を実行します。

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ます。」というエラー・メッセージが表示されます。

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。

エラーを感知すると、トリガ文とトリガ以前のデータの変更が取り消されます。

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。

Copyright © 2007, iAnywhere Solutions, Inc.

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、宣言参照整合性を使用したり、検査制約を使用したりして、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

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

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。

詳細については、「CREATE TRIGGER 文」『SQL Anywhere サーバ - SQL リファレンス』と「複合文の使用」803 ページを参照してください。

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。

トリガの変更

Sybase Central または Interactive SQL を使って既存のトリガを変更できます。トリガを定義するテーブルの所有者であるか、DBA 権限を所有しているか、テーブルに対する ALTER パーミッションと RESOURCE 権限を所有してなければなりません。

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。

または、ALTER TRIGGER 文を使用して既存のトリガを修正できます。トリガを作成した CREATE TRIGGER 文と同じ構文で、この文に新しいトリガ全体を含めます。

データベース・オブジェクトのプロパティの変更については、「データベース・オブジェクトのプロパティの設定」 36 ページを参照してください。

♦ トリガのコードを変更するには、次の手順に従います (Sybase Central の場合)。

1. [トリガ] フォルダを開きます。
2. 対象のトリガを選択します。次のいずれかを行います。

    ♦ 右ウィンドウ枠の [SQL] タブで、コードを直接編集する。

 Copyright © 2007, iAnywhere Solutions, Inc.

- 右ウィンドウ枠でトリガを右クリックし、ポップアップ・メニューで [新しいウィンドウで編集] を選択して、コードを別のウィンドウで編集する。

> **ヒント**
> トリガ間でコードをコピーする場合は、トリガごとに別のウィンドウを開きます。

トリガの変換については、「Sybase Central を使用するストアド・プロシージャの変換」 634 ページを参照してください。

♦ トリガのコードを変更するには、次の手順に従います (SQL の場合)。

1. データベースに接続します。
2. ALTER TRIGGER 文を実行します。この文に新しいトリガ全体を含めます。

詳細については、「ALTER TRIGGER 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

トリガの削除

作成したトリガは、明示的に削除されるまでデータベースに存在します。トリガを削除するには、トリガの関連するテーブルの ALTER パーミッションが必要となります。

♦ トリガを削除するには、次の手順に従います (Sybase Central の場合)。

1. [トリガ] フォルダを開きます。
2. 対象のトリガを右クリックし、ポップアップ・メニューで [削除] を選択します。

♦ トリガを削除するには、次の手順に従います (SQL の場合)。

1. データベースに接続します。
2. DROP TRIGGER 文を実行します。

例

次に、トリガ mytrigger をデータベースから削除する文を示します。

```
DROP TRIGGER mytrigger;
```

詳細については、「DROP 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

トリガを実行するためのパーミッション

ユーザはトリガを実行できないので、トリガを実行するパーミッションを与えることはできません。データベースに対するアクションに対応して SQL Anywhere がトリガを起動します。トリガが実行される場合は、トリガに関連するパーミッションがあり、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パーミッション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パーミッションや、トリガを作成したユーザのパーミッションではありません。

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、テーブル作成者のグループ・メンバシップを使います。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 の所有者であるグループの (直接または間接的に) メンバ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、「**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。

また、user_1 はトリガに指定された操作を実行するためのパーミッションを持っていなければなりません。

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。

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。

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が省略され、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。

 Copyright © 2007, iAnywhere Solutions, Inc.

- 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 トリガは実体化ビュー (Materialized View) には定義できません。実体化ビュー (Materialized View) 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。「CREATE TRIGGER 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
- INSTEAD OF トリガが再帰するかどうかは、トリガのターゲットがベース・テーブルであるか、ビューであるかで決まります。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対して 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、本来は更新できないビューに対して 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 V1 は更新不可能になっているので、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
```

Copyright © 2007, iAnywhere Solutions, Inc.

```
        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るようになります。V1 に対して INSERT 文や UPDATE 文を実行する他の INSTEAD OF トリガも定義できます。

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 と UPDATE も同様です。前の例の続きで、別のビューを作成するとします。

```
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、V1 が本来は更新できないため、V2 も更新できないからです。しかし、V1 に INSTEAD OF DELETE トリガが定義されていれば、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。

ネストされているビューに定義されている INSTEAD OF トリガがある場合は、そのことを覚えておくことが重要です。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ります。意図する動作を明示的にするには、このような場合に、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 トリガを定義することをおすすめします。この例では、次のトリガを V2 に定義し、DELETE 文の意図した動作を実装できます。

```
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 DELETE の動作は変わりません。

 Copyright © 2007, iAnywhere Solutions, Inc.

バッチの概要

簡単な Transact-SQL 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、「変数が見つかりません」というエラーが発生します。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
```

 Copyright © 2007, iAnywhere Solutions, Inc.

```
    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';
```

プロシージャとトリガ内で使用される多くの文は、バッチ内でも使用できます。バッチ内では制御文 (CASE、IF、LOOP など) を複合文 (BEGIN と END) を含めて使用できます。複合文の中には変数の宣言、例外、テンポラリ・テーブル、カーソルを含めることができます。

制御文

プロシージャまたはトリガの本文、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たくさんあります。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 文」『SQL Anywhere サーバ - SQL リファレンス』を参照。	BEGIN [ATOMIC] *Statement-list* END

条件実行：IF
「IF 文」『SQL Anywhere サーバ - SQL リファレンス』を参照。
	IF *condition* THEN
	  Statement-list
	ELSEIF *condition* THEN
	  Statement-list
	ELSE
	  Statement-list
	END IF

条件実行：CASE
「CASE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照。
	CASE *expression*
	WHEN *value* THEN
	  Statement-list
	WHEN *value* THEN
	  Statement-list
	ELSE
	  Statement-list
	END CASE

繰り返し：WHILE、LOOP 「LOOP 文」『SQL Anywhere サーバ - SQL リファレンス』を参照。	WHILE *condition* LOOP *Statement-list* END LOOP

繰り返し：FOR カーソル・ループ
「FOR 文」『SQL Anywhere サーバ - SQL リファレンス』を参照。
	FOR *loop-name*
	  AS *cursor-name* CURSOR FOR
	  select-statement
	DO
	  Statement-list
	END FOR

中断：LEAVE 「LEAVE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照。	LEAVE *label*
CALL 「CALL 文」『SQL Anywhere サーバ - SQL リファレンス』を参照。	CALL *procname*(*arg*, ...)

Copyright © 2007, iAnywhere Solutions, Inc.

複合文の使用

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。

複合文の詳細については、「BEGIN 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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」な文は、完全に実行されるか、まったく実行されません。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。

Copyright © 2007, iAnywhere Solutions, Inc.

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。

COMMIT 文と ROLLBACK 文、いくつかの ROLLBACK TO SAVEPOINT 文は、アトミックな複合文内で使用できません (「プロシージャとトリガでのトランザクションとセーブポイント」 826 ページを参照)。

アトミックな複合文中のいくつかの文のみが実行されるときもあります。複合文の中で例外ハンドラがエラーを処理するときに、このようなことが起こります。

詳細については、「プロシージャとトリガでの例外ハンドラの使用」 821 ページを参照してください。

 Copyright © 2007, iAnywhere Solutions, Inc.

プロシージャとトリガの構造

プロシージャとトリガの本体は、「複合文の使用」 803 ページで説明したように複合文から構成されています。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。

プロシージャ・パラメータの宣言

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 (「SQL データ型」 『SQL Anywhere サーバ - SQL リファレンス』を参照) 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;
```

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();
```

Copyright © 2007, iAnywhere Solutions, Inc.

パラメータをプロシージャに渡す

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 とお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;
```

次のように最初のパラメータだけを指定して、SampleProcedure を呼び出すこともできます。

```
CALL SampleProcedure( V1 );
```

この場合、パラメータの *var2* と *var3* にはデフォルト値が使われます。

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 );
```

パラメータを関数に渡す

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、「ユーザ定義関数の作成」 787 ページで定義した、従業員の氏名を取り出す FullName 関数を使用した例を次に示します。

♦ 全従業員の名前をリストするには、次の手順に従います。

- 次のように入力します。

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

Name
Fran Whitney

Copyright © 2007, iAnywhere Solutions, Inc.

Name
Matthew Cobb
Philip Chin
Julie Jordan
…

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。

プロシージャから返される結果

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。

プロシージャから結果を返す簡単な例については、「プロシージャの概要」 780 ページを参照してください。

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。次に RETURN 文を示します。

```
RETURN expression
```

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 myproc();
```

結果をプロシージャのパラメータとして返す

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
- INTO 句を持つ SELECT 文

SET 文の使用

次に示すプロシージャは、SET 文を使って割り当てた OUT パラメータに値を返します。

```
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;
```


Copyright © 2007, iAnywhere Solutions, Inc.

シングルロー SELECT 文の使用

シングルロー・クエリは 1 つのローをデータベースから取り出します。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、サーバは SELECT 文の結果を取り出して、変数に入れます。クエリの結果が複数のローを含んでいれば、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。プロシージャから複数のローを返す方法については、「プロシージャから結果セットを返す」 810 ページを参照してください。

クエリの結果、ローが取り出されなかった場合は、「**ローが見つかりません**」という警告が表示されます。

次にシングルローの SELECT 文の結果をパラメータに返すプロシージャの例を示します。

♦ **指定した顧客によって行われた発注の数を返すには、次の手順に従います。**

・ 次のように入力します。

```
CREATE PROCEDURE OrderCount( IN customer_ID INT,
                    OUT Orders INT )
BEGIN
  SELECT COUNT(SalesOrders.ID)
    INTO Orders
  FROM Customers
    KEY LEFT OUTER JOIN SalesOrders
  WHERE Customers.ID = customer_ID;
END;
```

このプロシージャは、Interactive SQL で次の文を使ってテストできます。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;
```

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。

Copyright © 2007, iAnywhere Solutions, Inc.

プロシージャから結果セットを返す

結果セットを使用して、プロシージャは複数のローの結果を呼び出し元の環境に返すことができます。

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。注文のなかった顧客はリストに含まれません。

```
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;
```

◆ 次のように入力します。

```
CALL ListCustomerValue ()
```

Company	Value
The Hat Company	5016
The Igloo	3564
The Ultimate	3348
North Land Trading	3144
Molly's	2808
…	…

注意

◆ RESULT 句のリスト中の変数の数は、SELECT 文のリスト中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、コマンド・デリミタは付きません。

◆ SELECT 文のリスト中の変数の名前は、RESULT 句のリスト中の変数の名前と一致する必要はありません。

◆ このプロシージャをテストするとき、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。[オプション] ダイアログの [結果] タブの [複数の結果セットを表示] オプションを設定して、複数の結果セットを表示するように Interactive SQL を設定できます。

Copyright © 2007, iAnywhere Solutions, Inc.

- ビューから作成されていない場合、プロシージャ結果セットを変更できます。プロシージャの結果を修正する場合、プロシージャを呼び出すユーザは基本のテーブルに対して適切なパーミッションを持っている必要があります。この点が、通常のプロシージャの実行のパーミッションとは異なります。通常は、プロシージャの*所有者*がテーブルに対するパーミッションを持っていなければなりません。

  Interactive SQL で結果セットを修正する方法については、「Interactive SQL での結果セットの編集」『SQL Anywhere サーバ - データベース管理』を参照してください。

- ストアド・プロシージャまたはユーザ定義関数が結果セットを返す場合、出力パラメータを設定したり戻り値を返したりすることはできません。

プロシージャから複数の結果セットを返す

Interactive SQL が複数の結果セットを返すように設定するには、[オプション] ダイアログの [結果] タブのこのオプションをオンに設定する必要があります。デフォルトでは、このオプションはオフに設定されます。設定を変更する場合、変更された設定は新しく接続を行うときに有効になります (新しいウィンドウなど)。

◆ 複数の結果セット機能を有効にするには、次の手順に従います。

1. Interactive SQL から、[ツール] − [オプション] を選択します。
2. 表示される [オプション] ダイアログで、[結果] 領域をクリックします。
3. [複数の結果セットを表示] チェック・ボックスをオンにします。

このオプションを有効にすると、プロシージャは呼び出しを行った環境に複数の結果セットを返すことができます。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。

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;
```

Interactive SQL でこのプロシージャをテストし、複数の結果セットを表示するには、[SQL 文] ウィンドウ枠で次の文を入力します。

```
CALL ListPeople ();
```

Copyright © 2007, iAnywhere Solutions, Inc.

プロシージャから変数結果セットを返す

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。

  DESCRIBE 文の詳細については、「DESCRIBE 文 [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **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。

Copyright © 2007, iAnywhere Solutions, Inc.

プロシージャとトリガでのカーソルの使用

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。

カーソル管理の概要

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。

カーソルの位置設定の詳細については、「カーソル位置」『SQL Anywhere サーバ - プログラミング』を参照してください。

プロシージャの SELECT 文でのカーソルの使用

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、「プロシージャから結果セットを返す」 810 ページで説明する 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 )
```

Copyright © 2007, iAnywhere Solutions, Inc.

```
      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
    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 "row not found" 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。

  例外の詳細については、「プロシージャとトリガでのエラーと警告」 817 ページを参照してください。

- 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。

Copyright © 2007, iAnywhere Solutions, Inc.

この TopCompanyValue プロシージャでの LOOP 文は標準的なフォームで、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;
```

ストアド・プロシージャ内のカーソルの更新

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 UPDATE 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
```

Copyright © 2007, iAnywhere Solutions, Inc.

```
    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 );
```


Copyright © 2007, iAnywhere Solutions, Inc.

プロシージャとトリガでのエラーと警告

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。SQLCODE と SQLSTATE のエラーと警告値とそれらの意味の詳細については、SQL Anywhere 10 - エラー・メッセージ『SQL Anywhere 10 - エラー・メッセージ』を参照してください。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。「プロシージャの SELECT 文でのカーソルの使用」 813 ページの項に示した 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。

プロシージャとトリガでの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「プロシージャとトリガでの例外ハンドラの使用」 821 ページで説明する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。詳細については、「プロシージャとトリガでのデフォルトの警告処理」 820 ページを参照してください。

エラーを処理するには、特に指定しないかぎり次の 2 とお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。詳細については、「on_tsql_error オプション [互換性]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

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。トリガの場合は、トリガを起動した操作も取り消され、アプリケーションにエラーが返されます。

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;
```

注意

- 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。

OuterProc プロシージャを実行するには、次の文を入力してください。

```
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 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();
```

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。

 Copyright © 2007, iAnywhere Solutions, Inc.

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。詳細については、「on_tsql_error オプション [互換性]」 『SQL Anywhere サーバ - データベース管理』を参照してください。

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

この機能を、次の例で説明します。

[SQL 文] ウィンドウ枠に次のコマンドを入力して、必ずプロシージャ InnerProc と OuterProc を削除してから、チュートリアルを続けてください。

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;
```

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。

```
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
```

Copyright © 2007, iAnywhere Solutions, Inc.

```
    EXCEPTION FOR SQLSTATE '52003';
  MESSAGE 'Hello from InnerProc.' TO CLIENT;
  SIGNAL column_not_found;
  MESSAGE 'SQLSTATE set to ',
  SQLSTATE, ' in InnerProc.' TO CLIENT;
END;
```

OuterProc プロシージャを実行するには、次の文を入力してください。

```
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。

プロシージャとトリガでの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。

[SQL 文] ウィンドウ枠に次のコマンドを入力して、必ずプロシージャ InnerProc と OuterProc を削除してから、チュートリアルを続けてください。

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;
```

次に示す例は、デフォルトの警告処理を示します。例文は、「プロシージャとトリガでのデフォルトのエラー処理」 817 ページで使用したプロシージャを基にしています。ここでは SIGNAL 文はエラーではなく、「**ローが見つかりません**」という警告を表示します。

```
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
```


Copyright © 2007, iAnywhere Solutions, Inc.

```
    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;
```

OuterProc プロシージャを実行するには、次の文を入力してください。

```
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。

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。

プロシージャとトリガでの例外ハンドラの使用

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 で定義します (「複合文の使用」 803 ページを参照)。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。

[SQL 文] ウィンドウ枠に次のコマンドを入力して、必ずプロシージャ InnerProc と OuterProc を削除してから、チュートリアルを続けてください。

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;
```

次に示す例では、「プロシージャとトリガでのデフォルトのエラー処理」 817 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。ここでは InnerProc プロシージャ中の「**カラムが見つかりません**」エラーを処理するために、コードが追加されています。

```
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 'Line following SIGNAL.' TO CLIENT;
   EXCEPTION
      WHEN column_not_found THEN
      MESSAGE 'Column not found handling.' TO CLIENT;
        WHEN OTHERS THEN
      RESIGNAL ;
END;
```

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。

この例では、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。

OuterProc プロシージャを実行するには、次の文を入力してください。

```
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.

注意

- InnerProc の SIGNAL 文に続く行ではなく、EXCEPTION ハンドラが実行されます。
- 「**カラムが見つかりません**」エラーのため、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 (エラーは起こらなかったことを示します)。
- 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。
- 「**カラムが見つかりません**」例外に対するエラー処理コードが単に RESIGNAL 文のとき、制御は OuterProc に戻され、SQLSTATE は値 52003 に設定されたままです。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。

Copyright © 2007, iAnywhere Solutions, Inc.

例外処理とアトミックな複合文

例外が複合文内で処理されるとき、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。これはアトミックな複合文でも同じです。アトミックな複合文の中でエラーが発生し、明示的に処理されると、複合文のいくつかの文は実行されます。

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。

[SQL 文] ウィンドウ枠に次のコマンドを入力して、必ずプロシージャ InnerProc と OuterProc を削除してから、チュートリアルを続けてください。

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;
```

次のプロシージャの例は、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。プロシージャは、「プロシージャとトリガでのデフォルトのエラー処理」 817 ページの例に使用したものに基づいています。

```
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;
```

InnerPro プロシージャを実行するには、次の文を入力してください。

```
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、「Column not found handling」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。

Copyright © 2007, iAnywhere Solutions, Inc.

内部複合文で「**カラムが見つかりません**」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。

次の文を実行して、InnerProc プロシージャを削除します。

```
DROP PROCEDURE InnerProc;
```

 Copyright © 2007, iAnywhere Solutions, Inc.

プロシージャでの EXECUTE IMMEDIATE 文の使用

EXECUTE IMMEDIATE 文を使うと、文字列 (引用符で囲む) と変数を使ってプロシージャ中に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。次に例を示します。

```
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;
```

このプロシージャを呼び出すには、次の文を入力してください。

```
CALL DynamicResult(
   'table_id,table_name',
   'SYSTAB',
   'table_id <= 10');
```

table_id	table_name
1	ISYSTAB
2	ISYSTABCOL
3	ISYSIDX
...	...

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。

EXECUTE IMMEDIATE 文の詳細については、「EXECUTE IMMEDIATE 文 [SP]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

プロシージャとトリガ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中の SQL 文は現在のトランザクションの一部です (「トランザクションと独立性レベル」 119 ページを参照)。1 つのトランザクション中で複数のプロシージャを呼び出すことや、1 つのプロシージャ中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 (「アトミックな複合文」 803 ページを参照)。トリガはアトミックな文である INSERT、UPDATE、DELETE によって起動されることに注意してください。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中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 (「トランザクション内のセーブポイント」 123 ページを参照) 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。

Copyright © 2007, iAnywhere Solutions, Inc.

プロシージャを作成するときのヒント

この項では、プロシージャを作成するためのヒントをいくつか説明します。

コマンド・デリミタを変更する必要があるかどうかをチェックする

Interactive SQL または Sybase Central でプロシージャを作成するときには、コマンド・デリミタの変更は必要ありません。他のブラウズ・ツールを使う場合には、コマンド・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、コマンド・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。

コマンド・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、コマンド・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

プロシージャ中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グループ・メンバシップを使い、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 の所有者であるグループの (直接または間接的に) メンバ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* の日付フォーマットを使用します。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。

日付と時刻の詳細については、「日付と時刻データ型」 『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。「レッスン 2 : ストアド・プロシージャのデバッグ」 841 ページを参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

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 (CREATE TABLE、ALTER TABLE などのデータ定義文を含む) を使用できますが、次の文は使用できません。

- ALTER DATABASE (syntax 3)
- CONNECT
- CREATE DATABASE
- CREATE DECRYPTED FILE
- CREATE ENCRYPTED FILE
- DISCONNECT
- DROP CONNECTION
- DROP DATABASE
- FORWARD TO
- INPUT、OUTPUT などの Interactive SQL コマンド
- PREPARE TO COMMIT
- STOP ENGINE

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。「プロシージャとトリガでのトランザクションとセーブポイント」 826 ページを参照してください。

詳細については、「SQL 文」『SQL Anywhere サーバ - SQL リファレンス』に示す各 SQL 文の使用方法を参照してください。

バッチで SELECT 文を使用する

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。

次に有効なバッチの例を示します。

```
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。

プロシージャからの外部ライブラリの呼び出し

ストアド・プロシージャまたはユーザ定義関数から外部ライブラリの関数を呼び出すことができます。Windows オペレーティング・システムでは DLL、NetWare では NLM、UNIX では共有オブジェクトの関数を呼び出すことができます。Windows CE では、外部関数を呼び出すことができません。

この項では、プロシージャ中で外部ライブラリ呼び出しを使用する方法について説明します。サンプルの外部ストアド・プロシージャと、プロシージャに含まれる DLL を構築するために必要なファイルは、フォルダ *samples-dir¥SQLAnywhere¥ExternalProcedures* に格納されています。*samples-dir* 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。

> **警告**
> プロシージャから呼び出された外部ライブラリは、サーバのメモリを共有します。プロシージャから呼び出した外部ライブラリがメモリ処理のエラーを含んでいると、サーバそのものがクラッシュしたり、データベースが損傷したりする可能性があります。運用データベースに配備する前にライブラリをテストする必要があります。

この項で説明する API は、以前の API の代わりに使用します。古い API は推奨されなくなりました。バージョン 7.0.x 以前の古い API を使用して作成されたライブラリもサポートされますが、新しく開発する場合は、最新の API を使用してください。

SQL Anywhere は、MAPI 電子メールの送信などでこの機能を使用するシステム・プロシージャのセットを含みます。

システム・プロシージャの詳細については、「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。

外部呼び出しを使ったプロシージャと関数の作成

ここでは、外部呼び出しを使ったプロシージャと関数の例を示します。

> **DBA 権限**
> 外部ライブラリを参照するプロシージャまたは関数を作成するには、DBA 権限が必要です。その他のプロシージャまたは関数の作成には RESOURCE 権限が必要ですが、外部プロシージャまたは関数の作成には DBA 権限が必須です。

構文

DDL ライブラリ *library.dll* の関数 *function_name* を呼び出すプロシージャは、次のように作成できます。

```
CREATE PROCEDURE dll_proc( parameter-list )
EXTERNAL NAME 'function_name@library.dll'
```


Copyright © 2007, iAnywhere Solutions, Inc.

プロシージャから外部 DLL を呼び出す場合、そのプロシージャは他のタスクを実行することはできません。プロシージャは単に DLL の入れ物 (ラッパ) になってしまいます。

上記のプロシージャと同じ機能を果たすユーザ定義関数を次に示します。

```
CREATE FUNCTION dll_func ( parameter-list )
RETURNS data-type
EXTERNAL NAME 'function_name@library.dll'
```

これらの文中の ***function_name*** はダイナミック・リンク・ライブラリ (DLL) のエクスポートされる関数名で、*library.dll* はライブラリの名前です。***parameter-list*** の引数の型と順序は、ライブラリ関数によって定義されている引数と対応しなければなりません。ライブラリ関数は、「外部関数のプロトタイプ」 832 ページで説明した API を使ってプロシージャ引数にアクセスします。

外部関数から返された値は、プロシージャから、呼び出しを行った環境に返されます。

他の文は使用不可

外部関数を参照するプロシージャは、その他の文を含むことはできません。このプロシージャの目的は、関数の引数を取ること、関数を呼び出すこと、関数から返ってきた値と引数を呼び出し元の環境に返すことです。IN、INOUT、OUT パラメータは、通常のプロシージャの場合と同じように使用できます。入力された値は外部関数に渡され、関数によって変更を受けたパラメータは OUT または INOUT パラメータを通して呼び出しを行った環境に返されます。

オペレーティング・システムに依存する呼び出し

あるオペレーティング・システムではある関数を呼び出し、もう 1 つのオペレーティング・システムでは (おそらく同じ機能の) 別の関数を呼び出せます。この場合の構文は、関数名にオペレーティング・システム名をプレフィクスとして付けます。オペレーティング・システムの識別子には、UNIX と NetWare のいずれかを指定してください。

関数のリストにサーバのオペレーティング・システムのエントリがなく、オペレーティング・システムが指定されていないエントリを含む場合、データベース・サーバはそのエントリのこの関数を呼び出します。

NetWare での呼び出しは他のオペレーティング・システムと少し異なります。NetWare では記号はすべてグローバルに知られており、エクスポートされる関数名のような記号は、システム上のすべての NLM に対してユニークでなければなりません。したがって NLM がすでにロードされているかぎり、NLM 名は呼び出し中には必要ありません。NLM がすでにロードされているかどうかに関係なく、常にライブラリ名を使用することをおすすめします。NLM がまだロードされていない場合は、ライブラリ名を指定する*必要があります*。ファイル拡張子 *.nlm* はオプションです。

CREATE PROCEDURE 文の構文の詳細については、「CREATE PROCEDURE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

CREATE FUNCTION 文の構文の詳細については、「CREATE FUNCTION 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

外部関数のプロトタイプ

ここでは、外部ライブラリの関数のための API について説明します。

API は、SQL Anywhere インストール・ディレクトリの *h* サブディレクトリのヘッダ・ファイル *extfnapi.h,* によって定義されます。このヘッダ・ファイルは、外部関数のプロトタイプのプラットフォームに依存する機能を処理します。この API は、外部ライブラリの関数のための以前の API に代わるものです。

API バージョンの宣言

外部ライブラリが古い API を使って作成されていないことをデータベース・サーバに通知するには、次のように関数を指定します。

uint32 extfn_use_new_api( )

関数は符号なし 32 ビット整数値を返します。戻り値は、*extfnapi.h* で定義した EXTFN_API_VERSION の API バージョン番号です。戻り値が 0 の場合は、古い API が使用されていることを示します。

関数が DLL によってエクスポートされない場合、データベース・サーバは古い API が使用中であるとみなします。UNIX プラットフォームと 64 ビット Windows を含むすべての 64 ビット・プラットフォームでは、新しい API を使用してください。

NetWare の場合、外部プロシージャが新しい API を使用して作成されていることをデータベース・サーバに通知するには、NLM で extfn_use_new_api と呼ばれる関数か、*name*_use_new_api と呼ばれる関数のいずれかをエクスポートする必要があります。*name* は NLM の名前です。たとえば、*external.nlm* という名前の NLM は、関数 external_use_new_api をエクスポートします。

*name*_use_new_api をエクスポートすることで、一度に 2 つ以上の外部 NLM が使用される場合に、エクスポート名の競合を防げます。NLM から *name*_use_new_api と呼ばれる関数をエクスポートする場合は、CREATE PROCEDURE 文または CREATE FUNCTION 文に NLM 名を含める必要があります。

関数のプロトタイプ

関数名は、CREATE PROCEDURE または CREATE FUNCTION 文に参照された関数名と一致しなければなりません。関数宣言は、次のようにします。

void *function-name***(an_extfn_api ****api*, **void ****argument-handle* **)**

この関数は、戻り値はなく、引数を渡すために使った構造体と SQL プロシージャによって指定された引数へのハンドルを引数として使用します。

an_extfn_api 構造体の形式は次のとおりです。

```
typedef struct an_extfn_api {
    short (SQL_CALLBACK *get_value)(
                void *          arg_handle,
                a_sql_uint32    arg_num,
                an_extfn_value *value
                );
    short (SQL_CALLBACK *get_piece)(
                void *          arg_handle,
                a_sql_uint32    arg_num,
```


Copyright © 2007, iAnywhere Solutions, Inc.

```
            an_extfn_value *value,
            a_sql_uint32    offset
            );
    short (SQL_CALLBACK *set_value)(
            void *          arg_handle,
            a_sql_uint32    arg_num,
            an_extfn_value *value
            short           append
            );
    void (SQL_CALLBACK *set_cancel)(
            void *  arg_handle,
            void *  cancel_handle
            );
} an_extfn_api;
```

an_extfn_value 構造体の形式は次のとおりです。

```
typedef struct an_extfn_value {
    void *          data;
    a_sql_uint32    piece_len;
    union {
        a_sql_uint32 total_len;
        a_sql_uint32 remain_len;
    } len;
    a_sql_data_type     type;
} an_extfn_value;
```

注意

OUT パラメータで get_value を呼び出すと、引数のデータ型が返され、データは NULL として返されます。

任意の引数の get_piece 関数は、同じ引数の get_value 関数の直後にのみ呼び出すことができます。

NULL を返すには、an_extfn_value 構造体において data を NULL に設定します。

set_value の append フィールドは、指定されたデータを既存のデータと置き換えるか (false)、または既存のデータに追加するか (true) を指定します。append=FALSE を指定して set_value を呼び出してから、同じ引数に対して append=TRUE を指定して呼び出してください。固定長データ型の場合、append フィールドは無視されます。

ヘッダ・ファイル自体が追加のノートを含みます。

次の表に、an_extfn_api に定義されている関数が false を返す条件を示します。

関数	true の場合は 0 を返し、それ以外の場合は 1 を返す条件
get_value()	♦ arg_num が無効な場合。たとえば、arg_num が ext_fn の引数の数より大きい場合。 ♦ 外部関数呼び出しが正常に初期化される前に呼び出される場合。

関数	true の場合は 0 を返し、それ以外の場合は 1 を返す条件
get_piece()	◆ arg_num が無効な場合。たとえば、arg_num が最後の get_value に対応していない場合。 ◆ オフセットが arg_num 引数の値の合計長より大きい場合。 ◆ 外部関数呼び出しが正常に初期化される前に呼び出される場合。
set_value()	◆ arg_num が無効な場合。たとえば、arg_num が ext_fn の引数の数より大きい場合。 ◆ arg_num 引数が入力専用の場合。 ◆ 指定された値の型が arg_num 引数の型と一致しない場合。 ◆ 外部関数呼び出しが正常に初期化される前に呼び出される場合。

a_sql_data_type フィールドに入力できる値の詳細については、「Embedded SQL のデータ型」『SQL Anywhere サーバ - プログラミング』を参照してください。

外部関数にパラメータを渡す場合の詳細については、「パラメータを外部関数に渡す」 834 ページを参照してください。

取り消し処理の実現

取り消されることが予想される外部関数は、set_cancel API 関数を呼び出してデータベース・サーバに通知します。外部動作を取り消すため特殊関数をエクスポートします。この関数は、次の形式にします。

void an_extfn_cancel(void * ***cancel_handle*** **)**

DLL がこの関数をエクスポートできない場合、データベース・サーバは DLL の関数のユーザ割り込みを無視します。この関数では、*cancel_handle* は上記の an_extfn_api 構造体に示した *set_cancel* API 関数による外部関数への呼び出しごとに取り消される関数によってデータベース・サーバへ提供されるポインタです。

パラメータを外部関数に渡す

データ型

外部ライブラリに渡されるのは、次に示す SQL データ型です。

SQL データ型	C データ型
CHAR	指定した長さの文字データ

 Copyright © 2007, iAnywhere Solutions, Inc.

SQL データ型	C データ型
VARCHAR	指定した長さの文字データ
LONG VARCHAR	指定した長さの文字データ
BINARY	指定した長さのバイナリ・データ
LONG BINARY	指定した長さの文字データ
TINYINT	1 バイト整数
[UNSIGNED] SMALLINT	[符号なし] 2 バイト整数
[UNSIGNED] INT	[符号なし] 4 バイト整数
[UNSIGNED] BIGINT	[符号なし] 8 バイト整数
VARBINARY	指定した長さのバイナリ・データ
REAL	単精度浮動小数点数
DOUBLE	倍精度浮動小数点数

日付または時刻データ型は使用できず、真数値データ型も使用できません。

INOUT または OUT パラメータに値を指定するには、set_value API 関数を使用します。IN と INOUT パラメータを読み取るには、get_value API 関数を使用します。

NULL を渡す

すべての引数に有効な値として NULL を渡すことができます。外部ライブラリの関数は、すべてのデータ型の戻り値として NULL を渡すことができます。

戻り値

外部関数の戻り値を設定するには、arg_num パラメータに 0 を指定して set_value 関数を呼び出します。arg_num=0 と指定して set_value を呼び出さなかった場合、関数の結果は NULL になります。

プロシージャ、関数、トリガ、ビューの内容を隠す

場合によっては、プロシージャ、関数、トリガ、ビューに含まれるロジックを公開せずに、アプリケーションとデータベースを配布できます。追加のセキュリティ対策として、ALTER PROCEDURE 文、ALTER FUNCTION 文、ALTER TRIGGER 文、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。

SET HIDDEN 句は、関連オブジェクトの内容にスクランブルをかけて読み取れないようにします。これらのオブジェクトは引き続き使用できます。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。

Copyright © 2007, iAnywhere Solutions, Inc.

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。

すでに隠されているオブジェクトに対して、前述のいずれかの文を実行しても効果はありません。

特定の型のすべてのオブジェクトのテキストを隠すには、次のようなループを使用できます。

```
BEGIN
   FOR hide_lp as hide_cr cursor FOR
      SELECT proc_name, user_name
      FROM SYS.SYSPROCEDURE p, SYS.SYSUSERPERM 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』

 Copyright © 2007, iAnywhere Solutions, Inc.

第 21 章

プロシージャ、関数、トリガ、イベントのデバッグ

目次

SQL Anywhere のデバッガの概要 838
チュートリアル：デバッガの使用開始 840
ブレークポイントの活用 845
変数の編集 847
接続の活用 848

SQL Anywhere のデバッガの概要

SQL Anywhere のデバッガは、SQL のストアド・プロシージャ、トリガ、イベント・ハンドラ、ユーザ定義関数の開発時に使用できます。

SQL Anywhere のデバッガを使用すると、次に示すような多くの作業を実行できます。

- **プロシージャとトリガのデバッグ** SQL ストアド・プロシージャやトリガをデバッグできます。
- **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。この章でのストアド・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適用できます。
- **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** 実行が SQL プロシージャのブレークポイントで停止したときに、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て、クエリ実行プランを参照したりできます。

ヒント
デフォルトでは、SOAP 接続は 60 秒でタイムアウトします。SOAP 関数やプロシージャをデバッグしようとするときは、接続がタイムアウトしないように -xs http(kto=0) を指定することができます。「-xs 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。

Copyright © 2007, iAnywhere Solutions, Inc.

Java デバッガの稼働条件

デバッガを使用するには、DBA 権限か SA_DEBUG グループのパーミッションが必要です。このグループは、データベースが作成されるときにすべてのデータベースに追加されます。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。

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、データベースに接続する方法、デバッガを起動する方法、簡単なストアド・プロシージャをデバッグする方法について説明します。

このチュートリアルでは、SQL Anywhere のサンプル・データベース *demo.db* を使用します。このファイルは、SQL Anywhere サンプル・ディレクトリ *samples-dir* にあります。*samples-dir* の詳細については、「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。

レッスン 1：データベースへの接続とデバッガの起動

♦ **デバッガを起動するには、次の手順に従います。**

1. [スタート] – [プログラム] – [SQL Anywhere 10] – [Sybase Central] を選択して、Sybase Central を起動します。
2. データベースに接続します。

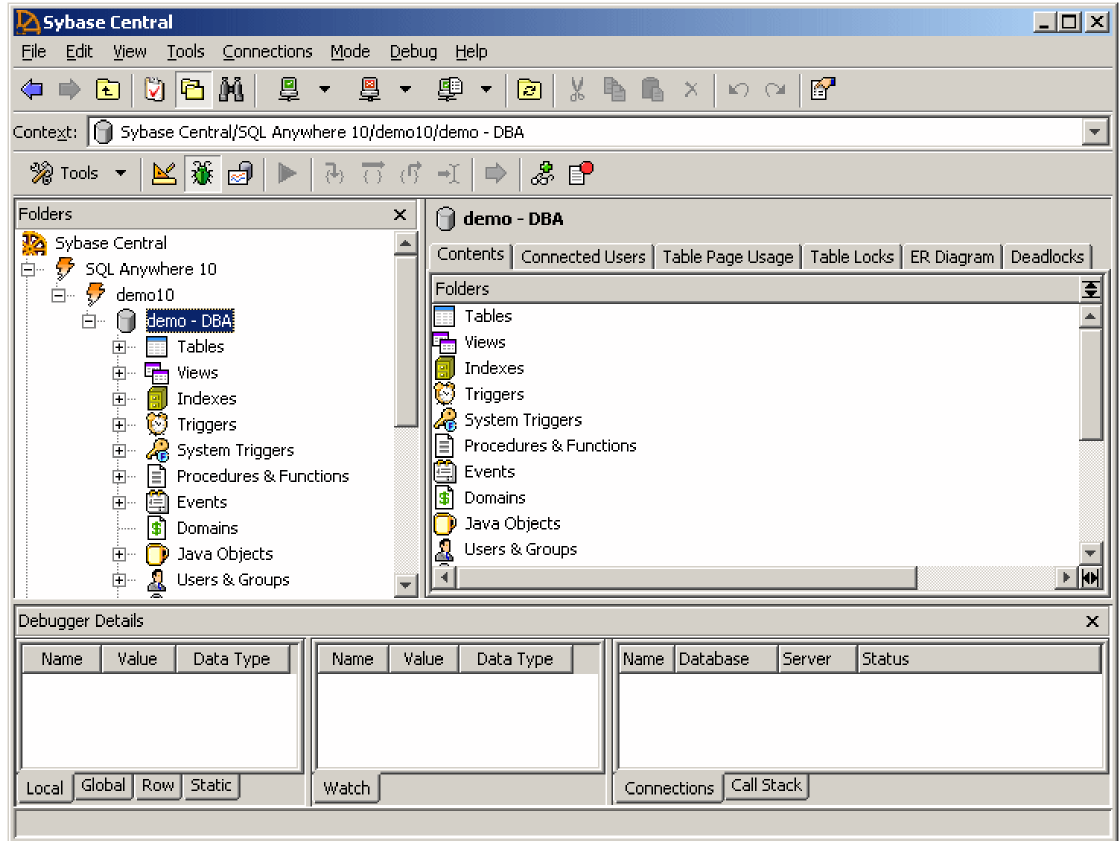
   a. [接続] メニューから [SQL Anywhere 10 に接続] を選択します。

      [接続] ダイアログが表示されます。

   b. [ODBC データ・ソース名] フィールドに「**SQL Anywhere 10 Demo**」と入力し、[OK] をクリックします。
3. [モード] – [デバッグ] を選択します。

   Sybase Central は、設計モード、デバッグ・モード、またはアプリケーション・プロファイリング・モードで使用できます。デバッグ・モードで動作中、デバッガ・ブレークポイントはアクティブになります。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。

Copyright © 2007, iAnywhere Solutions, Inc.



レッスン 2 : ストアド・プロシージャのデバッグ

このレッスンでは、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。この目的のために、SQL Anywhere サンプル・データベースの debugger_tutorial に意図的にエラーを含めます。

debugger_tutorial 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さなければなりません。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)。このプロシージャには、意図的にバグが含まれています。このチュートリアルでは、バグを診断し、修正してください。

debugger_tutorial プロシージャの実行

debugger_tutorial プロシージャは、発注額の最も多い会社と、その製品注文総額を含む結果セットを返さなければなりません。バグがあるため、この結果セットは返されません。このレッスンでは、ストアド・プロシージャを実行します。

Copyright © 2007, iAnywhere Solutions, Inc.

♦ **debugger_tutorial ストアド・プロシージャを実行するには、次の手順に従います。**

1. 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] フォルダを開きます。
2. プロシージャを実行します。

   debugger_tutorial プロシージャを右クリックし、ポップアップ・メニューで [Interactive SQL から実行] を選択します。

   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)

   これは、明らかに正しい結果ではありません。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。
3. Interactive SQL を閉じます。

バグの診断

プロシージャのバグを診断するために、プロシージャに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とともに変化する変数の値を監視します。

ここでは、プロシージャ内の最初の実行可能な文にブレークポイントを設定します。

♦ **バグを診断するには、次の手順に従います。**

1. Sybase Central がデバッグ・モードになっていることを確認します。
2. プロシージャ内の最初の実行可能な文にブレークポイントを設定します。

   その文には次のテキストが含まれます。

   ```
   OPEN cursor_this_customer;
   ```

   この行の左側にあるグレーの縦線をクリックして、ブレークポイントを設定します。ブレークポイントは、赤い円で表示されます。[F9] キーを押してブレークポイントを設定する方法もあります。
3. プロシージャを再度実行します。

   a. 左ウィンドウ枠で、debug_tutorial プロシージャを右クリックし、ポップアップ・メニューで [Interactive SQL から実行] を選択します。

   b. Sybase Central からの接続をデバッグするかどうかをたずねるメッセージ・ボックスが表示されます。[はい] をクリックします。

      プロシージャの実行がブレークポイントで停止します。[ソース・コード] ウィンドウ内の黄色の矢印は、現在位置を示します。そこがブレークポイントです。
4. 変数を検査します。

Copyright © 2007, iAnywhere Solutions, Inc.

[デバッガの詳細] ウィンドウ枠の [ローカル] タブに、プロシージャ内のローカル変数が現在の値とデータ型とともにリストされます。変数 top_company、top_value、this_value、this_company は、いずれも初期化されていないため NULL です。

5. [F11] キーを押してストアド・プロシージャをステップ・スルーします。ストアド・プロシージャの行をステップ・スルーするにつれて、変数の値が変化します。

6. 次の行に達したら、コードのステップ・スルーを停止します。

```
IF this_value > top_value THEN
```

IF 文を見ると、this_value は 1452 ですが、top_value は NULL のままです。

7. もう 1 つ先の文にステップします。

[F11] キーをもう一度押して、分岐を確認します。黄色の矢印が、次のテキストを含む、ループの開始点のラベル文に戻ります。

```
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。

診断内容を確認し、バグを修正する

top_value が初期化されていないことが原因であるという仮説を、プロシージャ・コードを変更せずに、デバッガでテストできます。

♦ 仮説をテストするには、次の手順に従います。

1. top_value に値を設定します。

[ローカル] ウィンドウで、変数 top_value の [値] フィールドをクリックし、値 3000 を入力します。

2. ループを再度ステップ・スルーします。

[F11] キーを押して命令を IF 文までステップ・スルーし、this_value と top_value の値を確認します。ループを何度かステップ・スルーすると、this_value の値が 3000 より大きくなります。

3. ブレークポイントを無効にし、プロシージャを実行します。

   a. ブレークポイントをクリックして、グレーにします (無効になります)。

   b. [F5] キーを押してプロシージャの実行を終了します。

   [Interactive SQL] ウィンドウが再び表示されます。正しい結果が表示されています。

top_company	top_value
Chadwicks	8076

仮説が正しいことが確認されました。top_value が初期化されていないことが原因でした。

♦ **バグを修正するには、次の手順に従います。**

1. [モード] – [設計] を選択します。
2. 次のテキストが含まれる行の直後に、

   ```
   OPEN cursor_this_customer;
   ```

   top_value 変数を初期化する新しい行を作成します。

   ```
   SET top_value = 0;
   ```

3. [Ctrl+S] キーを押して、変更したプロシージャを保存します。
4. 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。

このレッスンは終了です。[Interactive SQL] ウィンドウが開いている場合は閉じます。

 Copyright © 2007, iAnywhere Solutions, Inc.

ブレークポイントの活用

この項では、ブレークポイントを使用して、デバッガがソース・コードの実行をいつ中断するかを制御する方法について説明します。

ブレークポイントの設定

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。

♦ **ブレークポイントを設定するには、次の手順に従います。**

1. デバッグ・モードで実行している Sybase Central で、ブレークポイントを設定するコードを表示します。
2. 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。
3. F9 を押してブレークポイントを設定します。

   コード行の左側に赤い円が表示されます。

   コード行のすぐ左にあるグレーの部分をダブルクリックして、ブレークポイントを挿入または削除することもできます。

♦ **ブレークポイントを設定するには、次の手順に従います ([デバッグ] メニューの場合)。**

1. デバッグ・モードで実行している Sybase Central で、[デバッグ] – [ブレークポイント] を選択します。

   [ブレークポイント] ダイアログが表示されます。
2. [新規] をクリックします。

   [新しいブレークポイント] ダイアログが表示されます。
3. ドロップダウン・リストからプロシージャ名を選択し、任意で [条件] と ［カウント] に値を入力します。

   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。たとえば、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'
   ```

   カウントとは、ブレークポイントのヒット数で、その回数を超えると実行が停止されます。0 を指定した場合、常にブレークポイントによって実行が停止されます。
4. [OK] をクリックしてブレークポイントを設定します。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。

ブレークポイントの有効化と無効化

Sybase Central の右ウィンドウ枠、または [ブレークポイント] ダイアログからブレークポイントのステータスを変更できます。

♦ ブレークポイントのステータスを変更するには、次の手順に従います。

1. ステータスを変更するブレークポイントを含むプロシージャのソース・コードを表示します。
2. 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。行のステータスは、アクティブなブレークポイントから無効化されたブレークポイントに切り替わります。

♦ ブレークポイントのステータスを変更するには、次の手順に従います ([ブレークポイント] ダイアログの場合)。

1. [ブレークポイント] ダイアログを開きます。
2. ブレークポイントを編集します。

   また、ブレークポイントを選択して [Delete] キーを押すとブレークポイントを削除できます。

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すことができます。プロシージャとトリガについては、SQL 探索条件にします。

たとえば、特定の接続のみにブレークポイントを適用するには、ブレークポイントに条件を設定します。

♦ ブレークポイントに条件または回数を設定するには、次の手順に従います。

1. [デバッグ] – [ブレークポイント] を選択します。

   [ブレークポイント] ダイアログが表示されます。
2. 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。

   [ブレークポイントの編集] ダイアログが表示されます。
3. [条件] フィールドに条件を入力します。

   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。

Copyright © 2007, iAnywhere Solutions, Inc.

変数の編集

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。

ローカル変数

♦ **変数の値を監視するには、次の手順に従います。**

1. 検査する変数があるプロシージャにブレークポイントを設定します。

   ブレークポイントの設定については、「ブレークポイントの設定」 845 ページを参照してください。

2. [デバッガの詳細] ウィンドウ枠の [ローカル] タブをクリックします。

3. プロシージャを実行します。[ローカル] タブに変数が値とともに表示されます。

その他の変数

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、[デバッガの詳細] ウィンドウ枠の [グローバル] タブに表示されます。

グローバル変数のリストについては、「グローバル変数」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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、[デバッガの詳細] ウィンドウ枠の [ロー] タブに表示されます。

トリガの詳細については、「トリガの概要」 790 ページを参照してください。

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。

呼び出しスタック

ネストされたプロシージャをデバッグするときに、呼び出しの順番を検査すると便利なことがあります。[コール・スタック] タブで、プロシージャのリストを参照できます。

♦ **呼び出しスタックを表示するには、次の手順に従います。**

1. 検査する変数があるプロシージャにブレークポイントを設定します。

2. コードをブレークポイントまで実行します。

3. [デバッガの詳細] ウィンドウ枠の [コール・スタック] タブをクリックします。

   [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。

Copyright © 2007, iAnywhere Solutions, Inc.

接続の活用

[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。

[ソース・コード] ウィンドウには、1 接続の状態が表示されます。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 の特別な値、CURRENT USER には接続のユーザ ID が保持されます。

詳細については、「ブレークポイント条件の編集」 846 ページと 「CURRENT USER 特別値」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Copyright © 2007, iAnywhere Solutions, Inc.

索引

記号

@@identity グローバル変数
  IDENTITY カラム, 625
*=
  Transact-SQL 外部ジョイン, 362
* (アスタリスク)
  SELECT 文, 287
<
  比較演算子, 297
=*
  Transact-SQL 外部ジョイン, 362
>
  比較演算子, 297
1 対 1 の関係
  解析, 19
  定義, 6
1 対多の関係
  解析, 19
  定義, 7
2 つのテーブルのジョイン
  説明, 349
3 つ以上のテーブルのジョイン
  説明, 350

A

Access (参照 Microsoft Access)
Adaptive Server Enterprise
  GROUP BY 互換性, 338
  SQL Anywhere への移行, 716
  アーキテクチャ, 613
  互換性, 610
  データ型の変換, 766
  データのインポート／エクスポートの互換性, 723
Adaptive Server Enterprise の互換性
  説明, 723
ALL
  キーワードと UNION 句, 332
  キーワードと集合関数, 316
  サブクエリのテスト, 474
allow_nulls_by_default オプション
  Transact-SQL 互換性の設定, 621
allow_snapshot_isolation オプション
  使用, 129
all-rows 最適化ゴール
  パフォーマンス, 279
ALL 演算子
  サブクエリのテスト, 473
  説明, 473
  注意, 474
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, 128
ALTER SERVER 文
  リモート・サーバの変更, 732
ALTER TABLE 文
  外部キー, 55
  検査制約, 106
  スナップショット・アイソレーションでは使用不可, 128
  同時性, 182
  プライマリ・キー, 52
  例, 48
ALTER 文
  オートコミット, 121
AND
  論理演算子の使用, 305
ANSI
  ANSI 以外のジョイン, 351
  SQL 標準と矛盾, 133
ANSI update constraints
  実行プラン, 576
ANSI 以外のジョイン
  説明, 351
ANSI 準拠, ix
  (参照 SQL 規格)
ANY 演算子
  サブクエリのテスト, 472
  説明, 472
  問題, 472
asejdbc サーバ・クラス
  説明, 762
aseodbc サーバ・クラス
  説明, 765
AS キーワード
  エイリアス, 290
AT 句
  CREATE EXISTING TABLE 文, 739
auto_commit オプション
  Interactive SQL での変更のグループ分け, 121

AUTOINCREMENT
Ultra Light アプリケーション, 103
使用する場合, 181
符号付データ型, 103
負の値, 103
automatic_timestamp オプション
Transact-SQL 互換性の設定, 621
AUTO モード
使用, 658
AvgDiskReads
アクセス・プラン内の推定, 575
AvgDiskReadTime
アクセス・プラン内の推定, 575
AvgDiskWrites
アクセス・プラン内の推定, 575
AvgRowCount
アクセス・プラン内の推定, 575
AvgRunTime
アクセス・プラン内の推定, 575
AVG 関数
使い方, 433
等価な数学上の式, 458

B

BCP フォーマット
ASE でのインポートとエクスポート, 723
BEGIN TRANSACTION 文
トランザクション管理の制限, 750
リモート・データ・アクセス, 750
BETWEEN キーワード
範囲クエリ, 298
BLOB
圧縮されたカラムへの格納, 26
共有, 25
説明, 25
挿入, 502
データベースへの格納, 25
B ツリー・インデックス
説明, 599
B リンク・インデックス
説明, 599

C

CacheHits
アクセス・プラン内の統計, 574
CacheRead
アクセス・プラン内の統計, 574
CacheReadIndLeaf
アクセス・プラン内の統計, 574
CacheReadTable
アクセス・プラン内の統計, 574
CALL 文
制御文, 802
説明, 778
パラメータ, 806
例, 782
CASCADE アクション
説明, 116
CASE 文
制御文, 802
cdata ディレクティブ
使用, 669
CHECK 条件
Transact-SQL, 614
CLOSE 文
カーソル管理手順, 813
COMMENT 文
オートコミット, 121
COMMIT TRANSACTION 文
トランザクション管理の制限, 750
COMMIT 文
参照整合性の検証, 178
説明, 494
トランザクション, 121
複合文, 803
プロシージャとトリガ, 826
リモート・データ・アクセス, 750
COMPUTE 句
CREATE TABLE, 56
Transact-SQL SELECT 文のサポートされない構文, 628
CONNECTION_PROPERTY 関数
説明, 255
CONNECT 文
使用, 35
CORR 関数
式, 448
count (*) の使用
説明, 318
COUNT 関数
NULL, 319
グループ分けされたデータに対する集合関数の適用, 312
説明, 318

 Copyright © 2007, iAnywhere Solutions, Inc.

COVAR_POP 関数
式, 448
等価な数学上の式, 458
COVAR_SAMP 関数
式, 448
等価な数学上の式, 458
CREATE DATABASE 文
ASE, 613
使用, 33
CREATE DEFAULT 文
サポートされない, 614
CREATE DOMAIN 文
Transact-SQL との互換性, 614
ドメインの使用, 110
CREATE EXISTING TABLE 文
使用, 741
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, 734
プロキシ・テーブルのロケーションの指定, 739
CREATE EXTERNLOGIN 文
使用, 737
ディレクトリ・アクセス・サーバの外部ログインの作成, 734
CREATE FUNCTION 文
説明, 787
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, 128
同時性, 182
CREATE PROCEDURE 文
使用, 747
パラメータ, 805
例, 780
CREATE RULE 文
サポートされない, 614
CREATE SERVER 文
DB2 のデータ型の変換, 767
JDBC と Adaptive Server Enterprise, 762
Microsoft SQL Server のデータ型の変換, 771
ODBC と ASE のデータ型の変換, 766
Oracle のデータ型の変換, 769
ディレクトリ・アクセス・サーバの作成, 734
リモート・サーバ, 761
リモート・サーバの作成, 729
CREATE SERVER 文の USING パラメータ値
説明, 761
CREATE TABLE 文
Transact-SQL と互換性のあるテーブルの作成, 626
外部キー, 55
説明, 45
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, 734
同時性, 182
プライマリ・キー, 52
プロキシ・テーブル, 742
プロキシ・テーブルのロケーションの指定, 739
CREATE TABLE 文を使用したプロキシ・テーブルの作成
説明, 742
CREATE TRIGGER 文
説明, 791
CREATE VIEW 文
WITH CHECK OPTION 句, 63
CUBE 句
GROUPING SETS のショートカットとしての使用, 419
説明, 421
CUBE の使用
説明, 421
CUME_DIST 関数
値のセットの中央値を求める, 455
使い方, 455
等価な数学上の式, 458

D

DataSet
XML をインポートするために使用, 650
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, 644
DataSet オブジェクトを使用した XML のインポート
説明, 650
DataSet オブジェクト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
説明, 644
DataWindows
リモート・データ・アクセス, 728
DB_PROPERTY 関数
説明, 255
DB2
データ型変換, 767

db2odbc サーバ・クラス
説明, 767
DB2 リモート・データ・アクセス
説明, 767
dberase ユーティリティ
使用, 44
dbinit ユーティリティ
使用, 34
dbisql ユーティリティ
データベースの再構築, 711
dbo ユーザ
ASE, 615
dbunload ユーティリティ
再構築ツール, 711
データのエクスポート, 697
dbunload ユーティリティを使用したデータのエクスポート
説明, 697
dbunload ユーティリティを使用したデータの再構築
説明, 711
dbxtract ユーティリティ
データの抽出, 715
DB 領域
管理, 613
DDL
オートコミット, 121
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, 128
説明, 30
同時性, 182
debugger_tutorial プロシージャ
説明, 841
DECLARE 文
カーソル管理手順, 813
複合文, 803
プロシージャ, 818
DELETE 文
エラー, 497
使用, 507
例, 497
ロック, 179
DELETE 文、UPDATE 文、INSERT 文でのジョインの使用
説明, 351
DELETE 文または UPDATE 文でのエラー
説明, 497
demo.db ファイル
スキーマ, 344
DENSE_RANK 関数
使い方, 452
等価な数学上の式, 458
DISCONNECT 文
使用, 40
DiskRead
アクセス・プラン内の統計, 574
DiskReadIndInt
アクセス・プラン内の統計, 574
DiskReadIndLeaf
アクセス・プラン内の統計, 574
DiskReadTable
アクセス・プラン内の統計, 574
DiskWrite
アクセス・プラン内の統計, 574
DISK 文
サポートされない, 613
DISTINCT キーワード
集合関数, 319
DISTINCT 句
重複した結果の削除, 294
不要な DISTINCT の削除, 513
DISTINCT の削除
説明, 513
DISTINCT を伴う集合関数の使用
説明, 319
DLL
外部プロシージャの呼び出し, 831
プロシージャからの呼び出し, 830
DML
説明, 494
DROP CONNECTION 文
使用, 40
DROP DATABASE 文
ASE, 613
使用, 43
DROP EXTERNLOGIN 文
使用, 738
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, 128
DROP PROCEDURE 文
使用, 748
DROP SERVER 文
ディレクトリ・アクセス・サーバの削除, 736

リモート・サーバの削除, 731
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, 736
例, 50
DROP TRIGGER 文
説明, 795
DROP 文
オートコミット, 121
同時性, 182
DUMP DATABASE 文
サポートされない, 613
DUMP TRANSACTION 文
サポートされない, 613

E

element ディレクティブ
使用, 666
ER 図
説明, 5
読み込み, 8
ER 図の読み込み
説明, 8
EstCpuTime
アクセス・プラン内の推定, 575
EstDiskReads
アクセス・プラン内の推定, 575
EstDiskReadTime
アクセス・プラン内の推定, 575
EstDiskWrites
アクセス・プラン内の推定, 575
EstRowCount
アクセス・プラン内の推定, 575
EstRunTime
アクセス・プラン内の推定, 575
Excel とリモート・アクセス
説明, 772
EXCEPT アルゴリズム
EXCEPT ハッシュ・アルゴリズム, 562
EXCEPT マージ・アルゴリズム, 562
INTERSECT ハッシュ・アルゴリズム, 563
INTERSECT マージ・アルゴリズム, 563
ハッシュ・ジョイン・アルゴリズムでサポート, 554
マージ・ジョイン・アルゴリズムでサポート, 553
EXCEPT ハッシュ・アルゴリズム
Windows CE, 562
説明, 562
EXCEPT 文
NULL, 335
クエリの結合, 332
使用, 333
ルール, 333
EXCEPT マージ・アルゴリズム
説明, 562
EXECUTE IMMEDIATE 文
プロシージャ, 825
EXISTS 演算子
説明, 477
EXISTS 述部
サブクエリを書き換え, 521
EXPLICIT モード
cdata ディレクティブの使用, 669
element ディレクティブの使用, 666
hide ディレクティブの使用, 667
xml ディレクティブの使用, 668
クエリの記述, 662
構文, 661
使用, 660
EXPLICIT モードのクエリの記述
説明, 662
Extensible Markup Language (参照 XML)

F

FALSE 条件
NULL, 304
FALSE と UNKNOWN の違い
説明, 304
FASTFIRSTROW テーブルのヒント
ネスト・ループ・ジョイン・アルゴリズム, 557
fetchtst
説明, 260
FETCH 文
カーソル管理手順, 813
FIPS 準拠, ix
(参照 SQL 規格)
FIRST_VALUE 関数
使い方, 433
例, 442
First-Row 最適化ゴール
ネスト・ループ・ジョイン・アルゴリズム, 557

FIRST 句
　説明, 330
FOR BROWSE 句
　Transact-SQL SELECT 文のサポートされない構文, 628
FOR OLAP WORKLOAD オプション
　クラスタード・ハッシュ GROUP BY アルゴリズム, 560
FOR READ ONLY 句
　無視, 628
FOR UPDATE 句
　Transact-SQL SELECT 文のサポートされない構文, 628
FORWARD TO 文
　ネイティブ文, 746
　ネイティブ文のリモート・サーバへの送信, 746
FOR XML AUTO
　使用, 658
FOR XML AUTO の使用
　説明, 658
FOR XML EXPLICIT
　cdata ディレクティブの使用, 669
　element ディレクティブの使用, 666
　hide ディレクティブの使用, 667
　xml ディレクティブの使用, 668
　構文, 661
　使用, 660
FOR XML EXPLICIT の使用
　説明, 660
FOR XML RAW
　使用, 656
FOR XML RAW の使用
　説明, 656
FOR XML 句
　AUTO モードの使用, 658
　BINARY データ型, 653
　EXPLICIT モードの構文, 661
　EXPLICIT モードの使用, 660
　IMAGE データ型, 653
　LONG BINARY データ型, 653
　RAW モードの使用, 656
　VARBINARY データ型, 653
　クエリ結果を XML として取得, 653
　使用方法, 653
　制限, 653
FOR 句
　FOR XML AUTO の使用, 658
　FOR XML EXPLICIT の使用, 660
　FOR XML RAW の使用, 656
　クエリ結果を XML として取得, 653
FOR 文
　制御文, 802
FoxPro
　リモート・データ・アクセス, 773
FROM 句
　FROM 句内のストアド・プロシージャ, 295
　FROM 句内の抽出テーブル, 295
　概要, 295
　ジョインの説明, 347
　独立性レベル, 125
FullCompare
　アクセス・プラン内の統計, 574

G

get_value 関数
　使用, 835
GLOBAL AUTOINCREMENT
　デフォルト, 103
go
　バッチ文デリミタ, 799
GRANT 文
　Transact-SQL, 617
　同時性, 182
GROUP BY ALL 句
　Transact-SQL SELECT 文のサポートされない構文, 628
GROUP BY 句
　Adaptive Server Enterprise との互換性, 338
　order by, 331
　SQL/2003 標準, 338
　SQL 規格準拠, 338
　WHERE 句, 325
　WHERE 句と HAVING 句との使用, 321
　エラー, 312
　拡張, 416
　グループ分けされたデータに対する集合関数の適用, 312
　実行, 323
　集合関数, 321
　説明, 321
GROUP BY 句の概要
　説明, 323
GROUP BY のあるクエリを実行する方法

 Copyright © 2007, iAnywhere Solutions, Inc.

説明, 323
GROUP BY リスト
実行プラン内の項目, 579
GROUPING 関数
CUBE 句と使用 (OLAP), 421
NULL プレースホルダの検出, 423
ROLLUP 句と使用 (OLAP), 420
GUID
グローバル・オートインクリメントとの比較, 104
生成, 181
デフォルトのカラム値, 104

H

HAVING 句
GROUP BY 句との使用, 321, 327
WHERE 句, 314
サブクエリ, 469
集合関数を持つ場合と持たない場合, 327
データ・グループの選択, 327
パフォーマンス, 529
論理演算子, 328
HAVING 句でのサブクエリ
説明, 469
hide ディレクティブ
使用, 667
HOLDLOCK キーワード
Transact-SQL, 628

I

I/O
ビットマップのスキャン, 591
iAnywhere デベロッパー・コミュニティ
ニュースグループ, xvii
IBM DB2
DB2 へのリモート・データ・アクセス, 767
SQL Anywhere への移行, 716
id
メタプロパティ名, 648
IDENTITY カラム
値の検索, 625
特殊な IDENTITY, 624
IF 文
制御文, 802
IGNORE NULLS 句
LAST_VALUE 関数での使用, 443
IndAdd
アクセス・プラン内の統計, 574
IndLookup
アクセス・プラン内の統計, 574
INOUT パラメータ
定義, 805
INPUT 文
使用, 687
説明, 687
INPUT 文を使用したデータのインポート
説明, 687
INSERT によるデータの変更
説明, 506
INSERT 文
SELECT, 499
説明, 499
重複データ, 496
データの変更, 506
ロック, 176
INSERT 文を使用したデータのインポート
説明, 689
install-dir
マニュアルの使用方法, xiv
INSTEAD OF トリガ
再帰, 796
説明, 796
ビューの更新に使用, 797
instest
説明, 260
Interactive SQL
.sql ファイルのデフォルト・エディタ, 720
インデックス・コンサルタント, 211
クエリ結果のエクスポート, 700
コマンド・デリミタ, 827
コマンドのロード, 721
コマンド・ファイル, 720
終了の影響, 121
スクリプトの実行, 720
データベースの再構築, 711
テーブル・リストの表示, 342
トランザクションによる変更のグループ分け, 121
バッチ処理, 720
バッチ・モード, 720
リレーショナル・データを XML としてエクスポート, 644
Interactive SQL からリレーショナル・データを XML としてエクスポートする

説明, 644
Interactive SQL の [インポート] ウィザード
説明, 685
Interactive SQL の [インポート] ウィザードの使用
説明, 685
Interactive SQL を使用したデータのエクスポート
説明, 694
INTERSECT アルゴリズム
ハッシュ・ジョイン・アルゴリズムでサポート, 554
マージ・ジョイン・アルゴリズムでサポート, 553
INTERSECT ハッシュ・アルゴリズム
説明, 563
INTERSECT 文
NULL, 335
クエリの結合, 332
使用, 333
ルール, 333
INTERSECT マージ・アルゴリズム
Windows CE, 563
説明, 563
INTO 句
使用, 809
Invocations
アクセス・プラン内の統計, 574
IN キーワード
マッチング・リスト, 299
IN 条件
サブクエリ, 475
IN 条件によるセット・メンバシップのテスト
説明, 475
IN パラメータ
定義, 805
IN リスト
アルゴリズム, 569
最適化, 516
実行プラン内の項目, 579
ISNULL 関数
説明, 304
IS NULL キーワード
説明, 304
ISO SQL 標準
典型的な矛盾, 133
同時性, 133
ISO 準拠, ix
(参照 SQL 規格)
ISYSFKEY
システム・テーブルの使用法, 93
ISYSIDX
システム・テーブルの使用法, 93
ISYSIDXCOL
システム・テーブルの使用法, 93
ISYSIDX システム・テーブル
使用法, 593
ISYSPHYSIDX
システム・テーブルの使用法, 93
ISYSPHYSIDX システム・テーブル
使用法, 593

J

jConnect
メタデータ・サポートのインストール, 40
jConnect メタデータ・サポートを既存のデータベースに追加する
説明, 40
JDBC
実体化ビュー (Materialized View) の候補, 532
JDBC クラス
設定上の注意, 761
JDBC クラスの設定上の注意
説明, 761
JDBC ベースのサーバ・クラス
説明, 761

L

LAST_VALUE 関数
使い方, 433
例, 442
LEAVE 文
制御文, 802
LIKE 述部の最適化
説明, 517
LIKE 探索条件
概要, 299
最適化, 517
ワイルドカード, 301
LOAD DATABASE 文
サポートされない, 613
LOAD TABLE 文
使用, 690
説明, 688
LOAD TABLE 文を使用したデータのインポート
説明, 688

Copyright © 2007, iAnywhere Solutions, Inc.

LOAD TRANSACTION 文
サポートされない, 613
localname
メタプロパティ名, 648
locked tables
アクセス・プラン内の項目, 577
LONG VARCHAR データ型
XML の格納, 643
LOOP 文
制御文, 802
プロシージャ, 813
Lotus Notes
パスワード, 774
リモート・データ・アクセス, 774

M

master データベース
サポートされない, 613
materialized_view_optimization オプション
使い方, 83
max_query_tasks オプション
クエリ内並列処理の制御, 547
MAX 関数
使い方, 433
等価な数学上の式, 458
リライト最適化, 543
MESSAGE 文
プロシージャ, 818
Microsoft Access
SQL Anywhere への移行, 716
リモート・データ・アクセス, 773
Microsoft Excel
リモート・データ・アクセス, 772
Microsoft FoxPro
リモート・データ・アクセス, 773
Microsoft SQL Server
SQL Anywhere への移行, 716
MIN 関数
等価な数学上の式, 458
リライト最適化, 543
MIN 関数と MAX 関数の最適化
説明, 543
Mobile Link
データベースの再構築, 710
msodbc サーバ・クラス
説明, 770

N

NetWare
外部関数, 832
外部プロシージャの呼び出し, 831
NEWID 関数
使用する場合, 181
デフォルトのカラム値, 104
NLM
外部関数, 832
外部プロシージャの呼び出し, 831
プロシージャからの呼び出し, 830
NOHOLDLOCK キーワード
無視, 628
NOT
論理演算子の使用, 305
NOT BETWEEN キーワード
範囲クエリ, 298
Notes とリモート・アクセス
説明, 774
NOT キーワード
探索条件, 298
NULL
DISTINCT 句, 294
EXCEPT 文, 335
INTERSECT 文, 335
OLAP のプレースホルダ, 423
Transact-SQL との互換性, 626
UNION 文, 335
カラム定義, 305
カラムのデフォルト, 621
集合演算子, 335
集合関数, 319
出力, 702
説明, 303
ソート順, 330
デフォルト, 105
デフォルト・パラメータ, 304
比較, 304
プロパティ, 304
NULL 値
Transact-SQL 外部ジョイン, 364
カラムでの許可, 24
挿入, 500
データのインポート, 692
変換エラーの無視, 690
NULL 値の挿入

指定されていないカラムの動作, 500
NULL 入力テーブル
外部ジョイン, 358
NULL の出力
説明, 702
NULL のためのカラムのテスト
説明, 304
NULL のプロパティ
説明, 304
NULL への値の代入
説明, 304

O

ODBC
アプリケーションとロック, 137
外部サーバ, 764
実体化ビュー (Materialized View) の候補, 532
独立性レベルの設定, 137
odbcfet
説明, 260
ODBC サーバ・クラス
説明, 764, 772
ODBC 実行可能アプリケーションからの独立性レベルの設定
説明, 137
OLAP
CUBE 句, 421
GROUP BY 句の拡張, 416
OLAP パフォーマンスの向上, 414
ROLLUP 句, 420
Window 関数, 427
Window 集合関数, 432
Window ランキング関数, 450
WITH CUBE 句, 423
WITH ROLLUP 句, 421
概要, 414
基本集合関数, 433
説明, 413
線形回帰関数, 448
相関関数, 448
標準偏差関数, 444
平方偏差関数, 444
ロー番号付け関数, 457
OLAP 関数
式, 458
OLAP 機能の概要
説明, 414
OLAP の使用
説明, 413
OMNI (参照 リモート・データ・アクセス)
ON EXCEPTION RESUME 句
Transact-SQL, 638
ストアド・プロシージャ, 817
説明, 818
例外処理を伴わない, 822
ON 句
概要, 352
ジョイン, 352
openxml システム・プロシージャ
xp_read_file で使用, 648
使用, 645
openxml を使用したインポート
xp_read_file, 648
説明, 645
OPEN 文
カーソル管理手順, 813
optimization_goal オプション
使用, 527
optimization_workload オプション
クラスタード・ハッシュ GROUP BY, 560
使用, 527
OR
論理演算子の使用, 305
Oracle
SQL Anywhere への移行, 716
データ型変換, 769
Oracle とリモート・アクセス
説明, 769
oraodbc サーバ・クラス
説明, 769
Order-by
実行プラン内の項目, 579
ORDER BY 句
GROUP BY, 331
クエリ結果のソート, 329
結果の制限, 330
実体化ビュー (Materialized View) 定義, 75
パフォーマンス, 267
パフォーマンス改善のためのインデックス使用, 310
ビュー定義の制限, 62
複合インデックス, 598
例, 308

 Copyright © 2007, iAnywhere Solutions, Inc.

ローを常に同じ順序で表示するために必要, 309
ORDER BY と GROUP BY
説明, 331
OR 述部と IN リスト述部の最適化
説明, 516
OUTPUT 文
説明, 695, 700
データを XML としてエクスポートするために使用, 644
OUTPUT 文を使用したデータのエクスポート
説明, 695
OUT パラメータ
定義, 805

P

PCTFREE 設定
テーブルの断片化の削減, 265
PDF
マニュアル, x
PERCENT_RANK 関数
使い方, 456
等価な数学上の式, 458
PerformanceFetch
説明, 260
PerformanceInsert
説明, 260
PerformanceTraceTime
説明, 260
PerformanceTransaction
説明, 260
PowerBuilder
リモート・データ・アクセス, 728
PREPARE TRANSACTION 文
リモート・データ・アクセス, 750
PREPARE 文
リモート・データ・アクセス, 750
ProcCall アルゴリズム
説明, 569
PROPERTY 関数
説明, 255

Q

QOG
クエリ処理, 510
quoted_identifier オプション
Transact-SQL 互換性の設定, 621
説明, 302

R

RAISERROR 文
ON EXCEPTION RESUME, 638
Transact-SQL, 638
RANGE 句
使用, 429
RANK 関数
使い方, 450
等価な数学上の式, 458
RAW モード
使用, 656
READCOMMITTED テーブル・ヒント, ix
(参照 コミットされた読み込み)
readonly-statement-snapshot 独立性レベル
SELECT 文のロック, 176
使用, 144
READUNCOMMITTED テーブル・ヒント, ix
(参照 コミットされない読み込み)
READ 文
コマンド・ファイルの実行, 720
ReceivingTracingFrom
トレーシングの設定, 218
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, 128
REGR_AVGX 関数
式, 448
等価な数学上の式, 458
REGR_AVGY 関数
式, 448
等価な数学上の式, 458
REGR_COUNT 関数
式, 448
等価な数学上の式, 458
REGR_INTERCEPT 関数
式, 448
等価な数学上の式, 458
REGR_R2 関数
式, 448
等価な数学上の式, 458
REGR_SLOPE 関数
式, 448
等価な数学上の式, 458
REGR_SXX 関数
式, 448

等価な数学上の式, 458
REGR_SXY 関数
式, 448
等価な数学上の式, 458
REGR_SYY 関数
式, 448
等価な数学上の式, 458
reload.sql
データベースの再構築, 706
データベースの再ロード, 713
テーブル・データのエクスポート, 712
テーブルのエクスポート, 704
リモート・データベースの再構築, 706
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, 128
REPEATABLEREAD テーブル・ヒント, ix
(参照 繰り返し可能読み出し)
RESIGNAL 文
説明, 822
RESTRICT アクション
説明, 115
RETURN 文
説明, 808
RETURN 文を使って値を返す
説明, 808
REVOKE 文
Transact-SQL, 617
同時性, 182
ROLLBACK 文
説明, 495
トランザクション, 121
トリガ, 632
複合文, 803
プロシージャとトリガ, 826
ROLLUP 演算
GROUP BY 句の概要, 324
ROLLUP 句
GROUPING SETS のショートカットとしての使用, 419
説明, 420
ROLLUP の使用
説明, 420
ROW_NUMBER 関数
使い方, 458
ROWID 関数
ROWID スキャン・アルゴリズムで使用, 551
ROWID スキャン・アルゴリズム
説明, 551
RowReplicate
クエリ実行アルゴリズム, 564
RowsReturned
アクセス・プラン内の統計, 574
ROWS 句
使用, 429
RunTime
アクセス・プラン内の統計, 574

S

sa_ansi_standard_packages システム・プロシージャ
SQL FLAGGER の使用, 605
SA_DEBUG グループ
デバッガ, 839
sa_dependent_views システム・プロシージャ
使い方, 71
sa_locks システム・プロシージャ
使用, 167
sa_migrate_create_fks システム・プロシージャ
使用, 718
sa_migrate_create_remote_fks_list システム・プロシージャ
使用, 718
sa_migrate_create_remote_table_list システム・プロシージャ
使用, 718
sa_migrate_create_tables システム・プロシージャ
使用, 718
sa_migrate_data システム・プロシージャ
使用, 718
sa_migrate_drop_proxy_tables システム・プロシージャ
使用, 718
sa_migrate システム・プロシージャ
使用, 717
sa_migrate ストアド・プロシージャの使用
説明, 717
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, 239
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, 239
sa_server_option システム・プロシージャ
プロシージャ・プロファイリングのフィルタの設定, 238

 Copyright © 2007, iAnywhere Solutions, Inc.

プロシージャ・プロファイリングの無効化, 239
プロシージャ・プロファイリングの有効化, 237
プロシージャ・プロファイリングのリセット, 238
sa_server_option を使用したプロファイリングの無効化
説明, 239
sa_server_option を使用したプロファイリングの有効化
説明, 237
sa_server_option を使用したプロファイリングのリセット
説明, 238
SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT
独立性レベル, 137
SA_SQL_TXN_SNAPSHOT
独立性レベル, 137
SA_SQL_TXN_STATEMENT_SNAPSHOT
独立性レベル, 137
sajdbc サーバ・クラス
説明, 761
samples-dir
マニュアルの使用方法, xiv
saodbc サーバ・クラス
説明, 765
saplan ファイル
説明, 240
SATMP 環境変数
テンポラリ・ファイルのロケーション, 263
SELECT 文
INSERT, 499
INTO 句, 809
Transact-SQL, 627
エイリアス, 290
カラムの順序, 289
カラムの見出し, 290
カーソル, 813
キーとクエリ・アクセス, 272
サブクエリ, 462
説明, 284
ビューの制限, 62
表示文字列, 290
変数, 628
文字データ, 302
ローの指定, 296
select リスト
EXCEPT 文, 332
INTERSECT 文, 332
UNION 文, 332, 333
エイリアス, 290
計算カラム, 291
実行プラン, 576
説明, 287
SELECT リストの値の計算
説明, 291
SELECT を使用した新しいローの追加
説明, 501
self_recursion オプション
Adaptive Server Enterprise, 632
SendingTracingTo
トレーシングの設定, 218
SERIALIZABLE テーブル・ヒント, ix
(参照 直列可能)
set_value 関数
使用, 835
SET DEFAULT アクション
説明, 116
SET NULL アクション
説明, 115
SET OPTION 文
SQL FLAGGER で無視, 607
Transact-SQL, 621
SET 句
UPDATE 文, 504
SHARED キーワード
Transact-SQL SELECT 文のサポートされない構文, 628
SIGNAL 文
Transact-SQL, 638
プロシージャ, 818
SnapshotIsolationState プロパティ
使用, 130
snapshot 独立性レベル
使用, 143
SOAP 関数
デバッグ, 838
SOAP サービス
デバッグ, 838
SOUNDEX 関数
説明, 307
sp_addgroup システム・プロシージャ

Transact-SQL, 616
sp_addlogin システム・プロシージャ
Transact-SQL, 616
サポート, 613
sp_adduser システム・プロシージャ
Transact-SQL, 616
sp_bindefault プロシージャ
Transact-SQL, 614
sp_bindrule プロシージャ
Transact-SQL, 614
sp_changegroup システム・プロシージャ
Transact-SQL, 616
sp_dboption システム・プロシージャ
Transact-SQL, 621
sp_dropgroup システム・プロシージャ
Transact-SQL, 616
sp_droplogin システム・プロシージャ
Transact-SQL, 616
sp_dropuser システム・プロシージャ
Transact-SQL, 616
sp_remote_columns システム・プロシージャ
使用, 742
sp_remote_tables システム・プロシージャ
使用, 732
sp_servercaps システム・プロシージャ
使用, 733
SQL
他の SQL ダイアレクトとの違い, 608
入力, 285
sql_flagger_error_level オプション
SQL FLAGGER の使用, 605
sql_flagger_warning_level オプション
SQL FLAGGER の使用, 605
SQL_TXN_ISOLATION
説明, 137
SQL_TXN_READ_COMMITTED
独立性レベル, 137
SQL_TXN_READ_UNCOMMITTED
独立性レベル, 137
SQL_TXN_REPEATABLE_READ
独立性レベル, 137
SQL_TXT_SERIALIZABLE
独立性レベル, 137
SQL/2003
SQL 文の準拠のテスト, 605
SQL/2003 準拠, ix
(参照 SQL 規格)
SQL/XML
説明, 652
SQL Anywhere
DB2 データ型変換, 767
Microsoft SQL Server のデータ型変換, 771
ODBC と ASE のデータ型の変換, 766
Oracle のデータ型変換, 769
他の SQL ダイアレクトとの違い, 608
マニュアル, x
SQL Anywhere for MS Access 移行ユーティリティ
説明, 716
SQL Anywhere の SQL 機能
説明, 608
SQL Anywhere のデバッガ (参照 デバッガ)
SQL Anywhere へのデータベースの移行
説明, 716
SQLCA.lock
独立性レベル, 134
独立性レベルの選択, 137
SQLCODE 変数
概要, 817
SQL FLAGGER
Ultra Light SQL との SQL 準拠のテスト, 605
起動, 605
説明, 605
標準と互換性, 606
SQLFLAGGER 関数
SQL FLAGGER の使用, 605
SQL FLAGGER の起動
説明, 605
SQL Remote
リモート・データ・アクセス, 756
SQL Server
データ型変換, 771
リモート・アクセス, 770
SQLSetConnectOption
説明, 137
SQLSTATE 変数
概要, 817
SQLX (参照 SQL/XML)
SQL 規格
ANSI 以外のジョイン, 351
GROUP BY 句, 338
準拠, 603
説明, 3
SQL 規格への準拠, ix
(参照 SQL 規格)

 Copyright © 2007, iAnywhere Solutions, Inc.

SQL 規格への適合, ix
(参照 SQL 規格)
SQL クエリ
説明, 285
SQL コマンド・ファイル
Interactive SQL で開く, 720
作成, 720
実行, 720
出力の書き込み, 721
説明, 720
SQL コマンド・ファイルの実行
説明, 720
SQL コマンド・ファイルの使用
説明, 720
SQL プリプロセッサ
SQL FLAGGER の使用, 605
SQL 文
Sybase Central でのロギング, 38
互換性のある SQL 文の記述方法, 626
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, 128
SQL 文のロギング
説明, 38
SQL を使用したプロキシ・テーブルの作成
説明, 741
statement-snapshot 独立性レベル
SELECT 文のロック, 176
使用, 143
STDDEV_POP 関数
使い方, 445
等価な数学上の式, 458
例, 445
STDDEV_SAMP 関数
使い方, 446
等価な数学上の式, 458
例, 446
STDDEV 関数
STDDEV_SAMP 関数を参照, 445
等価な数学上の式, 458
SUM 関数
使い方, 433
等価な数学上の式, 458
Sybase Central
jConnect メタデータ・サポートのインストール, 40
SQL 文のロギング, 38
アプリケーション・プロファイリング, 202
[アプリケーション・プロファイリング] ウィザードの使用, 203
インデックスの検証, 91
インデックスの作成, 90
オプティマイザによる実体化ビュー (Materialized View) の使用の無効化, 82
オプティマイザによる実体化ビュー (Materialized View) の使用の有効化, 82
外部キーの管理, 54
カラム制約, 108
カラム・デフォルト, 101
システム・オブジェクトの内容の表示, 37
システム・オブジェクトの表示, 37
実体化ビュー (Materialized View) の削除, 85
実体化ビュー (Materialized View) の無効化, 81
実体化ビュー (Materialized View) の有効化, 81
接続しないでデータベースを起動, 34
データベース・オプションの設定, 36
データベースのアップグレード, 39
データベースのアンロード, 698
データベースの再構築, 711
データベースの削除, 43
データベースの作成, 32
データベースの消去, 43
データベースの停止, 41
テーブル制約, 108
テーブルのコピー, 59
テーブルの削除, 50
テーブルの作成, 45
テーブルの変更, 47
統合データベースの設定, 37
ビューの削除, 65
ビューの作成, 61
ビューの変更, 64
ビューの無効化, 67
ビューの有効化, 67
プライマリ・キーの管理, 52
プロシージャの変換, 634
Sybase Central からのデータベース統計値のモニタリング
説明, 243
Sybase Central でカラム・デフォルトを設定する
説明, 101
Sybase Central でテーブル制約およびカラム制約を編集する
説明, 108
Sybase Central パフォーマンス・モニタ

説明, 243
統計値の追加と削除, 244
Sybase Central パフォーマンス・モニタを開く
説明, 244
Sybase Central を使用したプロキシ・テーブルの作成
説明, 740
Sybase Central を使用するストアド・プロシージャの変換
説明, 634
SYSCOLSTAT
システム・ビューのカラム統計の更新, 526
SYSCOLUMNS
Transact-SQL の名前の重複, 620
SYSINDEXES
Transact-SQL の名前の重複, 620
SYSSERVER
リモート・サーバのシステム・ビュー, 729
SYSTAB
互換性ビューの情報, 71
SYSVIEWS
統合ビューの情報, 71

T

TEMP 環境変数
テンポラリ・ファイルのロケーション, 263
TIMESTAMP データ型
Transact-SQL, 623
TMPDIR 環境変数
テンポラリ・ファイルのロケーション, 263
TMP 環境変数
テンポラリ・ファイルのロケーション, 263
TOP 句
説明, 330
TRACEBACK 関数
説明, 818
Transact-SQL
IDENTITY カラム, 624
NULL, 626
NULL 値とジョイン, 364
timestamp カラム, 623
WITH ROLLUP の使用, 421
移植可能な SQL の記述, 626
外部ジョイン, 362
外部ジョインとビュー, 364
外部ジョインの制限事項, 363
概要, 610
結果セット, 635
後続ブランク, 620
互換性のある SQL 文の記述方法, 626
ジョイン, 629
説明, 610
データベースの作成, 619
トリガ, 631
バッチ, 632
プロシージャ, 631
変数, 636
Transact-SQL 外部ジョインに対する NULL の影響
説明, 364
Transact-SQL 外部ジョインを使ったビューの使用
説明, 364
Transact-SQL 互換性
データベース・オプションの設定, 621
Transact-SQL と互換性のあるデータベースの作成
説明, 619
Transact-SQL との互換性
説明, 610
データベース, 622
Transact-SQL との互換性を意識したデータベースの設定
説明, 619
Transact-SQL との互換性を維持するためのオプション設定
説明, 621
Transact-SQL のストアド・プロシージャの概要
説明, 631
Transact-SQL の特殊な timestamp カラムとデータ型
説明, 623
Transact-SQL のトリガの概要
説明, 631
Transact-SQL のバッチの概要
説明, 632
Transact-SQL のプロシージャ言語の概要
説明, 631
Transact-SQL プロシージャから返される結果セット
説明, 635
Transact-SQL プロシージャでのエラー処理
説明, 637
trantest
説明, 260
TRUNCATE TABLE 文

 Copyright © 2007, iAnywhere Solutions, Inc.

スナップショット・アイソレーションでの使用, 128
説明, 508
tsequal 関数
SQL 構文, 624

U

Ultra Light
SQL 文の準拠のテスト, 605
Ultra Light SQL
SQL Anywhere の文が Ultra Light SQL に準拠するかどうかのテスト, 605
UNION
クエリ実行アルゴリズム, 565
UNION ALL
クエリ実行アルゴリズム, 564
UNION 文
NULL, 335
クエリの結合, 332
ルール, 333
UNION または GROUP 化されたビューと抽出テーブルでの述部のプッシュダウン
説明, 515
Union リスト
実行プラン内の項目, 579
UNIX
最小キャッシュ・サイズ, 273
最大キャッシュ・サイズ, 273
初期キャッシュ・サイズ, 273
UNKNOWN
NULL, 304
UNLOAD TABLE 文
説明, 696
UNLOAD TABLE 文の使用
説明, 696
UNLOAD TABLE 文を使用したデータベースの再構築
説明, 712
UNLOAD 文
説明, 697
UNLOAD 文の使用
説明, 697
UPDATE によるデータの変更
説明, 504
UPDATE の競合
スナップショット・アイソレーション, 133
UPDATE 文
エラー, 497
使用, 504
例, 497
ロック, 178
Upsize ツール
Microsoft Access, 716
UUID
グローバル・オートインクリメントとの比較, 104
生成, 181
デフォルトのカラム値, 104

V

ValuePtr パラメータ
説明, 137
VAR_POP 関数
使い方, 447
等価な数学上の式, 458
例, 447
VAR_SAMP 関数
使い方, 448
等価な数学上の式, 458
例, 448
VARIANCE 関数
VAR_SAMP 関数を参照, 447
等価な数学上の式, 458
VersionStorePages プロパティ
使用, 128

W

wait_for_commit オプション
使用, 178
Watcom-SQL
互換性のある SQL 文の記述方法, 626
説明, 610
WHERE 句
GROUP BY 句, 325
GROUP BY 句との使用, 321
HAVING 句, 314
HAVING との比較, 327
NULL 値, 304
UPDATE 文, 505
サブクエリ, 468
ジョイン, 354
説明, 296
テーブル内のローの変更, 504
パターン一致, 299

パフォーマンス, 267, 529
日付の比較の概要, 306
文字列比較, 300
WHERE 句でのサブクエリの使用
説明, 468
WHERE 句での範囲の使用
説明, 298
WHERE 句での比較演算子の使用
説明, 297
WHERE 句での文字列のマッチング
説明, 299
WHERE 句でのリストの使用
説明, 299
WHERE 句のサブクエリのジョインへの変換
説明, 484
WHILE 文
制御文, 802
Windows
最小キャッシュ・サイズ, 273
最大キャッシュ・サイズ, 273
初期キャッシュ・サイズ, 273
Windows CE
キャッシュ・サイズとページ・サイズの考慮事項, 592
Windows パフォーマンス モニタ
説明, 245
Windows パフォーマンス・モニタを使用したデータベースの統計値のモニタリング
説明, 245
Window 関数
クエリ実行アルゴリズム, 570
集合関数のリスト, 432
説明, 425
ランキング関数のリスト, 450
ローの番号付け, 457
WINDOW 句
SELECT 文での使用, 428
Window 集合関数
OLAP, 432
サポートされる関数のリスト, 427
説明, 432
Window ランキング関数
サポートされる関数のリスト, 427
WITH CHECK OPTION 句
説明, 63
WITH CHECK OPTION 句の使い方
説明, 63
WITH CUBE 句
説明, 423
WITH CUBE の使用
説明, 423
WITH EXPRESS CHECK
パフォーマンス, 279
WITH ROLLUP 句
説明, 421
WITH ROLLUP の使用
説明, 421
WITH 句
オプティマイザのアルゴリズム, 564
共通テーブル式, 392

X

XML
DataSet オブジェクトを使用したインポート, 650
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, 644
FOR XML AUTO の使用, 658
FOR XML EXPLICIT の使用, 660
FOR XML RAW の使用, 656
Interactive SQL からデータを XML としてエクスポート, 644
openxml を使用したインポート, 645
SQL Anywhere データベースでの使用, 641
クエリ結果を XML として取得, 653
定義, 642
リレーショナル・データからクエリ結果を XML として取得, 652
リレーショナル・データとしてインポート, 645
リレーショナル・データベースにおける格納, 643
リレーショナル・データを XML としてエクスポート, 644
XMLAGG 関数
使用, 670
XMLCONCAT 関数
使用, 671
XMLELEMENT 関数
使用, 672
XMLFOREST 関数
使用, 674
XMLGEN 関数
使用, 675

 Copyright © 2007, iAnywhere Solutions, Inc.

xml ディレクティブ
using, 668
XML データ型
使用, 643
XML と SQL Anywhere
説明, 642
XML 文書をリレーショナル・データとしてインポートする
説明, 645
xp_read_file
XML のインポート, 648
XPath
使用, 645

あ

アイコン
マニュアルで使用, xv
アイドル・アクティブ／秒の統計値
説明, 247
アイドル書き込み／秒の統計値
説明, 247
アイドル・チェックポイント時間の統計値
説明, 247
アイドル・チェックポイント／秒の統計値
説明, 247
アクション
CASCADE, 116
RESTRICT, 115
SET DEFAULT, 116
SET NULL, 115
アクセス・プラン
説明, 523
統計の説明, 574
アスタリスク (*)
SELECT 文, 287
値の格納
共通テーブル式, 396
圧縮
カラム, 26
圧縮 B ツリー
インデックス, 599
圧縮機能の使用
パフォーマンス向上のためのヒント, 278
圧縮されたカラム
BLOB の格納, 26
説明, 26
アップグレード
データベース・ファイル・フォーマット, 707
アトミックなトランザクション
説明, 120
アトミックな複合文
説明, 803
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, 230
I/O 帯域幅が制限要因であるかどうかの判断, 230
インデックス・コンサルタント, 210
運用データベース, 216
説明, 202
トレーシング・セッションの作成, 226
トレーシング・データベース, 216
プロシージャ・プロファイリング, 204
メモリが制限要因であるかどうかの判断, 230
要求トレース分析, 232
[アプリケーション・プロファイリング] ウィザード
起動, 203
自動起動の有効化と無効化, 204
説明, 203
アプリケーション・プロファイリング・モード
使用, 202
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
説明, 204
アプリケーション論理のデバッグ
説明, 231
アポストロフィ
文字列, 302
アルゴリズム, ix
(参照 クエリ実行アルゴリズム)
クエリ実行, 546
アルファベット順
ORDER BY 句, 308
暗号化
オブジェクトを隠す, 835
キャッシュ・サイズ, 259
実体化ビュー (Materialized View), 79
アンロード・ツール
説明, 698
[データのアンロード] ダイアログ, 699
[データベース・アンロード] ウィザード, 698
アンロードと再ロード
データベース, 714
同期に関連しないデータベース, 709

同期に関連するデータベース, 710
アーキテクチャ
Adaptive Server, 613

い

以下
比較演算子, 297
意思決定支援
独立性レベル, 145
以上
比較演算子, 297
移植可能な SQL
記述, 626
移植可能な SQL を記述するための一般的なガイドライン
説明, 626
依存性
ビューの依存性, 67
依存性とスキーマ変更
説明, 70
一意性
インデックスを使用した強化, 599
一意性制約
生成されたインデックス, 595
位置テーブル・ロック
説明, 171
挿入ロック, 173
幻ロック, 172
位置ロック
期間, 167
説明, 166
一貫性
ISO SQL 標準, 133
繰り返し可能読み出し, 133
繰り返し可能読み出しとロック, 174
繰り返し可能読み出しのチュートリアル, 151
繰り返し不可能読み出しの例, 153
実際のロックの意味, 162
スナップショット・アイソレーション, 176
正当性とスケジューリング, 144
説明, 120
ダーティ・リード, 133
ダーティ・リードとロック, 174
ダーティ・リードのチュートリアル, 147
直列不可能なスケジュールの影響, 144
典型的なトランザクション, 145
独立性レベル, 125, 134
独立性レベル 0, 174
トランザクション中, 133
幻ロー, 133
幻ローとロック, 175
幻ローのチュートリアル, 157
ロックを使用して保証, 166
意図的ロック
スナップショット・アイソレーション, 170
説明, 170
インポート中の変換エラーの処理
説明, 690
イベント
許可される文, 829
プロファイリング結果の生成と確認, 204
イメージ
挿入, 502
インストール
jConnect メタデータ・サポート, 40
インデックス
B リンク, 599
HAVING 句のパフォーマンス, 529
Transact-SQL, 623
WHERE 句のパフォーマンス, 529
インデックス間の相関関係, 215
インデックス・コンサルタント, 87
インデックス・コンサルタントの使用, 210
インデックス・コンサルタントの推奨内容の解釈, 212
インデックス・ヒント, 88
概要, 310
仮想, 213
カラムの順序の効果, 597
共有される物理インデックスの特定, 594
クラスタード, 88
クラスタードと非クラスタード, 599
計算カラム, 90
検索引数可能な述部, 529
検証, 91
構造, 596
候補, 213
再構築, 92
最適化, 592
削除, 92
作成, 90
システム・ビューを使用した検査, 93
述部の分析, 529
述部を満たすために使用, 528

使用する場合, 87
処理, 87
制限事項と考慮事項, 592
生成, 595
説明, 592
タイプ, 599
断片化, 266
テンポラリ・テーブル, 595
統計値のリスト, 251
パフォーマンス, 261
パフォーマンスの改善, 596
費用対効果, 210
頻繁に検索するカラム, 88
ファンアウトとページ・サイズ, 592
複合, 597
物理, 593
ページ・サイズの推奨値, 592
未使用, 214
利点とロック, 146
リーフ・ページ, 596
論理, 593
インデックス DISTINCT アルゴリズム
クエリ実行アルゴリズム, 558
インデックス GROUP BY アルゴリズム
クエリ実行アルゴリズム, 560
インデックス関数
ローの番号付け, 457
インデックス・コンサルタント
概要, 87
クエリに対する推奨内容の表示, 211
サーバの状態, 214
使用, 211
推奨内容の解釈, 212
推奨内容の実装, 214
推奨内容の評価, 214
接続の状態, 214
説明, 210
バージョン 9 のデータベース・サーバへの接続, 211
インデックス・コンサルタントの使用
説明, 211
インデックス・コンサルタントの推奨内容の解釈
説明, 212
インデックス・コンサルタントの推奨内容の実装
説明, 214
[インデックス作成] ウィザード
使用, 90
インデックス・スキャン
説明, 550
並列インデックス・スキャン, 551
インデックス・セットの選択
説明, 87
インデックス・ネスト・ループ・ジョイン
説明, 553
インデックスの完全比較／秒の統計値
説明, 251
インデックスの検証
説明, 91
インデックスの再構築
説明, 92
インデックスの削除
インデックス, 92
インデックスの作成
説明, 90
インデックスの選択性
説明, 596
インデックスの操作
説明, 87
インデックスのその他の使用法
説明, 599
インデックスのタイプ
説明, 599
インデックスの断片化
説明, 266
インデックスの断片化削減
パフォーマンス向上のためのヒント, 266
インデックスの追加／秒の統計値
説明, 251
インデックスのパフォーマンス向上
説明, 596
インデックスの有効な使用
パフォーマンス向上のためのヒント, 271
インデックスのルックアップ／秒の統計値
説明, 251
インデックスはいつ使うか
説明, 87
インデックス・ヒント
使用, 88
インデックス・ファンアウト
説明, 596
インデックス名
実行プラン内の項目, 578
インポート
ASE 互換性, 723

DataSet オブジェクトを使用して XML 文書をリレーショナル・データとしてインポートする, 650
NULL 値, 691
openxml を使用して XML 文書をリレーショナル・データとしてインポートする, 645
XML 文書をリレーショナル・データとしてインポートする, 645
一致しないテーブル構造, 691
概要, 684
ツール, 685
テンポラリ・テーブルの使用, 94
[インポート] ウィザード
使用, 685
説明, 685
インポート中の変換エラー
説明, 690
インポート・ツール
INPUT 文, 687
INSERT 文, 689
Interactive SQL の [インポート] ウィザード, 685
LOAD TABLE 文, 688
説明, 685
プロキシ・テーブル, 689
インポート用のテーブル構造
説明, 691
引用符
Adaptive Server Enterprise, 302
文字列, 302

う

ウィザード
アプリケーション・プロファイリング, 203
インデックス作成, 90
インポート, 685
外部キー作成, 54
ディレクトリ・アクセス・サーバの作成, 734
データベース・アップグレード, 39
データベース・アンロード, 698
データベース移行, 716
データベース作成, 32
データベース消去, 43
データベース・トレーシング, 226
ドメイン作成, 110
トリガ作成, 792
ビュー作成, 61
プロキシ・テーブル作成, 740
プロシージャ作成, 780
リモート・サーバ作成, 730
リモート・プロシージャ作成, 747
ウィンドウ (OLAP)
RANGE 句を使用したサイズ変更, 429
ROWS 句を使用したサイズ変更, 429
SELECT 文の WINDOW 句, 428
インラインのウィンドウの定義, 428
サイズ, 429
ウィンドウでのランキング関数
説明, 450
ウィンドウでのロー番号付け関数
説明, 450, 457
ウォーミング
キャッシュ, 277
運用データベース
説明, 216

え

影響
直列不可能なトランザクション・スケジューリング, 144
トランザクション・スケジューリング, 144
エイリアス
計算カラム, 291
説明, 290
相関名, 295
エクスポート
ASE 互換性, 723
NULL 値, 702
概要, 694
クエリ結果, 700
スキーマ, 712
テーブル, 704
リレーショナル・データを XML としてエクスポートする, 644
エクスポート・ツール
dbunload ユーティリティ, 697
Interactive SQL ウィザード, 694
OUTPUT 文, 695
UNLOAD TABLE 文, 696
UNLOAD 文, 697
説明, 694
エラー
Transact-SQL, 637, 638
プロシージャとトリガ, 817

 Copyright © 2007, iAnywhere Solutions, Inc.

変換, 690
エラー処理
ON EXCEPTION RESUME, 818
プロシージャとトリガ, 817
演算子
NOT キーワード, 298
算術, 292
条件の接続, 305
ハッシュ・テーブル・スキャン, 549
並列テーブル・スキャン, 551
優先度, 292
エンティティ
整合性の確保, 113
説明, 5
選択, 11
属性, 5
定義, 5
エンティティ整合性
概要, 98
クライアント・アプリケーションによる違反, 113
プライマリ・キー, 608
エンティティ整合性と参照整合性の確保
説明, 113
エンティティと関係の決定
説明, 11

お

大文字と小文字の区別
ASE 互換データベースの作成, 620
SQL, 285
Transact-SQL との互換性, 622
識別子, 622
ソート順, 330
データ, 622
データベース, 622
テーブル名, 285
ドメイン, 622
パスワード, 622
ユーザ ID, 622
リモート・アクセス, 756
大文字と小文字の不要な変換の排除
説明, 521
オブジェクト
隠す, 835
ロック可能なオブジェクト, 166
オブジェクト名の互換性
説明, 623
オプション
blocking, 140
DEFAULTS, 692
isolation_level, 135
データベース・オプションの設定, 36
オプションの外部キー
説明, 114
オプションの関係
説明, 8
オプティマイザ
仮定条件, 527
クエリ処理のフェーズ, 510
最小限の管理作業, 527
実体化ビュー (Materialized View) の使用, 82
述部の分析, 529
省略, 511
説明, 523
セマンティック・サブクエリ変形, 512
セマンティック変形, 512
オプティマイザでの実体化ビュー (Materialized View) の古さ閾値の設定
説明, 83
オプティマイザによる実体化ビュー (Materialized View) の使用の有効化と無効化
説明, 82
オプティマイザの推定
説明, 524
オプティマイザの優先度の選択
パフォーマンス向上のためのヒント, 258
オンライン分析処理 (参照 OLAP)
オンライン・マニュアル
PDF, x
オートインクリメント
IDENTITY カラム, 624
デフォルト, 102
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
パフォーマンス向上のためのヒント, 270
オートコミット
ALTER 文, 121
COMMENT 文, 121
DROP 文, 121
データ定義文, 121
トランザクション, 121
パフォーマンス, 268
オートコミット・モードをオフにする

パフォーマンス向上のためのヒント, 268
オートメーション
ユニーク・キーの生成, 181
オーファンと参照整合性
説明, 177

か

改行
SQL, 285
開始
トランザクション, 121
解析
関係, 19
解析ツリー
クエリ処理, 510
階層データ構造
階層データ構造の探索, 398
構成部品の問題, 400
概念データベース・モデル
定義, 5
概念データ・モデリング
説明, 3
外部関数
取り消し, 834
パラメータを渡す, 834
プロトタイプ, 832
リターン・タイプ, 835
外部キー
SQL を使用した作成, 55
SQL を使用した変更, 55
Sybase Central で削除, 54
Sybase Central で作成, 54
Sybase Central で表示, 54
管理, 53
キー・ジョイン, 377
参照整合性, 114
整合性, 608
生成されたインデックス, 595
挿入, 496
パフォーマンス, 271
必須／オプション, 114
役割名, 378
[外部キー作成] ウィザード
使用, 54
外部キーの管理
説明, 53
外部キーを使ったクエリのパフォーマンス改善
パフォーマンス向上のためのヒント, 272
外部参照
HAVING 句, 469
集合関数, 317
説明, 479
定義, 467
外部サーバ
ODBC, 764
外部ジョイン
Transact-SQL, 362, 629
Transact-SQL とビュー, 364
Transact-SQL の制限, 363
ジョイン条件, 359
ジョインの削除によるリライト最適化, 518
スター・ジョインの例, 369
制限, 363
説明, 358
内部ジョインへの変換, 517
ビューと抽出テーブル, 361
複雑, 360
外部ジョインから内部ジョインへの変換
説明, 517
外部呼び出し
プロシージャと関数の作成, 830
外部ログイン
削除, 738
作成, 737
説明, 737
[外部ログイン作成] ウィザード
使用, 737
外部ログインの削除
説明, 738
外部ログインの作成
説明, 737
外部ログインの使用
説明, 737
外部ロード
説明, 682
書き込みロック
説明, 169
書き込みを意図したテーブル・ロック
説明, 171
各種独立性レベルでの典型的なトランザクション
説明, 145
カスケードされた参照アクションを最小限に抑える
説明, 260

 Copyright © 2007, iAnywhere Solutions, Inc.

カスタマイズ
グラフィカルなプランの表示, 582
仮想インデックス
インデックス・コンサルタント, 213
説明, 213
仮想メモリ
貴重なリソース, 529
カタログ
ASE 互換性, 614
依存性情報の検索, 71
インデックス情報, 93
カタログ内のインデックス情報
説明, 93
カタログ内のビューの依存性情報
説明, 71
カッコ
UNION 演算子, 332
算術文内, 292
稼働条件
SQL Anywhere のデバッガ, 839
カラム
GROUP BY 句, 321
IDENTITY, 624
NULL 値の許可, 24
SELECT 文, 289
select リスト, 288
SQL を使用した変更, 48
Sybase Central を使用した変更, 47
timestamp, 623
圧縮, 26
カラム制約の管理, 108
計算, 291
検査制約, 108
制約, 27
デフォルト, 100
データ型, 24
データ型とドメインの割り当て, 110
プロパティ, 24
命名, 24
カラム一意性の強化
説明, 599
カラム属性
AUTOINCREMENT, 181
NEWID, 181
デフォルト値の生成, 181
カラム・デフォルト
変更と削除, 101
カラム・デフォルトの使い方
説明, 100
カラム・デフォルトの変更と削除
説明, 101
カラム統計 (参照 ヒストグラム)
更新, 526
説明, 524
カラム統計の更新
説明, 526
カラムに対する検査制約の使い方
説明, 106
カラムの圧縮
説明, 26
カラムのデータ型の選択
説明, 24
カラム名の選択
説明, 24
関係
1 対 1, 6
1 対多, 7
エンティティに変更, 9
解析, 19
カーディナリティ, 6
再帰, 9
説明, 6
選択, 11
多対多, 7
定義, 5
必須／オプション, 8
役割, 7
関係の解析
データベース設計, 19
環状ブロックの競合
説明, 141
関数
SOUNDEX 関数, 307
TRACEBACK, 818
tsequal, 624
Window ランキング関数, 450
ウィンドウ, 427
外部, 830
キャッシュ, 566
決定性, 566
プロファイリング結果の生成と確認, 204
べき等, 566
ユーザ定義, 787
間接参照

データベース・オブジェクト, 71
完全比較
説明, 596
カンマ
スター・ジョイン, 366
テーブル式のジョイン時, 381
テーブル式リスト, 356
管理者の役割
ASE, 615
カーソル
LOOP 文, 813
SELECT 文, 813
安定性, 135
ジョインでの更新, 351
不安定性, 135
プロシージャ, 813
プロシージャとトリガ, 813
カーソル安定性
説明, 135
カーソル管理
概要, 813
カーソルの統計値
説明, 253
カーソル不安定性
説明, 135
カーディナリティ
関係, 6
実行プラン内の項目, 578

き

規格, ix
(参照 SQL 規格)
規格と互換性, ix
(参照 SQL 規格)
期間
ロック, 167
記号
文字列比較, 300
規則
表記, xii
マニュアルでのファイル名, xiv
基本集合関数
OLAP, 433
基本となる仮定条件
オプティマイザ, 527
キャッシュ
UNIX, 276
暗号化データベースには大きいキャッシュが必要, 259
キャッシュ・サイズの拡大によるパフォーマンスの向上, 259
サイズのモニタリング, 277
サブクエリ, 566
実行プラン, 544
準備, 277
初期サイズ、最小サイズ、最大サイズ, 273
動的サイズ決定, 275
パフォーマンス向上へのキャッシュの使用, 273
ユーザ定義関数, 566
読み込みヒット率, 585
キャッシュ・ウォーミング
説明, 277
キャッシュ・ウォーミングの使用
説明, 277
キャッシュ・サイズ
UNIX, 276
Windows, 275
Windows CE, 275
Windows CE の考慮事項, 592
初期サイズ、最小サイズ、最大サイズ, 273
パフォーマンスの考慮事項, 591
ページ・サイズ, 591
モニタリング, 277
キャッシュ・サイズ決定
パフォーマンス, 275
キャッシュ・サイズの拡大
パフォーマンス向上のためのヒント, 259
キャッシュ・サイズの現在の統計値
説明, 246
キャッシュ・サイズの最小の統計値
説明, 246
キャッシュ・サイズの最大の統計値
説明, 246
キャッシュ・サイズのピーク値の統計値
説明, 246
キャッシュ・サイズのモニタリング
説明, 277
キャッシュされたプラン
オプティマイザ・バイパス, 511
キャッシュ置換：合計ページ数／秒の統計値
説明, 251
キャッシュのスカベンジ・アクセスの統計値
説明, 251

 Copyright © 2007, iAnywhere Solutions, Inc.

キャッシュのスカベンジの統計値
説明, 251
キャッシュの統計値
リスト, 246
キャッシュのパニックの統計値
説明, 251
キャッシュ・ヒット／秒の統計値
説明, 246
キャッシュ・ページの空きの統計値
説明, 251
キャッシュ・ページの固定の統計値
説明, 251
キャッシュ・ページのファイル・ダーティの統計値
説明, 251
キャッシュ・ページのファイルの統計値
説明, 251
キャッシュ・ページの割り当て構造体の統計値
説明, 251
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, 246
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, 246
キャッシュ読み込みの合計ページ数／秒の統計値
説明, 246
キャッシュ読み込みのテーブル／秒の統計値
説明, 246
キャッシュに使用されるメモリの制限
パフォーマンス向上のためのヒント, 273
競合
環状ブロッキング, 141
スナップショット・アイソレーション, 133
テーブル・ロック, 170
トランザクション・ブロック, 140
トランザクション・ブロックの例, 155
ロック, 173
競合するトリガ
実行順序, 796
共通テーブル式
一般的な使用例, 395
階層データ構造の探索, 398
構成部品の問題, 400
再帰でのデータ型, 404
再帰に関する制限, 399
最短距離の問題, 406
使用できる条件, 394
説明, 391
定数セットの格納, 396
複数の集合レベル, 395
共有オブジェクト
プロシージャからの呼び出し, 830
共有テーブル・ロック
説明, 171
共有ロック
説明, 168
排他, 168
キー
パフォーマンス, 271
割り当て, 16
キー・ジョイン
2 つ以上の外部キー, 378
ON 句, 353
ON 句の使用, 378
カンマを含まないテーブル式, 381
カンマを含まないテーブル式とリスト, 384
規則, 387
説明, 377
テーブル式, 381
テーブル式リスト, 382
ビューと抽出テーブル, 385
キー・ジョイン操作規則
説明, 387
キー・タイプ
実行プラン内の項目, 578
キーの有効な使用
パフォーマンス向上のためのヒント, 271
キーワード
HOLDLOCK, 628
NOHOLDLOCK, 628
リモート・サーバ, 756

く

句
COMPUTE, 628
FOR BROWSE, 628
FOR READ ONLY, 628
FOR UPDATE, 628
GROUP BY ALL, 628
INTO, 809
ON EXCEPTION RESUME, 638, 822
説明, 284
クエリ

SELECT 文, 284
Transact-SQL, 627
エクスポート, 700
大文字と小文字の不要な変換の排除, 521
オプティマイザの省略, 511
共通テーブル式, 392
最適化, 523
実行プラン, 571
集合操作, 332
処理のフェーズ, 510
説明, 284
セマンティック変形, 512
セマンティック変形の種類, 512
定義された単純なクエリ, 511
テーブルからのデータの選択, 283
不要な内部ジョインと外部ジョインの削除, 518
並列処理が使用される状況, 548
クエリ・オプティマイザ, ix
(参照 オプティマイザ)
説明, 523
クエリ書き換えフェーズ
クエリ処理, 510
クエリ間並列処理, ix
(参照 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, 547
説明, 547
クエリ結果
エクスポート, 700
クエリ結果にあるカラム名の変更
説明, 290
クエリ結果のエクスポート
説明, 700
クエリ結果のグループへの編成
説明, 321
クエリ結果の効果的なソート
パフォーマンス向上のためのヒント, 267
クエリ結果の文字列
説明, 290
クエリ結果の要約、グループ化、ソート
説明, 315
クエリ結果を XML として取得
説明, 652
クエリ最適化
IN リスト述部, 516
オプティマイザのバイパス, 511
クエリ式アルゴリズム
EXCEPT アルゴリズム, 562
INTERSECT アルゴリズム, 563
説明, 562
クエリ実行
説明, 547
ビュー・マッチング, 531
並列処理, 547
クエリ実行アルゴリズム
EXCEPT と INTERSECT, 562
IN リスト, 569
ProcCall アルゴリズム, 569
ROWID スキャン・アルゴリズム, 551
RowReplicate, 564
UNION ALL, 564
Window 関数, 570
インデックス DISTINCT アルゴリズム, 558
インデックス GROUP BY アルゴリズム, 560
インデックス・スキャン, 550
クラスタード・ハッシュ GROUP BY, 560
グループ化アルゴリズム, 560
交換, 567
再帰 UNION, 564
再帰テーブル, 564
実行プランに使用される省略形, 572
順序付き DISTINCT, 559
順序付き GROUP BY, 561
順序付き GROUP BY SETS, 562
ジョイン, 552
上位 N のソート, 565
説明, 546
セミジョイン, 555, 557
ソート, 565
ソート GROUP BY SETS, 562
ソートと UNION, 565
単一の GROUP BY, 562
単一の XML GROUP BY, 562
逐次テーブル・スキャン, 552
抽出テーブル, 567
重複排除, 558
ネスト・ループ・ジョイン, 557
ネスト・ループ・セミジョイン, 557
ハッシュ DISTINCT, 558
ハッシュ GROUP BY, 560
ハッシュ GROUP BY SETS, 561
ハッシュ NOT EXIST, 554
ハッシュ・ジョイン, 554, 555
ハッシュ・セミジョイン, 555

Copyright © 2007, iAnywhere Solutions, Inc.

ハッシュ・テーブル・スキャン, 549
ハッシュ非セミジョイン, 554
ハッシュ・フィルタ, 568
非セミジョイン, 554
フィルタと事前フィルタ, 568
並列インデックス・スキャン, 551
並列テーブル・スキャン, 551
マージ・ジョイン, 556
ロー・コンストラクタ・スキャン・アルゴリズム, 570
ローの制限, 570
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, 757
クエリ処理
フェーズ, 510
クエリ処理におけるワーク・テーブルの使用
パフォーマンス向上のためのヒント, 279
クエリ内並列処理, ix
(参照 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, 547
交換アルゴリズム, 547
交換アルゴリズムの使用, 567
説明, 547
クエリに関する一般的な問題
リモート・データ・アクセス, 757
クエリの解析
リモート・データ・アクセス, 752
クエリの最初のローの検索
説明, 330
クエリの最適化, ix
(参照 オプティマイザ)
LIKE 述部, 517
アクセス・プランの解釈, 571
仮定条件, 527
サブクエリを EXISTS 述部として書き換え, 521
説明, 523
フェーズ, 510
クエリの最適化と実行
説明, 509
クエリの正規化
リモート・データ・アクセス, 752
クエリのパフォーマンス
実行プランの解釈, 571
クエリのパフォーマンスのモニタ
パフォーマンス向上のためのヒント, 260
クエリのプラン・キャッシュ・ページの統計値
説明, 254
クエリの前処理
リモート・データ・アクセス, 752
クエリのメモリ不足時方式の統計値
説明, 254
クエリのローの実体化／秒の統計値
説明, 254
クエリ・パフォーマンス
RowsReturned 統計値, 584
キャッシュの読み込み数とヒット数, 585
述部の選択性, 584
推定ソース, 584
選択性統計値, 584
データの断片化の問題, 585
有効なインデックスの不足, 585
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
説明, 113
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
説明, 115
クライアント・アプリケーションからのデータベース統計値の取得
説明, 255
クライアント側ロード
説明, 682
クライアントとサーバとの間の要求数の削減
パフォーマンス向上のためのヒント, 267
クラス
リモート・サーバ, 759
クラスタード・インデックス
インデックス・コンサルタントの推奨内容, 213
インデックス・コンサルタントの推奨内容の実装, 214
使用, 88
クラスタード・インデックスの使用
説明, 88
クラスタード・ハッシュ GROUP BY
クエリ実行アルゴリズム, 560
グラフィカルなプラン
Interactive SQL を使用したアクセス, 588
SQL 関数, 588
印刷, 582
コンテキスト別のヘルプ, 583
最適化のバイパス, 583
実行プランの解釈, 581

Copyright © 2007, iAnywhere Solutions, Inc.

述部, 587
説明, 240
単純なクエリ, 583
パフォーマンスのモニタリングと改善, 240
表示のカスタマイズ, 582
グラフィカルなプランのカスタマイズ
説明, 581
グラフ表示
パフォーマンス・モニタの使用, 243
繰り返し可能読み出し
ODBC 用の設定, 137
SELECT 文, 174
概要, 125
チュートリアル, 151
同時性の改善, 146
矛盾のケース, 134
繰り返し不可能読み出し
説明, 133
チュートリアル, 151
独立性レベル, 134
例, 153
繰り返し不可能読み出しによる矛盾
説明, 151, 174
グループ
ASE, 616
グループ化
複数のカラムの使用, 325
グループ化アルゴリズム
クエリ実行アルゴリズム, 560
グループ読み込み
テーブル, 591
グループ分けされたデータ
説明, 311
クロス・ジョイン
説明, 356
グローバル・オートインクリメント
GUID および UUID との比較, 104
グローバル・テンポラリ・テーブル
共有, 94
説明, 94
テーブル構造のマージ, 692
グローバル変数
デバッガ, 847

け

警告
プロシージャとトリガ, 820
計算カラム
インデックス, 90
検索引数可能な関数を使用したクエリ, 530
再計算, 58
制限事項, 58
説明, 56
挿入と更新, 58
トリガ, 58
計算カラム式の変更
説明, 57
計算カラムの再計算
説明, 58
計算カラムの使用
説明, 56
計算カラムの挿入と更新
説明, 58
計算値
説明, 316, 321
結果セット
Transact-SQL, 635
トラブルシューティング, 309
パーミッション, 810
ファイルへの保存, 721
複数, 811
複数回のクエリの実行, 309
プロシージャ, 784, 810
変数, 812
リモート・プロシージャ, 747
ロー数の制限, 330
結果セットの実体化
クエリ処理, 279
結果の小計
CUBE 句, 421
ROLLUP 句, 420
WITH CUBE 句, 423
WITH ROLLUP 句, 421
結果をプロシージャのパラメータとして返す
説明, 808
決定性関数
定義, 566
副次的影響, 566
現在のトレーシングの設定の確認
説明, 224
現在の日付／時刻デフォルト
説明, 101
検索引数可能な述部
説明, 529

 Copyright © 2007, iAnywhere Solutions, Inc.

検査制約
カラム, 106
削除, 108
選択, 27
テーブル, 107
ドメイン, 107
ドメインでの使用, 111
変更, 108
検査制約の変更と削除
説明, 108
検証
WITH EXPRESS CHECK を使用したテーブル, 279
XML, 669
インデックス, 91
カラム制約, 27
データベース設計, 22
限定比較テスト
サブクエリ, 472
説明, 486

こ

交換アルゴリズム
クエリ内並列処理で使用, 567
説明, 567
降順
ORDER BY 句, 329
[更新] タブ
インデックス・コンサルタントの推奨内容, 214
構成部品の問題
説明, 400
後続ブランク
Transact-SQL, 620
データベースの作成, 620
比較, 297
構築値
実行プラン内の項目, 578
構文に依存しない最適化
説明, 523
候補インデックス
インデックス・コンサルタント, 213
説明, 213
効率
改善とロック, 146
データのインポート時の時間の節約, 684
互換性
ANSI 以外のジョイン, 351
ASE, 610
ASE でのインポートとエクスポート, 723
GROUP BY 句, 338
NULL の出力, 702
Transact-SQL との互換性を維持するためのオプション設定, 621
大文字と小文字の区別, 622
サーバとデータベース, 613
互換性のある SQL 文の記述方法
説明, 626
互換性のあるクエリの記述方法
説明, 627
互換性のあるデータ型のジョイン
説明, 350
互換性のあるテーブルの作成
説明, 626
コスト
インデックス・コンサルタントの推奨内容, 214
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, 267
コストの高いユーザ定義関数の削減
パフォーマンス向上のためのヒント, 267
コストベースの最適化
説明, 523
バイパス, 511
コスト・モデル
説明, 523
コスト利益の合計
インデックス・コンサルタントの推奨内容, 214
異なるテーブル構造のマージ
説明, 692
異なるファイルの異なるデバイスへの配置
パフォーマンス向上のためのヒント, 261
コピー
INSERT を使用したデータ, 502
テーブル, 59
ビュー, 62
プロシージャ, 782
コマンド
Interactive SQL でのロード, 721
コマンド・デリミタ
設定, 827
コマンド・デリミタを変更する必要があるかどうかをチェックする

Copyright © 2007, iAnywhere Solutions, Inc.

プロシージャを作成するときのヒント, 827
コマンド・ファイル
Interactive SQL で開く, 720
[SQL 文] ウィンドウ枠, 720
概要, 720
構築, 720
作成, 720
実行, 720
出力の書き込み, 721
説明, 720
コミット
wait_for_commit, 178
コミットされた読み込み
ODBC 用の設定, 137
SELECT 文, 174
概要, 125
矛盾のケース, 134
コミットされない読み込み
ODBC 用の設定, 137
SELECT 文, 174
概要, 125
矛盾のケース, 134
コミット時の参照整合性の検査
説明, 178
混合負荷または OLAP 負荷の最適化
説明, 527

さ

再帰 UNION
アルゴリズム, 564
再帰関係
説明, 9
再帰クエリ
制限, 399
再帰サブクエリ
最短距離の問題, 406
説明, 391
データ型宣言, 404
複数の集合レベル, 395
分解部品の問題, 400
再帰テーブル
アルゴリズム, 564
再帰ハッシュ・ジョイン
クエリ実行アルゴリズム, 558
再帰左外部ハッシュ・ジョイン
クエリ実行アルゴリズム, 558
再計算
計算カラム, 58
再構築
インデックス, 92
ダウン時間の最短化, 714
ツール, 711
データベース, 706
目的, 706
再構築中のダウン時間の最短化
説明, 714
再構築ツール
dbisql ユーティリティ, 711
dbunload ユーティリティ, 711
UNLOAD TABLE 文, 712
説明, 711
再構築ツールと再ロード・ツール
説明, 711
最高のパフォーマンスのヒント
リスト, 257
最小キャッシュ・サイズ
説明, 273
最小限の管理作業
オプティマイザ, 527
最初のローの最適化または結果セット全体の最適化
説明, 527
最大
キャッシュ・サイズ, 273
最短距離の問題
説明, 406
最適化
コストベース, 523
実行プランの解釈, 571
説明, 523
最適化ゴール
実行プラン, 576
最適化時のサブクエリ変形
説明, 512
最適化に影響する仮定条件
説明, 527
最適化の手順
説明, 510
最適化のバイパス
単純なクエリ, 511
最適化フェーズ
クエリ処理, 510
最適化前フェーズ
クエリ処理, 510

 Copyright © 2007, iAnywhere Solutions, Inc.

最適化をバイパスするクエリ
説明, 511
削除
インデックス, 92
カラム・デフォルト, 101
検査制約, 108
実体化ビュー (Materialized View), 84
ディレクトリ・アクセス・サーバ, 736
データ型, 112
データベース, 42
データベース・ファイル, 42
テーブル, 50
ドメイン, 112
トリガ, 795
ビュー, 65
プロシージャ, 783
ユーザ定義データ型, 112
リモート・サーバ, 731
作成
SQL のデータベース, 33
SQL を使用したデータ型の作成, 111
SQL を使用したドメインの作成, 111
Sybase Central を使用したデータ型の作成, 110
Sybase Central を使用したドメインの作成, 110
インデックス, 90
外部呼び出しを使ったプロシージャと関数, 830
カラム・デフォルト, 100
コマンド・ラインからのデータベース, 34
実体化ビュー (Materialized View), 75
テンポラリ・プロシージャ, 780
データベース, 31
データベースのチュートリアル, 185
テーブル, 45
トリガ, 791
トレーシング・セッション, 226
ビュー, 61
プロシージャ, 780
別個のトレーシング・データベース, 234
ユーザ定義関数, 787
リモート・プロシージャ, 780
サブクエリ
ALL テスト, 473
ANY 演算子, 472
ANY テスト, 472
EXISTS 述部として書き換え, 521
GROUP BY, 469
HAVING 句, 469
IN キーワード, 299
WHERE 句, 468, 484
一般的なエラー, 466
演算子のタイプ, 470
オプティマイザ内部, 484
外部参照, 469
概要, 462
キャッシュ, 566
限定比較テスト, 470, 472
ジョイン, 466
ジョインとして書き換え, 480
ジョインへの変換, 480, 484
セット・メンバシップ・テスト, 470, 475
説明, 461, 462
相関, 479, 484
相関サブクエリ, 467
存在テスト, 470, 477
単一ローのサブクエリ, 465
トラブルシューティング, 466
ネスト, 482
ネスト解除, 514
比較, 464
比較演算子, 485
比較テスト, 470, 471
複数ローのサブクエリ, 464
複数ローを返す, 464
ロー・グループ選択, 469
ロー選択, 468
サブクエリ・セット・メンバシップ・テスト
説明, 475
サブクエリ操作
説明, 484
サブクエリと関数のキャッシュ
説明, 566
サブクエリとジョイン
説明, 480
サブクエリの使用
説明, 461
サブクエリのテスト
説明, 470
サブクエリのネスト解除
説明, 514
サブクエリ比較テスト
説明, 471
サブクエリを EXISTS 述部として書き換え
説明, 521

サブクエリを使用したデータの選択
概要, 461
サブトランザクション
セーブポイント, 123
プロシージャとトリガ, 826
サポート
ニュースグループ, xvii
算術
演算, 316
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, 54
参照整合性
アクション, 115
オーファン, 177
外部キー, 114
概要, 98
確保, 113, 114
カラム・デフォルト, 100
クライアント・アプリケーションによる違反, 115
検査, 116
コミット時の検証, 178
システム・テーブル内の情報, 117
システム・トリガ, 115
制約, 98, 106
説明, 96
喪失, 115
ツール, 97
プライマリ・キー, 608
参照整合性アクション
システム・トリガで実装, 116
参照整合性検証の遅延
説明, 178
参照整合性の確保
説明, 114
参照整合性の喪失
説明, 115
サンプル・データベース
demo.db のスキーマ, 344
サーバ
接続しないでデータベースを起動, 34
データベースの停止, 41
パフォーマンス・モニタでのグラフ表示, 243
サーバ側ロード
説明, 682
サーバ・クラス
asejdbc, 762
aseodbc, 765
db2odbc, 767
msodbc, 770
ODBC, 764, 772
oraodbc, 769
sajdbc, 761
saodbc, 765
説明, 728
定義, 727
サーバ上のリモート・テーブルのリスト
説明, 732
サーバとデータベース
互換性, 613
サーバの機能
リモート・データ・アクセス, 752
サーバの状態
インデックス・コンサルタント, 214
[サーバ・メッセージと実行された SQL] ウィンドウ枠
説明, 38

し

時間
プロシージャとトリガ, 828
時間節約の方式
データのインポート, 684
式
NULL 値, 305
OLAP 集合関数, 458
式 SQL
実行プラン内の項目, 579
識別子
大文字と小文字の区別, 622
修飾, 285
ドメインでの使用, 111
ユニーク, 623
システム・オブジェクト
内容の表示, 37
システム・カタログ
ASE, 614
システム関数
tsequal, 624
システム管理者
ASE, 615
システム障害
トランザクション, 495

 Copyright © 2007, iAnywhere Solutions, Inc.

システム・セキュリティ担当者
ASE, 615
システム・テーブル
ASE, 614
Transact-SQL の名前の重複, 620
参照整合性についての情報, 117
所有者, 615
内容の表示, 37
ビュー, 71
システム・テーブル・データの表示
説明, 71
システム・テーブルの整合性ルール
説明, 117
システム・トリガ
参照整合性アクションの実装, 116
参照整合性の実行, 115
プロファイリング結果の生成と確認, 204
システム・ビュー
インデックス, 93
参照整合性についての情報, 117
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, 237
プロファイリング結果の生成と確認, 204
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
説明, 237
システム・プロシージャを使用したプロファイリング情報の取り出し
説明, 239
事前フィルタ・アルゴリズム
説明, 568
実行
SQL スクリプト, 720
コマンド・ファイル, 720
複数回のクエリ, 309
実行フェーズ
クエリ処理, 511
実行プラン
Interactive SQL を使用したアクセス, 588
SQL 関数, 588
印刷, 582
キャッシュ, 544
グラフィカルなプラン, 581
コンテキスト別のヘルプ, 583
省略形, 572
長いテキスト・プラン, 581
ビュー・マッチングの結果, 576
表示のカスタマイズ, 582
短いテキスト・プラン, 580
実行プランに使用される省略形
説明, 572
実行プランの解釈
主な統計, 584
説明, 571
実行プランへのアクセス
説明, 588
実際のロックの重要性についてのチュートリアル
説明, 162
実体化ビュー (Materialized View)
COSTED ビュー・マッチングの原因, 576
暗号化と復号化, 79
オプション設定の決定, 86
オプティマイザが検討したかどうかを判断する, 533
オプティマイザでの古さ閾値の設定, 83
オプティマイザの考慮事項, 74
隠す, 84
カラム統計, 72
関連するタスク, 72
最適化での使用の有効化と無効化, 82
削除, 84
作成, 75
実体化ビュー (Materialized View) によるパフォーマンスの向上, 531
使用時の考慮事項, 73
使用するかどうかの評価, 531
情報の取得, 85
初期化, 77
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, 128
接続とオプションの不一致, 532
接続の候補リストの決定, 532
説明, 72
ディスク領域の考慮事項, 72
データの移植, 77
データの同時実行性と一貫性, 72
データベース・オプションの考慮事項, 74
ビュー・マッチング・アルゴリズムによる検査, 534
プランのキャッシュ, 544
保守コスト, 72
無効化, 80
有効化, 80

リフレッシュ, 78
実体化ビュー (Materialized View) の削除
説明, 84
実体化ビュー (Materialized View) の作成
説明, 75
実体化ビュー (Materialized View) の情報の取得
説明, 85
実体化ビュー (Materialized View) の初期化
説明, 77
実体化ビュー (Materialized View) の編集
説明, 72
実体化ビュー (Materialized View) の有効化と無効化
説明, 80
実体化ビュー (Materialized View) のリフレッシュ
説明, 78
実体化ビュー (Materialized View) を隠す
説明, 84
実体化ビュー (Materialized View) を管理するときの制限
説明, 74
指定カラムへの値の挿入
説明, 500
自動ジョイン
外部キー, 608
射影
定義, 288
集合関数
Adaptive Server Enterprise との互換性, 338
ALL キーワード, 316
DISTINCT キーワード, 319
GROUP BY 句, 321
NULL, 319
OLAP, 433
OLAP に等価な式, 458
ORDER BY と GROUP BY, 331
ウィンドウ (OLAP), 432
外部参照, 317
概要, 311
グループ分けされたデータに対する適用, 312
スカラ集合, 317
説明, 316
データ型, 318
複数レベル, 395
ベクトル集合, 321
集合関数に伴う GROUP BY の使用
説明, 321
集合関数を使用したクエリ結果の要約
説明, 316
集合関数を使用できる場所
説明, 317
集合操作
NULL, 335
説明, 332
ルール, 333
修飾
説明, 296
修飾された名前
データベース・オブジェクト, 285
修正
依存性があるビュー, 63
カラム・デフォルト, 101
ビューの依存性があるテーブル, 46
集約
実行プラン内の項目, 579
述部
IN リストの最適化, 516
LIKE の最適化, 517
概要, 296
実行プランでの解釈, 587
実行プラン内の項目, 578
説明, 529
パフォーマンス, 529
分析, 529
述部の推定による利用可能な条件の発見
説明, 519
述部の分析
説明, 529
出力
NULL, 702
エクスポート・ツール, 694
クエリ結果のエクスポート, 700
データのエクスポート, 694
データベースのエクスポート, 703
テーブルのエクスポート, 704
出力リダイレクション
説明, 700
順序付き DISTINCT
クエリ実行アルゴリズム, 559
順序付き GROUP BY
クエリ実行アルゴリズム, 561
順序付き GROUP BY SETS
クエリ実行アルゴリズム, 562
ジョイン

 Copyright © 2007, iAnywhere Solutions, Inc.

2 つのテーブル, 349
3 つ以上のテーブル, 350
ANSI 以外のジョイン, 351
DELETE 文、UPDATE 文、INSERT 文, 351
FROM 句, 347
NULL 入力テーブル, 358
ON 句, 352
Transact-SQL, 629
Transact-SQL 外部ジョインと NULL 値, 364
Transact-SQL 外部ジョインとビュー, 364
Transact-SQL 外部ジョインの制限, 363
WHERE 句, 354
外部, 358
外部ジョインから内部ジョインへの変換, 517
概要, 342
カンマ, 356
カーソルの更新, 351
キー, 608
キー・ジョイン, 377
クエリ実行アルゴリズム, 552
クロス・ジョイン, 356
異なるデータベースのテーブル, 745
サブクエリ, 466
サブクエリからジョインへの変換, 484
サブクエリの変換, 480
自動, 608
ジョインしたテーブル, 348
ジョイン条件, 347
ジョインの削除によるリライト最適化, 518
スター・ジョイン, 366
説明, 347
セルフジョイン, 365
探索条件, 354
抽出テーブル, 372
重複する相関名, 366
直積, 356
デフォルトは KEY JOIN, 348
データ型変換, 350
テーブル式, 350
等価ジョイン, 354
内部, 358
内部ジョインの計算方法, 349
内部と外部, 358
ナチュラル, 608
ナチュラル・ジョイン, 373
ネスト, 350
保護されたテーブル, 358
リモート・テーブル, 743
ジョイン・アルゴリズム
説明, 552
ネスト・ループ・ジョインの変形, 552
ハッシュ・ジョインの変形, 552
マージ・ジョインの変形, 552
ジョイン演算子
Transact-SQL, 629
ジョイン条件
種類, 354
ジョイン条件に対する WHERE 句の使用
説明, 354
ジョイン操作
説明, 346
ジョインで重複する相関名 (スター・ジョイン)
説明, 366
ジョインの互換性
説明, 629
ジョイン：複数テーブルからのデータ検索
説明, 341
ジョイン方式の区切りにコロンを使用
説明, 580
上位 N のソート
クエリ実行アルゴリズム, 565
使用可能 IO の統計値
説明, 254
消去
検査制約, 108
データベース, 42
テーブル, 50
消去ユーティリティ [dberase]
使用, 44
条件
GROUP BY 句, 313
パターン一致, 299
論理演算子との接続, 305
詳細情報の検索／フィードバックの提供
テクニカル・サポート, xvii
昇順
ORDER BY 句, 329
初期化
データベース, 31
初期化ユーティリティ [dbinit]
使用, 34
初期キャッシュ・サイズ
説明, 273
診断トレーシング

運用データベース, 216
情報の解釈, 229
設定, 218
説明, 216
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, 226
トレーシング・セッションの作成, 226
トレーシング・タイプ, 221
トレーシング・データベース, 216
トレーシングに関連するデータベースのプロパティ, 218
トレーシングの条件, 223
トレーシングのスコープ, 220
トレーシングの設定, 225
トレーシングの設定の確認, 224
トレーシング・レベル, 219
診断トレーシング条件
説明, 223
診断トレーシング情報の分析
説明, 229
診断トレーシングの条件
説明, 223
診断トレーシングのスコープ
説明, 220
診断トレーシングの設定
説明, 218
診断トレーシングのタイプ
説明, 221
診断トレーシングのレベル
説明, 219
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
説明, 216

す

スカラ集合
説明, 317
スキーマ
エクスポート, 712
ロック, 168
スキーマ・ロック
共有, 168
説明, 168
スクリプト
Interactive SQL で実行, 720
コマンド・ファイルの作成, 720
コマンド・ファイルの説明, 720
コマンド・ファイルのロード, 721
スケジュール
直列可能, 144
直列可能性の影響, 144
直列可能とロックの早期解放, 180
直列不可能なスケジュールの影響, 144
スコープ
診断トレーシング, 220
スター・ジョイン
説明, 366
ストアド・プロシージャ
FROM 句内での使用, 295
Transact-SQL ストアド・プロシージャの概要, 631
共通テーブル式を含む, 395
デバッグ, 841
プロファイリング結果の生成と確認, 204
ストアド・プロシージャ言語
概要, 631
ストアド・プロシージャの自動変換
説明, 634
スナップショット・アイソレーション
SELECT 文のロック, 176
意図的ロック, 170
更新の競合の回避, 133
実体化ビュー (Materialized View) マッチング, 128
説明, 127
トランザクション, 129
トランザクション内のレベルの変更, 139
パフォーマンス上の重要事項, 143
有効化, 129
レベルの選択, 143
ロー・バージョン, 128
スナップショット・アイソレーションの有効化
説明, 129
スナップショット・アイソレーションを使用した繰り返し不可能読み出しの回避
説明, 155
スナップショット・アイソレーションを使用したダーティ・リードの回避
説明, 149
スナップショット数の統計値
説明, 253
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, 141

スワップ領域
データベース・キャッシュ, 276

せ

正規化
説明, 15
パフォーマンス向上, 261
正規形
第 1 正規形, 17
第 2 正規形, 18
第 3 正規形, 19
データベース設計の正規化, 15
制御文
リスト, 802
制限
定義, 288
リモート・データ・アクセス, 756
整合性
確保, 113
カラム・デフォルト, 100
検査, 116
システム・テーブル内の情報, 117
整合性確保のためのトリガの使用, 98
整合性制約の実装, 98
制約, 106
説明, 96
喪失, 115
ツール, 97
整合性制約を実装するための SQL 文
説明, 98
生成
物理データ・モデル, 19
ユニーク・キー, 181
生成されたジョイン条件
説明, 347
生成されたジョインと ON 句
説明, 353
制約
Sybase Central, 108
一意性制約, 108
概要, 97
カラムとテーブル, 27
検査制約, 107
制約の宣言
パフォーマンス向上のためのヒント, 259
制約の選択
説明, 27
セキュリティ
オブジェクトを隠す, 835
プロシージャ, 779
要求ログ, 235
設計
データベース, 3
設計プロセス
説明, 11
接続
実体化ビュー (Materialized View) の候補, 532
接続しないでデータベースを起動, 34
デバッガ, 848
デバッグ, 840
データベース, 35
リモート, 750
ループバック, 745
接続オプション
実体化ビュー (Materialized View) に対する影響, 74
接続しないでデータベースを起動する
説明, 34
接続数の統計値
説明, 254
接続の削除
リモート・データ・アクセス, 758
接続の切断
データベース, 40
別のユーザとデータベースの接続, 40
接続の問題点
リモート・データ・アクセス, 756
接続ロック
期間, 167
切断
データベースの停止, 41
設定
診断トレーシング, 218
診断トレーシングの設定, 225
トレーシング・レベル, 225
セット・メンバシップ・テスト
=ANY, 475
説明, 488
否定, 475
セマンティック・クエリ変形
説明, 512
セマンティック変形
説明, 512
セマンティック変形の種類

説明, 512
セミコロン
コマンド・デリミタ, 827
セミジョイン
クエリ実行アルゴリズム, 555, 557
セルフジョイン
説明, 365
全外部ジョイン
説明, 358
線形回帰関数
OLAP, 448
選択性
実行プランでの解釈, 587
実行プラン内の項目, 578
実行プランの解釈, 584
選択性推定
実行プランでの解釈, 587
部分インデックス・スキャンの使用, 599
選択性統計値
説明, 584
全比較
アクセス・プラン内の統計, 574
セーブポイント
トランザクション内, 123
ネスト, 123
プロシージャとトリガ, 826
命名, 123
セーブポイントのネスト
説明, 123
セーブポイントの命名
説明, 123

そ

相関関数
OLAP, 448
相関サブクエリ
外部参照, 479
説明, 478, 484, 521
定義, 467
相関名
共通テーブル式での使用, 393
スター・ジョイン, 366
制限, 295
説明, 378
セルフジョイン, 365
テーブル名, 295
相対利益
インデックス・コンサルタントの推奨内容, 214
挿入されたローに対するディスク割り付け
説明, 590
挿入ロック
説明, 173
総利益
インデックス・コンサルタントの推奨内容, 214
属性
SQLCA.lock, 137
クエリ結果を XML として取得, 653
選択, 13
定義, 5
測定値
実行プラン内の項目, 578
速度が遅い文の検出
説明, 229
その他の診断ツールと方法
説明, 235
その他の統計値
リスト, 254
存在テスト
説明, 477
否定, 478
ソース・コード
ブレークポイントの設定, 845
ソート
インデックスの使用, 267
クエリ実行アルゴリズム, 565
クエリの結果, 308
ソート GROUP BY SETS
クエリ実行アルゴリズム, 562
ソート順
ORDER BY 句, 329

た

第 1 正規形
テスト, 17
第 2 正規形
テスト, 18
第 3 正規形
テスト, 19
待機
参照整合性の検証, 178
ロックされたローへのアクセス, 155
タイミング・ユーティリティ

Copyright © 2007, iAnywhere Solutions, Inc.

説明, 242
代理ロー
説明, 178
多対多の関係
解析, 19, 21
定義, 7
多対多の関係をエンティティに変更
説明, 9
正しいカーソル・タイプの指定
パフォーマンス向上のためのヒント, 268
他の SQL ダイアレクトとの違い
概要, 603
単一の GROUP BY
クエリ実行アルゴリズム, 562
単一の XML GROUP BY
クエリ実行アルゴリズム, 562
単一ローのサブクエリ
説明, 465
探索条件
GROUP BY 句, 313
サブクエリ, 462
パターン一致, 299
日付の比較, 306
単純なクエリ
オプティマイザのバイパス, 511
グラフィカルなプランに表示されない, 583
定義, 511
断片化
インデックス, 266
説明, 264
テーブル, 265
ファイル, 264
ファイル、テーブル、インデックス, 264
断片化の削減
パフォーマンス向上のためのヒント, 264
ダーティ・リード
クエリ中のロック, 174
チュートリアル, 147
独立性レベル, 134
矛盾, 133

ち

小さいテーブルに関する統計収集
パフォーマンス向上のためのヒント, 259
チェックポイントの緊急度の統計値
説明, 247
チェックポイントの統計値
リスト, 247
チェックポイント／秒の統計値
説明, 247
チェックポイント・フラッシュ／秒の統計値
説明, 247
チェックポイント・ログ
パフォーマンス, 261
チェックポイント・ログの書き込み／秒の統計値
説明, 247
チェックポイント・ログの使用ページ数の統計値
説明, 247
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, 247
チェックポイント・ログの統計値
説明, 247
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, 247
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, 247
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, 247
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, 247
チェックポイント・ログのページ再配置の統計値
説明, 247
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, 247
チェックポイント・ログのログ・サイズの統計値
説明, 247
逐次スキャン
説明, 552
ディスク割り付けとパフォーマンス, 590
逐次テーブル・スキャン
説明, 552
ディスク割り付けとパフォーマンス, 590
中央値
CUME_DIST を使用した計算, 455
注釈フェーズ
クエリ処理, 510
抽出
SQL Remote のデータベース, 715

Copyright © 2007, iAnywhere Solutions, Inc.

抽出テーブル
FROM 句内での使用, 295
外部ジョイン, 361
キー・ジョイン, 385
ジョイン, 372
抽出テーブル・アルゴリズム, 567
ナチュラル・ジョイン, 376
抽出テーブル・アルゴリズム
説明, 567
チュートリアル
繰り返し不可能読み出し, 151
実際のロックの意味, 162
ダーティ・リード, 147
デバッガ, 839
デバッガの使用開始, 839
データベースの作成, 187
独立性レベル, 147
幻ロー, 157
ロックの意味, 162
長期間の読み込みロック
説明, 169
重複結果
削除, 294
重複したクエリ結果の削除
説明, 294
重複するロー
UNION による削除, 332
重複排除
クエリ実行アルゴリズム, 558
直積
説明, 356
直接参照
データベース・オブジェクト, 71
直列可能
ODBC 用の設定, 137
SELECT 文, 174
概要, 125
スケジュール, 144
同時性の改善, 146
矛盾のケース, 134
直列可能なスケジュール
影響, 144
説明, 144
ロックの早期解放, 180
直列不可能なトランザクションのスケジューリング
影響, 144

つ

追加
データベースへのデータの追加, 684
通信の空きバッファの統計値
説明, 248
通信の失敗した送信／秒の統計値
説明, 248
通信の受信されるマルチパケット数／秒の統計値
説明, 248
通信の受信バイト数／秒の統計値
説明, 248
通信の受信パケット数／秒の統計値
説明, 248
通信の受信要求数の統計値
説明, 248
通信の送信されるマルチパケット数／秒の統計値
説明, 248
通信の送信バイト数／秒の統計値
説明, 248
通信の送信パケット数／秒の統計値
説明, 248
通信の総バッファ数の統計値
説明, 248
通信の統計値
リスト, 248
通信の無圧縮受信バイト数／秒の統計値
説明, 248
通信の無圧縮受信パケット数／秒の統計値
説明, 248
通信の無圧縮送信バイト数／秒の統計値
説明, 248
通信の無圧縮送信パケット数／秒の統計値
説明, 248
通信のユニークなクライアント・アドレスの統計値
説明, 248
常にトランザクション・ログを使用する
パフォーマンス向上のためのヒント, 257
ツール
データのアンロード, 698
データのインポート, 685
データのエクスポート, 694
データの再ロード, 711
データベースの再構築, 711

て

定数式デフォルト

説明, 105
ディスク I/O の統計値
リスト, 249
ディスク・アクセスのコスト・モデル
説明, 523
ディスク書き込みのアクティブの最大値の統計値
説明, 250
ディスク書き込みのアクティブの統計値
説明, 250
ディスク書き込みのコミット・ファイル／秒の統計値
説明, 250
ディスク書き込みのテンポラリ拡張／秒の統計値
説明, 250
ディスク書き込みのデータベース拡張／秒の統計値
説明, 250
ディスク書き込みの統計値
リスト, 250
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, 250
ディスク書き込みのページ／秒の統計値
説明, 250
ディスクのアクティブ I/O の統計値
説明, 249
ディスクの最大 I/O の統計値
説明, 249
ディスク読み込みのアクティブの最大値の統計値
説明, 250
ディスク読み込みのアクティブの統計値
説明, 250
ディスク読み込みのインデックス内部／秒の統計値
説明, 250
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, 250
ディスク読み込みの合計ページ数／秒の統計値
説明, 250
ディスク読み込みのテーブル／秒の統計値
説明, 250
ディスク読み込みの統計値
リスト, 250
ディレクトリ・アクセス・サーバ
削除, 736
作成, 734
説明, 734
プロキシ・テーブルの削除, 736
変更, 736
ディレクトリ・アクセス・サーバの削除
説明, 736
ディレクトリ・アクセス・サーバの作成
説明, 734
[ディレクトリ・アクセス・サーバの作成] ウィザード
使用, 734
ディレクトリ・アクセス・サーバの使用
説明, 734
テキスト・プラン
実行プランの解釈, 580
適切なデータ型の使用
パフォーマンス向上のためのヒント, 270
適切なハードウェアの入手
パフォーマンス向上のためのヒント, 257
適切なページ・サイズの使用
パフォーマンス向上のためのヒント, 269
テクニカル・サポート
ニュースグループ, xvii
テスト
データベースの検証, 22
データベースの正規化, 15
デッドロック
アプリケーション・プロファイリング・モードを使用した調査, 232
診断, 141
説明, 140
トランザクションのブロック, 141
理由, 141
レポート, 141
デッドロックの調査
説明, 232
デッドロック・レポート
説明, 141
デバイス
管理, 613
デバッガ
SOAP 関数, 838
稼働条件, 839
起動, 840
機能, 838
ストアド・プロシージャのデバッグ, 841
接続, 840
接続の活用, 848

説明, 837
チュートリアル, 840
はじめに, 840
ブレークポイントの活用, 845
変数の検査, 847
デバッグ
SOAP プロシージャ, 838
SQL Anywhere のデバッガの使用, 837
稼働条件, 839
ストアド・プロシージャ, 841
説明, 837
チュートリアル, 841
パーミッション, 839
デフォルト
GLOBAL AUTOINCREMENT, 103
INSERT 文, 500
NEWID, 104
NULL, 105
Sybase Central で作成, 101
Transact-SQL, 614
オートインクリメント, 102
概要, 97
カラム, 100
現在の日付と時刻, 101
作成, 100
定数式, 105
ドメインでの使用, 111
トランザクションとロック, 181
文字列と数値, 105
ユーザ ID, 102
デフラグメンテーション
説明, 264
データベース内の個々のテーブル, 265
データベース内のすべてのテーブル, 265
ハード・ディスク, 264
デベロッパー・コミュニティ
ニュースグループ, xvii
典型的な矛盾のケース
説明, 133
テンポラリ・テーブル
Transact-SQL との互換性, 627
インデックス, 595
クエリ処理中のワーク・テーブル, 279
作成の概要, 45
説明, 94
データのインポート, 691
テーブル構造のマージ, 692
非トランザクション指向にする, 94
非トランザクション指向の利点, 94
ローカルとグローバル, 94
テンポラリ・テーブルの編集
説明, 94
テンポラリ・テーブル・ページの統計
説明, 254
テンポラリ・ファイル
ワーク・テーブル, 261
テンポラリ・プロシージャ
作成の概要, 780
データ
Interactive SQL での編集, 51
Sybase Central での編集, 51
XML としてエクスポート, 644
一貫性, 133
インポート, 684
インポートとエクスポート, 682
エクスポート, 694
エクスポート・ツール, 694
大文字と小文字の区別, 622
検索, 88
実体化ビュー (Materialized View) のリフレッシュ, 78
実体化ビュー (Materialized View) への移植, 77
整合性と正当性, 144
追加、変更、削除, 493
データの修正に必要なパーミッション, 494
表示, 51
無効, 96
データ一貫性
実際のロックの意味, 162
スナップショット・アイソレーション, 176
独立性レベル 0, 174
データ型
EXCEPT 文, 332
INTERSECT 文, 332
SQL と C, 834
SQL を使用した作成, 111
Sybase Central を使用した作成, 110
Transact-SQL の timestamp, 623
UNION 文, 332
カラムへの割り当て, 110
再帰サブクエリ, 404
削除, 112
集合関数, 318
選択, 24

 Copyright © 2007, iAnywhere Solutions, Inc.

ユーザ定義, 110
リモート・プロシージャ, 748
データ型の変換
DB2, 767
ODBC と ASE, 766
Oracle, 769
データ型変換
Microsoft SQL Server, 771
データが有効でなくなる場合
説明, 96
データ修正のパーミッション
説明, 494
データ修正文
説明, 494
データ整合性
確保, 95, 113
カラム制約, 27
カラム・デフォルト, 100
検査, 116
システム・テーブル内の情報, 117
制約, 98, 106
説明, 96
喪失, 115
直列不可能なスケジュールの影響, 144
ツール, 97
データ整合性の確保
説明, 95
データ・ソース
外部サーバ, 764
[データ] タブ
SQL Anywhere プラグイン, 51
[データ] タブ
リモート・テーブルに関する制限, 727
データ定義言語 (参照 DDL)
データ定義文 (参照 DDL)
データ入力
独立性レベル, 145
[データのアンロード] ダイアログ
使用, 699
[データのアンロード] ダイアログの使用
説明, 699
データの一貫性
ISO SQL 標準, 133
繰り返し可能読み出し, 133
繰り返し可能読み出しとロック, 174
繰り返し可能読み出しのチュートリアル, 151
正当性, 144
ダーティ・リード, 133
ダーティ・リードとロック, 174
ダーティ・リードのチュートリアル, 147
幻ロー, 133
幻ローとロック, 175
幻ローのチュートリアル, 157
ロックを使用して保証, 166
データの移動
インポート, 684
インポートとエクスポート, 682
エクスポート, 694
説明, 681
パフォーマンス, 682
データのインポート
DEFAULTS オプション, 692
INPUT 文, 687
INSERT 文, 689
LOAD TABLE 文, 688, 690
NULL 値を含む, 692
一致しないテーブル構造, 692
[インポート] ウィザード, 685
インポート／エクスポートの状況, 682
概要, 681
考慮事項, 684
説明, 682
他のデータベースから, 689
ツール, 685
テンポラリ・テーブル, 691, 692
データベースのバックアップ, 683
データベースへのデータのインポート, 684
パフォーマンス, 682
パフォーマンスのヒント, 684
プロキシ・テーブル, 689
変換エラー, 690
データのインポートとエクスポート
説明, 681
データのインポートのパフォーマンスのヒント
説明, 684
データのエクスポート
dbunload ユーティリティ, 697
Interactive SQL ウィザード, 694
OUTPUT 文, 695
UNLOAD TABLE 文, 696
UNLOAD 文, 697
概要, 681
考慮事項, 694
スキーマ, 712

説明, 682, 694
ツール, 694
データベースのバックアップ, 683
データの削除
DELETE 文, 507
TRUNCATE TABLE 文, 508
データの出力
インポートとエクスポート, 682
ファイル, 697
ファイルへの保存, 721
データの挿入
BLOB, 502
INPUT 文, 687
INSERT の使用, 499
INSERT 文, 689
SELECT の使用, 501
カラム・データ INSERT 文, 500
指定されていないカラムの動作, 500
制約, 500
全カラム, 499
デフォルト, 500
データの追加
BLOB, 502
INSERT の使用, 499
SELECT の使用, 501
カラム・データ INSERT 文, 500
制約, 500
説明, 499
全カラム, 499
デフォルト, 500
データの追加、変更、削除
説明, 493
データの転送
INPUT 文, 687
INSERT 文, 689
LOAD TABLE 文, 688
OUTPUT 文, 695
UNLOAD TABLE 文, 696
UNLOAD 文, 697
インポート, 684
[インポート] ウィザード, 685
インポートとエクスポート, 682
説明, 681
パフォーマンス, 682
パフォーマンスのヒント, 684
変換エラー, 690
データの入力
INPUT 文, 687
INSERT 文, 689
LOAD TABLE 文, 688
インポートとエクスポート, 682
データの変更
UPDATE 文, 504
パーミッション, 494
複数のテーブルを使用したデータの更新, 505
データの編集
Sybase Central, 51
データの要約
概要, 311
データのロード
変換エラー, 690
データ編成
物理的, 590
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, 620
jConnect メタデータ・サポートのインストール, 40
SQL Anywhere への移行, 716
SQL Remote の抽出, 715
SQL からの作成, 33
SQL からの初期化, 33
Sybase Central からの作成, 32
Sybase Central からの初期化, 32
Transact-SQL との互換性, 619
XML のインポート, 645
XML の格納, 643
アンロード, 703
アンロードと再ロード, 714
エクスポート, 703
大文字と小文字の区別, 622
オブジェクトを使用した作業, 29
オプションの設定, 36
コマンド・ラインからの作成, 34
コマンド・ラインからの消去, 44
再ロード, 713
作業, 31
削除, 42
作成, 31, 185
システム・オブジェクトの表示, 37
消去, 42
初期化, 31
正規化, 15
設計, 3

Copyright © 2007, iAnywhere Solutions, Inc.

設計概念, 5
設計の検証, 22
接続しないで起動, 34
停止, 41
データベースとの接続の切断, 40
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, 706
データベースへの接続, 35
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, 709
同期に関連しないデータベースの再構築, 709
同期に関連するデータベースのアンロードと再ロード, 710
統合データベースの設定, 37
トランザクション・ログ, 31
ファイルの互換性, 31
複数のデータベースのテーブルのジョイン, 745
プロパティの表示と編集, 36
[データベース・アップグレード] ウィザード
jConnect メタデータ・サポートのインストール, 40
使用, 39
[データベース・アンロード] ウィザード
使用, 698, 703
[データベース移行] ウィザード
説明, 716
[データベース移行] ウィザードの使用
説明, 716
データベース・オブジェクト
間接参照, 71
直接参照, 71
プロパティの編集, 36
データベース・オブジェクトの使用
説明, 29
データベース・オブジェクトのプロパティの設定
説明, 36
データベース・オプション
Transact-SQL 互換性のための設定, 621
インデックス・コンサルタント, 214
実体化ビュー (Materialized View) に対する影響, 74
データベース・オプションの設定
説明, 36
データベース管理者
役割, 615
[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, 619
使用, 32
[データベース消去] ウィザード
使用, 43
データベース・スレッド
ブロック, 141
データベース設計の概念
説明, 5
[データベース抽出] ウィザード
SQL Remote, 715
データベースでのデバッグ論理
説明, 837
データベース・テーブル・プロパティの設計
説明, 24
データベース統計
説明, 245
データベースとの間でのデータの転送
説明, 682
データベースとの接続の切断
説明, 40
[データベース・トレーシング] ウィザード
使用, 226
データベース内の整合性制約
説明, 96
データベース内またはデータベース間でのテーブルまたはカラムのコピー
説明, 59
データベースにおける XML の使用
説明, 641
データベースのアップグレード
Sybase Central, 39
説明, 707
データベースのアンロード
Sybase Central, 698
カンマ区切りの ASCII フォーマット, 711
使用, 703
説明, 706
データベースの移行
sa_migrate ストアド・プロシージャ, 717
説明, 716
[データベース移行] ウィザード, 716
データベースのエクスポート
使用, 703
データベースの更新
概要, 493
データベースの再構築

Mobile Link, 710
UNLOAD TABLE 文, 712
エクスポートのコマンド, 706
考慮事項, 708
コマンド・ライン, 713
説明, 706
ツール, 711
テーブルの断片化の削減, 265
同期に関連するデータベースでの dbunload の使用, 710
パフォーマンス向上のためのヒント, 263
非レプリケート・データベース, 709
理由, 707
レプリケート・データベース, 710
データベースの再編成
テーブルの断片化の削減, 265
データベースの再ロード, ix
(参照 データベースの再構築)
コマンド・ライン, 713
説明, 706
データベースの作成
説明, 31
[データベース作成] ウィザード, 32
データベースのシステム・オブジェクトの表示
説明, 37
データベースの消去
Sybase Central, 43
説明, 42
データベースの初期化
Sybase Central, 32
データベースの設計
概念, 5
説明, 3
手順, 11
データベースの停止
説明, 41
データベースのデータが変わる場合
説明, 97
データベースのデータのエクスポート
説明, 694
データベースのパフォーマンスのモニタリング
説明, 243
データベースのブラウズ
独立性レベル, 145
データベースの編集
説明, 31
データベース・ファイル
断片化, 264
パフォーマンス, 261
データベースへの BLOB の格納
説明, 25
データベースへの接続
説明, 35
データベースへのデータのインポート
説明, 684
データベース・ページ
インデックス・コンサルタントの推奨内容, 213
データベースを再構築する理由
説明, 707
データ・モデリング
説明, 3
データ・モデルの正規化
説明, 15
データ・リカバリ
インポートとエクスポート, 683
トランザクション, 495
テーブル
Interactive SQL でデータを編集する, 51
SQL を使用した外部キーの追加, 55
SQL を使用したプライマリ・キーの追加, 52
SQL を使用したプロキシ・テーブルの作成, 741, 742
SQL を使用した変更, 48
Sybase Central からのアンロード, 699
Sybase Central でデータを編集する, 51
Sybase Central での外部キーの追加, 54
Sybase Central でのプライマリ・キーの追加, 52
Sybase Central でのプライマリ・キーの表示, 52
Sybase Central を使用したプロキシ・テーブルの作成, 740
Sybase Central を使用した変更, 47
Transact-SQL と互換性のあるテーブルの作成, 626
位置ロック, 171
インポート, 690
エクスポート, 704
外部キーの管理, 53, 54, 55
書き込みを意図したロック, 171
カラム名, 24
共有ロック, 171
クエリでの命名, 295

グループ読み込み, 591
検査制約, 108
コピー, 59
削除, 50
作成, 45
システム・テーブルの内容の表示, 37
処理, 45
制約, 27
相関名, 295
挿入ロック, 173
断片化, 265
データのエクスポート, 712
データのブラウズ, 51
データの編集, 51
データベース内の個々のテーブルのデフラグメンテーション, 265
データベース内のすべてのテーブルのデフラグメンテーション, 265
テーブル制約の管理, 108
排他ロック, 171
ビットマップ, 591
ビューの依存性, 50
複数のデータベースからのジョイン, 745
プライマリ・キーの管理, 51, 52
プロキシ, 739
プロパティ, 24
別のテーブルからの参照を表示, 54
変更時の考慮事項, 46
幻ロック, 172
リモート・アクセス, 727
リモートのリスト, 732
ロック, 170
ローのコピー, 502
ワーク・テーブル, 279
テーブル・アクセス・アルゴリズム
説明, 549
テーブルからのすべてのカラムの選択
説明, 287
テーブルからのすべてのローの削除
説明, 508
テーブルからの特定カラムの選択
説明, 288
テーブル検証時の WITH EXPRESS CHECK オプションの使用
パフォーマンス向上のためのヒント, 279
テーブル構造の正規化
パフォーマンス向上のためのヒント, 261
テーブル・サイズ
説明, 591
パフォーマンスの考慮事項, 591
[テーブル作成] ウィザード
使用, 45
テーブル式
キー・ジョイン, 381
構文, 347
ジョイン方法, 350
テーブル・スキャン
説明, 552
ディスク割り付けとパフォーマンス, 590
ハッシュ・テーブル・スキャン・アルゴリズム, 549
並列テーブル・スキャン・アルゴリズム, 551
テーブル制約とカラム制約の使い方
説明, 106
テーブル・データまたはテーブル・スキーマのエクスポート
説明, 712
テーブルとページのサイズ
説明, 591
テーブル内のカラムの順序の確認
パフォーマンス向上のためのヒント, 261
テーブル内のデータのブラウズ
説明, 51
テーブルに対する検査制約の使い方
説明, 107
テーブルのインポート
説明, 690
テーブルのエクスポート
スキーマ, 712
説明, 704
テーブルの削除
説明, 50
テーブルの作成
説明, 45
テーブルの断片化
説明, 265
テーブルの断片化の削減
パフォーマンス向上のためのヒント, 265
テーブルの断片化レベルの判断
説明, 265
テーブルの変更
説明, 46
テーブルの編集
説明, 45

テーブル幅の縮小
パフォーマンス向上のためのヒント, 267
テーブル・ヒント
対応する独立性レベル, 125
テーブルへのアクセス
テーブル・アクセス・アルゴリズム, 549
テーブル名
識別, 285
プロシージャでの完全修飾, 827
プロシージャとトリガ, 827
[テーブル名のルックアップ] ダイアログ
テーブル・リストの表示, 342
テーブル・ロック
位置, 171
書き込みを意図した, 171
競合, 170
共有, 171
説明, 166, 170
挿入, 173
排他, 171
幻, 172
[テーブル・ロック] タブ
Sybase Central, 167

と

等価ジョイン
説明, 354
同期
データベースの再構築, 710
同期に関連しないデータベースの再構築
説明, 709
同期に関連するデータベースの再構築
説明, 710
統計, ix
(参照 ヒストグラム)
ProcCall アルゴリズム, 569
アイドル・アクティブ／秒, 247
アイドル書き込み／秒, 247
アイドル・チェックポイント時間, 247
アイドル・チェックポイント／秒, 247
アクセス・プラン, 574
インデックス, 251
インデックスの完全比較／秒, 251
インデックスの追加／秒, 251
インデックスのルックアップ／秒, 251
カラム統計の更新, 526
カーソル, 253
キャッシュ, 246
キャッシュ・サイズの現在の値, 246
キャッシュ・サイズの最小値, 246
キャッシュ・サイズの最大値, 246
キャッシュ・サイズのピーク値, 246
キャッシュ置換の合計ページ数／秒, 246, 251
キャッシュのスカベンジ, 251
キャッシュのスカベンジ・アクセス, 251
キャッシュのパニック, 251
キャッシュ・ヒット／秒, 246
キャッシュ・ページの空き, 251
キャッシュ・ページの固定, 251
キャッシュ・ページのファイル, 251
キャッシュ・ページのファイル・ダーティ, 251
キャッシュ・ページの割り当て構造体, 251
キャッシュ読み込みのインデックス内部／秒, 246
キャッシュ読み込みのインデックス・リーフ／秒, 246
キャッシュ読み込みの合計ページ数／秒, 246
キャッシュ読み込みのテーブル／秒, 246
クエリのプラン・キャッシュ・ページ, 254
クエリのメモリ不足時方式, 254
クエリのローの実体化／秒, 254
実行プラン, 571
使用可能 IO, 254
スナップショット数, 253
接続数, 254
その他, 254
チェックポイントとリカバリ, 247
チェックポイントの緊急度, 247
チェックポイント／秒, 247
チェックポイント・フラッシュ／秒, 247
チェックポイント・ログの書き込み／秒, 247
チェックポイント・ログの使用ページ数, 247
チェックポイント・ログのディスクへのコミット／秒, 247
チェックポイント・ログのビットマップ・サイズ, 247
チェックポイント・ログのビットマップへの書き込み／秒, 247
チェックポイント・ログのプレイメージ保存／秒, 247
チェックポイント・ログのページ書き込み／秒, 247

 Copyright © 2007, iAnywhere Solutions, Inc.

チェックポイント・ログのページ再配置／秒, 247
チェックポイント・ログの保存ページ・イメージ／秒, 247
チェックポイント・ログのログ・サイズ, 247
通信, 248
通信の空きバッファ, 248
通信の失敗した送信／秒, 248
通信の受信されるマルチパケット数／秒, 248
通信の受信パケット数／秒, 248
通信の受信要求数, 248
通信の送信されるマルチパケット数／秒, 248
通信の送信バイト数／秒, 248
通信の送信パケット数／秒, 248
通信の総バッファ数, 248
通信の無圧縮受信バイト数／秒, 248
通信の無圧縮受信パケット数／秒, 248
通信の無圧縮送信バイト数／秒, 248
通信の無圧縮送信パケット数／秒, 248
通信のユニークなクライアント・アドレス, 248
通信のリモートプット待ち／秒, 248
ディスク I/O, 249
ディスク書き込み, 250
ディスク書き込みのアクティブ, 250
ディスク書き込みのアクティブの最大値, 250
ディスク書き込みのコミット・ファイル／秒, 250
ディスク書き込みのテンポラリ拡張／秒, 250
ディスク書き込みのデータベース拡張／秒, 250
ディスク書き込みのトランザクション・ログ／秒, 250
ディスク書き込みのページ／秒, 250
ディスクのアクティブ I/O, 249
ディスクのアクティブ I/O の最大値, 249
ディスク読み込み, 250
ディスク読み込みのアクティブ, 250
ディスク読み込みのアクティブの最大値, 250
ディスク読み込みのインデックス内部／秒, 250
ディスク読み込みのインデックス・リーフ／秒, 250
ディスク読み込みの合計ページ数／秒, 250
ディスク読み込みのテーブル／秒, 250
テンポラリ・テーブル数, 254
トランザクションのコミット, 253
トランザクションのロールバック, 253
トランザクション・ログ・グループのコミット, 250
パフォーマンス, 245
パフォーマンス・モニタ, 246
パフォーマンス・モニタからの削除, 244
パフォーマンス・モニタに追加, 244
バージョン・ストア・ページ, 254
表示, 245
開いているカーソル, 253
ヒープのカーバ, 251
ヒープのクエリ処理, 251
ヒープの再配置可能, 251
ヒープの再配置可能ロック, 251
プロシージャ, 569
文, 253
文キャッシュ・ヒット, 253
文キャッシュ・ミス, 253
文の準備, 253
マップ物理メモリ／秒, 251
マルチページ割り当て, 251
メイン・ヒープ・バイト, 254
メモリ・ページ, 252
メモリ・ページのカーバ, 251
メモリ・ページのクエリ処理, 251
メモリ・ページの固定カーソル, 251
メモリ・ページの再配置可能, 252
メモリ・ページの再配置／秒, 252
メモリ・ページのトリガ定義, 252
メモリ・ページのビュー定義, 252
メモリ・ページのプロシージャ定義, 252
メモリ・ページのマップ・ページ, 252
メモリ・ページのメイン・ヒープ, 252
メモリ・ページのロック・テーブル, 252
メモリ・ページのロック・ヒープ, 252
メモリ・ページのロールバック・ログ, 252
モニタリング, 243
有効, 245
要求, 253
要求の GET DATA／秒, 254
要求のアクティブ, 253
要求の交換, 253
要求の未スケジュール, 253
リカバリ I/O 予測, 247
リカバリの緊急度, 247
ロック数, 253
統計が存在し、正確である

説明, 528
統計情報付きのグラフィカルなプラン
説明, 581, 584
統計値
リスト, 246
統計値の削除
Sybase Central パフォーマンス・モニタ, 244
統計値の追加
Sybase Central パフォーマンス・モニタ, 244
統計値の追加と削除
説明, 244
等号演算子
比較演算子, 297
統合データベース
設定, 37
統合データベースの指定
説明, 37
同時性
DDL 文, 182
ISO SQL 標準, 133
一貫性, 133
インデックスの使用による改善, 146
改善, 146
説明, 122
パフォーマンス, 123
プライマリ・キー, 181
矛盾, 133
利点, 123
ロックの仕組み, 166
同時性の問題点の確認
パフォーマンス向上のためのヒント, 258
同時トランザクション
ブロック, 140
動的キャッシュ・サイズ決定
UNIX の説明, 276
Windows, 275
Windows CE, 275
パフォーマンス向上のためのヒント, 275
ドキュメント
挿入, 502
ドキュメントとイメージの挿入
説明, 502
特殊な IDENTITY カラム
説明, 624
特殊なジョイン
説明, 365
特殊な同時性の問題
説明, 181
独立性レベル
ODBC, 137
各レベルの典型的なトランザクション, 145
参照, 139
スナップショット・アイソレーションのレベルの選択, 143
設定, 135
説明, 125
選択, 143
チュートリアル, 147
典型的なトランザクション, 145
典型的な矛盾, 134, 157, 162
トランザクション内の変更, 138
レベル 0 で実装, 174
レベル 1 で実装, 174
レベル 2 で実装, 175
レベル 2 と 3 での同時性の改善, 146
レベル 3 で実装, 175
ロック・タイプの選択のチュートリアル, 154
独立性レベル 0
SELECT 文のロック, 174
例, 147
独立性レベル 1
SELECT 文のロック, 174
例, 151
独立性レベル 2
SELECT 文のロック, 175
例, 157, 162
独立性レベル 2 と 3 での同時性の改善
説明, 146
独立性レベル 3
SELECT 文のロック, 175
逐次テーブル・スキャン, 552
例, 158
独立性レベルと一貫性
説明, 125
独立性レベルの参照
説明, 139
独立性レベルの設定
説明, 135
独立性レベルの選択
説明, 143
独立性レベルの変更
説明, 135
どのようなときにインデックスを作成するか
説明, 595

 Copyright © 2007, iAnywhere Solutions, Inc.

ドメイン
SQL を使用した作成, 111
Sybase Central を使用した作成, 110
大文字と小文字の区別, 622
カラムへの割り当て, 110
検査制約, 107
削除, 112
使用, 110
使用例, 110
[ドメイン作成] ウィザード
使用, 110
ドメインのカラム検査制約
継承, 107
ドメインのカラム検査制約の継承
説明, 107
ドメインの使用
説明, 110
トラブルシューティング
ANY 演算子, 472
GROUP BY 句, 312
結果セットが変わったように見える, 309
サブクエリ, 466
デッドロック, 141
ナチュラル・ジョイン, 374
ニュースグループ, xvii
パフォーマンス, 257
リモート・データ・アクセス, 756
トランザクション
開始, 121
干渉, 140, 155
管理の制限, 750
完了, 121
サブトランザクションとセーブポイント, 123
使用, 120
スナップショット・アイソレーションでの開始, 129
説明, 120
セーブポイント, 123
デッドロック, 141
典型的な独立性レベル, 145
データ修正, 494
データ・リカバリ, 495
同時性, 122
独立性レベルの変更, 138
複数, 122
プロシージャとトリガ, 826
ブロック, 140, 141
ブロックとデッドロック, 140
ブロックの例, 155
リモート・データ・アクセス, 750
トランザクション間の干渉
説明, 140
例, 155
トランザクション管理の制限
説明, 750
トランザクション結果の保存
説明, 121
トランザクション処理
スケジューリング, 144
スケジューリングの影響, 144
直列可能なスケジューリング, 144
データ・リカバリ, 495
パフォーマンス, 123
ブロック, 140
ブロックの例, 155
トランザクション・スケジューリング
影響, 144
トランザクションとデータ修正
説明, 494
トランザクションとデータ・リカバリ
説明, 495
トランザクションと独立性レベル
説明, 119
トランザクションと独立性レベルの使用
説明, 119
トランザクション内の独立性レベルの変更
説明, 138
トランザクションによる変更のグループ分け
説明, 120
トランザクションのインタリーブ
説明, 144
トランザクションの管理
説明, 750
トランザクションの管理とリモート・データ
説明, 750
トランザクションの完了
説明, 121
トランザクションのコミットの統計値
説明, 253
トランザクションの終了
説明, 121
トランザクションの順序付け
説明, 144
トランザクションの使用

説明, 120
トランザクションのスケジューリング
説明, 144
トランザクションのロールバック
説明, 121
トランザクションのロールバックの統計値
説明, 253
トランザクション・ブロック
説明, 140
トランザクション・ログ
dbmlsync, 710
データ・リカバリ, 683
パフォーマンスのヒント, 257
レプリケーション, 710
トランザクション・ログ・グループのコミットの統計値
説明, 250
トランザクション・ロック
期間, 167
トリガ
AFTER トリガ, 790
BEFORE トリガ, 790
EXECUTE パーミッション, 796
INSTEAD OF トリガ, 796
ROLLBACK 文, 632
SQL を使用した変更, 795
Sybase Central を使用した変更, 794
Transact-SQL, 623, 631
エラー処理, 817
概要, 778
カーソル, 813
許可される文, 829
警告, 820
構造, 805
コマンド・デリミタ, 827
再帰, 632
削除, 795
作成, 791
時間, 828
実行, 794
実行順序, 796
使用, 790
説明, 777
セーブポイント, 826
タイプ, 790
[トリガ作成] ウィザード, 791
トリガの実行順序, 796
日付, 828
プロファイリング結果の生成と確認, 204
文レベル, 631
利点, 779
例外ハンドラ, 821
[トリガ作成] ウィザード
使用, 792
トリガの実行
説明, 794
トリガの条件
トリガの実行順序, 796
トリガの変更
説明, 794
取り消し
外部関数, 834
トレーシング
設定, 218
説明, 216
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, 216
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, 226
トレーシング・データベース, 216
トレーシングの設定, 225
トレーシングの設定の確認, 224
トレーシング・セッション
作成, 226
説明, 216
データ, 216
トレーシング・セッションの作成
説明, 226
トレーシング・セッションの実行中のトレーシングの設定の変更
説明, 226
トレーシング・セッションのデータ
説明, 216
トレーシング・タイプ
OPTIMIZATION_LOGGING_WITH_PLANS トレース・タイプ, 221
OPTIMIZATION_LOGGING トレース・タイプ, 221
診断トレーシング, 221
トレーシング・データ
説明, 216
トレーシング・データベース
説明, 216

 Copyright © 2007, iAnywhere Solutions, Inc.

別個のトレーシング・データベースの作成, 234
トレーシングの設定の変更
説明, 225
トレーシング・レベル
使用するトレーシング・レベルの決定, 218
診断トレーシング, 219
設定, 225
トレーシング・レベルの選択
説明, 218

な

内部オペレーション
リモート・データ・アクセス, 752
内部ジョイン
外部ジョインからの変換, 517
ジョインの削除によるリライト最適化, 518
説明, 358
内部ジョインと外部ジョイン
説明, 358
内部ロード
説明, 682
長いプラン
Interactive SQL を使用したアクセス, 588
SQL 関数, 588
説明, 581
ナチュラル・ジョイン
ON 句の使用, 374
エラー, 374
説明, 373
テーブル式, 375
ビューと抽出テーブル, 376

に

二重引用符
文字列, 302
入力
データのインポート, 684
テーブルのインポート, 690
ニュースグループ
テクニカル・サポート, xvii

ね

ネイティブ文
リモート・サーバへの送信, 746
ネイティブ文のリモート・サーバへの送信
説明, 746
ネスト
外部ジョイン, 360
ジョイン, 350
ジョインでの抽出テーブル, 372
ネストされたサブクエリ
説明, 482
ネストされた複合文と例外処理
説明, 823
ネスト・ループ・ジョイン・アルゴリズム
クエリ実行アルゴリズム, 557
ネスト・ループ・セミジョイン
クエリ実行アルゴリズム, 557
ネーム・スペース
インデックス, 623
トリガ, 623

は

排他テーブル・ロック
説明, 171
排他リスト
実行プラン内の項目, 579
排他ロック
説明, 168, 169
バイナリ・ファイル
インポート, 693
バイナリ・ファイルのインポート
説明, 693
バイナリ・ラージ・オブジェクト
挿入, 502
バグ
フィードバックの提供, xvii
バケット
ヒストグラム, 524
パスワード
Lotus Notes, 774
大文字と小文字の区別, 622
パターン一致
概要, 299
ハッシュ DISTINCT
クエリ実行アルゴリズム, 558
ハッシュ GROUP BY
クエリ実行アルゴリズム, 560
ハッシュ GROUP BY SETS
クエリ実行アルゴリズム, 561
ハッシュ NOT EXIST
クエリ実行アルゴリズム, 554
ハッシュ・ジョイン

クエリ実行アルゴリズム, 554, 555
ハッシュ・ジョインの変形
説明, 552
ハッシュ・セミジョイン
クエリ実行アルゴリズム, 555
ハッシュ・テーブル・スキャン
アルゴリズム, 549
ハッシュ非セミジョイン
クエリ実行アルゴリズム, 554
ハッシュ・フィルタ
クエリ実行アルゴリズム, 568
ハッシュ・マップ
クエリ実行アルゴリズム, 568
バッチ
OUTPUT 文, 800
SELECT 文の使用, 829
Transact-SQL の概要, 632
許可される SQL 文, 829
許可される文, 829
作成, 632
制御文, 801
説明, 799
複合文, 799
バッチ処理
Interactive SQL, 720
バッチで SELECT 文を使用する
説明, 829
バッチに使用できる文
説明, 829
バッチ・モード
Interactive SQL, 720
パフォーマンス
all-rows 最適化ゴール, 279
Windows でモニタリング, 245
Windows パフォーマンス モニタの統計, 245
WITH EXPRESS CHECK, 279
アプリケーション・プロファイリング, 202
インデックス, 87, 271
オプティマイザによる推定と実際の統計の比較, 584
オプティマイザの負荷, 258
改善, 88
改善とロック, 146
カスケードされた参照アクションを最小限に抑える, 260
キャッシュ読み込みヒット率, 585
キー, 271
クエリ速度の測定, 260
向上のためのヒントのリスト, 257
実行時間の実際値と推定値, 585
実行プランの解釈, 571
自動チューニング, 526
述部の分析, 529
詳細なアプリケーション・プロファイリング, 216
推定ソース, 585
説明, 257
選択性, 584
データベースの再構築, 263
テーブルとページのサイズ, 591
パフォーマンスのモニタリングと改善のためのツール, 199
バルク・ロード, 682
ファイルの断片化, 264
分散読み込み, 270
ページ・サイズ, 269
ページ・サイズの推奨値, 591
モニタリング, 243, 246
ワーク・テーブル, 279
パフォーマンス向上のためのヒント
圧縮機能の使用, 278
インデックスの断片化削減, 266
インデックスの有効な使用, 271
オプティマイザの優先度の選択, 258
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, 270
オートコミット・モードをオフにする, 268
外部キーを使ったクエリのパフォーマンス改善, 272
キャッシュ, 259
キャッシュに使用されるメモリの制限, 273
キーの有効な使用, 271
クエリ結果の効果的なソート, 267
クエリ処理におけるワーク・テーブルの使用, 279
クエリのパフォーマンスのモニタ, 260
クライアントとサーバとの間の要求数の削減, 267
コストの高いトリガを置き換える, 267
コストの高いユーザ定義関数の削減, 267
異なるファイルの異なるデバイスへの配置, 261
制約の宣言, 259
説明, 257

 Copyright © 2007, iAnywhere Solutions, Inc.

正しいカーソル・タイプの指定, 268
断片化の削減, 264
小さいテーブルに関する統計収集の検討, 259
適切なデータ型の使用, 270
適切なハードウェアの入手, 257
適切なページ・サイズの使用, 269
テーブル検証時の WITH EXPRESS CHECK オプションの使用, 279
テーブル構造の正規化, 261
テーブル内のカラムの順序の確認, 261
テーブルの断片化の削減, 265
テーブル幅の縮小, 267
同時性の問題点の確認, 258
動的キャッシュ・サイズ決定, 275
トランザクション・ログ, 257
パフォーマンス向上へのキャッシュの使用, 273
バルク・オペレーション, 270
プライマリ・キー幅の縮小, 267
プライマリ・キーを使ったクエリのパフォーマンス改善, 272
別個のプライマリ・キー・インデックスと外部キー・インデックス, 273
明示的な選択性推定の提供を控える, 268
パフォーマンス向上へのキャッシュの使用
パフォーマンス向上のためのヒント, 273
パフォーマンス・ツール
グラフィカルなプラン, 240
タイミング・ユーティリティ, 242
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, 237
パフォーマンスの向上
説明, 257
パフォーマンスの自動チューニング
説明, 526
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, 572
実行プランの解釈, 571
パフォーマンスのモニタリング
クエリ測定ツール, 260
パフォーマンスのモニタリングと改善
説明, 199
パフォーマンスの問題のトラブルシューティング
説明, 229
パフォーマンス・モニタ
Sybase Central, 243
Sybase Central で開く, 244
Windows, 245
概要, 243
起動, 245
サポートされている統計値のリスト, 246
パフォーマンス・モニタの統計値, 246
複数コピーの実行, 245
パフォーマンス・モニタの統計値
説明, 246
パラメータ
関数, 311
パラメータを外部関数に渡す
説明, 834
パラメータを関数に渡す
説明, 806
パラメータをプロシージャに渡す
説明, 806
バルク・オペレーション
データのリカバリの問題, 683
パフォーマンス向上のためのヒント, 270
バルク・オペレーションのデータ・リカバリの問題
説明, 683
バルク・オペレーションのパフォーマンスの側面
説明, 682
バルク・ロード
パフォーマンス, 682
範囲クエリ
説明, 298
バージョン・ストア・ページの統計値
説明, 254
ハードウェア・リソースが制限要因であるかどうかの判断
説明, 230
パーミッション
ASE, 616
外部関数を呼び出すプロシージャ, 830
デバッグ, 839
データの修正, 494
トリガ, 796
プロシージャの結果セット, 810
ユーザ定義関数, 789

ひ

比較
NULL 値, 304
概要, 306
後続ブランク, 297

サブクエリの使用, 464
比較演算子
NULL 値, 304
記号, 297
サブクエリ, 485
比較テスト
サブクエリ, 471
非決定性関数
副次的影響, 566
ヒストグラム
更新, 526
説明, 524
非セミジョイン
クエリ実行アルゴリズム, 554
非相関サブクエリ
説明, 484, 521
左外部ジョイン
説明, 358
非ダーティ・リード
チュートリアル, 147
日付
検索, 302
探索条件の概要, 306
入力規則, 302
プロシージャとトリガ, 828
必須
外部キー, 114
必須の関係
説明, 8
ビット
実行プラン内の項目, 578
ビットマップ
スキャン, 591
等しくない
比較演算子, 297
ビュー
DISABLED ステータス, 68
FROM 句, 295
INSTEAD OF トリガを使用した更新, 797
INVALID ステータス, 68
SELECT 文の制限, 62
SQL を使用した変更, 65
Sybase Central を使用した変更, 64
VALID ステータス, 68
依存性, 67
エクスポート, 695
外部ジョイン, 361
共通テーブル式, 392
キー・ジョイン, 385
更新, 62
コピー, 62
削除, 65
作成, 61
使用, 62
処理, 60
ステータス, 68
チェック・オプション, 63
データのブラウズ, 71
ナチュラル・ジョイン, 376
変更時の考慮事項, 63
変更とビューの依存性, 64
無効化, 66
有効化, 66
[ビュー作成] ウィザード
使用, 61
ビュー内のデータのブラウズ
説明, 71
ビューの依存性
カタログ内の情報, 71
スキーマ変更, 70
説明, 67
ビューのステータス, 68
ビューの一致
実体化ビュー (Materialized View) と外部ジョイン, 541
ビューのエクスポート
説明, 704
ビューの削除
説明, 65
ビューの作成
説明, 61
ビューのステータス
DISABLED, 68
INVALID, 68
VALID, 68
説明, 68
特定, 68
ビューの使い方
説明, 62
ビューの変更
説明, 63
ビューの編集
説明, 60
ビューの有効化と無効化

説明, 66
ビュー・マッチング
クエリ実行, 531
実行プランの結果, 576
スナップショット・アイソレーションでの使用, 128
説明, 534
ビュー・マッチング・アルゴリズムの実行プランの結果, 576
ビュー・マッチング・アルゴリズムの説明, 534
ヒューリスティック
クエリの最適化, 525
表記
規則, xii
表示
テーブル・データ, 51
ビュー・データ, 71
プロシージャ・プロファイリングの結果, 207
標準出力
ファイルへのリダイレクト, 700
標準偏差関数
OLAP, 444
開いているカーソルの統計値
説明, 253
ヒント
パフォーマンス向上, 257
頻繁に検索するカラムにインデックスを使う
説明, 88
ヒープのカーバの統計値
説明, 251
ヒープのクエリ処理の統計値
説明, 251
ヒープの再配置可能の統計値
説明, 251
ヒープの再配置可能ロックの統計値
説明, 251

ふ

ファイル
グラフィカルなプラン, 240
断片化, 264
ファイルの断片化
説明, 264
ファイルの断片化の削減
説明, 264
ファイルへのデータベース出力の書き込み
説明, 721
ファンアウト
インデックス, 596
フィルタ
クエリ実行アルゴリズム, 568
フィルタ・アルゴリズム
説明, 568
フィードバック
提供, xvii
マニュアル, xvii
深さ
実行プラン内の項目, 578
複合インデックス
ORDER BY 句, 598
カラムの順序の効果, 597
説明, 597
復号化
実体化ビュー (Materialized View), 79
複合文
アトミック, 803
使用, 803
宣言, 803
複合文での宣言
説明, 803
複合文の使用
説明, 803
複雑な外部ジョイン
説明, 360
複雑な外部ジョインについて
説明, 360
複数のカラムを使用した GROUP BY
説明, 325
複数のカラムを使用したグループ化
説明, 325
複数のデータベース
ジョイン, 745
複数のトランザクション
同時性, 122
複数のローカル・データベースのテーブルのジョイン
説明, 745
複数ローのサブクエリ
説明, 464
不正な XML 名のエンコーディング
説明, 654
物理インデックス
共有される物理インデックスの特定, 594

説明, 593
物理インデックスを共有する論理インデックスの特定
説明, 594
物理データの編成とアクセス
説明, 590
物理データ・モデル
生成, 19
不定の値
説明, 303
不等号
不等のテスト, 306
部分インデックス・スキャン
説明, 599
不要な DISTINCT の削除
説明, 513
不要な内部ジョインと外部ジョインの削除
説明, 518
プライマリ・キー
GLOBAL AUTOINCREMENT, 103
NEWID を使用して UUID を作成, 104
SQL を使用した作成, 52
SQL を使用した変更, 52
Sybase Central で作成, 52
Sybase Central で変更, 52
エンティティ整合性, 113
オートインクリメント, 102
管理, 51
整合性, 608
生成, 181
生成されたインデックス, 595
同時性, 181
パフォーマンス, 271
プライマリ・キーによるエンティティ整合性の確保
説明, 113
プライマリ・キーの管理
説明, 51
プライマリ・キー幅の縮小
パフォーマンス向上のためのヒント, 267
プライマリ・キーを使ったクエリのパフォーマンス改善
パフォーマンス向上のためのヒント, 272
ブラウズ
テーブル・データ, 51
ビュー, 71
プラス演算子
NULL 値, 305
プラン
Interactive SQL を使用したアクセス, 588
SQL 関数, 588
印刷, 582
解釈, 571
キャッシュ, 544
グラフィカルなプラン, 581
グラフィカルなプランのカスタマイズ, 582
コンテキスト別のヘルプ, 583
使用される省略形, 572
長いテキスト・プラン, 581
短いテキスト・プラン, 580
プラン構築フェーズ
クエリ処理, 511
プラン内の選択性
説明, 587
プランに使用される一般的な推定
説明, 575
プランに使用される一般的な統計
説明, 574
プランのキャッシュ
説明, 544
古いデータ
実体化ビュー (Materialized View) でのリフレッシュ, 78
ブルーム・フィルタ (参照 ハッシュ・フィルタ)
クエリ実行アルゴリズム, 568
ブレーク条件
設定, 845
ブレークポイント
カウント, 846
個々の接続, 846
個々のユーザ, 846
条件, 846
ステータス, 846
設定, 845
説明, 845
無効化, 846
有効化, 846
ブレークポイント条件の編集
説明, 846
ブレークポイントの活用
説明, 845
ブレークポイントの設定
デバッガ, 845
ブレークポイントの無効化

 Copyright © 2007, iAnywhere Solutions, Inc.

説明, 846
有効化, 846
ブレークポイントの有効化
説明, 846
プロキシ・テーブル
Sybase Central からの作成, 740
作成, 727, 741, 742
説明, 739
ディレクトリ・アクセス・サーバからの削除, 736
データのインポート, 689
ロケーション, 739
[プロキシ・テーブル作成] ウィザード
使用, 740
プロキシ・テーブルの使用
説明, 739
プロキシ・テーブルのロケーションの指定
説明, 739
プロキシ・テーブルを使用したデータのインポート
説明, 689
プログラム変数
共通テーブル式, 395
プロシージャ
EXECUTE IMMEDIATE 文, 825
FROM 句内での使用, 295
ProcCall アルゴリズム, 569
Sybase Central を使用した変更, 781
Transact-SQL, 634
Transact-SQL の概要, 631
エラー処理, 637, 638, 817
外部関数, 830
概要, 778
カーソル, 813
カーソルの使用, 813
許可される文, 829
警告, 820
結果セット, 784, 810
結果セットに対するパーミッション, 810
結果を返す, 783, 808
構造, 805
コピー, 782
コマンド・デリミタ, 827
削除, 783
作成, 780
作成のヒント, 827
時間, 828
使用, 780
セキュリティ, 779
説明, 777
セーブポイント, 826
デフォルトのエラー処理, 817
テーブル名, 827
統計, 525
入力検証, 828
パラメータ, 805, 806
日付, 828
複数の結果セット, 811
プロファイリング結果の生成と確認, 204
変換, 634
変数結果セット, 812
戻り値, 637
呼び出し, 782
利点, 779
リモート・プロシージャの削除, 748
リモート・プロシージャの追加, 747
例外ハンドラ, 821
プロシージャから返される結果
説明, 808
プロシージャから結果セットを返す
説明, 810
プロシージャからの外部ライブラリの呼び出し
説明, 830
プロシージャから複数の結果セットを返す
説明, 811
プロシージャから変数結果セットを返す
説明, 812
プロシージャ言語
概要, 631
[プロシージャ作成] ウィザード
使用, 780
説明, 780
プロシージャでの EXECUTE IMMEDIATE 文の使用
説明, 825
プロシージャでの日付と時刻の指定
プロシージャを作成するときのヒント, 828
プロシージャとトリガでのエラーと警告
説明, 817
プロシージャとトリガでのカーソルの使用
説明, 813
プロシージャとトリガでのデフォルトのエラー処理
説明, 817

プロシージャとトリガでのデフォルトの警告処理
説明, 820
プロシージャとトリガでのトランザクションとセーブポイント
説明, 826
プロシージャとトリガでの例外ハンドラの使用
説明, 821
プロシージャとトリガの構造
説明, 805
プロシージャとトリガの利点
説明, 779
プロシージャ、トリガ、バッチの使用
説明, 777
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
プロシージャを作成するときのヒント, 827
プロシージャの SELECT 文でのカーソルの使用
説明, 813
プロシージャの結果を結果セットとして返す
説明, 784
プロシージャの結果をパラメータとして返す
説明, 783
プロシージャの中での RAISERROR 文の使用
説明, 638
プロシージャの中で文を区切る
プロシージャを作成するときのヒント, 827
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
プロシージャを作成するときのヒント, 828
プロシージャの変更
説明, 781
プロシージャの呼び出し
説明, 782
プロシージャ・パラメータの宣言
説明, 805
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, 238
sa_server_option を使用したプロファイリングのリセット, 238
sa_server_option を使用した無効化, 239
Sybase Central, 204
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, 239
システム・プロシージャを使用して実行, 237
プロファイリング結果の解釈, 207
プロファイル情報を収集できるオブジェクト, 207
ベースライン, 208
無効化, 206
有効化, 204
リセット, 205
プロシージャ・プロファイリングの結果の分析
説明, 207
プロシージャ・プロファイリングの無効化
説明, 206
プロシージャ・プロファイリングの有効化
説明, 204
プロシージャ・プロファイリングのリセット
説明, 205
プロシージャを作成するときのヒント
コマンド・デリミタを変更する必要があるかどうかをチェックする, 827
説明, 827
プロシージャでの日付と時刻の指定, 828
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, 827
プロシージャの中で文を区切る, 827
ブロック
説明, 140
デッドロック, 141
トラブルシューティング, 141
トランザクション, 140
例, 155
ブロック・オプション
使用, 140
プロトタイプ
外部関数, 832
プロパティ
すべてのデータベース・オブジェクトのプロパティを設定する, 36
プロパティ・シート
説明, 36
プロファイリング・ログ・ファイルのベースラインとしての使用
説明, 208
文
CREATE DATABASE, 613
CREATE DEFAULT, 614
CREATE DOMAIN, 614
CREATE RULE, 614
DISK, 613
DROP DATABASE, 613

 Copyright © 2007, iAnywhere Solutions, Inc.

DUMP DATABASE, 613
DUMP TRANSACTION, 613
GRANT, 617
LOAD DATABASE, 613
LOAD TRANSACTION, 613
OUTPUT, 700
RAISERROR, 638
REVOKE, 617
ROLLBACK, 632
SELECT, 627, 628
SIGNAL, 638
最適化, 523
速度が遅い文の検出, 229
複合, 803
ロギング, 38
文キャッシュ・ヒットの統計値
説明, 253
文キャッシュ・ミスの統計値
説明, 253
分散読み込み
パフォーマンス, 270
分析
プロシージャ・プロファイリングの結果, 207
文の完全なパススルー
リモート・データ・アクセス, 753
文の準備の統計値
説明, 253
文の統計値
説明, 253
文の部分的なパススルー
リモート・データ・アクセス, 753
文レベルのトリガ
説明, 631

へ

平方偏差関数
OLAP, 444
並列インデックス・スキャン
アルゴリズム, 551
テーブルのスキャン, 551
並列処理
説明, 547
並列処理が使用される状況, 548
並列処理が使用される状況
並列処理, 548
並列テーブル・スキャン
アルゴリズム, 551
べき等関数
定義, 566
ベクトル集約値
説明, 321
別個のトレーシング・データベースの作成
説明, 234
別個のプライマリ・キー・インデックスと外部キー・インデックス
パフォーマンス向上のためのヒント, 273
ヘルプ
テクニカル・サポート, xvii
ヘルプへのアクセス
テクニカル・サポート, xvii
変形
リライト最適化, 512
変更
SQL を使用したカラムの変更, 48
SQL を使用したテーブルの変更, 48
SQL を使用したトリガの変更, 795
SQL を使用したビューの変更, 65
Sybase Central を使用したカラムの変更, 47
Sybase Central を使用したテーブルの変更, 47
Sybase Central を使用したトリガの変更, 794
Sybase Central を使用したビューの変更, 64
Sybase Central を使用したプロシージャの変更, 781
依存性があるビュー, 63
計算カラム式, 57
検査制約, 108
ビューの依存性があるテーブル, 46
変更の確定
説明, 494
変更のキャンセル
説明, 495
編集
データベース・オブジェクトのプロパティ, 36
テーブル・データ, 51
変数
SELECT 文, 628
SET 文, 628
Transact-SQL, 636
ローカル, 628
割り当て, 628
変数の検査
デバッガ, 847
編成
物理データ, 590

ページ
実行プラン内の項目, 578
挿入されたローに対するディスク割り付け, 590
ページ・サイズ
Windows CE の考慮事項, 592
インデックス, 592
説明, 591
挿入されたローに対するディスク割り付け, 590
パフォーマンス, 269
パフォーマンスの考慮事項, 591
ページ・マップ
実行プラン内の項目, 577
スキャン, 591
ベース・テーブル
作成, 45
ベースライン
プロシージャ・プロファイリングの使用, 208

ほ

方向
実行プラン内の項目, 578
保護されたテーブル
外部ジョイン, 358
保存
結果セット, 721
トランザクション結果, 121

ま

マップ物理メモリ／秒の統計値
説明, 251
マニュアル
SQL Anywhere, x
幻ロック
説明, 162, 172
幻ロー
チュートリアル, 157
データの矛盾, 133
独立性レベル, 134, 162
独立性レベル 2 で防止, 175
マルチページ割り当ての統計値
説明, 251
マージ・ジョイン
クエリ実行アルゴリズム, 556

み

右外部ジョイン
説明, 358
短いプラン
Interactive SQL を使用したアクセス, 588
SQL 関数, 588
説明, 580
未使用インデックス
インデックス・コンサルタントの推奨内容, 214
未スケジュールの要求の統計値
説明, 253

む

無効化
Sybase Central でのプロシージャ・プロファイリング, 206
実体化ビュー (Materialized View), 80
ビュー, 66
無効なデータ
説明, 96
矛盾
ISO SQL 標準, 133
繰り返し不可能読み出し, 133
繰り返し不可能読み出しの例, 153
実際のロックの意味, 162
ダーティ・リード, 133
ダーティ・リードとロック, 174
ダーティ・リードのチュートリアル, 147
直列不可能なスケジュールの影響, 144
幻ロー, 133
幻ローとロック, 175
幻ローのチュートリアル, 157
ロックを使用して防止, 166

め

明示的ジョイン条件
説明, 347
明示的ジョイン条件の種類
説明, 354
明示的なジョイン条件 (ON 句)
説明, 352
明示的な選択性推定の提供を控える
パフォーマンス向上のためのヒント, 268
命令グラフ
説明, 406

 Copyright © 2007, iAnywhere Solutions, Inc.

メイン・ヒープ・バイトの統計値
説明, 254
メタデータ・サポート
jConnect に対するインストール, 40
メタプロパティ名
id, 648
localname, 648
メモリ・ページのカーバの統計値
説明, 251
メモリ・ページのクエリ処理の統計値
説明, 251
メモリ・ページの固定カーソルの統計値
説明, 251
メモリ・ページの再配置可能の統計値
説明, 252
メモリ・ページの再配置／秒の統計値
説明, 252
メモリ・ページの統計値
リスト, 252
メモリ・ページのトリガ定義の統計値
説明, 252
メモリ・ページのビュー定義の統計値
説明, 252
メモリ・ページのプロシージャ定義の統計値
説明, 252
メモリ・ページのマップ・ページの統計値
説明, 252
メモリ・ページのメイン・ヒープの統計値
説明, 252
メモリ・ページのロック・テーブルの統計値
説明, 252
メモリ・ページのロック・ヒープの統計値
説明, 252
メモリ・ページのロールバック・ログの統計値
説明, 252
メンテナンス
パフォーマンス, 257

も

文字データ
検索, 302
文字列
select リストの使用, 291
引用符, 302
使用, 302
マッチング, 299
文字列と引用符
説明, 302
文字列と数値のデフォルト
説明, 105
文字列の連結
NULL, 305
戻り値
プロシージャ, 637
モニタ
Sybase Central パフォーマンス・モニタを開く, 244
パフォーマンス・モニタの概要, 243
役割
ASE, 615
定義, 7
役割名
説明, 378

ゆ

有効化
Sybase Central でのプロシージャ・プロファイリング, 204
実体化ビュー (Materialized View), 80
ビュー, 66
ユニーク・キー
生成と同時性, 181
ユニークな結果
制限, 294
ユニークな識別子
テーブル, 51
ユーザ ID
ASE, 616
大文字と小文字の区別, 622
デフォルト, 102
ユーザ定義関数
EXECUTE パーミッション, 789
外部関数, 830
キャッシュ, 566
削除, 789
作成, 787
説明, 787
パラメータ, 806
プロファイリング結果の生成と確認, 204
呼び出し, 787
ユーザ定義関数の呼び出し
説明, 787
ユーザ定義データ型
SQL を使用した作成, 111

Copyright © 2007, iAnywhere Solutions, Inc.

検査制約, 107
削除, 112
作成, 110
説明, 110

よ

要求
数の削減, 267
[要求] タブ
インデックス・コンサルタントの推奨内容, 214
要求トレース分析
実行, 232
説明, 232
要求トレース分析の実行
説明, 232
要求の GET DATA／秒の統計値
説明, 254
要求のアクティブの統計値
説明, 253
要求のキャンセル
リモート・データ・アクセス, 758
要求の交換の統計値
説明, 253
要求の統計値
説明, 253
リスト, 253
要求レベル・ログ (参照 要求ログ)
要求ロギング
説明, 235
要求ログ
セキュリティ, 235
説明, 235
要素
クエリ結果を XML として取得, 653
データベースへの XML の格納, 643
リレーショナル・データからの XML の生成, 644
容量の計画
説明, 230
予測行サイズ
実行プラン内の項目, 577
予測行数
実行プラン内の項目, 577
予測ページ数
実行プラン内の項目, 577
予測リーフ・ページ数
実行プラン内の項目, 578
呼び出しスタック
デバッガ, 847
読み込みロック
説明, 169
長期間, 169
予約語
リモート・サーバ, 756
より大きい
範囲指定, 298
比較演算子, 297
より大きくない
比較演算子, 297
より小さい
範囲指定, 298
比較演算子, 297
より小さくない
比較演算子, 297

ら

ライブラリ
プロシージャまたはユーザ定義関数からの外部ライブラリの呼び出し, 830
ランキング
集合で使用, 453
ランキング関数
最上位と最下位の百分位数の検索, 457
例, 450

り

利益
インデックス・コンサルタントの推奨内容, 214
リカバリ
インポート／エクスポート, 683
リカバリ I/O 予測の統計値
説明, 247
リカバリの緊急度の統計値
説明, 247
リカバリの統計値
リスト, 247
リセット
Sybase Central でのプロシージャ・プロファイリング, 205
リダイレクト
ファイルへの出力, 700
リターン・タイプ

外部関数, 835
リフレッシュ
実体化ビュー (Materialized View), 78
リモート・サーバ
Sybase Central からの作成, 730
外部ログイン, 737
クラス, 759, 760
削除, 731
作成, 729
説明, 729
トランザクション管理, 750
ネイティブ文の送信, 746
プロパティのリスト, 733
変更, 732
[リモート・サーバ作成] ウィザード
使用, 730
リモート・サーバの機能のリスト
説明, 733
リモート・サーバの削除
説明, 731
リモート・サーバの作成
説明, 729
リモート・サーバの使用
説明, 729
リモート・サーバの変更
説明, 732
リモート・データ
ロケーション, 739
リモート・データ・アクセス
Lotus Notes SQL 2.0, 774
Microsoft Access, 773
Microsoft Excel, 772
Microsoft FoxPro, 773
PowerBuilder DataWindows, 728
大文字と小文字の区別, 756
概要, 726
クエリ上でブロックされるクエリ, 757
クエリに関する一般的な問題, 757
クエリの解析, 752
クエリの正規化, 752
クエリの前処理, 752
サポートされていない SQL Remote, 756
サポートされていない機能, 756
サーバの機能, 752
接続の問題点, 756
接続名, 758
説明, 725
トラブルシューティング, 756
内部オペレーション, 752
パススルー・モード, 746
パフォーマンスの制限, 726
文の完全なパススルー, 753
文の部分的なパススルー, 753
リモート・サーバ, 729
リモート・データ・アクセスの基本概念
概要, 727
リモート・データ・アクセスのサーバ・クラス
説明, 759
リモート・データ・アクセスの接続の管理
説明, 758
リモート・データにサポートされていない機能
説明, 756
リモート・データへのアクセス
PowerBuilder DataWindows, 728
基本概念, 727
説明, 725
リモート・テーブル
アクセス, 725
カラムのリスト, 742
説明, 727
リスト, 732
リモート・テーブルのカラムのリスト
説明, 742
リモート・テーブルのジョイン
説明, 743
リモート・テーブル・マッピング
説明, 727
リモート・トランザクション管理
概要, 750
リモート プロシージャ
コール, 747
削除, 748
作成, 780
追加, 747
データ型, 748
リモート・プロシージャ・コール
説明, 747
リモート・プロシージャ・コールの使用
説明, 747
[リモート・プロシージャ作成] ウィザード
使用, 747
リモート・プロシージャの削除
説明, 748
リモート・プロシージャの作成

Copyright © 2007, iAnywhere Solutions, Inc.

説明, 747
リライト最適化
説明, 512
リレーショナル・データ
XML としてエクスポート, 644
リレーショナル・データベースにおける XML 文書の格納
説明, 643
リレーショナル・データを XML としてエクスポートする
説明, 644
リーフ・ページ
説明, 596

る

ループバック接続
説明, 745
ルール
Transact-SQL, 614

れ

例
繰り返し不可能読み出し, 151
ダーティ・リード, 147
幻ロック, 162
幻ロー, 157
ロックの意味, 162
例外
宣言, 818
例外ハンドラ
ネストされた複合文, 823
プロシージャとトリガ, 821
列挙フェーズ
クエリ処理, 510
レプリケーション
データベースの再構築, 710
同期に関連するデータベースの再構築, 710
同時性の問題点, 181
レプリケーションに関連しないデータベースの再構築
説明, 709
連邦情報処理標準刊行物への準拠, ix
(参照 SQL 規格)

ろ

ログ
ロールバック・ログ, 123
ロック, ix
(参照 ロックする)
sa_locks システム・プロシージャを使用したロックの表示, 167
Sybase Central での表示, 167
位置テーブル, 171
位置テーブル・ロック, 171
意図的, 170
意図的ロック, 170
インデックスによって影響を削減, 146
インデックスによる減少, 599
オーファンと参照整合性, 177
書き込み, 169
書き込みを意図したテーブル, 171
書き込みを意図したテーブル・ロック, 171
期間, 167
競合, 173
競合処理, 140, 155
競合のタイプ, 170
共有スキーマ, 168
共有テーブル, 171
共有テーブル・ロック, 171
クエリ時, 174
更新時, 178
更新手順, 178
削除時, 179
削除手順, 179
情報の表示, 167
スキーマ, 168
説明, 166
早期解放, 145
挿入, 173
挿入時, 176
挿入手順, 176
挿入ロック, 173
デッドロック, 141
典型的なトランザクションと独立性レベル, 145
テーブル, 170
独立性レベル, 125
独立性レベル 0 で実装, 174, 175
独立性レベルの選択のチュートリアル, 154
トランザクションのブロックとデッドロック, 140
排他, 169
排他スキーマ, 168
排他テーブル・ロック, 171

Copyright © 2007, iAnywhere Solutions, Inc.

排他ロック, 171
ブロック, 140
ブロックの例, 155
幻, 172
幻ロック, 172
幻ローと独立性レベル, 157, 162
矛盾と典型的な独立性レベル, 134
読み込み, 169
読み込みロックの早期解放, 180
ロックできるオブジェクト, 166
ロー, 168
ロック可能なオブジェクト
説明, 166
ロックされたローへのアクセス待機
デッドロック, 140
ロック情報の取得
説明, 167
ロック数の統計値
説明, 253
ロックする, ix
(参照 ロック)
ロックできるオブジェクト
説明, 166
ロックの影響の削減
説明, 146
ロックの早期解放
トランザクション, 145
例外, 180
論理インデックス
共有される物理インデックスの特定, 594
説明, 593
論理インデックスを使用したインデックスの共有
説明, 593
論理演算子
HAVING 句, 328
条件の接続, 305
論理演算子による複数の条件の接続
説明, 305
ロー
INSERT を使用したコピー, 502
意図的ロック, 170
削除, 507
削除の影響, 591
選択, 296
ロック, 168
ローカル・テンポラリ・テーブル
説明, 94
ローカル変数
デバッガ, 847
ロー・コンストラクタ・アルゴリズム
説明, 570
ロー制限数
実行プラン内の項目, 579
ロード
Interactive SQL 内のコマンド, 721
ローの制限
FIRST 句, 330
TOP 句, 330
クエリ実行アルゴリズム, 570
ローの全カラムへの値の挿入
説明, 499
ローの連続記憶領域
説明, 590
ロー・バージョン
説明, 128
ロールバック
トランザクション, 121
ロールバック・ログ
セーブポイント, 123
データ・リカバリ, 683
ロー・ロック
意図的, 170
書き込み, 169
説明, 166, 168
排他, 169
読み込み, 169

わ

ワイルドカード
LIKE 探索条件, 301
パターン一致, 299
文字列比較, 300
割り当て
カラムのデータ型, 110
カラムのドメイン, 110
ワーク・テーブル
クエリ処理, 279
説明, 279
パフォーマンスのヒント, 261

Copyright © 2007, iAnywhere Solutions, Inc.