SYBASE iAnywhere

SQL Anywhere® サーバ
SQL の使用法

2009 年 2 月

バージョン 11.0.1

版権と商標

目次

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 11 - 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 のマニュアルについて

SQL Anywhere の完全なマニュアルは 4 つの形式で提供されており、いずれも同じ情報が含まれています。

- **HTML ヘルプ**　オンライン・ヘルプには、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。

  Microsoft Windows オペレーティング・システムを使用している場合は、オンライン・ヘルプは HTML ヘルプ (CHM) 形式で提供されます。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 11] - [マニュアル] - [オンライン・マニュアル]** を選択します。

  管理ツールのヘルプ機能でも、同じオンライン・マニュアルが使用されます。

- **Eclipse**　UNIX プラットフォームでは、完全なオンライン・ヘルプは Eclipse 形式で提供されます。マニュアルにアクセスするには、SQL Anywhere 11 インストール環境の *bin32* または *bin64* ディレクトリから *sadoc* を実行します。

- **DocCommentXchange**　DocCommentXchange は、SQL Anywhere マニュアルにアクセスし、マニュアルについて議論するためのコミュニティです。

  DocCommentXchange は次の目的に使用できます (現在のところ、日本語はサポートされておりません)。

  ○ マニュアルを表示する

  ○ マニュアルの項目について明確化するために、ユーザによって追加された内容を確認する

  ○ すべてのユーザのために、今後のリリースでマニュアルを改善するための提案や修正を行う

  http://dcx.sybase.com を参照してください。

- **PDF**　SQL Anywhere の完全なマニュアル・セットは、Portable Document Format (PDF) 形式のファイルとして提供されます。内容を表示するには、PDF リーダが必要です。Adobe Reader をダウンロードするには、http://get.adobe.com/reader/ にアクセスしてください。

  Microsoft Windows オペレーティング・システムで PDF 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 11] - [マニュアル] - [オンライン・マニュアル - PDF]** を選択します。

  UNIX オペレーティング・システムで PDF マニュアルにアクセスするには、Web ブラウザを使用して *install-dir/documentation/ja/pdf/index.html* を開きます。

マニュアル・セットに含まれる各マニュアルについて

SQL Anywhere のマニュアルは次の構成になっています。

- **『SQL Anywhere 11 - 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 11 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 11 - 変更点とアップグレード』** このマニュアルでは、SQL Anywhere 11 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 データベースを実行、管理、構成する方法について説明します。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、管理ユーティリティとオプションについて説明します。
- **『SQL Anywhere サーバ - プログラミング』** このマニュアルでは、C、C++、Java、PHP、Perl、Python、および Visual Basic や Visual C# などの .NET プログラミング言語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。ADO.NET や ODBC などのさまざまなプログラミング・インタフェースについても説明します。
- **『SQL Anywhere サーバ - SQL リファレンス』** このマニュアルでは、システム・プロシージャとカタログ (システム・テーブルとビュー) に関する情報について説明します。また、SQL Anywhere での SQL 言語の実装 (探索条件、構文、データ型、関数) についても説明します。
- **『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。
- **『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 - 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。また、dbmlsync API についても説明します。dbmlsync API を使用すると、同期を C++ または .NET のクライアント・アプリケーションにシームレスに統合できます。
- **『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。
- **『Mobile Link - サーバ起動同期』** このマニュアルでは、Mobile Link サーバ起動同期について説明します。この機能により、Mobile Link サーバは同期を開始したり、リモート・デバイス上でアクションを実行することができます。
- **『QAnywhere』** このマニュアルでは、モバイル・クライアント、ワイヤレス・クライアント、デスクトップ・クライアント、およびラップトップ・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、QAnywhere について説明します。
- **『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。

- **『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 - 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・デバイス、モバイル・デバイス、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 Mobile、または Windows を搭載しているハンドヘルド・デバイス、モバイル・デバイス、または埋め込みデバイスの Web ベースのデータベース・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 - .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・デバイス、モバイル・デバイス、または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 J』** このマニュアルでは、Ultra Light J について説明します。Ultra Light J を使用すると、Java をサポートしている環境用のデータベース・アプリケーションを開発し、配備することができます。Ultra Light J は、BlackBerry スマートフォンと Java SE 環境をサポートしており、iAnywhere Ultra Light データベース製品がベースになっています。
- **『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示し、その診断情報を説明します。

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表記規則について説明します。

オペレーティング・システム

SQL Anywhere はさまざまなプラットフォームで稼働します。ほとんどの場合、すべてのプラットフォームで同じように動作しますが、いくつかの相違点や制限事項があります。このような相違点や制限事項は、一般に、基盤となっているオペレーティング・システム (Windows、UNIX など) に由来しており、使用しているプラットフォームの種類 (AIX、Windows Mobile など) またはバージョンに依存していることはほとんどありません。

オペレーティング・システムへの言及を簡素化するために、このマニュアルではサポートされているオペレーティング・システムを次のようにグループ分けして表記します。

- **Windows** Microsoft Windows ファミリを指しています。これには、主にサーバ、デスクトップ・コンピュータ、ラップトップ・コンピュータで使用される Windows Vista や Windows XP、およびモバイル・デバイスで使用される Windows Mobile が含まれます。

  特に記述がないかぎり、マニュアル中に Windows という記述がある場合は、Windows Mobile を含むすべての Windows ベース・プラットフォームを指しています。

- **UNIX**　特に記述がないかぎり、マニュアル中に UNIX という記述がある場合は、Linux および Mac OS X を含むすべての UNIX ベース・プラットフォームを指しています。

ディレクトリとファイル名

ほとんどの場合、ディレクトリ名およびファイル名の参照形式はサポートされているすべてのプラットフォームで似通っており、それぞれの違いはごくわずかです。このような場合は、Windows の表記規則が使用されています。詳細がより複雑な場合は、マニュアルにすべての関連形式が記載されています。

ディレクトリ名とファイル名の表記を簡素化するために使用されている表記規則は次のとおりです。

- **大文字と小文字のディレクトリ名**　Windows と UNIX では、ディレクトリ名およびファイル名には大文字と小文字が含まれている場合があります。ディレクトリやファイルが作成されると、ファイル・システムでは大文字と小文字の区別が維持されます。

  Windows では、ディレクトリおよびファイルを参照するとき、大文字と小文字は**区別されません**。大文字と小文字を混ぜたディレクトリ名およびファイル名は一般的に使用されますが、参照するときはすべて小文字を使用するのが通常です。SQL Anywhere では、*Bin32* や *Documentation* などのディレクトリがインストールされます。

  UNIX では、ディレクトリおよびファイルを参照するとき、大文字と小文字は**区別されます**。大文字と小文字を混ぜたディレクトリ名およびファイル名は一般的に使用されません。ほとんどの場合は、すべて小文字の名前が使用されます。SQL Anywhere では、*bin32* や *documentation* などのディレクトリがインストールされます。

  このマニュアルでは、ディレクトリ名に Windows の形式を使用しています。ほとんどの場合、大文字と小文字が混ざったディレクトリ名をすべて小文字に変換すると、対応する UNIX 用のディレクトリ名になります。

- **各ディレクトリおよびファイル名を区切るスラッシュ**　マニュアルでは、ディレクトリの区切り文字に円記号を使用しています。たとえば、PDF 形式のマニュアルは *install-dir¥Documentation¥ja¥pdf* にあります。これは Windows の形式です。

  UNIX では、円記号をスラッシュに置き換えます。PDF マニュアルは *install-dir/documentation/ja/pdf* にあります。

- **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、*.exe* や *.bat* などの拡張子が付きます。UNIX では、実行ファイルの名前に拡張子は付きません。

  たとえば、Windows でのネットワーク・データベース・サーバは *dbsrv11.exe* です。UNIX では *dbsrv11* です。

- ***install-dir***　インストール・プロセス中に、SQL Anywhere をインストールするロケーションを選択します。このロケーションを参照する環境変数 SQLANY11 が作成されます。このマニュアルでは、そのロケーションを *install-dir* と表します。

  たとえば、マニュアルではファイルを install-dir¥readme.txt のように参照します。これは、Windows では、%SQLANY11%¥readme.txt に対応します。UNIX では、$SQLANY11/readme.txt または ${SQLANY11}/readme.txt に対応します。

install-dir のデフォルト・ロケーションの詳細については、「SQLANY11 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・プロセス中に、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します。このロケーションを参照する環境変数 SQLANYSAMP11 が作成されます。このマニュアルではそのロケーションを *samples-dir* と表します。

  Windows エクスプローラ・ウィンドウで *samples-dir* を開くには、**[スタート] - [プログラム] - [SQL Anywhere 11] - [サンプル・アプリケーションとプロジェクト]** を選択します。

  samples-dir のデフォルト・ロケーションの詳細については、「SQLANYSAMP11 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。

コマンド・プロンプトとコマンド・シェル構文

ほとんどのオペレーティング・システムには、コマンド・シェルまたはコマンド・プロンプトを使用してコマンドおよびパラメータを入力する方法が、1 つ以上あります。Windows のコマンド・プロンプトには、コマンド・プロンプト (DOS プロンプト) および 4NT があります。UNIX のコマンド・シェルには、Korn シェルおよび bash があります。各シェルには、単純コマンドからの拡張機能が含まれています。拡張機能は、特殊文字を指定することで起動されます。特殊文字および機能は、シェルによって異なります。これらの特殊文字を誤って使用すると、多くの場合、構文エラーや予期しない動作が発生します。

このマニュアルでは、一般的な形式のコマンド・ラインの例を示します。これらの例に、シェルにとって特別な意味を持つ文字が含まれている場合、その特定のシェル用にコマンドを変更することが必要な場合があります。このマニュアルではコマンドの変更について説明しませんが、通常、その文字を含むパラメータを引用符で囲むか、特殊文字の前にエスケープ文字を記述します。

次に、プラットフォームによって異なるコマンド・ライン構文の例を示します。

- **カッコと中カッコ** 一部のコマンド・ライン・オプションは、詳細な値を含むリストを指定できるパラメータを要求します。リストは通常、カッコまたは中カッコで囲まれています。このマニュアルでは、カッコを使用します。次に例を示します。

  ```
  -x tcpip(host=127.0.0.1)
  ```

  カッコによって構文エラーになる場合は、代わりに中カッコを使用します。

  ```
  -x tcpip{host=127.0.0.1}
  ```

  どちらの形式でも構文エラーになる場合は、シェルの要求に従ってパラメータ全体を引用符で囲む必要があります。

  ```
  -x "tcpip(host=127.0.0.1)"
  ```

- **引用符** パラメータの値として引用符を指定する必要がある場合、その引用符はパラメータを囲むために使用される通常の引用符と競合する可能性があります。たとえば、値に二重引用符を含む暗号化キーを指定するには、キーを引用符で囲み、パラメータ内の引用符をエスケープします。

  ```
  -ek "my ¥"secret¥" key"
  ```


多くのシェルでは、キーの値は my "secret" key のようになります。

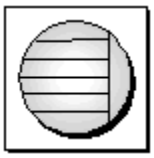
- **環境変数**　マニュアルでは、環境変数設定が引用されます。Windows のシェルでは、環境変数は構文 *%ENVVAR%* を使用して指定されます。UNIX のシェルでは、環境変数は構文 *$ENVVAR* または *${ENVVAR}* を使用して指定されます。

グラフィック・アイコン

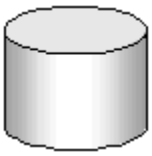
このマニュアルでは、次のアイコンを使用します。

- 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。



- データベース。ハイレベルの図では、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。

- プログラミング・インタフェース。

ドキュメンテーション・チームへのお問い合わせ

このヘルプに関するご意見、ご提案、フィードバックをお寄せください。

SQL Anywhere ドキュメンテーション・チームへのご意見やご提案は、弊社までご連絡ください。頂戴したご意見はマニュアルの向上に役立たせていただきます。ぜひとも、ご意見をお寄せください。

DocCommentXchange

DocCommentXchange を使用して、ヘルプ・トピックに関するご意見を直接お寄せいただくこともできます。DocCommentXchange (DCX) は、SQL Anywhere マニュアルにアクセスしたり、マニュアルについて議論するためのコミュニティです。DocCommentXchange は次の目的に使用できます (現在のところ、日本語はサポートされておりません)。

- マニュアルを表示する
- マニュアルの項目について明確化するために、ユーザによって追加された内容を確認する
- すべてのユーザのために、今後のリリースでマニュアルを改善するための提案や修正を行う

http://dcx.sybase.com を参照してください。

詳細情報の検索／テクニカル・サポートの依頼

詳しい情報やリソースについては、iAnywhere デベロッパー・コミュニティ (http://www.iAnywhere.jp/developers/index.html) 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。バージョンおよびビルド番号を調べるには、コマンド **dbeng11 -v** を実行します。

ニュースグループは、ニュース・サーバ *forums.sybase.com* にあります。

以下のニュースグループがあります。

- ianywhere.public.japanese.general

Web 開発に関する問題については、http://groups.google.com/group/sql-anywhere-web-development を参照してください。

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。

データベースの作成

この項では、SQL Anywhere データベースを作成する方法について説明します。テーブル、ビュー、マテリアライズド・ビュー、インデックスなどのデータベース・オブジェクトの使用方法について説明します。キーや制約を使用してデータの参照整合性を保つ方法について、および各独立性レベルでのトランザクションの処理方法について説明します。

SQL Anywhere でのデータベースの作成

目次

設計上の考慮事項

SQL Anywhere でデータベースを作成する前に、データベース内のすべてのテーブル (エンティティ)、各テーブル内のカラム (属性)、各テーブル間の関係 (キーおよび制約) を定義します。

データベースの概念データベース・モデル (CDM) の構築を検討します。CDM により、データベースをマップとして視覚化できます。CDM を構築する際には、紙に書くか、または Sybase PowerDesigner などのソフトウェアを使用できます。CDM ツールにより、検証しながら設計を進めることができるため、データベースの作成にも役立ちます。

概念データベース・モデルの構築など、データベースの設計に関する詳細については、http://infocenter.sybase.com/help/index.jsp にアクセスし、PowerDesigner のマニュアルを参照してください。

テーブルやビューなどのデータベース・オブジェクトの詳細については、「データベース・オブジェクトの使用」 15 ページを参照してください。

オブジェクト名の選択

データベース・オブジェクトの名前に予約語を使用しないでください。SQL Anywhere の予約語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、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。

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、「BLOB の格納」 5 ページを参照してください。

NULL か NOT NULL かの決定

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。

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。

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。このマニュアルでは、BLOB という略語のみ使用します。

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 をロー外のテーブル拡張ページに格納するよう指定できます。また、ロー外に格納した 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、あるいは INLINE USE DEFAULT または PREFIX 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 に大きな値を設定し、すべての BLOB をインラインで格納するようにした場合、ローの処理パフォーマンスは低下することがあります。また、PREFIX の値を大きくしすぎると、BLOB の一部を複製したプレフィクス・データのために、BLOB の格納に必要なディスク領域のサイズ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。

圧縮されたカラムのプレフィクス・データは圧縮されずに格納されるため、要求を満たすために必要なすべてのデータがプレフィクス内に格納されている場合は、解凍は必要ありません。

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。

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; のようになります。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もあります。

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。

130 バイト未満の値を含むカラムや、JPG ファイルなどの圧縮形式の値を含むカラムでは、カラムの圧縮を使用しないでください。すでに圧縮されている値を含むカラムを圧縮しようとすると、実際には、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」 85 ページを参照してください。

チュートリアル：SQL Anywhere データベースの作成

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 11 - 紹介』を参照してください。

データベースの設計上の考慮事項に関する詳細については、「設計上の考慮事項」 4 ページを参照してください。

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] - [SQL Anywhere 11] - [データベースの作成]** を選択します。
3. **[ようこそ]** 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. **[メイン・データベース・ファイルを保存]** フィールドに、*c:¥temp¥mysample.db* と入力します。

   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、適切なパスを指定します。
6. **[完了]** をクリックします。
7. **[閉じる]** をクリックします。

参照

- 「設計上の考慮事項」 4 ページ
- 「レッスン 2：データベースへの接続」 8 ページ
- 「レッスン 3 ：データベースへのテーブルの追加」 10 ページ
- 「レッスン 4：カラムへの NOT NULL 制約の設定」 12 ページ
- 「レッスン 5：外部キーの作成」 13 ページ

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 ：データベースへのテーブルの追加」 10 ページを参照してください。

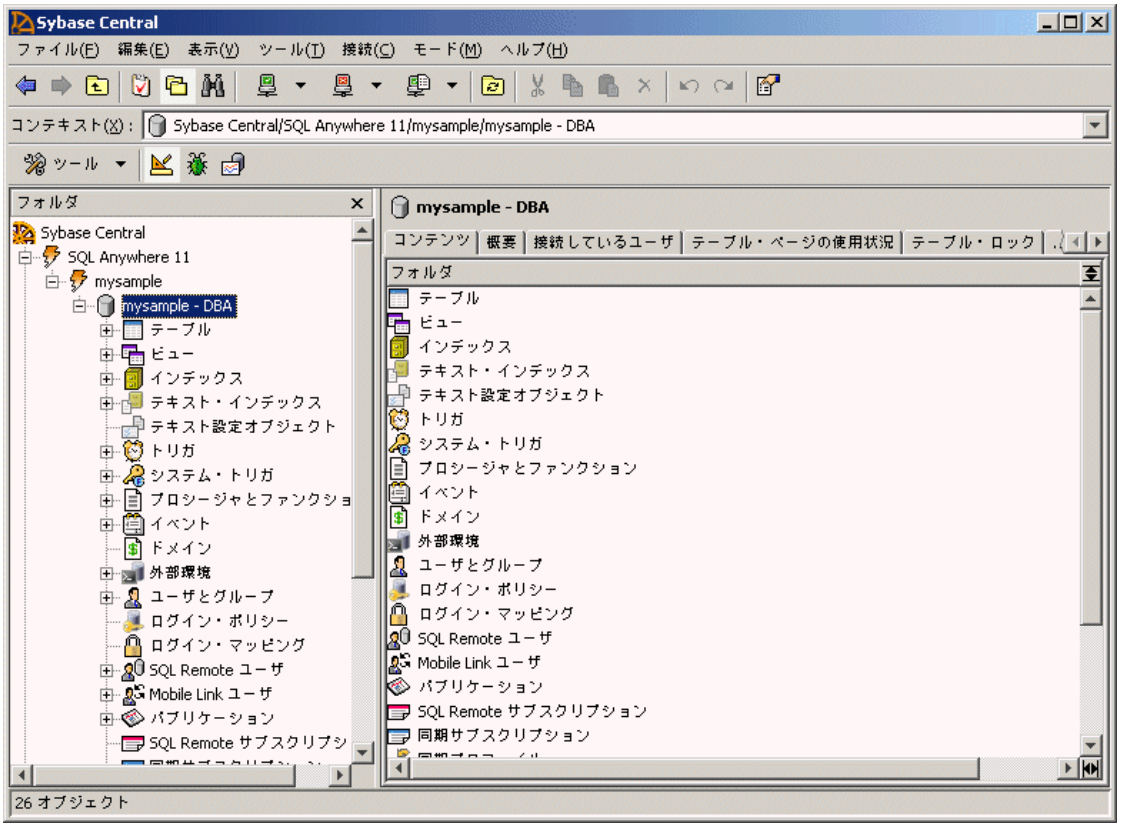
♦ データベースに接続するには、次の手順に従います。

1. Sybase Central を起動します。
2. **[接続] - [SQL Anywhere 11 に接続]** を選択します。
3. **[ユーザ ID]** フィールドに、**DBA** と入力します。これは新しいデータベースのデフォルトのユーザ ID です。
4. **[パスワード]** フィールドに、**sql** と入力します。これは新しいデータベースのデフォルトのパスワードです。
5. デフォルト接続の領域で**[なし]** を選択します。
6. **[データベース]** タブをクリックし、**[データベース・ファイル]** フィールドにデータベース・ファイルのフル・パスを入力します。たとえば、次のように入力します。

   ```
   c:¥temp¥mysample.db
   ```

7. **[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。



参照

- 「設計上の考慮事項」 4 ページ
- 「レッスン 1：データベース・ファイルの作成」 8 ページ
- 「レッスン 3 ：データベースへのテーブルの追加」 10 ページ
- 「レッスン 4：カラムへの NOT NULL 制約の設定」 12 ページ
- 「レッスン 5：外部キーの作成」 13 ページ

レッスン 3 ：データベースへのテーブルの追加

このレッスンでは、Products というテーブルを作成します。

テーブルの設計上の考慮事項に関する詳細については、「設計上の考慮事項」 4 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]** をダブルクリックします。
2. **[テーブル]** を右クリックし、**[新規]** - **[テーブル]** を選択します。
3. **[新しいテーブルの名前を指定してください。]** フィールドに、**Products** と入力します。

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。新しいカラムの **[名前]** フィールドが選択され、新しいカラムの名前を指定するよう求められます。

5. 新しいカラムの名前として、**ProductID** と入力します。

   このカラムがテーブル内の最初のカラムなので、テーブルのプライマリ・キーであることを示す **[プライマリ・キー]** が選択されます。

   テーブルを作成する場合、カラムを作成し、**[プライマリ・キー]** カラムのチェックマークをオンにすることで、複数のカラムから成るプライマリ・キーを作成できます。「プライマリ・キー」『SQL Anywhere 11 - 紹介』を参照してください。

6. **[データ型]** リストで、**[integer]** を選択します。

7. 省略記号 (ピリオド 3 つ) ボタンをクリックします。

8. **[値]** タブをクリックし、**[デフォルト値]** - **[システム定義]** - **[autoincrement]** を選択します。

   autoincrement 値は、テーブルにローが追加されるごとに増加します。これにより、カラムにはユニークな値が設定されます。これはプライマリ・キーの要件です。「プライマリ・キー」『SQL Anywhere 11 - 紹介』を参照してください。

9. **[OK]** をクリックします。

10. **[ファイル]** - **[新規]** - **[カラム]** を選択します。

11. 次のフィールドを完成させます。

    - **[名前]** フィールドに、**ProductName** と入力します。
    - **[データ型]** リストで、**[char]** を選択します。
    - **[サイズ]** リストで、**[15]** を選択します。

12. 次のテーブルをデータベースに追加します。

    - **Customers テーブル**　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。
      - **CustomersID**　各顧客の ID 番号です。**[プライマリ・キー]** が選択されていることを確認し、**[データ型]** を **[integer]** に、**[デフォルト値]** を **[autoincrement]** にそれぞれ設定します。
      - **CompanyName**　各会社の名前です。**[データ型]** を **[char]** に設定し、最大長を **35** 文字にします。
    - **SalesOrders テーブル**　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。
      - **SalesOrdersID**　各注文書の ID 番号です。**[データ型]** を **[integer]** に設定し、**[プライマリ・キー]** が選択されていることを確認します。**[デフォルト値]** に **[autoincrement]** を設定します。
      - **OrderDate**　注文日です。**[データ型]** を **[date]** に設定します。
      - **CustomerID**　発注した顧客の ID 番号です。**[データ型]** を **[integer]** に設定します。

- **SalesOrderItems table**　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。
  - **SalesOrderItemsID**　項目を含む注文書の ID 番号です。**[データ型]** を **[integer]** に設定し、**[プライマリ・キー]** が選択されていることを確認します。
  - **LineID**　各注文書の ID 番号です。**[データ型]** を **[integer]** に設定し、**[プライマリ・キー]** が選択されていることを確認します。

    注意
    [プライマリ・キー] は、SalesOrderItemsID と LineID の両方に設定されているため、この 2 つのカラムの連結値でテーブルのプライマリ・キーが構成されます。
  - **ProductID**　受注製品の ID 番号です。**[データ型]** を **[integer]** に設定します。

13. **[ファイル]** - **[保存]** を選択します。

参照

- 「設計上の考慮事項」 4 ページ
- 「レッスン 1：データベース・ファイルの作成」 8 ページ
- 「レッスン 2：データベースへの接続」 8 ページ
- 「レッスン 4：カラムへの NOT NULL 制約の設定」 12 ページ
- 「レッスン 5：外部キーの作成」 13 ページ

レッスン 4：カラムへの NOT NULL 制約の設定

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。

♦ **カラムに制約を追加する、またはカラムから制約を削除するには、次の手順に従います。**

1. Sybase Central の左ウィンドウ枠で、**[テーブル]** をダブルクリックします。
2. **[MyProducts]** をクリックし、右ウィンドウ枠で **[カラム]** タブをクリックします。
3. **[ProductName]** カラムを選択します。
4. **[ファイル]** - **[プロパティ]** を選択します。
5. **[制約]** タブをクリックし、**[NULL 値を禁止]** を選択します。

   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。
6. **[OK]** をクリックします。

   この制約により、**[Products]** テーブルに追加されたローごとに、**[ProductName]** カラムに値が必要になります。
7. **[ファイル]** - **[保存]** を選択します。

参照

- 「設計上の考慮事項」 4 ページ
- 「レッスン 1：データベース・ファイルの作成」 8 ページ
- 「レッスン 2：データベースへの接続」 8 ページ
- 「レッスン 3 ：データベースへのテーブルの追加」 10 ページ
- 「レッスン 5：外部キーの作成」 13 ページ

レッスン 5：外部キーの作成

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。前のレッスンで作成したテーブルを使用します。

♦ **外部キーを作成するには、次の手順に従います。**

1. Sybase Central の左ウィンドウ枠で、**[テーブル]** をダブルクリックします。
2. 左ウィンドウ枠で、**[SalesOrdersItems]** テーブルをクリックして選択します。
3. 右ウィンドウ枠で、**[制約]** タブを選択します。
4. **[ファイル]** - **[新規]** - **[外部キー]** を選択します。
5. **[この外部キーが参照するテーブルを指定してください。]** リストで、**[Products]** テーブルを選択します。
6. **[新しい外部キーの名前を指定してください。]** フィールドに、**ProductIDkey** と入力します。
7. **[次へ]** をクリックし、**[この外部キーが参照する主キー制約または一意性制約を指定してください。]** で、**[プライマリ・キー]** を選択します。
8. **[外部カラム]** リストで、**[SalesOrdersItemsID]** をクリックします。
9. **[完了]** をクリックします。

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。

参照

- 「設計上の考慮事項」 4 ページ
- 「レッスン 1：データベース・ファイルの作成」 8 ページ
- 「レッスン 2：データベースへの接続」 8 ページ
- 「レッスン 3 ：データベースへのテーブルの追加」 10 ページ
- 「レッスン 4：カラムへの NOT NULL 制約の設定」 12 ページ

データベース・オブジェクトの使用

目次

この項では、データベース・オブジェクトを追加し、データベース・プロパティを設定するための手順を説明します。

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。スキーマは、データベースの論理的なフレームワークです。

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」 873 ページを参照してください。

データベースの作成と設計に関する概念的な情報については、次の各項を参照してください。

- 「SQL Anywhere でのデータベースの作成」 3 ページ
- 「データ整合性の確保」 85 ページ
- 「Sybase Central の使用」 『SQL Anywhere サーバ - データベース管理』
- 「Interactive SQL の使用」 『SQL Anywhere サーバ - データベース管理』

データベース・オブジェクトのプロパティの設定

データベースのプロパティやほとんどのデータベース・オブジェクトは、表示または設定できます。データベースの作成時に選択されたプロパティの中には、設定できないものもあります。

プロパティを表示および設定するには、Sybase Central のプロパティ・ウィンドウを使用します。Sybase Central を使用しない場合は、オブジェクトの作成時に CREATE 文を使用してプロパティを指定します。オブジェクトがすでに存在する場合は、ALTER 文を使用してプロパティを変更します。

♦ 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、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・ウィンドウでも、プロパティを表示、編集できます。プロパティ・ウィンドウを表示するには、オブジェクトを右クリックし、**[プロパティ]** を選択します。

参照

- 「接続、データベース、データベース・サーバのプロパティ」『SQL Anywhere サーバ - データベース管理』

データベースのシステム・オブジェクトの表示

システム・テーブル、システム・ビュー、ストアド・プロシージャ、ドメインなどのシステム・オブジェクトは、データベース・オブジェクトや、データベース・オブジェクト間の関係に関する情報を保持しています。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・オブジェクトを表示するには、次の手順に従います (Sybase Central の場合)。

1. データベース・サーバを起動します。
2. DBA 権限のあるユーザとしてデータベースに接続します。
3. **[ファイル] - [所有者フィルタの設定]**を選択します。
4. **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、SYS と dbo が所有するすべてのテーブルとビューのリストを返します。SYSTAB システム・ビューに対してジョインを行ってオブジェクト名を返し、SYSUSER システム・ビューに対してジョインを行って所有者名を返します。

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

参照

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

テーブルの操作

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 や Sybase Central で SQL 文を使用して、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・テーブルの操作」 34 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. 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. 左ウィンドウ枠で **[テーブル]** を右クリックし、**[新規]** - **[テーブル]** を選択します。
3. **テーブル作成ウィザード**の指示に従います。
4. 右ウィンドウ枠で **[カラム]** タブをクリックして、テーブルを設定します。
5. **[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 権限のある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
);
```

「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」 16 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「ユーザのパーミッションと権限の取り消し」 『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

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。

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。

- **従属した通常のビュー**　テーブルのスキーマを変更すると、データベース内でテーブルの定義が更新されます。従属した通常のビューが存在する場合、データベース・サーバはテーブル変更を実行後にそれらを自動的に再コンパイルします。テーブルのスキーマを変更した後で、データベース・サーバが従属した通常のビューを再コンパイルできない場合、変更によってビュー定義が無効になったことが原因と考えられます。この場合は、ビュー定義を訂正する必要があります。「通常のビューの変更」 45 ページを参照してください。
- **従属したマテリアライズド・ビュー**　従属したマテリアライズド・ビューが存在する場合は、テーブルの変更前にそれらのビューを無効にし、テーブルの変更後にもう一度有効にする必要があります。テーブルのスキーマを変更した後で、従属した非マテリアライズド・ビュー

をもう一度有効にできない場合は、変更によってマテリアライズド・ビュー定義が無効になったことが原因と考えられます。この場合は、マテリアライズド・ビューを削除してから、有効な定義を使用してもう一度作成する必要があります。または、基となるテーブルに適切な変更を加えてから、マテリアライズド・ビューをもう一度有効にしてみてください。「マテリアライズド・ビューの作成」 59 ページ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」 38 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。従属マテリアライズド・ビューがある場合はテーブルの変更は失敗します。あらかじめ従属マテリアライズド・ビューを無効にしておく必要があります。テーブルの変更が完了したら、従属マテリアライズド・ビューをもう一度有効にする必要があります。「ビューの依存性」 38 ページを参照してください。

従属マテリアライズド・ビューがあるかどうかを判断するには、sa_dependent_views システム・プロシージャを使用します。「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。

♦ **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 権限のあるユーザとして、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. テーブルに依存するマテリアライズド・ビューが存在するようなスキーマを変更する場合は、次のようにして各ビューを無効にします。
   a. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
   b. マテリアライズド・ビューを右クリックして、**[無効にする]** を選択します。
3. **[テーブル]** をダブルクリックして、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]** タブをクリックして、テーブルの設定を変更します。
5. **[ファイル]** - **[保存]** を選択します。
6. マテリアライズド・ビューを無効にしたら、各ビューをもう一度有効にし、初期化します。「マテリアライズド・ビューの有効化と無効化」 67 ページを参照してください。

ヒント
テーブルの **[カラム]** タブを選択して **[ファイル]** - **[新しいカラム]** を選択し、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。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

- 「マテリアライズド・ビューの有効化と無効化」 67 ページ
- 「データ整合性の確保」 85 ページ
- 「ビューの依存性」 38 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。従属マテリアライズド・ビューを持つテーブルに対して、ADD FOREIGN KEY 以外の句を ALTER TABLE 文で使用すると、ALTER TABLE 文は失敗します。その他すべての句の場合は、従属マテリアライズド・ビューを無効にし、変更が完了してから、再度有効にする必要があります。「ビューの依存性」 38 ページを参照してください。

従属マテリアライズド・ビューがあるかどうかを判断するには、sa_dependent_views システム・プロシージャを使用します。「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。

♦ **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 権限のあるユーザとしてデータベースに接続します。
2. 従属マテリアライズド・ビューを持つテーブルに対してスキーマ変更操作を実行しており、ADD FOREIGN KEY 以外の句を ALTER TABLE 文で使用する場合、ALTER MATERIALIZED VIEW ... DISABLE 文を使用して各従属マテリアライズド・ビューを無効にします。通常の従属ビュー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。

   データベース内のテーブルの定義が更新されます。
4. 無効にしたマテリアライズド・ビューがある場合は、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』
- 「通常のビューの変更」 45 ページ
- 「マテリアライズド・ビューの有効化と無効化」 67 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」 85 ページ
- 「ビューの依存性」 38 ページ

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。従属した通常のビューが存在する場合、データベース・サーバはテーブル変更を実行後にそれらを再コンパイルしてもう一度有効にしようとします。失敗した場合は、テーブルの削除によってビューの定義が無効になったことが原因と考えられます。この場合は、ビュー定義を訂正する必要があります。「通常のビューの変更」 45 ページを参照してください。

従属したマテリアライズド・ビューが存在する場合、その定義は有効でなくなっているため、以降のリフレッシュは失敗します。この場合は、マテリアライズド・ビューを削除してから、有効な定義を使用してもう一度作成する必要があります。「マテリアライズド・ビューの作成」 59 ページを参照してください。

テーブルを変更する前に、sa_dependent_views システム・プロシージャを使用して、そのテーブルに依存するビューがあるかどうかを判断できます。「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」 38 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 権限のあるユーザとして、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. マテリアライズド・ビューが依存するテーブルを削除している場合は、各マテリアライズド・ビューを無効にします。
   a. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
   b. マテリアライズド・ビューを右クリックして、**[無効にする]** を選択します。
3. **[テーブル]** をダブルクリックします。
4. テーブルを右クリックして、**[削除]** を選択します。
5. **[はい]** をクリックします。

♦ 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 権限のあるユーザとして、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. マテリアライズド・ビューが依存するテーブルを削除している場合は、ALTER MATERIALIZED VIEW ... DISABLE 文を使用して各マテリアライズド・ビューを無効にします。
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。「DROP 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。

参照

- 「Interactive SQL の使用」『SQL Anywhere サーバ - データベース管理』

プライマリ・キーの管理

「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ます。このカラムの値でテーブル内のユニークなローが識別されます。プライマリ・キーの値は、ローのデータが存在するかぎり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、テーブル内のユニークなローを識別します。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・キー設定ウィザード**が開始されます。**プライマリ・キー設定ウィザード**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. DBA 権限のあるユーザとして、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. 左ウィンドウ枠で、**[テーブル]** をダブルクリックします。
3. テーブルを右クリックして、**[プライマリ・キーの設定]** を選択します。
4. **プライマリ・キー設定ウィザード**の指示に従います。

プライマリ・キーの管理 (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 権限のある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。

♦ **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 権限のある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 権限のある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、Skills という名前のテーブルを作成し、SkillID カラムをプライマリ・キーとして割り当てます。

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

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

プライマリ・キーに SkillID カラムと Skillname カラムを組み合わせて使用するようにプライマリ・キーを変更する場合は、作成したプライマリ・キーを削除してから、新しいプライマリ・キーを追加する必要があります。

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

「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)」 25 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。

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

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. DBA 権限のあるユーザとして、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. 左ウィンドウ枠で、**[テーブル]** をダブルクリックします。
3. テーブルを右クリックして、**[新規]** - **[外部キー]** を選択します。
4. **外部キー作成ウィザード**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. DBA 権限のあるユーザとして、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. 左ウィンドウ枠で、**[テーブル]** をダブルクリックします。
3. 外部キーを削除するテーブルを選択します。
4. 右ウィンドウ枠で、**[制約]** タブをクリックします。
5. 外部キーを右クリックして、**[削除]** を選択します。
6. **[はい]** をクリックします。

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. DBA 権限のあるユーザとして、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. 左ウィンドウ枠で、**[テーブル]** をダブルクリックします。
3. テーブルをクリックします。
4. 右ウィンドウ枠で、**[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)

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、外部キーは複数定義できます。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。

♦ **外部キーを作成するには、次の手順に従います (SQL の場合)。**

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. ALTER TABLE 文を実行します。

外部キー作成時でのカラム名の省略 (SQL の場合)

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

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

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

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

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

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

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

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

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

例

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

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

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

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

外部キーを作成するとき、そのプロパティを指定できます。たとえば、次の文は例 2 と同じ外部キーを作成しますが、この外部キーは、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;
```

参照

- 「外部キーの管理 (Sybase Central の場合)」 27 ページ
- 「CREATE TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「ALTER TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』

計算カラムの使用

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

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

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

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

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

しかし、上記のクエリで、WHERE 句の述部は単一ベースのカラムを参照しないため、検索引数が使用できません。「クエリでの述部の使用」 598 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。インデックス検索を向上させるには、次のように Shipments テーブルに AverageCost という名前の計算カラムを作成してから、そのカラムにインデックスを作成します。

```
ALTER TABLE Shipments
  ADD AverageCost DECIMAL(21,13)
  COMPUTE( TotalPrice / Quantity );
CREATE INDEX IDX_average_cost
  ON Shipments( AverageCost ASC );
```

計算カラムのタイプを選択することは重要です。クエリ内の式のデータ型が計算カラムのデータ型と正確に一致した場合、SQL Anywhere のオプティマイザは複雑な式のみを計算カラムに置き換えます。式のタイプを判別するために、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 decimal(21,13) を返します。最適化中に、SQL Anywhere オプティマイザは上記のクエリを次のように書き換えます。

```
SELECT *
  FROM Shipments
```

```
WHERE AverageCost
BETWEEN 2.00 AND 4.00;
```

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

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。
- **カラムの依存性**　たとえば、NULL 値を NULL 値以外の値に変更するなど、計算カラムの定義で参照されるカラムの値を設定するときにトリガを使用しないことを強くおすすめします。これは、計算カラムの値が意図した計算内容を反映していない可能性があるためです。
- **カラム名をリストする**　計算カラムのあるテーブルに対する INSERT 文では、常にカラム名を明示的に指定してください。
- **トリガ**　計算カラムにトリガを定義すると、そのカラムに影響するすべての INSERT 文または UPDATE 文はトリガを起動します。

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

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

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

計算カラムの再計算

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

計算カラムは、次の状況で再計算されます。

- いずれかのカラムが削除、追加、または名前が変更された場合
- テーブル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

計算カラムは、次の状況では再計算**されません**。

- 計算カラムが問い合わせされている。
- 計算カラムが依存するカラムで値が変更されている。

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。「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 を使用していない場合は、次の手順に従います。

- 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。

テンポラリ・テーブルの操作

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。

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』

ローカル・テンポラリ・テーブル

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

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

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

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

- 明示的に削除されるまで、またはデータベースが停止するまで、テーブルのコンテンツは持続する。
- データベースの起動時、テーブルは空である。
- テーブルでのローのロック処理動作は、ベース・テーブルの場合と同じである。

非トランザクション指向のテンポラリ・テーブル

テンポラリ・テーブルを非トランザクション指向として宣言するには、CREATE TABLE 文の NOT TRANSACTIONAL 句を使用します。状況によっては、NOT TRANSACTIONAL 句を使用するとパフォーマンスが向上します。これは、トランザクション単位でないテンポラリ・テーブルでの操作では、ロールバック・ログにエントリが作成されないためです。たとえば、テンポラ

リ・テーブルを使用するプロシージャが COMMIT や ROLLBACK の介入を受けずに繰り返し呼び出される場合や、テーブルに多くのローが含まれる場合は、NOT TRANSACTIONAL が有用です。非トランザクション指向テンポラリ・テーブルへの変更は、COMMIT または ROLLBACK の影響を受けません。

テンポラリ・テーブルの作成

テンポラリ・テーブルは、SQL 文または Sybase Central を使用して作成できます。

♦ **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. DBA 権限のあるユーザとして、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. **[テーブル]** を右クリックし、**[新規]** - **[グローバル・テンポラリ・テーブル]** を選択します。
3. **グローバル・テンポラリ・テーブル作成ウィザード**の指示に従います。
4. 右ウィンドウ枠で **[カラム]** タブをクリックして、テーブルを設定します。
5. **[ファイル]** - **[保存]** を選択します。

♦ **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 権限のあるユーザとしてデータベースに接続します。
2. CREATE TABLE 文または DECLARE LOCAL TEMPORARY TABLE 文を実行します。

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』

プロシージャ内でのテンポラリ・テーブルの参照

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

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

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

これは、procA が呼び出されたときに temp_table が作成され、その後で sharedProc が呼び出されたためです。sharedProc が呼び出されたときに、内部の SELECT 文が解析されて検証され、その

後で、SELECT 文がもう一度実行されたときのために解析された文の表現がキャッシュされました。キャッシュされたバージョンは、procA のテーブル定義のカラムの順序を反映しています。

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

次のいずれかを実行すると、このような状況の発生を防ぐことができます。

- このような方法で使用されるテンポラリ・テーブルは、一致するように定義する
- 代わりにグローバル・テンポラリ・テーブルを使用することを検討する

ビューの操作

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

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

表記の規則

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

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

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

通常のビュー、マテリアライズド・ビュー、ベース・テーブルの比較

次の表に、通常のビュー、マテリアライズド・ビュー、ベーステーブルの比較を示します。

機能	通常のビュー	マテリアライズド・ビュー	ベース・テーブル
アクセス・パーミッションの許可	○	○	○
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	×	○	○

機能	通常のビュー	マテリアライズド・ビュー	ベース・テーブル
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**　通常のビューでは、データを格納するために追加の記憶領域は必要ありません。呼び出しごとに再計算されます。マテリアライズド・ビューはディスク領域を必要としますが、呼び出しごとに再計算する必要はありません。データベースのサイズが大きく、同じテーブルに対して頻繁に繰り返し発生するジョイン要求をデータベース・サーバが処理するような環境では、マテリアライズド・ビューを使用すると、応答時間が向上します。
- **セキュリティの向上**　関連データへのアクセスだけが許可されます。
- **利便性の向上**　ベース・テーブルよりも見やすい形でユーザとアプリケーション開発者にデータを提供します。
- **一貫性の向上**　よく使われるクエリの定義をデータベース内で集中管理します。

ビューの依存性

ビュー定義は、カラム、テーブル、他のビューなど、その他のオブジェクトを参照することがあります。ビューが別のオブジェクトを参照している場合、そのビューは「参照元のオブジェクト」、ビューが参照しているオブジェクトは「参照先のオブジェクト」と呼ばれます。また、参照元のオブジェクトは、参照先のオブジェクトに「依存している」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、そのビューが直接的または間接的に参照するすべてのオブジェクトが含まれます。たとえば、ビューはあるテーブルを参照する別のビューを参照することにより、間接的にそのテーブルを参照できます。

次のテーブルとビューのセットを考えてみます。

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

次のビューの依存性は、上記の定義から特定できます。

- ビュー v1 は t1 の各カラムと t1 自体に依存しています。
- ビュー v2 は、t2.c3 と t2 自体に依存しています。

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

データベース・サーバは指定されたビューが参照するカラム、テーブル、ビューを追跡します。データベース・サーバではこの依存性情報を使用して、参照先オブジェクトのスキーマ変更が使用できない状態で参照元ビューに残らないようにします。次の表では、ビューの依存性が通常のビューとマテリアライズド・ビューに与える影響について説明します。

依存性とスキーマ変更

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

- テーブル、ビュー、マテリアライズド・ビュー、またはカラムの削除
- テーブル、ビュー、マテリアライズド・ビュー、またはカラムの名前変更
- カラムの追加、削除、または変更
- カラムのデータ・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的または間接的に依存するビューのリストを生成します。DISABLED ステータスのビューは無視されます。

   いずれかの従属ビューがマテリアライズド・ビューである場合、要求は失敗し、エラーが返され、残りのイベントは発生しません。従属したマテリアライズド・ビューを明示的に無効にしてから、スキーマ変更操作を続行してください。「マテリアライズド・ビューの有効化と無効化」 67 ページを参照してください。
2. データベース・サーバは、変更するオブジェクトとすべての従属した通常のビューに対して、排他スキーマ・ロックを取得します。
3. データベース・サーバはすべての従属した通常のビューのステータスを INVALID に設定します。
4. データベース・サーバはスキーマ変更操作を実行します。操作が失敗すると、ロックは解放され、従属した通常のビューのステータスは VALID にリセットされます。さらにエラーが返され、残りの手順は発生しません。
5. データベース・サーバは従属した通常のビューを再コンパイルし、成功した場合は各ビューのステータスを VALID に設定します。いずれかの通常のビューでコンパイルが失敗した場合も、そのビューのステータスは INVALID のままです。INVALID である通常のビューに対する以後の要求により、データベース・サーバはビューを再コンパイルしようとします。それらの試行に失敗した場合は、INVALID ビューまたはそのビューが依存するオブジェクトを変更する必要があります。

依存性とスキーマ変更 (通常のビュー)

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

依存性とスキーマ変更 (マテリアライズド・ビュー)

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

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示します。「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。

通常のビューの操作

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

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。

通常のビューの SELECT 文に対する制限

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

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

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

通常のビューの更新

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

- UNION 句
- DISTINCT 句
- GROUP BY 句
- FIRST または TOP 句
- 集合関数
- FROM 句に複数のテーブル (ansi_update_constraints オプションが Strict または Cursors に設定されている場合)。「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。
- ORDER BY 句 (ansi_update_constraints オプションが Strict または Cursors に設定されている場合)。「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。

- すべての *select* リスト項目がベース・テーブルのカラムではない

通常のビューのコピー

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。

WITH CHECK OPTION オプションの使用

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「ベース・テーブル 'Employees' の挿入／更新に対して WITH CHECK OPTION が違反しています。」というエラーを返します。

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

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

参照

- 「SELECT 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「クエリ結果の要約、グループ化、ソート」 391 ページ
- 「マテリアライズド・ビューの操作」 51 ページ

通常のビューのステータス

通常のビューには、ステータスが関連付けられています。ステータスは、データベース・サーバが使用するビューの利用可能性を反映しています。すべてのビューのステータスを表示するには、Sybase Central の左ウィンドウ枠で **[ビュー]** を選択し、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、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。

通常のビューのステータスの種類について、次に説明します。

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

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

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

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

  通常のビューは、次の場合にこのステータスになります。

- ビューを明示的に無効にした場合。ALTER VIEW ... DISABLE 文を実行した場合など。
- そのビューが依存するビュー (マテリアライズド・ビューまたは非マテリアライズド・ビュー) を無効にした場合。
- テーブルのビューの依存性を無効にした場合。ALTER TABLE ... DISABLE VIEW DEPENDENCIES 文を実行した場合など。

通常のビューの有効化と無効化については、「通常のビューの有効化と無効化」 47 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。「SYSOBJECT 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。

通常のビューの作成

通常のビューの作成時に、データベース・サーバはビュー定義をデータベースに格納します。ただしビューのデータは格納されません。ビュー定義は、そのビュー定義が参照される場合で、かつそのビューの使用中のみ実行されます。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。

♦ **新しい通常の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. 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. 左ウィンドウ枠で **[ビュー]** を右クリックし、**[新規] - [ビュー]** を選択します。
3. **ビュー作成ウィザード**の指示に従います。
4. 右ウィンドウ枠で、**[SQL]** タブをクリックし、定義のコードを編集します。変更を保存するには、**[ファイル] - [保存]** を選択します。

♦ **新しい通常の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;
```

「CREATE VIEW 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

通常のビューの変更

通常のビューは 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。

ALTER VIEW 文を使用して別のユーザが所有するマテリアライズド・ビューを変更する場合は、所有者を含めて名前を修飾する必要があります (たとえば、GROUPO.EmployeeConfidential)。名前を修飾しなかった場合、データベース・サーバは、ユーザ本人が所有する同名のビューを検索して変更します。見つからない場合は、エラーが返されます。

ビューの変更とビューの依存性

通常の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、変更後に追加の操作が必要なことがあります。たとえば、ビューの変更後、データベース・サーバはそのビューを自動的に再コンパイルして、データベース・サーバが使用できるように有効にします。従属した通常のビューが存在する場合、データベース・サーバはそれらも無効にしてからもう一度有効にします。有効にできない場合、ステータスは INVALID になるため、通常のビューの定義と従属した通常のビューの定義が一貫性を保つようにする必要があります。

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

基本となるオブジェクトに対するスキーマの変更によってビューが受ける影響の詳細については、「ビューの依存性」 38 ページを参照してください。

♦ 通常のビューを変更するには、次の手順に従います (Sybase Central の場合)。

1. DBA 権限のあるユーザとして、または通常のビューの所有者として、データベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. ビューを選択します。
4. 右ウィンドウ枠で、**[SQL]** タブをクリックし、定義のコードを編集します。

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

5. **[ファイル] - [保存]** を選択します。

♦ **通常のビューを変更するには、次の手順に従います (SQL の場合)。**

1. DBA 権限のあるユーザとして、または通常の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、通常のビューを変更する場合は、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、通常の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;
```

「ALTER VIEW 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

通常のビューの削除

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

従属ビューを持つ通常のビューを削除すると、削除操作の一環としてその従属ビューは INVALID になります。従属ビューは、変更されるか、元の削除済みビューが再作成されるまで、使用できません。「通常のビューの変更」 45 ページを参照してください。

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

基本となるオブジェクトに対する変更によって通常のビューが受ける影響の詳細については、「ビューの依存性」 38 ページを参照してください。

♦ **通常の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 権限のあるユーザとして、または通常のビューの所有者として、データベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. ビューを右クリックして、**[削除]** を選択します。
4. **[はい]** をクリックします。

♦ **通常の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 権限のあるユーザとして、または通常の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 という名前の通常のビューを削除します。

```
DROP VIEW DepartmentSize;
```

「DROP VIEW 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

通常のビューの有効化と無効化

この項では、通常のビューの有効化と無効化について説明します。マテリアライズド・ビューの有効化と無効化については、「マテリアライズド・ビューの有効化と無効化」 67 ページを参照してください。

通常のビューを有効または無効にすることで、データベース・サーバがそのビューを使用できるかどうかを制御できます。通常の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、クエリを満たすときにそのビューを使用でません。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、そのビューを直接的または間接的に参照する他のビューも自動的に無効になります。このため、ビューを再度有効にしたら、無効にした時点でそのビューに依存していた他のすべてのビューを、もう一度有効にする必要があります。ビューを無効にする前に sa_dependent_views システム・プロシージャを使用することで、従属ビューのリストを特定できます。「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。

通常のビューを有効にすると、データベース・サーバはデータベース内に格納されたビューの定義を使用してそのビューを再コンパイルします。コンパイルが成功すると、ビューのステータスが VALID に変更されます。再コンパイルに失敗した場合、1 つ以上の参照先オブジェクトでスキーマが変更された可能性があります。その場合は、ビュー定義と参照先のオブジェクトのどちらかを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
通常のビューを有効にする前に、そのビューが参照しているビューをもう一度有効にしたり、無効にしたりする必要があります。

無効にされたオブジェクトのパーミッションを付与できます。無効にされたオブジェクトのパーミッションはデータベースに格納され、オブジェクトが有効になったときに有効になります。

♦ **通常の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 権限のあるユーザとして、または通常のビューの所有者として、データベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. ビューを右クリックして、**[無効にする]** を選択します。

♦ **通常の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 権限のあるユーザとして、または通常の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、GROUPO が所有する通常のビュー ViewSalesOrders が無効になります。

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

♦ **通常の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 権限のあるユーザとして、または通常のビューの所有者として、データベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. ビューを右クリックして、**[再コンパイルして有効にする]** を選択します。

♦ **通常の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 権限のあるユーザとして、または通常の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。

例

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

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

参照

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

通常のビュー内のデータのブラウズ

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。「データのクエリ」 305 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。

参照

- 「Interactive SQL の使用」 『SQL Anywhere サーバ - データベース管理』

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、システム・ビューを問い合わせ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。いくつかの例外を除いて、各システム・テーブルには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、ISYSTAB システム・テーブルのデータを表示するには、SYSTAB システム・ビューを問い合わせます。

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、「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。

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 権限のあるユーザとしてデータベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. システム・テーブルに対応するビューを選択します。
4. 右ウィンドウ枠で、**[データ]** タブをクリックします。

♦ 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 権限のあるユーザとしてデータベースに接続します。
2. 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 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 TEXT
ESCAPES OFF
QUOTE '';
```


マテリアライズド・ビューの操作

「マテリアライズド・ビュー」とは、ベース・テーブルとよく似ていて、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、マテリアライズド・ビューはビューでもあり (カタログに格納されたクエリ指定がある)、テーブルでもあります (永続的な実体化したローがある)。したがって、テーブルで実行する多くの操作は、マテリアライズド・ビューでも実行できます。たとえば、マテリアライズド・ビューに対して、インデックス構築やアンロードを実行できます。

負荷の高い集約操作やジョイン操作などを含むクエリのように、頻繁に実行されるコストの高いクエリでは、マテリアライズド・ビューを使用することを検討してください。マテリアライズド・ビューには、集約されジョインされたデータを格納するクエリ可能な構造体があります。マテリアライズド・ビューは、データベースのサイズが大きく、頻繁にクエリが行われるために大量のデータで繰り返し集約やジョイン操作が発生するような環境において、パフォーマンスが向上するように設計されています。たとえば、マテリアライズド・ビューは、データ・ウェアハウス・アプリケーションでの使用に適しています。

マテリアライズド・ビューは、参照先となるベース・テーブルからのデータを使用して、事前に計算されます。また、マテリアライズド・ビューは読み込み専用であるため、データ変更操作 (INSERT、LOAD、DELETE、UPDATE など) を適用できません。

マテリアライズド・ビューのカラム統計は、テーブルの場合と同じ方法で生成され、管理されます。「オプティマイザの推定とカラム統計」 591 ページを参照してください。

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

参照

- 「マテリアライズド・ビューによるパフォーマンスの向上」 603 ページ
- 「CREATE MATERIALIZED VIEW 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「ALTER MATERIALIZED VIEW 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「REFRESH MATERIALIZED VIEW 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「sa_materialized_view_info システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』
- 「sa_materialized_view_can_be_immediate システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』
- 「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』

手動マテリアライズド・ビューと即時マテリアライズド・ビュー

マテリアライズド・ビューには、手動と即時の 2 種類があります。これらはマテリアライズド・ビューの「再表示タイプ」を表しています。

- **手動ビュー**　手動のマテリアライズド・ビューまたは「手動ビュー」とは、再表示タイプが MANUAL REFRESH と定義されているマテリアライズド・ビューです。手動ビューは、たとえば REFRESH MATERIALIZED VIEW 文または sa_refresh_materialized_views システム・プロ

シージャを使用して再表示を明示的に要求するまでリフレッシュされないため、手動ビューのデータが古くなる場合があります。デフォルトでは、マテリアライズド・ビューを作成すると手動ビューになります。

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

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

  また、即時ビューがリフレッシュされるときは、変更が必要なローだけが影響を受けます。この点が手動ビューのリフレッシュとは異なります。手動ビューのリフレッシュでは、リフレッシュするためにすべてのデータが削除され、再作成されます。

手動ビューを即時ビューに、または即時ビューを手動ビューに変更できます。ただし、手動ビューから即時ビューへの変更処理では、少し多くの手順があります。「手動ビューから即時ビューへの変更」 64 ページを参照してください。

マテリアライズド・ビューの再表示タイプを変更すると、特に手動ビューから即時ビューに変更する場合は、ビューのステータスとプロパティに影響を与える場合があります。「マテリアライズド・ビューのステータスとプロパティ」 54 ページを参照してください。

データベースからのマテリアライズド・ビュー情報の取得

- **ステータスとプロパティの情報**　マテリアライズド・ビュー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。

  「マテリアライズド・ビューのステータスとプロパティ」 54 ページも参照してください。

- **データベース・オプションの情報**　SYSMVOPTION システム・ビューを問い合わせると、マテリアライズド・ビューの作成時に一緒に格納されたデータベース・オプションを取得できます。次の最初の文はマテリアライズド・ビューを作成します。次の文で、データベースに問い合わせて、ビューの作成時に使用されたデータベース・オプションを特定します。

```
CREATE MATERIALIZED VIEW EmployeeConfid15 AS
  SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary, ManagerID,
    Departments.DepartmentName, Departments.DepartmentHeadID
  FROM Employees, Departments
  WHERE Employees.DepartmentID=Departments.DepartmentID;

SELECT option_name, option_value
FROM SYSMVOPTION JOIN SYSMVOPTIONNAME
WHERE SYSMVOPTION.view_object_id=(
  SELECT object_id FROM SYSTAB
```



```
    WHERE table_name='EmployeeConfid15' )
ORDER BY option_name;
```

警告
この例を実行し終わったら、作成したマテリアライズド・ビューを削除してください。そうしないと、他の例を試すときに、基本となるテーブル Employees および Departments に対するスキーマ変更ができなくなります。有効化されている従属マテリアライズド・ビューを持つテーブルのスキーマは変更できません。「マテリアライズド・ビューの削除」 73 ページを参照してください。

- **依存性の情報**　マテリアライズド・ビュー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。

マテリアライズド・ビューを使用する状況

マテリアライズド・ビューを使用する前に、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**　マテリアライズド・ビューにはベース・テーブルからのデータの複製が含まれるため、作成するマテリアライズド・ビューのサイズ分の領域をデータベースのディスク上に追加で割り付ける必要があることがあります。得られる利点がマテリアライズド・ビューの使用コストと釣り合うように、追加領域の要件は慎重に検討する必要があります。
- **保守コストとデータの最新性の要件**　マテリアライズド・ビューのデータは、基本となるテーブルのデータが変更されたときに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、マテリアライズド・ビュー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**　データに対して頻繁な変更や大規模な変更が行われると、手動ビューが古くなります。データの最新性が重要な場合は、即時ビューの使用を検討します。
  - **リフレッシュのコスト**　各マテリアライズド・ビューの基本となるクエリの複雑さや関係するデータの量に応じて、リフレッシュに必要な計算のコストが非常に高くなることがあります。そのためマテリアライズド・ビューが頻繁にリフレッシュされると、データベース・サーバが耐えられないほどの負荷がかかる可能性があります。さらに、マテリアライズド・ビューはリフレッシュ操作中は使用できません。
  - **アプリケーションのデータ最新性の要件**　データベース・サーバが古いマテリアライズド・ビューを使用すると、アプリケーションに対して古いデータを提示することになります。古いデータとは、基本となるテーブル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、マテリアライズド・ビュー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。マテリアライズド・ビュー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マテリアライズド・ビューに対する古さのしきい値の設定」 71 ページを参照してください。

- **データの一貫性の要件**　マテリアライズド・ビューをリフレッシュするときは、マテリアライズド・ビュー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マテリアライズド・ビューを検討することを検証してください。特定のクエリで使用されるマテリアライズド・ビューのリストは、Interactive SQL でクエリのグラフィカルなプランの **[高度な詳細]** ウィンドウで確認できます。「実行プランの解釈」 642 ページと「マテリアライズド・ビューによるパフォーマンスの向上」 603 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マテリアライズド・ビューが検討されたかどうかを判断できます。トレーシングはオンにする必要があります。また、オプティマイザによって列挙されるアクセス・プランを確認できるように、OPTIMIZATION_LOGGING トレーシング・タイプを含めるように設定してください。「アプリケーション・プロファイリング」 191 ページと「診断トレーシング・レベルの選択」 207 ページを参照してください。

参照

- 「手動マテリアライズド・ビューと即時マテリアライズド・ビュー」 51 ページ
- 「マテリアライズド・ビューによるパフォーマンスの向上」 603 ページ

マテリアライズド・ビューのステータスとプロパティ

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

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

また、Sybase Central の **[ビュー]** フォルダを選択して個々のビュー詳細を調べるか、SYSTAB および SYSVIEW システム・ビューを問い合わせることにより、マテリアライズド・ビューの情報を表示できます。「SYSTAB システム・ビュー」 『SQL Anywhere サーバ - SQL リファレンス』と「SYSVIEW 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。

マテリアライズド・ビューのステータス

マテリアライズド・ビューには、次の 2 種類のステータスがあります。

- **有効**　マテリアライズド・ビューは、コンパイルが正常に実行され、データベース・サーバが使用できる場合に「有効」というステータスになります。有効にされたマテリアライズド・ビューにはデータがない場合もあります。たとえば、有効にされたマテリアライズド・ビューからデータをトランケートすると、有効に変わりますが、初期化されていません。マテリアライズド・ビューを初期化することはできますが、マテリアライズド・ビューの定義を満たす基本となるテーブルにデータがない場合は、空になります。これは、初期化されていないためにデータがないマテリアライズド・ビューと同じではありません。
- **無効**　マテリアライズド・ビューは、ALTER MATERIALIZED VIEW ... DISABLE 文を実行するなどして、明示的に無効にした場合だけ「無効」というステータスになります。マテリアライズド・ビューを無効にすると、そのビューのデータとインデックスは削除されます。また、即時ビューを無効にすると、手動ビューに変わります。

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

マテリアライズド・ビューの有効化と無効化については、「マテリアライズド・ビューの有効化と無効化」 67 ページを参照してください。

マテリアライズド・ビューのプロパティ

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

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

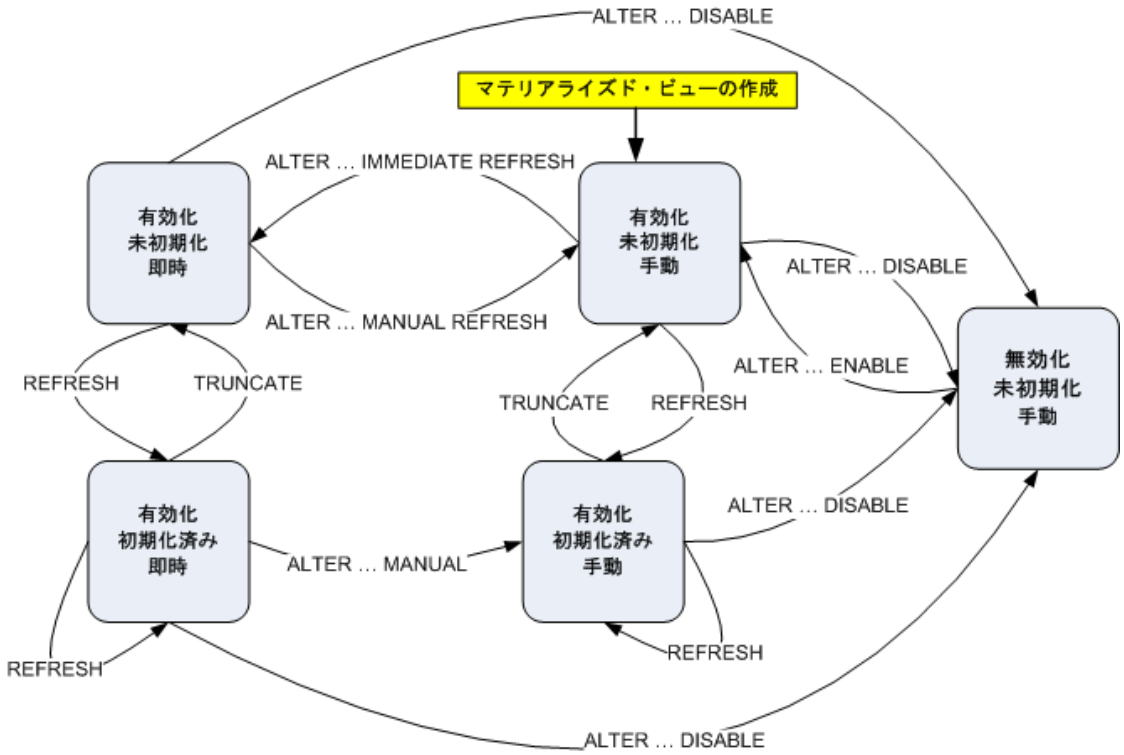
各プロパティの可能な値のリストについては、「sa_materialized_view_info システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

手動ビューを即時ビューに変換できるかどうかを示すプロパティはありませんが、sa_materialized_view_can_be_immediate システム・プロシージャを使用してこれを判断できます。「sa_materialized_view_can_be_immediate システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

マテリアライズド・ビューの変更、リフレッシュ、トランケート実行時のステータスとプロパティの変更

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

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



図から分かる重要な概念は次のとおりです。

- マテリアライズド・ビューを作成すると、そのビューは有効な手動ビューになり、未初期化状態である (データは含まれない)。
- 未初期化状態のビューをリフレッシュすると、初期化された状態になる (データが入力される)。
- 手動ビューから即時ビューに変更するにはいくつかの手順が必要である。また、即時ビューには追加の制限がある。「手動ビューから即時ビューへの変更」 64 ページと「即時ビューの追加の制限」 58 ページを参照してください。

- マテリアライズド・ビューを無効にすると、次のことが行われる。
  - データが削除される
  - ビューが未初期化状態に戻る
  - インデックスが削除される
  - 即時ビューが手動に戻る

参照

- 「マテリアライズド・ビューの操作」 51 ページ
- 「CREATE MATERIALIZED VIEW 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「ALTER MATERIALIZED VIEW 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「REFRESH MATERIALIZED VIEW 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「TRUNCATE 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「DROP MATERIALIZED VIEW 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「マテリアライズド・ビューの制限」 57 ページ
- 「手動マテリアライズド・ビューと即時マテリアライズド・ビュー」 51 ページ
- 「SYSOBJECT システム・ビュー」 『SQL Anywhere サーバ - SQL リファレンス』

マテリアライズド・ビューの制限

マテリアライズド・ビューを作成、初期化、リフレッシュ、ビュー・マッチングする場合は、次の制限が適用されます。

- マテリアライズド・ビューを作成するときは、マテリアライズド・ビューの定義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。
- マテリアライズド・ビューを作成するときは、マテリアライズド・ビューの定義に次の項目を含めることはできません。
  - 他のビュー (マテリアライズド・ビューまたは通常のビュー) に対する参照
  - リモート・テーブルまたはテンポラリ・テーブルに対する参照
  - CURRENT USER などの変数。すべての式は確定的でなければなりません
  - ストアド・プロシージャ、ユーザ定義関数、または外部関数の呼び出し
  - 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、マテリアライズド・ビューを作成するときに特定の設定が必要です。そのように設定しない場合は、エラーが返されます。これらのデータベース・オプションの値は、オプティマイザが使用するビューにも必要です。

  - ansinull=On
  - conversion_error=On
  - sort_collation=Null
  - string_rtruncation=On

- 次のデータベース・オプションは、マテリアライズド・ビューを作成するときにマテリアライズド・ビューごとに格納されます。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マテリアライズド・ビューで格納された値と一致する必要があります。

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

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

マテリアライズド・ビュー定義での ORDER BY 句の指定

マテリアライズド・ビューは、ローが特定の順序で格納されない点がベース・テーブルに似ています。データベース・サーバは、データの計算時に最も効率の良い方法でローを並べます。そのため、マテリアライズド・ビュー定義で ORDER BY 句を指定すると、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。

マテリアライズド・ビューと、オプティマイザによるビュー・マッチングについては、「マテリアライズド・ビューによるパフォーマンスの向上」 603 ページを参照してください。

即時ビューの追加の制限

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

> **注意**
> sa_materialized_view_can_be_immediate システム・プロシージャを使用して、手動ビューを即時ビューにする条件がそろっているかどうかを調べることができます。
> 「sa_materialized_view_can_be_immediate システム・プロシージャ」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- ビューは未初期化状態である必要がある。「マテリアライズド・ビューのステータスとプロパティ」 54 ページを参照してください。
- ビューには、NULL 入力不可のカラムにユニーク・インデックスがある必要がある。無い場合は追加する必要があります。「インデックスの作成」 78 ページを参照してください。
- ビューの定義がグループ化されたクエリである場合、ユニーク・インデックスのカラムは、集合関数ではない select リスト項目に対応する必要がある。
- ビューの定義に次のものを含むことはできない。
  - NULL 入力可の式に対する SUM 関数
  - GROUPING SETS 句
  - CUBE 句
  - ROLLUP 句
  - LEFT OUTER JOIN (左外部ジョイン) 句
  - RIGHT OUTER JOIN (右外部ジョイン) 句
  - FULL OUTER JOIN 句
  - DISTINCT 句
  - ロー制限句
  - 非決定的な式
  - セルフ・ジョインと再帰ジョイン
- ビューの定義は、単一のプロジェクト選択ジョインまたはグループ化されたプロジェクト選択ジョインのブロックである必要があり、グループ化されたプロジェクト選択ジョインのクエリ・ブロックには HAVING 句を含めることはできない。
- グループ化されたプロジェクト選択ジョインのクエリ・ブロックには select リストに COUNT(*) を含める必要があり、SUM と COUNT 集合関数のみを使用できます。

  これらの構造体の詳細については、「マテリアライズド・ビュー検査」 606 ページを参照してください。
- select リストの集合関数は、複雑な式の中では参照できない。たとえば、SUM(expression) + 1 は select リストで使用できません。

マテリアライズド・ビューの作成

マテリアライズド・ビューを作成すると、その定義はデータベースに格納されます。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。マテリアライズド・ビューの作成に成功したら、REFRESH MATERIALIZED VIEW 文を使用してデータを設定します。データの設定は、ビューの「初期化」とも呼ばれます。「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

マテリアライズド・ビューを作成、初期化、またはリフレッシュする前に、マテリアライズド・ビューの制限をすべて満たしていることを確認してください。「マテリアライズド・ビューの制限」 57 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マテリアライズド・ビュー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。

マテリアライズド・ビューの定義の作成が終了すると、Sybase Central の **[ビュー]** フォルダに表示されます。

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 11 - 紹介』

♦ **マテリアライズド・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. DBA 権限または RESOURCE 権限のあるユーザとしてデータベースに接続します。
2. 左ウィンドウ枠で **[ビュー]** を右クリックし、**[新規]** - **[マテリアライズド・ビュー]** を選択します。
3. **マテリアライズド・ビュー作成ウィザード**の指示に従います。
4. マテリアライズド・ビューにデータが含まれるように、初期化します。「マテリアライズド・ビューの初期化」 61 ページを参照してください。

警告
この例を実行し終わったら、作成したマテリアライズド・ビューを削除してください。そうしないと、他の例を試すときに、基本となるテーブル Employees および Departments に対するスキーマ変更ができなくなります。有効化されている従属マテリアライズド・ビューを持つテーブルのスキーマは変更できません。「マテリアライズド・ビューの削除」 73 ページを参照してください。

♦ **マテリアライズド・ビューを作成するには、次の手順に従います (SQL の場合)。**

1. DBA 権限または RESOURCE 権限のあるユーザと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。「CREATE MATERIALIZED VIEW 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
3. マテリアライズド・ビューにデータが含まれるように、初期化する必要があります。「マテリアライズド・ビューの初期化」 61 ページを参照してください。

例

次の文は、従業員に関する情報を格納するマテリアライズド・ビュー EmployeeConfid16 を作成し、初期化して、データを設定します。

```
CREATE MATERIALIZED VIEW EmployeeConfid16 AS
  SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary, ManagerID,
    Departments.DepartmentName, Departments.DepartmentHeadID
```



```
    FROM Employees, Departments
    WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid16;
```

警告
この例を実行し終わったら、作成したマテリアライズド・ビューを削除してください。そうしないと、他の例を試すときに、基本となるテーブル Employees および Departments に対するスキーマ変更ができなくなります。有効化されている従属マテリアライズド・ビューを持つテーブルのスキーマは変更できません。「マテリアライズド・ビューの削除」 73 ページを参照してください。

マテリアライズド・ビューの初期化

データベース・サーバが利用できるように、マテリアライズド・ビューを初期化する必要があります。初期化するには、ビューをリフレッシュします。リフレッシュに失敗した場合は、マテリアライズド・ビューは未初期化状態に戻ります。

マテリアライズド・ビューを作成、初期化、またはリフレッシュする前に、マテリアライズド・ビューの制限をすべて満たしていることを確認してください。「マテリアライズド・ビューの制限」 57 ページを参照してください。

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マテリアライズド・ビューを 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。

♦ **マテリアライズド・ビューを初期化するには、次の手順に従います (Sybase Central の場合)。**

1. DBA 権限のあるユーザとして、またはマテリアライズド・ビューの INSERT パーミッションを持つユーザとして、データベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. マテリアライズド・ビューを右クリックして、**[データの再表示]** を選択します。
4. 独立性レベルを選択して、**[OK]**をクリックします。

警告
この例を実行し終わったら、作成したマテリアライズド・ビューを削除してください。そうしないと、他の例を試すときに、基本となるテーブル Employees および Departments に対するスキーマ変更ができなくなります。有効化されている従属マテリアライズド・ビューを持つテーブルのスキーマは変更できません。「マテリアライズド・ビューの削除」 73 ページを参照してください。

♦ **マテリアライズド・ビューを初期化するには、次の手順に従います (SQL の場合)。**

1. DBA 権限のあるユーザとして、またはマテリアライズド・ビューの INSERT パーミッションを持つユーザとして、データベースに接続します。
2. REFRESH MATERIALIZED VIEW 文を実行します。

例

次の文は、マテリアライズド・ビュー EmployeeConfid6 を作成し、初期化します。

```
CREATE MATERIALIZED VIEW EmployeeConfid6 AS
  SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary, ManagerID,
    Departments.DepartmentName, Departments.DepartmentHeadID
  FROM Employees, Departments
  WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid6;
```

警告
この例を実行し終わったら、作成したマテリアライズド・ビューを削除してください。そうしないと、他の例を試すときに、基本となるテーブル Employees および Departments に対するスキーマ変更ができなくなります。有効化されている従属マテリアライズド・ビューを持つテーブルのスキーマは変更できません。「マテリアライズド・ビューの削除」 73 ページ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』
- 「マテリアライズド・ビューの有効化と無効化」 67 ページ

手動ビューのリフレッシュ

基本となるベース・テーブルで変更が発生すると、手動ビューは古くなります。手動ビューをリフレッシュするということは、データベース・サーバがそのビューのクエリ定義を再実行し、ビューのデータをそのクエリの新しい結果セットで置き換えることを意味します。リフレッシュを行うと、ビューのデータが基本となるデータと一致します。手動ビューのデータの古さについて許容可能な程度を検討し、リフレッシュ方式を考案してください。リフレッシュ操作中はビューでクエリを処理できないため、リフレッシュ方式でリフレッシュを完了するのにかかる時間を考慮する必要があります。

また、イベントを使用してビューをリフレッシュする方式を設定することもできます。たとえば、イベントを作成して一定の間隔でリフレッシュすることができます。

即時ビューでは、トランケートされた後などの未初期化状態 (データがない) 場合を除き、リフレッシュする必要はありません。

sa_refresh_materialized_views システム・プロシージャを使用してビューをリフレッシュ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。

materialized_view_optimization データベース・オプションを使用して、古さのしきい値を設定できます。オプティマイザは、クエリの処理時にこのしきい値を超えたマテリアライズド・ビューを使用しません。「オプティマイザでのマテリアライズド・ビューに対する古さのしきい値の設定」71 ページを参照してください。

REFRESH MATERIALIZED VIEW 文を使用する場合、WITH ISOLATION LEVEL 句を使用して接続の独立性レベルを上書きできます。マテリアライズド・ビューをリフレッシュするときの同時実行性を制御する方法については、「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ - SQL リファレンス』の WITH 句を参照してください。

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

♦ **手動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 権限のあるユーザとして、またはマテリアライズド・ビューの INSERT パーミッションを持つユーザとして、データベースに接続します。基本となるテーブルに対する SELECT パーミッションも必要です。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. マテリアライズド・ビューを右クリックして、**[データの再表示]** を選択します。
4. 独立性レベルを選択して、**[OK]**をクリックします。

♦ **手動ビュー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 権限のあるユーザとして、またはマテリアライズド・ビューの INSERT パーミッションを持つユーザとして、データベースに接続します。基本となるテーブルに対する SELECT パーミッションも必要です。
2. REFRESH MATERIALIZED VIEW 文を実行します。

例

次の文は、マテリアライズド・ビュー EmployeeConfid33 を作成し、リフレッシュします。

```
CREATE MATERIALIZED VIEW EmployeeConfid33 AS
  SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary, ManagerID,
    Departments.DepartmentName, Departments.DepartmentHeadID
  FROM Employees, Departments
  WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid33;
```

警告
この例を実行し終わったら、作成したマテリアライズド・ビューを削除してください。そうしないと、他の例を試すときに、基本となるテーブル Employees および Departments に対するスキーマ変更ができなくなります。有効化されている従属マテリアライズド・ビューを持つテーブルのスキーマは変更できません。「マテリアライズド・ビューの削除」 73 ページを参照してください。

参照

- 「手動ビューから即時ビューへの変更」 64 ページ
- 「スケジュールとイベントの使用によるタスクの自動化」 『SQL Anywhere サーバ - データベース管理』
- 「materialized_view_optimization オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』

手動ビューから即時ビューへの変更

マテリアライズド・ビューを作成すると、再表示タイプは手動になります。ただし、即時タイプに変更できます。手動から即時に変更するには、ビューが未初期化状態である (データを含まない) 必要があります。ビューが作成された直後でまだリフレッシュされていない場合は、未初期化状態になっています。ビューにデータがある場合は、そのデータをトランケートする必要があります。また、ビューにはユニーク・インデックスも必要で、即時ビューに必要な制限に従う必要があります。「即時ビューの追加の制限」 58 ページを参照してください。

即時ビューは、再表示タイプを変更するだけで、いつでも手動に変換できます。

次の手順では、手動ビューを即時ビューに変更する方法について説明します。これらの手順のいずれかを実行する前に、手動ビューにユニーク・インデックスがあり、未初期化状態であることを確認してください。そして、必要に応じて sa_materialized_view_can_be_immediate システム・プロシージャを使用して、再表示タイプを即時にするための条件を確認します。「sa_materialized_view_can_be_immediate システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

♦ **手動ビューを即時ビューに変更するには、次の手順に従います (Sybase Central の場合)。**

1. DBA 権限のあるユーザとして、またはそのビューとビューが参照するすべての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、**[ビュー]** をダブルクリックします。
3. マテリアライズド・ビューを右クリックして、**[プロパティ]** を選択します。
4. **[再表示タイプ]** フィールドで、**[即時]** を選択します。
5. **[OK]** をクリックします。

♦ **手動ビューを即時ビューに変更するには、次の手順に従います (SQL の場合)。**

1. DBA 権限のあるユーザとして、またはそのビューとビューが参照するすべてのテーブルの所有者として、データベースに接続します。
2. 再表示タイプを即時に変更するには、ALTER MATERIALIZED VIEW ... IMMEDIATE REFRESH 文を実行します。

次の手順では、即時ビューを手動ビューに変更する方法について説明します。

♦ **即時ビューを手動ビューに変更するには、次の手順に従います (Sybase Central の場合)。**

1. ビューの所有者として、または 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. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. マテリアライズド・ビューを右クリックして、**[プロパティ]** を選択します。
4. **[再表示タイプ]** フィールドで、**[手動]** を選択します。
5. **[OK]** をクリックします。

♦ **即時ビューを手動ビューに変更するには、次の手順に従います (SQL の場合)。**

1. ビューの所有者として、または 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. 再表示タイプを手動に変更するには、ALTER MATERIALIZED VIEW ... MANUAL REFRESH 文を実行します。

例

次の例は、マテリアライズド・ビューを作成し、初期化します。即時ビューにはユニーク・インデックスが必要なため、次にユニーク・インデックスを追加します。再表示タイプを変更する際にビューにデータがあってはいけないため、ビューはトランケートされます。最後に、再表示タイプが変更されます。

```
CREATE MATERIALIZED VIEW EmployeeConfid44 AS
  SELECT EmployeeID, Employees.DepartmentID,
       SocialSecurityNumber, Salary, ManagerID,
       Departments.DepartmentName, Departments.DepartmentHeadID
    FROM Employees, Departments
    WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid44;
CREATE UNIQUE INDEX EmployeeIDIdx
  ON EmployeeConfid44 ( EmployeeID );
TRUNCATE MATERIALIZED VIEW EmployeeConfid44;
ALTER MATERIALIZED VIEW EmployeeConfid44
  IMMEDIATE REFRESH;
```

次の文は、再表示タイプを手動に戻します。

```
ALTER MATERIALIZED VIEW EmployeeConfid44
  MANUAL REFRESH;
```

警告
この例を実行し終わったら、作成したマテリアライズド・ビューを削除してください。そうしないと、他の例を試すときに、基本となるテーブル Employees および Departments に対するスキーマ変更ができなくなります。有効化されている従属マテリアライズド・ビューを持つテーブルのスキーマは変更できません。「マテリアライズド・ビューの削除」 73 ページを参照してください。

参照

- 「ALTER MATERIALIZED VIEW 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「マテリアライズド・ビューのステータスとプロパティ」 54 ページ
- 「インデックスの作成」 78 ページ
- 「マテリアライズド・ビューの初期化」 61 ページ

マテリアライズド・ビューの暗号化と復号化

セキュリティを高めるために、マテリアライズド・ビューを暗号化できます。たとえば、基本となるテーブルで暗号化されていたデータがマテリアライズド・ビューに含まれる場合、そのマテリアライズド・ビューも暗号化する状況も考えられます。マテリアライズド・ビューを暗号化するには、データベースでテーブルの暗号化をあらかじめ有効にしておく必要があります。データベースの作成時に指定した暗号化アルゴリズムとキーを使用して、マテリアライズド・ビューを暗号化します。テーブル暗号化が有効であるかどうかなど、暗号化設定がデータベースで有効であることを確認するには、次のように DB_PROPERTY 関数を使用して Encryption データベース・プロパティの値を取得します。

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

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

♦ マテリアライズド・ビューを暗号化するには、次の手順に従います (Sybase Central の場合)。

1. ビューの所有者として、または 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. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. マテリアライズド・ビューを右クリックして、**[プロパティ]** を選択します。
4. **[その他]** タブをクリックします。
5. **[マテリアライズド・ビューのデータは暗号化済み]** チェック・ボックスをオンにします。
6. **[OK]** をクリックします。

♦ マテリアライズド・ビューを暗号化するには、次の手順に従います (SQL の場合)。

1. DBA 権限のあるユーザとして、またはマテリアライズド・ビューの所有者として、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。

例

次の文は、マテリアライズド・ビュー EmployeeConfid44 を作成し、初期化して、暗号化します。この文が動作するためには、暗号化されたテーブルを許可するようにデータベースが設定されている必要があります。

```
CREATE MATERIALIZED VIEW EmployeeConfid44 AS
  SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary, ManagerID,
```



```
    Departments.DepartmentName, Departments.DepartmentHeadID
  FROM Employees, Departments
  WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid44;
ALTER MATERIALIZED VIEW GROUPO.EmployeeConfid44 ENCRYPTED;
```

警告
この例を実行し終わったら、作成したマテリアライズド・ビューを削除してください。そうしないと、他の例を試すときに、基本となるテーブル Employees および Departments に対するスキーマ変更ができなくなります。有効化されている従属マテリアライズド・ビューを持つテーブルのスキーマは変更できません。「マテリアライズド・ビューの削除」 73 ページを参照してください。

♦ **マテリアライズド・ビューを復号化するには、次の手順に従います (Sybase Central の場合)。**

1. ビューの所有者として、または 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. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. マテリアライズド・ビューを右クリックして、**[プロパティ]** を選択します。
4. **[その他]** タブをクリックします。
5. **[マテリアライズド・ビューのデータは暗号化済み]** チェック・ボックスをオフにします。
6. **[OK]** をクリックします。

♦ **マテリアライズド・ビューを復号化するには、次の手順に従います (SQL の場合)。**

1. DBA 権限のあるユーザとして、またはマテリアライズド・ビューの所有者として、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、マテリアライズド・ビュー EmployeeConfid44 を復号化します。

```
ALTER MATERIALIZED VIEW GROUPO.EmployeeConfid44 NOT ENCRYPTED;
```

参照

- 「データベース内のテーブル暗号化の有効化」 『SQL Anywhere サーバ - データベース管理』
- 「ALTER MATERIALIZED VIEW 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「DB_PROPERTY 関数 [システム]」 『SQL Anywhere サーバ - SQL リファレンス』

マテリアライズド・ビューの有効化と無効化

マテリアライズド・ビューを有効または無効にすることで、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、無効になったマテリアライズド・ビューは、最適化時にオプティマイザによって検討されません。クエリが無効なマテリアライズド・ビューを明示的に参照している場合、そのクエリは失敗し、エラーが返されます。マテリアライズド・ビュー

を無効にすると、データベース・サーバはそのビューのデータを削除しますが、定義はデータベース内に保持します。マテリアライズド・ビューをもう一度有効にすると、初期化されていない状態になるため、データを設定するためにはそのビューをリフレッシュする必要があります。

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

マテリアライズド・ビューを無効にすると、データとインデックスは削除されます。ビューが即時ビューの場合は、手動ビューに変わります。そのため、再度有効にする場合は、リフレッシュしてインデックスを再構築し、必要に応じてもう一度即時ビューに変更する必要があります。

パーミッションは無効化されたオブジェクトに対して付与できます。無効化されたオブジェクトに対するパーミッションはデータベースに保存され、オブジェクトが有効になると使用できるようになります。

♦ **マテリアライズド・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 権限のあるユーザとして、またはマテリアライズド・ビューの所有者として、データベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. マテリアライズド・ビューを右クリックして、**[無効にする]** を選択します。

♦ **マテリアライズド・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 権限のあるユーザとして、またはマテリアライズド・ビューの所有者として、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

次の例は、マテリアライズド・ビュー EmployeeConfid55 を作成し、初期化して、無効化します。無効化されると、マテリアライズド・ビューのデータは削除されますが、マテリアライズド・ビューの定義はデータベース内に残ります。データベース・サーバはマテリアライズド・ビュー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。

```
CREATE MATERIALIZED VIEW EmployeeConfid55 AS
  SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary, ManagerID,
    Departments.DepartmentName, Departments.DepartmentHeadID
  FROM Employees, Departments
  WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid55;
ALTER MATERIALIZED VIEW EmployeeConfid55 DISABLE;
```


♦ **マテリアライズド・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 権限のあるユーザとして、またはマテリアライズド・ビューの所有者として、データベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. マテリアライズド・ビューを右クリックして、**[再コンパイルして有効にする]** を選択します。
4. (省略可) ビューを右クリックして **[データの再表示]** を選択することで、ビューを初期化して、データを設定します。

♦ **マテリアライズド・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 権限のあるユーザとして、またはマテリアライズド・ビューの所有者として、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. (省略可) REFRESH MATERIALIZED VIEW を実行して、ビューを初期化し、データを設定します。

例

次の 2 つの文はそれぞれ、マテリアライズド・ビュー EmployeeConfid55 をもう一度有効にし、データを設定します。

```
ALTER MATERIALIZED VIEW EmployeeConfid55 ENABLE;
REFRESH MATERIALIZED VIEW EmployeeConfid55;
```

警告
この例を実行し終わったら、作成したマテリアライズド・ビューを削除してください。そうしないと、他の例を試すときに、基本となるテーブル Employees および Departments に対するスキーマ変更ができなくなります。有効化されている従属マテリアライズド・ビューを持つテーブルのスキーマは変更できません。「マテリアライズド・ビューの削除」 73 ページを参照してください。

参照

- 「手動ビューから即時ビューへの変更」 64 ページ
- 「REFRESH MATERIALIZED VIEW 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』
- 「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」 38 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マテリアライズド・ビュー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マテリアライズド・ビューのリストを管理します。マテリアライズド・ビュー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、またはマテリアライズド・ビュー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。最適化処理でマテリアライズド・ビューが候補と見なされる条件については、「マテリアライズド・ビューによるパフォーマンスの向上」 603 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マテリアライズド・ビューを使用できます。ただし、マテリアライズド・ビュー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マテリアライズド・ビューの使用を無効にできます。

オプティマイザが使用する場合にマテリアライズド・ビューが有効であるか、無効であるかを判断するには、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。

♦ 最適化でマテリアライズド・ビュー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 権限のあるユーザとしてデータベースに接続します。
2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. マテリアライズド・ビューを右クリックして、**[プロパティ]** を選択します。
4. **[一般]** タブをクリックして、**[最適化に使用]** をオンにします。
5. **[OK]** をクリックします。

♦ 最適化でマテリアライズド・ビュー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 権限のあるユーザとして、またはマテリアライズド・ビューの所有者として、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。

例

次の文は、最適化でビュー EmployeeConfid77 の使用を有効にします。

```
ALTER MATERIALIZED VIEW GROUPO.EmployeeConfid77 ENABLE USE IN OPTIMIZATION;
```

♦ 最適化でマテリアライズド・ビュー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 権限のあるユーザとしてデータベースに接続します。

2. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. マテリアライズド・ビューを右クリックして、**[プロパティ]** を選択します。
4. **[一般]** タブをクリックして、**[最適化に使用]** をオフにします。
5. **[OK]** をクリックします。

♦ 最適化でマテリアライズド・ビュー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 権限のあるユーザとして、またはマテリアライズド・ビューの所有者として、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、マテリアライズド・ビュー EmployeeConfid77 を作成し、リフレッシュして、最適化での使用を無効化します。

```
CREATE MATERIALIZED VIEW EmployeeConfid77 AS
  SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary, ManagerID,
    Departments.DepartmentName, Departments.DepartmentHeadID
  FROM Employees, Departments
  WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid77;
ALTER MATERIALIZED VIEW EmployeeConfid77 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マテリアライズド・ビューに対する古さのしきい値の設定

マテリアライズド・ビュー内のデータは、そのビューが参照するテーブルのデータが変更されることによって古くなります。materialized_view_optimization データベース・オプションを使用すると、古さのしきい値を設定できます。オプティマイザは、クエリの処理時にこのしきい値を超えたマテリアライズド・ビューを使用しなくなります。materialized_view_optimization データベース・オプションは、マテリアライズド・ビューがリフレッシュされる頻度に影響しません。

クエリがマテリアライズド・ビューを明示的に参照している場合、そのビューのデータの古さに関係なく、ビューがクエリの処理に使用されます。さらに、materialized_view_optimization データベース・オプションの設定を上書きし、マテリアライズド・ビューを強制的に使用する場合に、SELECT 文の OPTION 句を使用できます。「SELECT 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

マテリアライズド・ビューがオプティマイザによって検討されない場合、古さに原因がある可能性があります。ビューをリフレッシュするイベントまたはトリガ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マテリアライズド・ビューがリフレッシュされると、オプティマイザはそのマテリアライズド・ビューを使用しません。

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。

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。

マテリアライズド・ビューがオプティマイザによって検討され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」 642 ページと「クエリのパフォーマンスのモニタ」 256 ページを参照してください。

マテリアライズド・ビューを隠す

マテリアライズド・ビューの定義をユーザから隠すことができます。マテリアライズド・ビューを隠すには、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。マテリアライズド・ビューを隠すと、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。

マテリアライズド・ビュー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。

♦ **マテリアライズド・ビューを隠すには、次の手順に従います (SQL の場合)。**

1. DBA 権限のあるユーザとして、またはマテリアライズド・ビューの所有者として、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、マテリアライズド・ビュー EmployeeConfid3 を作成し、リフレッシュして、ビュー定義を難読化します。

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

警告
この例を実行し終わったら、作成したマテリアライズド・ビューを削除してください。そうしないと、他の例を試すときに、基本となるテーブル Employees および Departments に対するスキーマ変更ができなくなります。有効化されている従属マテリアライズド・ビューを持つテーブルのスキーマは変更できません。「マテリアライズド・ビューの削除」 73 ページを参照してください。

参照

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

マテリアライズド・ビューの削除

不要になったマテリアライズド・ビューは削除できます。

マテリアライズド・ビューの削除とビューの依存性

マテリアライズド・ビューを削除する前に、すべての従属ビューを削除または無効にする必要があります。マテリアライズド・ビューに従属ビューが存在するかどうかを判断するには、sa_dependent_views システム・プロシージャを使用します。「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」 38 ページも参照してください。

♦ **マテリアライズド・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 権限のあるユーザとして、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. 左ウィンドウ枠で、**[ビュー]** をダブルクリックします。
3. マテリアライズド・ビューを右クリックして、**[削除]** を選択します。
4. **[はい]** をクリックします。

♦ **マテリアライズド・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 権限のあるユーザとして、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. DROP MATERIALIZED VIEW 文を実行します。

例

次の文は、マテリアライズド・ビュー EmployeeConfid4 を作成し、初期化 (データを設定)して、削除します。

```
CREATE MATERIALIZED VIEW EmployeeConfid4 AS
  SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary, ManagerID,
    Departments.DepartmentName, Departments.DepartmentHeadID
  FROM Employees, Departments
  WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid4;
DROP MATERIALIZED VIEW EmployeeConfid4;
```

参照

- 「DROP MATERIALIZED VIEW 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「sa_dependent_views システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』
- 「ビューの依存性」 38 ページ

インデックスの操作

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。

どのようなときにインデックスを使うか

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。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。「クエリに対するインデックス・コンサルタントの推奨内容の確認」 199 ページを参照してください。

作成するインデックスの決定

データベースに適切なインデックス・セットを選択することは、パフォーマンスを最適化する上で重要です。適切なセットを識別することは、労力を要する作業でもあります。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。「クエリに対するインデックス・コンサルタントの推奨内容の確認」 199 ページを参照してください。

頻繁に検索するカラムのインデックス

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。

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。インデックスによるパフォーマンスの向上の詳細については、「インデックスの使用」 268 ページを参照してください。

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

インデックス・ヒント

クエリの作成時にインテックス・ヒントを指定できます。インデックス・ヒントは、特定のインデックスを強制的に使用させることにより、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します。通常、インデックス・ヒントはオプティマイザのプランの選択を評価する場合のみに使用され、上級ユーザやデータベース管理者のみが使用します。インデックス・ヒントが不適切なアプリケーションでは、クエリのパフォーマンスが低下する場合があります。

インデックス・ヒントは FROM 句のサブ句を使用して指定します。たとえば、INDEX 句を使用すると 4 つまでのインデックスを指定できます。オプティマイザは、指定されたすべてのインデックスを使用できる必要があります。使用できない場合はエラーが返されます。

クエリでインデックスの使用を無効にするには NO INDEX を指定します。代わりにテーブルの逐次スキャンが強制的に実行されます。ただし、逐次スキャンはコストが非常に高く、実行に時間がかかります。この句は、オプティマイザのインデックスの選択を評価する場合に比較の目的のみで使用してください。

デフォルトでは、インデックス・データのみを使用してクエリの条件を満たすことができる場合 (つまり、テーブル内のローにアクセスする必要がない場合)、データベース・サーバはインデックス専用取得を実行します。ただし、インデックス専用取得でインデックスを使用できない (たとえば、インデックスが変更されたか削除された) というエラーがイベントで返されるように、INDEX ONLY ON を指定することもできます。

FROM 句で指定できるインデックス・ヒント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。

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、特定範囲のキーの値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、大規模なインデックス・スキャンをさらに改善することができます。クラスタード・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りません。テーブルのローがソートされた順序で返されるようにするには、インデックスを使用してローにアクセスするアクセス・プランか、物理ソートを実行するアクセス・プランが必要です。

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

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

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

- クエリの応答に必要なテーブル・ページの多くが、メモリ内にまだ存在しない。テーブル・ページがすでにメモリ内に存在する場合、サーバはこれらのページを読み込む必要がないため、クラスタリングは影響しません。
- 非自明な数のローが返されると予想されるインデックス検索を実行し、クエリが応答できる。たとえば、通常、クラスタリングは単純なプライマリ・キーの検索には影響しません。
- インデックス専用取得の実行とは対照的に、データベース・サーバは実際にテーブル・ページを読み込む必要がある。

SQL 文を使用したインデックスのクラスタ化

インデックスのクラスタ化プロパティは、SQL 文を使用していつでも追加または削除できます。あらゆるプライマリ・キー・インデックス、外部キー・インデックス、一意性制約インデック

ス、セカンダリ・インデックスは、CLUSTERED プロパティを使用して宣言できます。ただし、宣言できるクラスタード・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。

Sybase Central でのクラスタード・インデックスの作成

インデックス作成ウィザードを使用して、Sybase Central でクラスタード・インデックスを作成することもできます。メッセージが表示されたら、[クラスタード・インデックスを作成します] を選択します。「インデックスの作成」 78 ページを参照してください。

クラスタード・インデックスと一致させるためのローの並べ替え

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 を使用します。「REORGANIZE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

♦ **新しいインデックスを作成するには、次の手順に従います (Sybase Central の場合)。**

1. DBA 権限のあるユーザとしてデータベースに接続します。
2. 左ウィンドウ枠で **[テーブル]** をダブルクリックし、インデックスを作成するテーブルを選択します。
3. 右ウィンドウ枠で、**[インデックス]** タブをクリックします。
4. 左ウィンドウ枠でテーブルを右クリックし、**[新規]** - **[インデックス]** を選択します。
5. **インデックス作成ウィザード**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]** にも表示されます。

♦ **新しいインデックスを作成するには、次の手順に従います (SQL の場合)。**

1. DBA 権限のあるユーザとして、またはインデックスが作成され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。「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);
```

「CREATE INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』と「データベース・パフォーマンスの改善」 189 ページを参照してください。

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 権限のあるユーザとして、またはインデックスが作成され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]** をダブルクリックします。
3. インデックスを右クリックして、**[検証]** を選択します。
4. **[OK]** をクリックします。

♦ **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 権限のあるユーザとして、またはインデックスが作成され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 (dbvalid ユーティリティの場合)。**

● -i オプションを指定して、dbvalid コマンドを実行します。

例 1

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;
```

「VALIDATE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

例 2

インデックス EmployeeNames を検証します。-I オプションは、指定されたオブジェクト名がインデックスであることを指定します。

```
dbvalid -I EmployeeNames
```

「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

テーブルに対する多くの挿入操作や削除操作によって、インデックスが断片化したり無駄が多くなったりするために、インデックスを再構築する必要がある場合があります。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。論理インデックスで再構築を実行する必要はありません。「論理インデックスを使用したインデックスの共有」 674 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、Sybase Central を使用するか、ALTER INDEX ... REBUILD 文を実行します。また、REORGANIZE TABLE 文を使用して、テーブルの断片化を削除する作業の一部としてインデックスを再構築することもできます。この項では、Sybase Central と ALTER INDEX ... REBUILD 文を使用してインデックスを再構築する方法について説明します。REORGANIZE TABLE 文の使用の詳細については、「REORGANIZE TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

♦ **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 権限のあるユーザとして、またはインデックスが作成され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]** をダブルクリックします。
3. インデックスを右クリックして、**[再構築]** を選択します。
4. **[OK]** をクリックします。

♦ **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 権限のあるユーザとして、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;
```

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

参照

インデックスの断片化およびスキューと、これらの削減方法の詳細については、「インデックスの断片化とスキューの削減」 262 ページを参照してください。

インデックスの断片化とスキューの検出方法の詳細については、「アプリケーション・プロファイリング・ウィザード」 191 ページと「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』を参照してください。

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 権限のあるユーザとして、またはインデックスが作成され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]** をダブルクリックします。
3. インデックスを右クリックして、**[削除]** を選択します。
4. **[はい]** をクリックします。

♦ **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 権限のあるユーザとして、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;
```

「DROP INDEX 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、これらのテーブルのビューをブラウズして、それに含まれるデータを確認できます。

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

- **ISYSIDX システム・テーブル** インデックスを追跡するための中央テーブルです。ISYSIDX システム・テーブルの各ローは、データベース内の論理インデックス (PKEY、FKEY、一意性制約、セカンダリ・インデックス) を定義します。「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」 674 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」 674 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 つを示すように、SYSIDXCOL システム・ビューの各ローは、SYSIDX システム・ビューで記述されているインデックスのカラム 1 つを示します。「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。

データ整合性の確保

目次

データに整合性があるということは、データが有効、つまり適切であり正確で、データベースの関係構造が保たれていることを意味します。参照整合性制約を使うと、データベースの関係構造を確保できます。また、この規則によって、各テーブル間でデータの一貫性を保つことができます。データベースに整合性制約を構築すると、データの一貫性を確実に保つことができます。

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」 873 ページを参照してください。

データが有効でなくなる状況

適切な検査が行われないと、データベース内のデータが有効でなくなる可能性があります。それぞれのケースは、この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

- 組織のデータベースの Departments テーブルに、2 人の異なる従業員が同じ部署 (DepartmentID は 200) を新しく追加した。

失われた外部キー関係

- 300 という DepartmentID で識別される部署が閉鎖になったのに、1 人の従業員レコードだけが他の部署へ移されていない。

データベースへの整合性制約の構築

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。

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。

データベースの内容が変更される状況

クライアント・アプリケーションから SQL 文が送られてくると、データベース・テーブル内の情報に変更が生じます。データベースの情報を実際に変更する SQL 文は、それほど多くはありません。次の文で変更が行われます。

- UPDATE 文を使って、テーブルのローを更新する。
- DELETE 文を使って、テーブルのローを削除する。
- INSERT 文を使って、テーブルに新しいローを挿入する。

データの整合性を維持するためのツール

データ整合性を確保するため、デフォルト値、データ制約、データベースの参照構造を保つための制約を利用できます。

デフォルト値

各エントリのデータの信頼性を高めるために、カラムにデフォルト値を設定できます。次に例を示します。

- すべてのユーザ、またはクライアント・アプリケーションによるトランザクション日を記録するデフォルト値を、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」 92 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、Salary (給与) カラムに上限と下限を設定して入力エラーを防げます。
- 検査制約を使って、それぞれのカラム値の差を検査できる。たとえば、図書館データベースで、DateReturned (返却日) エントリが DateBorrowed (貸出日) エントリよりも必ず後になるように指定できます。
- 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」 873 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」 99 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」 107 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、「SQL Anywhere でのデータベースの作成」 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」 873 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。
- **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。

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、カラム・デフォルトの追加、修正、削除を行うには、**[カラム・プロパティ]** ウィンドウの **[値]** タブを使用します。

♦ **カラムの [プロパティ] ウィンドウを表示するには、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. 左ウィンドウ枠で、**[テーブル]** をダブルクリックします。
3. テーブルをクリックします。
4. **[カラム]** タブをクリックします。
5. カラムを右クリックして、**[プロパティ]** を選択します。

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」 107 ページを参照)。

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」 705 ページを参照してください。

参照

- 「オートインクリメント・カラムのあるテーブルの再ロード」 『SQL Anywhere 11 - 変更点とアップグレード』

GLOBAL AUTOINCREMENT デフォルト

GLOBAL AUTOINCREMENT デフォルトは、SQL Remote レプリケーション環境または Mobile Link 同期環境で複数のデータベースを使用するときのために用意されたものです。複数のデータベースにユニークなプライマリ・キーを保証します。

このオプションは AUTOINCREMENT と同じですが、ドメインはパーティションに分割されます。各分割には同じ数の値が含まれます。データベースの各コピーにユニークなグローバル・データベース ID 番号を割り当てます。SQL Anywhere では、データベースのデフォルト値は、そのデータベース番号でユニークに識別された分割からのみ設定されます。

この分割サイズには任意の正の整数を設定できますが、通常、分割サイズは、サイズの値がすべての分割で不足しないように選択されます。

カラムの型が BIGINT または UNSIGNED BIGINT である場合、デフォルトの分割サイズは $2^{32} = 4294967296$ です。それ以外の型のカラムの場合、デフォルトの分割サイズは $2^{16} = 65536$ です。特に、カラムの型が INT または BIGINT ではない場合は、これらのデフォルト値が適切ではないことがあるため、分割サイズを明示的に指定するのが最も賢明です。

このオプションを使用する場合、各データベース内のパブリック・オプション global_database_id は、ユニークな正の整数に設定します。この値は、データベースをユニークに識別し、デフォルト値の割り当て元の分割を示します。使用できる値の範囲は $np + 1$ 〜 $(n+ 1) p$ です。ここで、n はパブリック・オプション global_database_id の値を表し、p は分割サイズを表します。たとえば、分割サイズを 1000、global_database_id を 3 に設定すると、範囲は 3001 〜 4000 になります。

前の値が $(n + 1) p$ 未満であれば、このカラム内でこれまで使用した最大値より 1 大きい値が次のデフォルト値になります。カラムに値が含まれていない場合、最初のデフォルト値は $np + 1$ になります。デフォルトのカラム値は、現在の分割以外のカラムの値の影響を受けません。つまり、

np + 1 より小さいか *p*(*n* + 1) より大きい数には影響されません。Mobile Link 同期を介して別のデータベースからレプリケートされた場合に、このような値が存在する可能性があります。

public オプション global_database_id は、負の値に設定できないため、選択された値は常に正になります。ID 番号の最大値を制限するのは、カラムのデータ型と分割サイズだけです。

public オプション global_database_id がデフォルト値の 2147483647 に設定されると、NULL 値がカラムに挿入されます。NULL 値が許可されていない場合に、ローの挿入を試みるとエラーが発生します。たとえば、テーブルのプライマリ・キーにカラムが含まれている場合に、この状況が発生します。

デフォルトの NULL 値は、分割で値が不足したときにも生成されます。この場合には、別の分割からデフォルト値を選択できるように、データベースに global_database_id の新しい値を割り当ててください。カラムで NULL が許可されていない場合に NULL 値を挿入しようとすると、エラーが発生します。未使用の値が残り少ないことを検出し、このような状態を処理するには、GlobalAutoincrement タイプのイベントを作成します。「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」 107 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

- 「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・カラム」 『SQL Remote』
- 「CREATE TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「オートインクリメント・カラムのあるテーブルの再ロード」 『SQL Anywhere 11 - 変更点とアップグレード』

NEWID デフォルト

ユニバーサル・ユニーク識別子 (UUID) を使用して、テーブルのユニークなローを識別できます。UUID は、グローバル・ユニーク識別子 (GUID) とも呼ばれます。この値は、1 台のコンピュータで生成された値が、他のコンピュータで生成された値と一致しないように生成されます。したがって、これらの値は、レプリケーション環境と同期環境でキーとして使用できます。

プライマリ・キーとして使用する場合、GLOBAL AUTOINCREMENT 値と比べると、UUID 値にはいくつかのトレードオフがあります。次に例を示します。

- 各リモート・データベースにユニークなデータベース ID を割り当てる必要がないので、GLOBAL AUTOINCREMENT より UUID の方が簡単に設定できます。システムのデータベース数や個々のテーブルのロー数を考慮する必要もありません。抽出ユーティリティ (dbxtract) を使用してデータベース ID の割り当てを処理できます。GLOBAL AUTOINCREMENT では通常、BIGINT データ型を使用する場合はこの点を考慮する必要はありませんが、BIGINT より小さいデータ型を使用する場合は考慮する必要があります。
- UUID 値は GLOBAL AUTOINCREMENT に必要な値よりかなり大きいため、プライマリ・テーブルと外部テーブルの両方でより多くのテーブル領域が必要です。また、UUID を使用すると、これらのカラムのインデックスの効率も悪くなります。つまり、GLOBAL AUTOINCREMENT の方がパフォーマンスに優れています。
- UUID には暗黙的な順序付けがありません。たとえば、A と B が UUID 値で、A が B よりも大きい場合に、A と B が同じコンピュータ上で生成されたとしても、A が B の後で生成されたとはかぎりません。暗黙的な順序付けが必要な場合は、追加のカラムとインデックスが必要になります。

参照

- 「NEWID 関数 [その他]」 『SQL Anywhere サーバ - SQL リファレンス』
- 「UNIQUEIDENTIFIER データ型」 『SQL Anywhere サーバ - SQL リファレンス』

NULL デフォルト

NULL 値を許容するカラムに NULL デフォルトを指定しても、デフォルトを指定しなくても、まったく同じです。ローを挿入するクライアントが特に値を指定しなければ、そのローは自動的に NULL 値が設定されます。

NULL デフォルトは、カラムの入力を省略できる場合、または入力データが入手できないことがある場合に使います。

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。

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、データが正しいかどうかを確認するためのものであったり、企業のポリシーや内部規定などが反映される厳密なものであったりします。個々のカラムに対する検査条件は、カラム内の値を一定の範囲に収める必要がある場合に便利です。

検査条件が使用されると、その後はローの修正前に値が条件に対して評価されます。検査制約が設定された値を更新すると、その値の制約が検査されるだけでなく、残りのローの制約も検査されます。

ドメインに検査制約を付加することもできます。「ドメインのカラム検査制約の継承」 100 ページを参照してください。

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。

例 2

エントリに入力されるデータが、あらかじめ決められたいくつかの値のいずれかになるように設定できます。たとえば、City カラムに、その会社の事業所の所在地など、いくつかの許可された都市のいずれかしか入力できないようにするには、次のように制約を使用します。

```
ALTER TABLE Customers
ALTER City
CHECK ( City IN ( 'city_1', 'city_2', 'city_3' ) );
```

データベースの作成時に特に指定をしなかった場合、文字列の比較において大文字と小文字は区別されません。

例 3

日付や数値が一定の範囲内に収まるように設定できます。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。

参照

- 「ドメイン」 『SQL Anywhere サーバ - SQL リファレンス』
- 「カラムに対する検査制約の使い方」 99 ページ

制約の管理

Sybase Central では、カラム制約の追加、変更、削除を、テーブルまたは **[カラム・プロパティ]** ウィンドウの **[制約]** タブで行います。

♦ **制約を管理するには、次の手順に従います (Sybase Central の場合)。**

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. 左ウィンドウ枠で、**[テーブル]** をダブルクリックします。
3. 変更するテーブルをクリックします。
4. 右ウィンドウ枠で、**[制約]** タブをクリックし、既存の制約を変更するか、新しい制約を追加します。

一意性制約の管理

カラムの場合、一意性制約では、カラム値をユニークにする必要があることを指定します。テーブルの場合、一意性制約では、テーブル内のユニークなローを識別する 1 つ以上のカラムを指定します。テーブル内の異なるローが、指定されているすべてのカラムで同じ値を持つことはできません。1 つのテーブルには、複数の一意性制約を設定することができます。

♦ **一意性制約を管理するには、次の手順に従います (Sybase Central の場合)。**

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. 左ウィンドウ枠で、**[テーブル]** をダブルクリックします。
3. 変更するテーブルをクリックします。

4. 右ウィンドウ枠で、**[制約]** タブをクリックします。
5. **[制約]** テーブルを右クリックし、**[新規] - [一意性制約]** を選択します。
6. **一意性制約作成ウィザード**の指示に従います。

検査制約の変更と削除

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。「ALTER TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

テーブル上の既存の検査制約を変更するには、いくつかの方法があります。

- テーブルまたはカラムに新しい検査制約を追加できます。
- カラムの検査制約を NULL に設定すると削除できます。次に、Customers テーブルの Phone カラムから検査制約を削除する文の例を示します。

```
ALTER TABLE Customers
ALTER Phone CHECK NULL;
```

- 検査制約の追加と同じ方法で、カラムの検査制約を置換できます。次に、Customers テーブルの Phone カラムの検査制約を追加または置換する文の例を示します。

```
ALTER TABLE Customers
ALTER Phone
CHECK ( Phone LIKE '___-___-____' );
```

- テーブルに定義された検査制約を変更できます。
  - ALTER TABLE と ADD *table-constraint* 句を使って新しい検査制約を追加できます。
  - 制約名を定義済みの場合は、制約を個別に変更できます。
  - 制約名を定義していない場合は、ALTER TABLE DELETE CHECK を使って、既存のすべての検査制約 (カラム検査制約、ドメインから継承した検査制約など) を削除してから、新しい検査制約を追加できます。

    ALTER TABLE 文で DELETE CHECK 句を使用するには、次のように指定します。

```
ALTER TABLE table-name
DELETE CHECK;
```

Sybase Central では、テーブル検査制約とカラム検査制約の両方を追加、変更、削除できます。詳細については、「制約の管理」 101 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、テーブルに対してデータの挿入や、単に問い合わせを行っただけでも「**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。通常、値の許容範囲の制限には検査制約が使われます。さらに、ドメインではデフォルト値を設定できます。値は NULL であってもそうでなくてもかまいません。

独自のドメインを定義することには次のような利点があります。

- 不適切な値が入力された場合の一般的なエラーを防ぐ。ドメインに設定した制約で、値を一定の範囲またはフォーマットに保持させたいすべてのカラムと変数に、ある範囲内またはフォーマットの値だけを保持させることができます。たとえば、あるデータ型によって、データベースに入力されるクレジット・カード番号に確実に正しい桁数が入るようにできます。
- アプリケーションやデータベースの構造をわかりやすくする。
- 利便性。たとえば、テーブルの識別子をすべて正の整数にし、デフォルト値としてオートインクリメントにするとします。テーブルを新しく作成するたびにこうした制約を設定するよりも、新しいドメインを定義して識別子がそのドメイン型の値以外をとらないように設定する方が、作業が少なくて済みます。

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. 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. 左ウィンドウ枠で、**[ドメイン]** を右クリックし、**[新規]** - **[ドメイン]** を選択します。
3. **ドメイン作成ウィザード**の指示に従います。

♦ 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. 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. 左ウィンドウ枠で、**[テーブル]** をダブルクリックします。
3. テーブルをクリックします。
4. 右ウィンドウ枠で、**[カラム]** タブをクリックします。
5. **[データ・タイプ]** フィールドでカラムを選択し、省略記号 (ピリオド 3 つ) ボタンをクリックします。
6. **[データ・タイプ]** タブをクリックし、**[ドメイン]** を選択します。

7. **[ドメイン]** リストからドメインを選択します。
8. **[OK]** をクリックします。

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。「CREATE DOMAIN 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

SQL Anywhere には、定義済みドメインがいくつかあります。通貨データ・ドメインの MONEY などがその例です。

♦ **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

従業員名と住所をそれぞれ格納するカラムを含んだデータベースがあるとします。この場合、次のようなドメインを定義します。

```
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 Customers2 (
  ID identifier PRIMARY KEY,
```

```
    Name persons_name,
    Street street_address
);
```

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 権限のあるユーザ、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. 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. 左ウィンドウ枠で、**[ドメイン]** をダブルクリックします。
3. 右ウィンドウ枠で、ドメインを右クリックし、**[削除]** を選択します。
4. **[はい]** をクリックします。

♦ **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. 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. 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 DOMAIN 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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で (データベース・スキーマに定義されている) テーブル間の関係が維持されていることを保証できます。

エンティティ整合性の確保

ユーザがローを挿入または更新すると、データベース・サーバによってそのテーブルのプライマリ・キーの有効性が確実に保持されます。つまり、テーブル内の各ローがプライマリ・キーによりユニークに識別されます。

例 1

SQL Anywhere サンプル・データベースの Employees テーブルでは、employee ID をプライマリ・キーとして使用します。新しい従業員がこのテーブルに追加されると、データベース・サーバは新しい employee ID 値がユニークであり、NULL でないことを検査します。

例 2

SQL Anywhere サンプル・データベースの SalesOrderItems テーブルは、プライマリ・キーとして 2 つのカラムを使用します。

このテーブルは注文された製品の情報を格納します。ID カラムには注文番号が入っていますが、1 つの注文番号に対して複数の製品が注文される場合があるため、このカラムだけではプライマリ・キーになりません。一方、LineID カラムは製品に対応する行を識別します。ID カラムと LineID カラムの 2 つがセットになって、ある製品をユニークに指定できるので、この 2 つがプライマリ・キーになります。

クライアント・アプリケーションがエンティティ整合性に違反する場合

エンティティ整合性は、プライマリ・キーの値がユニークで、かつそこに NULL が含まれていないことが必要です。クライアント・アプリケーションが重複するプライマリ・キー値を追加または更新すると、エンティティ整合性が破られます。エンティティ整合性の違反が検出されると、新しい情報はデータベースに追加されず、クライアント・アプリケーションにエラーが返されます。

整合性の違反をどのようにしてユーザに通知し、どう処理させるかは、アプリケーション側の問題です。適切な処置といっても、ユーザに対して重複しない値をプライマリ・キーに指定するよう促すことしかできません。

プライマリ・キーによるエンティティ整合性の確保

各テーブルにプライマリ・キーが設定されれば、エンティティ整合性を確保するためにクライアント・アプリケーション開発者やデータベース管理者がそれ以上することはありません。

テーブルの所有者は、テーブルの作成時にプライマリ・キーを指定します。後日、テーブル構造を修正した場合は、プライマリ・キーも再定義します。

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」 25 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。

外部キー作成の詳細については、「プライマリ・キーの管理」 25 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。

INSERT 文の整合性検査

SQL Anywhere は、INSERT 文の実行時に整合性検査を実行します。たとえば、部署を新設しようとする場合に、すでに使用されている DepartmentID 値を指定すると仮定します。

```
INSERT
INTO Departments ( DepartmentID, DepartmentName, DepartmentHeadID )
VALUES ( 200, 'Eastern Sales', 902 );
```

テーブルのプライマリ・キーがユニークでなくなるため、INSERT は拒否されます。DepartmentID カラムはプライマリ・キーなので、重複する値は許可されません。

関係に違反する値の挿入

次の文は SalesOrders テーブルに新しいローを挿入しますが、Employees テーブル内には存在しない SalesRepresentative ID を誤って指定します。

```
INSERT
INTO SalesOrders ( ID, CustomerID, OrderDate, SalesRepresentative)
VALUES ( 2700, 186, '2000-10-19', 284 );
```

Employees テーブルと SalesOrders テーブルとの関係は、SalesOrders テーブルの SalesRepresentative カラムと Employees テーブルの EmployeeID カラムに基づいた 1 対多の関係です。プライマリ・テーブル (Employees) にレコードが入力されている場合にのみ、外部テーブル (SalesOrders) 内の対応するレコードを挿入できます。

外部キー

Employees テーブルのプライマリ・キーは従業員 ID 番号です。SalesRepresentative テーブル内の営業担当者 ID 番号は、Employees テーブルの外部キーです。つまり、SalesOrders テーブル内にある各営業担当者 ID 番号と、Employees テーブル内にある同じ従業員の従業員 ID 番号を一致させる必要があります。

ID 番号が 284 の営業担当者に対して注文を追加しようとすると、「**テーブル 'SalesOrders' の外部キー 'FK_SalesRepresentative_EmployeeID' に対するプライマリ・キーの値がありません。**」のようなエラー・メッセージが表示されます。

Employees テーブルには、その ID 番号を持つ従業員は存在しません。これによって、有効な営業担当者 ID のない注文が挿入されるのを防ぐことができます。

参照

- 「テーブル間の関係」 『SQL Anywhere 11 - 紹介』

DELETE 文または UPDATE 文の整合性検査

更新オペレーションまたは削除オペレーションを行う場合にも、外部キー・エラーが発生する可能性があります。たとえば、Departments テーブルから R&D 部を削除するとします。DepartmentID フィールドは Departments テーブルのプライマリ・キーなので、1 対多の関係の「1」の側を構成します (対応する外部キーは Employees テーブルの DepartmentID フィールドで、「多」の側を構成します)。1 対多の関係の「1」のレコードは、対応する「多」のレコードすべてが削除されるまで削除できません。

DELETE 文の参照整合性エラー

Departments テーブルの R&D 部 (DepartmentID は 100) を削除するとします。R&D 部を参照するその他のレコードがデータベース内にあることを示すエラーがレポートされて、削除オペレーションが実行されません。R&D 部を削除するには、次のように、その部署に所属しているすべての従業員を削除する必要があります。

```
DELETE
FROM Employees
WHERE DepartmentID = 100;
```

これで、R&D 部に所属するすべての従業員が削除されたため、R&D 部を削除することができます。

```
DELETE
FROM Departments
WHERE DepartmentID = 100;
```

次の ROLLBACK 文を入力して、データベースのこれらの変更をキャンセルしてください。

```
ROLLBACK;
```

UPDATE 文の参照整合性エラー

たとえば、Employees テーブルの DepartmentID フィールドを変更するとします。DepartmentID フィールドは Employees テーブルの外部キーなので、1 対多の関係の「多」の側を構成します (対応するプライマリ・キーは Departments テーブルの DepartmentID フィールドで、「1」の側を構成します)。1 対多の関係の「多」の側のレコードは、「1」の側のレコードに対応しないかぎり、つまり参照するプライマリ・キーがなければ変更できません。

たとえば次の UPDATE 文を実行すると、整合性エラーが発生します。

```
UPDATE Employees
SET DepartmentID = 600
WHERE DepartmentID = 100;
```

「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・キーに加えた変更に合わせて、外部キーが自動的に修正されるように定義することもできます。上の例では、Employees テーブルから Departments テーブルへの外部キーが ON DELETE CASCADE を使用して定義された場合、部署 ID を削除すると、Employees テーブルの対応するエントリが自動的に削除されます。

ここまでの例では、整合性のないデータベースが確定的なものとしてコミットされることはありません。変更することによってデータベースの整合性が失われる可能性がある場合、SQL Anywhere では代替の動作もサポートされています。「データ整合性の確保」 85 ページを参照してください。

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・ビューの各ローは、SYS.SYSCONSTRAINT システム・ビューにリストされている検査制約を定義します。「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・ビューの各ローは、キーに定義した一致タイプなど、外部キーに関して記述します。「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・ビューの各ローは、データベース内の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』を参照してください。

システム・ビュー	説明
SYS.SYSTRIGGER	SYS.SYSTRIGGER システム・ビューの各ローは、参照トリガ・アクション (ON DELETE CASCADE など) が設定された外部キー制約を自動的に作成するトリガなどの、データベース内のトリガ 1 つを示します。 referential_action カラムには、アクションがカスケード (C)、削除 (D)、NULL 設定 (N)、制限 (R) のいずれであるかを示すアルファベット 1 文字が格納されています。 event カラムには、各アクションを実行するイベントを示すアルファベット 1 文字が格納されます (A=挿入と削除、B=挿入と更新、C=更新、D=削除、E=削除と更新、I=挿入、U=更新、M=挿入、削除、更新)。 trigger_time カラムには、アクションの実行がイベントのトリガの後 (A) または前 (B) のどちらに行われるかが格納されます。 「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。

トランザクションと独立性レベルの使用

目次

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、「同時実行性の概要」 121 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。データを変更すると、変更がトランザクション・ログに記録されます。変更は、COMMIT コマンドを入力するまでは永続的なものにはなりません。

トランザクションは次のいずれかのイベントで開始します。

- データベースへの接続後、最初の文
- トランザクションの終了後、最初の文

トランザクションは次のいずれかのイベントで完了します。

- データベースの変更を確定する COMMIT 文
- トランザクションで行われたすべての変更を取り消す ROLLBACK 文
- オートコミットが実行される文。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。
- 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」 691 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
```


同時実行性の概要

同時実行性とは複数のトランザクションを同時に処理するためにデータベース・サーバで必要な機能です。データベース・サーバ内に特殊なメカニズムがない場合、同時に発生したトランザクションがお互いに干渉して、情報の一貫性が失われる可能性があります。たとえば、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できる必要があります。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。

同時実行性に関する知識の必要な方

同時実行性は、すべてのデータベース管理者と開発者に関係のある問題です。シングルユーザ・データベースで作業する場合でも、複数のアプリケーションからの要求や、単一のアプリケーションの複数の接続からの要求を処理する場合には、同時実行性を考慮してください。これらの複数のアプリケーションや接続は、ネットワーク上でマルチユーザが経験するのとまったく同様に、お互いに干渉し合います。

同時実行性に影響するトランザクション・サイズ

SQL 文をトランザクションにまとめる方法は、データの整合性とシステムのパフォーマンスに大きな影響を与えます。トランザクションが短かすぎて論理的にひとまとまりにならない場合、データベースの一貫性が失われる可能性があります。トランザクションが長すぎて複数の互いに関係のない操作が含まれていると、本来なら安全にデータベースにコミットできたはずの操作が、不要な ROLLBACK によって取り消される可能性が高くなります。

トランザクションが長い場合、他のトランザクションが同時処理することを防ぐため、同時実行性が低下します。

アプリケーションのタイプや環境要因によって、トランザクションの適切な長さを決定する要素は数多くあります。

SQL Anywhere データベース・サーバの同時実行の詳細については、「SQL Anywhere データベース・サーバ実行の概要」 『SQL Anywhere サーバ - データベース管理』を参照してください。

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。

セーブポイントはバルク・オペレーション・モードでは使用できません。セーブポイントを使用しても、オーバヘッドはほとんど増えません。

独立性レベルと一貫性

SQL Anywhere では、あるトランザクションの操作が、同時に実行される他のトランザクションの操作で認識される程度を制御できます。この制御には、「独立性レベル」というデータベース・オプションを使用します。

また、対応するテーブル・ヒントを使用して、クエリ内の個々のテーブルの独立性レベルを制御することもできます。「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出し	● 書き込みロックの有無にかかわらず、ローで読み込みが許可される。 ● 読み込みロックは適用されない。 ● 同時トランザクションがローを変更しないこと、またはローに対しての変更がロールバックされないことは保証されない。 ● テーブル・ヒント NOLOCK と READUNCOMMITTED に対応する。 ● ダーティ・リード、繰り返し不可能読み出し、幻ローを許可する。
1 - コミットされた読み出し	● 書き込みロックのないローでは読み込みのみ許可される。 ● 現在のローでの読み込みに対してのみ読み込みロックが取得されて保持されるが、カーソルがローから移動すると解放される。 ● トランザクション中にデータが変更されないという保証はない。 ● テーブル・ヒント READCOMMITTED に対応する。 ● ダーティ・リードを防ぐ。 ● 繰り返し不可能読み出しと幻ローを許可する。
2 - 繰り返し可能読み出し	● 書き込みロックのないローでは読み込みのみ許可される。 ● 結果セットの各ローとして取得された読み込みロックが読み込まれ、トランザクションが終了するまで保持される。 ● テーブル・ヒント REPEATABLEREAD に対応する。 ● ダーティ・リードと繰り返し不可能読み出しを防ぐ。 ● 幻ローを許可する。

独立性レベル	特性
3 - 直列化可能	● 書き込みロックのない結果内のローに対しては読み込みのみ許可される。 ● カーソルが開いているときに取得された読み込みロックは、トランザクションの終了時まで保持される。 ● テーブル・ヒント HOLDLOCK と SERIALIZABLE に対応する。 ● ダーティ・リード、繰り返し不可能読み出し、幻ローを防ぐ。
snapshot [1]	● 読み込みロックは適用されない。 ● すべてのローで読み込みが許可される。 ● コミットされたデータのデータベース・スナップショットは、トランザクションが最初のローの読み込みまたは更新を行った時点で作成される。
statement-snapshot[1]	● 読み込みロックは適用されない。 ● すべてのローで読み込みが許可される。 ● コミットされたデータのデータベース・スナップショットは、文が最初のローの読み込みを行った時点で作成される。
readonly-statement-snapshot[1]	● 読み込みロックは適用されない。 ● すべてのローで読み込みが許可される。 ● コミットされたデータのデータベース・スナップショットは、読み込み専用の文が最初のローの読み込みを行った時点で作成される。 ● 更新可能な文の updatable_statement_isolation オプションで指定された独立性レベル (0、1、2、または 3) を使用する。

[1] データベースで allow_snapshot_isolation が On に設定され、そのデータベースでスナップショット・アイソレーションが有効である必要があります。「スナップショット・アイソレーションの有効化」 128 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。

Mobile Link の独立性レベルの詳細については、「Mobile Link 独立性レベル」 『Mobile Link - サーバ管理』を参照してください。

ロックベースの独立性レベルは、一部またはすべての干渉を防ぎます。レベル 3 は最も高いレベルの独立性を提供します。2 以下のレベルでは、一貫性のレベルは低くなりますが、パフォーマンスは一般にレベル 3 より高くなります。デフォルトでは、レベルは 0 (コミットされない読み出し) に設定されています。

スナップショット・アイソレーションのレベルは、読み込みと書き込み間の干渉を防ぎます。書き込みは相互に干渉する可能性があります。一貫性のない動作が多少生じる可能性がありますが、競合のパフォーマンスは独立性レベルを 0 に設定した場合と同じです。ロー・バージョンを保存して使用する必要があるため、競合以外のパフォーマンスは低下します。

注意
どの独立性レベルにおいても、各トランザクションが完全に実行されるかまったく実行されないこと、および更新内容が失われないことが保証されます。

独立性レベルはトランザクション間にのみあります。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。

スナップショット・アイソレーション

複数のユーザが同じデータに対して同時に読み込みと書き込みを行うと、ブロックやデッドロックが発生することがあります。スナップショット・アイソレーションは、さまざまなバージョンのデータを管理することにより同時実行性と一貫性を向上させる目的で設計されています。トランザクションでスナップショット・アイソレーションを使用すると、データベース・サーバは読み込み要求の応答としてコミットされたバージョンのデータを返します。これは読み込みロックを取得せずに行われるため、データを書き込んでいるユーザとの干渉は発生しません。

「スナップショット」とは、データベースでコミットされた一連のデータです。スナップショット・アイソレーションを使用すると、トランザクション内のすべてのクエリで同じデータのセットが使用されます。データベースのテーブルでロックは取得されません。これにより、他のトランザクションはブロックせずにアクセスして修正できます。SQL Anywhere では、スナップショットを作成するときに制御できる 3 つのスナップショット・アイソレーションのレベルをサポートしています。

- **snapshot**　トランザクションが最初のローの読み込み、挿入、更新、または削除を行った時点から、コミットされたデータのスナップショットを使用します。
- **statement-snapshot**　文で最初のローが読み込まれた時点から、コミットされたデータのスナップショットを使用します。トランザクション内の各文で参照されるデータのスナップショットはそれぞれ異なる時点のものになります。
- **readonly-statement-snapshot**　読み込み専用の文についてのみ、最初のローが読み込まれた時点から、コミットされたデータのスナップショットを使用します。トランザクション内の各読み込み専用文で参照されるデータのスナップショットはそれぞれ異なる時点のものになります。挿入、更新、削除の文については、updatable_statement_isolation オプションに指定された独立性レベル (0 (デフォルト)、1、2、3 のいずれか) を使用します。

BEGIN SNAPSHOT 文を使用して、トランザクションのスナップショットの開始時に指定するオプションもあります。「BEGIN SNAPSHOT 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

スナップショット・アイソレーションは、主に次のような場合に便利です。

- **読み込みは多いが更新は少ないアプリケーション**　スナップショット・トランザクションは、データベースを修正する文の場合のみ、書き込みロックを取得します。トランザクションが主に読み込み操作を実行する場合、スナップショット・トランザクションは、他のユーザと干渉する可能性のある読み込みロックを取得しません。
- **他のユーザがデータにアクセスする必要があるにもかかわらず、時間がかかるトランザクションを実行するアプリケーション**　スナップショット・トランザクションは読み込みロックを取得

しないため、他のユーザはスナップショット・トランザクションの実行中にデータの読み込みや更新を実行できます。

- **データベースからの一貫したデータのセットを読み取る必要があるアプリケーション**　スナップショットはある時点からのコミット済みデータのセットを表示するため、トランザクションの実行中に他のユーザが変更を加えた場合でも、スナップショット・アイソレーションを使用すれば、そのトランザクション内では変更されない一貫したデータを確認できます。

スナップショット・アイソレーションは、すべてのユーザが共有するベース・テーブルとグローバル・テンポラリ・テーブルのみに影響します。その他のテーブルの種類では読み込み操作を行っても、古いバージョンのデータは表示されず、スナップショットは開始されません。別のテーブルの種類に対する更新によってスナップショットが開始されるのは、isolation_level オプションが snapshot に設定されていて、更新によりトランザクションが開始される場合だけです。

文またはトランザクションのスナップショットを使用する、WITH HOLD 句を使用して開かれたカーソルがある場合、次の文は実行できません。

- ALTER INDEX
- ALTER TABLE
- CREATE INDEX
- DROP INDEX
- REFRESH MATERIALIZED VIEW
- REORGANIZE TABLE
- CREATE TEXT INDEX
- REFRESH TEXT INDEX

WITH HOLD 句でカーソルを開くと、スナップショットの開始時にコミットされたすべてのローのスナップショットが表示されます。カーソルを開いたトランザクションの開始以降の、現在の接続で完了された変更もすべて表示されます。

高速トランケーションが実行されない場合にのみ、TRUNCATE TABLE を使用できます。これは、このような場合に個別の DELETE がトランザクションログに記録されるためです。「TRUNCATE 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リフレッシュされた場合、マテリアライズド・ビュー・マッチングではそのビューは使用されません。

スナップショット・アイソレーションのレベルは、すべてのプログラミング・インタフェースでサポートされています。スナップショット・アイソレーションのレベルは SET OPTION 文を使用して設定できます。スナップショット・アイソレーションの使用については、次の項を参照してください。

- 「isolation_level オプション [データベース] [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を確認できます。そのため、スナップショット・トランザクションの開始前にローの更新がコミットされなかったかロールバックされた場合、スナップショット・トランザクションは元のロー値にアクセスできる必要があります。これにより、スナップショット・アイソレーションを使用するトランザクションは、基本となるテーブルにロックを設定せずにデータを閲覧できます。

VersionStorePages データベース・プロパティは、バージョン・ストア用に現在使用されているテンポラリ・ファイル内のページ数を返します。この値を取得するには、次のクエリを実行します。

```
SELECT DB_PROPERTY ( 'VersionStorePages' );
```

古いロー・バージョンのエントリは、不要になると削除されます。BLOB の古いバージョンは、不要になるまで、テンポラリ・ファイルではなく元のテーブルに格納されています。古いロー・バージョンのインデックス・エントリは、不要になるまで元のインデックスに格納されています。

テンポラリ・ファイル内の空き領域のサイズは、sa_disk_free_space システム・プロシージャを使用すると取得できます。「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。

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 11 Demo;UID=DBA;PWD=sql;ConnectionName=Connection1"
   ```

2. 次のコマンドを実行して、SQL Anywhere サンプル・データベースへの Interactive SQL 接続 (Connection2) を作成します。

   ```
   dbisql -c "DSN=SQL Anywhere 11 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出し) 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	...

ID	Name	Description	Size	Color	Quantity	...
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例の詳細については、次の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たダーティ・リードの回避」 165 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」 171 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」 176 ページ

更新の競合とスナップショット・アイソレーション

スナップショット・アイソレーションを使用する場合、トランザクションがローの古いバージョンを認識し、更新または削除しようとすると、更新の競合が発生することがあります。このような状況では、競合が検出されるとサーバでエラーが発生します。コミットされた変更の場合、これは更新または削除が試みられた時になります。コミットされていない変更の場合、更新または削除はブロックされ、変更がコミットされるときにサーバがエラーを返します。

readonly-statement-snapshot を使用すると、更新可能な文は、スナップショット・アイソレーションではなく実行し、常に最新バージョンのデータベースを認識するため、更新の競合は発生しません。そのため、readonly-statement-snapshot 独立性レベルには、スナップショット・アイソレーションの多くの利点があり、元々別の独立性レベルで実行するように設計されたアプリケーションを大きく変更する必要はありません。readonly-statement-snapshot 独立性レベルを使用するときは、次のようになります。

- 読み込み専用文では、読み込みロックは取得されない。
- 読み込み専用文は、データベースのコミットされた状態を常に認識する。

典型的な矛盾のケース

トランザクションの同時実行中に発生する可能性のある典型的な矛盾には 3 つのタイプがあります。(他のタイプの矛盾も発生し得るため、この 3 つがすべてというわけではありません)。これら 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」 163 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」 167 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」 173 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出し	X	X	X
readonly-statement-snapshot	X[1]	X[2]	X[3]
1 - コミットされた読み出し		X	X
statement-snapshot		X[2]	X[3]
2 - 繰り返し可能読み出し			X
3 - 直列化可能			
snapshot			

[1] ダーティ・リードは、updatable_statement_isolation オプションで指定された独立性レベルがその発生を妨げない場合に、トランザクション内の更新可能な文で発生することがあります。

[2] 繰り返し不可能読み出しは、updatable_statement_isolation オプションで指定された独立性レベルがその発生を妨げない場合に、トランザクション内の文で発生することがあります。繰り返し不可能読み出しは、各文によって新しいスナップショットが開始する場合に発生することがあります。そのため、ある文が認識しない変更を別の文が認識する可能性があります。

[3] 幻ローは、updatable_statement_isolation オプションで指定された独立性レベルがその発生を妨げない場合に、トランザクション内の文で発生することがあります。幻ローは、各文によって新しいスナップショットが開始する場合に発生することがあります。そのため、ある文が認識しない変更を別の文が認識する可能性があります。

この表から、以下の 2 つのことがわかります。

- 各独立性レベルは、3 つの典型的な矛盾のケースのうち 1 つを防止します。
- 各レベルは、それ以下のレベルで防止される矛盾のケースも防止します。
- statement snapshot 独立性レベルでは、繰り返し不可能読み出しと幻ローは、トランザクション内で発生する可能性がありますが、トランザクションの単一文内では発生しません。

ODBC では、独立性レベルに異なる名前が付いています。これらの名前は、それぞれのレベルで防止される矛盾の名前に基づいて付けられています。「ValuePtr パラメータ」 135 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」 873 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」 128 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、権限、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。

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」 137 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
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。

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」 173 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');
```

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」 140 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、通常は同時実行性が低下します。低下の度合いは、同時に実行するトランザクションによって異なります。

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。

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。

トランザクションのデッドロックを排除するには、SQL Anywhere はデッドロックに関わっている接続を選択し、その接続でアクティブなトランザクションの変更をロールバックして、エラーを返します。SQL Anywhere は内部のヒューリスティックを使用してロールバックする接続を選択し、blocking_timeout によって決められた残りのブロックの待機時間が最も短い接続を優先します。すべての接続が永久に待機するように設定されている場合は、サーバによってデッドロックが検出された接続を、犠牲にする接続として選択します。

スレッドのデッドロックを排除するには、SQL Anywhere はブロックしている最後の接続を選択し、その接続でアクティブなトランザクションの変更をロールバックして、エラーを返します。

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。

Deadlock イベントを使用して、デッドロックの発生時にアクションを実行できます。イベント・ハンドラでは、sa_report_deadlocks プロシージャを使用して、デッドロックが発生するに至った状況に関する情報を取得できます。データベース・サーバからデッドロックの詳細を取り出すには、log_deadlocks オプションを使用して RememberLastStatement 機能を有効にします。

次の手順では、デッドロックが発生した場合にその情報を取得するために使用できるテーブルとシステム・イベントの設定方法を示します。アプリケーションで頻繁にデッドロックが発生する場合は、アプリケーション・プロファイリングを使用するとデッドロックの原因の究明に役立ちます。「チュートリアル：デッドロックの診断」 280 ページを参照してください。

♦ デッドロックの発生時にアクションを実行するには、次の手順に従います。

1. 次のように、sa_report_deadlocks システム・プロシージャが返すデータを格納するテーブルを作成します。

```
CREATE TABLE DeadlockDetails(
 deadlockId INT PRIMARY KEY DEFAULT AUTOINCREMENT,
 snapshotId BIGINT,
 snapshotAt TIMESTAMP,
 waiter INTEGER,
 who VARCHAR(128),
 what LONG VARCHAR,
 object_id UNSIGNED BIGINT,
 record_id BIGINT,
 owner INTEGER,
 is_victim BIT,
 rollback_operation_count UNSIGNED INTEGER );
```

2. デッドロックが発生したときに起動するイベントを作成します。

   このイベントにより、sa_report_deadlocks システム・プロシージャの結果がテーブルにコピーされ、管理者にデッドロックが通知されます。

```
CREATE EVENT DeadlockNotification
TYPE Deadlock
HANDLER
BEGIN
 INSERT INTO DeadlockDetails WITH AUTO NAME
 SELECT snapshotId, snapshotAt, waiter, who, what, object_id, record_id,
       owner, is_victim, rollback_operation_count
   FROM sa_report_deadlocks ();
 COMMIT;
 CALL xp_startmail ( mail_user ='George Smith',
                     mail_password ='mypwd' );
 CALL xp_sendmail( recipient='DBAdmin',
                   subject='Deadlock details added to the DeadlockDetails table.' );
 CALL xp_stopmail ( );
END;
```

3. log_deadlocks オプションを On に設定します。

```
SET OPTION PUBLIC.log_deadlocks = 'On';
```

4. 最後に実行された文のロギングを有効にします。

```
CALL sa_server_option( 'RememberLastStatement', 'YES' );
```

参照

- 「log_deadlocks オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』
- 「sa_report_deadlocks システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』
- 「sa_server_option システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』
- 「CREATE EVENT 文」 『SQL Anywhere サーバ - SQL リファレンス』

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。デッドロックに関する情報は、内部バッファに記録されています。

♦ **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。
2. log_deadlocks オプションをオンにします。
   a. **[オプション]** リストで **[log_deadlocks]** を選択します。
   b. **[値]** フィールドに **On** と入力します。
   c. **[恒久的な設定を行う]** をクリックします。
   d. **[閉じる]** をクリックします。

   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。

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」 157 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て、データベース・サーバはこれらのロックのいくつかまたはすべての種類を使用して必要な一貫性レベルを維持します。

ロック期間

ロックのクラスが異なると、保持される期間も異なることがあります。

- **位置**　特定のローにおける読み込みロックのような短期間のロックで、独立性レベル 1 でカーソルの安定性を実装するために使用される
- **トランザクション**　トランザクションの終了まで保持されるロー・ロック、テーブル・ロック、位置ロック
- **接続**　WITH HOLD カーソルの使用時に作成されるスキーマ・ロックのように、トランザクションが終了しても保持されるスキーマ・ロック

ロック情報の取得

データベースのロックの問題を診断するために、ロックされているローの内容を調べると役に立つ場合があります。データベースで現在保持されているロックを確認するには、sa_locks システム・プロシージャ、または Sybase Central の **[テーブル・ロック]** タブを使用します。どちらの方法でも、ロックを保持している接続、ロックの期間、ロックの種類などの必要な情報を得ることができます。

注意
データベースのロックは一時的なものであるため、Sybase Central で参照できるローや sa_locks システム・プロシージャによって返されるローは、クエリの完了時にはすでに存在しません。

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。左ウィンドウ枠のデータベースを選択し、右ウィンドウ枠の **[テーブル・ロック]** タブをクリックします。このタブでは、それぞれのロックに対する接続 ID、ユーザ ID、テーブル名、ロック・タイプ、ロック名が表示されます。

sa_locks システム・プロシージャを使用したロックの表示

sa_locks システム・プロシージャの結果セットには、ロックが参照するテーブルのローを識別できる row_identifier カラムが含まれます。ロックされたローに格納されている実際の値を判断するために、ジョイン述部でテーブルの rowID を使用して、sa_locks システム・プロシージャの結果を特定のテーブルにジョインできます。次に例を示します。

```
SELECT S.conn_id, S.user_id, S.lock_class, S.lock_type, E.*
  FROM sa_locks() S JOIN Employees E WITH( NOLOCK )
    ON RowId(E) = S.row_identifier
  WHERE S.table_name = 'Employees';
```

注意
NOLOCK テーブル・ヒントを指定する必要はない場合があります。ただしクエリが 0 以外の独立性レベルで発行された場合、ロックが解放されるまでこのクエリがブロックされることがあり、この確認方法の便利さは低下してしまいます。

参照

sa_locks システム・プロシージャの詳細については、「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・スキーマは共有 (読み込み) モードでロックされる。
- **排他ロック**　テーブル・スキーマは、単一接続の排他的使用のためにロックされる。

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。

共有スキーマ・ロックを保持する接続は、変更が他の接続と競合しない場合に、テーブル・データを変更できます。

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。

テーブル・スキーマの排他ロックを保持する接続のみがテーブル・データを変更できます。

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。

ロー・ロックには、読み込み (共有) ロック、書き込み (排他) ロック、意図的ロックという 3 つのクラスがあります。データベース・サーバは、トランザクションごとにこれらのロックを自動的に取得します。

読み込みロック

トランザクションがローを読み込むと、トランザクションのアイソレーション・レベルは読み込みロックが取得されているかどうかを判断します。一度読み込みロックのかかったローに対しては、他のどのトランザクションも書き込みロックを取得できません。読み込みロックが取得されると、ローの読み込み中は、別のトランザクションはそのローを修正または削除しません。任意のローに同時に取得できる読み込みロックの数に制限はありません。そのため、読み込みロックは、共有ロックまたは非排他ロックと呼ばれることもあります。

読み込みロックは、保持される期間が異なることがあります。独立性レベル 2 と 3 では、トランザクションが取得したどの読み込みロックも、トランザクションが COMMIT または ROLLBACK によって完了するまで保持されます。これらの読み込みロックは、長期間の読み込みロックと呼ばれます。

独立性レベル 1 で実行されるトランザクションの場合、データベース・サーバはカーソルが位置するローで短期間の読み込みロックを取得します。アプリケーションがカーソルをスクロールすると、カーソルが直前に位置していたローで短期間の読み込みロックは解放され、新しい短期間の読み込みロックがその次のローで取得されます。この技術は「カーソルの安定性」と呼ばれます。アプリケーションは現在のローで読み込みロックを保持するため、アプリケーションがそのローからカーソルを移動するまで、別のトランザクションがそのローに対して変更を加えることができません。カーソルが複数のテーブルを伴うクエリに対する場合は、複数のロックを取得できます。短期間の読み込みロックは、カーソル内の位置が要求 (通常は、アプリケーションによって発行される FETCH 文) 間で維持される必要がある場合だけ取得されます。たとえば SELECT COUNT(*) クエリの処理時は短期間の読み込みロックは取得されません。この文で開かれているカーソルが、ベース・テーブルの特定ローに位置することがないためです。この場合、データベース・サーバは、コミットされた読み出しのセマンティック、つまりこの文で処理されるローは他のトランザクションによってコミットされたことを保証すれば良いことになります。

独立性レベル 0 (コミットされない読み出し) で実行されるトランザクションの場合は、長期間または短期間の読み込みロックを取得しないため、他のトランザクションと競合しません (排他スキーマ・ロックの場合を除く)。ただし、独立性レベル 0 のトランザクションは、同時に実行している他のトランザクションによって加えられたコミットされていない変更を処理することがあります。コミットされていない変更を処理しないようにするには、スナップショット・アイソレーションを使用します。「スナップショット・アイソレーション」 125 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。スナップショット・トランザクションによってそのローを更新しようとしても失敗となり、スナップショットの更新競合エラーが返されます。

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
- 書き込みを意図
- 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
共有		競合	競合
意図的	競合		競合
排他	競合	競合	競合

共有テーブル・ロック

共有テーブル・ロックでは、複数のトランザクションがベース・テーブルのデータを読み込めます。ベース・テーブルに共有テーブル・ロックを持つトランザクションは、他のトランザクションが修正中のローにロックをかけていない場合に、テーブルを変更できます。

共有テーブル・ロックは、LOCK TABLE .. IN SHARED MODE 文を実行するなどして取得できます。REFRESH MATERIALIZED VIEW や REFRESH TEXT INDEX 文も WITH SHARE MODE 句を提供するため、リフレッシュ操作中に、基本となるテーブルの共有テーブル・ロックの作成に使用できます。

参照

- 「LOCK TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「REFRESH TEXT INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』

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、他のトランザクションが読み込み、書き込み、スキーマの変更などの操作を行うためにテーブルにアクセスするのを防ぎます。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。

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。REFRESH MATERIALIZED VIEW や REFRESH TEXT INDEX 文も WITH

EXCLUSIVE MODE 句を提供するため、リフレッシュ操作中に、基本となるテーブルの排他テーブル・ロックの作成に使用できます。

参照

- 「LOCK TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「REFRESH TEXT INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』

位置ロック

ロー・ロックのほかに、SQL Anywhere では幻または幻ローが存在するための異常事態を避けるために設計された、キー範囲のロック形式を実装しています。位置ロックが関係するのは、独立性レベル 3 で実行しているトランザクションをデータベース・サーバが処理しているときだけです。

独立性レベル 3 で実行されるトランザクションは、直列化可能であるとされています。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実行される更新アクティビティの影響を受けてはならないということです。実際に独立性レベル 3 のトランザクションは、計算結果に影響を与える可能性のあるローが取り込まれる INSERT または UPDATE (つまり幻) による影響を受けることはありません。SQL Anywhere では位置ロックを使用して、そのような更新が発生することを防ぎます。位置ロックは、独立性レベル 2 (繰り返し可能読み出し) と独立性レベル 3 とを区別する追加のロック処理です。

幻ローが作成されないように、テーブルの物理スキャン内で位置のロックが取得されます。逐次スキャンの場合は、現在のローのロー識別子を基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・キー値を基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、他のトランザクションが、ローの順序付けに使用される特定範囲の値に関する挿入を行うことを防ぐことができます。これには、INSERT 文やインデックス付き属性の値を変更する UPDATE 文も含まれます。スキャン位置がロックされた場合、UPDATE 文は、直後に INSERT 要求が続く、インデックス・エントリに対する DELETE 要求であると見なされます。

SQL Anywhere でサポートされる位置ロックには、幻ロックと対幻ロックの 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。独立性レベル 3 は、幻に関して一貫性を保証する唯一の独立性レベルです。

インデックス・スキャンでは、幻ロックはインデックスを介して読み込まれる各ローで取得されます。また、条件を満たすインデックス範囲の最後にインデックスが挿入されることを防ぐため、インデックス・スキャンの最後に幻ロックが 1 つ追加取得されます。インデックス・スキャンで幻ロックを使用すると、テーブルに新しいローが挿入されたり、インデックス付きの値 (幻ロックの対象となるポイントにインデックス・エントリが作成される原因となる) が更新されたりすることが原因で、幻が作成されないようになります。

逐次スキャンでは、挿入処理によって結果セットが変更されないように、幻ロックはテーブルの行ごとに取得されます。このため、独立性レベル 3 のスキャンにより、データベースの同時実行性が悪影響を受けることがあります。1 つ以上の幻ロックは挿入ロックと競合し、1 つ以上の読み込みロックは書き込みロックと競合しますが、幻ロック／挿入ロックと読み込みロック／書き込みロックの間に相互作用はありません。たとえば、書き込みロックは読み込みロックのかかったローにかけることはできませんが、幻ロックだけがかかったローにはかけることができます。この柔軟性のため、データベース・サーバでは多くのオプションを利用できます。しかしそのために、幻ロックをかける場合は、読み込みロックの設定に特別な注意が要求されます。これを怠ると、他のトランザクションがローを削除してしまう可能性があります。

挿入ロック

挿入ロックは対幻ロックとも呼ばれ、ローを挿入する権利を確保するためにスキャン位置に設定される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。このため、ローの書き込みロックは排他ロックであることが重要です。これに対して、複数のトランザクションが 1 つのローを読んでも問題は生じません。ローを変更するわけではないので、

競合することはありません。このため、ローの読み込みロックは多くの接続間で共有されていても構いません。

次の表に、競合するロックの組み合わせを示します。スキーマ・ロックはローに適用されないため、含めてありません。

	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		
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 で実行するときほど多くのロックを使用しません。それぞれのレベルでデータベース・サーバのオペレーションが異なる点は、2 つしかありません。

オペレーションの最初の違いはロックの設定とは無関係で、むしろロックへの配慮に関するものです。独立性レベル 0 では、別のトランザクションが書き込みロックを取得しても、トランザクションはすべてのローを読み込むことができます。一方独立性レベル 1 のトランザクションは、各ローを読み込む前に書き込みロックがかかっているかをチェックします。書き込みロックがかかっているローは読み込むことができません。この場合、ダーティ・データを読み込むことになるからです。READPAST ヒントを使用すると、サーバは書き込みロックがかかっているローを無視できます。ただし、トランザクションのブロックがなくなると、READPAST ヒントのセマンティックは独立性レベル 1 のセマンティックと一致しなくなります。「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。

オペレーションの 2 番目の違いは、カーソル安定性に影響します。カーソル安定性は、カーソルの現在のローに短期間の読み込みロックを設定して達成されます。この読み込みロックはカーソルを移動すると解放されます。カーソルの内容がジョインの結果を示している場合は、複数のローが影響を受けます。この場合、データベース・サーバはカーソルの現在のローに情報を提供したすべてのローに短期間の読み込みロックをかけ、カーソルの別のローが現在のローになるとこれらのロックを解放します。

独立性レベル 2 の SELECT 文

独立性レベル 2 では、データベース・サーバはそのオペレーションを修正し、繰り返し可能読み出しのセマンティックを保証します。SELECT 文がテーブルのすべてのローから値を返す場合、データベース・サーバはローを読み込むときに各ローに読み込みロックをかけます。SELECT に WHERE 句や結果におけるローを制限する他の条件が含まれている場合は、データベース・サーバは各ローを読み込み、ローの値が条件を満たしているかをテストし、条件を満たすローに読み込みロックをかけます。取得された読み込みロックは、長期間の読み込みロックであり、暗黙的または明示的な COMMIT 文または ROLLBACK 文によってトランザクションが完了するまで保持されます。独立性レベル 1 と同じように、独立性レベル 2 ではカーソル安定性が保証され、ダーティ・リードは許可されません。

独立性レベル 3 の SELECT 文

独立性レベル 3 では、データベース・サーバはすべてのトランザクションが直列化可能であることを確認する必要があります。特に、独立性レベル 2 での要件に加えて、同じ文を再実行するとすべての環境で同じ結果を返すことが保証されるように、幻ローを防ぐ必要があります。

この要件を満たすために、データベース・サーバは読み込みロックと幻ロックを使用します。独立性レベル 3 で SELECT 文を実行すると、データベース・サーバは結果セットの計算で処理される各ローで読み込みロックを取得します。こうすることで、そのトランザクションが完了するまで他のトランザクションがそれらのローを修正できないようにします。

この要件は、データベース・サーバが独立性レベル 2 で実行するオペレーションと似ていますが、これらのローが SELECT 文の WHERE 句、ON 句、または HAVING 句の述部を満たすかどうかに関係なく、読み込まれた各ローにロックをかけなければならない点が異なります。たとえば、販売部のすべての従業員名を選択する場合、サーバはトランザクションが独立性レベル 2 または 3 のどちらで実行されているかに関係なく、販売部の従業員に関する情報が含まれているすべてのローにロックをかける必要があります。ただし、独立性レベル 3 では、販売部に所属しない従業員のローにも読み込みロックをかける必要があります。そうでない場合、最初のトランザ

クションが実行されている間に、別のトランザクションが別の従業員を販売部に移動する可能性があります。

読み込まれた各ローに読み込みロックをかける必要がある場合、次の 2 つの影響があります。

- データベース・サーバは、独立性レベル 2 で必要とされるロックよりも多くのロックをかけなければならない場合がある。取得される幻ロックの数は、スキャンのために取得される読み込みロックの数よりも 1 つ多くなります。倍増したロックのオーバーヘッドは、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、新しいページがテーブルに割り当てられることがあります。また、データベース・ファイルのサイズが増大することがあります。
4. 新しいローに値を入れます。
5. ローを追加するテーブルに挿入ロックをかけます。挿入ロックは排他ロックであるため、一度挿入ロックをかけると、独立性レベル 3 の他のトランザクションは、幻ロックをかけて挿入をブロックすることができません。
6. 新しいローに書き込みロックをかけます。書き込みロックが取得されると、挿入ロックが解放されます。
7. テーブルにローを挿入します。ここで、独立性レベル 3 の他のトランザクションは初めて新しいローの存在に気が付きます。ただし、すでに書き込みロックがかかっているため、これらのトランザクションはそのローの修正や削除はできません。
8. 影響を受けるすべてのインデックスを更新し、必要に応じてユニークであることを確認します。プライマリ・キーの値はユニークである必要があります。他のカラムもユニークな値だけを含むように定義される場合があります。このようなカラムが存在する場合は、ユニーク性が検証されます。
9. テーブルが外部テーブルである場合は、プライマリ・テーブルの共有スキーマ・ロックを保持していなければ取得し、挿入される外部キー・カラムの値が NULL でない場合は、プライマリ・テーブルの一致するプライマリ・ローで読み込みロックを取得します。データベース・サーバは、挿入トランザクションで COMMIT が実行されたときにプライマリ・ローが存在していることを保証する必要があります。この確認は、プライマリ・ローの読み込みロックを取得して行います。読み込みロックをかけても他のトランザクションは自由にそのローを読むことができますが、削除や更新はできません。

   対応するプライマリ・ローが存在しない場合は、参照整合性制約違反が発生します。

最後の手順の後、テーブルで定義された AFTER INSERT トリガが起動します。トリガ内の処理におけるロック動作は、アプリケーションの場合と同じです。トランザクションがコミット (すべての参照整合性制約が満たされる) またはロールバックされると、すべての長期間ロックが解放されます。

ユニーク性

特定のカラムまたはカラムの組み合わせに設定される値のすべてをユニークにすることができます。ユーザがあえて作成しなくても、データベース・サーバがそのユニークなカラムに対するインデックスを作成し、それによって値のユニーク性を保証しています。

特に、プライマリ・キーの値はすべてユニークである必要があります。データベース・サーバは、すべてのテーブルのプライマリ・キーに対するインデックスを自動的に作成します。プライマリ・キーに対するインデックスを作成するようデータベース・サーバに要求しないでください。これをすると、重複するインデックスが作成されてしまいます。

オーファンと参照整合性

外部キーは、通常別のテーブルにあるプライマリキーまたは一意性制約を参照します。そのプライマリ・キーが存在しない場合、対応する外部キーは「オーファン」と呼ばれます。SQL Anywhere は、データベースに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。
   a. 更新の候補となるローを識別します。ローがスキャンされている間は、ローはロックされます。デフォルトのロック動作については、「独立性レベルと一貫性」 123 ページを参照してください。

      独立性レベル 2 と 3 では、デフォルトのロック動作とは異なる次のような違いが発生します。読み込みロックではなく書き込みを意図したローレベルのロックが取得されます。また、書き込みを意図したロックは、更新の候補としては最終的には拒否されたローで取得される場合があります。

   b. 手順 2.a で識別された候補となる各ローは、残りのシーケンスに従います。
3. 影響を受けるローに書き込みロックをかけます。
4. UPDATE 文により、影響を受けるそれぞれのカラムの値を更新します。
5. インデックス付きの値を変更した場合は、新しいインデックス・エントリを追加します。ローの元のインデックス・エントリは残りますが、削除済みのマークが付けられます。短期間の挿入ロックが保持されている間に、新しい値の新しいインデックス・エントリが挿入されます。サーバは、必要に応じてインデックスのユニーク性を検証します。
6. 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」 154 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においてプライマリ・テーブルであり、かつ関係の UPDATE アクションが RESTRICT でない場合、外部テーブル (間) で共有スキーマ・ロックを、各外部テーブルで書き込みを意図したテーブル・ロックをそれぞれ取得することで、外部テーブル内で影響のあるローを特定し、次に影響のあるすべてのローで書き込みロックを取得して、必要に応じてそれぞれのローを修正します。このプロセスは、参照整合性制約のネストした階層でカスケードされることがあります。

最後の手順の後で、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。
   a. 更新の候補となるローを識別します。ローがスキャンされている間は、ローはロックされます。デフォルトのロック動作については、「独立性レベルと一貫性」 123 ページを参照してください。

      独立性レベル 2 と 3 では、デフォルトのロック動作とは異なる次のような違いが発生します。読み込みロックではなく書き込みを意図したローレベルのロックが取得されます。また、書き込みを意図したロックは、更新の候補としては最終的には拒否されたローで取得される場合があります。

   b. 手順 2.a で識別された候補となる各ローは、残りのシーケンスに従います。
3. 削除するローに書き込みロックをかけます。
4. 他のトランザクションから見えなくなるように、ローをテーブルから削除します。ローは、トランザクションがコミットされるまで破壊できません。ローを破壊すると、トランザクションをロールバックするオプションが削除されてしまうからです。削除されたローのインデックス・エントリはトランザクションの完了まで残りますが、削除済みのマークが付けられます。これにより、他のトランザクションは同じローを再挿入できません。
5. テーブルが参照整合性関係においてプライマリ・テーブルであり、かつ関係の DELETE アクションが RESTRICT でない場合、外部テーブル (間) で共有スキーマ・ロックを、各外部テーブルで書き込みを意図したテーブル・ロックをそれぞれ取得することで、外部テーブル内で影響のあるローを特定し、次に影響のあるすべてのローで書き込みロックを取得して、必要に応じてそれぞれのローを修正します。このプロセスは、参照整合性制約のネストした階層でカスケードされることがあります。

参照整合性に違反しない場合は、トランザクションをコミットできます。参照整合性を調べるために、データベース・サーバは削除によって作成されたオーファンを追跡します。COMMIT の実行時、サーバはオペレーションをトランザクション・ログ・ファイルに記録し、すべてのロックを解放します。

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化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。これにより、別のトランザクションがそのローを修正したために最初のトランザクションをロールバックできなくなる状況を回避します。

読み込みロックはある特定の状況でのみ解放されます。独立性レベル 1 では、ローがカーソルの現在のローになった場合にのみ、トランザクションによって読み込みロックがかけられます。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 で繰り返し可能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」 159 ページを参照してください。

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化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」 125 ページを参照してください。

直列化可能なスケジュール

トランザクションを同時に処理するには、データベース・サーバは 1 つのトランザクションでいくつかのコンポーネント文を実行し、次に他のトランザクションのコンポーネント文を実行してから、最初のトランザクションのオペレーション処理を続行する必要があります。各種トランザクションのコンポーネント・オペレーションをインタリーブする順序を、「スケジュール」と呼びます。

この方法でトランザクションを同時に適用すると、前の項で説明した一貫性が失われる 3 つのケースを含む、さまざまな結果が生じる可能性があります。トランザクションが順次実行された場合、つまり 1 つのトランザクションが完全に終了した後、次のトランザクションが開始された場合、データベースの最終状態が達成されることがあります。トランザクションをある順序で順次実行した結果、データベースが実際のスケジュールと同じ状態になった場合、そのスケジュールは「直列化可能」であるといいます。

直列化可能性は、一般に正当性の基準として受け入れられています。直列化可能なスケジュールは、データベースがトランザクションの同時実行により影響を受けないため、正しいスケジュールとして受け入れられます。

トランザクションの直列化可能性は、独立性レベルの影響を受けます。独立性レベル 3 では、すべてのスケジュールは直列化可能です。デフォルト設定値は 0 です。

直列化可能とは、同時実行性が影響を与えないということ

トランザクションが順次実行されるときでも、データベースの最終状態はこれらのトランザクションが実行される順序によって異なります。たとえば、1 つのトランザクションが特定のセルに値 5 を設定し、別のトランザクションが同じセルに 6 を設定すると、そのセルの最終値は最後に実行されたトランザクションによって決まります。

スケジュールが直列化可能であることがわかっても、トランザクションを最も効率よく実行する順序が決まるわけではありません。同時実行性の影響がないというだけです。トランザクション・セットをある順序で順次実行することによって達成される結果は、すべて正しいと見なされます。

直列化不可能なスケジュールは矛盾を生じさせる

「典型的な矛盾のケース」 131 ページで説明した矛盾のケースは、スケジュールが直列化可能でないときに生じる典型的な問題です。いずれのケースでも、すべてのトランザクションが順次実行された場合には起こり得ないような結果を生じさせる方法で文がインタリーブされたため、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。

各種独立性レベルでの典型的なトランザクション

各種の独立性レベルは、それぞれ適するタスクが異なります。以下の情報を参考にして、特定のオペレーションに最も合うレベルを判断してください。

典型的なレベル 0 トランザクション

データのブラウズや入力を伴うトランザクションは、終了するのに何分もかかり、相当数のローを読み込みます。独立性レベルが 2 または 3 の場合、同時実行性が犠牲になります。この種のトランザクションには、レベル 0 または 1 が使われます。

たとえば、データベースから大量のデータを読み込んで統計的にまとめる作業を行う意思決定支援アプリケーションは、後で修正されるローを多少読み込んでも大きな影響は受けません。このようなアプリケーションに上位レベルの独立性を要求すると、大量のデータに読み込みロックをかけてしまい、他のアプリケーションが書き込みアクセスできなくなります。

典型的なレベル 1 トランザクション

独立性レベル 1 は、カーソルとともに使用すると便利です。この 2 つを組み合わせると、ロック要件をそれほど増大せずにカーソルの安定性を保てます。SQL Anywhere は、カーソルの現在ローにかけられた読み込みロックを早期に解放してこれを達成します。読み込みロックは、レベル 2 や 3 では繰り返し可能読み出しを保証するためにトランザクションが終了するまで維持する必要があります。

たとえば、カーソルを使用して在庫レベルを更新するトランザクションには独立性レベル 1 が適しています。なぜなら、品目の入荷や販売があるたびに在庫レベルを調整した内容が失われることがなく、このように頻繁に調整しても、他のトランザクションへの影響が最小限で済むからです。

典型的なレベル 2 トランザクション

独立性レベル 2 では、条件を満たすローは他のトランザクションが変更することはできません。このレベルは、ローを複数回読み込み、最初の結果セットに含まれるローが変更されないことを保証する必要があるときに使用します。

比較的多数の読み込みロックが必要となるため、この独立性レベルの使用には注意を要します。レベル 3 のトランザクションと同様に、データベースとインデックスを慎重に設計することで、ロック数も減り、データベースのパフォーマンスを向上させることができます。

典型的なレベル 3 トランザクション

独立性レベル 3 はセキュリティを最も重んじるトランザクションに適しています。幻ローを防ぐと、新しいローがオペレーションの途中で追加されて結果が壊される心配をせずに、マルチステップ・オペレーションを実行できます。

独立性レベル 3 がいかに高度の整合性を提供するとしても、同時に多数のトランザクションの実行をサポートする必要がある大きなシステムでは使用を控える必要があります。SQL Anywhere はこのレベルでは他のレベルよりも多くのロックを設定するため、1 つのトランザクションが他の多くのトランザクションの処理を妨げる可能性があります。

独立性レベル 2 と 3 での同時実行性の改善

独立性レベル 2 と 3 は多くのロックを使用するため、これらのレベルを常用するデータベースでは、精密な設計が特に重要です。直列化可能なトランザクションを利用する必要がある場合、データベース、特にインデックスに関しては、プロジェクトのビジネス・ルールを念頭において設計することが重要です。大きなトランザクションを小さく分割すると、ローがロックされる時間も短縮し、パフォーマンスが向上します。

直列化可能なトランザクションは、他のトランザクションをブロックする可能性が最も大きくなりますが、必ずしも効率が低下するわけではありません。これらのトランザクションを処理する場合、SQL Anywhere では、ロック数が増加してもパフォーマンスは向上するという最適化を実行できます。たとえば、探索条件と一致するかどうかに関係なく、すべてのローには読み込みロックがかけられるので、データベース・サーバはローの読み込みとロックの設定オペレーションを自由に実行することができます。

ロックの影響の削減

同時に実行される他のトランザクションに影響を及ぼす可能性のある多数のロックを設定しないで済むようにするには、トランザクションを独立性レベル 3 で実行しないことをおすすめします。

オペレーションの性質上、独立性レベル 3 で実行する必要がある場合は、読み込むローやインデックス・エントリの数をできるだけ少なくするようにクエリを設計し、同時実行性への影響を減らすことができます。これによって、レベル 3 のトランザクションの処理速度が増し、さらに重要なこととして、設置するロック数を減らすことができます。

独立性レベル 3 で実行するオペレーションが 1 つでもある場合は、インデックスを追加することでトランザクションの速度が向上する場合があります。インデックスには次の 2 つの利点があります。

- インデックスの使用により、ローを効率良く見つけることができる。
- 検索にインデックスを使用するとロック数が少なくて済む。

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」 143 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、「データベース・パフォーマンスの改善」 189 ページを参照してください。

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仮定します。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。

注意
このチュートリアルが正常に動作するためには、Interactive SQL (**[ツール]** - **[オプション]** - **[SQL Anywhere]**) の**[データベース・ロックの自動解放]** オプションをオフにする必要があります。

この例では、2 人の従業員が同時に SQL Anywhere 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. **[接続]** ウィンドウで、Sales Manager として SQL Anywhere サンプル・データベースに接続します。
   - **[ODBC データ・ソース名]** フィールドで、**[SQL Anywhere 11 Demo]** を選択します。
   - **[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]** をクリックします。
3. Interactive SQL をもう 1 つ起動します。
4. **[接続]** ウィンドウで、Accountant として SQL Anywhere サンプル・データベースに接続します。
   - **[ODBC データ・ソース名]** フィールドで、**[SQL Anywhere 11 Demo]** を選択します。
   - **[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」 123 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;
```

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. **[接続]** ウィンドウで、Sales Manager として SQL Anywhere サンプル・データベースに接続します。
   - **[ODBC データ・ソース名]** フィールドで、**[SQL Anywhere 11 Demo]** を選択します。
   - **[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. **[接続]** ウィンドウで、Accountant として SQL Anywhere サンプル・データベースに接続します。
   - **[ODBC データ・ソース名]** フィールドで、**[SQL Anywhere 11 Demo]** を選択します。
   - **[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 )
   ```



```
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」 127 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」 163 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。この例では、前の例と同じ 2 人の従業員が同時に SQL Anywhere サンプル・データベースを使用するケースを示します。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仮定します。Accountant は最近注文があったいくつかの品目の価格を調べるとします。

この例では、独立性レベル 0 ではなく 1 を使って、両方の接続を開始します。独立性レベル 0 は SQL Anywhere 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。

注意
このチュートリアルが正常に動作するためには、Interactive SQL (**[ツール]** - **[オプション]** - **[SQL Anywhere]**) の**[データベース・ロックの自動解放]** オプションをオフにする必要があります。

1. Interactive SQL を起動します。
2. **[接続]** ウィンドウで、Sales Manager として SQL Anywhere サンプル・データベースに接続します。
   - **[ODBC データ・ソース名]** フィールドで、**[SQL Anywhere 11 Demo]** を選択します。
   - **[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]** をクリックします。

3. Interactive SQL をもう 1 つ起動します。
4. **[接続]** ウィンドウで、Accountant として SQL Anywhere サンプル・データベースに接続します。
   - **[ODBC データ・ソース名]** フィールドで、**[SQL Anywhere 11 Demo]** を選択します。
   - **[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

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化不可能となります。

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、Sales Manager のトランザクションが完了します。

ID	Name	UnitPrice
500	Visor	7.00
501	Visor	7.00

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。

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」 139 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. **[接続]** ウィンドウで、Sales Manager として SQL Anywhere サンプル・データベースに接続します。
   - **[ODBC データ・ソース名]** フィールドで、**[SQL Anywhere 11 Demo]** を選択します。
   - **[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]** をクリックします。
3. Interactive SQL をもう 1 つ起動します。

4. **[接続]** ウィンドウで、Accountant として SQL Anywhere サンプル・データベースに接続します。
   - **[ODBC データ・ソース名]** フィールドで、**[SQL Anywhere 11 Demo]** を選択します。
   - **[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 ID, Name, UnitPrice
   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;
```

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

このチュートリアルでは、表示される幻ローを確認します。

注意
このチュートリアルが正常に動作するためには、Interactive SQL (**[ツール] - [オプション] - [SQL Anywhere]**) の**[データベース・ロックの自動解放]** オプションをオフにする必要があります。

1. Interactive SQL の 2 つのインスタンスを起動します。「チュートリアル：繰り返し不可能読み出し」 167 ページの手順 1 ～ 4 を参照してください。
2. 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;
```

3. 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;
```

4. 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

5. 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

6. 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

DepartmentID	DepartmentName	DepartmentHeadID
600	Foreign Sales	129

新しく追加されたローは、「幻ロー」と呼ばれます。これは、Accountant の観点から見ると、このローがどこからともなく出現した幻のように映るためです。Accountant は独立性レベル 2 で接続されます。このレベルでは、サーバは使用中のローにだけロックをかけます。他のローにはロックがかけられないため、Sales Manager が新しいローを挿入するのを妨げるものはありません。

7. 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。

```
SET TEMPORARY OPTION isolation_level = 3;
SELECT *
FROM Departments
ORDER BY DepartmentID;
```

8. 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] - [中断]**) をクリックしてこのエントリを一時停止します。

9. 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

応じて、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」 143 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」 173 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、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

DepartmentID	DepartmentName	DepartmentHeadID
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。

```
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。

   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

このチュートリアルでは、Accountant と Sales Manager はどちらも SalesOrder テーブルと SalesOrderItems テーブルに関わるタスクを行います。Accountant は、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。

注意
このチュートリアルが正常に動作するためには、Interactive SQL (**[ツール]** - **[オプション]** - **[SQL Anywhere]**) の**[データベース・ロックの自動解放]** オプションをオフにする必要があります。

1. Interactive SQL の 2 つのインスタンスを起動します。「チュートリアル：繰り返し不可能読み出し」 167 ページの手順 1 ～ 4 を参照してください。

2. 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;
   ```

3. 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、注文情報、従業員データがそれぞれ別のテーブルに保存されます。外部キー関係を使用してこれらのテーブルをジョインすることにより、必要な情報を結合します。

   ```
   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
   ORDER BY EmployeeID;
   ```

EmployeeID	GivenName	Surname	April sales
129	Philip	Chin	2160.00
195	Marc	Dill	2568.00
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
...	...	...	...

4. 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;
```

5. 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。

6. 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;
```

7. 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;
```

8. 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。

9. 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]** - **[中断]**) をクリックしてこのエントリを一時停止します。

10. 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。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。

11. 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]** - **[中断]**) をクリックしてこのエントリを一時停止します。

12. 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;
    ```

両方のウィンドウを閉じます。

プライマリ・キーの生成と同時実行性

状況によっては、データベースに自動的にユニークな番号を生成させたいという場合があります。たとえば、商品の送り状を格納するテーブルを作成する場合、販売スタッフではなくデータベースが自動的にユニークな送り状番号を割り当てることができます。

例

たとえば、商品の送り状番号は、1 つ前の送り状番号に 1 を加えて作成できます。しかし、複数の人間がデータベースに送り状番号を入力するときは、この方法は使えません。2 人の従業員が同じ番号を選択する可能性があるからです。

この問題を解決するには、次に示すように方法がいくつかあります。

- 送り状番号を入力するユーザごとに数字の範囲を設定する。

  このスキームは、カラム user name と invoice number を持つテーブルを作成することで実装します。ローの 1 つは、送り状番号を入力するユーザを記録するのに使います。ユーザが送り状を追加するごとに、テーブル内の数字は 1 増えて新しい送り状に使われます。データベースのすべてのテーブルを処理するには、テーブルに 3 つのカラム (テーブル名、ユーザ名、最後のキー値) が必要です。各ユーザに十分な数字が確保されているかどうかを定期的に確認する必要があります。

- 2 つのカラム table name と last key を持つテーブルを作成する。

  このテーブルには、最後に使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ユニークなローを識別できます。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」 96 ページを参照してください。

- 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実行性

CREATE INDEX、ALTER TABLE、TRUNCATE TABLE のように、テーブル全体を変更するようなデータ定義文は、その文が動作しているテーブルがその時点で他の接続に使われている場合は処理されません。これらのデータ定義文は時間がかかり、データベース・サーバは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 文は同時実行性の問題は起こしません。

GRANT 文、REVOKE 文、SET OPTION 文も同時実行性の問題を起こしません。これらのコマンドは、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 - サーバ管理と「データ定義文」『SQL Remote』を参照してください。

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。トランザクションを完了するには、すべての変更をロールバックするか、変更をコミットして永続的なものにします。

システム障害が発生したときにデータ・リカバリを行うためには、トランザクションが必要です。トランザクションは、同時に実行されるトランザクションの文を編成する上で中心的な役割を果たします。

パフォーマンスを向上させるには、複数のトランザクションを同時に実行する必要があります。各トランザクションは、コンポーネント SQL 文で構成されています。複数のトランザクションを同時に実行する場合、データベース・サーバは個々の文の実行をスケジュールします。トランザクションを同時に実行すると、順次実行の場合とは異なり、矛盾が生じる可能性があります。

独立性レベルの定義には、次の 4 種類の矛盾が使用されます。

- **ダーティ・リード**　あるトランザクションがデータを修正した後、コミットする前に別のトランザクションがそのデータを読み込んでしまうこと。
- **繰り返し不可能読み出し**　あるトランザクションが同じローを 2 度読み込んだ場合に、得られる値が異なること。
- **幻ロー**　トランザクションが特定の基準に従ってローを 2 回選択したときに、2 回目の結果セットに新しいローが含まれること。
- **更新内容の消失**　トランザクションがローに対して行った変更が、別のトランザクションが前のデータを基にして更新内容を保存することを許可されたため、完全に消失してしまうこと。

スケジュールに従って文を実行した結果が、各トランザクションを順次実行した結果と同じ場合、そのスケジュールは直列化可能であるといいます。スケジュールが直列化可能である場合、それは「正しい」スケジュールと言えます。直列化可能なスケジュールは上記のような矛盾を引き起こしません。

ロックは、許可する干渉の量とタイプを制御します。SQL Anywhere では、独立性レベル 0、1、2、3 の 4 つのロック・レベルが使用できます。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化可能であること、つまり、すべてのトランザクションを実行した結果とそれらを順次実行した結果が同じになることを保証します。

残念なことに、あるトランザクションが設定したロックが他のトランザクションの進行を妨げることがあります。この問題を解決するには、矛盾が許されるかぎり、低い独立性レベルを使用するのが得策です。独立性レベルが高いほどデータの一貫性は向上しますが、同時実行性は低下し、データベースがトランザクションを同時に実行する効率も低下します。オペレーションごとに最適な独立性を決定するには、一貫性とパフォーマンス向上のバランスを取る必要があります。

異なるトランザクション間でロックの競合が起きると、ブロックまたはデッドロックとなります。SQL Anywhere には、この両方を扱うメカニズムが含まれており、それらを制御するオプションを備えています。

しかし、独立性レベルの高いトランザクションが**必ずしも**同時実行性に影響を与えるわけではありません。他のトランザクションは、ロックされたローにアクセスする場合にだけ妨げられます。データベースとトランザクションを注意深く設計することで、同時実行性を向上させることができます。たとえば、1 つのトランザクションを 2 つの短いトランザクションに分割してロックが保持される時間を短縮したり、インデックスを追加して、独立性レベルの高いトランザクションを少ないロックで実行できます。

データベース・パフォーマンスのモニタリングと改善

この項では、データベースとアプリケーションのプロファイリング・アクティビティを実行する方法、パフォーマンスをモニタリングおよび改善する方法、特定のパフォーマンス問題をトラブルシューティングする方法について説明します。

データベース・パフォーマンスの改善

目次

データベースのパフォーマンスを改善するためには、既存のデータベースが最適レベルで稼働しているかどうかを確認する必要があります。この項では、SQL Anywhere 分析ツールを使用したデータベース・パフォーマンスの分析および調整について説明します。

SQL Anywhere には、運用データベースのパフォーマンス上の問題を検出するための診断ツールがいくつかあります。これらのツールのほとんどは「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。診断トレーシングのデータを使用して、「アプリケーション・プロファイリング」などの診断やモニタリングのタスクを行うことができます。

SQL Anywhere のパフォーマンス・データを分析するには、次のような方法があります。

- **アプリケーション・プロファイリング・ウィザード** Sybase Central のアプリケーション・プロファイリング・モードからこのウィザードを使用すると、パフォーマンスを完全に自動的に確認できます。ウィザードの終了時に、パフォーマンスを向上させるための推奨内容が表示されます。「アプリケーション・プロファイリング」 191 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** 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。これにより、特定のユーザやアクティビティのパフォーマンスをモニタリングできます。「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」 205 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」 222 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」 199 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**　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。「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」 193 ページを参照してください。

  システム・プロシージャを使用してプロシージャ・プロファイリングを実装することもできます。「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」 227 ページを参照してください。
- **実行プラン**　この機能では、実行プランを使用して、文に関連するデータベース内の情報にアクセスできます。Interactive SQL または SQL 関数を使用すると、実行プランを表示できます。実行プランは、さまざまなフォーマットで取り出すことができます。また、プランを保存することもできます。「実行プランの解釈」 642 ページを参照してください。

注意
このマニュアルでは、「アプリケーション・プロファイリング」と「診断トレーシング」の各用語を同じ意味で使うことがあります。診断トレーシングは、詳細なアプリケーション・プロファイリングです。

アプリケーション・プロファイリング

アプリケーション・プロファイリングによって生成されたデータを使用して、アプリケーションとデータベースとのやりとりを理解し、パフォーマンス上の問題を特定および解消できます。プロファイリング情報を生成する方法は 2 つあります。**アプリケーション・プロファイリング・ウィザード**を使用して自動的に生成する方法と、Sybase Central のアプリケーション・プロファイリング・モードのツールや機能を使用して生成する方法です。

アプリケーション・プロファイリング・ウィザードは、Windows Mobile ではサポートされていません。ただし、**データベース・トレーシング・ウィザード**はサポートされています。Windows Mobile デバイスからトレーシング・データベースを自動的に作成することはできません。また、Windows Mobile デバイス上のローカル・データベースにトレースすることはできません。Windows Mobile デバイスからトレースし、デスクトップ・コンピュータ上のデータベース・サーバで実行している Windows Mobile データベースのコピーに格納する必要があります。

- **自動アプリケーション・プロファイリング**　Sybase Central の**アプリケーション・プロファイリング・ウィザード**を使用すると、一般的なパフォーマンス上の問題を特定できます。アプリケーション・プロファイリング・ウィザードにより、プロファイリングするアクティビティのタイプを定義できます。完了するとデータベースのパフォーマンスを改善するための推奨内容が表示されます。**アプリケーション・プロファイリング・ウィザード**にはインデックス・コンサルタントも統合されています。インデックス・コンサルタントでは、このデータを使用して、インデックスの改善のための推奨内容が検討されます。

  この方法は、データベースへの接続が少ない環境や、詳細なプロファイリングを必要としない場合に最適です。

- **診断トレーシングを使用した詳細なアプリケーション・プロファイリング**　**データベース・トレーシング・ウィザード**を使用すると、トレーシング・セッション中に返されたデータや、そのデータが格納される場所をカスタマイズできます。また、コマンド・ラインを使用して、カスタマイズされたトレーシング・データを格納したり返すことができます。プロファイリングするアクティビティを制御し、対象とする特定の問題を指定できます。たとえば、データベース・サーバで実行される特定の文、使用されるクエリ・プラン、デッドロック、互いにブロックし合う接続、パフォーマンス統計値などを対象にできます。

  この方法は、データベースの負荷が高い環境や、問題を診断するために詳細なプロファイリングが必要な環境に適しています。トレーシング・セッションをカスタマイズすることで、トレーシングのスコープを特定のアクティビティに制限できます。また、トレーシング・データをリモート・データベースに格納できます。これらの操作で、プロファイリング対象のデータベースの負荷を下げることができます。

  「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」 205 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

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、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実行できます。このウィザードは、アプリケーションとデータベースとのやりとりに関するデータを収集し、収集

されたデータへのアクセスを可能にし、インデックスに関する推奨内容がある場合には表示します。「アプリケーション・プロファイリング・ウィザード」 191 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」 205 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 Mobile で実行しているデータベースのトレーシング・セッションは作成できません。**データベース・トレーシング・ウィザード**を使用する必要があります。「診断トレーシング・セッションの作成」 218 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・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**が自動的に開始されないようにするには、ウィザードの最初のページで **[今後、アプリケーション・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 11] - [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・ウィザードを使用するには、次の手順に従います (Sybase Central の場合)。

1. Sybase Central を開きます。
2. DBA 権限または PROFILE 権限のあるユーザとしてデータベースに接続します。
3. **[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. **アプリケーション・プロファイリング・ウィザード**の指示に従います。**[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。

   ウィザードは次のことを行います。

   - 診断トレーシング情報を格納するローカル・データベースを作成する。
   - ネットワーク・サーバを起動する。
   - トレーシング・セッションを開始する。
   - プロファイリング対象のアプリケーションの実行を要求するプロンプトを表示する。
5. **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」 200 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」 197 ページを参照してください。

参照

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

アプリケーション・プロファイリング・モードでのプロシージャ・プロファイリング

この項では、Sybase Central のアプリケーション・プロファイリング・モードを使用して、プロシージャ・プロファイリングを実行する方法について説明します。プロシージャ・プロファイリングの結果にアクセスするには、この方法を推奨します。ただし、SQL 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」 227 ページを参照してください。

プロシージャ・プロファイリングは、プロシージャ、ユーザ定義関数、イベント、システム・トリガ、トリガの実行所要時間を示します。プロファイリング中にこれらのオブジェクトが実行されたら、行ごとの実行時間も表示できます。プロシージャ・プロファイリングの結果の情報を使用すると、どのオブジェクトを微調整すればデータベース内のパフォーマンスを向上できるかを判断できます。

プロシージャ・プロファイリングでは、要求ロギングでコストが高いと判断されたストアド・プロシージャ、関数、イベント、トリガなどの特定のデータベース・プロシージャの分析もできます。また、トリガ、イベント、ネストされたストアド・プロシージャ・コールなどの隠れたコストの高いプロシージャを発見するためにも役立ちます。さらに、プロシージャ本体内の問題となりそうな箇所をピン・ポイントで見つけるためにも役立ちます。

プロシージャ・プロファイリングの結果は、データベース・サーバによってメモリに格納されます。プロファイリング情報は累積されます。その精度は、1 ミリ秒です。

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。

注意
データベース・サーバが停止すると、プロファイリング情報はすべて削除されます。プロファイリング情報をエクスポートするには、sa_procedure_profile システム・プロシージャを使用します。「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。

SQL 文を使用して、データベース・サーバが保持するプロファイリング情報を問い合わせることはできません。プロファイリング情報は、メモリ内データベース・サーバのデータ構造に保管されます。

♦ 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 (Sybase Central の場合)。

1. Sybase Central を開きます。
2. DBA 権限または PROFILE 権限のあるユーザとしてデータベースに接続します。
3. 左ウィンドウ枠でデータベースを選択します。
4. **[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。
5. **アプリケーション・プロファイリング・ウィザード**の指示に従います。
6. **[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。
7. **[完了]** をクリックします。

   別のモードに切り替えると、プロシージャ・プロファイリング情報の収集を停止するかどうかを確認するメッセージが表示されます。**[いいえ]** を選択すると、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」 194 ページ
- 「プロシージャ・プロファイリングの無効化」 195 ページ
- 「プロシージャ・プロファイリングの結果の分析」 196 ページ
- 「PROFILE 権限」『SQL Anywhere サーバ - データベース管理』

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にするには、次の手順に従います (Sybase Central の場合)。**

1. Sybase Central を開きます。
2. DBA 権限または PROFILE 権限のあるユーザとしてデータベースに接続します。
3. 左ウィンドウ枠でデータベースを選択します。
4. **[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]** をクリックします。
5. 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、**[アプリケーション・プロファイリングの詳細]** ウィンドウ枠で、データベースをクリックし、**[選択されたデータベースにおけるプロファイリング設定の表示]** をクリックします。

   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、左ウィンドウ枠でデータベースを右クリックして **[プロパティ]** を選択します。
6. **[プロファイリング設定]** タブをクリックします。
7. **[すぐにリセット]** をクリックします。
8. **[OK]** をクリックします。

参照

- 「プロシージャ・プロファイリングの有効化」 193 ページ
- 「プロシージャ・プロファイリングの無効化」 195 ページ
- 「プロシージャ・プロファイリングの結果の分析」 196 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を取得し終わったら、プロシージャ・プロファイリングを無効にできます。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 (Sybase Central の場合)。**

1. Sybase Central を開きます。
2. DBA 権限または PROFILE 権限のあるユーザとしてデータベースに接続します。
3. 左ウィンドウ枠でデータベースを選択します。
4. **[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]** をクリックします。

5. **[アプリケーション・プロファイリングの詳細]** ウィンドウ枠で、**[選択されたデータベースにおけるプロファイリング情報の収集の停止]** をクリックします。

♦ 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 (Sybase Central の場合)。

1. Sybase Central を開きます。
2. DBA 権限または PROFILE 権限のあるユーザとしてデータベースに接続します。
3. 左ウィンドウ枠でデータベースを選択します。
4. **[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]** をクリックします。
5. **[アプリケーション・プロファイリングの詳細]** ウィンドウ枠で、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。
6. **[プロファイリング設定]** タブをクリックします。
7. **[すぐにクリア]** をクリックします。
8. **[OK]** をクリックします。

参照

- 「プロシージャ・プロファイリングの有効化」 193 ページ
- 「プロシージャ・プロファイリングのリセット」 194 ページ
- 「プロシージャ・プロファイリングの結果の分析」 196 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。

♦ 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 (Sybase Central の場合)。

1. DBA 権限のあるユーザとしてデータベースに接続し、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」 193 ページを参照してください。
2. 左ウィンドウ枠で、**[トリガ]**、**[システム・トリガ]**、**[プロシージャとファンクション]**、または **[イベント]** のいずれかをダブルクリックします。
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。

必要なオブジェクトが実行されていないために見つからない場合があります。また、実行されたが、結果が再表示されていない可能性があります。[F5] キーを押して、リストを再表示します。

予想よりも多くのオブジェクトが表示される場合もあります。1 つのオブジェクトから別のオブジェクトが呼び出される場合は、ユーザが明示的に呼び出す数よりも多くの項目が表示されます。

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、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。

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。
# msec.	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。

カラム	説明
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。

インデックス・コンサルタント

インデックス・コンサルタントを実行するには、DBA 権限または PROFILE 権限が必要です。

適切なインデックス・セットを選択すると、データベースのパフォーマンスを向上させることができます。SQL Anywhere インデックス・コンサルタントを使用すると、データベースに最適なインデックス・セットが推奨されるため、インデックスを選択する際に役立ちます。

インデックス・コンサルタントは、Interactive SQL を使用して単一のクエリに対して実行するか、Sybase Central のアプリケーション・プロファイリング・モードを使用してデータベースに対して実行できます。データベースを分析する際、インデックス・コンサルタントはトレーシング・セッションを使用してデータを収集し、推奨内容を表示します。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、複数カラムのインデックスおよび単一カラムのインデックスを評価し、クラスタード・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ることにより、データベースまたは単一のクエリを分析します。異なる候補インデックスの効果を調べるために、インデックス・コンサルタントは、インデックス・セットごとにクエリの最適化を繰り返します。実際にクエリは実行しません。

注意
Sybase Central を使用してバージョン 9 のデータベース・サーバに接続できます。ただし、Sybase Central のウィンドウのレイアウトが、アプリケーション・プロファイリング・モードを含まないバージョン 9 のレイアウトに戻ります。Sybase Central のインデックス・コンサルタントの場所や使用方法に関する詳細については、バージョン 9 のマニュアルを参照してください。

参照

- 「インデックスの操作」 75 ページ
- 「インデックス」 673 ページ
- 「アプリケーション・プロファイリング」 191 ページ
- 「PROFILE 権限」 『SQL Anywhere サーバ - データベース管理』
- 「アプリケーション・プロファイリング」 191 ページ
- 「インデックス・コンサルタントの推奨内容の解釈」 200 ページ

クエリに対するインデックス・コンサルタントの推奨内容の確認

♦ 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 (Sybase Central の場合)。

1. Sybase Central を開きます。
2. DBA 権限または PROFILE 権限のあるユーザとしてデータベースに接続します。

3. データベースを右クリックして、**[Interactive SQL を開く]** を選択します。
4. **[SQL 文]** ウィンドウ枠で、クエリを入力します。
5. **[ツール] - [インデックス・コンサルタント]** を選択します。

データベースに対するインデックス・コンサルタントの推奨内容の確認

データベース全体に対するインデックス・コンサルタントの推奨内容を確認するには、Sybase Central のアプリケーション・プロファイリング・モードを使用します。インデックス・コンサルタントが推奨内容を決定するためには、プロファイリング・データが必要になります。次の手順は、データを収集し、**アプリケーション・プロファイリング・ウィザード**で収集したデータを使用して推奨内容を表示させるための簡単な方法です。ただし、アプリケーション・プロファイリング・データがすでにある場合 (**データベース・トレーシング・ウィザード**を使用して、すでにデータベースのプロファイリングを実行した場合など) は、作成したトレーシング・データベースに対してインデックス・コンサルタントを実行することもできます。

♦ 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、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 (Sybase Central の場合)。

1. DBA または PROFILE 権限のあるユーザ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。
3. **アプリケーション・プロファイリング・ウィザード**の指示に従います。

   アプリケーション・プロファイリング・ウィザードが表示されない場合は、**[アプリケーション・プロファイリング] - [アプリケーション・プロファイリング・ウィザードを開く]** を選択し、完了するまでウィザードの指示に従います。
4. Sybase Central で、**[アプリケーション・プロファイリング] - [トレーシング・データベースでのインデックス・コンサルタントの実行]** を選択します。
5. **インデックス・コンサルタント・ウィザード**の指示に従います。

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・インデックスの使用」 77 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。

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・コンサルタントの結果の解釈

インデックス・コンサルタントは、指定された分析の結果を一連のタブで示します。分析の結果は、保存して後で確認できます。

[概要] タブ

[概要] タブには、分析の概要が示されます。これには、クエリ数、推奨インデックス数、推奨インデックスに必要なページ数、推奨インデックスによってもたらされる予測利益などの情報が含まれます。利益値は、内部的なコスト単位に基づいて測定されます。

[推奨インデックス] タブ

[推奨インデックス] タブには、各推奨インデックスに関するデータが含まれます。表示される情報は次のとおりです。

- **[クラスタード]** 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・インデックスの使用」 77 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。「テーブルとページのサイズ」 672 ページ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、利益が大きいことを示します。

  相対利益は、内部アルゴリズムを使用して、[コスト利益の合計] カラムとは別に計算されます。相対利益の推定に含まれるいくつかの要素は、コスト利益の合計には含まれません。たとえば、あるインデックスの存在が、別のインデックスに関連する利益に大きく影響することがあります。この場合、相対利益では、各インデックスの影響を個別に推定しようとします。

  詳細については、「インデックス・コンサルタントの結果の実装」 203 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 (コスト・モデル) に基づいて測定されます。「オプティマイザの仕組み」 590 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。これには、推奨インデックスを適用する前と後の推定コストや、クエリが使用した仮想インデックスなどの情報が含まれます。ボタンを使用すると、要求に対して最適と判断された実行プランを表示できます。

[更新] タブ

[更新] タブには、推奨内容の効果の内訳が示されます。

[未使用のインデックス] タブ

[未使用のインデックス] 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。

参照

- 「インデックスの操作」 75 ページ
- 「インデックス」 673 ページ
- 「アプリケーション・プロファイリング」 191 ページ

インデックス・コンサルタントの結果の実装

インデックス・コンサルタントに用意されている SQL スクリプトを実行することで、その結果を実装することもできますが、結果を評価してから実装することが必要な場合もあります。たとえば、分析中に提案されたインデックスの名前を変更する場合などです。

結果を評価する際には、次の点を検討してください。

- **提案されたインデックスは、期待どおりか。** データベース内のデータと、データベースに対して実行されるクエリをよく理解している場合は、提案されたインデックスの有用性を自分自身の知識に照らして確認するとよい場合があります。提案されたインデックスがめったに実行されない単一のクエリにのみ影響する場合や、小さなテーブルに対するインデックスで全体への影響があまりない場合もあります。インデックス・コンサルタントが削除するように提案したインデックスが、トレーシング・セッションに含まれていなかった他のタスクに使用される場合もあります。
- **提案されたインデックスの効果に密接な相関関係はあるか。** インデックスの推奨では、各インデックスの相対利益を別々に評価しようとします。しかし、2 つのインデックスが、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

- 「インデックス・コンサルタントの推奨内容の解釈」 200 ページ
- 「SQL コマンド・ファイルの使用」 811 ページ
- 「インデックスの操作」 75 ページ
- 「インデックス」 673 ページ
- 「REORGANIZE TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「アプリケーション・プロファイリング」 191 ページ

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た診断トレーシングのデータには、データベース・サーバで処理された文のタイムスタンプと接続 ID が含まれる場合があります。クエリについては、独立性レベル、フェッチされたローの数、カーソル・タイプ、クエリの実行プランも診断とレーシングのデータに含まれます。INSERT 文、UPDATE 文、DELETE 文については、影響を受けるロー数も含まれます。診断トレーシングを使用して、ロックとデッドロックに関する情報を記録し、多数のパフォーマンス統計値を取得することもできます。

診断トレーシング中に収集されたデータを使用して、次の問題点の特定やトラブルシューティングなどの詳細なアプリケーション・プロファイリング・アクティビティを行います。

- 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 (CPU、メモリ、ディスク I/O) の競合
- アプリケーションの論理の問題

トレーシング・データは、インデックス・コンサルタントなどのツールでも、パフォーマンスを向上させるために推奨されるデータベースまたはアプリケーションの具体的な変更方法を判断するために使用されます。

トレーシング・アーキテクチャは信頼性とスケーラビリティに優れています。要求ロギングで記録されるすべての情報と、個別要件に合った分析のための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」 222 ページを参照してください。

参照

- 「アプリケーション・プロファイリング」 191 ページ

トレーシング・セッションのデータ

診断トレーシングのデータは、「トレーシング・セッション」中に収集されます。トレーシング・セッション・データを取得するには、次の 3 つの方法があります。

- Sybase Central の**データベース・トレーシング・ウィザード**を使用する。
- **アプリケーション・プロファイリング・ウィザード**の自動処理の一環として透過的に行う。
- ATTACH TRACING 文と DETACH TRACING 文を使用する。

トレーシング・セッションの実行中は、SQL Anywhere によって指定されたデータベースの診断情報が生成されます。生成されるトレーシング・データの量は、トレーシングの設定によって異

なります。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」 207 ページを参照してください。

プロファイリング対象のデータベースは、「運用データベース」、ソース・データベース、またはプロファイリング対象のデータベースと呼ばれます。トレーシング・データが格納されるデータベースは、「トレーシング・データベース」と呼ばれます。運用データベースとトレーシング・データベースは同じデータベースでもかまいません。ただし、運用データベースのサイズが増大するのを防ぐために、トレーシング・データは別のデータベースに格納することをおすすめします。データベース・ファイルのサイズは一度増大してしまうと、縮小することはできません。また、特に運用データベースが大きく、頻繁に使用される場合は、トレーシング・データの格納および管理にかかるオーバヘッドを別のデータベースで処理した方が、運用データベースのパフォーマンスが向上します。

トレーシング・データが格納されるトレーシング・データベース内のテーブルは、「診断トレーシング・テーブル」と呼ばれます。これらのテーブルの所有者は dbo です。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。

注意
アプリケーション・プロファイリング・ウィザードは、Windows Mobile ではサポートされていません。ただし、**データベース・トレーシング・ウィザード**はサポートされています。また、Windows Mobile デバイスからトレースし、デスクトップ・コンピュータ上のデータベース・サーバで実行している Windows Mobile データベースのコピーに格納する必要があります。Windows Mobile デバイスからトレーシング・データベースを自動的に作成することはできません。また、Windows Mobile デバイス上のローカル・データベースにトレースすることはできません。

トレーシング・セッション中に作成されるファイル

トレーシング・セッションで作成され、使用されるファイルは、**アプリケーション・プロファイリング・ウィザード**を使用するか、**データベース・トレーシング・ウィザード**を使用するかによって異なります。

アプリケーション・プロファイリング・ウィザードを実行すると、ウィザードによって通知せずにバックグラウンドでトレーシング・セッションが取得され、診断テーブルを格納するトレーシング・データベースが作成されます。この外部データベースはウィザードで指定する名前とロケーションで作成され、拡張子が *.adb* です。ウィザードでは、トレーシング・データベースと同じディレクトリに名前が同じで拡張子が *.alg* の分析ログ・ファイルも作成されます。この分析ログ・ファイルには、ウィザードによって行われた分析の結果が含まれ、テキスト・エディタでいつでも開くことができます。

アプリケーション・プロファイリング・ウィザードによって生成されたデータが不要になったら、セッションに関連するトレーシング・データベースと分析ログ・ファイルを削除できます。

データベース・トレーシング・ウィザードを使用してトレーシング・セッションを作成すると、トレーシング・データを内部の運用データベースに保存するか、外部データベース (*tracingData.db* など) に個別に保存するかを確認するメッセージが表示されます。外部トレーシング・データベースを作成することをおすすめします。「外部トレーシング・データベースの作成」 223 ページを参照してください。

注意
トレーシング情報は、データベースのアンロードまたは再ロード操作の一環としてアンロードされません。トレーシング情報を別のデータベースに転送する場合は、sa_diagnostic_* テーブルの内容をコピーして、手動で転送してください。ただし、この方法はおすすめしません。

診断トレーシングの設定

Sybase Central の**アプリケーション・プロファイリング・ウィザード**では、事前に設定されたトレーシングの設定を変更できません。ただし、**データベース・トレーシング・ウィザード**を使用すると、トレーシング・アクティビティのほとんどの側面を設定できます。次のいずれかの方法により、診断トレーシングを設定します。

- Sybase Central の**データベース・トレーシング・ウィザード**を使用する。この方法では有効なトレーシングの設定がすべて表示されるので、この方法をおすすめします。「診断トレーシングの設定の変更」 217 ページを参照してください。
- システム・プロシージャを使用して診断トレーシング・テーブルに格納されている設定を変更する。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。

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。

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・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。

診断トレーシング・レベルの選択

診断トレーシングの設定は、複数のレベルに分類されていますが、これらのレベル内で設定をさらにカスタマイズすることもできます。各レベルで収集される情報のタイプは、「診断トレーシング・タイプ」と呼ばれます。指定できるレベルと、それぞれに含まれる診断トレーシング・タイプについてこの後で説明します。ここに示す診断トレーシング・タイプの説明については、「診断トレーシングのタイプ」 210 ページを参照してください。

診断トレーシングの設定をカスタマイズすると、診断トレーシング・セッション内の不要なトレーシング・データの量を減らすことができます。たとえば、ユーザ AliceB のアプリケーションの実行は遅いが、他のユーザは同じ問題が発生していないとします。このとき必要なのは、AliceB のクエリがどのように実行されているかです。したがって、AliceB がアプリケーションで実行しているすべてのクエリとその他の文、および実行時間が長いクエリのクエリ・プランのリストを収集する必要があります。そのためには、診断トレーシング・レベルを 3 に設定し、1 ～ 2 日のトレーシング・データを生成できます。ただし、このレベルは、他のユーザのパフォー

マンスに大きく影響するので、AliceB のアクティビティだけをトレースします。そのためには、診断トレーシング・レベルを 3 に設定し、診断トレーシングのスコープを USER にカスタマイズし、ユーザ名として AliceB を指定します。診断トレーシング・セッションを数時間実行し、結果を確認します。

診断トレーシングの設定のカスタマイズには**データベース・トレーシング・ウィザード**を使用することをおすすめします。「診断トレーシングの設定の変更」 217 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診断トレーシングの設定を変更しないことをおすすめします。データの解釈が困難になるからです。ただし、変更することはできます。「トレーシング・セッション実行中の診断トレーシングの設定の変更」 218 ページを参照してください。

診断トレーシングのレベル

次のリストは、**データベース・トレーシング・ウィザード**で指定される診断トレーシング・レベルを示したものです。各種の診断トレーシング・タイプの説明については、「診断トレーシングのタイプ」 210 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・カウンタと実行された文のサンプリングが (5 秒に 1 回) 収集されます。このレベルでは、次の診断トレーシング・タイプがあります。
  - volatile_statistics、1 秒ごとのサンプリング
  - 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・カウンタと実行された文のサンプリングが (5 秒に 1 回) 収集され、すべての実行された文が記録されます。このレベルでは、次の診断トレーシング・タイプがあります。
  - volatile_statistics、1 秒ごとのサンプリング
  - 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、次の診断トレーシング・タイプがあります。
  - volatile_statistics、1 秒ごとのサンプリング
  - 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 (通常はオブジェクト名やユーザ名などの識別子) が identifier カラムに格納されていることがあります。スコープ・カラムの値は、**データベース・トレーシング・ウィザード**で指定される設定を反映しています。

scope カラムの値	説明
DATABASE	データベース内で発生し、指定するレベルと条件に対応するすべてのイベントのトレーシング・データを記録します。高コストのクエリのソースを判断するために、バックグラウンドでのデータベースの長期のモニタリングや、短期の診断に使用します。 DATABASE を指定するときに指定する識別子はありません。
ORIGIN	データベースの外部または内部から発生するクエリのトレーシング・データを記録します。 スコープ ORIGIN を指定するときは、External または Internal のいずれかの識別子を指定できます。External は、データベース・サーバ外で発生し、指定するレベルと条件に対応するクエリの文のテキストと関連する詳細情報のログを取るように指定します。Internal は、データベース・サーバ内で発生し、指定するレベルと条件に対応するクエリに関する同じ情報のログを取るように指定します。

scope カラムの値	説明
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。
FUNCTION、PROCEDURE、EVENT、TRIGGER、または TABLE	指定するオブジェクトを使用する文のトレーシング・データが記録されます。オブジェクトが他のオブジェクトを参照する場合、参照先オブジェクトのデータもすべて記録されます。たとえば、イベントをトリガする関数を使用するプロシージャのトレースを行っている場合、3 つのオブジェクトで、指定するレベルと条件に対応する文のログが取られます。特定のオブジェクトのコストが高い場合や、オブジェクトを参照する文が完了するまでの時間が異常に長い場合に使用します。 TABLE スコープは、テーブル、マテリアライズド・ビュー、非マテリアライズド・ビューに使用します。 このスコープの識別子は、オブジェクトの完全に修飾された名前です。

参照

- 「診断トレーシングのタイプ」 210 ページ
- 「診断トレーシング条件」 215 ページ

診断トレーシングのタイプ

次の表は、診断トレーシングで選択できるトレーシング・「タイプ」を示します。各診断トレーシング・タイプには、次に示すように、対応する条件が必要です。また、dbo.sa_diagnostic_tracing_level 診断テーブルの trace_type カラムに格納されます。対応する診断トレーシング条件が trace_condition カラムに格納されていることがあります。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」 215 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収集します。 スコープと条件：この診断トレーシング・タイプは、DATABASE スコープが必要で、SAMPLE_EVERY 条件をデータ収集の間隔として使用します。「診断トレーシングのスコープ」 209 ページと「診断トレーシング条件」 215 ページを参照してください。
NONVOLATILE_STATISTICS	頻繁に変化しないデータベースとサーバの統計値のサンプルを収集します。不揮発性の統計値は、揮発性の統計値よりも頻繁に収集できません。不揮発性の統計値を収集するには、揮発性の統計値を収集する必要があります。不揮発性の統計値のサンプリング間隔は、揮発性の統計値に指定する間隔の倍数である必要があります。 スコープと条件：この診断トレーシング・タイプは、DATABASE スコープが必要で、SAMPLE_EVERY 条件をデータ収集の間隔として使用します。「診断トレーシングのスコープ」 209 ページと「診断トレーシング条件」 215 ページを参照してください。
CONNECTION_STATISTICS	接続の統計値のサンプルを収集します。スコープがデータベースの場合は、データベースへのすべての接続の統計値が収集されます。スコープがユーザの場合は、指定するユーザのすべての接続の統計値が収集されます。スコープが CONNECTION_NAME または CONNECTION_NUMBER の場合、指定する接続の統計値だけが収集されます。CONNECTION_STATISTICS を収集するには、揮発性の統計値を収集する必要があります。サンプリング間隔は、VOLATILE_STATISTICS に指定する間隔の倍数である必要があります。 スコープと条件：この診断トレーシング・タイプは、DATABASE、USER、CONNECTION_NUMBER、CONNECTION_NAME スコープで使用できます。また、SAMPLE_EVERY 条件をデータ収集の間隔として使用します。「診断トレーシングのスコープ」 209 ページと「診断トレーシング条件」 215 ページを参照してください。

trace_type カラムの値	説明
BLOCKING	指定するスコープと条件に従ってブロックに関する情報を収集します。スコープが CONNECTION_NAME または CONNECTION_NUMBER の場合、接続が別の接続をブロックしたとき、または別の接続によってブロックされたときにブロックを記録できます。 スコープと条件：この診断トレーシング・タイプはすべてのスコープで使用でき、NONE、NULL、SAMPLE_EVERY のいずれかの収集条件を指定できます。「診断トレーシングのスコープ」 209 ページと「診断トレーシング条件」 215 ページを参照してください。
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：この診断トレーシング・タイプはすべてのスコープで使用でき、NONE、NULL、SAMPLE_EVERY、ABSOLUTE_COST のいずれかの収集条件を指定できます。「診断トレーシングのスコープ」 209 ページと「診断トレーシング条件」 215 ページを参照してください。
PLANS_WITH_STATISTICS	実行の統計値があるプランを収集します。プランはカーソルのクローズ時間に記録されます。RELATIVE_COST_DIFFERENCE 条件を指定した場合、出力内の統計値の一部は推測された統計値である可能性があります。 スコープと条件：この診断トレーシング・タイプはすべてのスコープで使用でき、すべての収集条件を指定できます。

trace_type カラムの値	説明
STATEMENTS	指定するスコープと条件の SQL 文を収集します。内部変数は、各プロシージャが初めて実行されるときに収集されます。この診断トレーシング・タイプは、STATEMENTS_WITH_VARIABLES、PLANS、PLANS_WITH_STATISTICS、OPTIMIZATION_LOGGING、OPTIMIZATION_LOGGING_WITH_PLANS のいずれかの診断トレーシング・タイプを指定した場合に自動的に追加されます。 スコープと条件：この診断トレーシング・タイプはすべてのスコープで使用でき、すべての収集条件を指定できます。「診断トレーシングのスコープ」 209 ページと「診断トレーシング条件」215 ページを参照してください。
STATEMENTS_WITH_VARIABLES	SQL 文と文に付加された変数を収集します。各変数 (内部変数またはホスト変数) に割り当てられた値も収集されます。 スコープと条件：この診断トレーシング・タイプはすべてのスコープで使用でき、すべての収集条件を指定できます。「診断トレーシングのスコープ」 209 ページと「診断トレーシング条件」 215 ページを参照してください。

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。この診断トレーシング・タイプは、OPTIMIZATION_LOGGING_WITH_PLANS 診断トレーシング・タイプを指定すると自動的に追加されます。 この診断トレーシング・タイプはすべてのスコープに対応し、条件は指定できません。「診断トレーシングのスコープ」 209 ページを参照してください。
OPTIMIZATION_LOGGING_WITH_PLANS	オプティマイザで考慮されたジョイン方式に関するデータを収集します。各方式の実行コストに関する情報や、ジョイン方式のツリー構造を表す完全な XML プランが収集されます。クエリに適用された書き換えに関する情報も収集されます。スコープが DATABASE、CONNECTION_NAME、CONNECTION_NUMBER、ORIGIN、または USER 以外の場合、最初に記録される文のテキストが、クエリの最初のテキストと異なる場合があります。これは、現在の文に最適化ロギングを適用するかどうかを判断する前に書き換えが適用される場合があるからです。OPTIMIZATION_LOGGING トレーシング・タイプは、OPTIMIZATION_LOGGING_WITH_PLANS トレーシング・タイプを指定すると自動的に追加されます。 この診断トレーシング・タイプはすべてのスコープに対応し、条件は指定できません。「診断トレーシングのスコープ」 209 ページを参照してください。

参照

- 「診断トレーシングのスコープ」 209 ページ
- 「診断トレーシング条件」 215 ページ

診断トレーシング条件

次の表は、設定可能な診断トレーシング「条件」を示します。条件は、特定の診断トレーシング・タイプのトレーシング・データが記録されるために満たす必要がある条件です。次の表に示すように、ほとんどの条件には値が必要です。条件は、dbo.sa_diagnostic_tracing_level 診断テーブルの trace_condition カラムに格納されます。対応する値 (ミリ秒単位の時間) が value カラムに格納されていることがあります。条件カラムの値は、**データベース・トレーシング・ウィザード**で指定される設定を反映しています。

trace_condition カラムの値	説明
NONE または NULL	レベルとスコープの条件を満たすトレーシング・データをすべて記録します。この条件を高コストの診断トレーシング・レベル (たとえばプラン) と同時に長時間使用することはおすすめしません。
SAMPLE_EVERY	最後のイベントが記録されてから指定された間隔以上の時間が経過した場合に、レベルとスコープの要件を満たすトレーシング・データを記録します。 値：この条件には時間をミリ秒単位で表した正の整数を指定します。
ABSOLUTE_COST	実行コストが指定する値以上である文を記録します。 値：この条件には、ミリ秒単位のコスト値を指定します。
RELATIVE_COST_DIFFERENCE	予想実行時間と実際の実行時間の差が、指定する値以上である文を記録します。 値：この条件には、パーセント単位のコスト値を指定します。たとえば、予想よりも 2 倍以上低速の文をログに取るには、値 200 を指定します。

参照

- 「診断トレーシングのスコープ」 209 ページ
- 「診断トレーシングのタイプ」 210 ページ

現在の診断トレーシングの設定の確認

Sybase Central の**データベース・トレーシング・ウィザード**を使用して、現在の診断トレーシングの設定を表示できます。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、有効になっている診断トレーシングの設定を取得することもできます。

診断トレーシング設定は、トレーシング・セッションが実行中であるかどうかに関係なく取得できます。

♦ 現在の診断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 権限または PROFILE 権限のあるユーザ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]** をクリックします。
3. 左ウィンドウ枠でデータベースを右クリックして **[トレーシング]** を選択します。

   データベース・トレーシング・ウィザードが表示されない場合は、**[トレーシング]** - **[設定]** を選択します。
4. **[トレーシング・レベルの編集]** リストで、診断トレーシングに現在指定されている設定を確認します。
5. **[キャンセル]** をクリックします。

♦ 現在の診断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 権限または PROFILE 権限のあるユーザ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、現在使用中の診断トレーシングの設定が返されます。enabled カラムが 1 の場合、設定が有効であることを示します。

例

次の文は、sa_diagnostic_tracing_level 診断テーブルに問い合わせ、現在の診断トレーシングの設定を取得する方法を示します。

```
SELECT * FROM sa_diagnostic_tracing_level WHERE enabled = 1;
```

次の表は、クエリの結果セットの例を示します。

id	scope	identifier	trace_type	trace_condition	value	enabled
1	database	(NULL)	volatile_statistics	sample_every	1,000	1
2	database	(NULL)	nonvolatile_statistics	sample_every	60.000	1
3	database	(NULL)	connection_statistics	(NULL)	60,000	1
4	database	(NULL)	blocking	(NULL)	(NULL)	1

id	scope	identifier	trace_type	trace_condition	value	enabled
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』
- 「PROFILE 権限」『SQL Anywhere サーバ - データベース管理』

診断トレーシングの設定の変更

診断トレーシングの設定は、運用データベースに固有です。Sybase Central の**データベース・トレーシング・ウィザード**を使用すると、トレーシング・セッションを作成するときに診断トレーシングの設定を変更できます。**データベース・トレーシング・ウィザード**を起動する方法については、「診断トレーシング・セッションの作成」 218 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診断トレーシングの設定は、**アプリケーション・プロファイリング・ウィザード**の設定または動作に影響しません。**アプリケーション・プロファイリング・ウィザード**の設定は事前に設定され、変更できません。

sa_set_tracing_level システム・プロシージャを使用して診断トレーシング・レベルを変更することもできます。この方法では、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。

♦ **診断トレーシング・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 権限または PROFILE 権限のあるユーザ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診断トレーシング・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診断トレーシング・レベルを 1 に設定します。

```
CALL sa_set_tracing_level( 1 );
```

既存の設定は、診断トレーシング・レベル 1 に関連付けられているデフォルトの設定で上書きされます。各診断トレーシング・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」 208 ページを参照してください。

トレーシング・セッション実行中の診断トレーシングの設定の変更

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診断トレーシングの設定を変更できます。

♦ トレーシング・セッション中に診断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 権限または PROFILE 権限のあるユーザとしてデータベースに接続します。
2. 左ウィンドウ枠でデータベースを右クリックして **[トレーシング] - [トレーシング・レベルの変更]** を選択します。
3. 新しいトレーシング・レベルを追加するか、既存のトレーシング・レベルを削除します。
4. **[OK]** をクリックします。

診断トレーシング・セッションの作成

診断トレーシング・セッションを開始するときに、実行するトレーシングのタイプも設定し、トレーシング・データの格納場所を指定します。トレーシング・セッションは、明示的に停止を要求するまで続行されます。

トレーシング・セッションを開始するには、トレーシング・データベースと運用データベースが稼働しているデータベース・サーバで TCP/IP が実行されている必要があります。「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。

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。

♦ 診断トレーシング・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. DBA 権限または PROFILE 権限のあるユーザとしてデータベースに接続します。
2. データベースを右クリックして、**[トレーシング]** を選択します。
3. **[次へ]** をクリックします。
4. **[トレーシング詳細レベル]** ページで、トレーシングのレベルを選択します。
5. **[トレーシング・レベルの編集]** ページで、診断トレーシングの設定をカスタマイズします。
6. **[外部データベースの作成]** ページで、次の操作を行います。
   - **[新しいトレーシング・データベースを作成]** を選択します。

- データベースを保存するロケーションを選択します。
- **[ユーザ名]** と **[パスワード]** フィールドに入力します。
- **[現在のサーバでデータベースを起動]** を選択します。
- **[データベースの作成]** をクリックします。

7. **[トレースの開始]** ページで、次の操作を行います。
   - **[外部データベースにトレーシング・データを保存]** を選択します。
   - **[ユーザ名]** と **[パスワード]** フィールドに入力します。運用データベースに接続するために使用したユーザ名とパスワードを指定します。
   - **[その他の接続パラメータ]** フィールドに、部分的な接続文字列として、データベース・サーバとデータベース名を入力します。たとえば ENG=Server47;DBN=TracingDB のように指定します。

     注意
     外部データベースの場合、接続文字列でサポートされるのは、DBN、DBF、ENG、DBKEY、LINKS (CommLinks) だけです。

   - **[格納するトレーシング・データの量を制限するかどうかを指定してください。]** リストで、オプションを選択します。
8. **[完了]** をクリックします。
9. 診断トレーシング・データの収集を終了したら、データベースを右クリックし、**[トレーシング]** - **[トレーシングを停止して保存]** を選択します。

♦ 診断トレーシング・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 または PROFILE 権限のあるユーザ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」 191 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診断トレーシングを開始し、トレーシング・データを別個のデータベースに格納し、格納するデータ量を 2 時間に制限する方法を示します。この例はすべて 1 行で記述します。

```
ATTACH TRACING TO
'UID=DBA;PWD=sql;ENG=dbsrv11;DBN=tracing;LINKS=tcpip' LIMIT HISTORY 2 HOURS;
```

この例は、現在のデータベースの診断トレーシングを開始し、トレーシング・データをローカル・データベースに格納し、格納するデータ量を 2 MB に制限する方法を示します。

```
ATTACH TRACING TO LOCAL DATABASE LIMIT SIZE 2 MB;
```

この例は、診断トレーシングを停止し、トレーシング・セッション中に取得された診断データを保存する方法を示します。

```
DETACH TRACING WITH SAVE;
```

この例は、診断トレーシングを停止し、診断データを保存しない方法を示します。

```
DETACH TRACING WITHOUT SAVE;
```

参照

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

診断トレーシング情報の分析

診断トレーシングのデータは、データベース・サーバで発生し、診断トレーシング・レベルとトレーシング・セッションの設定に対応するすべてのアクティビティの記録です。データを分析するときは、どのような設定だったかを考慮する必要があります。たとえば、予想していた文がトレーシング・セッションになかった場合、文が実行されなかった可能性がありますが、高コストの文をトレースするように指定した条件を満たさなかった可能性もあります。

さまざまな理由より、データベース・サーバで実行されているアクティビティを詳細に調べることができます。パフォーマンスの問題のトラブルシューティング、リソースの使用量の予測による今後の負荷の計画、アプリケーションの論理のデバッグなどです。

参照

- 「アプリケーション・プロファイリングのチュートリアル」 279 ページ

パフォーマンスの問題のトラブルシューティング

アプリケーション・プロファイリング機能を使用すると、パフォーマンス上の問題を引き起こしている原因を次の中から特定できます。

- アプリケーションによる処理時間が長い
- 不適切なクエリ・プラン
- CPU やディスク I/O などの共有ハードウェア・リソースの競合
- データベース・オブジェクトの競合
- 不適切なデータベース設計

低いデータベース・パフォーマンスに関するトラブルシューティングを行う場合、一番の原因がアプリケーションにあるのか、データベース・サーバにあるのかをまず判断します。クライアン

ト・アプリケーションがどれだけの処理時間を消費しているかを確認するには、アプリケーション・プロファイリング・ツールの **[詳細]** タブで、1 つの接続で結果をフィルタします。その接続からの各要求に間隔がある場合は、速度が下がった一番の原因はアプリケーション・クライアントにあります。

データベース・サーバがパフォーマンスに影響している場合は、その原因を特定する必要があります。

参照

- 「アプリケーション・プロファイリングのチュートリアル」 279 ページ

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

通常、データベースの負荷が増えると、CPU サイクル、メモリ容量、ディスク I/O 帯域幅によってパフォーマンスが制限されます。非効率的なアプリケーションやデータベース・サーバが原因となっている可能性があります。効率の悪い箇所が見つからなかった場合は、ハードウェア・リソースを追加する必要がある場合があります。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」 220 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解決されず、コンピュータのパフォーマンスが改善されない場合があります。たとえば、データベース・サーバが割り当てられた CPU を完全に使用していいる場合は、CPU リソースの追加が必要なことを示している可能性があります。ただし、データベース・サーバが使用できる CPU を倍増しても、実行できる処理量が 2 倍になるとはかぎりません。

[アプリケーション・プロファイリングの詳細] 領域の **[統計情報]** タブを使用して、ハードウェア・リソースがパフォーマンスを制限している要因であるかどうかを判断できます。

- **CPU が制限要因であるかどうかの判断**　CPU が制限要因であるかどうかを判断するには、ProcessCPU 統計を確認します。この統計がグラフに表示されていない場合は、**[統計の追加]** 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 ～ 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」 237 ページ
- 「アプリケーション・プロファイリングのチュートリアル」 279 ページ
- 「パフォーマンスの問題のトラブルシューティング」 220 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。

プロシージャやトリガなどの内部データベース・オブジェクトには、Sybase Central のデバッガを使用できます。ただし、データベース・サーバで、特定のプロシージャによって実行される文をすべてトレースするようにして、アプリケーション・プロファイリング・ツールを使用してこれらの文を確認した方が効果的である場合もあります。たとえば、特定のストアド・プロシージャが、呼び出し 1000 回のうち 1 回、不正な結果を返すが、失敗する条件がわからないとします。この場合、デバッガでプロシージャのコードを 1000 回実行しないで、このプロシージャの診断トレーシングをオンにしてアプリケーションを実行できます。次に、データベース・サーバで実行された一連の文を確認し、プロシージャの不正な実行に対応する一連の文を見つけ、プロシージャが失敗した理由または予想外の動作をする条件を特定できます。プロシージャが予想外の動作をする条件がわかれば、プロシージャにブレークポイントを設定し、デバッガで詳細に調べることができます。「プロシージャ、関数、トリガ、イベントのデバッグ」 927 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 権限または PROFILE 権限のあるユーザ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]** をクリックします。
3. データベースを右クリックして **[トレーシング]** を選択するか、**[トレーシング]** - **[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**の指示に従います。
5. トレーシング・データの収集を終了したら、データベースを右クリックし、**[トレーシング]** - **[トレーシングを停止して保存]** を選択します。
6. **[アプリケーション・プロファイリングの詳細]** ウィンドウ枠で、**[分析ファイルを開くかトレーシング・データベースに接続します。]** をクリックします。
7. **[トレーシング・データベース内]** を選択し、**[開く]** をクリックします。
8. **[ユーザ名]** と **[パスワード]** フィールドに入力し、**[OK]** をクリックします。
9. **[アプリケーション・プロファイリングの詳細]** ウィンドウ枠で、**[ロギング・セッション ID]** リストの最後のエントリを選択します。
10. **[アプリケーション・プロファイリングの詳細]** ウィンドウ枠の最下部にある **[データベース・トレーシング・データ]** タブをクリックします。

    タブを選択すると、分析のために収集されたデータをさまざまなビューで表示できます。たとえば、**[概要]** タブには、トレーシング・セッション中にデータベースに対して実行されたすべての要求が表示されます。これには、要求の実行回数、実行の継続時間、要求を実行したユーザなどが含まれます。リストが長く、特定の要求を探している場合は、**[概要]** タブの **[フィルタリング]** タイトル・バーをクリックし、**[SQL 文に含まれる内容]** フィールドに文字列を入力します。

    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。**[詳細]** タブが表示されます。要求を含むローを右クリックすると、その他の SQL 文、接続、ブロックの詳細など、さらに表示する情報を選択できます。

外部トレーシング・データベースの作成

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、データベースで一定時間に 10 以上の接続が処理される場合は、パフォーマンスへの影響を抑えるために、トレーシング・データを外部トレーシング・データベースに格納することをおすすめします。

トレーシング・セッションを開始する場合は、**データベース・トレーシング・ウィザード**を使用して外部トレーシング・データベースを作成します。**データベース・トレーシング・ウィザード**により、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。トレーシング・データベースに、後で実行するトレーシング・セッションのデータを格納できます。トレーシング・セッションの作成については、「診断トレーシング・セッションの作成」 218 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・セッションのないトレーシング・データベースを手動で作成できます。

♦ アンロード・ユーティリティ (dbunload) を使用して外部トレーシング・データベースを作成するには、次の手順に従います。

1. DBA 権限または PROFILE 権限のあるユーザ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=demo;DBN=demo" -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・データベースの *demo.db*) から新しいトレーシング・データベースにスキーマがアンロードされます。-k オプションによって、アプリケーション・プロファイリング・ツールでトレーシング・データの分析に使用される情報がトレーシング・データベースに格納されます。
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』
- 「PROFILE 権限」 『SQL Anywhere サーバ - データベース管理』

その他の診断ツールと方法

アプリケーション・プロファイリングと診断トレーシングのほかにも、SQL Anywhere データベースの現在のパフォーマンスを分析し、モニタリングするのに役立つさまざまな診断ツールと方法が用意されています。

要求ロギング

要求ロギングは、アプリケーションから受け取った要求と、アプリケーションに送られた応答のログを個別に記録します。データベース・サーバがアプリケーションに何を要求されているかを特定したい場合に最も役立ちます。

特定のアプリケーションのパフォーマンスを分析するときに、データベース・サーバとクライアントのどちらに原因があるか不明な場合は、要求ロギングから始めることもおすすめします。要求ロギングは、問題の原因となっている可能性の高い、データベース・サーバに対する特定の要求を特定するためにも使用できます。

注意
要求ロギング機能で提供されるすべての機能とデータは、診断トレーシングを使用した場合も利用できます。診断トレーシングでは、それ以外の機能やデータも提供されます。「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」 205 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』

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。「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」『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。

♦ **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」 193 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

♦ **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 権限または PROFILE 権限のあるユーザ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 または PROFILE 権限のあるユーザ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。

参照

- 「sa_server_option システム・プロシージャ」『SQL Anywhere サーバ - SQL リファレンス』

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。

次の項の説明は、DBA または PROFILE 権限のあるユーザ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' );
  ```

参照

- 「sa_server_option システム・プロシージャ」『SQL Anywhere サーバ - SQL リファレンス』

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 **[プロファイリング結果]** 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 **[プロファイリング結果]** 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。

♦ **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' );
  ```

参照

- 「sa_server_option システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』

システム・プロシージャ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。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」 227 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。

♦ 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. DBA 権限または PROFILE 権限のあるユーザとしてデータベースに接続します。
2. 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;
   ```
3. **[SQL] - [実行]** を選択します。

   データベース内のすべてのプロシージャ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. DBA 権限または PROFILE 権限のあるユーザとしてデータベースに接続します。
2. 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;
   ```
3. **[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。

参照

- 「sa_procedure_profile_summary システム・プロシージャ」『SQL Anywhere サーバ - SQL リファレンス』
- 「sa_procedure_profile システム・プロシージャ」『SQL Anywhere サーバ - SQL リファレンス』
- 「sa_server_option システム・プロシージャ」『SQL Anywhere サーバ - SQL リファレンス』

タイミング・ユーティリティ

Fetchtst、Instest、Trantest など、いくつかのパフォーマンス・テスト・ユーティリティが、*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。

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。

- **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」 232 ページを参照してください。
- **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」 234 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」 235 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。

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」 205 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」 237 ページを参照してください。

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。

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。「Sybase Central パフォーマンス・モニタを使用したモニタリング」 234 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詳細な情報を追跡するのに役立ちます。Sybase Central パフォーマンス・モニタでは、接続できる任意の SQL Anywhere データベース・サーバの統計値をグラフ表示できます。

次に、Sybase Central パフォーマンス・モニタの機能を示します。

- リアルタイム更新 (調整可能な間隔で)
- 色分けされたサイズ変更可能な凡例
- 設定可能な表示プロパティ

Sybase Central パフォーマンス・モニタでは、データベースに問い合わせて統計値が収集されます。このため、キャッシュ読み込み／秒などの統計値に影響する可能性があります。代わりに Windows パフォーマンス・モニタを使用すると、統計値がモニタの影響を受けません。「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」 235 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」 237 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・モニタのグラフから統計値を追加および削除する方法については、「統計値の追加と削除」 235 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」 235 ページ
- 「統計値の追加と削除」 235 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. 左ウィンドウ枠で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. モニタされていない統計値を右クリックし、**[パフォーマンス・モニタに追加]** 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. 左ウィンドウ枠で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. 現在モニタされている統計値を右クリックし、**[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・ウィンドウにある [パフォーマンス・モニタ] からもできます。

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」 237 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」 234 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」 235 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」 237 ページを参照してください。

Windows パフォーマンス・モニタで使用されるメモリを制御するデータベース・サーバを起動する際には、データベース・サーバのオプション、およびパフォーマンス・モニタでモニタできる接続の最大数やデータベースを指定できます。次の項を参照してください。

- 「-ks サーバ・オプション」『SQL Anywhere サーバ - データベース管理』
- 「-ksc サーバ・オプション」『SQL Anywhere サーバ - データベース管理』
- 「-ksd サーバ・オプション」『SQL Anywhere サーバ - データベース管理』

次に、Windows XP で Windows パフォーマンス・モニタを起動する方法について説明します。他のバージョンの Windows で Windows パフォーマンス・モニタを起動する方法については、使用する Windows オペレーティング・システムのマニュアルを参照してください。

♦ Windows XP で 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]** を選択します。
2. ツールバーのプラス記号ツール (+) をクリックします。
3. **[パフォーマンス オブジェクト]** リストから、次のいずれかを選択します。
   - **SQL Anywhere 11 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 11 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 11 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]** 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 11 接続] または [SQL Anywhere 11 データベース] を選択した場合は、**[インスタンス]** ボックスから、モニタするデータベース接続またはデータベースを選択します。
6. 選択したカウンタの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. 表示するカウンタをすべて選択したら **[閉じる]** をクリックします。

パフォーマンス・モニタの統計値

SQL Anywhere で提供される統計値を次に示します。

処理速度は毎秒レポートされます。

キャッシュの統計値

これらの統計値により、キャッシュの使用状態を分析できます。

統計情報	スコープ	説明
キャッシュ・ヒット／秒	接続とデータベース	ルックアップの対象となるデータベース・ページがキャッシュ内で検出された頻度を示します。
キャッシュ読み込み：インデックス内部／秒	接続とデータベース	インデックスの内部ノードのページがキャッシュから読み込まれる頻度を示します。
キャッシュ読み込み：インデックス・リーフ／秒	接続とデータベース	インデックス・リーフ・ページがキャッシュから読み込まれる頻度を示します。

統計情報	スコープ	説明
キャッシュ読み込み：テーブル／秒	接続とデータベース	テーブル・ページがキャッシュから読み込まれる頻度を示します。
キャッシュ読み込み：合計ページ数／秒	接続とデータベース	データベースのページをキャッシュで検索する頻度を示します。
キャッシュ読み込み：ワーク・テーブル	接続とデータベース	テーブル・ページがキャッシュから読み込まれる頻度を示します。
キャッシュ置換：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。
キャッシュ・サイズ：現在の値	サーバ	データベース・サーバ・キャッシュの現在のサイズ (キロバイト) を示します。
キャッシュ・サイズ：最大値	サーバ	データベース・サーバ・キャッシュの許容最大サイズ (キロバイト) を示します。
キャッシュ・サイズ：最小値	サーバ	データベース・サーバ・キャッシュの許容最小サイズ (キロバイト) を示します。
キャッシュ・サイズ：ピーク値	サーバ	データベース・サーバ・キャッシュのピーク時のサイズ (キロバイト) を示します。

チェックポイントとリカバリの統計値

これらの統計値により、データベースがアイドル状態のときに行われたチェックポイントとリカバリ動作を分析できます。

統計情報	スコープ	説明
チェックポイント・フラッシュ／秒	データベース	チェックポイントの間に隣接ページを書き出す頻度を示します。
チェックポイントの緊急度	データベース	チェックポイントの緊急度 (パーセント) を示します。
チェックポイント／秒	データベース	チェックポイントを実行する頻度を示します。
チェックポイント・ログ：ビットマップ・サイズ	データベース	チェックポイント・ログのビットマップのサイズを示します。
チェックポイント・ログ：ディスクへのコミット／秒	データベース	チェックポイント・ログの commit_to_disk 操作の実行頻度を示します。
チェックポイント・ログ：ログ・サイズ	データベース	チェックポイント・ログのサイズ (ページ数) を示します。
チェックポイント・ログ：保存ページ・イメージ／秒	データベース	変更前にページがチェックポイント・ログに保存されている頻度を示します。
チェックポイント・ログ：使用ページ数	データベース	チェックポイント・ログ内で現在使用中のページ数を示します。
チェックポイント・ログ：ページ再配置／秒	データベース	チェックポイント・ログ内のページが再配置されている頻度を示します。

統計情報	スコープ	説明
チェックポイント・ログ：プレイメージ保存／秒	データベース	新しいデータベース・ページのプレイメージがチェックポイント・ログに追加されている頻度を示します。
チェックポイント・ログ：ページ書き込み／秒	データベース	ページがチェックポイント・ログに書き込まれている頻度を示します。
チェックポイント・ログ：書き込み／秒	データベース	チェックポイント・ログでディスク書き込みが行われている頻度を示します。1 回の書き込みに複数のページが含まれる場合があります。
チェックポイント・ログ：ビットマップへの書き込み／秒	データベース	チェックポイント・ログでビットマップ・ページのディスク書き込みが行われている頻度を示します。
アイドル・アクティブ／秒	データベース	データベース・サーバのアイドル・スレッドがアクティブになって、アイドル書き込み、アイドル・チェックポイントなどを行う頻度を示します。
アイドル・チェックポイント時間	データベース	アイドル・チェックポイントに費やされた合計時間 (秒) を示します。
アイドル・チェックポイント／秒	データベース	チェックポイントがデータベース・サーバのアイドル・スレッドによって最後まで行われる頻度を示します。アイドル・スレッドが最後のダーティ・ページをキャッシュに書き出すたびに、アイドル・チェックポイントが発生します。
アイドル書き込み／秒	データベース	データベース・サーバのアイドル・スレッドによってディスク書き込みが発行される頻度を示します。
リカバリ I/O 予測	データベース	データベースのリカバリに必要な I/O 操作の推定回数を示します。

統計情報	スコープ	説明
リカバリの緊急度	データベース	リカバリの緊急度 (パーセント) を示します。

通信の統計値

これらの統計値により、クライアント／サーバ間の通信アクティビティを分析できます。

統計情報	スコープ	説明
通信：受信バイト数／秒	接続とサーバ	ネットワーク・データが受信される速度 (バイト単位) を示します。
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します。
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します。
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します。
通信：空きバッファ	サーバ	空きネットワーク・バッファの数を示します。
通信：受信されるマルチパケット数／秒	サーバ	マルチパケット・デリバリの受信速度を示します。
通信：送信されるマルチパケット数／秒	サーバ	マルチパケット・デリバリの送信速度を示します。

統計情報	スコープ	説明
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します。
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します。
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します。
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します。
通信：リモートプット待ち／秒	サーバ	情報の送信に使用できるバッファがないため、通信リンクが待機する必要がある頻度を示します。この統計は TCP/IP の場合にのみ収集されます。
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・トリップの数を示します。通信：受信パケット数の統計値とは異なり、マルチパケット要求を 1 つの要求として数え、活性パケットを計数の対象から除外します。
通信：失敗した送信／秒	サーバ	基本のプロトコルがパケット送信に失敗した頻度を示します。
通信：総バッファ数	サーバ	ネットワーク・バッファの総数を示します。
通信：ユニークなクライアント・アドレス	サーバ	データベース・サーバに接続しているユニークなクライアント・ネットワーク・アドレスの数を示します。通常は接続しているクライアント・マシン数であり、接続の合計数よりも少ない場合があります。

ディスク I/O の統計値

これらの統計値により、ディスクへのアクセス (読み込みと書き込み) 状況を取得し、ディスク I/O に使用されたアクティビティの負荷について全体的な情報を把握できます。

統計情報	スコープ	説明
ディスク：アクティブ I/O	データベース	データベース・サーバによって発行され、まだ完了していない現在のファイル I/O 数を示します。
ディスク：アクティブ I/O の最大値	データベース	[ディスク：アクティブ I/O] が到達した最大値を示します。

ディスク読み込みの統計値

これらの統計値により、ディスクから情報を読み込むのに使用されたアクティビティの負荷とそのタイプを分析できます。

統計情報	スコープ	説明
ディスク読み込み：合計ページ数／秒	接続とデータベース	ページがファイルから読み込まれる速度を示します。
ディスク読み込み：アクティブ	データベース	データベース・サーバによって発行され、まだ完了していない現在のファイル読み込み数を示します。
ディスク読み込み：インデックス内部／秒	接続とデータベース	インデックス内部ノードのページがディスクから読み込まれる頻度を示します。

統計情報	スコープ	説明
ディスク読み込み：インデックス・リーフ／秒	接続とデータベース	インデックス・リーフ・ページがディスクから読み込まれる頻度を示します。
ディスク読み込み：テーブル／秒	接続とデータベース	テーブル・ページがディスクから読み込まれる頻度を示します。
ディスク読み込み：アクティブの最大値	データベース	[ディスク読み込み：アクティブ] が到達した最大値を示します。
ディスク読み込み：ワーク・テーブル	接続とデータベース	ワーク・テーブル・ページがディスクから読み込まれる頻度を示します。

ディスク書き込みの統計値

これらの統計値により、ディスクへ情報を書き込むのに使用されたアクティビティの負荷とそのタイプを分析できます。

統計情報	スコープ	説明
ディスク書き込み：アクティブ	データベース	データベース・サーバによって発行され、まだ完了していない現在のファイル書き込み数を示します。
ディスク書き込み：アクティブの最大値	データベース	[ディスク書き込み：アクティブ] が到達した最大値を示します。

統計情報	スコープ	説明
ディスク書き込み：コミット・ファイル／秒	データベース	データベース・サーバが強制的に行うディスク・キャッシュのフラッシュの頻度を示します。Windows プラットフォームではバッファなし (ダイレクト) の IO が使われるため、フラッシュは不要です。
ディスク書き込み：データベース拡張／秒	データベース	データベース・ファイルの拡張頻度 (ページ／秒) を示します。
ディスク書き込み：テンポラリ拡張／秒	データベース	テンポラリ・ファイルの拡張頻度 (ページ／秒) を示します。
ディスク書き込み：ページ／秒	接続とデータベース	修正されたページがディスクに書き込まれる頻度を示します。
ディスク書き込み：トランザクション・ログ／秒	接続とデータベース	ページをトランザクション・ログに書き込む頻度を示します。
トランザクション・ログ・グループのコミット／秒	接続とデータベース	トランザクション・ログのコミットが要求されたときすでにログが書き込まれている (コミットはいつでも可能) 頻度を示します。

インデックスの統計値

これらの統計値により、インデックスの使用状態を分析できます。

統計情報	スコープ	説明
インデックス：追加／秒	接続とデータベース	インデックスにエントリが追加される頻度を示します。
インデックス：ルックアップ／秒	接続とデータベース	インデックスでエントリを検索する頻度を示します。
インデックス：完全比較／秒	接続とデータベース	インデックスのハッシュ値を超える比較が必要となる頻度を示します。

メモリ診断の統計値

これらの統計値により、データベース・サーバによるメモリの使用状況を分析できます。

統計情報	スコープ	説明
キャッシュ：マルチページ割り当て	サーバ	マルチページ割り当ての数を示します。
キャッシュ：パニック	サーバ	キャッシュ・マネージャが割り付けるページの検索に失敗した回数を示します。
キャッシュ：スカベンジ・アクセス	サーバ	割り付けるページのスカベンジ中にアクセスしたページの数を示します。
キャッシュ：スカベンジ	サーバ	キャッシュ・マネージャが割り付けるページをスカベンジした回数を示します。
キャッシュ・ページ：割り当て構造体	サーバ	データベース・サーバのデータ構造に割り付けられたキャッシュ・ページの数を示します。
キャッシュ・ページ：ファイル	サーバ	データベース・ファイルから取得したデータを格納するキャッシュ・ページの数を示します。
キャッシュ・ページ：ファイル・ダーティ	サーバ	ダーティな (書き込みが必要な) キャッシュ・ページの数を示します。

統計情報	スコープ	説明
キャッシュ・ページ：空き	サーバ	未使用のキャッシュ・ページ数を示します。
キャッシュ・ページ：固定	サーバ	現在再利用できないページ数を示します。
キャッシュ置換：合計ページ数／秒	サーバ	必要な別のページの領域を確保するためにデータベース・ページが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ピング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。

統計情報	スコープ	説明
クエリ・メモリ：現在アクティブ	接続とサーバ	クエリ・メモリをアクティブに使用する現在の要求数を示します。
クエリ・メモリ：予測アクティブ	サーバ	安定した状態のデータベース・サーバで、クエリ・メモリをアクティブに使用する要求の推定平均数を示します。
クエリ・メモリ：追加利用可能	サーバ	基本的なメモリ集約型の付与を超えて付与可能なメモリ量を示します。
クエリ・メモリ：付与失敗数	接続とサーバ	要求がクエリ・メモリ待ちになり、取得できなかった合計回数を示します。
クエリ・メモリ：付与要求数	接続とサーバ	要求がクエリ・メモリを取得しようとした合計回数を示します。
クエリ・メモリ：付与待機数	接続とサーバ	要求がメモリ待ちになった合計回数を示します。
クエリ・メモリ：付与されたページ	接続とサーバ	要求に現在付与されているページ数を示します。
クエリ・メモリ：待機中の要求数	接続とサーバ	クエリ・メモリ待ちになっている現在の要求数を示します。

メモリ・ページの統計値

これらの統計値により、データベース・サーバが使用しているメモリ量とその目的を分析できます。

統計情報	スコープ	説明
メモリ・ページ：ロック・テーブル	データベース	ロック情報の保持に使用されているページの数を示します。
メモリ・ページ：ロック・ヒープ	サーバ	キャッシュ内でロックされているヒープ・ページの数を示します。
メモリ・ページ：メイン・ヒープ	サーバ	グローバル・データベース・サーバのデータ構造に使用されているページ数を示します。
メモリ・ページ：マップ・ページ	データベース	ロック・テーブル、頻度テーブル、テーブル・レイアウトへのアクセスに使用されたマップ・ページの数を示します。
メモリ・ページ：プロシージャ定義	データベース	プロシージャに使用された再配置可能なヒープ・ページ数を示します。
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。

統計情報	スコープ	説明
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。
タスク：交換	サーバ	クエリの並列実行に現在使用されているデータベース・サーバのスレッド数を示します。
要求：未スケジュール	サーバ	使用できるデータベース・サーバのスレッドの解放を待ってキューイングされている要求の数を示します。
スナップショット数	接続とデータベース	アクティブなスナップショットの数を示します。
文キャッシュ・ヒット	接続とサーバ	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。

統計情報	スコープ	説明
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	接続とデータベース	データベース・サーバにより文の準備が処理される頻度を示します。
文	接続	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。

統計情報	スコープ	説明
クエリ：メモリ不足時方式	接続とデータベース	メモリ不足状態のため、データベース・サーバがその実行中に実行プランを変更した回数を示します。
クエリ：ローの実体化／秒	接続とデータベース	クエリ処理中にローがワーク・テーブルに書き込まれる割合を示します。
要求：GET DATA／秒	接続とデータベース	接続で GET DATA 要求が発行されている頻度を示します。
テンポラリ・テーブル・ページ	接続とデータベース	テンポラリ・テーブルで使用されるテンポラリ・ファイルのページ数を示します。
バージョン・ストア・ページ	データベース	スナップショット・アイソレーションが有効な場合にロー・バージョン・ストアで現在使用されているテンポラリ・ファイルのページ数を示します。

パフォーマンス向上のためのヒント

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、ディスクについて次の最低要件が満たされていることを確認してください。

- 4 MB 以上のメモリ。Sybase Central や Interactive SQL などの管理ツールを使用している場合、48 MB 以上の RAM が必要です。
- 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。

サーバがハードウェアの最低要件しか満たしていない場合は、パフォーマンスが低下することがあります。その場合は、ハードウェアをアップグレードする必要があります。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

トランザクション・ログを使用すると、データを保護できます。また、SQL Anywhere のパフォーマンスを大幅に向上できます。

トランザクション・ログなしで操作すると、SQL Anywhere は各トランザクションの終わりにチェックポイントを実行します。これにより、大量のリソースが消費されます。

トランザクション・ログを使用して操作すると、SQL Anywhere は変更が発生したときにその詳細を示すメモを書き込むだけです。新しいデータベース・ページすべてを、最も効率のいい時に一度に書き込むように選択できます。「チェックポイント」は、情報がデータベース・ファイルに入力され、矛盾せず、最新のものであるということを確実にします。

トランザクション・ログを、プライマリ・データベース・ファイルとは別の物理デバイスに入れることで、パフォーマンスをさらに改善できます。追加されたドライブ・ヘッドは、通常、トランザクション・ログの終わりを探す必要はありません。

同時実行性の問題点の確認

データベース・サーバがトランザクションを処理するとき、テーブル・ローを 1 つまたは複数ロックできます。ロックは他のトランザクションが同時にアクセスすることを防止し、データ

ベースに格納されている情報の信頼性を維持します。また、更新中の情報を識別して、結果クエリの精度を高めます。

データベース・サーバは自動的にこれらのロックを設定するので、明示的な指示は必要ありません。データベース・サーバは、トランザクションが獲得したすべてのロックを、そのトランザクションが完了するまで保持します。ローにアクセスしているトランザクションは、ロックを保持しているといえます。ロックの種類により、他のトランザクションのそのローへのアクセスは限定されるか、まったくできなくなります。

頻繁に 1 つまたは複数のローに複数のユーザが同時にアクセスすると、パフォーマンスは低下することがあります。ロックが問題になっていると考えられる場合は、sa_locks プロシージャを使用して、データベースのロックに関する情報を入手できます。「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。

オプティマイザの目標の選択

optimization_goal オプションを使用すると、SQL Anywhere において SQL 文を応答時間 (First-row) に対して最適化するか、リソースの総消費量 (All-rows) に対して最適化するかを制御できます。すなわち、クエリ処理の最適化の目的を、最初のローを迅速に返すことに設定するか、または完全な結果セットを返すコストを最小限に抑えることに設定できます。

このオプションを First-row に設定すると、SQL Anywhere は、クエリの結果の最初のローをフェッチするまでの時間を短縮するアクセス・プランを選択します。この場合、検索にかかる合計時間は長くなることがあります。また、通常オプティマイザでは、可能であれば結果の実体化を必要とするアクセス・プランは使用しないで、最初のローを返すまでの時間を短縮します。この設定では、たとえばオプティマイザは、明示的なソートの操作を必要とするアクセス・プランではなく、クエリの ORDER BY 句を満たすインデックスを使用するアクセス・プランを採用します。

オプティマイザが特定の文に使用する最適化ゴールは、次の規則に従って決まります。

- メイン・クエリ・ブロックの FROM 句で、テーブル・ヒントが FASTFIRSTROW に設定されたテーブルが存在する場合、文は First-row 最適化ゴールを使用して最適化されます。
- 文の OPTION 句で optimization_goal オプションが設定されている場合、文はその設定を使用して最適化されます。
- それ以外の場合、オプティマイザは現在の optimization_goal オプションの設定を使用します。

最適化ゴールが First-row であっても、最初のローをすぐに返すことができるプランをオプティマイザが見つけられない可能性があります。たとえば、DISTINCT、GROUP BY、または ORDER BY 句が存在するために実体化が必要な文で、必要な順序を実現するためのインデックスが存在しない場合は、All-rows 目標で最適化されます。

このオプションを All-rows (デフォルト) に設定すると、SQL Anywhere のクエリは最適化され、予測される合計検索時間が最短になるアクセス・プランを選択します。PowerBuilder

DataWindow アプリケーションなど、処理の前に結果セット全体が必要になるアプリケーションでは、optimization_goal を All-rows に設定するのが適切です。

参照

- 「optimization_goal オプション [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』

次の SQL 文の OPTION 句も参照してください。

- 「DELETE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「INSERT 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「SELECT 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「MERGE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「UPDATE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「UNION 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。「テーブル制約とカラム制約の使い方」 99 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。

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。

「パフォーマンス向上のためのキャッシュの使用」 269 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。「データ整合性の確保」 85 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するツールが複数用意されています。これらのツールは、この後に示す *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*

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、従業員が複数の異なるオフィスに勤務しているとします。データベースを正規化するには、住所や代表電話番号などオフィスに関する情報を、各従業員用にコピーせず、個別のテーブルに入れておくことを考えます。

重複する情報量が少ない場合は、情報をコピーし、トリガなどの制約を使用して整合性を維持する方が良い場合があります。

データ正規化の詳細については、「SQL Anywhere でのデータベースの作成」 3 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、ローの末尾のカラムにアクセスするために、SQL Anywhere は、ロー内でそのカラムより前にあるカラムをスキップします。プライマリ・キー・カラムは、常にローの先頭に格納されます。このため、テーブル内でサイズの小さいまたは頻繁にアクセスされるカラムが、アクセス頻度の低いカラムより前に配置されるようにテーブルを作成することが重要です。

参照

- 「SQL Anywhere でのデータベースの作成」 3 ページ
- 「プライマリ・キーの管理」 25 ページ

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、データベースのパフォーマンスが向上する場合があります。たとえば、1 つのディスク・ドライブがキャッシュとデータベース・ページとの相互スワップを実行中の場合も、別のデバイスはログ・ファイルに書き込みができます。

このようなパフォーマンスの向上を得るためには、各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。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、操作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。テンポラリ・ファイルのロケーションを指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。

コンピュータに十分な数の高速デバイスがある場合、これらのファイルをそれぞれ別のデバイスに入れることで、パフォーマンスをさらに改善できます。データベースを複数の DB 領域に分割し、別のデバイスに配置することも可能です。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。

すべてのテーブルやインデックスをシステム DB 領域以外のロケーションに作成した場合、システム DB 領域は、チェックポイント・ログとシステム・テーブルの保存にのみ使用されます。この設定は、パフォーマンス上の理由からチェックポイント・ログを他のデータベース・オブジェクトとは別のディスクに保存する場合に便利です。別の DB 領域にベース・テーブルを作成するには、IN DBSPACE 句を使用するようにすべての CREATE TABLE 文を変更して、代わりの DB 領域を指定するか、テーブルを作成する前に default_dbspace オプションの設定を変更します。テンポラリ・テーブルは TEMPORARY DB 領域だけに作成できます。「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。

ベース・テーブルやテンポラリ・テーブルのデフォルト DB 領域の詳細については、「追加 DB 領域の使用」 『SQL Anywhere サーバ - データベース管理』を参照してください。

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)」 276 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。

再構築すると、データやスキーマを含むすべての情報が削除され、同一の形式ですべての情報が元に戻されます。ディスク・ドライブのデフラグメンテーションと同じように、空き領域が埋められてパフォーマンスが改善されます。また、再構築のタイミングで特定の設定を変更することもできます。「データベースの再構築」 796 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。断片化の削減は、データベースのサイズに比例して重要になります。SQL Anywhere には、ファイル、テーブル、インデックスの断片化情報を生成するストアド・プロシージャが用意されています。

パフォーマンスが大幅に低下した場合は、次の方法で対処できます。

- 複数のテーブルで多数の削除／更新／挿入アクティビティを実行した場合は、テーブルやインデックスの断片化を削減するためにデータベースを再構築する。
- データベースをディスク・パーティションに単独で入れ、ファイルの断片化を低減する。
- 利用できる Windows ユーティリティの 1 つを定期的に実行して、ファイルの断片化を低減する。
- データベースを再編成して、データベースの断片化を低減する。
- REORGANIZE TABLE 文を使用して、テーブル内のローをデフラグしたり、DELETE によって散在したインデックスを圧縮する。テーブルを再編成すると、テーブルとインデックスの格納に使用される合計ページ数を減らし、インデックス・ツリーのレベル数も減らすことができます。

ファイルの断片化の削減

断片化したデータベース・ファイルは、データベース・サーバのパフォーマンスに影響することがあります。ディスク断片化を減らすことは、データベースのサイズが大きくなるに従って重要になります。

Windows でデータベースを起動すると、データベース・サーバが各 DB 領域内でファイルの断片化数を判断します。断片化の数が 1 より大きい場合は、データベース・サーバ・メッセージ・ウィンドウに「**データベース・ファイル "mydatabase.db" は nnn のディスク・フラグメントで構成されています。**」と表示されます。

また、DBFileFragments データベース・プロパティを使用しても、データベース・ファイルの断片化の数を取得できます。「データベース・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。

テーブルの断片化削減

テーブルの断片化は、ローが連続して格納されていない場合、またはローが複数のページに分割されている場合に発生します。これらのローには追加のページ・アクセスが必要になるため、データベース・サーバのパフォーマンスを低下させます。

断片化がパフォーマンスに与える影響は、さまざまなものがあります。テーブルが極端に断片化されていても、メモリ内に収まっていて、かつアクセス方法によってページのキャッシュが許可されている場合、影響は最小限に抑えられる可能性があります。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 のコストがキャッシュによって削減されない場合は、パフォーマンスが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。この方法によって、テーブルの断片化がパフォーマンスに影響をきたすまでの状態になる速度を確認でき、それによりテーブルのデフラグを実行する最適な頻度を決定できます。

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' ] ] );
```

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とスキューの削減

インデックスは特定のカラムの検索が高速になるように設計されていますが、インデックス付きテーブルに対して多数の DELETE 操作が実行されると、インデックスが断片化 (密度が下がる) およびスキューする (偏る) ことがあります。

インデックスの密度は、インデックス・ページの平均の満杯度を反映しています。インデックス・スキューは、平均密度からの一般的な偏差を反映しています。オプティマイザが選択性推定を行う場合、このスキュー量は重要です。

データベース内に許容できないレベルの断片化またはスキューを含むインデックスがあるかどうかを判断するには、**アプリケーション・プロファイリング・ウィザード**を使用します。「アプリケーション・プロファイリング・ウィザード」 191 ページを参照してください。

sa_index_fragmentation システム・プロシージャを使用して、インデックスの断片化およびスキューのレベルを確認することもできます。たとえば、次の文は sa_index_density システム・プロシージャを呼び出して、Customers テーブルのインデックスを調べます。

```
CALL sa_index_density( 'Customers' );
```

TableName	TableId	IndexName	IndexID	IndexType	LeafPages	Density	Skew
Customers	686	CustKey	0	PKEY	1	0.645992	1.002772
Customers	686	IX_cust_name	1	NUI	1	0.789795	1.432239

SQL Anywhere はプライマリ・キーのインデックスを自動的に作成します。sa_index_density システム・プロシージャの結果で、IndexID が 0 のインデックスが存在する点に注意してください。

リーフ・ページ数が少ない場合は、密度とスキューの値を考慮する必要はありません。密度とスキューの値は、リーフ・ページ数が多いときだけ重要です。リーフ・ページ数が多い場合、密度の値が低いと断片化を表し、不均衡値が高いとインデックスが偏っていることを示します。どちらも、パフォーマンス低下の要因になります。REORGANIZE TABLE 文を実行すると、この両方の問題に対処できます。「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

- 「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』
- 「インデックスの操作」 75 ページ

プライマリ・キー幅の縮小

幅の広いプライマリ・キーは、2 つ以上のカラムで構成されます。プライマリ・キーに含まれるカラム数が多いほど、データベース・サーバに負荷がかかります。プライマリ・キーに含まれるカラム数を減らすと、パフォーマンスが改善されます。

参照

- 「プライマリ・キーの管理」 25 ページ
- 「キーを使ったクエリのパフォーマンス改善」 269 ページ

テーブル幅の縮小

結合されたカラム (または個々のローのサイズ) がデータベース・ページ・サイズを超えるため、複数のデータベース・ページに分割する必要があるテーブルは、ワイド・テーブルと呼ばれます。ローが多数のページに分割されるほど、データベース・サーバは各ローの読み込みに時間がかかります。ワイド・テーブルが存在し、パフォーマンスが低下している場合は、テーブルを正規化してカラム数を減らすことを検討してください。テーブルを正規化できない場合は、データベース・ページ・サイズを拡大するとパフォーマンスが改善されることがあります。特にほとんどのテーブルの幅が広い場合は有効です。「SQL Anywhere でのデータベースの作成」 3 ページを参照してください。

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」 882 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」 886 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」 404 ページまたは「GROUP BY 句：クエリ結果のグループへの編成」 397 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、または削除されてから実行されたクエリが少ない場合です。統計が不正確な場合、または利用できない場合、パフォーマンスに悪影響を与えます。統計の更新に SQL Anywhere が長時間要している場合、CREATE STATISTICS または DROP STATISTICS を実行して、統計を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。「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ループの後に明示的にコミットを実行してください。また、論理トランザクションが大きい場合は、ブロッキングとデッドロックが発生する可能性があることに注意してください。

トランザクション・ログ・ファイルを使用していない場合、オートコミット・モードを使用するコストは高くなります。各文の終わりで、チェックポイントが強制的に実行されます。チェックポイントとは、多数の情報ページをディスクに書き込む操作です。

各アプリケーション・インタフェースには、オートコミット動作を設定する独自の方法があります。Open Client、ODBC、JDBC インタフェースでは、オートコミットはデフォルトの動作です。

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。大きいページ・サイズにも小さいページ・サイズにも、それぞれ利点と欠点があります。

小さいページは保持できる情報量が少ないため、特に、ページの半分以上のサイズのローを挿入する場合に領域の使用効率が低下します。ただし、ページ・サイズが小さいと、同じサイズのキャッシュにより多くのページを格納できるため、SQL Anywhere をより少ないリソースで実行できます。メモリが限られている小型のコンピュータでデータベースを実行する場合に便利です。また、不特定のロケーションから少量の情報を取得するためにデータベースを使う場合にも便利です。

ページ・サイズが大きいと、SQL Anywhere はデータベースをより効率的に読み込めます。データベースの規模が大きい場合や、逐次テーブル・スキャンを実行するクエリの場合にパフォーマンスが高くなる傾向があります。通常は、ディスクの物理的な設計のために、大きなブロックを

少数取り出す方が、小さなブロックを多数取り出す場合よりも効率的です。大きいページ・サイズには、他の利点もあります。インデックスのファンアウトを改善することによってインデックス・レベル数を減らし、カラム数の多いテーブルが作成できます。

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 (かつ適切な) ページ・サイズの選択が重要です。

既存のデータベースのページ・サイズは変更できません。その代わり、新しいデータベースを作成し、dbinit の -p オプションを使用してページ・サイズを指定します。たとえば、次のコマンドは 4 KB のページ・サイズのデータベースを作成します。

```
dbinit -p 4096 new.db
```

新しいページ・サイズでデータベースを作成するために、PAGE SIZE 句を指定した CREATE DATABASE 文を使用することもできます。「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、分散読み込みは使用されません。

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 を割り当てないでください。また、コストの高い数値型や文字列型でなく、効率的なデータ型をできるだけ選択してください。「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。

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」 762 ページを参照してください。

コミットの遅延の使用

データベースに対してコミットされる変更の頻度が高い場合、トランザクション・ログの書き込みの頻度が、データベース全体のパフォーマンスを左右する最大の要因になります。トランザクション・ログのパフォーマンスを改善する場合は、delayed_commits オプションを On に設定します。On に設定すると、データベース・サーバは COMMIT のトランザクション・ログ・エントリがディスクに書き込まれるのを待たずに、直ちに COMMIT 文に応答します。Off に設定すると、アプリケーションは COMMIT がディスクに書き込まれるまで待たなければなりせん。
delayed_commits オプションをオンにすると、一部が埋まっているログ・ページの複数回の書き換えが回避され、トランザクション・ログの書き込みが減少します。このオプションは、接続ごとまたは接続全体に対して設定できます。delayed_commits オプションをオンにすると、トランザクション・ログ・ページがディスクにフラッシュされる前にサーバがダウンした場合、コミットされた操作が失われるリスクがあります。「delayed_commits 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。

イン・メモリ・モードの使用

最新のチェックポイント後にコミットされたトランザクションが失われても構わないアプリケーションには、イン・メモリ・モードを使用すると便利な場合があります。

大量のメモリを使用できる (通常、すべてのデータベース・ファイルをキャッシュ内に保持できる) システムを実行している場合、パフォーマンスの向上が求められるアプリケーションでこのモードを使用すると便利です。

イン・メモリ・モードは 2 種類から選択できます。非書き込みモードでは、コミットされたトランザクションはディスク上のデータベース・ファイルに書き込まれません。非書き込みモードを指定すると、複数の LOAD TABLE 文を同じまたは異なるテーブルで同時にアクティブにできます。データベースが停止した場合、または接続が切断された場合は、すべての変更が失われます。チェックポイント専用モードでは、データベース・サーバはトランザクション・ログを使用しないため、最新のコミットされたトランザクションにリカバリすることはできません。ただし、チェックポイント・ログは有効になっているため、データベースを最新のチェックポイントにリカバリすることができます。

イン・メモリ・モードの設定方法およびアプリケーションに対して適切か判断する方法については、「-im 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。

別途ライセンスが必要な必須コンポーネント
イン・メモリ・モードには別途ライセンスが必要です。「別途ライセンスが必要なコンポーネント」 『SQL Anywhere 11 - 紹介』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従業員を検索する必要があるのに、姓か名前のどちらかしかわからないとします。インデックスがない場合、SQL Anywhere はテーブル全体をスキャンします。しかし、姓が先に入力されているインデックスと名前が先に入力されているインデックスの 2 つのインデックスが作成されていれば、SQL Anywhere はこの 2 つのインデックスを先にスキャンするので、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、「インデックスの操作」 75 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」 199 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」 673 ページと「クラスタード・インデックスの使用」 77 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、1000000 の場合もほとんど同じ時間で行うことができます。

プライマリ・キーと外部キーについて、インデックスが個別に自動的に作成されます。これによって、SQL Anywhere で多数のオペレーションを効率的に実行できます。

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 11 - 紹介』を参照してください。

パフォーマンス向上のため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。

データベース・サーバ・メッセージ・ウィンドウには起動時のキャッシュ・サイズが表示されます。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・メモリ使用の制限

キャッシュの初期サイズ、最小サイズ、最大サイズはすべて、データベース・サーバのコマンド・ラインから制御できます。

- **初期キャッシュ・サイズ** 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 Mobile** 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。
  - **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)」 272 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づいています。Windows Mobile では、デフォルトの最大キャッシュ・サイズは、使用可能なプログラム・メモリから 4 MB を引いた値です。他の UNIX 以外のコンピュータでは、最大キャッシュ・サイズは、非 AWE の最大キャッシュ・サイズとコンピュータの物理メモリの 90% のうち、いずれか低い方になります。UNIX では、デフォルトの最大キャッシュ・サイズは次のように計算されます。
  - 32 ビットの UNIX プラットフォームでは、物理メモリ量の合計の 90% または 1,834,880 KB のいずれか小さい方です。

- 64 ビットの UNIX プラットフォームでは、物理メモリ量の合計の 90% または 8,589,672,320 KB のいずれか小さい方です。

- **最小キャッシュ・サイズ**　最小キャッシュ・サイズを制御するには、データベース・サーバの -cl サーバ・オプションを指定します。Windows Mobile を除き、デフォルトでは、最小キャッシュ・サイズは初期キャッシュ・サイズと同じです。Windows Mobile では、デフォルトの最小キャッシュ・サイズは 600 KB です。

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、「データベース・サーバ・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。

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 を使用して、データベース・キャッシュを自動的にリサイズできます。ただし、動的キャッシュ・サイズ決定の効果は、データベース・サーバが実行されているオペレーティング・システム、および使用できる物理メモリ量によって制限されます。

完全な「動的キャッシュ・サイズ決定」を行う場合、不適切なメモリの割り付けがデータベース・サーバのパフォーマンスに影響を与えることはありません。多くのキャッシュを使用することでデータベース・サーバのパフォーマンスが向上する場合は、使用可能なメモリがあるかぎりキャッシュ・サイズは大きくなり、他のアプリケーションがキャッシュ・メモリを必要とするときはキャッシュ・サイズは小さくなります。これにより、データベース・サーバがシステムの他のアプリケーションに影響を与えるのを防ぐことができます。

キャッシュの必要量は、動的キャッシュ・サイズ決定によって通常 1 分ごとに評価されます。ただし、新しいデータベースが起動されたときやファイル量が大幅に増加した際は、30 秒間は 5 秒ごとに評価されるよう実行間隔が狭まることがあります。最初の 30 秒間が経過すると、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・サイズ決定を使用する場合は、明示的にデータベース・キャッシュを設定する必要はありません。

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 Mobile 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 Mobile では、データベース・サーバがキャッシュとオペレーションの統計を 1 分ごとに評価し、最適なキャッシュ・サイズを計算します。データベース・サーバは、ターゲットのキャッシュ・サイズを計算します。このキャッシュは、現在使用されていない物理メモリの約 5 MB をシステムが使用できるように残し、それ以外をすべて使用します。ターゲット・キャッシュ・サイズが、明示的または暗黙的に指定された最小キャッシュ・サイズより小さくなることはありません。ターゲットのキャッシュ・サイズは、明示的または暗黙的に指定された最大キャッシュ・サイズ、またはすべてのオープン・データベースとテンポラリ・ファイル合計サイズにメイン・ヒープのサイズを加えたものを超えることはありません。

キャッシュ・サイズの変動を防ぐために、データベース・サーバはキャッシュ・サイズを段階的に増やします。すぐにターゲット値に調整するのではなく、調整するたびに現在のサイズとターゲット・サイズの差の 75% ずつキャッシュ・サイズを修正します。

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 Mobile 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「**メモリが不足しています。**」という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「**メモリが不足しています。**」というエラーで終了します。使用可能なメモリより大きい最大キャッシュ・サイズを指定すると、データベース・サーバはパフォーマンス低下の警告を出しますが、終了はしません。

データベース・サーバはすべての最大キャッシュ・サイズをシステム・リソースから割り付け、終了まで解放しません。他のアプリケーション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、「データベース・サーバ・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」 232 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、収集ページ数が最大数に到達するまで (値はキャッシュ・サイズとデータベース・サイズに基づく)、収集速度が最小しきい値を下回るまで、またはデータベースが停止するまで、データベース起動時に要求されたすべてのデータベース・ページを追跡し続けます。データベース・サーバは、収集最大ページ数と収集しきい値を制御します。いったん収集が完了すると、参照されたページはデータベースに記録されるので、次回のデータベース起動時にキャッシュの準備に使用できます。

キャッシュ・ウォーミング (再ロード) はデフォルトでオンになっていて、-cr データベース・サーバ・オプションで制御されます。キャッシュを準備するために、データベース・サーバはデータベースにすでに記録された収集ページがあるかどうかをチェックします。ある場合、データベース・サーバが対応するページをキャッシュにロードします。データベースはキャッシュがページをロードしている間引き続き要求を処理できますが、大量の I/O アクティビティがデータベースで検出された場合は準備が停止することがあります。この場合、キャッシュに再ロードされるページ・セットに含まれないページにアクセスするクエリのパフォーマンスの低下を防ぐため

に、キャッシュ・ウォーミングは停止します。キャッシュ・ウォーミングに関する情報をデータベース・サーバ・メッセージ・ウィンドウに表示する場合は、-cv オプションを指定できます。

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。

圧縮しきい値を指定すると、圧縮対象になるデータ・パケットの最小サイズを選択できます。圧縮しきい値の最適値は、使用するネットワークのタイプや速度など、さまざまな要素の影響を受けることがあります。

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 First-row です。optimization_goal が First-row に設定されている場合、SQL Anywhere はワーク・テーブルを使用しないようにします。All-rows に設定されている場合、クエリの合計実行コストが減少するなら、ワーク・テーブルが使用されます。

optimization_goal 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 All-rows に設定されている場合は、ワーク・テーブルを作成してローをソートするよりも、インデックスを使用してクエリのローをすべてフェッチする方が高コストになることがあります。最適化ゴールが All-rows に設定されている場合、SQL Anywhere は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。
- ハッシュ・ジョイン・アルゴリズムが選択されたとき。この場合、ワーク・テーブルが中間結果の格納 (入力がメモリに収まらない場合) とジョイン結果の格納に使用されます。
- sensitive 値を使用してカーソルが開かれたとき。この場合、ベース・テーブルのロー識別子とプライマリ・キーを入れるワーク・テーブルが作成されます。このワーク・テーブルは、クエリから前方にローがフェッチされるたびに埋められます。ただし、カーソルから最後のローをフェッチすると、テーブル全体が埋められます。
- insensitive セマンティックを使用してカーソルが開かれたとき。この場合、クエリが開かれるときに、ワーク・テーブルにクエリの結果が設定されます。
- 複数のローに UPDATE を実行し、UPDATE の WHERE 句または更新に使用するインデックスに、更新されるカラムが表示されるとき。
- 複数のローに対する UPDATE または DELETE が、修正されるテーブルを参照する WHERE 句にサブクエリを持つとき。
- SELECT 文から INSERT を実行し、その SELECT 文が挿入テーブルを参照するとき。
- 複数のローに INSERT、UPDATE、または DELETE を実行し、その操作中に起動するように、対応するトリガがテーブルに定義されているとき。

これらの場合は、操作の影響を受けるレコードがワーク・テーブルに入れられます。キーセット駆動型カーソルなど、場合によっては、ワーク・テーブルにテンポラリ・インデックスが作成されます。要求されたレコードをワーク・テーブルに抽出する操作は、クエリの結果が出るまでかなりの時間がかかります。上記の最初の例でソートを実行するのに使用できるインデックスを作成すると、最初のいくつかのローを検索する時間が短縮されます。ただし、ワーク・テーブルを使用すると、すべてのローをフェッチするための合計時間を短縮できる場合があります。これは、ハッシュとマージ・ソートに基づくクエリ・アルゴリズムが許可されるためです。これらのアルゴリズムでは、シーケンシャル I/O が使用されます。これは、インデックス・スキャンと併用されるランダム I/O より高速です。

オプティマイザは、各クエリを分析し、ワーク・テーブルを使用した場合に最適なパフォーマンスが得られるかどうかを判断します。こうした最適化を利用するのに、ユーザ側の操作は必要ありません。

注意

上述の INSERT、UPDATE、DELETE は 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。

アプリケーション・プロファイリングのチュートリアル

目次

アプリケーション・プロファイリングのチュートリアルでは、**アプリケーション・プロファイリング・ウィザード**と**データベース・トレーシング・ウィザード**を使用して、デッドロック、速度の遅い文、インデックスの断片化、テーブルの断片化、低速のプロシージャなどの一般的なパフォーマンス問題を分析する方法を学習します。

警告

チュートリアルでは、サンプル・データベース (*demo.db*) ではなく、テスト・データベース *app_profiling.db* を作成して使用します。チュートリアルを実行する場合は、サンプル・データベースは使用しないでください。

アプリケーション・プロファイリングを実行するには、PROFILE 権限が必要です。アプリケーション・プロファイリングのチュートリアルでは、DBA 権限のあるユーザとして接続します。

チュートリアル：デッドロックの診断

このチュートリアルでは、**データベース・トレーシング・ウィザード**を使用してデータベースに発生する可能性があるデッドロックを参照する方法を学びます。**データベース・トレーシング・ウィザード**を使用すると、デッドロックが発生した状態や、デッドロック発生の原因となっている接続を調査することもできます。

デッドロックは、複数のトランザクションが互いにブロックされている場合に発生します。たとえば、トランザクション A がテーブル B にアクセスする必要があるが、テーブル B は トランザクション B によってロックされ、トランザクション B はテーブル A にアクセスする必要があるが、テーブル A はトランザクション A によってロックされている場合などです。これは環状ブロッキングの競合が発生している状態です。

デッドロックが発生している状況は、SQLCODE -306 と -307 が返されることで分かります。デッドロックを解決するために、SQL Anywhere はデッドロックが作成された最後の文を自動的にロールバックします。文が絶え間なくロールバックされる場合は、パフォーマンス問題が発生しています。

レッスン 1：テスト・データベースの作成

次の手順により、サンプル・データベースのデータを使用して、テスト・データベース *app_profiling.db* を作成します。このテスト・データベースは、アプリケーション・プロファイリングのすべてのチュートリアルで使用します。

♦ **テスト・データベースを作成するには、次の手順に従います。**

1. ディレクトリ *C:¥AppProfilingTutorial* を作成します。
2. コマンド・プロンプトを開き、次のコマンドを入力してテスト・データベース *app_profiling.db* を作成します。*samples-dir* はサンプル・ディレクトリのロケーションです。

   ```
   dbunload -c "UID=DBA;PWD=sql;DBF=samples-dir¥demo.db" -an C:¥AppProfilingTutorial
   ¥app_profiling.db
   ```

   たとえば、Windows XP と SQL Anywhere がデフォルトのロケーションにインストールされているコンピュータでは、コマンドは次のようになります。

   ```
   dbunload -c "UID=DBA;PWD=sql;DBF=C:¥Documents and Settings¥All Users¥Documents¥SQL
   Anywhere 11¥Samples¥demo.db" -an C:¥AppProfilingTutorial¥app_profiling.db
   ```

   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。

ヒント
アプリケーション・プロファイリングのチュートリアルでは、トレーシング情報はプロファイリングを行う同じデータベース (*app_profiling.db*) に格納されます。ただし、負荷の大きなデータベースをプロファイリングする場合は、運用データベースのパフォーマンスへの影響を避けるため、別のデータベースにトレーシング・データを格納することを検討してください。

参照

- 「アプリケーション・プロファイリング・ウィザード」 191 ページ
- 「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」 205 ページ
- 「PROFILE 権限」 『SQL Anywhere サーバ - データベース管理』

レッスン 2：デッドロックの作成

このチュートリアルは、テスト・データベースが作成されていることを前提としています。テスト・データベースを作成していない場合は、「レッスン 1：テスト・データベースの作成」 280 ページを参照してください。

ヒント
このチュートリアルの SQL 文をコピーして Interactive SQL にペーストできます。

♦ **デッドロックを作成するには、次の手順に従います。**

1. Sybase Central を起動し、ユーザ ID に **DBA**、パスワードに **sql** を使用してテスト・データベース *app_profiling.db* に接続します。

   Sybase Central の起動とデータベースへの接続の操作に慣れていない場合は、「ローカル・データベースへの接続」 『SQL Anywhere サーバ - データベース管理』を参照してください。

2. 左ウィンドウ枠で、*app_profiling - DBA* をクリックし、**[ファイル] - [Interactive SQL を開く]** を選択します。

   Interactive SQL が起動し、*app_profiling.db* データベースに接続します。

3. Interactive SQL で、次の SQL 文を実行します。

   a. テーブルを 2 つ作成します。

   ```
   CREATE TABLE "DBA"."deadlock1" (
     "id" UNSIGNED BIGINT NOT NULL DEFAULT AUTOINCREMENT,
     "val" CHAR(1) );
   CREATE TABLE "DBA"."deadlock2" (
     "id" UNSIGNED BIGINT NOT NULL DEFAULT AUTOINCREMENT,
     "val" CHAR(1) );
   ```

   b. 各テーブルに値を挿入します。

   ```
   INSERT INTO "deadlock1"("val") VALUES('x');
   INSERT INTO "deadlock2"("val") VALUES('x');
   ```

   c. 後でデッドロックを発生させるために使用する 2 つのプロシージャを作成します。

   ```
   CREATE PROCEDURE "DBA"."proc_deadlock1"( )
     BEGIN
       LOCK TABLE "DBA"."deadlock1" IN EXCLUSIVE MODE;
       WAITFOR DELAY '00:00:20:000';
       UPDATE deadlock2 SET val='y';
     END;
   CREATE PROCEDURE "DBA"."proc_deadlock2"( )
     BEGIN
       LOCK TABLE "DBA"."deadlock2" IN EXCLUSIVE MODE;
       WAITFOR DELAY '00:00:20:000';
   ```

```
    UPDATE deadlock1 SET val='y';
END;
```

d. データベースに加えた変更をコミットします。

```
COMMIT;
```

4. Interactive SQL を終了します。

レッスン 3：デッドロック・データの取得

データベース・トレーシング・ウィザードを使用すると、診断トレーシング・セッションを作成できます。トレーシング・セッションではデッドロック・データを取得します。

♦ **デッドロック・データを取得するには、次の手順に従います。**

1. Sybase Central で、**[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]** をクリックします。

2. **データベース・トレーシング・ウィザード**を起動します。
   a. 左ウィンドウ枠で、*app_profiling - DBA* をクリックし、**[ファイル]** - **[トレーシング]** を選択します。
   b. **[ようこそ]** ページで、**[次へ]** をクリックします。
   c. **[トレーシング詳細レベル]** ページで、**[高 (短期間の集中的なモニタリングに推奨)]**を選択し、**[次へ]** をクリックします。
   d. **[トレーシング・レベルの編集]** ページで、**[次へ]** をクリックします。
   e. **[外部データベースの作成]** ページで、**[新しいデータベースを作成せず、既存のトレーシング・データベースを使用]** を選択し、**[次へ]** をクリックします。
   f. **[トレースの開始]** ページで、**[このデータベースにトレーシング・データを保存]** を選択します。
   g. 格納するトレーシング・データの量を制限しない場合は、**[制限なし]** を選択し、**[完了]** をクリックします。
   h. **[完了]** をクリックします。
3. デッドロックを作成します。
   a. Sybase Central の左ウィンドウ枠で、*app_profiling - DBA* データベースを選択し、**[ファイル]** - **[Interactive SQL を開く]** を選択します。

      Interactive SQL が起動し、*app_profiling - DBA* データベースに接続します。
   b. 前の手順を繰り返し、2 番目の Interactive SQL ウィンドウを開きます。
   c. 1 つの Interactive SQL ウィンドウで、次の SQL 文を実行します。

```
CALL "DBA"."proc_deadlock1"();
```

   d. 2 番目の Interactive SQL ウィンドウで、20 秒以内に次の SQL 文を実行します。


```
CALL "DBA"."proc_deadlock2"();
```

しばらくすると、デッドロックが検出されたことを示す **[ISQL エラー]** ウィンドウが表示されます。これは、proc_deadlock1 が deadlock2 テーブルにアクセスする必要があるが、このテーブルは proc_deadlock2 によってロックされているために発生します。また、proc_deadlock2 は deadlock1 テーブルにアクセスする必要があるが、このテーブルも proc_deadlock1 によってロックされています。

e. **[OK]** をクリックします。

4. 2 つの Interactive SQL ウィンドウを閉じます。
5. トレーシング・セッションを停止するには、Sybase Central で *app_profiling - DBA* 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。

レッスン 4：ブロックされた接続データの確認

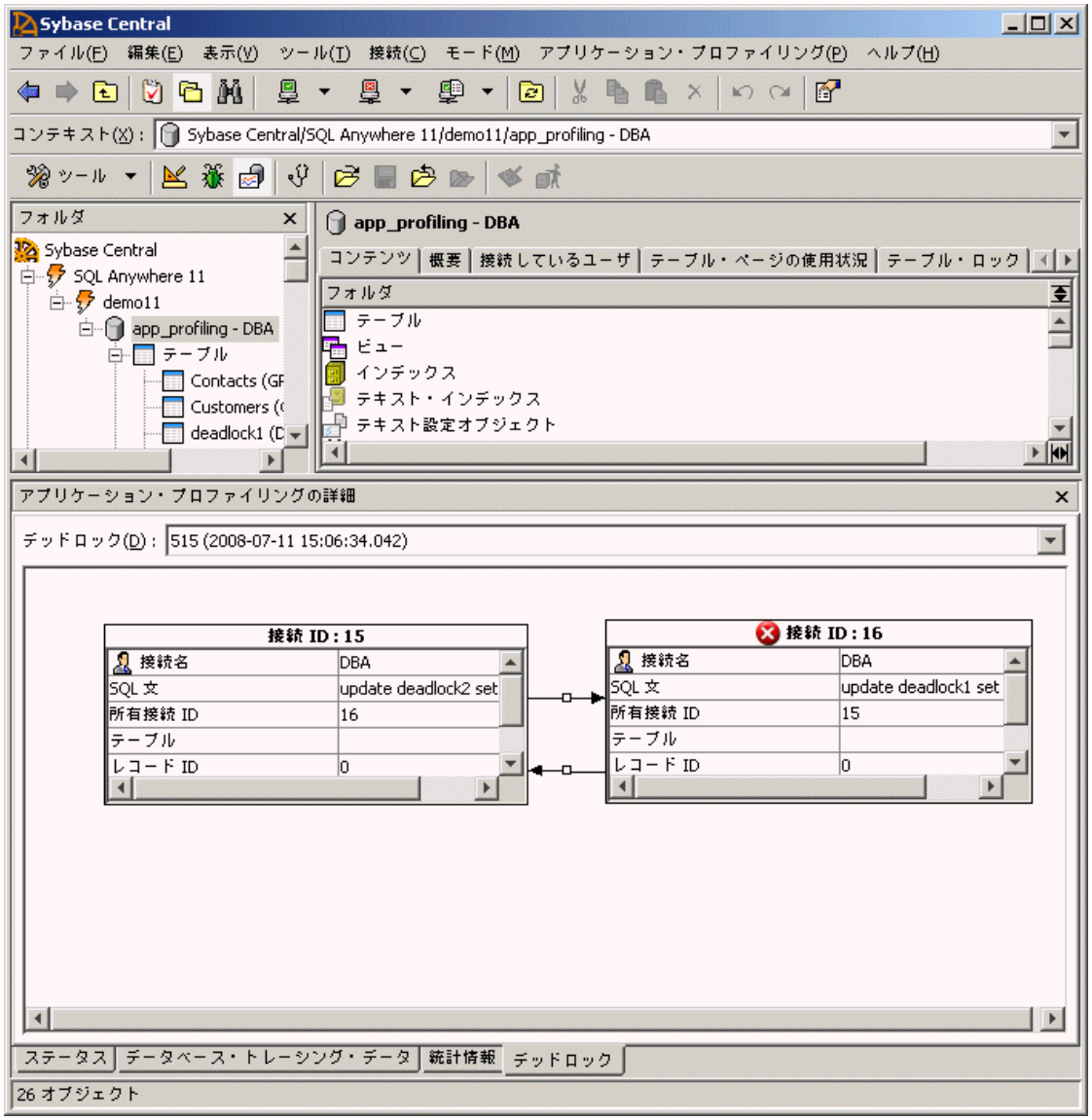
[アプリケーション・プロファイリング] モードを使用すると、デッドロックに関係する接続がグラフィックで表示されます。また、**[接続のブロック]** タブには、ブロックされた接続に関する追加情報も表示されます。

♦ **ブロックされた接続データを確認するには、次の手順に従います。**

1. トレーシング・セッション中に作成された分析ファイルを開きます。
   a. Sybase Central で、**[アプリケーション・プロファイリング] - [分析ファイルを開くかトレーシング・データベースに接続]** を選択します。
   b. **[トレーシング・データベース内]** を選択します。
   c. **[開く]** をクリックします。
   d. **[ID]** タブをクリックし、**[ユーザ ID]** フィールドに **DBA** と入力し、**[パスワード]** フィールドに **sql** と入力します。
   e. **[データベース]** タブをクリックし、**[データベース・ファイル]** フィールドで *app_profiling - DBA* を参照して選択します。
   f. **[OK]** をクリックします。
2. デッドロックのグラフィック表示を参照します。
   a. **[アプリケーション・プロファイリングの詳細]** ウィンドウ枠で、**[ステータス]** タブをクリックし、**[ロギング・セッション ID]** リストから最新の ID を選択します。

      [アプリケーション・プロファイリングの詳細] ウィンドウ枠が表示されない場合は、**[表示] - [アプリケーション・プロファイリングの詳細]** を選択します。
   b. **[アプリケーション・プロファイリングの詳細]** ウィンドウ枠の下部で、**[デッドロック]** タブをクリックします。最新のデッドロックが表示されます。別のデッドロックを参照するには、**[デッドロック]** リストをクリックします。

   次の図は、UPDATE 文により、どのようにデッドロック状態が作成されたかを示しています。



デッドロックに関連する各接続は、次のフィールドが含まれるテーブルで表示されます。

- **接続名**　このフィールドには、接続を開いたユーザ ID が表示されます。
- **SQL 文**　このフィールドには、デッドロックに関係する実際の文が表示されます。この場合、Interactive SQL の各インスタンスから実行したプロシージャで検出された UPDATE 文が原因でデッドロックが発生しました。
- **所有接続 ID**　このフィールドには、現在の接続をブロックしている接続の ID が表示されます。
- **レコード ID**　このフィールドには、現在の接続がブロックされているローの ID が表示されます。

- **ロールバック操作のカウント**　このフィールドには、デッドロックの結果としてロールバックされる必要がある操作回数が表示されます。この場合、プロシージャには UPDATE 文しか含まれていないため、この数は 0 になります。

レッスン 5：デッドロック・データの表示

次の手順により、発生頻度や存続時間など、デッドロックに関する追加データを表示します。データベース・トレーシング・セッション中に記録されたすべてのデッドロックのリストを表示するには、**[接続のブロック]** タブを使用します。

♦ **デッドロック・データを表示するには、次の手順に従います。**

1. **[アプリケーション・プロファイリングの詳細]** ウィンドウ枠で、**[データベース・トレーシング・データ]** タブをクリックします。
2. **[データベース・トレーシング・データ]** の上にある **[接続のブロック]** タブをクリックします。

   [接続のブロック] タブが開き、ブロック時間、ブロックが解除された時間、各接続に対するブロックの継続時間が表示されます。

参照

- 「トランザクションのブロックとデッドロック」 139 ページ
- 「診断トレーシング・レベルの選択」 207 ページ
- 「デッドロック」 140 ページ
- 「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」 205 ページ

チュートリアル：速度が遅い文の診断

このチュートリアルでは、**データベース・トレーシング・ウィザード**を使用して文の実行時間を表示する方法や、実行速度が遅い文を識別する方法を学習します。

速度が遅い文は、データベース・サーバがその文の処理に長時間かかる場合に発生します。長い処理時間は、データベースが正しく設計されていない、インデックスの使用方法が不適切、インデックスとテーブルの断片化、キャッシュ・サイズが小さいなど、いくつかの問題が原因で発生する可能性があります。また、文の形式が不正であったり、結果を得るためのショートカットが効率的に使用されていないために、文の実行速度が遅くなっている場合もあります。

このチュートリアルでは、各文に特別な要件がある場合があるため、速度の遅い文の書き換え方法は示していません。ただし、このチュートリアルでは、代替構文を使用してクエリを書き換えたときに、実行時間を検出する場所や、実行時間を比較する方法を示します。

参照

- 「データのクエリ」 305 ページ
- 「ジョイン：複数テーブルからのデータ検索」 413 ページ
- 「サブクエリの使用」 531 ページ

レッスン 1：診断トレーシング・セッションの作成

データベース・トレーシング・ウィザードを使用して、診断トレーシング・セッションを作成できます。トレーシング・セッションは、処理の継続時間を含む、処理中の文のデータを取得します。

このチュートリアルは、テスト・データベースが作成されていることを前提としています。テスト・データベースを作成していない場合は、「レッスン 1：テスト・データベースの作成」 280 ページを参照してください。

ヒント
このチュートリアルの SQL 文をコピーして Interactive SQL にペーストできます。

♦ **診断トレーシング・セッションを作成するには、次の手順に従います。**

1. Sybase Central を起動し、ユーザ ID に **DBA**、パスワードに **sql** を使用してテスト・データベース *app_profiling.db* に接続します。

   Sybase Central の起動とデータベースへの接続の操作に慣れていない場合は、「ローカル・データベースへの接続」『SQL Anywhere サーバ - データベース管理』を参照してください。

2. **データベース・トレーシング・ウィザード**を起動します。

   a. Sybase Central で、**[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]** をクリックします。

   b. **[ファイル]** - **[トレーシング]** - **[トレーシングの設定と開始]** を選択します。

c. **[ようこそ]** ページで、**[次へ]** をクリックします。

d. **[トレーシング詳細レベル]** ページで、**[高 (短期間の集中的なモニタリングに推奨)]** を選択し、**[次へ]** をクリックします。

e. **[トレーシング・レベルの編集]** ページで、**[次へ]** をクリックします。

f. **[外部データベースの作成]** ページで、**[新しいデータベースを作成せず、既存のトレーシング・データベースを使用]** を選択し、**[次へ]** をクリックします。

g. **[トレースの開始]** ページで、**[このデータベースにトレーシング・データを保存]** を選択します。

h. 格納するトレーシング・データの量を制限しない場合は、**[制限なし]** を選択し、**[完了]** をクリックします。

3. Sybase Central の左ウィンドウ枠で、*app_profiling - DBA* データベースを選択し、**[ファイル]** - **[Interactive SQL を開く]** を選択します。

   Interactive SQL が起動し、*app_profiling - DBA* データベースに接続します。

4. Interactive SQL で、次の SQL 文を実行します。

```
SELECT SalesOrderItems.ID, LineID, ProductID, SalesOrderItems.Quantity, ShipDate
FROM SalesOrderItems, SalesOrders
WHERE SalesOrders.CustomerID = 105 AND
  SalesOrderItems.ID=SalesOrders.ID;
```

5. Interactive SQL で、次の SQL 文を実行します。このクエリは、前のクエリと同じ結果を返しますが、非相関サブクエリを使用します。

```
SELECT *
FROM SalesOrderItems
WHERE SalesOrderItems.ID IN (
  SELECT SalesOrders.ID
  FROM SalesOrders
  WHERE SalesOrders.CustomerID = 105 );
```

6. Interactive SQL を終了します。

7. Sybase Central で、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、トレーシング・セッションを停止します。

データベース・トレーシング・ウィザードの詳細については、「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」 205 ページを参照してください。

レッスン 2：データベース・サーバによって処理された文の確認

Sybase Central の**[アプリケーション・プロファイリング]** ウィンドウ枠にある **[概要]** タブと **[詳細]** タブを使用すると、データベース・サーバが最も処理に時間を要する文を識別できます。

♦ **データベース・サーバによって処理された文を確認するには、次の手順に従います。**

1. 分析ファイルを開きます。

a. Sybase Central で、**[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]** をクリックします。

b. **[アプリケーション・プロファイリング] - [分析ファイルを開くかトレーシング・データベースに接続]** を選択します。

c. **[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

d. **[ID]** タブをクリックし、**[ユーザ ID]** フィールドに **DBA** と入力し、**[パスワード]** フィールドに **sql** と入力します。

e. **[データベース]** タブをクリックし、**[データベース・ファイル]** フィールドで *app_profiling - DBA* を参照して選択します。

f. **[OK]** をクリックします。

ウィンドウの下部に **[アプリケーション・プロファイリングの詳細]** ウィンドウ枠が表示されない場合は、**[表示] - [アプリケーション・プロファイリングの詳細]** を選択します。

2. トレーシング・セッション中に処理された文の実行時間を調査します。

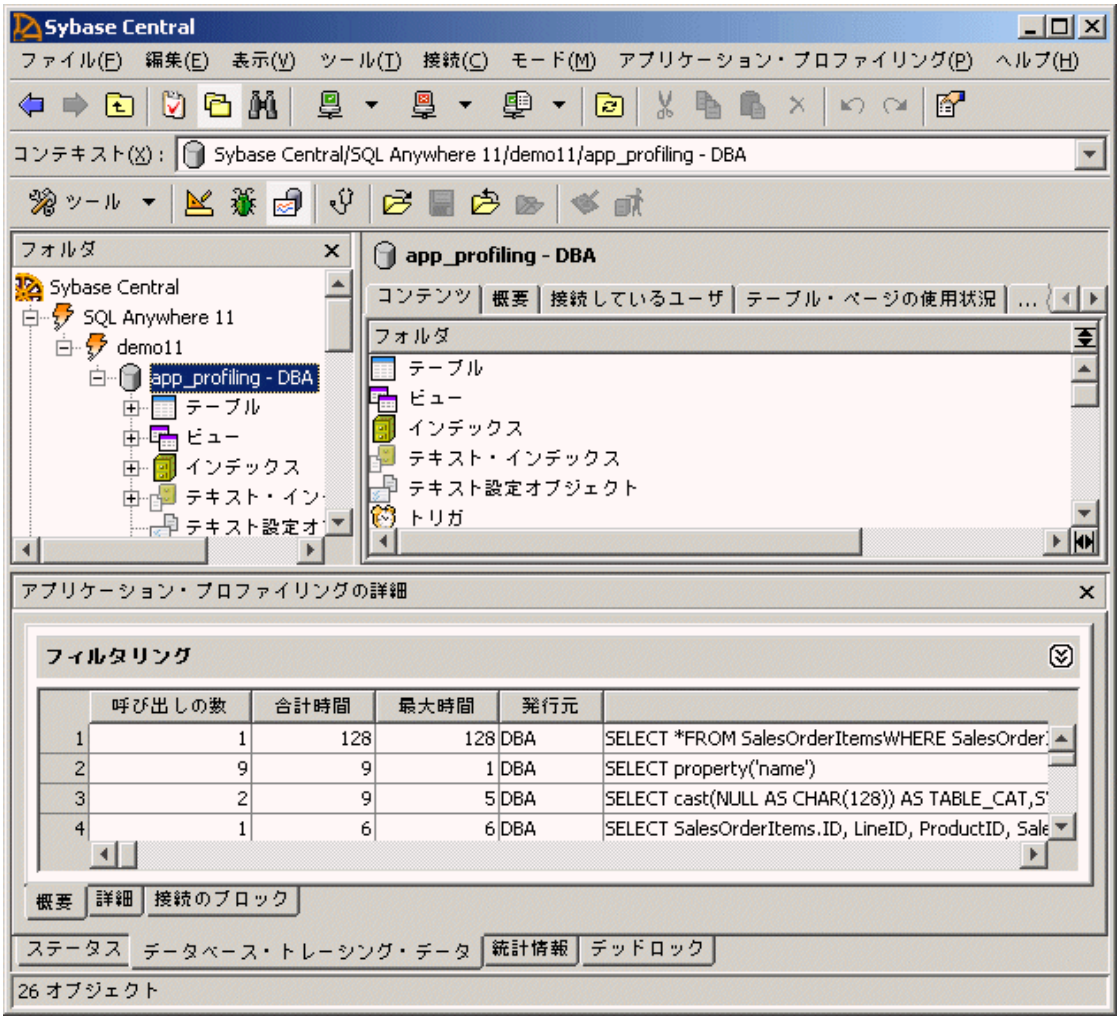
a. **[アプリケーション・プロファイリングの詳細]** ウィンドウ枠の **[ステータス]** タブで、**[ロギング・セッション ID]** フィールドから最新の ID (最も大きな数値) を選択し、**[データベース・トレーシング・データ]** タブをクリックします。

b. 選択したセッションのデータが表示されます。

[概要] タブに、そのセッション中に実行した SQL 文が表示されます。さらに別の文も表示される場合があります。これは、実行した文によって他の文が自動的に実行されたためです (たとえば、トリガなど)。

[概要] タブでは、似ている文がグループ化され、合計呼び出し回数と合計処理時間がまとめて表示されます。SELECT、INSERT、UPDATE、DELETE の各文は、参照するテーブル、カラム、式ごとに分類されます。その他の文は 1 つにまとめられます。たとえば、CREATE TABLE 文は **[概要]** タブに 1 つのエントリとして表示されます。**[概要]** タブに文のコストが高いと表示された場合は、その文のコストが高いか、その文が頻繁に実行された可能性があります。

[合計時間] と **[最大時間]** カラムを使用して、このチュートリアルより前に実行した 2 つのクエリの実行時間を調査します。最初のクエリには、実行の合計時間として 20 ミリ秒が表示されます。2 番目のクエリには、1 番目より速い実行時間 (16 ミリ秒) が表示されます。これにより、非相関サブクエリを使用する 2 番目のクエリは、より効率的な構文であると考えられます。



3. **[概要]** タブで任意の SQL 文に関する追加情報を表示するには、文を右クリックし、**[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。

   - 文を実行した接続に関する情報を表示するには、文を右クリックし、**[選択された文に対する接続の詳細を表示]** を選択します。
   - 文に使用された実行プランを表示するには、**[詳細]** タブで文を右クリックし、**[選択された文に対する SQL 文の詳細を表示]** を選択します。

     [SQL 文の詳細] ウィンドウが開き、文が使用されたコンテキストに関する詳細情報とともに、文の完全なテキストが表示されます。表示される文のテキストは、実行した元の SQL 文とは一致しない場合があります。さらに、**[SQL 文の詳細]** ウィンドウには、データベース・サーバによって処理されたときに書き換えられたフォームで文が表示されます。たとえば、ビュー定義はクエリの実行時にオプティマイザによって書き換えられることが多いため、ビューに表示されるクエリは大幅に異なっている可能性があります。

     実行プランを表示するには、**[クエリ情報]** タブをクリックします。

実行プランに表示される項目の詳細については、「実行プランの解釈」 642 ページを参照してください。

相関サブクエリと非相関サブクエリの詳細については、「サブクエリの使用」 531 ページを参照してください。

[概要] タブと **[詳細]** タブの使用方法の詳細については、「要求トレース分析の実行」 222 ページを参照してください。

チュートリアル：インデックスの断片化の診断

このチュートリアルでは、**アプリケーション・プロファイリング・ウィザード**を使用して、データベースに許容できないレベルのインデックスの断片化が発生しているかどうかを判別する方法を学習します。

インデックスが作成されると、テーブル・データが読み込まれ、インデックスの値が論理順序に従ってインデックス・ページに記録されます。テーブル内のデータを変更した場合は、新しいインデックス値を既存の値の間に挿入できます。データベース・サーバは、インデックス値の論理順序を維持するために、新しいインデックス・ページを作成し、移動された既存の値を調整する必要がある場合があります。この新しいページは、通常、その値が最初に格納されていたページとは隣接していません。このようなインデックス・ページの順序の崩れが累積することを、インデックスの断片化と呼びます。

インデックスの断片化が起こると、大きなローのブロックが継続的に挿入、更新、削除されるテーブルに対してよく実行されるクエリの実行時間が長くなるという現象が発生します。

レッスン 1：インデックスの断片化の設定

このチュートリアルは、テスト・データベースが作成されていることを前提としています。テスト・データベースを作成していない場合は、「レッスン 1：テスト・データベースの作成」 280 ページを参照してください。

ヒント
このチュートリアルの SQL 文をコピーして Interactive SQL にペーストできます。

♦ **インデックスの断片化を設定するには、次の手順に従います。**

1. Sybase Central を起動し、ユーザ ID に **DBA**、パスワードに **sql** を使用してテスト・データベース *app_profiling.db* に接続します。

   Sybase Central の起動とデータベースへの接続の操作に慣れていない場合は、「ローカル・データベースへの接続」 『SQL Anywhere サーバ - データベース管理』を参照してください。

2. 左ウィンドウ枠で、*app_profiling - DBA* を選択し、**[ファイル] - [Interactive SQL を開く]** を選択します。

   Interactive SQL が起動し、*app_profiling - DBA* データベースに接続します。

3. Interactive SQL で、次の SQL 文を実行し、インデックスの断片化を発生させます。これらの文は完了するまでに数分かかる場合があります。

   ```
   CREATE TABLE fragment ( id INT );
   CREATE INDEX idx_fragment ON fragment ( id );
   INSERT INTO fragment SELECT * FROM sa_rowgenerator ( 0, 100000 );
   DELETE FROM fragment WHERE MOD ( id, 2 ) = 0;
   INSERT INTO fragment SELECT * FROM sa_rowgenerator ( 0, 100000 );
   INSERT INTO fragment SELECT * FROM sa_rowgenerator ( 0, 100000 );
   COMMIT;
   ```

4. Interactive SQL を終了します。

レッスン 2：インデックスの断片化の特定

この手順により、インデックスの断片化を特定し、インデックスの断片化に関する警告の参照場所を確認します。運用データベースの断片化に関する警告は、定期的に確認することをおすすめします。

注意
前の手順で実行した文により、インデックスの断片化が発生します。ただし、システムによっては、この手順に記載した警告や推奨事項が表示されるほどの断片化は発生しない場合があります。

♦ **インデックスの断片化を特定するには、次の手順に従います。**

1. Sybase Central で、**[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。
2. **[ようこそ]** ページで、**[次へ]** をクリックします。
3. **[プロファイリング・オプション]** ページで、**[データベース・スキーマに基づく全体的なデータベース・パフォーマンス]** を選択し、**[次へ]** をクリックします。
4. **[分析ファイル]** ページの **[次のファイルに分析を保存]** フィールドに、*C:¥AppProfilingTutorial* と入力します。
5. **[完了]** をクリックします。

   [アプリケーション・プロファイリングの詳細] ウィンドウ枠に推奨リストが表示されます。
6. さらに詳細な情報を表示するには、**[断片化されたインデックス]** をダブルクリックします。**[推奨]** ウィンドウが開き、SQL 文が表示されます。この SQL 文を実行するとインデックスの断片化を解決できます。

レッスン 3：テーブルのインデックス密度の確認

テーブルのインデックスの密度を定期的に確認するには、sa_index_density システム・プロシージャを実行します。密度の値は 0 から 1 の範囲の値です。1 に近い値は、インデックスの断片化がほとんどないことを示しています。0.5 未満の値は、インデックスの断片化のレベルがパフォーマンスに影響を与える可能性があることを示しています。

Interactive SQL で、次の SQL 文を実行し、このチュートリアルの実行中に断片化テーブルに表示されたインデックスの断片化を参照します。

```
CALL sa_index_density( 'fragment' );
```

TableName	TableId	IndexName	IndexId	IndexType	LeafPages	Density
fragment	736	idx_fragment	1	NUI	1,177	0.597509

結果はさまざまに異なる可能性がありますが、**[Density]** 列の値は、ほぼ 0.6 になります。

Interactive SQL で、次の SQL 文を実行し、インデックスの密度を向上させます。

```
ALTER INDEX idx_fragment ON fragment REBUILD;
```

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』
- 「インデックスの再構築」 80 ページ
- 「インデックスの断片化とスキューの削減」 262 ページ
- 「アプリケーション・プロファイリング」 191 ページ

チュートリアル：テーブルの断片化の診断

このチュートリアルでは、**アプリケーション・プロファイリング・ウィザード**を使用して、データベースに許容できないレベルのテーブルの断片化が発生しているかどうかを判別する方法を学習します。

テーブル・データはデータベース・ページに格納されています。INSERT、UPDATE、DELETE などのデータ修正言語 (DML) 文をテーブルに対して実行すると、ローが連続して格納されなかったり、ローが複数のページに渡って分割される場合があります。CPU のアクティビティが高い場合でも、テーブルの断片化は、テーブルのスキャンを必要とするクエリのパフォーマンスに悪影響を与える可能性があります。

レッスン 1：テーブルの断片化の設定

このチュートリアルは、テスト・データベースが作成されていることを前提としています。テスト・データベースを作成していない場合は、「レッスン 1：テスト・データベースの作成」 280 ページを参照してください。

ヒント
このチュートリアルの SQL 文をコピーして Interactive SQL にペーストできます。

♦ **テーブルの断片化を設定するには、次の手順に従います。**

1. Sybase Central を起動し、ユーザ ID に **DBA**、パスワードに **sql** を使用してテスト・データベース *app_profiling.db* に接続します。

   Sybase Central の起動とデータベースへの接続の操作に慣れていない場合は、「ローカル・データベースへの接続」 『SQL Anywhere サーバ - データベース管理』を参照してください。

2. 左ウィンドウ枠で、*app_profiling - DBA* を選択し、**[ファイル] - [Interactive SQL を開く]** を選択します。

   Interactive SQL が起動し、*app_profiling - DBA* データベースに接続します。

3. Interactive SQL で、次の SQL 文を実行し、テーブルの断片化を発生させます。

   a. テーブルを作成します。

```
CREATE TABLE "DBA"."tablefrag" (
"id" UNSIGNED BIGINT NOT NULL DEFAULT AUTOINCREMENT,
"val1" LONG VARCHAR NULL,
"val2" LONG VARCHAR NULL,
"val3" LONG VARCHAR NULL,
"val4" LONG VARCHAR NULL,
"val5" LONG VARCHAR NULL,
"val6" LONG VARCHAR NULL,
"val7" LONG VARCHAR NULL,
"val8" LONG VARCHAR NULL,
"val9" LONG VARCHAR NULL,
"val10" LONG VARCHAR NULL,
PRIMARY KEY ( id ) );
```

   b. テーブルに値を挿入するプロシージャを作成します。


```
CREATE PROCEDURE "DBA"."proc_tablefrag"( )
  BEGIN
    DECLARE I INTEGER;
    SET I = 0;
    WHILE I < 1000
      LOOP
        INSERT INTO "DBA"."tablefrag" ( "val1" )
        VALUES('a');
        SET I = I + 1;
      END LOOP;
  END;
```

c. 値を挿入します。

```
CALL proc_tablefrag( );
```

d. テーブル内の値を更新します。

```
UPDATE "DBA"."tablefrag"
SET "val1" = 'abcdefghijklmnopqrstuvwxyz0123456789',
   "val2" = 'abcdefghijklmnopqrstuvwxyz0123456789',
   "val3" = 'abcdefghijklmnopqrstuvwxyz0123456789',
   "val4" = 'abcdefghijklmnopqrstuvwxyz0123456789',
   "val5" = 'abcdefghijklmnopqrstuvwxyz0123456789',
   "val6" = 'abcdefghijklmnopqrstuvwxyz0123456789',
   "val7" = 'abcdefghijklmnopqrstuvwxyz0123456789',
   "val8" = 'abcdefghijklmnopqrstuvwxyz0123456789',
   "val9" = 'abcdefghijklmnopqrstuvwxyz0123456789',
   "val10" = 'abcdefghijklmnopqrstuvwxyz0123456789';
```

e. データベースに加えた変更をコミットします。

```
COMMIT;
```

4. Interactive SQL を終了します。

レッスン 2：テーブルの断片化の特定

この手順により、テーブルの断片化を特定し、テーブルの断片化に関する警告の参照場所を確認します。運用データベースの断片化に関する警告は、定期的に確認することをおすすめします。

注意
前の手順で実行した文により、テーブルの断片化が発生します。ただし、システムによっては、この手順に記載した警告や推奨事項が表示されるほどの断片化は発生しない場合があります。

♦ **テーブルの断片化を特定するには、次の手順に従います。**

1. Sybase Central で、**[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。

2. **[プロファイリング・オプション]** ページで、**[データベース・スキーマに基づく全体的なデータベース・パフォーマンス]** を選択します。

3. **[分析ファイル]** ページで、適切なディレクトリに分析ファイルを保存します。たとえば、*C:¥AppProfilingTutorial* などのディレクトリに保存します。
4. **[完了]** をクリックします。

   [アプリケーション・プロファイリングの詳細] ウィンドウ枠に推奨リストが表示されます。
5. さらに詳細な情報を表示するには、**[断片化されたテーブル]** をダブルクリックします。**[推奨]** ウィンドウが開き、SQL 文が表示されます。この SQL 文を実行するとテーブルの断片化を解決できます。

レッスン 3：テーブルの断片化の確認

テーブルの断片化 (たとえば、CALL sa_table_fragmentation('tablefrag'); など) がないかどうか確認するには、sa_table_fragmentation システム・プロシージャを実行します。ローごとのセグメント数が 1.1 よりも多い場合は、テーブルの断片化が存在します。断片化の程度が高いほど、パフォーマンスに悪影響を与える可能性があります。「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。

このチュートリアルで作成したテーブルの断片化の値は、約 1.9 になります。

Interactive SQL で、次の SQL 文を実行し、テーブルの断片化を削減します。

```
REORGANIZE TABLE tablefrag;
```

「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

- 「テーブルの断片化削減」 261 ページ
- 「アプリケーション・プロファイリング」 191 ページ

チュートリアル：プロシージャ・プロファイリングをベースラインとして使用

このチュートリアルでは、**アプリケーション・プロファイリング・ウィザード**を使用して、ベースラインを作成する方法を学習します。このベースラインは、パフォーマンスを改善するときに比較する目的で使用できます。

プロシージャ・プロファイリングでは、プロシージャ、ユーザ定義関数、イベント、システム、システム・トリガ、トリガの実行時間の測定値が収集されます。保存された結果をベースラインとして設定し、プロシージャを段階的に変更して、変更後にプロシージャを実行できます。これにより、新しい結果をベースラインと比較できます。

レッスン 1：baseline プロシージャの作成

このチュートリアルは、テスト・データベースが作成されていることを前提としています。テスト・データベースを作成していない場合は、「レッスン 1：テスト・データベースの作成」 280 ページを参照してください。

ヒント
このチュートリアルの SQL 文をコピーして Interactive SQL にペーストできます。

♦ **baseline プロシージャを作成するには、次の手順に従います。**

1. Sybase Central を起動し、ユーザ ID に **DBA**、パスワードに **sql** を使用してテスト・データベース *app_profiling - DBA* に接続します。

   Sybase Central の起動とデータベースへの接続の操作に慣れていない場合は、「ローカル・データベースへの接続」 『SQL Anywhere サーバ - データベース管理』を参照してください。

2. 左ウィンドウ枠で、*app_profiling - DBA* を選択し、**[ファイル] - [Interactive SQL を開く]** を選択します。

   Interactive SQL が起動し、*app_profiling - DBA* データベースに接続します。

3. Interactive SQL で、次の SQL 文を実行します。

   a. テーブルを作成します。

   ```
   CREATE TABLE table1 (
   Count INT );
   ```

   b. ストアド・プロシージャを作成します。

   ```
   CREATE PROCEDURE baseline( )
     BEGIN
       INSERT table1
         SELECT COUNT (*)
           FROM rowgenerator r1, rowgenerator r2,
                rowgenerator r3
         WHERE r3.row_num < 5;
     END;
   ```

c. データベースに加えた変更をコミットします。

```
COMMIT;
```

4. Interactive SQL を閉じます。

レッスン 2：baseline プロシージャに対する更新されたプロシージャの実行

♦ baseline プロシージャに対して、更新された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 で、**[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。

2. **[ようこそ]** ページで、**[次へ]** をクリックします。
3. **[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。
4. **[完了]** をクリックします。

   データベース・サーバでプロシージャ・プロファイリングが開始します。

5. Sybase Central の左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックします。
6. baseline プロシージャを右クリックし、**[Interactive SQL から実行]** を選択します。プロシージャ・プロファイリングが有効になっているため、プロシージャの実行の詳細が取得されます。
7. Interactive SQL を閉じます。
8. プロファイリング結果を表示します。
   a. Sybase Central の左ウィンドウ枠で、baseline プロシージャを選択します。
   b. 右ウィンドウ枠で、**[プロファイリング結果]** タブをクリックします。結果が表示されない場合は、**[表示] - [フォルダの再表示]** を選択します。

      baseline プロシージャの行ごとの実行時間が表示されます。

9. プロファイリング結果を保存します。
   a. データベースを右クリックして、**[プロパティ]** を選択します。
   b. **[プロファイリング設定]** タブをクリックします。
   c. **[データベース内に現在あるプロファイリング情報を次のプロファイリング・ログ・ファイルに保存する]** を選択し、プロファイリング・ログ・ファイルのロケーションとファイル名を指定します。
   d. **[適用]** をクリックします。プロパティ・ウィンドウは閉じないでください。

収集したプロシージャのプロファイリング情報は、指定したプロファイリング・ログ・ファイル (*.plg*) に保存されます。

10. プロファイリング・ログ・ファイルをベースラインとして使用することを指定します。
    a. **[App_Profiling - DBA データベースのプロパティ]** ウィンドウの **[プロファイリング設定]** タブで、**[次のプロファイリング・ログ・ファイルのプロファイリング情報を比較のベースラインとして使用する]** を選択します。
    b. 作成したプロファイリング・ログ・ファイルを探して選択します。
    c. **[適用]** をクリックします。
    d. **[OK]** をクリックし、**[App_Profiling - DBA データベースのプロパティ]** ウィンドウを閉じます。
11. baseline プロシージャに変更を加えます。
    a. Sybase Central で、**[モード]** - **[設計]** を選択します。
    b. 左ウィンドウ枠の **[プロシージャとファンクション]** で baseline プロシージャを参照して選択します。
    c. 右ウィンドウ枠の **[SQL]** タブで、既存の INSERT 文を削除します。
    d. 次の SQL 文をコピーし、プロシージャにペーストします。

```
INSERT table1
  SELECT COUNT ( * ) FROM rowgenerator r1, rowgenerator r2, rowgenerator r3
  WHERE r3.row_num < 250;
```

    e. **[ファイル]** - **[保存]** を選択します。
12. **[プロシージャとファンクション]** で、baseline プロシージャを右クリックし **[Interactive SQL から実行]** を選択します。
13. プロシージャが完了したら、Interactive SQL を終了します。

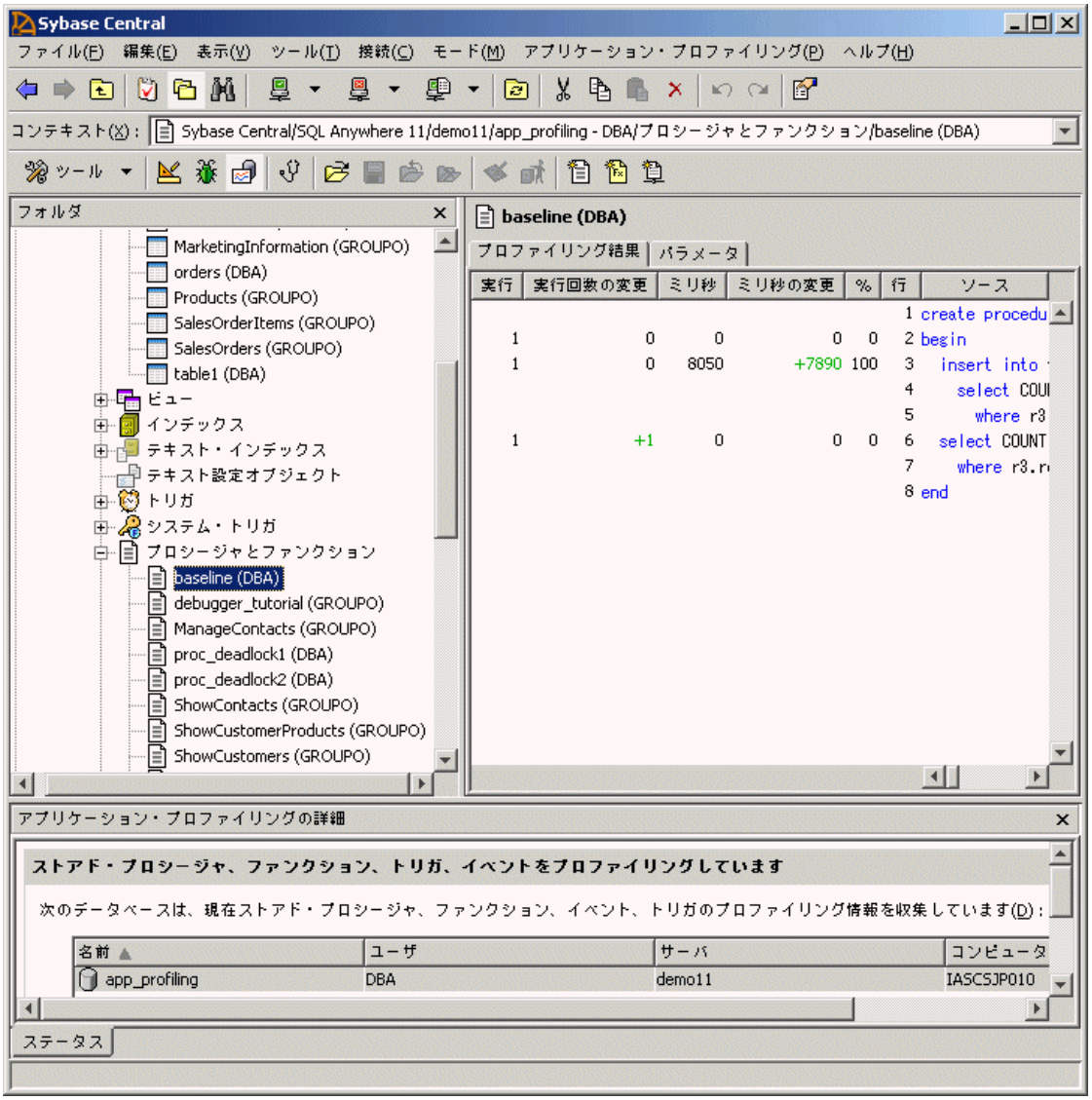
レッスン 3：プロシージャ・プロファイリングの結果の比較

♦ **プロシージャ・プロファイリングの結果を比較するには、次の手順に従います。**

1. Sybase Central で、**[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]** をクリックします。
2. Sybase Central の左ウィンドウ枠の **[プロシージャとファンクション]** で、baseline プロシージャをクリックします。
3. 右ウィンドウ枠で、**[プロファイリング結果]** タブをクリックします。
4. **[表示]** - **[フォルダの再表示]** を選択します。

   2 つの新しいカラム、**[実行回数の変更]** と **[ミリ秒の変更]** が表示されます。



[実行回数の変更] と **[ミリ秒の変更]** 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に、それぞれの実行回数と実行時間を比較します。

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 **[ミリ秒の変更]** 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。このチュートリアルでは、**[ミリ秒の変更]** カラムの値は、+ 記号の付いた実行時間が緑の文字で表示されます。つまり、更新されたプロシージャの INSERT 文の方が、baseline プロシージャの INSERT 文よりも実行時間が長いということです。

参照

- 「プロシージャ・プロファイリングの結果の分析」 196 ページ
- 「アプリケーション・プロファイリング・モードでのプロシージャ・プロファイリング」 193 ページ

データのクエリと変更

この項では、データの問い合わせ方法と変更方法、ジョインの使用方法について説明します。クエリについて、数章に分けて簡単な内容から複雑な内容へと説明します。また、データの挿入、削除、更新についても説明します。さらに、多次元の結果を返す分析クエリの作成方法の詳細についても説明します。

データのクエリ

目次

クエリはデータベースにデータを要求し、結果を受け取ります。この処理はデータ検索とも呼ばれます。SQL クエリはすべて、SELECT 文を使用して表現されます。SELECT 文は、1 つ以上のテーブルですべてのローまたはそのサブセットを検索するとき、および 1 つ以上のテーブルですべてのカラムまたはそのサブセットを検索するときに使用します。

クエリと 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・アプリケーションに配信されます。クライアントは、配信された結果セットを処理できます。たとえば、Interactive SQL では、結果セットが [結果] ウィンドウ枠に表示されます。結果セットは、データベースのテーブルと同様に一連のローで構成されます。

SELECT 文には「句」を含めます。句は返される結果のスコープを定義するコマンドです。次の SELECT 構文では、各行が別々の句です。ここではごく一般的な句を示します。

SELECT *select-list*
[**FROM** *table-expression*]
[**WHERE** *search-condition*]
[**GROUP BY** *column-name*]
[**HAVING** *search-condition*]
[**ORDER BY** { *expression* | *integer* }]

次に、SELECT 文の各句について説明します。

- SELECT 句は、どのカラムを取得するかを指定します。この句は、SELECT 文で必須の句です。
- FROM 句は、どのテーブルからカラムを検索するかを指定します。この句は、テーブルからデータを取り出すすべてのクエリに必要です。FROM 句を持たない SELECT 文には別の意味がありますが、それについてはこの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 (派生テーブル)、ストアド・プロシージャの結果セットなど、カラムとローを持つ他のオブジェクトからデータを取り出せます。「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」 391 ページを参照してください。

これらの句の大半は省略可能ですが、SELECT 文に記述するときは、正しい順序で記述してください。

「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。これは文を読みやすくするためであり、必須条件ではありません。SQL キーワードは大文字／小文字のいずれでも入力でき、文のどこでも改行できます。

キーワードと改行

たとえば、次の SELECT 文は、Contacts テーブルからカリフォルニア州内の連絡先の名前と姓を検索します。

```
SELECT GivenName, Surname
FROM Contacts
WHERE State = 'CA';
```

読みやすくはありませんが、この文を次のように入力しても有効です。

```
SELECT GivenName,
Surname from Contacts
WHERE State
 = 'CA';
```

文字列と識別子の大文字と小文字の区別

SQL Anywhere データベースでは、テーブル名、カラム名などの識別子は、大文字と小文字を区別しません。

文字列はデフォルトでは大文字と小文字が区別されないので、'CA'、'ca'、'cA'、'Ca' は同じです。ただし、大文字と小文字を区別するデータベースを作成する場合は、文字列中の大文字／小文字が重要になります。SQL Anywhere サンプル・データベースでは大文字と小文字を区別しません。

参照

- 「データベースの作成」 『SQL Anywhere サーバ - データベース管理』
- 「初期化ユーティリティ (dbinit)」 『SQL Anywhere サーバ - データベース管理』
- 「大文字と小文字の区別」 703 ページ

識別子の修飾

どのオブジェクトのことを指しているのかはっきりしない場合には、データベース識別子の名前を修飾します。たとえば、SQL Anywhere サンプル・データベースにはカラム City を含んだテーブルがいくつかあるので、City への参照をテーブル名で修飾する必要があります。より規模の大きいデータベースでは、テーブルの所有者の名前を使用してテーブルを識別する場合もあります。

```
SELECT Contacts.City
FROM Contacts
WHERE State = 'CA';
```

この項の例で示すのは単一テーブルのクエリですから、構文のモデルや例に出てくるカラム名を、カラムが属しているテーブルや所有者の名前で修飾することは通常はありません。

これらの要素は読みやすさを考慮して省略してあるものなので、修飾しても決して間違いではありません。

結果セットのローの順序

結果セットのローの順序に意味はありません。データベースからローが返される順序に保証はなく、またその順序に意味はありません。ローを特定の順序で取り出す場合は、クエリで順序を指定する必要があります。

select リスト：カラムの指定

select リストは、データを問い合わせる 1 つ以上のオブジェクトから構成されます。select リストは、通常はカンマで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。リストの最終カラムの後や、リストにカラムが 1 つしかない場合、カンマはありません。

select リストの一般的な構文は次のようになります。

SELECT *expression* [, *expression*]...

リスト中に、有効な識別子としての規則を満たしていないテーブルやカラムを記述する場合は、それらの識別子を二重引用符で囲んでください。

select リストの式に記述することができるのは、* (すべてのカラム)、カラム名のリスト、文字列、カラムの見出し、算術演算子のある式です。また、集合関数も記述できます。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」 391 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。

テーブルからのすべてのカラムの選択

SELECT 文の中のアスタリスク (*) には特殊な意味があります。アスタリスクは、FROM 句で指定されたすべてのテーブルにあるすべてのカラム名を表します。1 つのテーブルのカラムをすべて参照したいときにアスタリスクを使用すると、入力する時間が節約でき、入力ミスを防ぐことができます。

SELECT * を使用すると、テーブルが作成されたときに定義された順で、カラムが返されます。

1 つのテーブルのカラムをすべて選択するための構文は次のとおりです。

```
SELECT *
FROM table-expression;
```

SELECT * は、1 つのテーブルに現在登録されているカラムをすべて検索するので、カラムの追加、削除、名前の変更など、テーブル構造の変更によって、SELECT * の結果も自動的に変わります。カラムを個別にリストする方が、結果の正確さを確保できます。

例

次の文は Departments テーブルのすべてのカラムを検索します。WHERE 句はありません。そのため、この文はテーブルのすべてのローを検索します。

```
SELECT *
FROM Departments;
```

結果は次のようになります。

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
...	...	...

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カラムをリストすることで制限できます。この SELECT 文の構文は、次のとおりです。

SELECT *column-name* [, *column-name*]...
FROM *table-name*

この構文では、*column-name* と *table-name* は、問い合わせるカラムとテーブルの名前に置き換えてください。

次に例を示します。

```
SELECT Surname, GivenName
FROM Employees;
```

射影と制限

「射影」とは、テーブル内のカラムのサブセットです。「制限」 (「選択」とも言う) は、いくつかの条件に基づいたテーブル内のローのサブセットとのことです。

たとえば、次の SELECT 文では、SQL Anywhere サンプル・データベースで価格が $15 より高い製品すべての名前と価格が検索されます。

```
SELECT Name, UnitPrice
FROM Products
WHERE UnitPrice > 15;
```

このクエリでは、射影 (SELECT Name, UnitPrice) と制限 (WHERE UnitPrice > 15) を使用しています。

カラムの順番の再調整

カラム名をリストする順番で、カラムが表示される順番が決まります。次の 2 つの例は、表示におけるカラム順の指定方法を示しています。2 つとも、Departments テーブルの 5 つのロー全部から部署名と ID を検出して表示します。ただし、順番が異なります。

```
SELECT DepartmentID, DepartmentName
FROM Departments;
```

DepartmentID	DepartmentName
100	R & D
200	Sales
300	Finance
400	Marketing
...	...

```
SELECT DepartmentName, DepartmentID
FROM Departments;
```

DepartmentName	DepartmentID
R & D	100
Sales	200
Finance	300
Marketing	400
...	...

ジョイン

ジョインは、各テーブルのカラムの値を比較して、2 つ以上のテーブル内のローをリンクします。たとえば、1 ダースを超える数が出荷されたすべての注文項目について、注文項目 ID 番号と製品名を次のように選択できます。

```
SELECT SalesOrderItems.ID, Products.Name
FROM Products JOIN SalesOrderItems
WHERE SalesOrderItems.Quantity > 12;
```

Products テーブルと SalesOrderItems テーブルは、この両テーブル間の外部キー関係に基づいてジョインされます。

「ジョイン：複数テーブルからのデータ検索」 413 ページを参照してください。

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

ほとんどの SELECT 文は、FROM 句のテーブルのデータだけで構成された結果を生成します。ただし、文字列を一重引用符で囲み、それを select リストの他の要素とカンマで区切ると、その文字列もクエリ結果に表示されます。文字列に引用符を記述するには、その前に引用符をもう 1 つ記述します。次に例を示します。


```
SELECT 'The department"s name is' AS "Prefix",
   DepartmentName AS Department
FROM Departments;
```

Prefix	Department
The department's name is	R & D
The department's name is	Sales
The department's name is	Finance
The department's name is	Marketing
The department's name is	Shipping

SELECT リストの値の計算

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
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;
```

確実に正しい結果を得るためには、可能な限りカッコを使用します。次のクエリは前述のクエリと同じ意味で、同じ結果を生成しますが、構文の精度が高くなります。

```
SELECT Name, ( Quantity * UnitPrice ) - 5
FROM Products;
```

「演算子の優先度」 『SQL Anywhere サーバ - SQL リファレンス』も参照してください。

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。たとえば、次の文では、結果の Surname と GivenName の値を取り出して、連結しています。

```
SELECT EmployeeID, GivenName || ' ' || Surname AS Name
FROM Employees;
```

EmployeeID	Name
102	Fran Whitney
105	Matthew Cobb
129	Philip Chin
148	Julie Jordan
...	...

日付と時刻の演算

日付カラムと時刻カラムでも演算子を使用できますが、そのためには一般的に関数を使用することになります。「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」 『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;
  ```

ID	UCASE(Products.name)
300	TEE SHIRT
301	TEE SHIRT
302	TEE SHIRT
400	BASEBALL CAP
...	...

「SQL 関数」 『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。「不要な DISTINCT 条件の排除」 578 ページを参照してください。

FROM 句：テーブルの指定

テーブル、ビュー、またはストアド・プロシージャのデータに関係のある SELECT 文には、FROM 句が必須です。

FROM 句には、2 つ以上のテーブルをリンクする JOIN 条件を記述できるほか、他のクエリ (派生テーブル) へのジョインを記述できます。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」 413 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、または所有者のユーザ ID が現在の接続のユーザ ID が所属するグループ名と異なる場合だけです。

相関名の使用

テーブル名に相関名を指定すると、読みやすさが向上し、テーブル名を参照する箇所で毎回完全な名前を入力する手間が省けます。相関名は、次のように、FROM 句でテーブル名の後に入力して割り当てます。

```
SELECT d.DepartmentID, d.DepartmentName
FROM Departments d;
```

相関名が使用される場合、たとえば WHERE 句の中など、そのテーブルに対する他のすべての参照には、テーブル名ではなく、**必ず**相関名を使用してください。相関名は有効な識別子としての規則に従っている必要があります。

「FROM 句」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

派生テーブルのクエリ

派生テーブルは、クエリ式の評価によって、1 つまたは複数のテーブルから直接または間接的に派生されるテーブルです。派生テーブルは、SELECT 文の FROM 句に定義します。

派生テーブルのクエリは、ビューのクエリと同様に動作します。つまり、派生テーブルの値は、派生テーブル定義の評価時に決定されます。ただし、派生テーブルは、その定義がデータベースに保存されないという点で、ビューと異なります。また、派生テーブルは実体化されず、それが定義されているクエリ外からは参照できないという点で、ベース・テーブルやテンポラリ・テーブルと異なります。

次のクエリでは派生テーブル (my_drv_tbl) を使用して、各部署の最高給与を保持しています。派生テーブルのデータは、Employees テーブルにジョインして、その給与の従業員の姓を取得できます。

```
SELECT Surname,
  my_drv_tbl.max_sal AS Salary,
  my_drv_tbl.DepartmentID
```

```
FROM Employees e,
  ( SELECT MAX( Salary ) AS max_sal, DepartmentID
      FROM Employees
      GROUP BY DepartmentID ) my_drv_tbl
  WHERE e.Salary = my_drv_tbl.max_sal
  AND e.DepartmentID = my_drv_tbl.DepartmentID
ORDER BY Salary DESC;
```

Surname	Salary	DepartmentID
Shea	138948.00	300
Scott	96300.00	100
Kelly	87500.00	200
Evans	68940.00	400
Martinez	55500.80	500

次の例では、Products テーブルの項目をランクする派生テーブル (MyDerivedTable) を作成し、派生テーブルを問い合わせて、最も低価格の 3 つの項目を返しています。

```
SELECT TOP 3 *
      FROM ( SELECT Description,
                  Quantity,
                  UnitPrice,
                  RANK() OVER ( ORDER BY UnitPrice ASC )
               AS Rank
               FROM Products ) AS MyDerivedTable
ORDER BY Rank;
```

参照：「FROM 句」 『SQL Anywhere サーバ - SQL リファレンス』。

テーブル以外のオブジェクトのクエリ

FROM 句内の最も一般的な要素は、テーブル名です。ただし、テーブルに似た構造を持つ、すなわち、適切に定義されたローとカラムを持つ他のデータベース・オブジェクトからローを問い合わせることも可能です。たとえば、ビューを問い合わせることも、結果セットを返すストアドプロシージャを問い合わせることもできます。

たとえば、次の文は ShowCustomerProducts というストアド・プロシージャの結果セットを問い合わせます。

```
SELECT *
FROM ShowCustomerProducts( 149 );
```


WHERE 句：ローの指定

SELECT 文の WHERE 句は、どのローを検索するのかを正確に決定する探索条件を指定します。探索条件は「述部」とも呼ばれます。一般的なフォーマットは次のとおりです。

SELECT *select-list*
FROM *table-list*
WHERE *search-condition*

WHERE 句の探索条件には、次のものがあります。

- **比較演算子** (=、<、> など) たとえば、給料が $50,000 を上回る従業員全員をリストする場合は、次のようになります。

```
SELECT Surname
  FROM Employees
  WHERE Salary > 50000;
```

- **範囲** (BETWEEN と NOT BETWEEN) たとえば、給料が $40,000 ～ $60,000 の従業員をすべてリストする場合は、次のようになります。

```
SELECT Surname
  FROM Employees
  WHERE Salary BETWEEN 40000 AND 60000;
```

- **リスト** (IN、NOT IN) たとえば、オンタリオ州、ケベック州、またはマニトバ州の顧客をすべてリストする場合は、次のようになります。

```
SELECT CompanyName, State
  FROM Customers
  WHERE State IN( 'ON', 'PQ', 'MB');
```

- **文字の一致** (LIKE と NOT LIKE) たとえば、電話番号が 415 ではじまる顧客をすべてリストする場合は、次のようになります (データベースでは、電話番号は文字列として保存されています)。

```
SELECT CompanyName, Phone
  FROM Customers
  WHERE Phone LIKE '415%';
```

- **不定の値** (IS NULL と IS NOT NULL) たとえば、マネージャがいる部署をすべてリストする場合は、次のようになります。

```
SELECT DepartmentName
  FROM Departments
  WHERE DepartmentHeadID IS NOT NULL;
```

- **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%';
```

探索条件の完全な構文については、「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。

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」『SQL Anywhere サーバ - SQL リファレンス』と「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"
  ```

  Sybase Central からも **[データベースのプロパティ]** の [詳細情報] タブに移動して、照合順を確認することができます。

- **後続ブランク**　データベースの作成時に、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。

- **日付の比較**　日付を比較するときには、< はより古いことを意味し、> はより新しいことを意味します。

- **大文字と小文字の区別**　データベースの作成時に、文字列比較が大文字と小文字を区別するかどうかを指定します。

  デフォルトでは、データベースは大文字と小文字を区別しないで作成されます。たとえば、'Dirk' は 'DIRK' と同じです。大文字と小文字を区別するように、データベースを作成できます。

次は比較演算子を使用する SELECT 文です。

```
SELECT *
  FROM Products
  WHERE Quantity < 20;
SELECT E.Surname, E.GivenName
  FROM Employees E
  WHERE Surname > 'McBadden';
SELECT ID, Phone
  FROM Contacts
  WHERE State  != 'CA';
```

NOT 演算子

NOT 演算子は式を否定します。次の 2 つのクエリはどちらも、単価が $10 以下の T シャツ (Tee Shirt) と野球帽 (BaseBall Cap) をすべて検索します。ただし、否定の論理演算子 (NOT) と否定の比較演算子 (!>) では、位置が異なることに注意してください。

```
SELECT ID, Name, Quantity
  FROM Products
  WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
  AND NOT UnitPrice > 10;
SELECT ID, Name, Quantity
  FROM Products
  WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
  AND UnitPrice !> 10;
```

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
Baseball Cap	9
Visor	7
Visor	7

Name	UnitPrice
...	...

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。

記号	意味
[specifier]	カッコ内の指定子の形式は、次のとおりです。 ● **範囲**　範囲の形式は *rangespec1-rangespec2* です。*rangespec1* は文字の範囲の始点を指し、ハイフンは範囲、*rangespec2* は文字範囲の終点を指します。 ● **セット**　セットには、任意の順序での不連続な値の集合が含まれます。たとえば、[a2bR] などです。 範囲 [a-f] と、セット [abcdef] と [fcbdae] は、同じ値セットを返します。
[^specifier]	指定子の前にある脱字記号 (^) は非包含を表します。[^a-f] は a ～ f の範囲ではないとの意味で、[^a2bR] は a、2、b、R ではないとの意味です。

カラム・データを、定数、変数、またはテーブルに示されているワイルドカード文字を含むその他のカラムに一致させることができます。定数を使用するときは、マッチ文字列と文字列を一重引用符で囲みます。

例

次の例はすべて、Contacts テーブルの Surname カラムで LIKE を使用しています。クエリの形式は次のとおりです。

```
SELECT Surname
  FROM Contacts
  WHERE Surname LIKE match-expression;
```

最初の例は次のように入力します。

```
SELECT Surname
  FROM Contacts
  WHERE Surname LIKE 'Mc%';
```

マッチ式	説明	検索結果
'Mc%'	Mc で始まるすべての名前を検索します。	McEvoy
'%er'	er で終わるすべての名前を検索します。	Brier, Miller, Weaver, Rayner
'%en%'	en を含むすべての名前を検索します。	Pettengill, Lencki, Cohen
'_ish'	ish で終わるすべての 4 文字の名前を検索します。	Fish
'Br[iy][ae]r'	Brier、Bryer、Briar、または Bryar を検索します。	Brier

マッチ式	説明	検索結果
'[M-Z]owell'	M ～ Z の範囲の 1 文字で始まり、owell で終わるすべての名前を検索します。	Powell
'M[^c]%'	M で始まり、2 番目の文字が c ではないすべての名前を検索します。	Moore, Mulley, Miller, Masalsky

ワイルドカードには LIKE が必須

ワイルドカード文字を LIKE なしで使用すると、パターンとは解釈されず、「文字列リテラル」と解釈されます。つまり、ワイルドカード文字そのものの値を表すことになります。次のクエリは、415% の 4 文字だけから成る電話番号を検出しようとします。415 で始まる電話番号は検出しません。

```
SELECT Phone
  FROM Contacts
  WHERE Phone = '415%';
```

「文字列リテラル」 『SQL Anywhere サーバ - SQL リファレンス』も参照してください。

日付値と時刻値での LIKE の使用

LIKE は、日付フィールド、時刻フィールド、文字データに対して使用できます。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。

たとえば、カラム arrival_time に値 9:20 と現在の日付を入力した場合、エントリに時刻だけではなく日付も格納されているため、次の句は値の検索に失敗します。

```
WHERE arrival_time = '9:20'
```

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。「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。「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。

文字列での引用符

文字エントリ内でリテラル引用符を指定する方法は 2 つあります。1 つめの方法は、引用符を 2 回連続して使用する方法です。たとえば、一重引用符で文字列を開始し、そのエントリの一部として一重引用符を 1 つ入力する場合は、一重引用符を 2 つ使用します。

```
'I don''t understand.'
```

二重引用符の場合は次のように指定します (quoted_identifier が Off の場合)。

```
"He said, ""It is not really confusing."""
```

2 つめの方法は、quoted_identifier が Off の場合にしか使用できませんが、引用符を別の種類の引用符で囲む方法です。つまり、二重引用符が入っているエントリは一重引用符で囲み、逆に一重引用符が入っているエントリは二重引用符で囲みます。次にその例を示します。

```
'George said, "There must be a better way."'
"Isn't there a better way?"
'George asked, "Isn''t there a better way?"'
```

不定の値：NULL

カラムに NULL がある場合は、ユーザまたはアプリケーションがそのカラムになにも入力していないことを意味します。つまり、このカラムのデータ値は、不定か、使用不可です。

NULL は 0 (数値) やブランク (文字値) と同じではありません。むしろ、NULL 値によって、数値カラムの場合の 0 や文字カラムの場合のブランクなどの意図的な入力と、入力がない場合とを区別できます。入力が行われていない場合は、数値カラムと文字カラムは両方とも NULL です。

NULL の入力

NULL は、NULL 値が許可されたカラムにのみ入力できます。カラムに NULL 値が許可されるかどうかは、テーブルの作成時に決まります。カラムに NULL 値が許可されることを前提として、NULL を挿入します。

- **デフォルト**　データが入力されず、カラムに他のデフォルト設定がない場合。
- **明示的な入力**　引用符なしで NULL と明示的に挿入できます。NULL という単語を引用符をつけて文字カラムに入力すると、NULL は NULL 値としてではなく、1 つのデータとして処理されます。

たとえば、Departments テーブルの DepartmentHeadID カラムは NULL 値を許可します。次のように、マネージャのいない部署のローを 2 種類入力できます。

```
INSERT INTO Departments (DepartmentID, DepartmentName)
  VALUES (201, 'Eastern Sales')
INSERT INTO Departments
  VALUES (202, 'Western Sales', null);
```

NULL 値を返す

NULL 値は他の値とまったく同じように、クライアントアプリケーションに返され、表示されます。たとえば、次の例では、Interactive SQL で NULL 値がどのように表示されるかを示しています。

```
SELECT *
FROM Departments;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	904
300	Finance	1293
400	Marketing	1576
500	Shipping	703
201	Eastern Sales	(null)

DepartmentID	DepartmentName	DepartmentHeadID
202	Western Sales	(null)

NULL のためのカラムのテスト

IS NULL 探索条件を使用すると、カラム値を NULL と比較したり、その比較の結果に基づいてカラム値を選択したりするなど、特定の動作を実行できます。TRUE の値を返すカラムだけが、選択されたり、結果として指定の動作を行ったりします。FALSE や UNKNOWN を返すカラムの場合、そういうことはありません。

次の例では、UnitPrice が $15 未満または NULL のローだけを選択します。

```
SELECT Quantity, UnitPrice
  FROM Products
  WHERE UnitPrice < 15
  OR UnitPrice IS NULL;
```

NULL が指定の値や別の NULL に等しいか (または等しくないか) わからないので、NULL 比較の結果はすべて UNKNOWN になります。

true を決して返さない条件があり、そうした条件を使用するクエリは結果セットを返しません。たとえば、NULL は不定の値があるという意味であるため、次の比較は決して true とは見なされません。

```
WHERE column1 > NULL
```

WHERE 句でカラム名を 2 つ使用する場合、つまり 2 つのテーブルをジョインする場合にも、この論理は適用されます。条件 WHERE column1 = column2 を含む句は、カラムに NULL が含まれるローを返しません。

次のパターンで NULL や NOT NULL も検索できます。

```
WHERE column_name IS NULL
```

```
WHERE column_name IS NOT NULL
```

次に例を示します。

```
WHERE advance < $5000
OR advance IS NULL
```

「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 値のプロパティを詳しく説明します。

- **FALSE と UNKNOWN の相違**　FALSE と UNKNOWN はどちらも値を返しませんが、FALSE と UNKNOWN の間には大きな論理的相違があります。false の反対 ("not false") は true です

が、UNKNOWN の反対は何かを知っていることを意味しません。たとえば、1 = 2 は false に評価され、1 != 2 (1 は 2 に等しくない) は true に評価されます。

ただし、比較の中に NULL がある場合は、式を否定しても、反対のロー・セットや反対の真の値は得られません。UNKNOWN 値は UNKNOWN のままです。

- **複数の NULL 値を 1 つの値で置き換える**　ISNULL 組み込み関数を使用して、複数の NULL 値を 1 つの特定の値と置き換えることができます。置換は表示目的のためだけに実行されます。実際のカラム値には影響しません。構文は次のとおりです。

  ISNULL(*expression*, *value*)

  たとえば、次の文を使用して Departments のすべてのローを選択し、すべての NULL 値を -1 という値でカラム DepartmentHeadID に表示します。

  ```
  SELECT DepartmentID,
      DepartmentName,
      ISNULL( DepartmentHeadID, -1 ) AS DepartmentHead
    FROM Departments;
  ```

- **NULL と評価される式**　オペランドのいずれかが NULL 値の場合、算術演算子またはビット処理演算子のある式は NULL と評価されます。たとえば 1 + column1 は、column1 が NULL であれば NULL と評価されます。「算術演算子」『SQL Anywhere サーバ - SQL リファレンス』と「ビット処理演算子」『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 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。1 つの文で複数の論理演算子が使用されている場合、通常は、AND 演算子が OR 演算子の前に評価されます。実行順序はカッコを使用して変更できます。

AND の使用

AND 演算子は 2 つ以上の条件を結合し、それらの条件のすべてが true である場合にだけ、結果を返します。たとえば次のクエリは、連絡先の姓が Purcell で、名前が Beth のローだけを検出します。

```
SELECT *
  FROM Contacts
  WHERE GivenName = 'Beth'
    AND Surname = 'Purcell';
```

OR の使用

OR 演算子は 2 つ以上の条件を結合し、それらの条件のうち、**いずれかが** true の場合に、結果を返します。次のクエリは、GivenName カラムの中で Elizabeth の別名を含むローを検索します。

```
SELECT *
  FROM Contacts
  WHERE GivenName = 'Beth'
    OR GivenName = 'Liz';
```

NOT の使用

NOT 演算子は、その後に来る式を否定します。次のクエリは、カリフォルニア州以外の連絡先をすべてリストします。

```
SELECT *
  FROM Contacts
  WHERE NOT State = 'CA';
```

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
...	...

注意

- **日付への自動変換**　SQL Anywhere データベース・サーバは BirthDate カラムに日付が入っていることを認識して、'March 13, 1964' の文字列を自動的に日付に変換します。
- **日付の指定方法**　日付を指定するには、さまざまな方法があります。次に例を示します。

```
'March 13, 1964'
'1964/03/13'
'1964-03-13'
```

date_order オプション・データベース・オプションを設定して、クエリに含まれる日付の解釈を設定できます。「date_order 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。

yyyy/mm/dd または *yyyy-mm-dd* フォーマットの日付は、date_order 設定がされているかどうかに関わらず、日付として常に正確に認識されます。

- **その他の比較演算子** SQL Anywhere では、さまざまな比較演算子をサポートしています。「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。次のクエリを実行して、Brown のように聞こえる名前を持つ従業員を検索することができます。

♦ **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。「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

ID	Quantity
700	80
302	75
301	54
600	39
...	...

- **複数のカラム順に順序を設定できる**　次のクエリは、最初にサイズ順 (アルファベット順)、次に名前順にソートします。

```
SELECT ID, Name, Size
  FROM Products
  ORDER BY Size, Name;
```

ID	Name	Size
600	Sweatshirt	Large
601	Sweatshirt	Large
700	Shorts	Medium
301	Tee Shirt	Medium
...	...	...

- **ORDER BY カラムが select リストになくてもよい**　次のクエリは、価格が結果セットになくても、製品を単価順にソートします。

```
SELECT ID, Name, Size
  FROM Products
  ORDER BY UnitPrice;
```

ID	Name	Size
500	Visor	One size fits all
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。「インデックスの使用」 268 ページを参照してください。

集合関数

一部のクエリでは、テーブル内の、個々のローではなくロー・グループのプロパティを反映するデータの内容を検査します。たとえば、顧客が注文した総額の平均値や、各部門で何人の従業員が仕事をしているかなどです。この種のタスクには、「集合」関数と GROUP BY 句を使用します。

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
...	...

GROUPBY 句は SQL Anywhere に対して、もしこれがなければ返されるすべてのローのセットを分割するように指示します。各分割、つまりグループに含まれるすべてのローは、指定のカラムに同じ値を持ちます。ユニークな値ごと、または値セットごとに、1 つだけグループがあります。この場合、各グループに含まれるすべてのローの SalesRepresentative 値が同じになります。

COUNT などの集合関数は、各グループのローに適用されます。したがって、この結果セットには各グループのローの合計数が表示されます。クエリの結果は、各営業担当者の ID 番号が入った 1 つのローで構成されています。各ローには、営業担当者 ID と、その担当者の受注の合計数が入ります。

GROUP BY を使用した場合には、結果テーブルには GROUP BY 句に指定したカラムまたはカラム・セットごとにローが 1 つずつあります。「GROUP BY 句：クエリ結果のグループへの編成」 397 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;
```

指定された ID を持つ従業員についての結果ローごとに、姓がすべて同じであることを SQL Anywhere は保証できないため、「**'Surname' 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。**」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：データ・グループの選択」 402 ページも参照してください。

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。意図する結果を最も明確に記述する構文を使用するのが最善です。通常は、前にある句の不要なローが削除されます。

例

受注数が 55 を超え、かつ ID が 1000 よりも大きいすべての営業担当者をリストするには、次の文を入力します。

```
SELECT SalesRepresentative, COUNT( * )
FROM SalesOrders
WHERE SalesRepresentative > 1000
GROUP BY SalesRepresentative
HAVING count( * ) > 55
ORDER BY SalesRepresentative;
```

次の SQL 文も同じ結果になります。

```
SELECT SalesRepresentative, COUNT( * )
FROM SalesOrders
GROUP BY SalesRepresentative
HAVING count( * ) > 55 AND SalesRepresentative > 1000
ORDER BY SalesRepresentative;
```

SQL Anywhere は、両方の文で同じ結果セットが記述されていることを検出するため、それぞれの文が効率的に実行されます。

全文検索

全文検索を行うと、テーブルのローをスキャンする必要や単語が格納されているカラムを知る必要がなく、データベース内の特定の単語のすべてのインスタンスを簡単に検索できます。全文検索は、「テキスト・インデックス」を使用することで機能します。テキスト・インデックスには、インデックス付けされたカラム内の単語の位置情報が格納されます。テキスト・インデックスを使用して、単語を含むローを検索すると、テーブル内のすべてのローをスキャンするより速くなる可能性があります。それは、通常にインデックスを使用して、特定の値を含むローを検索すると速くなる可能性があるのと同じ理由です。「テキスト・インデックス」 353 ページを参照してください。

全文検索では、CONTAINS 探索条件が使用されます。全文検索は、マッチングの対象がパターン単位ではなく単語単位であるという点で、LIKE、REGEXP、SIMILAR TO などの述部を使用した検索とは異なります。「CONTAINS 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。

全文検索の文字列の比較では、データベースのすべての通常の照合設定が使われます。たとえば、データベースが大文字と小文字を区別しないように設定されている場合、全文検索でも大文字と小文字が区別されません。「照合の知識」 『SQL Anywhere サーバ - データベース管理』を参照してください。

指定がないかぎり、全文検索では、SQL Anywhere でサポートされているすべての国際化機能が使われます。「SQL Anywhere の国際化機能」 『SQL Anywhere サーバ - データベース管理』を参照してください。

中国語、日本語、韓国語 (CJK) データを含むデータベースで全文検索を実行する方法については、http://www.ianywhere.jp/tech/1061814.html にあるホワイトペーパー『SQL Anywhere 11 で中国語、日本語、韓国語の全文検索を実行する』を参照してください。

全文クエリの実行

全文クエリを実行するには、SELECT 文の FROM 句で CONTAINS 句を使用するか、または WHERE 句で CONTAINS 探索条件 (述部) を使用します。どちらも同じローを返しますが、CONTAINS 句は一致するローのスコアも返します。

たとえば、次の 2 つの文は、MarketingInformation テーブルの Description カラムを問い合わせて、Description カラムの値に **cotton** という単語が含まれるローを返します。2 番目の文では、一致するローのスコアも返します。

```
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( Description, 'cotton' );

SELECT *
  FROM MarketingInformation
  CONTAINS ( Description, 'cotton' );
```

「FROM 句」 『SQL Anywhere サーバ - SQL リファレンス』と「CONTAINS 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。

テキスト設定オブジェクト

テキスト設定オブジェクトは、テクスト・インデックスの作成時または再表示時に格納される単語を制御し、全文クエリの解釈方法を制御します。各テキスト設定オブジェクトの設定は、ISYSTEXTCONFIG システム・テーブルにローとして格納されます。

データベース・サーバがテキスト・インデックスを作成または再表示する場合、テキスト・インデックスの作成時に指定されたテキスト設定オブジェクトの設定が使用されます。テキスト・インデックスの作成時にテキスト設定オブジェクトを指定しなかった場合、データベース・サーバは、インデックス付けされるカラムのデータ型に基づいてデフォルトのテキスト設定オブジェクトを 1 つ選択します。SQL Anywhere には、デフォルトのテキスト設定オブジェクトが 2 つあります。「テキスト設定オブジェクトの例」 346 ページを参照してください。

既存のテキスト設定オブジェクトの設定を表示するには、SYSTEXTCONFIG システム・ビューを問い合わせます。「SYSTEXTCONFIG 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。

テキスト設定オブジェクトの設定

SQL Anywhere には、非 NCHAR データに使用する default_char、および default_nchar というデフォルトのテキスト設定オブジェクトが 2 つあります。これらの設定の詳細については、「デフォルトのテキスト設定オブジェクト」 346 ページを参照してください。

次の表は、テキスト設定オブジェクトの設定、インデックス付けされる対象に与える影響、および全文クエリの解釈方法を説明したものです。テキスト設定オブジェクト、およびテキスト・インデックスや全文検索にテキスト設定オブジェクトが与える影響の例については、「テキスト設定オブジェクトの例」 346 ページを参照してください。

- **単語区切りアルゴリズム (TERM BREAKER)**　TERM BREAKER 設定では、文字列を単語に分割するために使用されるアルゴリズムを指定します。単語を格納する GENERIC (デフォルト)、または N-gram を格納する NGRAM から選択できます。「N-gram」とは、長さ *n* の文字のグループです。*n* は MAXIMUM TERM LENGTH の値を表します。

  指定する単語区切りにかかわらず、単語がテキスト・インデックスに挿入されると、データベース・サーバは単語の元の位置情報をテキスト・インデックスに記録します。N-gram の場合、元の単語の位置情報ではなく、N-gram の位置情報が格納されます。

TERM BREAKER のテキスト・インデックスへの影響	TERM BREAKER のクエリ単語への影響
GENERIC テキスト・インデックス　GENERIC テキスト・インデックス (デフォルト) を作成すると、データベース・サーバは、英数字以外の文字に挟まれた英数字の文字グループを単語として処理します。単語が定義されると、単語長の設定値を超える単語やストップリストに含まれる単語は、カウントはされますがテキスト・インデックスには挿入されません。 GENERIC テキスト・インデックスは、NGRAM テキスト・インデックスをパフォーマンスで上回ります。ただし、GENERIC テキスト・インデックスではあいまい検索は実行できません。 **NGRAM テキスト・インデックス**　NGRAM テキスト・インデックスを作成すると、データベース・サーバは、英数字以外の文字に挟まれた英数字の文字グループをすべて単語として処理します。単語が定義されると、データベース・サーバは単語を N-gram に分割します。その際、*n* よりも短い単語、およびストップリストに含まれる N-gram は破棄されます。 たとえば、MAXIMUM TERM LENGTH が 3 の NGRAM テキスト・インデックスでは、'my red table' という文字列はテキスト・インデックスで red tab abl ble という N-gram として表されます。	**GENERIC テキスト・インデックス**　GENERIC テキスト・インデックスを問い合わせると、クエリ文字列内の単語は、インデックス付けされる場合と同じ方法で処理されます。クエリ単語をテキスト・インデックス内の単語と比較することで、マッチングが実行されます。 **NGRAM テキスト・インデックス**　NGRAM テキスト・インデックスを問い合わせると、クエリ文字列内の単語は、インデックス付けされる場合と同じ方法で処理されます。クエリ単語の N-gram をインデックス付けされた単語の N-gram と比較することで、マッチングが実行されます。

- **単語の最小長の設定 (MINIMUM TERM LENGTH)**　MINIMUM TERM LENGTH 設定では、インデックスに挿入される単語、または全文クエリで検索される単語の最小長を文字数で指定します。MINIMUM TERM LENGTH は、NGRAM テキスト・インデックスでは関係ありません。

  MINIMUM TERM LENGTH は、プレフィクス検索に対して特別な影響があります。「プレフィクス検索」 363 ページを参照してください。

  MINIMUM TERM LENGTH の値は 0 より大きい値にする必要があります。MAXIMUM TERM LENGTH より大きい値を設定すると、MAXIMUM TERM LENGTH が MINIMUM TERM LENGTH と等しくなるように自動的に調整されます。

  MINIMUM TERM LENGTH のデフォルト値は、デフォルトのテキスト設定オブジェクトで設定された値 (通常は 1) になります。「デフォルトのテキスト設定オブジェクト」 346 ページを参照してください。

MINIMUM TERM LENGTH のテキスト・インデックスへの影響	MINIMUM TERM LENGTH のクエリ単語への影響
GENERIC テキスト・インデックス　GENERIC テキスト・インデックスでは、MINIMUM TERM LENGTH よりも短い単語をテキスト・インデックスに含めることはできません。 **NGRAM テキスト・インデックス**　NGRAM テキスト・インデックスでは、この設定は無視されます。	**GENERIC テキスト・インデックス**　GENERIC テキスト・インデックスを問い合わせる場合、MINIMUM TERM LENGTH よりも短いクエリ単語はテキスト・インデックスには存在できないため、無視されます。 **NGRAM テキスト・インデックス**　MINIMUM TERM LENGTH 設定は、NGRAM テキスト・インデックスの全文クエリには影響ありません。

- **単語の最大長の設定 (MAXIMUM TERM LENGTH)**　MAXIMUM TERM LENGTH 設定がどのように使用されるかは、単語区切りアルゴリズムによって異なります。

  MAXIMUM TERM LENGTH の値は 60 以下の値にする必要があります。MINIMUM TERM LENGTH より小さい値を設定すると、MINIMUM TERM LENGTH が MAXIMUM TERM LENGTH と等しくなるように自動的に調整されます。

  MAXIMUM TERM LENGTH のデフォルト値は、デフォルトのテキスト設定オブジェクトで設定された値 (通常は 20) になります。「デフォルトのテキスト設定オブジェクト」 346 ページを参照してください。

MAXIMUM TERM LENGTH のテキスト・インデックスへの影響	MAXIMUM TERM LENGTH のクエリ単語への影響
GENERIC テキスト・インデックス　GENERIC テキスト・インデックスでは、MAXIMUM TERM LENGTH はテキスト・インデックスに挿入される単語の最大長を文字数で指定します。 **NGRAM テキスト・インデックス**　NGRAM テキスト・インデックスでは、MAXIMUM TERM LENGTH は単語が分割される N-gram の長さを決定します。MAXIMUM TERM LENGTH の適切な長さの選択は、言語によって異なります。英語の場合の一般的な値は 4 文字または 5 文字で、中国語の場合は 2 文字または 3 文字です。	**GENERIC テキスト・インデックス**　GENERIC テキスト・インデックスでは、MAXIMUM TERM LENGTH よりも長いクエリ単語はテキスト・インデックスには存在できないため、無視されます。 **NGRAM テキスト・インデックス**　NGRAM テキスト・インデックスでは、クエリ単語は長さ *n* の N-gram に分割されます。*n* は MAXIMUM TERM LENGTH と同じ値になります。データベース・サーバは N-gram を使用してテキスト・インデックスを検索します。MAXIMUM TERM LENGTH よりも短い単語はテキスト・インデックスの N-gram と一致しないため、無視されます。

- **ストップリストの設定 (STOPLIST)**　ストップリスト設定は、インデックス付けしてはならない単語を指定します。

  ストップリスト設定のデフォルト値は、デフォルトのテキスト設定オブジェクトの設定 (通常は空の状態) になります。「デフォルトのテキスト設定オブジェクト」 346 ページを参照してください。

STOPLIST のテキスト・インデックスへの影響	STOPLIST のクエリ単語への影響
GENERIC テキスト・インデックス　GENERIC テキスト・インデックスでは、ストップリストに含まれる単語はテキスト・インデックスに挿入されません。 **NGRAM テキスト・インデックス**　GENERIC テキスト・インデックスでは、ストップリストに含まれる単語から構成される N-gram はテキスト・インデックスには含まれません。	**GENERIC テキスト・インデックス**　GENERIC テキスト・インデックスでは、ストップリストに含まれるクエリ単語はテキスト・インデックスには存在できないため、無視されます。 **NGRAM テキスト・インデックス**　ストップリストに含まれる単語は N-gram に分割され、N-gram はストップリストで使用されます。同様に、クエリ単語も N-gram に分割され、ストップリスト内の N-gram と一致するものはテキスト・インデックスには存在できないため、削除されます。

単語をストップリストに加えるかどうかは、慎重に検討してください。特に、アポストロフィやダッシュなどの英数字以外の文字を含む単語はストップリストには含めないでください。このような文字は、単語区切りとして機能します。たとえば、you'll という単語 ('you''ll' と指定する) は、you と ll に分割され、2 つの単語としてストップリストに格納されます。その後、'you' または 'they''ll' のような単語を全文検索する際に悪影響となります。

NGRAM テキスト・インデックスの場合、ストップリストは、実際に指定したストップリストの単語ではなく、N-gram の形式で格納されるため、予期しない結果になる場合があります。たとえば、MAXIMUM TERM LENGTH が 3 の NGRAM テキスト・インデックスの場合、STOPLIST 'there' を指定すると、the her ere の N-gram がストップリストとして格納されます。これは、the、her、ere の N-gram を含むすべての単語に対する問い合せに影響します。

注意
文字列リテラルの指定に関連する制約事項は、ストップリストにも該当します。たとえば、アポストロフィはエスケープする必要があります。文字列リテラルのフォーマットの詳細については、「文字列リテラル」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Samples ディレクトリには、複数の言語用のストップリストをロードするサンプル・コードが含まれています。これらのサンプル・ストップリストは、GENERIC テキスト・インデックスでのみ使用することをおすすめします。Samples ディレクトリの場所については、「サンプル・ディレクトリ」 『SQL Anywhere サーバ - データベース管理』を参照してください。

参照

- 「デフォルトのテキスト設定オブジェクト」 346 ページ
- 「テキスト設定オブジェクトの作成」 343 ページ
- 「テキスト設定オブジェクトの変更」 344 ページ
- 「テキスト設定オブジェクトの例」 346 ページ
- 「あいまい検索」 370 ページ
- 「テキスト・インデックス」 353 ページ
- 「CONTAINS 探索条件」『SQL Anywhere サーバ - SQL リファレンス』
- 「CREATE TEXT CONFIGURATION 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「ALTER TEXT CONFIGURATION 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「DROP TEXT CONFIGURATION 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「SYSTEXTIDX システム・ビュー」『SQL Anywhere サーバ - SQL リファレンス』

テキスト設定オブジェクトの作成

SQL 文を使用してテキスト設定オブジェクトを作成する場合は、別のテキスト設定オブジェクトをテンプレートとして使用します。次に設定を変更し、新しいテキスト設定を使用して独自のテキスト・インデックスを作成します。

Sybase Central でテキスト設定オブジェクトを作成する場合は、**テキスト設定オブジェクト作成ウィザード**によって作成中に設定を変更できます。

♦ **テキスト設定オブジェクトを作成するには、次の手順に従います (SQL の場合)。**

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. CREATE TEXT CONFIGURATION 文を実行します。

   たとえば、次の文は、default_char テキスト設定オブジェクトをテンプレートとして使用して、myTxtConfig というテキスト設定オブジェクトを作成します。

   ```
   CREATE TEXT CONFIGURATION myTxtConfig FROM default_char;
   ```

♦ **テキスト設定オブジェクトを作成するには、次の手順に従います (Sybase Central の場合)。**

1. DBA 権限のあるユーザとしてデータベースに接続します。
2. **[テキスト設定オブジェクト]** を右クリックし、**[新規]** - **[テキスト設定オブジェクト]** を選択します。
3. **テキスト設定オブジェクト作成ウィザード**の指示に従います。

   [テキスト設定オブジェクト] ウィンドウ枠に、新しいテキスト設定オブジェクトが表示されます。

参照

- 「全文検索」 338 ページ
- 「テキスト設定オブジェクトの設定」 339 ページ
- 「テキスト設定オブジェクトの例」 346 ページ
- 「データベースのテキスト設定オブジェクトの表示」 345 ページ
- 「CREATE TEXT CONFIGURATION 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「ALTER TEXT CONFIGURATION 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「デフォルトのテキスト設定オブジェクト」 346 ページ

テキスト設定オブジェクトの変更

テキスト・インデックスは、作成時に使用したテキスト設定オブジェクトに依存しています。したがって、テキスト設定オブジェクトを変更するためには、依存するテキスト・インデックスをトランケートする必要があります。IMMEDIATE REFRESH テキスト・インデックスはトランケートできないため、このテキスト・インデックスを削除してからしかテキスト設定オブジェクトは変更できません。

♦ **テキスト設定オブジェクトを変更するには、次の手順に従います (SQL の場合)。**

1. DBA 権限のあるユーザとして、またはテキスト設定オブジェクトの所有者として、データベースに接続します。
2. ALTER TEXT CONFIGURATION 文を実行します。たとえば、次の文は、テキスト設定オブジェクト myTxtConfig の単語の最小長を変更します。

```
ALTER TEXT CONFIGURATION myTxtConfig
MINIMUM TERM LENGTH 2;
```

♦ **テキスト設定オブジェクトを変更するには、次の手順に従います (Sybase Central の場合)。**

1. DBA 権限のあるユーザとして、またはテキスト設定オブジェクトの所有者として、データベースに接続します。
2. 左ウィンドウ枠で、**[テキスト設定オブジェクト]** をクリックします。
3. テキスト設定オブジェクトを右クリックし、**[プロパティ]**を選択します。
4. テキスト設定オブジェクトのプロパティを編集し、**[OK]**をクリックします。

参照

- 「全文検索」 338 ページ
- 「テキスト設定オブジェクトの設定」 339 ページ
- 「テキスト設定オブジェクトの例」 346 ページ
- 「データベースのテキスト設定オブジェクトの表示」 345 ページ
- 「CREATE TEXT CONFIGURATION 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「ALTER TEXT CONFIGURATION 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「デフォルトのテキスト設定オブジェクト」 346 ページ

データベース・オプションとテキスト設定オブジェクト

テキスト設定オブジェクトが作成されると、date_format、time_format、timestamp_format の各データベース・オプションの現在の設定はテキスト設定オブジェクトに格納されます。このようになるのは、テキスト設定オブジェクトに依存するテキスト・インデックスの作成時および再表示時に、これらの設定が文字列の変換に影響するためです。

設定をテキスト設定オブジェクトに格納することで、依存するテキスト・インデックスに格納されるデータのフォーマットを変えることなく、データベース・オプションの設定を変更できます。

テキスト・インデックスにある日付や時刻を表す文字列のフォーマットを変更するには、次の手順に従います。

1. テキスト・インデックスとテキスト設定オブジェクトを削除します。
2. データベース・オプションを目的のフォーマットに変更します。
3. テキスト設定オブジェクトを作成します。
4. 新しいテキスト設定オブジェクトを使用して、テキスト・インデックスを作成します。

注意
テキスト・インデックスを作成または再表示する場合は、conversion_error オプションを ON に設定する必要があります。

参照

- 「テキスト設定オブジェクトの設定」 339 ページ
- 「date_format オプション [データベース]」 『SQL Anywhere サーバ - データベース管理』
- 「time_format オプション [互換性]」 『SQL Anywhere サーバ - データベース管理』
- 「timestamp_format オプション [互換性]」 『SQL Anywhere サーバ - データベース管理』
- 「conversion_error オプション [互換性]」 『SQL Anywhere サーバ - データベース管理』

データベースのテキスト設定オブジェクトの表示

データベースのテキスト設定オブジェクトに関する情報は、Sybase Central または SQL 文を使用して表示できます。

♦ テキスト設定オブジェクトの設定を表示するには、次の手順に従います (Sybase Central の場合)。

1. DBA 権限のあるユーザとして、またはテキスト設定オブジェクトの所有者として、データベースに接続します。
2. 左ウィンドウ枠で、**[テキスト設定オブジェクト]** をクリックします。
3. テキスト設定オブジェクトをダブルクリックして、設定を表示します。

♦ **テキスト設定オブジェクトの設定を表示するには、次の手順に従います (SQL の場合)。**

1. DBA 権限のあるユーザとして、またはテキスト設定オブジェクトの所有者として、データベースに接続します。
2. 次のように、SYSTEXTCONFIG システム・ビューを問い合わせます。

```
SELECT * FROM SYSTEXTCONFIG;
```

参照

- 「テキスト設定オブジェクトの設定」 339 ページ
- 「SYSTEXTCONFIG システム・ビュー」 『SQL Anywhere サーバ - SQL リファレンス』

テキスト設定オブジェクトの例

テキスト設定オブジェクトの設定、およびそれがテキスト・インデックスの内容やテキスト・インデックスへの問い合わせ結果にどのような影響を与えるかに関する詳細については、「テキスト設定オブジェクトの設定」 339 ページを参照してください。

データベースのすべてのテキスト設定オブジェクト、およびその設定に関するリストについては、SYSTEXTCONFIG システム・ビューを問い合わせてください (例：SELECT * FROM SYSTEXTCONFIG)。「SYSTEXTCONFIG 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。

デフォルトのテキスト設定オブジェクト

SQL Anywhere には、NCHAR データに使用する 「default_nchar」 と非 NCHAR データに使用する 「default_char」 というデフォルトのテキスト設定オブジェクトが 2 つあります。これらの設定は、テキスト設定オブジェクトやテキスト・インデックスを初めて作成する際に作成されます。誤って 1 つを削除してしまった場合は、テキスト設定オブジェクトやテキスト・インデックスを次回作成する際に再作成されます。

インストール時における default_char と default_nchar の設定を以下の表に示します。これらの設定は、ほとんどの文字ベースの言語で最適であるという理由で選択されています。デフォルトのテキスト設定オブジェクトの設定を変更しないことを強くおすすめします。

設定	インストールされる値
TERM BREAKER	0 (GENERIC)
MINIMUM TERM LENGTH	1
MAXIMUM TERM LENGTH	20
STOPLIST	(空)

デフォルトのテキスト設定オブジェクトを削除すると、テキスト・インデックスやテキスト設定オブジェクトを次回作成する際に自動的に作成されます。「DROP TEXT CONFIGURATION 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

テキスト設定オブジェクトの例

次の表は、さまざまなテキスト設定オブジェクトの設定、インデックス付けされる対象に与える影響、および全文クエリ文字列の解釈方法を示したものです。すべての例で、文字列 'I'm not sure I understand' を使用しています。

設定	インデックス付けされる単語	クエリ解釈
TERM BREAKER GENERIC MINIMUM TERM LENGTH 1 MAXIMUM TERM LENGTH 20 STOPLIST ''	I m not sure I understand	"I m" AND not AND sure AND I AND understand'
TERM BREAKER GENERIC MINIMUM TERM LENGTH 2 MAXIMUM TERM LENGTH 20 STOPLIST 'not and'	sure understand	'sure AND understand'.
TERM BREAKER NGRAM MAXIMUM TERM LENGTH 3 STOPLIST 'not and'	sur ure und nde der ers rst sta tan	'sur AND ure AND und AND nde AND der AND ers AND rst AND sta AND tan'. あいまい検索の場合： 'sur OR ure OR und OR nde OR der OR ers OR rst OR sta OR tan'
TERM BREAKER GENERIC MINIMUM TERM LENGTH 1 MAXIMUM TERM LENGTH 20 STOPLIST 'not and'	I m sure I understand	'"I m" AND sure AND I AND understand'.

設定	インデックス付けされる単語	クエリ解釈
TERM BREAKER NGRAM MAXIMUM TERM LENGTH 20 STOPLIST 'not and'	20 文字以上の単語がないため、インデックス付けされるものはありません。 これは、MAXIMUM TERM LENGTH の GENERIC テキスト・インデックスへの影響と NGRAM テキスト・インデックスへの影響が異なることを示しています。NGRAM テキスト・インデックスの場合、MAXIMUM TERM LENGTH は、テキスト・インデックスに挿入される N-gram の長さを設定します。	クエリ文字列から 20 文字の N-gram を構成できないため、検索を実行すると空の結果セットが返されます。

文字列解釈の例

次の表は、テキスト設定オブジェクトの文字列の設定がどのように解釈されるかについて、例を示したものです。

解釈される文字列の列にあるカッコ内の数字は、各単語の位置情報を表しています。この数字は、説明の目的でマニュアルに記載されています。格納される実際の単語にはカッコ内の数字は含まれません。

設定	文字列	解釈される文字列
TERM BREAKER GENERIC MINIMUM TERM LENGTH 3 MAXIMUM TERM LENGTH 20	'w*'	'"w*(1)"'
	'we*'	'"we*(1)"'
	'wea*'	'"wea*(1)"'
	'we* -the'	'"we*(1)" -"the(1)"'
	'we* the'	"we*(1)" & "the(1)"'
	'for* | wonderl*'	'"for*(1)" | "wonderl*(1)"'
	'wonderlandwonderlandwonderland*'	''
	'"tr* weather"'	'"weather(1)"'
	'"tr* the weather"'	'"the(1) weather(2)"'
	'"wonderlandwonderlandwonderland* wonderland"'	'"wonderland(1)"'

設定	文字列	解釈される文字列
	'"wonderlandwonderlandwonderland* weather"'	'"weather(1)"'
	'"the_wonderlandwonderlandwonderland* weather"'	'"the(1) weather(3)"'
	'the_wonderlandwonderlandwonderland* weather'	'"the(1)" & "weather(1)"'
	'"light_a* the end" & tunnel'	'"light(1) the(3) end(4)" & "tunnel(1)"'
	light_b* the end" & tunnel'	'"light(1) the(3) end(4)" & "tunnel(1)"'
	'"light_at_b* end"'	'"light(1) end(4)"'
	'and-te*'	'"and(1) te*(2)"'
	'a_long_and_t* & journey'	'"long(2) and(3) t*(4)" & "journey(1)"'
TERM BREAKER NGRAM MAXIMUM TERM LENGTH 3	'w*'	'"w*(1)"'
	'we*'	'"we*(1)"'
	'wea*	'"wea(1)"'
	'we* -the'	'"we*(1)" -"the(1)"'
	'we* the'	'"we*(1)" & "the(1)"'
	'for | la*'	'"for(1)" | "la*(1)"'
	'weath*'	'"wea(1) eat(2) ath(3)"'
	'"ful weat*"'	'"ful(1) wea(2) eat(3)"'
	'"wo* la*"'	'"wo*(1)" & "la*(2)"'
	'"la* won* "'	'"la*(1)" & "won(2)"'

設定	文字列	解釈される文字列
	'"won* weat*"'	'"won(1)" & "wea(2) eat(3)"'
	'"won* weat"'	'"won(1)" & "wea(2) eat(3)"'
	'"wo* weat*"'	'"wo*(1)" & "wea(2) eat(3)"'
	'"weat* wo* "'	'"wea(1) eat(2)" & "wo*(3)"'
	'"wo* weat"'	'"wo*(1)" & "wea(2) eat(3)"'
	'"weat wo* "'	'"wea(1) eat(2) wo*(3)"'
	'w* NEAR[1] f*'	'"w*(1)" & "f*(1)"'
	'weat* NEAR[1] f*'	"wea(1) eat(2)" & "f*(1)"'
	'f* NEAR[1] weat*'	'"f*(1)" & "wea(1) eat(2)"'
	'weat NEAR[1] f*'	'"wea(1) eat(2)" & "f*(1)"'
	'f* NEAR[1] weat'	'"f*(1)" & "wea(1) eat(2)"'
	'for NEAR[1] weat*'	'"for(1)" & "wea(1) eat(2)"'
	'weat* NEAR[1] for'	'"wea(1) eat(2)" & "for(1)"'
	'and_tedi*'	'"and(1) ted(2) edi(3)"'
	'and-t*'	'"and(1) t*(2)"'
	'"and_tedi*"'	'"and(1) ted(2) edi(3)"'
	'"and-t*"'	'"and(1) t*(2)"'

設定	文字列	解釈される文字列
	'"ligh* at_the_end of_the tun* nel"'	'"lig(1) igh(2)" & ("the(4) end(5) the(7) tun(8)" & "nel(9)")'
	'"ligh* at_the_end_of_the_tun* nel"'	'"lig(1) igh(2)" & ("the(4) end(5) the(7) tun(8)" & "nel(9)")'
	'"at_the_end of_the tun* ligh* nel"'	'"the(2) end(3) the(5) tun(6)" & ("lig(7) igh(8)" & "nel(9)")'
	'l* NEAR[1] and_t*'	"l*(1)" & "and(1) t*(2)"'
	'long NEAR[1] and_t*'	'"lon(1) ong(2)" & "and(1) t*(2)"'
	'end NEAR[3] tunne*'	'"end(1)" & "tun(1) unn(2) nne(3)"'
TERM BREAKER NGRAM MAXIMUM TERM LENGTH 3 SKIPPED TOKENS IN TABLE AND IN QUERIES	'"cat in a hat"'	'"cat(1) hat(4)"'
	'"cat in_a hat"'	'"cat(1) hat(4)"'
	'"cat_in_a_hat"'	'"cat(1) hat(4)"'
	'"cat_in a_hat"'	'"cat(1) hat(4)"'
	'cat in a hat'	'"cat(1)" & "hat(1)"'
	'cat in_a hat'	'"cat(1)" & "hat(1)"'
	'"ice hat"'	'"ice(1) hat(2)"'
	'ice NEAR[1] hat'	'"ice(1)" NEAR[1] "hat(1)"'
	'ear NEAR[2] hat'	'"ear(1)" NEAR[2] "hat(1)"'
	'"ear a hat"'	'"ear(1) hat(3)"'
	'"cat hat"'	'"cat(1) hat(2)"'

設定	文字列	解釈される文字列
	'cat NEAR[1] hat'	'"cat(1)" NEAR[1] "hat(1)"'
	'ear NEAR[1] hat'	'"ear(1)" NEAR[1] "hat(1)"'
	'"ear hat"'	'"ear(1) hat(2)"'
	'"wear a a hat"'	'"wea(1) ear(2) hat(5)"'

参照

- 「sa_char_terms システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』
- 「sa_nchar_terms システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』

テキスト・インデックス

全文検索を実行すると、テーブル・ローではなく、「テキスト・インデックス」が検索されます。したがって、全文検索を実行する前に、検索するカラムにテキスト・インデックスを作成する必要があります。テキスト・インデックスには、インデックス付けされたカラム内の単語の位置情報が格納されます。テキスト・インデックスを使用したクエリは、テーブル内のすべての値をスキャンする必要があるクエリよりも高速になる可能性があります。

テキスト・インデックスを作成する場合、テキスト・インデックスの作成と再表示に使用される「テキスト設定オブジェクト」を指定できます。テキスト設定オブジェクトには、インデックスの構築方法に影響する設定が含まれます。テキスト設定オブジェクトを指定しない場合、データベース・サーバはデフォルトの設定オブジェクトを使用します。「テキスト設定オブジェクト」 339 ページを参照してください。

テキスト・インデックスの「再表示タイプ」も指定できます。再表示タイプによって、テキスト・インデックスの再表示の頻度が決まります。最後に再表示されたテキスト・インデックスが最も正確な結果を返します。ただし、再表示には時間がかかるため、パフォーマンスに影響する可能性があります。たとえば、インデックス付けされたテーブルを頻繁に更新する場合、基本となるデータが変更されるたびにテキスト・インデックスを再表示するように設定すると、パフォーマンスに影響する可能性があります。「テキスト・インデックスの再表示タイプ」 353 ページを参照してください。

既存のテキスト・インデックスの設定を表示するには、sa_text_index_stats システム・プロシージャを使用します。「sa_text_index_stats システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

テキスト・インデックスの再表示タイプ

テキスト・インデックスを作成する場合、「再表示タイプ」も選択する必要があります。テキスト・インデックスでは、即時、自動、手動の 3 つの再表示タイプがサポートされています。再表示タイプは、テキスト・インデックスの作成時に定義します。即時テキスト・インデックスの場合を除き、再表示タイプはテキスト・インデックスの作成後に変更できます。

再表示タイプの設定方法については、「CREATE TEXT INDEX 文」 『SQL Anywhere サーバ - SQL リファレンス』と「ALTER TEXT INDEX 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **即時再表示 (デフォルト)** 即時再表示のテキスト・インデックスは、基本となるテーブルのデータに変更があると再表示されます。データを常に最新の状態に保つ必要がある場合、またはインデックス付けされたカラムが比較的短い場合においてのみ使用することをおすすめします。

  テキスト・インデックスのデフォルトの再表示タイプは即時です。

  自動再表示または手動再表示のテキスト・インデックスを、即時再表示のテキスト・インデックスに変更することはできません。変更したい場合は、該当するテキスト・インデックスを削除して、即時再表示のテキスト・インデックスとして再作成する必要があります。

即時再表示のテキスト・インデックスはすべての独立性レベルをサポートします。即時再表示のテキスト・インデックスは、作成時にデータが設定され、この初期再表示中に、テーブルに排他ロックがかけられます。

- **自動再表示**　自動再表示のテキスト・インデックスは、指定された間隔で自動的に再表示されます。データがある程度古くなってもかまわない場合に使用することをおすすめします。古いインデックスを問い合わせると、一致するローのうち、最後に再表示されてから変更されていないものが返されます。したがって、最後の再表示以降に挿入、削除、更新されたローはクエリで返されません。

  自動再表示のテキスト・インデックスは、次のどちらかの状態が該当する場合は、指定された間隔よりも頻繁に再表示される可能性があります。それは、最後の再表示からの経過時間が再表示間隔よりも長い場合、または保留中のすべてのローの合計長 (sa_text_index_stats システム・プロシージャによって返される pending_length) が合計インデックス・サイズ (sa_text_index_stats によって返される doc_length) の 20 % を超える場合、のどちらかです。

  自動再表示のテキスト・インデックスは、独立性レベル 0 を使用して再表示されます。

  自動再表示のテキスト・インデックスには、作成時にはデータが含まれていないため、最初の再表示が行われるまで利用できません。最初の再表示は通常、テキスト・インデックスが作成されてから 1 分以内に行われます。REFRESH TEXT INDEX 文を使用することで、自動再表示のテキスト・インデックスを手動で再表示することもできます。

  自動再表示のテキスト・インデックスは、dbunload で -g オプションを指定しないかぎり、再ロード時に再表示されません。「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。

- **手動再表示**　手動再表示のテキスト・インデックスは、ユーザの実行によってのみ再表示されます。基本となるテーブルのデータがめったに変更されない場合、データの古さの程度が大きくても許容できる場合、またはイベントや条件を満たした後に再表示したい場合に使用することをおすすめします。古いインデックスを問い合わせると、一致するローのうち、最後に再表示されてから変更されていないものが返されます。したがって、最後の再表示以降に挿入、削除、更新されたローはクエリで返されません。

  手動再表示のテキスト・インデックスには、独自の再表示方針を定義できます。たとえば、引数として渡される再表示間隔と、自動再表示のテキスト・インデックスに使用されるものと同様のルールを使用して、すべての手動再表示のテキスト・インデックスを再表示するプロシージャを使用できます。

  次の例で、*text-index-name*、*table-owner*、*table-name* を適宜置き換えてください。

```
CREATE PROCEDURE refresh_manual_text_indexes(
  refresh_interval UNSIGNED INT )
BEGIN
 FOR lp1 AS c1 CURSOR FOR
  SELECT ts.*
  FROM SYS.SYSTEXTIDX ti JOIN sa_text_index_stats( ) ts
  ON ( ts.index_id = ti.index_id )
  WHERE ti.refresh_type = 1 -- manual refresh indexes only
 DO
  BEGIN
   IF last_refresh IS null
   OR cast(pending_length as float) / (
     IF doc_length=0 THEN NULL ELSE doc_length ENDIF) > 0.2
   OR DATEDIFF( MINUTE, CURRENT TIMESTAMP, last_refresh )
```



```
      > refresh_interval THEN
     EXECUTE IMMEDIATE 'REFRESH TEXT INDEX ' || text-index-name || ' ON "'
     || table-owner || '"."' || table-name || '"';
    END IF;
   END;
  END FOR;
END;
```

いつでも、sa_text_index_stats システム・プロシージャを使用して、再表示が必要かどうか、再表示を完全再構築にするか、増分更新にするかを決定できます。「sa_text_index_stats システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

手動再表示のテキスト・インデックスには、作成時にはデータが含まれていないため、再表示するまで利用できません。手動再表示のテキスト・インデックスを再表示するには、REFRESH TEXT INDEX 文を使用します。

手動再表示のテキスト・インデックスは、dbunload で -g オプションを指定しないかぎり、再ロード時に再表示されません。「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。

参照

- 「全文検索」 338 ページ
- 「テキスト・インデックスの作成」 355 ページ
- 「テキスト設定オブジェクトの設定」 339 ページ
- 「sa_text_index_stats システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』
- 「REFRESH TEXT INDEX 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「isolation_level オプション [データベース] [互換性]」 『SQL Anywhere サーバ - データベース管理』

テキスト・インデックスの作成

任意のタイプのカラムのテキスト・インデックスを作成できます。VARCHAR または NVARCHAR タイプ以外のカラムは、インデックスの作成時に文字列に変換されます。「データ型変換」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

テキスト・インデックスはディスク領域を消費し、再表示を必要とします。作成する場合は、クエリをサポートするために必要となるカラムのみを対象にしてください。

マテリアライズド・ビュー、通常のビュー、テンポラリ・テーブルに対してテキスト・インデックスを作成することはできません。

予期しない結果が生じる可能性があるので、1 つのカラムを参照するテキスト・インデックスを 2 つ以上作成しないでください。

♦ **テキスト・インデックスを作成するには、次の手順に従います (Sybase Central の場合)。**

1. DBA 権限のあるユーザとして、またはテキスト・インデックスが作成され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. **[テキスト・インデックス]** タブをクリックします。

3. **[ファイル] - [新規] - [テキスト・インデックス]** を選択します。
4. **インデックス作成ウィザード**の指示に従います。

   [テキスト・インデックス] タブに新しいテキスト・インデックスが表示されます。また、**[テキスト・インデックス]** フォルダにも表示されます。
5. 即時再表示のテキスト・インデックスを作成した場合は、データが自動的に設定されます。その他の再表示タイプの場合は、テキスト・インデックスを右クリックし、**[データの再表示]** を選択することで、テキスト・インデックスを再表示する必要があります。

♦ **新しいテキスト・インデックスを作成するには、次の手順に従います (SQL の場合)。**

1. DBA 権限のあるユーザとして、またはテキスト・インデックスが作成され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 TEXT INDEX 文を実行します。「CREATE TEXT INDEX 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
3. 即時再表示のテキスト・インデックスを作成した場合は、データが自動的に設定されます。その他の再表示タイプの場合は、REFRESH TEXT INDEX 文を実行して、テキスト・インデックスを再表示する必要があります。「REFRESH TEXT INDEX 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

参照

- 「テキスト・インデックスの再表示タイプ」 353 ページ
- 「全文検索」 338 ページ

テキスト・インデックスの再表示

再表示できるテキスト・インデックスは、AUTO REFRESH および MANUAL REFRESH として定義されているテキスト・インデックスだけです。

♦ **テキスト・インデックスを再表示するには、次の手順に従います (Sybase Central の場合)。**

1. DBA 権限のあるユーザとして、またはテキスト・インデックスが作成され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、**[テキスト・インデックス]** をクリックします。
3. テキスト・インデックスを右クリックし、**[データの再表示]**を選択します。
4. 再表示の独立性レベルを選択し、**[OK]**をクリックします。

♦ **テキスト・インデックスを再表示するには、次の手順に従います (SQL の場合)。**

1. DBA 権限のあるユーザとして、またはテキスト・インデックスが作成され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. REFRESH TEXT INDEX 文を実行します。

たとえば、次の文は、デモ・データベースの Products テーブルの Description カラムについての txt_index_manual というテキスト・インデックスを再表示します。

```
REFRESH TEXT INDEX txt_index_manual ON Products
  WITH ISOLATION LEVEL 0;
```

「REFRESH TEXT INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

参照

- 「テキスト・インデックスの再表示タイプ」 353 ページ
- 「全文検索」 338 ページ

テキスト・インデックスの変更の概要

テキスト・インデックスの次の特徴を変更できます。

- **再表示タイプ** 再表示タイプを AUTO REFRESH から MANUAL REFRESH に、またはその逆に変更することができます。再表示タイプを変更するには、ALTER TEXT INDEX 文の REFRESH 句を使用します。「ALTER TEXT INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

  テキスト・インデックスを IMMEDIATE REFRESH に、または IMMEDIATE REFRESH から変更することはできません。この変更を実行するには、テキスト・インデックスを削除して、再作成する必要があります。「DROP TEXT INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』と「CREATE TEXT INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **名前** テキスト・インデックスの名前を変更するには、ALTER TEXT INDEX 文の RENAME 句を使用します。「ALTER TEXT INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **コンテンツ** カラム・リストを除き、インデックス付けの対象を制御する設定は、テキスト設定オブジェクトに格納されます。インデックス付けの対象を変更する場合は、テキスト・インデックスが参照するテキスト設定オブジェクトを変更します。依存するテキスト・インデックスをトランケートしてからテキスト設定オブジェクトを変更し、その後テキスト・インデックスを再表示する必要があります。即時再表示のテキスト・インデックスの場合、テキスト・インデックスを削除し、テキスト設定オブジェクトを変更した後にテキスト・インデックスを再作成する必要があります。

  次の項を参照してください。

  - 「テキスト設定オブジェクト」 339 ページ
  - 「TRUNCATE TEXT INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』
  - 「ALTER TEXT INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』
  - 「REFRESH TEXT INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』
  - 「sa_refresh_text_indexes システム・プロシージャ」『SQL Anywhere サーバ - SQL リファレンス』

別のテキスト設定オブジェクトを参照するように、テキスト・インデックスを変更することはできません。テキスト・インデックスで別のテキスト設定オブジェクトを参照させるには、テキスト・インデックスを削除して、新しいテキスト設定オブジェクトを指定して再作成します。

テキスト・インデックスの変更

テキスト・インデックスの名前または再表示タイプを変更できます。

♦ テキスト・インデックスの再表示タイプを変更するには、次の手順に従います (Sybase Central の場合)。

1. DBA 権限のあるユーザとして、またはテキスト・インデックスが作成され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、**[テキスト・インデックス]** をクリックします。
3. テキスト・インデックスを右クリックし、**[プロパティ]** を選択します。
4. テキスト・インデックスのプロパティを編集し、**[OK]**をクリックします。

♦ テキスト・インデックスの再表示タイプを変更するには、次の手順に従います (SQL の場合)。

1. DBA 権限のあるユーザとして、またはテキスト・インデックスの所有者として、データベースに接続します。
2. ALTER TEXT INDEX 文を実行します。「ALTER TEXT INDEX 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

♦ テキスト・インデックスの名前を変更するには、次の手順に従います (Sybase Central の場合)。

1. DBA 権限のあるユーザとして、またはテキスト・インデックスが作成され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、**[テキスト・インデックス]** をクリックします。
3. テキスト・インデックスを右クリックし、**[プロパティ]** を選択します。
4. **[一般]** タブをクリックし、テキスト・インデックスの新しい名前を入力します。
5. **[OK]** をクリックします。

♦ テキスト・インデックスの名前を変更するには、次の手順に従います (SQL の場合)。

1. DBA 権限のあるユーザとして、またはテキスト・インデックスの所有者として、データベースに接続します。
2. ALTER TEXT INDEX 文を実行します。「ALTER TEXT INDEX 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

参照

- 「テキスト・インデックス」 353 ページ
- 「全文検索」 338 ページ

データベースのテキスト・インデックスの表示

データベースのテキスト・インデックスに関する情報は、Sybase Central または SQL 文を使用して表示できます。

♦ データベースのテキスト・インデックスを表示するには、次の手順に従います (Sybase Central の場合)。

1. DBA 権限のあるユーザとして、またはテキスト・インデックスの所有者として、データベースに接続します。
2. 左ウィンドウ枠で、**[テキスト・インデックス]** をクリックします。
3. テキスト・インデックス内の単語を表示するには、左ウィンドウ枠でテキスト・インデックスをダブルクリックし、右ウィンドウ枠で **[ボキャブラリ]** タブを選択します。
4. インデックスが参照するテキスト設定オブジェクトや再表示タイプなど、テキスト・インデックスの設定を表示するには、テキスト・インデックスを右クリックし、**[プロパティ]** を選択します。

♦ データベースのテキスト・インデックスを表示するには、次の手順に従います (SQL の場合)。

1. DBA 権限のあるユーザとして、またはテキスト・インデックスの所有者として、データベースに接続します。
2. 次のように、sa_text_index_stats システム・プロシージャを呼び出します。

```
CALL sa_text_index_stats( );
```

参照

- 「sa_text_index_stats システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』

全文検索のタイプ

全文検索を使用して、「単語」、「フレーズ」 (単語のシーケンス)、または「プレフィクス」を検索できます。複数の単語、フレーズ、またはプレフィクスをブール式に組み合わせることもできます。または、その式が近接検索で互いに近くに出現する必要があります。

全文検索を実行するには、SELECT 文の WHERE 句または FROM 句で CONTAINS 句を使用します。さらに、IF 探索条件の一部として全文検索を実行することもできます (SELECT IF CONTAINS... など)。

単語とフレーズの検索

単語のリストの全文検索を実行する場合は、フレーズ内の単語でないかぎり、単語の順序は重要ではありません。単語がフレーズ内にある場合、データベース・サーバは、出現する順序と相対位置が指定されたものと厳密に一致する単語を検索します。

単語検索やフレーズ検索を実行する際、単語長の設定を超過するため、またはストップリストに含まれているために単語がクエリから削除された場合、予期しない数のローが返される可能性があります。これは、単語をクエリから削除することは探索条件を変更することと同じであるためです。たとえば、'"grown cotton"' というフレーズを検索し、grown がストップリストに入っている場合、cotton を含むインデックス付けされたローがすべて返されます。

CONTAINS 句の文法でキーワードと見なされる単語は、フレーズに含まれている場合のみ検索できます。

単語検索

デモ・データベースでは、MarketingInformation テーブルの Description カラムに MarketingTextIndex というテキスト・インデックスが作成されています。次の文は、MarketingInformation テーブルの Description カラムを問い合わせて、Description カラムの値に **cotton** という単語が含まれるローを返します。

```
SELECT ID, Description
  FROM MarketingInformation
  WHERE CONTAINS ( Description, 'cotton' );
```

ID	Description
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 construction. Shields against sun and precipitation.cotton Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>

ID	Description
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% 「cotton」/20% polyester blend makes it easy to keep them clean. </body></html>
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying 「cotton」 shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>

次の例は、MarketingInformation テーブルを問い合わせて、各ローについて、Description カラムの値に **cotton** という単語が含まれるかどうかを示す 1 つの値を返します。

```
SELECT ID, IF CONTAINS ( Description, 'cotton' )
    THEN 1
    ELSE 0
    ENDIF AS Results
  FROM MarketingInformation;
```

ID	Results
901	0
902	0
903	0
904	0
905	0
906	1
907	0
908	1
909	1

ID	Results
910	1

次の例は、MarketingInformation テーブルを問い合わせて、Description カラムに **cotton** という単語が含まれる項目を検索し、一致ごとのスコアを表示します。

```
SELECT ID, ct.score, Description
  FROM MarketingInformation CONTAINS ( MarketingInformation.Description, 'cotton' ) as ct
  ORDER BY ct.score DESC;
```

ID	score	Description
908	0.9461597363521859	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>
910	0.9244136988525732	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying 「cotton」 shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>
906	0.9134171046194403	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>
909	0.8856420222728282	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% 「cotton」/20% polyester blend makes it easy to keep them clean. </body></html>

クエリの FROM 句で CONTAINS を使用した場合のスコア結果の詳細については、「全文検索結果のスコア付け」 372 ページを参照してください。

フレーズ検索

フレーズの全文検索を実行する場合、フレーズを二重引用符で囲みます。指定の順序と相対位置の単語が含まれる場合、カラムは一致します。

単語がフレーズ内に含まれている場合を除き、AND や FUZZY などの CONTAINS キーワードを、検索する単語として指定することはできません (単一単語のフレーズは検索可能)。たとえば、NOT は CONTAINS キーワードですが、次の文は実行できます。CONTAINS キーワードと特殊文字のリストについては、「CONTAINS 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。

```
SELECT * FROM table-name CONTAINS ( Remarks, '"NOT"' );
```

フレーズの中で使用される特殊文字は、特殊文字として解釈されません (アスタリスクを除く)。

フレーズは、近接検索の引数としては使用できません。

次の文は、MarketingInformation テーブルの Description カラムを問い合わせて、"grown cotton" というフレーズを検索し、一致ごとのスコアを表示します。

```
SELECT ID, ct.score, Description
  FROM MarketingInformation CONTAINS ( MarketingInformation.Description, '"grown cotton"' ) as ct
  ORDER BY ct.score DESC;
```

ID	score	Description
908	1.6619019465461564	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>
906	1.6043904700786786	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>

プレフィクス検索

全文検索では、単語の先頭部分を検索できる機能があります。これは「プレフィクス検索」と呼ばれます。プレフィクス検索を実行するには、検索するプレフィクスを指定した後に、アスタリスクを付けます。これは「プレフィクス単語」と呼ばれます。

CONTAINS 句のキーワードは、フレーズに含まれていないかぎり、プレフィクス検索では使用できません。CONTAINS キーワードのリストについては、「CONTAINS 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。

クエリ文字列に複数のプレフィクス単語 (フレーズ内のプレフィクス単語も含む) を指定することもできます (たとえば、'"shi* fab"' など)。

プレフィクス検索の構文に関する完全な制約事項については、「アスタリスク (*) を使用できる構文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

次の例は、MarketingInformation テーブルを問い合わせて、プレフィクス **shi** で始まる項目を検索します。

```
SELECT ID, ct.score, Description
  FROM MarketingInformation CONTAINS ( MarketingInformation.Description, 'shi*' ) AS ct
  ORDER BY ct.score DESC;
```

ID	score	Description
906	2.295363835537917	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. 「Shi」elds against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>
901	1.6883275743936228	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee 「Shi」rt</title></head><body lang=EN-US> We've improved the design of this perennial favorite. A sleek and technical 「shi」rt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>
903	1.6336529491832605	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee 「Shi」rt</title></head><body lang=EN-US> A sporty, casual 「shi」rt made of recycled water bottles. It will serve you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>
902	1.6181703448678983	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee 「Shi」rt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical 「shi」rt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>

テキスト・インデックスで shield という単語が出現する頻度は shirt よりも低いため、ID 906 のスコアが最も高くなります。

GENERIC テキスト・インデックスへのプレフィクス検索

GENERIC テキスト・インデックスでは、プレフィクス検索は次のように動作します。

- プレフィクス単語が MAXIMUM TERM LENGTH よりも長い場合、MAXIMUM TERM LENGTH よりも長い単語はテキスト・インデックスには存在できないため、クエリ文字列から削除されます。したがって、MAXIMUM TERM LENGTH が 3 のテキスト・インデックスで 'red appl*' を検索することは、'red' を検索することと同義になります。
- プレフィクス単語が MINIMUM TERM LENGTH よりも短く、フレーズ検索の一部でもない場合、プレフィクス検索は通常通り処理されます。したがって、MINIMUM TERM LENGTH が 5 の GENERIC テキスト・インデックスで、'macintosh a*' を検索すると、macintosh と a で始まる 5 文字以上の任意の単語を含むインデックス付けされたローが返されます。
- プレフィクス単語が MINIMUM TERM LENGTH よりも短いものの、フレーズ検索の一部になっている場合は、プレフィクス単語はクエリから削除されます。したがって、MINIMUM TERM LENGTH が 5 の GENERIC テキスト・インデックスで、'"macintosh appl* turnover"' を検索することは、「macintosh 任意の単語 turnover」を検索することと同義になります。"macintosh turnover" を含むローは検索されません。macintosh と turnover の間に単語が必要になります。

NGRAM テキスト・インデックスへのプレフィクス検索

NGRAM テキスト・インデックスの場合、NGRAM テキスト・インデックスには N-gram しか含まれておらず、単語の先頭に関する情報は含まれていないため、プレフィクス検索を実行すると予期しない結果が返される可能性があります。また、クエリ単語は N-gram に分割され、検索はクエリ単語ではなく N-gram を使用して実行されます。そのため、次のような動作に注意する必要があります。

- プレフィクス単語が N-gram の長さ (MAXIMUM TERM LENGTH) よりも短い場合、クエリはそのプレフィクス単語で始まる N-gram を含むすべてのインデックス付けされたローを返します。たとえば、3-gram のテキスト・インデックスの場合、'ea*' を検索すると、ea で始まる N-gram を含むすべてのインデックス付けされたローが返されます。したがって、weather と fear という単語がインデックス付けされている場合、これらの N-gram にはそれぞれ eat と ear が含まれているため、ローは一致すると見なされます。
- プレフィクス単語が N-gram の長さよりも長く、フレーズの一部ではなく、近接検索の引数ではない場合、プレフィクス単語は N-gram フレーズに変換され、アスタリスクは削除されます。たとえば、3-gram のテキスト・インデックスの場合、'purple blac*' を検索することは、'"pur urp rpl ple" AND "bla lac"' を検索することと同義になります。
- フレーズの場合は、次のような動作も発生します。
  - フレーズ内にプレフィクス単語以外の単語がない場合、そのプレフィクス単語は N-gram フレーズに変換され、アスタリスクは削除されます。たとえば、3-gram のテキスト・インデックスの場合、'"purpl*"' を検索することは、'"pur urp rpl"' を検索することと同義になります。
  - プレフィクス単語がフレーズの末尾に位置する場合は、アスタリスクは削除され、すべての単語は N-gram のフレーズに変換されます。たとえば、3-gram のテキスト・インデック

スの場合、'"purple blac*"' を検索することは、'"pur urp rpl ple bla lac"' を検索することと同義になります。

- ○ プレフィクス単語がフレーズの末尾に位置していない場合、フレーズは複数のフレーズに分割され、AND で結合されます。たとえば、3-gram のテキスト・インデックスの場合、'"purp* blac*"' を検索することは、'"pur urp" AND "bla lac"' を検索することと同義になります。

- プレフィクス単語が近接検索の引数の場合、近接検索は AND に変換されます。たとえば、3-gram のテキスト・インデックスの場合、'red NEAR[1] appl*' を検索することは、'red AND "app ppl"' を検索することと同義になります。

参照

- 「テキスト・インデックス」 353 ページ
- 「CONTAINS 探索条件」 『SQL Anywhere サーバ - SQL リファレンス』

近接検索

全文検索では、単一のカラム内で互いに近接する単語を検索できる機能があります。これは「近接検索」と呼ばれます。近接検索を実行するには、2 つの単語を指定して、間に NEAR キーワードまたはチルダ (~) を挿入します。

NEAR キーワードを使用して、整数引数を指定し、最大距離を指定できます。たとえば、*term1* NEAR[5] *term2* は *term2* から 5 語以内に出現する *term1* のインスタンスを検索します。単語の順序に意味はなく、'term1 NEAR term2' は 'term2 NEAR term1' と同義です。

距離を指定しない場合、データベース・サーバはデフォルトの距離として 10 を使用します。

NEAR キーワードの代わりに、チルダ (~) を指定することもできます。たとえば 'term1 ~ term2' のようになります。ただし、チルダ形式を使用する場合は距離を指定できず、デフォルトの 10 が適用されます。

近接検索の引数としてフレーズを指定することはできません。

近接検索でプレフィクス単語を引数として指定すると、近接検索は AND 式に変換されます。たとえば、3-gram のテキスト・インデックスの場合、'red NEAR[1] appl*' を検索することは、'red AND "app ppl"' を検索することと同義になります。これは近接検索ではなくなったため、CONTAINS 句で複数のカラムが指定されている場合に、検索が 1 つのカラムに制限されることはありません。

例

MarketingInformation テーブルの Description カラムで、単語 skin から 10 語以内の距離にある単語 fabric を検索するとします。この場合、次の文を実行できます。

```
SELECT ID, "contains".score, Description
  FROM MarketingInformation CONTAINS ( Description, 'fabric ~ skin' );
```


ID	score	Description
902	1.5572371866083279	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester 「fabric」 is gentle on the earth and soft against your 「skin」. </body></html>

デフォルトの距離が 10 語なので、距離を指定する必要はありませんでした。しかし、距離を 1 語拡張することで、次のような別のローが返されます。

```
SELECT ID, "contains".score, Description
  FROM MarketingInformation CONTAINS ( Description, 'fabric NEAR[11] skin' );
```

ID	score	Description
903	1.5787803210404958	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will serve you equally well on trails or around town. The 「fabric」 has a wicking finish to pull perspiration away from your 「skin」. </body></html>
902	2.163125855043747	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester 「fabric」 is gentle on the earth and soft against your 「skin」. </body></html>

単語間の距離が近いため、ID 903 のスコアの方が高くなります。

ブール検索

全文検索を実行する場合に、複数の単語をブール演算子で区切って指定できます。SQL Anywhere では、全文検索の実行時にブール演算子 AND、OR、AND NOT をサポートします。

ブール検索の構文の詳細については、「CONTAINS 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。

全文検索での AND 演算子の使用

AND 演算子では、AND の両側に指定された単語を両方とも含むローが一致となります。AND 演算子としてアンパサンド (&) を使用することもできます。複数の単語を、間に演算子を入れないで指定した場合、AND を暗黙で指定したことになります。

単語がリストされる順番は重要ではありません。

たとえば、次の文はすべて、MarketingInformation テーブルの Description カラムで **fabric** という単語と **ski** で始まる単語を含むローを検索します。

```
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'ski* AND fabric' );
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'fabric & ski*' );
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'ski* fabric' );
```

全文検索での OR 演算子の使用

OR 演算子では、OR の両側に指定された検索語のうちの少なくとも 1 つを含むローが一致となります。OR 演算子としてパイプ記号 (|) を使用することもできます。この 2 つは同等です。

単語がリストされる順番は重要ではありません。

たとえば、次の文はいずれも、MarketingInformation テーブルの Description カラムで **fabric** という単語または **ski** で始まる単語を含むローを返します。

```
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'ski* OR fabric' );
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'fabric | ski*' );
```

全文検索での AND NOT 演算子の使用

AND NOT 演算子は、左の引数に一致し、右の引数に一致しない結果を検索します。AND NOT 演算子としてハイフン (-) を使用することもできます。この 2 つは同等です。

たとえば、次の文は同義であり、いずれも **fabric** という単語を含み、**ski** で始まる単語は含まないローを返します。

```
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'fabric AND NOT ski*' );
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'fabric -ski*' );
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'fabric & -ski*' );
```

異なるブール演算子の組み合わせ

ブール演算子は、クエリ文字列で組み合わせて使用できます。たとえば、次の文は同義であり、いずれも MarketingInformation テーブルの Description カラムで **fabric** と **skin** を含み、**cotton** は含まない項目を検索します。

```
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'skin fabric -cotton' );
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'fabric -cotton AND skin' );
```

次の文は同義であり、いずれも MarketingInformation テーブルの Description カラムで **fabric** を含むか、または **cotton** と **skin** の両方を含む項目を検索します。

```
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'fabric | cotton AND skin' );
SELECT *
  FROM MarketingInformation
  WHERE CONTAINS ( MarketingInformation.Description, 'cotton skin OR fabric' );
```

単語とフレーズのグループ化

単語と式はカッコを使用してグループ化できます。たとえば、次の文は MarketingInformation テーブルの Description カラムで **cotton** または **fabric** を含み、**ski** で始まる単語を含む項目を検索します。

```
SELECT ID, Description FROM MarketingInformation
  WHERE CONTAINS( MarketingInformation.Description, '( cotton OR fabric ) AND shi*' );
```

	Description
902	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical 「shi」rt is designed for high-intensity workouts in hot and humid weather. The recycled polyester 「fabric」 is gentle on the earth and soft against your skin. </body></html>
903	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual 「shi」rt made of recycled water bottles. It will serve you equally well on trails or around town. The 「fabric」 has a wicking finish to pull perspiration away from your skin. </body></html>
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 construction. 「Shi」elds against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>

複数カラムに渡る検索

すべてのカラムが同じテキスト・インデックスの一部である場合は、1 つのクエリで複数のカラムに渡る全文検索を実行できます。

```
SELECT *
  FROM Products
    WHERE CONTAINS ( t.c1, t.c2, 'term1|term2' );

SELECT *
  FROM t
  WHERE CONTAINS( t.c1, 'term1' )
    OR CONTAINS( t.c2, 'term2' );
```

t1.c1 に *term1* が含まれている、または *t1.c2* に *term2* が含まれている場合、1 つ目のクエリは一致します。

t1.c1 または *t1.c2* に、*term1* または *term2* が含まれている場合、2 つ目のクエリは一致します。この方法で contains を使用すると、一致のスコアも返されます。「全文検索結果のスコア付け」 372 ページを参照してください。

あいまい検索

あいまい検索を使用して、スペルミスや単語の変形を検索できます。あいまい検索を実行するには、FUZZY 演算子の後に二重引用符で囲まれた文字列を付けて、文字列の近似一致を検索します。たとえば、CONTAINS (Products.Description, 'FUZZY "cotton"') では、**cotton** および **coton** や **cotten** などのスペルミスが返されます。

> **注意**
> あいまい検索は、NGRAM 単語区切りを使用して作成されたテキスト・インデックスにしか実行できません。NGRAM 単語区切り、およびあいまい検索との関係については、「テキスト設定オブジェクトの設定」 339 ページを参照してください。

FUZZY 演算子を使用することは、文字列を長さ *n* の部分文字列に手動で分割し、それらを OR 演算子で区切ることと同じです。たとえば、NGRAM 単語区切りと MAXIMUM TERM LENGTH 3 を指定して設定されたテキスト・インデックスがあるとします。'FUZZY "500 main street"' を指定することは、'500 OR mai OR ain OR str OR tre OR ree OR eet' を指定することと同義になります。

FUZZY 演算子は、スコアを返す全文検索で便利です。これは、多くの近似一致が返されても、通常は最高スコアを持つ一致だけが意味のある一致だからです。

インデックスから削除される単語

テキスト・インデックスは、テキスト・インデックスの作成に使用されるテキスト設定オブジェクトに定義された設定に従って、構築されます。次の条件のうち 1 つ以上が当てはまる場合、単語はテキスト・インデックスに含まれません。

- 単語がストップリストに含まれる
- 単語が単語の最小長より短い (GENERIC のみ)
- 単語が単語の最大長より長い

同じ規則はクエリ文字列にも適用されます。削除された単語は、フレーズの末尾または先頭で 0 個以上の単語と一致できます。たとえば、'the' という単語がストップリストに含まれているとします。

- 単語が AND、OR、または NEAR のいずれかの側にある場合、演算子と単語の両方が削除されます。たとえば、'the AND apple'、'the OR apple'、'the NEAR apple' を検索することは、'apple' を検索することと同義になります。
- 単語が AND NOT の右側にある場合、AND NOT と単語の両方が削除されます。たとえば、'apple AND NOT the' を検索することは、'apple' を検索することと同義になります。

  単語が AND NOT の左側にある場合は、式全体が削除され、ローは返されません。
  例：'orange and the AND NOT apple' = 'orange'。
- 単語がフレーズ内にある場合、そのフレーズは削除された単語の位置で、単語と一致させることができます。たとえば、'feed the dog' の検索は、'feed the dog'、'feed my dog'、'feed any dog' などに一致します。

検索している単語がすべてテキスト・インデックスに含まれていない場合は、ローが返されません。たとえば、'the' と 'a' の両方がストップリストにあるとします。'a OR the' の検索では、ローが返されません。

参照

- 「テキスト設定オブジェクト」 339 ページ

ビュー検索

全文検索でビューを使用するためには、ベース・テーブルの必要なカラムにテキスト・インデックスを作成する必要があります。たとえば、Employees テーブルの Address カラムに EmployeeAddressTxtIdx というテキスト・インデックスを作成したとします。次に、Employees テーブルに MyEmployeesView というビューを作成します。次のような文を使用すると、基本となるテーブルのテキスト・インデックスを使用してビューを問い合わせることができます。

```
SELECT COUNT(*) FROM MyEmployeesView WHERE CONTAINS( EmployeeAddressTxtIdx, 'Avenue' );
```

基本となるテーブルのテキスト・インデックスを使用してビューを検索する場合、次のような制限事項があります。

- ビューには、TOP 句、FIRST 句、DISTINCT 句、GROUP BY 句、ORDER BY 句、UNION 句、INTERSECT 句、EXCEPT 句、または Window 関数を含めることはできない。
- CONTAINS クエリはビュー内のベース・テーブルは参照できるが、別のビュー内にあるビュー内のベース・テーブルは参照できない。

全文検索結果のスコア付け

クエリの FROM 句に CONTAINS 句を含めると、一致ごとに関連するスコアが付けられます。このスコアは一致がどれだけ近いかを示しており、スコア情報を使用してデータをソートできます。

スコアは、主に次の 2 つの基準に基づいて付けられます。

- **インデックス付けされたローにその単語が出現する回数**　その単語が、インデックス付けされたローに多く出現するほどスコアは高くなります。
- **テキスト・インデックスにその単語が出現する回数**　その単語が、テキスト・インデックスに多く出現するほどスコアは低くなります。Sybase Central では、テキスト・インデックスの **[ボキャブラリ]** タブを表示することで、テキスト・インデックスに出現する各単語の回数を確認できます。単語をアルファベット順にソートするには、**term** カラムを選択します。**freq** カラムに、その単語がテキスト・インデックスに出現する回数が表示されます。

また、全文検索のタイプによっては、スコアに影響する基準が他にもあります。たとえば近接検索では、検索語にどれだけ近接しているかがスコアに影響します。

スコアの使用方法

デフォルトでは、CONTAINS 句の結果セットには **contains** という相関名があり、**score** というカラムが 1 つ含まれています。"contains".score は、SELECT リスト、ORDER BY 句、またはクエリの他の部分で参照できます。ただし、contains は SQL の予約語なので、二重引用符で囲む必要があります。また、別の相関名を指定する方法もあります (たとえば、CONTAINS (expression) AS ct)。このマニュアルで使用している全文検索の例では、スコア・カラムは ct.score と記載しています。

次の文は、MarketingInformation テーブルの Description カラムで **stretch** または **comfort** で始まる単語を検索します。

```
SELECT ID, ct.score, Description
   FROM MarketingInformation CONTAINS ( MarketingInformation.Description, 'stretch* | comfort*' ) AS
ct
   ORDER BY ct.score DESC;
```

ID	score	Description
910	5.570408968026068	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts provide all day 「comfort」 on or off the trails. Now with a more 「comfort」able and 「stretch」y fabric and an adjustable drawstring waist. </body></html>
907	3.658418186470189	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> A polycarbonate visor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of 「stretch」 to give you a snug yet 「comfort」able fit every time you wear it. </body></html>
905	1.6750365447462499	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side vents for breathable 「comfort」 during aerobic activities. Moisture-absorbing headband liner. </body></html>

プレフィクス単語 **comfort** のインスタンスが、他の項目には 1 つしかないのに対して、項目 910 には 2 つ含まれているため、910 のスコアが最も高くなります。また、項目 910 にはプレフィクス単語 **stretch.** のインスタンスも含まれています。

例 2：複数のカラムの検索

次の例では、複数のカラムに渡る全文検索を実行し、結果のスコアを表示する方法を示しています。

1. 次のようにして、Products テーブルで即時テキスト・インデックスを作成します。

```
CREATE TEXT INDEX scoringExampleMult
  ON Products ( Description, Name );
```

2. 次にようにして、Description カラムと Name カラムで、**cap** または **visor** という単語の全文検索を実行します。CONTAINS 句の結果には ct という相関名が割り当てられ、結果に含まれるように SELECT リストで参照されます。また、ct.score カラムは ORDER BY 句で参照され、結果はスコアの降順でソートされます。

```
SELECT Products.Description, Products.Name, ct.score
   FROM Products CONTAINS ( Products.Description, Products.Name, 'cap OR visor' ) ct
   ORDER BY ct.score DESC;
```

Description	Name	score
Cloth Visor	Visor	3.5635154905713042
Plastic Visor	Visor	3.4507856451176244
Wool cap	Baseball Cap	3.2340501745357333
Cotton Cap	Baseball Cap	3.090467108972918

複数カラムの検索のスコアは、カラム値が連結され、1 つの値としてインデックス付けされているかのように計算されます。ただし、そのフレーズと NEAR 演算子は、カラムの境界を超えて一致することはなく、複数のカラムに出現する検索語は、単一の連結された値よりもスコアが大きくなります。

3. このマニュアルの他の例を正常に動作させるためには、Products テーブルで作成したテキスト・インデックスを削除する必要があります。削除するには、次の文を実行します。

```
DROP TEXT INDEX scoringExampleMult ON Products;
```

チュートリアル：GENERIC テキスト・インデックスへの全文検索の実行

次の手順に従って、GENERIC 単語区切りを使用するテキスト・インデックスに全文検索を実行します。

参照：「チュートリアル：あいまい全文検索の実行」 379 ページ。

♦ GENERIC テキスト・インデックスへの全文検索の実行

1. テキスト設定オブジェクトを作成します。

   次の例では、myTxtConfig というテキスト設定オブジェクトを作成します。FROM 句を含めて、テンプレートとして使用するテキスト設定オブジェクトを指定する必要があります。

   ```
   CREATE TEXT CONFIGURATION myTxtConfig FROM default_char;
   ```

2. テキスト設定オブジェクトをカスタマイズします。

   because、about、therefore、only という単語を含むストップリストを追加します。次に、単語の最大長を 30 に設定します。次のように、個別の ALTER TEXT CONFIGURATION 文で設定する必要があります。

   ```
   ALTER TEXT CONFIGURATION myTxtConfig
     STOPLIST 'because about therefore only';
   ALTER TEXT CONFIGURATION myTxtConfig
     MAXIMUM TERM LENGTH 30;
   ```

3. MarketingInformation テーブルのコピーを作成します。

   a. Sybase Central で **[テーブル]** フォルダを拡張します。

   b. **[MarketingInformation]** を右クリックし、**[コピー]** を選択します。

   c. **[テーブル]** フォルダを右クリックし、**[貼り付け]** を選択します。

   d. **[名前]** フィールドに、**MarketingInformation1** と入力します。**[OK]** をクリックします。

4. Interactive SQL で次のコマンドを実行し、新しいテーブルにデータを設定します。

   ```
   INSERT INTO MarketingInformation1
     SELECT * FROM MarketingInformation;
   ```

5. デモ・データベースの MarketingInformation1 テーブルの Description カラムに、myTxtConfig テキスト設定オブジェクトを参照するテキスト・インデックスを作成します。再表示間隔を 24 時間に設定します。

   ```
   CREATE TEXT INDEX myTxtIndex ON MarketingInformation1 ( Description )
     CONFIGURATION myTxtConfig
     AUTO REFRESH EVERY 24 HOURS;
   ```

6. 次の文を実行して、テキスト・インデックスを再表示します。

   ```
   REFRESH TEXT INDEX myTxtIndex ON MarketingInformation1;
   ```

7. 次の文を実行して、テキスト・インデックスをテストします。

   a. この文により、テキスト・インデックスで **cotton** または **cap** という単語が検索されます。結果はスコアの降順でソートされます。テキスト・インデックスで、**cap** という単語が出現する頻度は **cotton** よりも低いため、**Cap** のスコアの方が高くなります。

      ```
      SELECT ID, Description, ct.*
        FROM MarketingInformation1
          CONTAINS ( Description, 'cotton | cap' ) ct
        ORDER BY score DESC;
      ```

ID	Description	score
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball 「Cap」 </title></head><body lang=EN-US> A lightweight wool 「cap」 with mesh side vents for breathable comfort during aerobic activities. Moisture-absorbing headband liner. </body></html>	2.2742084275032632
904	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball 「Cap」 </title></head><body lang=EN-US> This fashionable hat is ideal for glacier travel, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	1.6980426550094467
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	0.9461597363521859
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying 「cotton」 shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	0.9244136988525732
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	0.9134171046194403

ID	Description	score
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% 「cotton」 /20% polyester blend makes it easy to keep them clean. </body></html>	0.8856420222728282

b. クエリ 2：

```
SELECT ID, Description
  FROM MarketingInformation1
  WHERE CONTAINS( Description, 'cotton -visor' );
```

ID	Description
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% 「cotton」 /20% polyester blend makes it easy to keep them clean. </body></html>
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying 「cotton」 shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>

c. 次の文は、**cotton** という単語について各ローをテストします。ローにこの単語が含まれている場合は Results カラムに 1 が表示され、含まれていない場合は 0 が返されます。

```
SELECT ID, Description, IF CONTAINS ( Description, 'cotton' )
    THEN 1
    ELSE 0
    ENDIF AS Results
  FROM MarketingInformation1;
```

ID	Description	Results
901	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> We've improved the design of this perennial favorite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>	0
902	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>	0
903	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will serve you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>	0
904	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier travel, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	0
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side vents for breathable comfort during aerobic activities. Moisture-absorbing headband liner. </body></html>	0
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	1

ID	Description	Results
907	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> A polycarbonate visor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to give you a snug yet comfortable fit every time you wear it. </body></html>	0
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	1
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% 「cotton」/ 20% polyester blend makes it easy to keep them clean. </body></html>	1
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying 「cotton」 shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	1

参照

- 「全文検索」 338 ページ
- 「テキスト設定オブジェクト」 339 ページ
- 「CREATE TEXT CONFIGURATION 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「ALTER TEXT CONFIGURATION 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「テキスト・インデックス」 353 ページ
- 「CREATE TEXT INDEX 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「ALTER TEXT INDEX 文」 『SQL Anywhere サーバ - SQL リファレンス』

チュートリアル：あいまい全文検索の実行

次の手順に従って、NGRAM 単語区切りを使用するテキスト・インデックスにあいまい全文検索を実行します。

参照：「チュートリアル：GENERIC テキスト・インデックスへの全文検索の実行」 374 ページ。

♦ NGRAM テキスト・インデックスへのあいまい全文検索の実行

1. 次の文を実行して、myFuzzyTextConfig というテキスト設定オブジェクトを作成します。

   ```
   CREATE TEXT CONFIGURATION myFuzzyTextConfig FROM default_char;
   ```

2. 次の文を実行して、単語区切りを NGRAM に変更し、単語の最大長を 3 に設定します。あいまい検索は N-gram を使用して実行されます。これらの変更を行うには、個別の ALTER TEXT CONFIGURATION 文を使用します。

   ```
   ALTER TEXT CONFIGURATION myFuzzyTextConfig
     TERM BREAKER NGRAM;
   ALTER TEXT CONFIGURATION myFuzzyTextConfig
     MAXIMUM TERM LENGTH 3;
   ```

3. MarketingInformation テーブルのコピーを作成します。
   a. Sybase Central で **[テーブル]** フォルダを拡張します。
   b. **[MarketingInformation]** を右クリックし、**[コピー]** を選択します。
   c. **[テーブル]** フォルダを右クリックし、**[貼り付け]** を選択します。
   d. **[名前]** フィールドに、**MarketingInformation2** と入力します。**[OK]** をクリックします。

4. 次のコマンドを実行して、MarketingInformation2 テーブルにデータを追加します。

   ```
   INSERT INTO MarketingInformation2
     SELECT * FROM MarketingInformation;
   ```

5. 次のコマンドを実行して、MarketingInformation2 テーブルの Description カラムに、myFuzzyTextConfig テキスト設定オブジェクトを参照するテキスト・インデックスを作成します。

   ```
   CREATE TEXT INDEX myFuzzyTextIdx ON MarketingInformation2 ( Description )
     CONFIGURATION myFuzzyTextConfig;
   ```

6. 次の文を実行して、テキスト・インデックスをテストします。

   次のあいまいクエリにより、**coten** に似た単語が検索されます。

   ```
   SELECT MarketingInformation2.Description, ct.*
     FROM MarketingInformation2 CONTAINS ( MarketingInformation2.Description, 'FUZZY "coten"' )
   ct
     ORDER BY ct.score DESC;
   ```

Description	score
<html><head><meta http-equiv=「Content」-Type 「content」 ="text/html; charset=windows-1252"><title>Sweatshirt</title></ head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	0.9461597363521859

Description	score
<html><head><meta http-equiv=「Content」-Type 「content」 ="text/html; charset=windows-1252"><title>Shorts</title></ head><body lang=EN-US> These quick-drying 「cotton」 shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </ body></html>	0.9244136988525732
<html><head><meta http-equiv=「Content」-Type 「content」 ="text/html; charset=windows-1252"><title>Visor</title></ head><body lang=EN-US> Lightweight 100% organically grown 「cotton」 construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor.</ p></body></html>	0.9134171046194403
<html><head><meta http-equiv=「Content」-Type 「content」 ="text/html; charset=windows-1252"><title>Sweatshirt</title></ head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% 「cotton」/20% polyester blend makes it easy to keep them clean. </body></html>	0.8856420222728282
<html><head><meta http-equiv=「Content」-Type 「content」 ="text/html; charset=windows-1252"><title>Baseball Cap</title></ head><body lang=EN-US> This fashionable hat is ideal for glacier travel, sea-kayaking, and hiking. With concealed draw cord for windy days.</ span> </body></html>	0
<html><head><meta http-equiv=「Content」-Type 「content」 ="text/html; charset=windows-1252"><title>Baseball Cap</title></ head><body lang=EN-US> A lightweight wool cap with mesh side vents for breathable comfort during aerobic activities. Moisture-absorbing headband liner. </body></html>	0
<html><head><meta http-equiv=「Content」-Type 「content」 ="text/html; charset=windows-1252"><title>Tee Shirt</title></ head><body lang=EN-US> We've improved the design of this perennial favorite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>	0

Description	score
<html><head><meta http-equiv=「Content」-Type 「content」 ="text/html; charset=windows-1252"><title>Tee Shirt</title></ head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will serve you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </ body></html>	0
<html><head><meta http-equiv=「Content」-Type 「content」 ="text/html; charset=windows-1252"><title>Tee Shirt</title></ head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>	0
<html><head><meta http-equiv=「Content」-Type 「content」 ="text/html; charset=windows-1252"><title>Visor</title></ head><body lang=EN-US> A polycarbonate visor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to give you a snug yet comfortable fit every time you wear it. </body></ html>	0

注意
最後の 6 つのローに、一致する N-gram を含む単語があります。ただし、これらの単語はテーブル内のすべてのローに含まれているため、スコアは割り当てられていません。

参照

- 「全文検索」 338 ページ
- 「テキスト設定オブジェクト」 339 ページ
- 「CREATE TEXT CONFIGURATION 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「ALTER TEXT CONFIGURATION 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「テキスト・インデックス」 353 ページ
- 「CREATE TEXT INDEX 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「ALTER TEXT INDEX 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「全文検索結果のスコア付け」 372 ページ

チュートリアル：NGRAM テキスト・インデックスへの全文検索の実行

次の手順に従って、NGRAM 単語区切りを使用するテキスト・インデックスに全文検索を実行します。中国語、日本語、韓国語データの全文検索を作成する際もこの手順で実行できます。

マルチバイト文字セットを含むデータベースでは、全角カンマや全角スペースなどの一部の句読表記文字やスペース文字は、英数字として処理される場合があります。

参照：「チュートリアル：あいまい全文検索の実行」 379 ページ。

♦ NGRAM テキスト・インデックスへの全文検索の実行

1. 次の文を実行して、myNcharNGRAMTextConfig という NCHAR テキスト設定オブジェクトを作成します。

   ```
   CREATE TEXT CONFIGURATION myNcharNGRAMTextConfig FROM default_nchar;
   ```

2. 次の文を実行して、TERM BREAKER アルゴリズムを NGRAM に変更し、MAXIMUM TERM LENGTH (N) を 2 に設定します。

   ```
   ALTER TEXT CONFIGURATION myNcharNGRAMTextConfig
     TERM BREAKER NGRAM;

   ALTER TEXT CONFIGURATION myNcharNGRAMTextConfig
     MAXIMUM TERM LENGTH 2;
   ```

   中国語、日本語、韓国語データの場合は、N の値を 2 または 3 に設定することをおすすめします。検索語が 1 文字または 2 文字以内の場合は、N の値を 1 に設定します。クエリが長い場合に N の値を 1 に設定すると、実行に時間がかかる可能性があります。

3. MarketingInformation テーブルのコピーを作成します。
   a. Sybase Central で **[テーブル]** フォルダを拡張します。
   b. **[MarketingInformation]** を右クリックし、**[コピー]** を選択します。
   c. **[テーブル]** フォルダを右クリックし、**[貼り付け]** を選択します。
   d. **[名前]** フィールドに、**MarketingInformationNgram** と入力します。**[OK]** をクリックします。

4. 次の文を実行して、MarketingInformationNgram テーブルにデータを追加します。

   ```
   INSERT INTO MarketingInformationNgram
     SELECT *
       FROM MarketingInformation;
   COMMIT;
   ```

5. 次の文を実行して、myNcharNGRAMTextConfig テキスト設定オブジェクトを使用することで、MarketingInformationNgram テーブルの Description カラムに IMMEDIATE REFRESH テキスト・インデックスを作成します。

   ```
   CREATE TEXT INDEX ncharNGRAMTextIndex
     ON MarketingInformationNgram( Description )
       CONFIGURATION myNcharNGRAMTextConfig;
   ```

6. 次の文を実行して、テキスト・インデックスをテストします。

   a. この文により、2-GRAM のテキスト・インデックスで **sw** を含む単語が検索されます。結果はスコアの降順でソートされます。

```
SELECT M.Description, ct.*
  FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 'sw' ) ct
  ORDER BY ct.score DESC;
```

Description	score
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>「Sw」eatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded 「Sw」eatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	2.262071918398649
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>「Sw」eatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean. </body></html>	1.5556043490424176

   b. 次の文は、**ams** を含む単語を検索します。結果はスコアの降順でソートされます。

```
SELECT M.Description, ct.*
  FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 'ams' ) ct
  ORDER BY ct.score DESC;
```

   2-GRAM テキスト・インデックスでは、前述の文はセマンティック上、次の文と同義になります。

```
SELECT M.Description, ct.*
  FROM MarketingInformationNgram AS M
CONTAINS( M.Description, '"am ms"' ) ct
  ORDER BY ct.score DESC;
```

Description	score
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded sweatshirt with taped neck se「ams」. Comes pre-washed for softness and to lessen shrinkage. </body></html>	1.6619019465461564

Description	score
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched se「ams」 for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean. </body></html>	1.5556043490424176

c. 次の文は、**v** の後に任意の英数字が続く単語を検索します。インデックス付けされたデータで、**ve** の出現する頻度の方が高いため、2-gram の **ve** を含むローのスコアは、**vi** を含むローよりも低くなります。結果はスコアの降順でソートされます。

```
SELECT M.ID, M.Description, ct.*
  FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 'v*' ) ct
  ORDER BY ct.score DESC;
```

ID	Description	score
901	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> We'「ve」 impro「ve」d the design of this perennial fav「vo」rite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>	3.3416789108071976
907	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>「Vi」sor</title></head><body lang=EN-US> A polycarbonate「vi」sor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to gi「ve」 you a snug yet comfortable fit e「ve」ry time you wear it. </body></html>	2.1123084896159376
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side 「ve」nts for breathable comfort during aerobic acti「vi」ties. Moisture-absorbing headband liner. </body></html>	1.6750365447462499

ID	Description	score
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts pro「vi」de all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	0.9244136988525732
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>「Vi」sor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	0.9134171046194403
904	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier tra「ve」l, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	0.7313071661212746
903	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will ser「ve」 you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>	0.6799436746197272

d. 次の文は、各ローで v を含む任意の単語を検索します。2 番目の文の後、変数には文字列 av OR ev OR iv OR ov OR rv OR ve OR vi OR vo が含まれます。結果はスコアの降順でソートされます。インデックス付けされたすべてのローに N-gram があると、スコアは 0 になります。

これは、ホワイトスペースや英数字以外の文字の前にある 1 文字を検索するための唯一の方法です。

```
CREATE VARIABLE query NVARCHAR (100);
SELECT LIST (term, ' OR ' )
INTO query
  FROM sa_text_index_vocab( 'ncharNGRAMTextIndex', 'MarketingInformationNgram', 'dba' )
  WHERE term LIKE '%v%';
```



```
SELECT M.ID, M.Description, ct.*
  FROM MarketingInformationNgram AS M
  CONTAINS( M.Description, query ) ct
  ORDER BY ct.score DESC;
```

ID	Description	score
901	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> We'「ve」 impr「ove」d the design of this perennial f「avo」rite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>	6.654350268810443
907	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>「Vi」sor</title></head><body lang=EN-US> A polycarbonate「vi」sor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to g「ive」 you a snug yet comfortable fit 「eve」ry time you wear it. </body></html>	4.265623837817126
903	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will se「rve」 you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>	2.9386676702799504
910	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts pr「ovi」de all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	2.5481193655722336

ID	Description	score
904	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier tr「ave」l, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	2.4293498211307214
905	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side 「ve」nts for breathable comfort during aerobic act「ivi」ties. Moisture-absorbing headband liner. </body></html>	1.6750365447462499
906	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>「Vi」sor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	0.9134171046194403
902	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>	0
908	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	0

ID	Description	score
909	<html><head><meta http-equ「iv」=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean. </body></html>	0

e. 次の文は、Description カラムで **ea**、**ka**、**k** を含むローを検索します。

```
SELECT M.ID, M.Description, ct.*
  FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 'ea ka ki' ) ct
 ORDER BY ct.score DESC;
```

ID	Description	score
904	<html><h「ea」d><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></h「ea」d><body lang=EN-US> This fashionable hat is id「ea」l for glacier travel, s「ea」-「ka」yaking, and hi「ki」ng. With conc「ea」led draw cord for windy days. </body></html>	3.4151032739119733

f. 次の文は、Description カラムで **ve** と **vi** を含み、**gg** は含まないローを検索します。

```
SELECT M.ID, M.Description, ct.*
  FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 've & vi -gg' ) ct
  ORDER BY ct.score DESC;
```

ID	Description	score
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side 「ve」nts for breathable comfort during aerobic acti「vi」ties. Moisture-absorbing headband liner. </body></html>	1.6750365447462499

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

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。

テーブル内のすべてのロー、WHERE 句によって指定されるテーブルのサブセット、またはテーブル内の 1 つ以上のロー・グループに、集合関数を適用できます。集合関数を適用するそれぞれのロー・セットから、SQL Anywhere が単一の値を生成します。

サポートされる集合関数の一部を次に示します。

- **avg(expression)**　返されたローについて提供された式の平均。
- **count(expression)**　提供されたグループで、式が NOT NULL のロー数。
- **count(*)**　各グループの中のロー数。
- **list(string-expr)**　各ロー・グループの中の *string-expr* に対するすべての値で構成されている、カンマで区切られたリストを含む文字列。
- **max(expression)**　返されたローの最大値。
- **min(expression)**　返されたローの最小値。
- **stddev(expression)**　返されたローの標準偏差。
- **sum(expression)**　返されたローの合計。
- **variance(expression)**　返されたローについての式の分散。

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。

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：データ・グループの選択」 402 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内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ます。

- 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。

```
SELECT Name,
    ( SELECT SUM( p.Quantity )
```

```
        FROM SalesOrderItems )
  FROM Products p;
```

Name	SUM(p.Quantity)
Tee Shirt	30,716
Tee Shirt	59,238

新しいバージョンで同じクエリを実行すると、「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。」というエラー・メッセージが生成されます。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(*) は、重複を除外しないで、指定されたテーブルのロー数を返します。NULL の入っているローを含め、各ローを個別にカウントします。この関数は、引数として式を必要としません。定義上、この関数は、特定のカラムに関する情報を使用しないからです。

次の文は、Employees テーブルの全従業員数を検出します。

```
SELECT COUNT(*)
  FROM Employees;
```

他の集合関数と同じように、COUNT(*) も、select リストにある他の集合関数や WHERE 句などと結合できます。次に例を示します。


```
SELECT COUNT(*), AVG( UnitPrice )
  FROM Products
  WHERE UnitPrice > 10;
```

COUNT(*)	AVG(Products.UnitPrice)
5	18.2

DISTINCT を伴う集合関数の使用

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、次の文を実行します。

```
SELECT COUNT( DISTINCT City )
  FROM Contacts;
```

COUNT(DISTINCT Contacts.City)
16

クエリ内で DISTINCT を伴って集合関数を 2 つ以上使用できます。各 DISTINCT は個別に評価されます。次に例を示します。

```
SELECT COUNT( DISTINCT GivenName ) "first names",
       COUNT( DISTINCT Surname ) "last names"
  FROM Contacts;
```

first 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

```
SELECT AVG( UnitPrice )
  FROM Products
  WHERE UnitPrice > 50;
```

AVG(Products.UnitPrice)
(NULL)

GROUP BY 句：クエリ結果のグループへの編成

GROUP BY 句は、テーブルの出力をグループに分けます。1 つ以上のカラム名によって、または計算カラムの結果によって、ローをグループ化することができます。

句の順序
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。

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、「GROUPING SETS のショートカットとしての ROLLUP と CUBE の使用」 488 ページを参照してください。

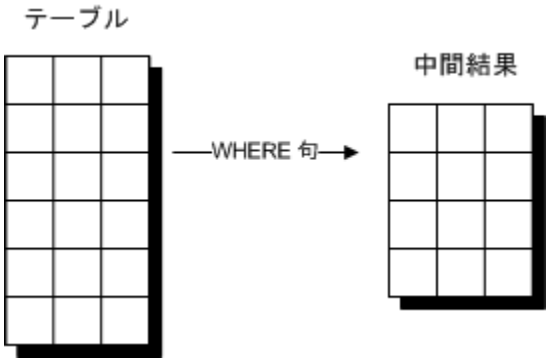
次のような形式の単一テーブルのクエリを考えてみます。

SELECT *select-list*
  FROM *table*
  WHERE *where-search-condition*
  GROUP BY [*group-by-expression* | **ROLLUP** *(group-by-expression)*]
  HAVING *having-search-condition*

このクエリは、次のように実行します。

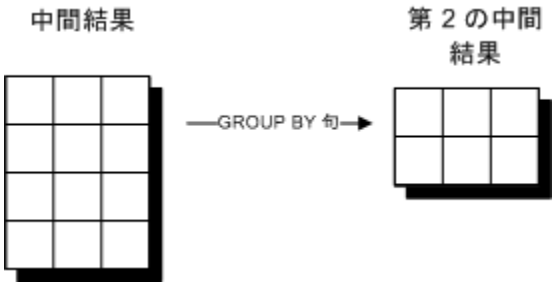
1. 「WHERE 句を適用する」

   テーブルのローの一部だけで構成される中間結果が生成されます。



2. 「結果をグループに分割する」

   GROUP BY 句の指示どおりに、各グループに対してローが 1 つある中間結果が生成されます。生成された各ローには、グループごとの *group-by-expression* と、*select-list* および *having-search-condition* の集合関数の計算結果が含まれます。



3. 「ROLLUP 演算があれば適用する」

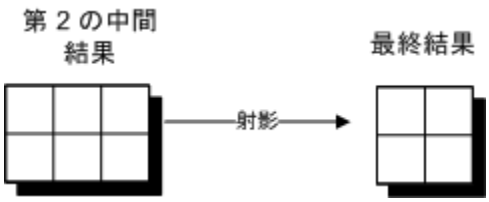
   ROLLUP 演算の一部として計算された小計ローが、結果セットに追加されます。

4. 「HAVING 句を適用する」

   第 2 の中間結果で HAVING 句の基準に満たないローは、この時点で削除されます。

5. 「プロジェクトの結果を表示する」

   クエリの結果セットに表示する必要のあるカラムだけが手順 3 から取り出されます。つまり、*select-list* にある式に対応するカラムだけが取り出されます。



このプロセスでは、GROUP BY 句のあるクエリについて、いくつかの要件が作成されます。

- WHERE 句が最初に評価される。したがって、どの集合関数も、WHERE 句を満たすローについてのみ評価されます。
- 最終結果セットは、分割されたローを保持している第 2 の中間結果から構築される。第 2 の中間結果は、*group-by-expression* に一致するローを保持しています。したがって、集合関数ではない式が *select-list* にある場合、同様に *group-by-expression* にもその式がなければなりません。プロジェクションの段階では関数の評価は行われません。
- *group-by-expression* にある式を、*select-list* に含まないことも可能です。その式は、結果にプロジェクションされます。

複数のカラムを使用した GROUP BY

GROUP BY 句に 1 つ以上の式をリストできます。つまり、式の組み合わせによって、テーブルをグループ化できます。

次のクエリは、まず名前別にグループ化し、次にサイズ別にグループ化した製品の平均価格をリストします。

```
SELECT Name, Size, AVG( UnitPrice )
  FROM Products
  GROUP BY Name, Size;
```

Name	Size	AVG(Products.UnitPrice)
Baseball Cap	One size fits all	9.5
Sweatshirt	Large	24
Tee Shirt	Large	14
Tee Shirt	One size fits all	14
...	...	...

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%'
  GROUP BY Name
  HAVING SUM( Quantity ) > 100;
```

Name	SUM(Products.Quantity)
Tee Shirt	157

この例では次のようになっています。

- WHERE 句によって、**shirt** という語を含む名前 (Tee Shirt、Sweatshirt) があるローだけが選択されます。
- GROUP BY 句によって、共通の名前のローが集められます。
- SUM 集合関数によって、各グループにある製品の総数が計算されます。
- HAVING 句によって、最終結果から在庫総数が 100 以下のグループが除外されます。

集合関数に伴う 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 文が生成するベクトル集約値は、結果の各ローにカラムとして表示されます。それとは対照的に、集合関数があって GROUP BY 句がないクエリが生成するスカラ集約値は、カラムとして表示されますが、ローは 1 つだけです。次に例を示します。

```
SELECT AVG( UnitPrice )
  FROM Products;
```

AVG(Products.UnitPrice)
13.3

GROUP BY と SQL/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 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 では、select リストまたは GROUP BY 句に存在しない集合関数も HAVING 句で使用できる拡張機能をサポートしています。

SQL Anywhere の他の標準への準拠については、「SQL ダイアレクト」 683 ページを参照してください。

HAVING 句：データ・グループの選択

HAVING 句は、クエリが返すローを制限します。HAVING 句は、WHERE 句が SELECT 句の条件を設定するのと同じような方法で、GROUP BY 句の条件を設定します。

HAVING 句の探索条件は、WHERE 探索条件と同じです。ただし、WHERE 探索条件では集合関数を指定できません。たとえば次の使用方法は有効です。

```
HAVING AVG( UnitPrice ) > 20
```

次の使用方法は無効です。

```
WHERE AVG( UnitPrice ) > 20
```

集合関数を伴う HAVING の使用

次の文は、集合関数を持つ HAVING 句を使用する簡単な例です。

2 種類以上のサイズまたは色がある製品をリストするには、1 種類だけの製品を含むグループは省き、Products テーブルのローを名前でグループ化するクエリが必要です。

```
SELECT Name
  FROM Products
  GROUP BY Name
  HAVING COUNT( * ) > 1;
```

Name
Tee Shirt
Baseball Cap
Visor
Sweatshirt

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」 393 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%';
```

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

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。

注意
TART AT 値は 1 以上にする必要があります。TOP 値は、定数の場合に 1 以上、変数の場合に 0 以上にする必要があります。

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。

- データが正規化されていない場合、たとえテーブルが関連付けられていなくても、異なるように思われる情報を 1 つの結果セットにまとめたいと思うことがあります。
- WHERE 句や HAVING 句内では、集合演算子によって、NULL を扱う方法が異なります。WHERE 句や HAVING 句内では、NOT NULL のエントリが同じである NULL を含む 2 つのローは、同じと見なされません。2 つの NULL 値は、同じと定義されていないためです。集合演算子は、そうした 2 つのローを同じと見なします。

参照

- 「集合演算子と NULL」 410 ページ
- 「EXCEPT 句」 『SQL Anywhere サーバ - SQL リファレンス』
- 「INTERSECT 句」 『SQL Anywhere サーバ - SQL リファレンス』
- 「UNION 句」 『SQL Anywhere サーバ - SQL リファレンス』

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。

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。

参照

- 「EXCEPT 句」『SQL Anywhere サーバ - SQL リファレンス』
- 「INTERSECT 句」『SQL Anywhere サーバ - SQL リファレンス』

集合操作のルール

UNION 文、EXCEPT 文、INTERSECT 文には、次のルールが適用されます。

- **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
```

```
    FROM Customers
ORDER BY 1;
```

集合演算子と NULL

集合演算子 UNION、EXCEPT、INTERSECT と探索条件内では、NULL を扱う方法が異なります。この違いが、集合演算子を使用する主な理由の 1 つです。

ローを比較するとき、集合演算子は、NULL 値を互いに等しいものとして扱います。対照的に、探索条件で NULL が NULL と比較された場合、結果は不定 (真ではない) となります。

この違いがもたらす結果の 1 つとして、query-1 EXCEPT ALL query-2 の結果セット内のロー数が、**常に**各クエリの結果セット内のロー数の差異であるということです。

テーブル T1 と T2 を例に説明します。各テーブルには、次のカラムがあります。

```
col1 INT,
col2 CHAR(1)
```

テーブルとデータは次のように設定されています。

```
CREATE TABLE T1 (col1 INT, col2 CHAR(1));
CREATE TABLE T2 (col1 INT, col2 CHAR(1));
INSERT INTO T1 (col1, col2) VALUES(1, 'a');
INSERT INTO T1 (col1, col2) VALUES(2, 'b');
INSERT INTO T1 (col1) VALUES(3);
INSERT INTO T1 (col1) VALUES(3);
INSERT INTO T1 (col1) VALUES(4);
INSERT INTO T1 (col1) VALUES(4);
INSERT INTO T2 (col1, col2) VALUES(1, 'a');
INSERT INTO T2 (col1, col2) VALUES(2, 'x');
INSERT INTO T2 (col1) VALUES(3);
```

テーブル内のデータは次のようになっています。

- テーブル T1

col1	col2
1	a
2	b
3	(NULL)
3	(NULL)
4	(NULL)
4	(NULL)

- テーブル T2

col1	col2
1	a
2	x
3	(NULL)

T2 にもある T1 のローを要求するクエリの一例を次に示します。

```
SELECT T1.col1, T1.col2
  FROM T1 JOIN T2
  ON T1.col1 = T2.col1
  AND T1.col2 = T2.col2;
```

T1.col1	T1.col2
1	a

ロー (3, NULL) は、結果セットにありません。これは、NULL と NULL の比較が真ではないためです。対照的に、INTERSECT 演算子を使用してこの問題にアプローチすると、結果に NULL を持つローが含まれます。

```
SELECT col1, col2
  FROM T1
  INTERSECT
    SELECT col1, col2
      FROM T2;
```

col1	col2
1	a
3	(NULL)

次のクエリは、探索条件を使用して T2 にはない T1 のローをリストしています。

```
SELECT col1, col2
  FROM T1
  WHERE col1 NOT IN (
     SELECT col1
       FROM T2
       WHERE T1.col2 = T2.col2 )
  OR col2 NOT IN (
     SELECT col2
       FROM T2
       WHERE T1.col1 = T2.col1 );
```

col1	col2
2	b
3	(NULL)

col1	col2
4	(NULL)
3	(NULL)
4	(NULL)

T1 の NULL を含むローは、比較によって除外されていません。対照的に、EXCEPT ALL を使用してこの問題にアプローチすると、両方のテーブルに含まれる NULL を持つローが結果から除外されます。この場合、T2 の (3, NULL) ローは、T1 の (3, NULL) ローと同じと認識されています。

```
SELECT col1, col2
  FROM T1
  EXCEPT ALL
    SELECT col1, col2
      FROM T2;
```

col1	col2
2	b
3	(NULL)
4	(NULL)
4	(NULL)

EXCEPT 演算子を使用すると、結果がさらに制限されます。EXCEPT 演算子は、T1 から (3, NULL) のローを両方とも削除し、また (4, NULL) ローの 1 つを重複として除外しています。

```
SELECT col1, col2
  FROM T1
  EXCEPT
    SELECT col1, col2
      FROM T2;
```

col1	col2
2	b
4	(NULL)

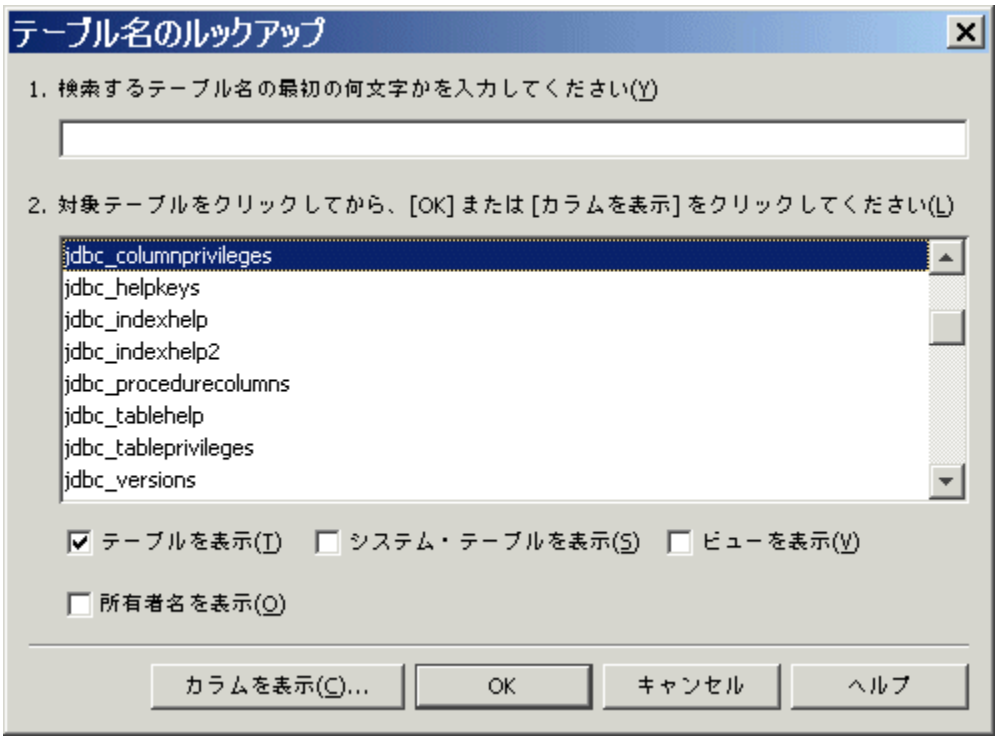
ジョイン：複数テーブルからのデータ検索

目次

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・リストに戻ります。ここでもう一度 [Esc] キーを押すと **[SQL 文]** ウィンドウ枠に戻ります。[Enter] キーを押すと、選択されているテーブルまたはカラム名が **[SQL 文]** ウィンドウ枠の現在カーソルが置かれている場所にコピーされます。

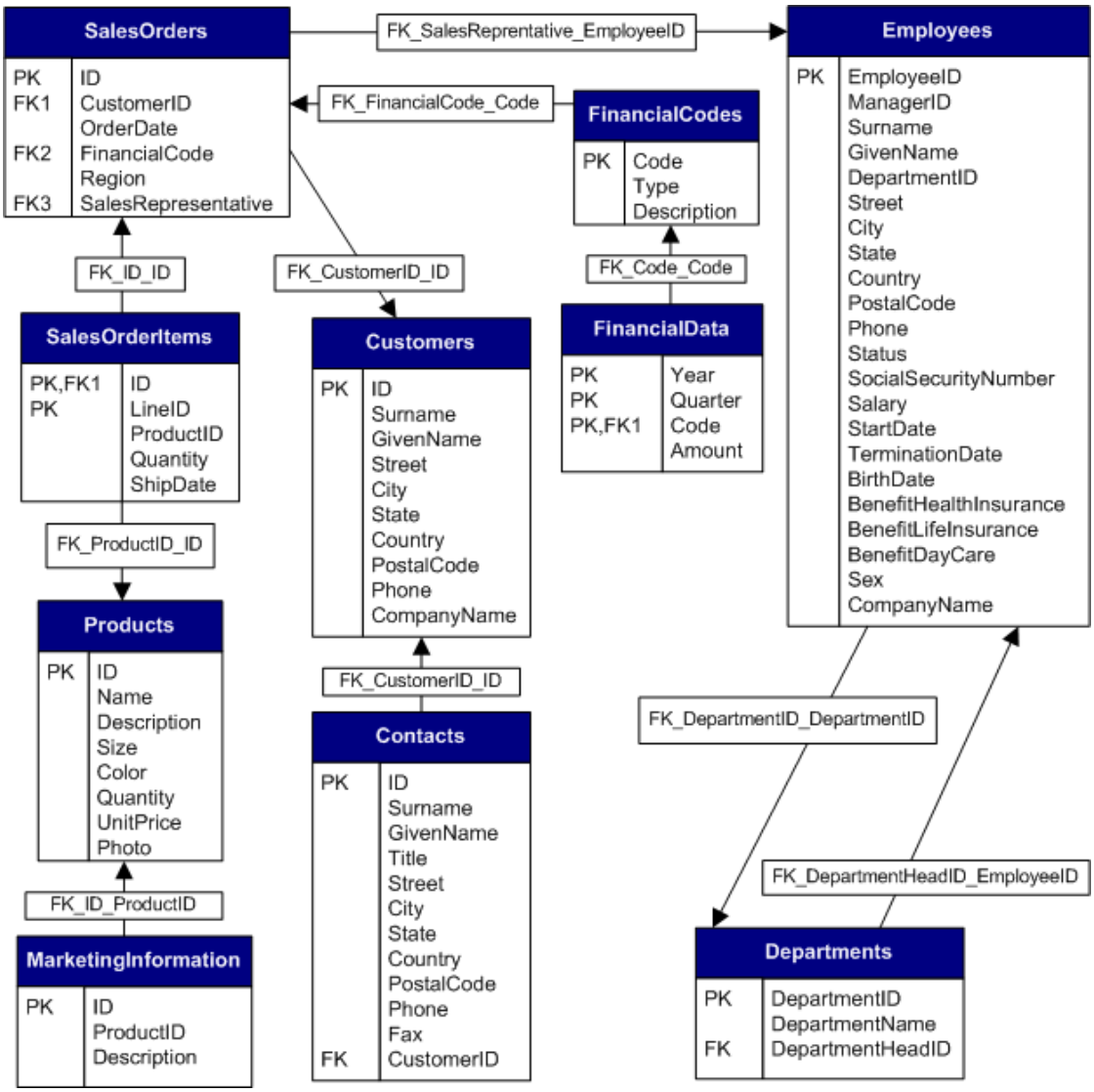
リストを終了するには、[Esc] キーを押します。

SQL Anywhere サンプル・データベース内のテーブルの詳細については、「チュートリアル：サンプル・データベースの使用」『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」 449 ページを参照してください。



ジョイン操作

「ジョイン」とは、テーブル内のローを、指定したカラムの値と比較することによって結合する操作のことです。この項では、SQL Anywhere のジョイン構文の概要を説明します。

リレーショナル・データベースは別々のタイプのオブジェクトに関する情報を別々のテーブルに保存します。たとえば、あるテーブルには従業員だけの情報があり、別のテーブルには部署関連の情報があります。Employees テーブルには、従業員の名前や住所などの情報が保存されています。Departments テーブルには、部署名や部長名などの情報が入ります。

ほとんどの問い合わせに対する答えは、さまざまなテーブルの情報を組み合わせることによってのみ取得できます。たとえば、「営業部の責任者は誰か」という質問に対する回答を取得するためには、Departments テーブルで適切な従業員を特定し、Employees テーブルでその従業員名を検索します。

ジョインは複数のテーブルからの情報を取り入れた新規の仮想テーブルを作ることによって、そのような質問に答える手段です。たとえば、Employees テーブルと Departments テーブルの情報を組み合わせて、部長のリストを作成できます。FROM 句を使用して、必要な情報の入ったテーブルを特定します。

ジョインを有効なものにするには、各テーブルの適切なカラムを組み合わせてください。部長のリストを作成するには、組み合わせたテーブルの各ローに部署名とその部署を管理する従業員の名前を指定してください。特定のタイプのジョイン操作を指定するか、ON 句を使用して、複合テーブルにカラムをどのように適合させるかを調節します。

参照

- 「FROM 句」 『SQL Anywhere サーバ - SQL リファレンス』

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、派生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 *table-expression*, ...

文中の各項目を次に説明します。

table-expression :
table-name
| *view-name*
| *derived-table-name*
| *lateral-derived-table-name*
| *join-expression*
| **(** *table-expression*, ... **)**
| *openstring-expression*
| *apply-expression*

table-name or *view-name*:
[*owner*.] *table-or-view-name* [[**AS**] *correlation-name*]

derived-table-name :
(*select-statement* **)** [**AS**] *correlation-name* [**(** *column-name*, ... **)**]

join-expression :
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**]

apply-expression :
table-expression { **CROSS** | **OUTER** } **APPLY** *table-expression*

join-condition :

「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。

注意

CROSS JOIN には ON 句を使用できません。

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。

ジョイン条件

「ジョイン条件」を使用するとテーブルをジョインできます。ジョイン条件とは、簡単に言えば探索条件のことです。ジョイン条件は、カラム内の値と値の関係に基づいて、ジョインしたテーブルからローのサブセットを選択します。たとえば、次のクエリでは Products テーブルと SalesOrderItems テーブルからデータが取り出されます。

```
SELECT *
FROM Products JOIN SalesOrderItems
  ON Products.ID = SalesOrderItems.ProductID;
```

このクエリのジョイン条件は、次の部分です。

```
Products.ID = SalesOrderItems.ProductID
```

このジョイン条件は、両方のテーブルのローに同じ製品 ID がある場合にかぎり、結果セットでこのローを結合できることを示しています。

ジョイン条件には明示的なものと、生成されたものがあります。「明示的ジョイン条件」とは、ON 句または WHERE 句の中に置かれたジョイン条件のことです。以下のクエリでは ON 句が使用されています。このクエリでは、2 つのテーブルの直積 (すべてのローの組み合わせ) が生成されます。ただし、ID 番号が一致しないローは除外されます。結果は、注文の詳細が記載された顧客リストになります。

```
SELECT *
FROM Customers
JOIN SalesOrders
ON SalesOrders.CustomerID = Customers.ID;
```

これに対し、「生成されたジョイン条件」とは、KEY JOIN または NATURAL JOIN を指定すると自動的に作成されるジョイン条件のことです。キー・ジョインの場合、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 も指定せずに JOIN キーワードを使用してから、ON 句内で同じジョイン条件を明示的に記述しても、同じ結果が得られます。

キー・ジョインまたはナチュラル・ジョインを指定した ON 句を使用すると、使用されるジョイン条件は、明示的に指定したジョイン条件と生成されたジョイン条件の「論理積」になります。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 つのテーブルにこのタイプのジョインを指定すると、両方のテーブルにあるローの可能な組み合わせがすべて生成されます。結果セットのサイズは、1 番目のテーブルにあるローの数と 2 番目のテーブルにあるローの数を乗算したものです。クロス・ジョインは、直積とも呼ばれます。クロス・ジョインでは ON 句を使用できません。
- **KEY JOIN (キー・ジョイン)**　このタイプのジョイン条件では、テーブル間の外部キー関係が使用されます。ジョイン・タイプ (INNER、OUTER など) を指定しないで JOIN キーワードを使用する場合や、ON 句がない場合は、キー・ジョインはデフォルトになります。
- **NATURAL JOIN (ナチュラル・ジョイン)**　このジョインでは、同じ名前のカラムに基づいて、ジョイン条件が自動的に生成されます。
- **ON 句を使用したジョイン**　ON 句内にジョイン条件を明示的に指定すると、このタイプのジョインになります。これをキー・ジョインまたはナチュラル・ジョインと併用すると、ジョイン条件には生成されたジョイン条件と明示的ジョイン条件の両方が含まれます。KEY や NATURAL の付かない JOIN キーワードと併用すると、生成されるジョイン条件はありません。「明示的なジョイン条件 (ON 句)」 422 ページを参照してください。

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)。
- 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」 428 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。そのため、A JOIN B JOIN C は (A JOIN B) JOIN C と同じです。また、次の 2 つの SELECT 文は同じです。

```
SELECT *
FROM A JOIN B JOIN C JOIN D;

SELECT *
FROM ( ( A JOIN B ) JOIN C ) JOIN D;
```

3 つ以上のテーブルをジョインした場合、そのジョインにはテーブル式が含まれます。A JOIN B JOIN C の例では、テーブル式 A JOIN B が C とジョインされます。つまり、概念上は A と B がジョインされ、その結果が C にジョインされます。

テーブル式に外部ジョインが含まれるときは、ジョインの順序が重要になります。たとえば、A JOIN B LEFT OUTER JOIN C は、(A JOIN B) LEFT OUTER JOIN C と解釈されます。つまり、テーブル式 A JOIN B が C にジョインされます。このとき、テーブル式 A JOIN B は保護され、テーブル C には NULL が入力されます。

外部ジョインの詳細については、「外部ジョイン」 428 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」 452 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」 445 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。

DELETE 文、UPDATE 文、INSERT 文でのジョインの使用

ジョインは、DELETE 文、UPDATE 文、INSERT 文、SELECT 文で使用できます。ansi_update_constraints オプションが Off に設定されていれば、ジョインを含むカーソルをいくつか更新できます。SQL Anywhere 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 標準のジョインをサポートしています。また、次に示す標準以外のジョインもサポートしています。

- 「Transact-SQL の外部ジョイン (*= or =*)」 432 ページ
- 「ジョインで重複する相関名 (スター・ジョイン)」 436 ページ
- 「キー・ジョイン」 448 ページ
- 「ナチュラル・ジョイン」 443 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。「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。

明示的なジョイン条件 (ON 句)

キー・ジョインやナチュラル・ジョインの代わりに、またはこれらとともに、明示的ジョイン条件を使用してジョインを指定できます。ジョインの直後に ON 句を挿入し、ジョイン条件を指定してください。ジョイン条件は、常にその直前にあるジョインを参照します。ON 句は、ジョインのローに制限を適用します。これは、WHERE 句がクエリのローに制限を適用するのと同様です。

ON 句を使用すると、CROSS JOIN よりも使用しやすいジョインを構成できます。たとえば、SalesOrders テーブルと Employees テーブルのジョインに ON 句を適用できます。この場合、取得される結果のすべてのローで、SalesOrders テーブル内の SalesRepresentative が Employees テーブル内のものと同じになるように制限されます。各ローには、注文とその注文を担当する営業担当者についての情報が入っています。

たとえば、次のクエリでは、最初の ON 句を使用して SalesOrders を Customers にジョインします。また、2 番目の ON 句を使用して、テーブル式 (SalesOrders JOIN Customers) をベース・テーブル SalesOrderItems にジョインします。

```
SELECT *
FROM SalesOrders JOIN Customers
   ON SalesOrders.CustomerID = Customers.ID
 JOIN SalesOrderItems
   ON SalesOrderItems.ID = SalesOrders.ID;
```

ON 句でのテーブルの参照

ON 句で参照されるテーブルは、その ON 句が修飾するジョインの一部である必要があります。たとえば、次の構文は無効です。

```
FROM ( A KEY JOIN B ) JOIN ( C JOIN D ON A.x = C.x )
```

ここでの問題は、ジョイン条件 A.x = C.x がテーブル A を参照していることです。テーブル A は、このジョイン条件が修飾するジョイン (この場合 C JOIN D) の一部ではありません。

ただし、ANSI/ISO 標準の 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」 426 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;
```

Surname	ID	OrderDate
Chin	2008	4/2/2001
Chin	2020	3/4/2001
Chin	2032	7/5/2001
Chin	2044	7/15/2000
Chin	2056	4/15/2001
...	...	...

次はこの例に関する説明です。

- このクエリの結果に含まれているのは、648 個のロー (SalesOrders テーブルの各ローに対応) のみです。直積における 48,600 のローのうち、2 つのテーブルで同じ従業員番号を持っているのは 648 のローだけだからです。
- 結果の順序に意味はありません。ORDER BY 句を追加すると、クエリに特定の順序を指定できます。
- ON 句によって、最終的な結果セットには含まれないカラムが組み込まれます。

生成されたジョインと ON 句

キーワード JOIN を使用し、ジョイン・タイプを指定していない場合、ON 句を使用しなければ、キー・ジョインがデフォルトです。指定のない JOIN とともに ON 句を使用すると、キー・ジョインはデフォルトにはならず、生成されたジョイン条件は何も適用されません。

たとえば、次の例はキー・ジョインです。キーワード JOIN が使用されており、ON 句もない場合はキー・ジョインがデフォルトだからです。

```
SELECT *
FROM A JOIN B;
```

次は、テーブル A とテーブル B のジョインであり、ジョイン条件 A.x = B.y も使用されています。したがって、このジョインはキー・ジョインではありません。

```
SELECT *
FROM A JOIN B ON A.x = B.y;
```

KEY JOIN または NATURAL JOIN を指定し、さらに ON 句も使用すると、最終的なジョイン条件は、生成されたジョイン条件と明示的ジョイン条件の論理積になります。たとえば、次の文にはジョイン条件が 2 つ入っています。1 つはキー・ジョインから生成されたジョイン条件で、もう 1 つは ON 句で明示的に記述されたジョイン条件です。

```
SELECT *
FROM A KEY JOIN B ON A.x = B.y;
```

キー・ジョインによって生成されたジョイン条件が A.w = B.z の場合、次の文は上の文と同等です。

```
SELECT *
FROM A JOIN B
  ON A.x = B.y
  AND A.w = B.z;
```

キー・ジョインの詳細については、「キー・ジョイン」 448 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) などの探索条件を使用できます。

例

次の例は、「在庫数以上の受注があったのはどの製品か」という質問に対する回答です。

```
SELECT DISTINCT Products.Name
FROM Products JOIN SalesOrderItems
ON Products.ID = SalesOrderItems.ProductID
   AND SalesOrderItems.Quantity > Products.Quantity;
```

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。

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」 430 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」 448 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;
```

および

```
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」 452 ページを参照してください。

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)」 436 ページを参照してください。

内部ジョインと外部ジョイン

キーワード INNER、LEFT OUTER、RIGHT OUTER、FULL OUTER は、キー・ジョイン、ナチュラル・ジョイン、ON 句付きジョインを修飾するときに使用します。デフォルトは INNER です。これらの変更子はクロス・ジョインには適用されません。

内部ジョイン

デフォルトのジョインは「内部ジョイン」です。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、次のように FROM 句を使用しても前述の例と同じ結果が得られます。

```
SELECT GivenName, Surname, OrderDate
FROM Customers JOIN SalesOrders
ORDER BY OrderDate;
```

外部ジョイン

通常は、ジョイン条件を満たす場合のみローを返すジョインを作成します。これは内部ジョインと呼ばれ、クエリ時に使用されるデフォルトのジョインです。ただし、1 つのテーブルのすべてのローを保護したい場合があります。そのような場合は「外部ジョイン」を使用します。

2 つのテーブルの左または右の「外部ジョイン」を使用すると、一方のテーブルではすべてのローが保護され、他方のテーブルにはジョイン条件が満たされないときに NULL が入力されま

す。「左外部ジョイン」では、左側のテーブルのローがすべて保護され、「右外部ジョイン」では右側テーブルのローがすべて保護されます。「全外部ジョイン」では、両方のテーブルのローがすべて保護されます。

左外部ジョインまたは右外部ジョインのそれぞれの側のテーブル式は、「保護された」テーブル式と「NULL 入力」テーブル式と呼ばれます。左外部ジョインでは、左側のテーブル式が保護テーブル式で、右側のテーブル式は NULL 入力テーブル式です。

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 =*)」 432 ページを参照してください。

例

次の文にはすべての顧客が含まれます。顧客が注文していない場合には、注文情報に対応する結果のそれぞれのカラムに NULL 値が入ります。

```
SELECT Surname, OrderDate, City
FROM Customers LEFT OUTER JOIN SalesOrders
  ON Customers.ID = SalesOrders.CustomerID
WHERE Customers.State = 'NY'
ORDER BY OrderDate;
```

Surname	OrderDate	City
Thompson	(NULL)	Bancroft
Reiser	2000-01-22	Rockwood
Clarke	2000-01-27	Rockwood
Mentary	2000-01-30	Rockland
...	...	...

この文の外部ジョインは次のように解釈できます。ここで説明しているのは概念であり、SQL Anywhere が実際に結果セットを計算する過程を示すものではありません。

- 顧客からの発注ごとにローが 1 つ返されます。1 つの発注に対して 1 つのローが返されるため、顧客が 2 つ以上注文した場合には、ローも 2 つ以上返されます。これは内部ジョインの結果と同じです。ON 条件は、customer ローと sales order ローを一致させるために使用します。この手順では、WHERE 句は使用されません。
- 注文しなかったそれぞれの顧客のローが 1 行入ります。これにより、Customers テーブルのすべてのローが確実に含まれます。これらのすべてのローに対して、SalesOrders のカラムに NULL が挿入されます。キーワード OUTER が使用されているため、これらのローは追加されますが、内部ジョインには表示されません。この手順では ON 条件も WHERE 句も使用されません。
- WHERE 句を使用して、New York 在住ではない顧客のローをすべて除外します。

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。

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。

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 にジョインされます。このとき、テーブル式 (A JOIN B) は保護され、テーブル C には NULL が入力されます。

次に、以下の文を考えてみます。A、B、C はテーブルです。


```
SELECT *
FROM A LEFT OUTER JOIN B RIGHT OUTER JOIN C;
```

この文を理解するには、まず「SQL Anywhere では文が左から右に評価され、カッコが追加される」という規則を思い出してください。結果は次のようになります。

```
SELECT *
FROM (A LEFT OUTER JOIN B) RIGHT OUTER JOIN C;
```

次に、両方のジョインが同じタイプになるように、右外部ジョインを左外部ジョインに変換します。これを行うには、右外部ジョインにあるテーブルの位置を単純に逆にします。結果は次のようになります。

```
SELECT *
FROM C LEFT OUTER JOIN (A LEFT OUTER JOIN B);
```

ネストした外部ジョインでは、A が保護テーブルであり、B が NULL 入力テーブルです。最初の外部ジョインでは C が保護テーブルです。

この結合は次のように解釈できます。

- A を B に結合します。このとき、A のローはすべて保護されます。
- 次に、C を A と B のジョインの結果に結合します。このとき、C のローはすべて保護されます。

このジョインには ON 句がないため、デフォルトのキー・ジョインになります。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」 452 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 標準に基づくものであり、あいまいさを排除するためのものです。たとえば、次の 2 つの文は構文的に正しくありません。テーブル自体が参照される前に C がジョイン条件内で参照されるからです。

```
SELECT *
FROM (A LEFT OUTER JOIN B ON B.x = C.x) JOIN C;
```

と

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = C.x, C;
```

ビューと派生テーブルの外部ジョイン

外部ジョインは、ビューと派生テーブルにも指定できます。

次に例を示します。

```
SELECT *
FROM V LEFT OUTER JOIN A ON (V.x = A.x);
```

この例は、次のように解釈できます。

- ビュー V が計算されます。
- ジョイン条件 V.x = A.x を使用して V のローをすべて保護すると、計算されたビュー V のすべてのローが A にジョインされます。

例

次の例では、ビュー V を定義します。ここで、ビュー V は、$60,000 を上回る収入がある女性の従業員 ID と部署名を返します。

```
CREATE VIEW V AS
SELECT Employees.EmployeeID, DepartmentName
  FROM Employees JOIN Departments
    ON Employees.DepartmentID = Departments.DepartmentID
  WHERE Sex = 'F' and Salary > 60000;
```

次に、このビューを使用してそれらの女性が勤務する部署と販売地区のリストを追加します。ビュー V は保護ビューであり、SalesOrders は NULL 入力です。

```
SELECT DISTINCT V.EmployeeID, Region, V.DepartmentName
  FROM V LEFT OUTER JOIN SalesOrders
    ON V.EmployeeID = SalesOrders.SalesRepresentative;
```

EmployeeID	Region	DepartmentName
243	(NULL)	R & D
316	(NULL)	R & D
529	(NULL)	R & D
902	Eastern	Sales
...	...	...

Transact-SQL の外部ジョイン (*= or =*)

注意
Transact-SQL 外部ジョイン演算子 *= と =* は旧式であるため、将来のリリースではサポートから除外されます。

SQL Anywhere では ANSI/ISO SQL 標準に基づき、キーワード LEFT OUTER、RIGHT OUTER、FULL OUTER をサポートします。また、バージョン 12 以前の Adaptive Server Enterprise との互換性を保つために、tsql_outer_joins データベース・オプションが On に設定されている場合は、LEFT OUTER と RIGHT OUTER のキーワードに対応する Transact-SQL の *= と =* もサポートします。「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.sybase.com/detail?id=1017447) を参照してください。

Transact-SQL ダイアレクトでは、FROM 句内にカンマで区切られたテーブルのリストを指定し、WHERE 句内で特殊な演算子 *= or =* を使用して外部ジョインを作成します。Adaptive Server Enterprise のバージョン 12 より前のバージョンでは、ジョイン条件を WHERE 句内に記述してください (ON はサポートされていませんでした)。

警告
外部ジョインを作成するときには、*= 構文と ON 句の構文を混在させないでください。この規則は、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。

特殊なジョイン

この項では、セルフジョイン、スター・ジョイン、派生テーブルを使用したジョインなど、特殊なジョインについて説明します。

セルフジョイン

「セルフジョイン」では、異なる相関名を使用して同一テーブルを参照することによって、テーブルがそれ自体にジョインされます。

例 1

次のセルフジョインは従業員のペアのリストを作成します。各従業員の名前が全従業員の名前との組み合わせで表示されます。

```
SELECT a.GivenName, a.Surname,
    b.GivenName, b.Surname
FROM Employees AS a CROSS JOIN Employees AS b;
```

GivenName	Surname	GivenName	Surname
Fran	Whitney	Fran	Whitney
Fran	Whitney	Matthew	Cobb
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 標準の拡張機能です。重複名を使用しても機能は追加されませんが、使用すると特定のクエリを簡単に作成できます。

重複名は、構文が意味をなすように、必ず異なるジョイン内に置きます。同一ジョイン内でテーブル名やビュー名を 2 回使用すると、2 番目のインスタンスは無視されます。たとえば、FROM A,A と FROM A CROSS JOIN A は、どちらも FROM A と解釈されます。

次の例は SQL Anywhere で有効です。A、B、C はテーブルです。この例では、テーブル A の同一インスタンスは B と C のどちらにもジョインされます。スター・ジョインでジョインを分けるときにはカンマが必要です。スター・ジョインでのカンマの使用方法は、スター・ジョインの構文特有のものです。

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = B.x,
    A LEFT OUTER JOIN C ON A.y = C.y;
```

これは、次の例と同義です。

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = B.x,
    C RIGHT OUTER JOIN A ON A.y = C.y;
```

2 つの例はどちらも次の ANSI/ISO 標準構文と同義です (カッコはオプションです)。

```
SELECT *
FROM (A LEFT OUTER JOIN B ON A.x = B.x)
LEFT OUTER JOIN C ON A.y = C.y;
```

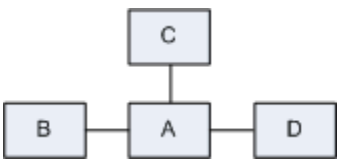
次の例では、テーブル A が 3 つのテーブル B、C、D にジョインされます。

```
SELECT *
FROM A JOIN B ON A.x = B.x,
    A JOIN C ON A.y = C.y,
    A JOIN D ON A.w = D.w;
```

上の例は、次の ANSI/ISO 標準構文と同義です (カッコはオプションです)。

```
SELECT *
FROM ((A JOIN B ON A.x = B.x)
JOIN C ON A.y = C.y)
JOIN D ON A.w = D.w;
```

複雑なジョインは図にするとわかりやすくなります。前述の例は次の図で説明できます。この図から、テーブル B、C、D がテーブル A を介してジョインされることがわかります。



注意
重複するテーブル名を使用できるのは、extended_join_syntax オプションが On (デフォルト) になっている場合だけです。

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、ます、どの情報を取り出すのかをリストします。ここでは、製品、数量、顧客名、管理者名を取り出します。次に、これらの情報を保持しているテーブルをリストします。ここでは、Products、SalesOrderItems、Customers、Employees になります。SQL Anywhere サンプル・データベースの構造 (「サンプル・データベース・スキーマ」 415 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。

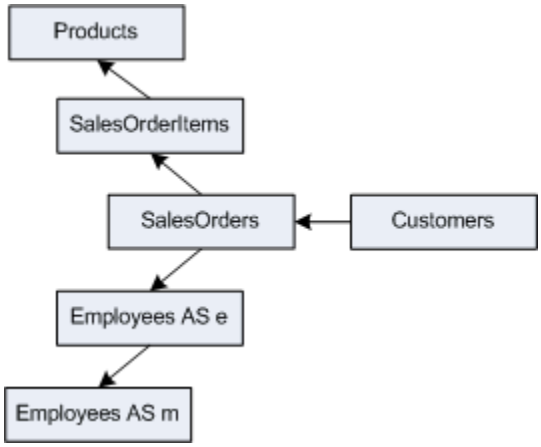
また、管理者名を取得するには、セルフジョインを作成する必要があります。Employees テーブルには、管理者の ID 番号とすべての従業員の名前はありますが、管理者名だけをリストしたカラムがないからです。詳細については、「セルフジョイン」 435 ページを参照してください。

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。



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;
```

派生テーブルに関連したジョイン

派生テーブルでは FROM 句内にクエリをネストできます。派生テーブルを使用すると、別のビューやテーブルを作成してそれにジョインすることなく、グループのグループ化を実行したり、グループを使用してジョインを構成することができます。

次の例では、内側の SELECT 文 (カッコに囲まれている) は顧客 ID 値でグループ分けされた派生テーブルを作成します。外側の SELECT 文はこのテーブルに相関名 sales_order_counts を割り当て、ジョイン条件を使用してそれを Customers テーブルとジョインします。

```
SELECT Surname, GivenName, number_of_orders
FROM Customers JOIN
   (  SELECT CustomerID, COUNT(*)
      FROM SalesOrders
      GROUP BY CustomerID  )
   AS sales_order_counts ( CustomerID, number_of_orders )
   ON ( Customers.ID = sales_order_counts.CustomerID )
WHERE number_of_orders > 3;
```

結果は各顧客の注文数を含む、4 件以上の注文をした顧客名のテーブルです。

派生テーブルのキー・ジョインについては、「ビューと派生テーブルのキー・ジョイン」 456 ページを参照してください。

派生テーブルのナチュラル・ジョインについては、「ビューと派生テーブルのナチュラル・ジョイン」 446 ページを参照してください。

派生テーブルの外部ジョインについては、「ビューと派生テーブルの外部ジョイン」 431 ページを参照してください。

適用式から生成されるジョイン

適用式を使用すると、右側が左側に依存するジョインを簡単に指定できます。たとえば、適用式を使用して、テーブル式内のローごとに 1 回ずつプロシージャまたは派生テーブルを評価できます。適用式は SELECT 文の FROM 句内に配置し、ON 句を使用することはできません。

APPLY を使用すると複数のソースからローを結合できます。この動作は JOIN に似ていますが、APPLY には ON 条件を指定することはできません。APPLY と JOIN の主な相違点は、APPLY の右側は左側の現在のローによって変わる場合があるという点です。左側のローごとに、右側が再計算され、結果のローが左側のローに結合されます。左側の 1 つのローが右側の複数の行を返すケースでは、右側から返されたローの数だけ左側が重複する結果となります。

指定できる APPLY には、CROSS APPLY と OUTER APPLY の 2 つのタイプがあります。CROSS APPLY は、右側の結果を生成する左側のローのみを返します。OUTER APPLY は、CROSS APPLY が返すすべてのローと、(右側に NULL が入力されたため) 右側からローが返されない左側のすべてのローを返します。

適用式の構文は、次のとおりです。

table-expression { **CROSS** | **OUTER** } **APPLY** *table-expression*

例

次の例では、部署 ID を入力値として受け取り、その部署内で給与が 80,000 ドルを超えるすべての従業員の名前を返すプロシージャ EmployeesWithHighSalary を作成します。

```
CREATE PROCEDURE EmployeesWithHighSalary( IN dept INTEGER )
 RESULT ( Name LONG VARCHAR )
  BEGIN
   SELECT E.GivenName || ' ' || E.Surname
      FROM Employees E
      WHERE E.DepartmentID = dept AND E.Salary > 80000;
  END;
```

次のクエリでは、OUTER APPLY を使用して Departments テーブルを EmployeesWithHighSalary プロシージャの結果にジョインし、各部署で給与が 80,000 ドルを超えるすべての従業員の名前を返します。また、このクエリでは、右側が NULL のローを返し、給与が 80,000 ドルを超える従業員が存在しない各部署も示します。

```
SELECT D.DepartmentName, HS.Name
   FROM Departments D
   OUTER APPLY EmployeesWithHighSalary( D.DepartmentID ) AS HS;
```

DepartmentName	Name
R & D	Kim Lull
R & D	David Scott
R & D	John Sheffield
Sales	Moira Kelly
Finance	Mary Anne Shea
Marketing	NULL
Shipping	NULL

次のクエリでは、CROSS APPLY を使用して Departments テーブルを EmployeesWithHighSalary プロシージャの結果にジョインします。この場合、右側に NULL が指定されたローは含まれません。

```
SELECT D.DepartmentName, HS.Name
   FROM Departments D
   CROSS APPLY EmployeesWithHighSalary( D.DepartmentID ) AS HS;
```

DepartmentName	Name
R & D	Kim Lull
R & D	David Scott
R & D	John Sheffield
Sales	Moira Kelly
Finance	Mary Anne Shea

次のクエリでは、前述のクエリと同じ結果が返されますが、CROSS APPLY の右側として派生テーブルが使用されます。

```
SELECT D.DepartmentName, HS.Name
  FROM Departments D
   CROSS APPLY (
      SELECT E.GivenName || ' ' || E.Surname
         FROM Employees E
         WHERE E.DepartmentID = D.DepartmentID AND E.Salary > 80000
   ) HS( Name );
```

参照

- 「FROM 句」 『SQL Anywhere サーバ - SQL リファレンス』
- 「クロス・ジョイン」 426 ページ
- 「内部ジョインと外部ジョイン」 428 ページ

ナチュラル・ジョイン

ナチュラル・ジョインを指定すると、同じ名前を持つカラムに基づいてジョイン条件が生成されます。生成されたジョイン条件がベース・テーブルのナチュラル・ジョインに有効になるためには、同じ名前のカラムがどちらのテーブルにも少なくとも 1 つは存在する必要があります。共通するカラム名がなければ、エラーが発生します。

テーブル A と B が共通のカラム名を 1 つ持っており、そのカラムが x であるとします。その場合は次のようになります。

```
SELECT *
FROM A NATURAL JOIN B;
```

これは、次のクエリと同義です。

```
SELECT *
FROM A JOIN B
 ON A.x = B.x;
```

テーブル A と B が共通のカラム名を 2 つ持っており、そのカラムが a と b である場合、A NATURAL JOIN B は次のクエリと同等です。

```
A JOIN B
 ON A.a = B.a
 AND A.b = B.b;
```

例 1

たとえば、テーブル Employees と Departments には共通のカラム名 DepartmentID が 1 つあるため、ナチュラル・ジョインを使用してこの 2 つのテーブルをジョインできます。

```
SELECT GivenName, Surname, DepartmentName
FROM Employees NATURAL JOIN Departments
ORDER BY DepartmentName, Surname, GivenName;
```

GivenName	Surname	DepartmentName
Janet	Bigelow	Finance
Kristen	Coe	Finance
James	Coleman	Finance
Jo Ann	Davidson	Finance
...	...	...

次の文は同義です。この文では、さきほどの例で生成されたジョイン条件が明示的に指定されています。

```
SELECT GivenName, Surname, DepartmentName
FROM Employees JOIN Departments
  ON (Employees.DepartmentID = Departments.DepartmentID)
ORDER BY DepartmentName, Surname, GivenName;
```

例 2

Interactive SQL で次のクエリを実行します。

```
SELECT Surname, DepartmentName
FROM Employees NATURAL JOIN Departments;
```

Surname	DepartmentName
Whitney	R & D
Cobb	R & D
Breault	R & D
Shishov	R & D
Driscoll	R & D
...	...

SQL Anywhere は 2 つのテーブルを参照し、共通するカラム名が DepartmentID だけであると判断します。次の ON CLAUSE は内部的に生成され、ジョインの実行に使用されます。

```
FROM Employees JOIN Departments
  ON Employees.DepartmentID = Departments.DepartmentID
```

NATURAL JOIN は ON 句を入力するための単なるショートカットで、この 2 つのクエリは同じです。

NATURAL JOIN を使用した場合のエラー

NATURAL JOIN 演算子は、等価ではないカラムを等価と見なすと問題が発生する可能性があります。たとえば、次のクエリを実行すると、意図しない結果が生成されます。

```
SELECT *
FROM SalesOrders NATURAL JOIN Customers;
```

このクエリを実行してもローは返されません。内部的に次の ON 句が生成されます。

```
FROM SalesOrders JOIN Customers
  ON SalesOrders.ID = Customers.ID
```

SalesOrders テーブル内の ID カラムは注文の ID 番号です。Customers テーブル内の ID カラムは顧客の ID 番号です。これらはどれも一致しません。もちろん、一致があったとしても意味がありません。

ON 句を使用したナチュラル・ジョイン

NATURAL JOIN を指定し、かつ ON 句内にジョイン条件を置くと、2 つのジョイン条件の論理積が生成されます。

たとえば、次の 2 つのクエリは同義です。最初のクエリではジョイン条件 Employees.DepartmentID = Departments.DepartmentID が生成されます。このクエリには明示的ジョイン条件も含まれています。

```
SELECT GivenName, Surname, DepartmentName
FROM Employees NATURAL JOIN Departments
  ON Employees.ManagerID = Departments.DepartmentHeadID;
```

次のクエリは同義です。このクエリでは、前の例で生成されたナチュラル・ジョイン条件が ON 句で指定されています。

```
SELECT GivenName, Surname, DepartmentName
FROM Employees JOIN Departments
  ON Employees.ManagerID = Departments.DepartmentHeadID
   AND Employees.DepartmentID = Departments.DepartmentID;
```

テーブル式のナチュラル・ジョイン

ナチュラル・ジョインの少なくともどちらか一方の側に複数テーブルの式が 1 つ存在する場合、SQL Anywhere は、ジョイン演算子の左右にあるカラムを比較して、同じ名前のカラムを検索することでジョイン条件を生成します。

次の文を例にとります。

```
SELECT *
FROM (A JOIN B) NATURAL JOIN (C JOIN D);
```

この文には 2 つのテーブル式があります。テーブル式 A JOIN B のカラム名がテーブル式 C JOIN D のカラム名と比較されると、一致するカラム名のうちあいまいさのないペアに対してジョイン条件が生成されます。「一致するカラム名のうちあいまいさのないペア」とは、カラム名が両方のテーブル式に出現することがあっても同一テーブル式内では 2 回出現しないという意味です。

あいまいなカラム名のペアが 1 つでもあると、エラーになります。ただし、カラム名がもう一方のテーブル式内のカラム名とも一致しなければ、カラム名が同じテーブル式で 2 回出現しても構いません。

ナチュラル・ジョイン・リスト

ナチュラル・ジョインの少なくとも片方の側にテーブル式のリストがある場合、そのリスト内の各テーブル式に対して別のジョイン条件が生成されます。

次のテーブルを考えてみます。

- テーブル A はカラム a、b、c で構成されている。
- テーブル B は、カラム a と d で構成されている。
- テーブル C はカラム d と c で構成されている。

このような場合、SQL Anywhere ではジョイン (A,B) NATURAL JOIN C によって次の 2 つのジョイン条件が生成されます。

```
ON A.c = C.c
 AND B.d = C.d
```

A-C または B-C に共通のカラム名がなければ、エラーが発行されます。

テーブル C がカラム a、d、c で構成されている場合、ジョイン (A,B) NATURAL JOIN C は無効です。その理由は、カラム a が 3 つのテーブルすべてに出現するため、ジョインがあいまいになってしまうためです。

例

次の例は、「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;
```

ビューと派生テーブルのナチュラル・ジョイン

ANSI/ISO SQL 標準の拡張機能では、ナチュラル・ジョインのどちらの側にもビューまたは派生テーブルを指定できます。次の文を考えてみます。

```
SELECT *
FROM View1 NATURAL JOIN View2;
```

View1 内のカラムは View2 内のカラムと比較されます。たとえば、両方のビューにカラム EmployeeID が出現し、それと同じ名前を持つカラムがほかになければ、生成されるジョイン条件は (View1.EmployeeID = View2.EmployeeID)(View1.EmployeeID = View2.EmployeeID) になります。

例

次の例で、ナチュラル・ジョインに使用するビューにはカラムだけでなく式を指定することができ、これらの式やカラムはナチュラル・ジョインでは同じように扱われることを説明します。まず、カラム x のあるビュー V を次のように作成します。

```
CREATE VIEW V(x) AS
SELECT R.y + 1
FROM R;
```

次に、このビューから派生テーブルへのナチュラル・ジョインを作成します。派生テーブルには、カラム x があり、相関名 T が付けられています。

```
SELECT *
FROM V NATURAL JOIN (SELECT P.y FROM P) as T(x);
```

このジョインは、次のクエリと同義です。


```
SELECT *
FROM V JOIN (SELECT P.y FROM P) as T(x) ON (V.x = T.x);
```

キー・ジョイン

一般的なジョインの多くは 2 つのテーブル間で外部キーによって関連付けられます。最も一般的なジョインは、外部キー値がプライマリ・キー値と同じになるように制限します。KEY JOIN 演算子は、外部キーの関係に基づいて 2 つのテーブルをジョインします。つまり、SQL Anywhere は、一方のテーブルのプライマリ・キー・カラムを他方のテーブルの外部キー・カラムと同等とする ON 句を生成します。キー・ジョインを使用するには、テーブル間に外部キー関係が必要になります。この関係がない場合は、エラーになります。

キー・ジョインは ON 句のショートカットで、この 2 つのクエリは同じです。ただし、ON 句は KEY JOIN でも使用できます。JOIN を指定しても CROSS、NATURAL、KEY を指定しない場合、または ON 句を使用する場合のデフォルトは、キー・ジョインです。SQL Anywhere サンプル・データベースの図では、テーブル間を結ぶ線は外部キーを表します。KEY JOIN 演算子は、図の中で 1 本の線によって 2 つのテーブルがジョインされているところならどこでも使用できます。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;
```


ON 句を使用したキー・ジョイン

KEY JOIN を指定し、かつ ON 句内にジョイン条件を置くと、2 つのジョイン条件の論理積が生成されます。次に例を示します。

```
SELECT *
FROM A KEY JOIN B
ON A.x = B.y;
```

A と B のキー・ジョインによって生成されたジョイン条件が A.w = B.z の場合、前述の例は次のクエリと同等です。

```
SELECT *
FROM A JOIN B
ON A.x = B.y AND A.w = B.z;
```

複数の外部キー関係がある場合のキー・ジョイン

外部キー関係に基づいてジョイン条件が生成されるときに、外部キー関係が 2 つ以上ある場合があります。このような場合、SQL Anywhere は、外部キーが参照するプライマリ・キー・テーブルの相関名と、外部キーの役割名とを一致させることによって、使用する外部キー関係を決定します。

次の項では、SQL Anywhere でキー・ジョインのジョイン条件が生成される過程について説明します。この情報の概要は、「キー・ジョイン操作規則」 458 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」 415 ページを参照してください。

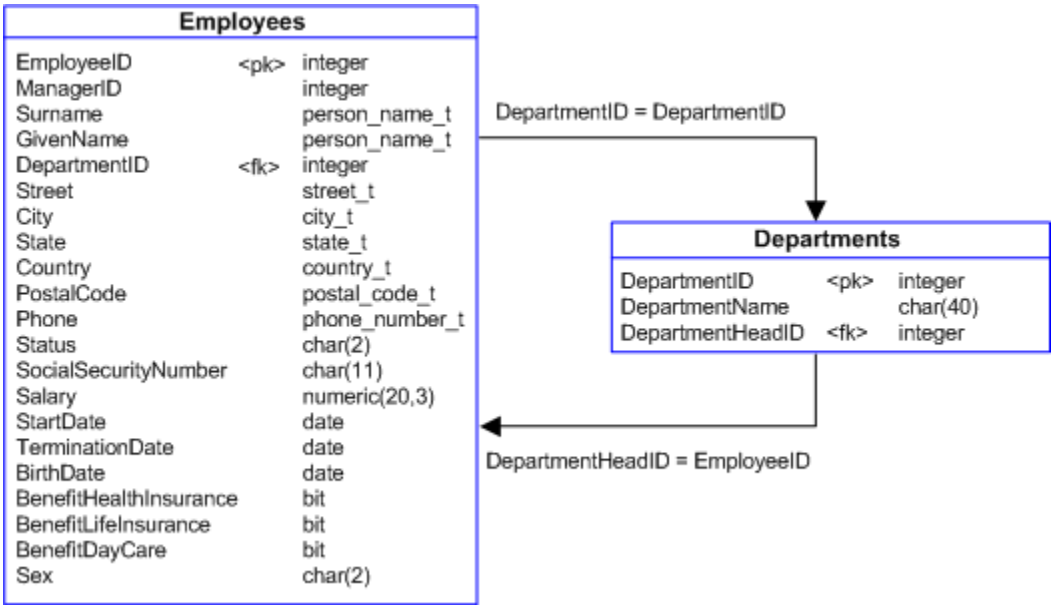
ジョイン条件の生成

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。



次のクエリはあいまいです。外部キー関係が 2 つあり、そのどちらにもプライマリ・キー・テーブル名と同じ役割名が指定されていないからです。そのため、このクエリを実行しようとしても、結果は構文エラー SQLE_AMBIGUOUS_JOIN (-147) になります。

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees KEY JOIN Departments;
```


例 2

このクエリは、Departments テーブルに相関名 FK_DepartmentID_DepartmentID を指定して例 1 のクエリを修正したものです。ここで、外部キー FK_DepartmentID_DepartmentID にはその参照テーブルと同じ名前が付けられているため、ジョイン条件を定義するために使用されます。結果には、すべての従業員の姓と所属部署が入っています。

```
SELECT Employees.Surname,
   FK_DepartmentID_DepartmentID.DepartmentName
FROM Employees KEY JOIN Departments
   AS FK_DepartmentID_DepartmentID;
```

次のクエリは上の例と同義です。この例では、Departments テーブルのエイリアスを作成する必要はありません。このクエリでは、上の例で生成されたジョイン条件と同じものが ON 句で指定されています。

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees JOIN Departments
  ON Departments.DepartmentID = Employees.DepartmentID;
```

例 3

ある部署の責任者である従業員をすべてリストする場合は、外部キー FK_DepartmentHeadID_EmployeeID を使用し、例 1 を次のように書き換えます。このクエリは、プライマリ・キー・テーブル Employees に相関名 FK_DepartmentHeadID_EmployeeID を指定することで外部キー FK_DepartmentHeadID_EmployeeID を使用します。

```
SELECT FK_DepartmentHeadID_EmployeeID.Surname, Departments.DepartmentName
FROM Employees AS FK_DepartmentHeadID_EmployeeID
   KEY JOIN Departments;
```

次のクエリは上の例と同義です。このクエリでは、上の例で生成されたジョイン条件が ON 句で指定されています。

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees JOIN Departments
  ON Departments.DepartmentHeadID = Employees.EmployeeID;
```

例 4

外部キーの役割名がプライマリ・キー・テーブル名と同じ場合、相関名は不要です。たとえば、次のように、Employees テーブルに外部キー Departments を定義するとします。

```
ALTER TABLE Employees
  ADD FOREIGN KEY Departments (DepartmentID)
  REFERENCES Departments (DepartmentID);
```

ここで、KEY JOIN が 2 つのテーブル間で指定されていれば、この外部キー関係がデフォルトのジョイン条件になります。外部キー Departments が定義されていれば、次のクエリは例 3 と同義になります。

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees KEY JOIN Departments;
```

注意

この例を Interactive SQL で実行する場合は、次の文を使って SQL Anywhere サンプル・データベースへの変更をリバースする必要があります。

```
ALTER TABLE Employees DROP FOREIGN KEY Departments;
```

テーブル式のキー・ジョイン

SQL Anywhere は、文中にあるテーブルのペアごとに外部キー関係を調べることで、テーブル式のキー・ジョインに対してジョイン条件を生成します。

次の例では 4 つのペアのテーブルがジョインされています。

```
SELECT *
FROM (A NATURAL JOIN B) KEY JOIN (C NATURAL JOIN D);
```

テーブルのペアは A-C、A-D、B-C、B-D です。SQL Anywhere はそれぞれのペアの関係を調べてから、テーブル式全体に対して生成されるジョイン条件を作成します。処理方法はテーブル式にカンマを使用するかどうかによって異なります。したがって、次の 2 つの例では生成されたジョイン条件が異なります。A JOIN B はカンマを含まないテーブル式であり、(A,B) はテーブル式のリストです。

```
SELECT *
FROM (A JOIN B) KEY JOIN C;
```

この例は、セマンティック上、次の例と異なります。

```
SELECT *
FROM (A,B) KEY JOIN C;
```

この 2 種類のジョインの動作については以下の項を参照してください。

- 「カンマを含まないテーブル式のキー・ジョイン」 452 ページ
- 「テーブル式リストのキー・ジョイン」 453 ページ

カンマを含まないテーブル式のキー・ジョイン

ジョインされている 2 つのテーブル式のどちらにもカンマが含まれていない場合、SQL Anywhere は文中にあるテーブルのペアの外部キー関係を調べ、ジョイン条件を 1 つだけ生成します。

たとえば、次のジョインには A-C と B-C という 2 つのテーブル・ペアがあります。

```
(A NATURAL JOIN B) KEY JOIN C
```

C と (A NATURAL JOIN B) をジョインするために、SQL Anywhere はテーブル・ペア A-C と B-C の外部キー関係を調べて、ジョイン条件を 1 つだけ生成します。複数の外部キー関係が存在する場合は、次のキー・ジョイン決定規則に基づき、この 2 つのペアに対して 1 つのジョイン条件を生成します。

- まず、参照先となるプライマリ・キー・テーブルのうち、その 1 つの相関名と同じ役割名を持つ単一の外部キーを指定するために A-C および B-C の両方が調べられる。この基準に合う外部キーが 1 つだけ存在する場合には、それが使用される。テーブルの相関名と同じ役割名の外部キーが 2 つ以上ある場合、そのジョインはあいまいと見なされてエラーが発行される。

- テーブルの相関名と同じ名前の外部キーがない場合は、そのテーブルの外部キー関係が検索される。見つかった外部キー関係が 1 つであれば、それが使用される。2 つ以上見つかると、そのジョインはあいまいと見なされてエラーが発行される。
- 外部キー関係がまったく存在しない場合は、エラーが発行される。

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」 449 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。

- SQL Anywhere はテーブル式 (Employees KEY JOIN Departments as FK_DepartmentID_DepartmentID) を調べ、外部キー FK_DepartmentID_DepartmentID に基づいて、ジョイン条件 Employees.DepartmentID = FK_DepartmentID_DepartmentID.DepartmentID を生成する。
- 次に Employees/SalesOrders と Departments/SalesOrders のテーブル・ペアを調べる。テーブル SalesOrders と Employees 間、および、テーブル SalesOrders と Departments 間に存在できる外部キーは 1 つだけである。それ以外の場合は、ジョインはあいまいになる。この場合、テーブル SalesOrders と Employees 間には外部キー関係が 1 つだけ存在し (FK_SalesRepresentative_EmployeeID)、テーブル SalesOrders と Departments 間に外部キーは存在しない。したがって、ジョイン条件 SalesOrders.EmployeeID = Employees.SalesRepresentative が生成される。

したがって、次のクエリは上のクエリと同義です。

```
SELECT Employees.Surname, Departments.DepartmentName
FROM ( Employees JOIN Departments
   ON ( Employees.DepartmentID = Departments.DepartmentID ) )
JOIN SalesOrders
   ON ( Employees.EmployeeID = SalesOrders.SalesRepresentative );
```

テーブル式リストのキー・ジョイン

2 つのテーブル式リストのキー・ジョインに対してジョイン条件を生成する場合、SQL Anywhere は文中のテーブルのペアを調べて、各ペアに対してジョイン条件を生成します。最後のジョイン条件は各ペアのジョイン条件の論理積です。各ペア間には外部キー関係が存在する必要があります。

次の例では、2 つのテーブル・ペア A-C と B-C をジョインします。

```
SELECT *
FROM ( A,B ) KEY JOIN C;
```

SQL Anywhere では、2 つのテーブル・ペア A-C と B-C のそれぞれに対してジョイン条件を生成することで、C と (A,B) をジョインするためのジョイン条件が生成されます。このように複数の外部キー関係が存在する場合は、次のキー・ジョイン規則に従って処理されます。

- 各ペアに対して、プライマリ・キー・テーブルの相関名と同じ名前の役割名を持つ外部キーが検索される。この基準に合う外部キーが 1 つだけ存在する場合には、それが使用される。2 つ以上見つかると、そのジョインはあいまいと見なされてエラーが発行される。
- 各ペアに、テーブルの相関名と同じ名前の外部キーがない場合は、そのテーブル間の外部キー関係が検索される。見つかった外部キー関係が 1 つであれば、それが使用される。2 つ以上見つかると、そのジョインはあいまいと見なされてエラーが発行される。
- 各ペアに、外部キー関係がまったく存在しない場合は、エラーが発行される。
- 各ペアにジョイン条件を 1 つだけ決定できる場合は、AND によってジョイン条件が組み合わされる。

「複数の外部キー関係がある場合のキー・ジョイン」 449 ページも参照してください。

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・テーブルの相関名と同じ名前の外部キーは他にないため、このテーブル・ペアのジョイン条件は FK_DepartmentID_DepartmentID を使用して作成されます。生成されるジョイン条件は

(Employees.DepartmentID = FK_DepartmentID_DepartmentID.DepartmentID) です。2 つのテーブルに関連する外部キーはもう 1 つありますが、この外部キーの名前はどちらのテーブルの名前とも異なるため、要因にはなりません。

最後のジョイン条件は、それぞれのテーブル・ペアに対して生成されたジョイン条件を 1 つにまとめます。したがって、次のクエリは同義です。

```
SELECT DISTINCT Employees.Surname,
  Departments.DepartmentName,
  SalesOrders.Region
FROM ( SalesOrders, Departments )
   JOIN Employees
   ON Employees.EmployeeID = SalesOrders.SalesRepresentative
   AND Employees.DepartmentID = Departments.DepartmentID;
```

カンマを含まないテーブル式とリストのキー・ジョイン

テーブル式リストがカンマを含まないテーブル式を持つキー・ジョインを介してジョインされる場合、SQL Anywhere はテーブル式リストの各テーブルに対してジョイン条件を生成します。

たとえば、次の文は、テーブル式リストと、カンマを含まないテーブル式のキー・ジョインです。この例では、テーブル式 C NATURAL JOIN D が指定されたテーブル A と、テーブル式 C NATURAL JOIN D が指定されたテーブル B のジョイン条件が生成されます。

```
SELECT *
FROM (A,B) KEY JOIN (C NATURAL JOIN D);
```

(A,B) はテーブル式のリストで、C NATURAL JOIN D はテーブル式です。したがって、SQL Anywhere は 2 つのジョイン条件を生成する必要があります。1 つは A-C と A-D ペアのジョイン条件で、もう 1 つは B-C と B-D ペアのジョイン条件です。このように外部キー関係が複数存在する場合は、次のキー・ジョイン規則に従って処理されます。

- テーブル・ペアの各セットに対して、プライマリ・キー・テーブルの 1 つの相関名と同じ名前の役割名を持つ外部キーが検索される。この基準に合う外部キーが 1 つだけ存在する場合には、それが使用される。2 つ以上見つかると、そのジョインはあいまいになりエラーが発行される。
- テーブル・ペアの各セットに、テーブルの相関名と同じ名前の外部キーがない場合は、そのテーブル間の外部キー関係が検索される。存在する外部キー関係が 1 つだけであれば、それが使用される。2 つ以上見つかると、そのジョインはあいまいになりエラーが発行される。
- テーブル・ペアの各セットに、外部キー関係がまったく存在しない場合は、エラーが発行される。
- 各ペアにジョイン条件を 1 つだけ決定できる場合は、キーワード AND によってジョイン条件が組み合わされる。

例 1

次の 5 つのテーブルのジョインを考えてみます。

```
((A,B) JOIN (C NATURAL JOIN D) ON A.x = D.y) KEY JOIN E
```

この場合、(A,B) と E の間の条件、または C NATURAL JOIN D と E の間の条件のどちらか 1 つが生成されることで、E に対するキー・ジョイン条件が生成されます。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」 452 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・ペア内では有効な外部キー関係が検出さ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」 453 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 つだけ作成されるため、C-E と D-E のペア内で検出される必要がある外部キー関係は 1 つだけです。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」 452 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 つだけ存在する (SalesOrders.SalesRepresentative = Employees.EmployeeID)。
- テーブル・ペア FK_DepartmentID_DepartmentID/Employees と FK_DepartmentID_DepartmentID/d の間には外部キー関係が 1 つだけ存在する (FK_DepartmentID_DepartmentID.DepartmentID = Employees.DepartmentID)。

この例は次の文と同義です。次の例では、相関名 Departments AS FK_DepartmentID_DepartmentID を作成する必要はありません。相関名が必要になるのは、Employees と Departments のジョインに使用する 2 つの外部キーを明示する場合だけです。

```
SELECT DISTINCT Employees.Surname,
   Departments.DepartmentName
FROM ( SalesOrders, Departments )
   JOIN ( Employees JOIN Departments AS d
      ON Employees.EmployeeID = d.DepartmentHeadID )
   ON SalesOrders.SalesRepresentative = Employees.EmployeeID
      AND Departments.DepartmentID = Employees.DepartmentID;
```

ビューと派生テーブルのキー・ジョイン

キー・ジョインにビューまたは派生テーブルが含まれている場合、SQL Anywhere ではテーブルと同じ基本手順に従います。ただし、次の点が異なります。

- 各キー・ジョインについて、クエリとビューの FROM 句内のテーブルのペアが調べられ、すべてのペアのセットに対してジョイン条件が 1 つ生成される。この場合、ビュー内の FROM 句にカンマやジョインのキーワードが含まれているかどうかは考慮されない。
- ビューまたは派生テーブルの相関名と同じ役割名を持つ外部キーに基づいてテーブルがジョインされる。
- キー・ジョイン内にビューまたは派生テーブルが含まれる場合、ビューまたは派生テーブル定義には UNION、INTERSECT、EXCEPT、ORDER BY、DISTINCT、GROUP BY などの集合関数や TOP、FIRST、START AT、FOR XML などの Window 関数を含めることはできない。これらが 1 つでも入っていると、エラーが返される。さらに、派生テーブルは再帰テーブル式として定義することはできない。

  派生テーブルとビューの機能は同じです。派生テーブルの場合、定義済みのビューを参照しないで、テーブルの定義を文中に記述する点だけが異なります。

  再帰テーブル式の詳細については、「再帰共通テーブル式」 469 ページと「RecursiveTable アルゴリズム (RT)」 634 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」 452 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。

前述のビュー定義のいずれかを使用すると、View1 KEY JOIN B の処理を次のように解釈できます。

テーブル・ペア C-B と D-B が調べられ、ジョイン条件が 1 つだけ生成されます。複数の外部キー関係が存在する場合には、次のキー・ジョイン決定規則に基づいてジョイン条件が生成されます。

- まず、ビューの相関名と同じ役割名を持つ 1 つの外部キーに対して C-B と D-B が両方とも調べられる。この基準に合う外部キーが 1 つだけ存在する場合には、それが使用される。ビューの相関名と同じ役割名の外部キーが 2 つ以上ある場合、そのジョインはあいまいと見なされてエラーが発行される。
- ビューの相関名と同じ名前の外部キーがない場合は、そのテーブル間の外部キー関係が検索される。見つかった外部キー関係が 1 つであれば、それが使用される。2 つ以上見つかると、そのジョインはあいまいと見なされてエラーが発行される。
- 外部キー関係がまったく存在しない場合は、エラーが発行される。

ここで生成されたジョイン条件が B.y = D.z であると想定します。次に示す元のジョインを展開できます。たとえば、次の 2 つの文は同等です。

```
SELECT *
FROM View1 KEY JOIN B;

SELECT *
FROM View1 JOIN B ON B.y = View1.z;
```

「複数の外部キー関係がある場合のキー・ジョイン」 449 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 );
```

次のクエリは前述のクエリと同等です。

```
SELECT *
FROM V JOIN ( SalesOrders,
   Departments FK_DepartmentID_DepartmentID )
ON ( V.EmployeeID = SalesOrders.SalesRepresentative
AND V.DepartmentID =
    FK_DepartmentID_DepartmentID.DepartmentID );
```

キー・ジョイン操作規則

以下の規則は、これまでに説明した情報をまとめたものです。

規則 1：2 つのテーブルのキー・ジョイン

この規則は、A KEY JOIN B に適用されます。ここで、A と B はベース・テーブルまたはテンポラリ・テーブルです。

1. B を参照する A のすべての外部キーを見つける。

   テーブル B の相関名が役割名になる外部キーが存在する場合には、それを優先外部キーとする。

2. A を参照する B のすべての外部キーを見つける。

   テーブル A の相関名が役割名になる外部キーが存在する場合には、それを優先外部キーとする。

3. 優先キーが 2 つ以上存在する場合、そのジョインはあいまいである。構文エラー SQLE_AMBIGUOUS_JOIN (-147) が発行される。

4. 優先キーが 1 つだけ存在する場合には、この KEY JOIN 式に生成されたジョイン条件を定義するために、この外部キーが選択される。

5. 優先キーがまったくない場合は、A と B 間にある他の外部キーが使用される。

   - A と B の間に外部キーが 2 つ以上ある場合、そのジョインはあいまいである。構文エラー SQLE_AMBIGUOUS_JOIN (-147) が発行される。
   - 外部キーが 1 つだけ存在する場合には、この KEY JOIN 式に生成されたジョイン条件を定義するために、この外部キーが選択される。
   - 外部キーがまったくない場合、ジョインは無効であり、エラーが生成される。

規則 2：カンマを含まないテーブル式のキー・ジョイン

この規則は、A KEY JOIN B に適用されます。ここで、A と B はカンマを含まないテーブル式です。

1. テーブルの各ペア (テーブル式 A から 1 つとテーブル式 B から 1 つ) について、すべての外部キーをリストし、テーブル間のすべての優先キーにマークする。優先キー決定規則は上記の規則 1 で指定されている。

2. 優先キーが 2 つ以上存在する場合、そのジョインはあいまいである。構文エラー SQLE_AMBIGUOUS_JOIN (-147) が発行される。

3. 優先キーが 1 つだけ存在する場合には、この KEY JOIN 式に生成されたジョイン条件を定義するために、この外部キーが選択される。

4. 優先キーがまったくない場合は、テーブルのペア間にある他の外部キーが使用される。

   - 外部キーが 2 つ以上存在する場合、そのジョインはあいまいである。構文エラー SQLE_AMBIGUOUS_JOIN (-147) が発行される。
   - 外部キーが 1 つだけ存在する場合には、この KEY JOIN 式に生成されたジョイン条件を定義するために、この外部キーが選択される。
   - 外部キーがまったくない場合、ジョインは無効であり、エラーが生成される。

規則 3：テーブル式リストのキー・ジョイン

この規則は、(A1, A2, ...) KEY JOIN (B1, B2, ...) に適用されます。ここで、A1、B1 はカンマを含まないテーブル式です。

1. テーブル式 Ai と Bj の各ペアに対して、規則 1 または規則 2 を適用し、テーブル式 (Ai KEY JOIN Bj) 用にユニークに生成されたジョイン条件を検出する。テーブル式のペアの KEY JOIN のいずれかが規則 1 または規則 2 によってあいまいであると判断されると、構文エラーになる。
2. この KEY JOIN 式の生成されたジョイン条件は、手順 1 のジョイン条件の論理積である。

規則 4：カンマを含まないテーブル式とリストのキー・ジョイン

この規則は、(A1, A2, ...) KEY JOIN (B1, B2, ...) に適用されます。ここで、A1、B1 はカンマを含む可能性のあるテーブル式です。

1. テーブル式 Ai と Bj の各ペアに対して、規則 1、規則 2、または規則 3 を適用し、テーブル式 (Ai KEY JOIN Bj) 用にユニークに生成されたジョイン条件を検出する。テーブル式のペアの KEY JOIN のいずれかが規則 1、規則 2、または規則 3 によってあいまいであると判断されると、構文エラーになる。
2. この KEY JOIN 式の生成されたジョイン条件は、手順 1 のジョイン条件の論理積である。

共通テーブル式

目次

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。

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。

参照

- 「複数の相関名の指定」 463 ページ
- 「複数のテーブル式の使用」 464 ページ
- 「共通テーブル式を使用できる条件」 465 ページ

複数の相関名の指定

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;
```

参照

- 「共通テーブル式の使用」 462 ページ
- 「複数のテーブル式の使用」 464 ページ
- 「共通テーブル式を使用できる条件」 465 ページ

複数のテーブル式の使用

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
      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 );
```

参照

- 「共通テーブル式の使用」 462 ページ
- 「複数の相関名の指定」 463 ページ
- 「共通テーブル式を使用できる条件」 465 ページ

共通テーブル式を使用できる条件

共通テーブル式はクエリ本体または任意のサブクエリ内から参照可能ですが、共通テーブル式を定義できるのは 3 か所のみです。

- **最上位レベルの SELECT 文**　共通テーブル式は、最上位レベルの SELECT 文内で使用できますが、サブクエリ内では使用できません。

```
WITH 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 );
```

参照

- 「共通テーブル式の使用」 462 ページ
- 「複数の相関名の指定」 463 ページ
- 「複数のテーブル式の使用」 464 ページ

共通テーブル式の一般的な使用例

一般的に、共通テーブル式は、単一のクエリ内でテーブル式を複数回使用しなければならない場合に役立ちます。よくある次のような状況では、共通テーブル式の使用が適切です。

- 複数の集合関数を含むクエリ
- プログラム変数への参照を含める必要がある、プロシージャ内のビュー
- 値のセットを格納するためにテンポラリ・ビューを使用するクエリ

このリストはすべてを網羅したものではありません。共通テーブル式が役に立つ状況は、ほかにも数多くあります。

複数の集合関数

共通テーブル式は、単一のクエリ内で複数レベルの集約を使用しなければならない場合に役立ちます。前項で使用した例がこれに当てはまります。そのタスクは、従業員数が最も多い部署の部署 ID を取り出すことでした。そのために、COUNT 集合関数を使用して各部署の従業員数を計算し、MAX 関数を使用して最も従業員数の多い部署を選択しています。

支払い給与総額が最も多い部を判別するクエリを記述する場合も、似たような状況となります。SUM 集合関数を使用して各部署の支払い給与総額を計算し、MAX 関数を使用してどの部署が一番多いかを判別します。クエリに両方の関数があることが、共通テーブル式が役立つかどうかを判断する手がかりとなります。

```
WITH 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 )
```

集合関数の詳細については、「Window 集合関数」 501 ページを参照してください。

プログラム変数を参照するビュー

プログラム変数への参照を含むビューを作成すると便利な場合があります。たとえば、特定の顧客を識別する変数をプロシージャ内に定義するとします。その顧客の購入履歴を問い合わせる場合、同様の情報に何度もアクセスしたり、複数の集合関数を使用する必要が出じるのであれば、その特定の顧客に関する情報を含むビューを作成すると便利です。

ビューのスコープをプロシージャのスコープに制限する方法はないため、プログラム変数を参照するビューを作成することはできません。ビューは、一度作成すると、このような目的以外にも使用できます。しかし、プロシージャのクエリ内で共通テーブル式を使用することもできます。共通テーブル式のスコープはその文に制限されるため、変数を参照してもあいまい性はなく、変数の参照は許可されます。

次の文は、SQL Anywhere サンプル・データベース内のさまざまな販売担当者の総売上高を選択します。

```
SELECT GivenName || ' ' || Surname AS sales_rep_name,
      SalesRepresentative AS sales_rep_id,
      SUM( p.UnitPrice * i.Quantity ) AS total_sales
FROM Employees LEFT OUTER JOIN SalesOrders AS o
           INNER JOIN SalesOrderItems AS I
           INNER JOIN Products AS p
WHERE OrderDate BETWEEN '2000-01-01' AND '2001-12-31'
GROUP BY SalesRepresentative, GivenName, Surname;
```

前述のクエリは、次のプロシージャ内で使用されている共通テーブル式の基礎となります。調査する販売担当者の ID 番号と年度が入力パラメータです。次のプロシージャからわかるように、WITH 句内では、プロシージャ・パラメータと宣言済みのすべてのローカル変数を参照できます。

```
CREATE PROCEDURE sales_rep_total (
  IN rep  INTEGER,
  IN yyyy INTEGER )
BEGIN
  DECLARE StartDate DATE;
  DECLARE EndDate   DATE;
  SET StartDate = YMD( yyyy,  1,  1 );
  SET EndDate = YMD( yyyy, 12, 31 );
  WITH total_sales_by_rep ( sales_rep_name,
                        sales_rep_id,
                        month,
                        order_year,
                        total_sales ) AS
  ( SELECT GivenName || ' ' || Surname AS sales_rep_name,
         SalesRepresentative AS sales_rep_id,
         month( OrderDate),
         year( OrderDate ),
         SUM( p.UnitPrice * i.Quantity ) AS total_sales
    FROM Employees LEFT OUTER JOIN SalesOrders o
                 INNER JOIN SalesOrderItems I
                 INNER JOIN Products p
    WHERE OrderDate BETWEEN StartDate AND EndDate
         AND SalesRepresentative = rep
    GROUP BY year( OrderDate ), month( OrderDate ),
          GivenName, Surname, SalesRepresentative )
  SELECT sales_rep_name,
       monthname( YMD(yyyy, month, 1) ) AS month_name,
       order_year,
       total_sales
  FROM total_sales_by_rep
  WHERE total_sales =
    ( SELECT MAX( total_sales) FROM total_sales_by_rep )
  ORDER BY order_year ASC, month ASC;
END;
```

次の文を実行すると、前述のプロシージャが呼び出されます。

```
CALL sales_rep_total( 129, 2000 );
```

値を格納するビュー

SELECT 文内またはプロシージャ内に特定の値のセットを格納すると便利な場合があります。たとえば、ある会社で販売担当者の成績を分析する際の単位として、四半期ではなく、1 年を 3 期に分けた期間を採用しているとします。四半期という日付の単位は組み込まれていますが、1 年を 3 期に分けた日付の単位はないため、プロシージャ内で日付を格納する必要があります。

```
WITH thirds ( q_name, q_start, q_end ) AS
( SELECT 'T1', '2000-01-01', '2000-04-30' UNION
  SELECT 'T2', '2000-05-01', '2000-08-31' UNION
  SELECT 'T3', '2000-09-01', '2000-12-31' )
SELECT q_name,
       SalesRepresentative,
       count(*) AS num_orders,
       SUM( p.UnitPrice * i.Quantity ) AS total_sales
FROM thirds LEFT OUTER JOIN SalesOrders AS o
    ON OrderDate BETWEEN q_start and q_end
                KEY JOIN SalesOrderItems AS I
                KEY JOIN Products AS p
 GROUP BY q_name, SalesRepresentative
 ORDER BY q_name, SalesRepresentative;
```

この方法は、値を定期的に保守する必要があるため、注意して使用する必要があります。たとえば、前述の文は、他の年に対して使用する場合は修正する必要があります。

この方法をプロシージャ内で適用することもできます。次の例は、対象の年を引数として受け取るプロシージャを宣言しています。

```
CREATE PROCEDURE sales_by_third ( IN y INTEGER )
BEGIN
  WITH thirds ( q_name, q_start, q_end ) AS
  ( SELECT 'T1', YMD( y, 01, 01), YMD( y, 04, 30 ) UNION
    SELECT 'T2', YMD( y, 05, 01), YMD( y, 08, 31 ) UNION
    SELECT 'T3', YMD( y, 09, 01), YMD( y, 12, 31 ) )
  SELECT q_name,
         SalesRepresentative,
         count(*) AS num_orders,
         SUM( p.UnitPrice * i.Quantity ) AS total_sales
  FROM thirds LEFT OUTER JOIN SalesOrders AS o
    ON OrderDate BETWEEN q_start and q_end
          KEY JOIN SalesOrderItems AS I
          KEY JOIN Products AS p
  GROUP BY q_name, SalesRepresentative
  ORDER BY q_name, SalesRepresentative;
END;
```

次の文を実行すると、前述のプロシージャが呼び出されます。

```
CALL sales_by_third (2000);
```

再帰共通テーブル式

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。

どの部署の従業員数が最も多いかを特定する問題を例にとります。SQL Anywhere サンプル・データベースの Employees テーブルは、架空の会社で働くすべての従業員をリストし、それぞれがどの部署で働いているかを示します。次のクエリは、部署 ID コードと各部署の従業員の総数をリストします。

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

このクエリは、次に示すように従業員の最も多い部署を抽出するために使用できます。

```
SELECT DepartmentID, n
FROM ( SELECT DepartmentID, COUNT( * ) AS n
      FROM Employees GROUP BY DepartmentID ) AS a
WHERE a.n =
  ( SELECT MAX( n )
    FROM ( SELECT DepartmentID, COUNT( * ) AS n
          FROM Employees GROUP BY DepartmentID ) AS b );
```

この文は正確な結果をもたらしますが、いくつか欠点もあります。最初の欠点は、サブクエリの繰り返しによってこの文の効率が悪くなることです。2 つ目の欠点は、この文ではサブクエリ間の関係が明確でないことです。

このような問題を回避する 1 つの方法として、ビューを作成し、そのビューを使用してクエリを再作成します。この方法によって、前述の問題を回避できます。

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

SELECT DepartmentID, n
  FROM CountEmployees
  WHERE n = ( SELECT MAX( n )
            FROM CountEmployees );
```

この方法の欠点は、ビューの作成時にデータベース・サーバがシステム・テーブルを更新しなければならないため、オーバヘッドが発生することです。ビューが頻繁に使用される場合は、この方法は合理的です。しかし、ビューが特定の SELECT 文内で 1 度しか使用されない場合は、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。共通テーブル式の詳細については、「共通テーブル式の使用」 462 ページを参照してください。

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。

階層データの選択

クエリの記述方法によって、再帰レベル数を制限できます。たとえば、レベル数を制限して、トップレベルの管理者のみを返すことができますが、指揮系統が予期したよりも長い場合、除外される従業員も出てきます。レベル数を制限しない場合は、従業員が除外されることはありません。ただし、ある従業員が直接または間接的に自分自身に対してレポートするなど、実行に循環が必要な場合は、無限の再帰が発生する可能性があります。この状況は、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。

次のクエリは、管理レベルで従業員をリストする方法を示しています。level 0 は、管理者を持たない従業員を表します。level 1 は、level 0 の管理職の 1 人に直接報告を行う従業員を表し、level 2 は、level 1 の管理職に直接報告を行う従業員を表します。以下、同様に続きます。

```
WITH RECURSIVE
  manager ( EmployeeID, ManagerID,
          GivenName, Surname, mgmt_level ) AS
( ( SELECT EmployeeID, ManagerID,       -- initial subquery
          GivenName, Surname, 0
    FROM Employees AS e
    WHERE ManagerID = EmployeeID )
  UNION ALL
  ( SELECT e.EmployeeID, e.ManagerID,   -- recursive subquery
          e.GivenName, e.Surname, m.mgmt_level + 1
    FROM Employees AS e JOIN manager AS m
     ON   e.ManagerID =  m.EmployeeID
      AND e.ManagerID <> e.EmployeeID
      AND m.mgmt_level < 20 ) )
SELECT * FROM manager
ORDER BY mgmt_level, Surname, GivenName;
```

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獲す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ますが、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

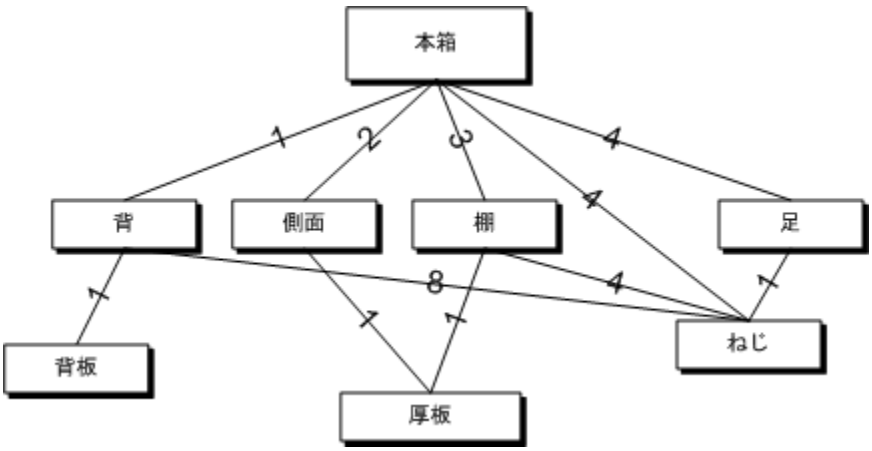
再帰共通テーブル式に適用される制限を次に示します。

- 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。このため、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。

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。



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

コンポーネント	サブコンポーネント	数量
棚	ねじ	4

次の文を実行して、bookcase テーブルを作成し、コンポーネント・データとサブコンポーネント・データを挿入します。

```
CREATE TABLE bookcase (
     component      VARCHAR(9),
     subcomponent   VARCHAR(9),
     quantity       INTEGER,
   PRIMARY KEY ( component, subcomponent )
);
INSERT INTO bookcase
  SELECT 'bookcase', 'back',      1 UNION
  SELECT 'bookcase', 'side',      2 UNION
  SELECT 'bookcase', 'shelf',     3 UNION
  SELECT 'bookcase', 'foot',      4 UNION
  SELECT 'bookcase', 'screw',     4 UNION
  SELECT 'back',     'backboard', 1 UNION
  SELECT 'back',     'screw',     8 UNION
  SELECT 'side',     'plank',     1 UNION
  SELECT 'shelf',    'plank',     1 UNION
  SELECT 'shelf',    'screw',     4;
```

次の文を実行して、本棚の組み立てに必要なコンポーネント、サブコンポーネント、およびその数量のリストを生成します。

```
SELECT * FROM bookcase
ORDER BY component, subcomponent;
```

次の文を実行して、本棚の組み立てに必要なサブコンポーネントとその数量のリストを生成します。

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT component, subcomponent, quantity
  FROM bookcase WHERE component = 'bookcase'
    UNION ALL
  SELECT b.component, b.subcomponent, p.quantity * b.quantity
  FROM parts p JOIN bookcase b ON p.subcomponent = b.component )
SELECT subcomponent, SUM( quantity ) AS quantity
FROM parts
WHERE subcomponent NOT IN ( SELECT component FROM bookcase )
GROUP BY subcomponent
ORDER BY subcomponent;
```

このクエリの結果を次に示します。

subcomponent	quantity
backboard	1
foot	4
plank	5

subcomponent	quantity
screw	24

また、このクエリを別の再帰レベルを実行するように書き直すこともできます。こうすると、メインの SELECT 文内でサブクエリを記述する必要がなくなります。次のクエリの結果は、前述のクエリの結果とまったく同じです。

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT component, subcomponent, quantity
  FROM bookcase WHERE component = 'bookcase'
    UNION ALL
  SELECT p.subcomponent, b.subcomponent,
    IF b.quantity IS NULL
    THEN p.quantity
    ELSE p.quantity * b.quantity
    ENDIF
  FROM parts p LEFT OUTER JOIN bookcase b
  ON p.subcomponent = b.component
    WHERE p.subcomponent IS NOT NULL
 )
SELECT component, SUM( quantity ) AS quantity
FROM parts
WHERE subcomponent IS NULL
GROUP BY component
ORDER BY component;
```

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。「構成部品の問題」 472 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 SMALL INT と見なされるが、quantity カラムのデータ型は INT である。

2 番目のカラムには、キャストは不要です。初期クエリのこのカラムがすでに文字列であるためです。

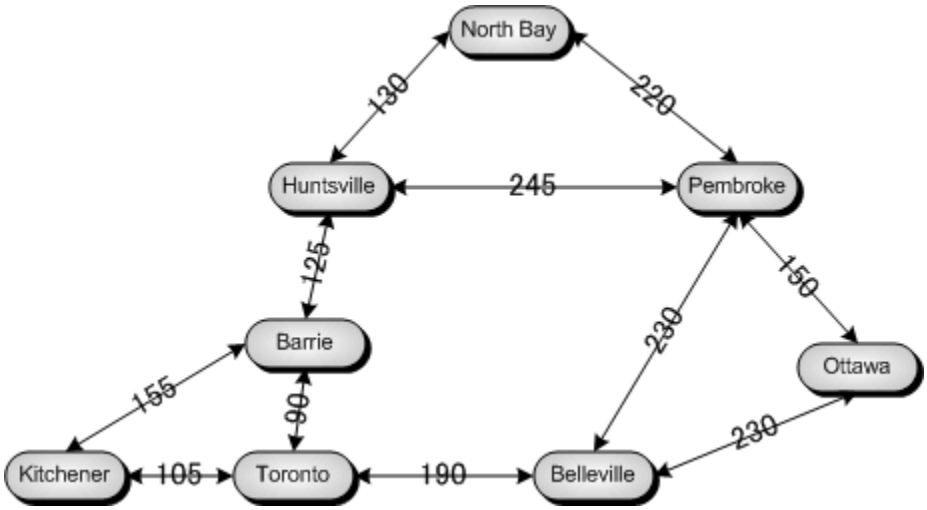
初期サブクエリでデータ型をキャストすれば、クエリを意図したとおりに動作させることができます。

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT CAST( NULL AS VARCHAR ),  -- CASTs must be used
         'bookcase',               -- to declare the
         CAST( 1   AS INT )        -- correct datatypes
     UNION ALL
  SELECT b.component, b.subcomponent,
        p.quantity * b.quantity
  FROM parts p JOIN bookcase b
    ON p.subcomponent = b.component )
SELECT * FROM parts
ORDER BY component, subcomponent;
```

最短距離の問題

再帰共通テーブル式を使用して、命令グラフ上の望ましいパスを見つけられます。データベース・テーブルの各ローは、命令エッジを表します。各ローは、出発地から目的地まで移動するときの出発地、目的地、およびコストを示します。問題の種類によって、コストは距離であったり、所要時間であったり、または別の基準であることも考えられます。再帰を使用すると、このグラフを介して可能なルートを探索できます。そして、いくつかの可能なルートから目的のルートを選ぶことができます。

たとえば、キッチナー市とペンブローク市の間を車で移動するときの望ましいルートを見つける問題を考えてみます。いくつも可能なルートがあり、それぞれ異なるいくつかの都市を中継点として通ります。目的は、最短ルートをいくつか見つけ、それらを別の適当な選択肢と比較することです。



まず、このグラフのエッジを表すテーブルを定義し、各エッジに対して 1 ローを挿入します。このグラフのすべてのエッジは 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。効率性を考慮して、次の条件のいずれかに該当した場合、ルートは終了します。

- パスが出発地に戻る場合。
- パスが直前の地点に戻る場合。
- パスが最後の目的地に達した場合。

この例では、キッチナー市に戻るパスはなく、全てのパスはペンブローク市に達した場合、終了します。

再帰クエリを使用して環状のグラフを探索する場合は、再帰クエリが適切に終了することを確認することが重要です。この例の場合、前述の条件では不十分です。ルートに 2 つの中継地の間を行ったり来たりする移動が無制限に含まれる可能性があるからです。次の再帰クエリでは、指定したルート内の最大セグメント数を 7 つまでに制限することで、ルートの終わりを保証しています。

このクエリ例のポイントは現実的なルートを選択することであるため、メイン・クエリでは、距離が最短ルート比 50% 増未満のルートのみを選択しています。

```
WITH RECURSIVE
   trip ( route, destination, previous, distance, segments ) AS
( SELECT CAST( origin || ', ' || destination AS VARCHAR(256) ),
        destination, origin, distance, 1
  FROM travel
  WHERE origin = 'Kitchener'
    UNION ALL
  SELECT route || ', ' || v.destination,
        v.destination,           -- current endpoint
        v.origin,                -- previous endpoint
        t.distance + v.distance,  -- total distance
        segments + 1             -- total number of segments
  FROM trip t JOIN travel v ON t.destination = v.origin
  WHERE v.destination <> 'Kitchener'  -- Don't return to start
    AND v.destination <> t.previous   -- Prevent backtracking
    AND v.origin      <> 'Pembroke'   -- Stop at the end
    AND segments                      -- TERMINATE RECURSION!
        < ( SELECT count(*)/2 FROM travel ) )
SELECT route, distance, segments FROM trip
WHERE destination = 'Pembroke' AND
     distance < 1.5 * ( SELECT MIN( distance )
                        FROM trip
                        WHERE destination = 'Pembroke' )
ORDER BY distance, segments, route;
```

この文を前述のデータ・セットに対して実行すると、次のような結果となります。

route	distance	segments
Kitchener, Barrie, Huntsville, Pembroke	525	3
Kitchener, Toronto, Belleville, Pembroke	525	3
Kitchener, Toronto, Barrie, Huntsville, Pembroke	565	4
Kitchener, Barrie, Huntsville, North Bay, Pembroke	630	4
Kitchener, Barrie, Toronto, Belleville, Pembroke	665	4
Kitchener, Toronto, Barrie, Huntsville, North Bay, Pembroke	670	5
Kitchener, Toronto, Belleville, Ottawa, Pembroke	675	4

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
```

```
    WHERE v.destination <> initial     -- Don't return to start
      AND v.destination <> t.previous  -- Prevent backtracking
      AND v.origin      <> final       -- Stop at the end
      AND segments                     -- TERMINATE RECURSION!
          < ( SELECT count(*)/2 FROM travel ) )
  SELECT route, distance, segments FROM trip
  WHERE destination = final AND
       distance < 1.4 * ( SELECT MIN( distance )
                          FROM trip
                          WHERE destination = final )
  ORDER BY distance, segments, route;
END;
```

次の文を実行すると、前述のプロシージャが呼び出されます。

```
CALL best_routes ( 'Pembroke', 'Kitchener' );
```


OLAP のサポート

目次

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 句の拡張では、入力ローを複数の方法で分割できるため、さまざまなグループを連結する結果セットを得ることができます。データ・マイニング用に散在した多次元結果セット (「データ・キューブ」とも呼ばれる) を作成することもできます。さらに、この拡張により、サブ合計のローと総合計のローを使用して、分析をより便利にすることができます。「GROUP BY 句の拡張」 484 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」 494 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」の概念がサポートされています。ウィンドウの移動とともにウィンドウ内のデータに対する追加の計算を実行できるため、セマンティック上同等なセルフジョイン・クエリや相関サブクエリを使用する方法よりも効率的な方法で、詳細な分析を実行できます。

  たとえば Window 集合関数を GROUP BY 句の CUBE、ROLLUP、GROUPING SETS 拡張と組み合わせることで、単一の SQL 文内における百分位数、移動平均、累積合計を効率的に計算できます。それ以外の方法では、セルフジョイン、相関サブクエリ、テンポラリ・テーブル、またはこれら 3 つを組み合わせなければなりません。

Window 集合関数を使用すると、ダウ・ジョーンズ工業平均株価の四半期の移動平均や、部門ごとの全従業員と給与の累積合計などの情報を取得できます。また、平方偏差、標準偏差、相関、回帰などの測定を計算することもできます。「Window 集合関数」 501 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」 519 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』
- 「ClusteredHashGroupBy アルゴリズム (GrByHClust)」 631 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

	City	State	CompanyName	Cnt
3	(NULL)	(NULL)	Westend Dealers	1
4	(NULL)	(NULL)	Toto's Active Wear	1
5	(NULL)	(NULL)	North Land Trading	1
6	(NULL)	(NULL)	The Ultimate	1
7	(NULL)	(NULL)	Molly's	1
8	(NULL)	(NULL)	Overland Army Navy	1
9	(NULL)	(NULL)	Out of Town Sports	1
10	'Pembroke'	'MB'	(NULL)	4
11	'Petersburg'	'KS'	(NULL)	1
12	'Drayton'	'KS'	(NULL)	3

ロー 2 ～ 9 は、CompanyName ごとにグループ化して生成されたローで、ロー 10 ～ 12 は、City と State の組み合わせでグループ化されたローです。ロー 1 は、一致したカッコ () のペアを使用して指定される空のグループ化セットで表される総合計です。空のグループ化セットは、GROUP BY に対する入力のすべてのローの分割 1 つを表します。

NULL 値は、グループ化セットで使用されない任意の式のプレースホルダとして使用されていることに注意してください。これは、結果セットが結合可能である必要があるためです。たとえば、ロー 2 ～ 9 は、クエリの 2 番目のグループ化セット (CompanyName) から得られます。グループ化セットには式として City または State が含まれないため、ロー 2 ～ 9 では City と State の値にプレースホルダの NULL が含まれますが、CompanyName の値には CompanyName に出現する重複しない値が含まれます。

NULL はプレースホルダとして使用されるため、プレースホルダの NULL とデータ内の実際の NULL を混乱しがちです。プレースホルダの NULL を NULL データと区別するには、GROUPING 関数を使用してください。「GROUPING 関数を使用したプレースホルダの NULL の検出」 492 ページを参照してください。

例

次の例では、クエリから返される結果を調整するために GROUPING SETS を使用する方法と、結果をわかりやすく整理するために ORDER BY 句を使用する方法を示します。クエリは、各年 (Year) の四半期 (Quarter) ごとの注文の合計数と、年 (Year) ごとの合計数を返します。年 (Year)、四半期 (Quarter) の順で並べることで、結果を理解しやすくなります。

```
SELECT Year( OrderDate ) AS Year,
       Quarter( OrderDate ) AS Quarter,
       COUNT (*) AS Orders
FROM SalesOrders
GROUP BY GROUPING SETS ( ( Year, Quarter ), ( Year ) )
ORDER BY Year, Quarter;
```

このクエリは、次の結果を返します。

	Year	Quarter	Orders
1	2000	(NULL)	380
2	2000	1	87
3	2000	2	77
4	2000	3	91
5	2000	4	125
6	2001	(NULL)	268
7	2001	1	139
8	2001	2	119
9	2001	3	10

ロー 1 と 6 は、それぞれ 2000 年と 2001 年の注文の小計です。ロー 2 ～ 5 とロー 7 ～ 9 は、小計ローのディテール・ローに当たります。つまり、これらのローは、四半期ごとと年ごとの注文の合計数を示します。

この結果セットには、すべての年のすべての四半期の総合計がありません。総合計が含まれるようにするには、クエリで GROUPING SETS 指定に空のグループ化指定「()」を含める必要があります。

空のグループ化指定の使用

GROUP BY 句で空の GROUPING SETS 指定「()」を使用すると、結果で合計されるすべての項目について総合計のローが生成されます。総合計の行では、すべてのグループ化の式に対するすべての値にプレースホルダの NULL が含まれます。GROUPING 関数を使用すると、ローの基本となるデータで値を評価するため、プレースホルダの NULL と実際の NULL を区別できます。「GROUPING 関数を使用したプレースホルダの NULL の検出」 492 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 ～ 3 がロー 4 ～ 6 と同一であることに注意してください。

	City	Cnt
1	'Drayton'	3
2	'Petersburg'	1
3	'Pembroke'	4
4	'Drayton'	3
5	'Petersburg'	1
6	'Pembroke'	4

適切な形式の実践

GROUP BY GROUPING SETS 句でのグループ化構文の解釈は、単純な GROUP BY 句の場合とは異なります。たとえば、GROUP BY (X, Y) は X と Y の値の異なる組み合わせによってグループ化されます。しかし GROUP BY GROUPING SETS (X, Y) の場合は、2 つの個別のグループ化セットを指定し、その 2 つのグループ化の結果がユニオンされます。つまり、結果は (X) でグループ化されてから、(Y) でグループ化された同じ結果にユニオンされます。

適切な形式を使用し、複雑な式の場合のあいまいさを避けるために、エラーが発生する可能性がある場合は指定内の各グループ化セットをカッコで囲んでください。たとえば、次の両方の文は正しく、セマンティック上同一ですが、2 番目が推奨される形式を反映した文です。

```
SELECT * FROM t GROUP BY GROUPING SETS ( X, Y );
SELECT * FROM t GROUP BY GROUPING SETS( ( X ), ( Y ) );
```

GROUPING SETS のショートカットとしての ROLLUP と CUBE の使用

複数の異なるデータ分割を単一の結果セットに連結する場合は、GROUPING SETS を使用すると便利です。ただし、多くのグループ化を指定する必要があり、かつ小計を含める場合は、ROLLUP 拡張と CUBE 拡張を使用できます。

ROLLUP 句と CUBE 句は、事前に定義された GROUPING SETS 指定のショートカットと見なすことができます。

ROLLUP は、空のグループ化セット「()」から始まり、追加する式を前の式に連結させるグループ化セットが次々と続くような、一連のグループ化を指定するのと同等です。たとえば、3 つのグループ化式 a、b、c があり、ROLLUP を指定した場合は、セット ()、(a)、(a, b)、(a, b, c) を使用して GROUPING SETS 句を指定した場合と同じ結果になります。この構成は、階層グループ化と呼ばれることもあります。

CUBE を使用すると、さらに多くのグループ化を実現できます。CUBE を指定することは、すべての可能な GROUPING SETS を指定するのと同等です。たとえば、同様の 3 つのグループ化式 a、b、c があり、CUBE を指定した場合は、セット ()、(a)、(a, b)、(a, c)、(b)、(b, c)、(c)、(a, b, c) を使用して GROUPING SETS 句を指定した場合と同じ結果になります。

ROLLUP または CUBE を指定する場合は、GROUPING 関数を使用して、結果内にあるプレースホルダの NULL を識別してください。プレースホルダの NULL は、ROLLUP または CUBE による結果セット内で暗黙的である小計のローが原因で発生します。「GROUPING 関数を使用したプレースホルダの NULL の検出」 492 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。

```
{(), (X1), (X1,X2), (X1,X2,X3), ... , (X1,X2,X3, ... , Xn)}
```

例

次のクエリは、年ごとと四半期ごとに販売注文を要約し、次の表に示す結果セットを返します。

```
SELECT QUARTER( OrderDate ) AS Quarter,
    YEAR( OrderDate ) AS Year,
    COUNT( * ) AS Orders,
    GROUPING( Quarter ) AS GQ,
    GROUPING( Year ) AS GY
FROM SalesOrders
GROUP BY ROLLUP( Year, Quarter )
ORDER BY Year, Quarter;
```


このクエリは、次の結果を返します。

	Quarter	Year	Orders	GQ	GY
1	(NULL)	(NULL)	648	1	1
2	(NULL)	2000	380	1	0
3	1	2000	87	0	0
4	2	2000	77	0	0
5	3	2000	91	0	0
6	4	2000	125	0	0
7	(NULL)	2001	268	1	0
8	1	2001	139	0	0
9	2	2001	119	0	0
10	3	2001	10	0	0

結果セットの 1 番目のローは、2 年間のすべての四半期について、すべての注文の総合計 (648) を示します。

ロー 2 は 2000 年の注文の合計数 (380) を示し、ロー 3 ～ 6 はこの年の四半期ごとの注文の小計を示します。同様に、ロー 7 は 2001 年の注文の合計数 (268) を示し、ロー 8 ～ 10 はこの年の四半期ごとの小計を示します。

GROUPING 関数で返される値を使用して、総合計が含まれるローと小計のローを区別できます。ロー 2 と 7 では、四半期カラムは NULL で、GQ (Grouping by Quarter) カラムは値 1 です。これは、このローが年ごとのすべての四半期の注文の合計であることを示します。

同様に、ロー 1 の四半期 (Quarter) カラムと年 (Year) カラムは NULL で、GQ カラムと GY カラムは値 1 です。これは、このローがすべての年のすべての四半期における注文の合計であることを示します。

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。

```
{(), (X1), (X1,X2), (X1,X2,X3), ... , (X1,X2,X3, ...,Xn),
(X2), (X2,X3), (X2,X3,X4), ... , (X2,X3,X4, ... , Xn), ... , (Xn)}.
```

例

次のクエリは、年ごと、四半期ごと、および四半期単位に販売注文を要約し、次の表に示す結果セットを生成します。

```
SELECT QUARTER( OrderDate ) AS Quarter,
       YEAR( OrderDate ) AS Year,
       COUNT( * ) AS Orders,
       GROUPING( Quarter ) AS GQ,
       GROUPING( Year ) AS GY
FROM SalesOrders
GROUP BY CUBE ( Year, Quarter )
ORDER BY Year, Quarter;
```

このクエリは、次の結果を返します。

	Quarter	Year	Orders	GQ	GY
1	(NULL)	(NULL)	648	1	1
2	1	(NULL)	226	0	1
3	2	(NULL)	196	0	1
4	3	(NULL)	101	0	1
5	4	(NULL)	125	0	1
6	(NULL)	2000	380	1	0
7	1	2000	87	0	0
8	2	2000	77	0	0

	Quarter	Year	Orders	GQ	GY
9	3	2000	91	0	0
10	4	2000	125	0	0
11	(NULL)	2001	268	1	0
12	1	2001	139	0	0
13	2	2001	119	0	0
14	3	2000	10	0	0

結果セットの 1 番目のローは、2000 年と 2001 年を結合した、すべての四半期についての、すべての注文の総合計 (648) を示します。

ロー 2 ～ 5 は、すべての年の暦四半期ごとの販売注文を要約します。

ロー 6 と 11 の Orders は、それぞれ 2000 年と 2001 年の注文の合計を示します。

ロー 7 ～ 10 と 12 ～ 14 は、それぞれ 2000 年と 2001 年の四半期ごとの合計を示します。

GROUPING 関数で返される値を使用して、総合計が含まれるローと小計のローを区別できます。ロー 6 と 11 は、四半期 (Quarter) カラムは NULL で、GQ (Grouping by Quarter) カラムは値 1 です。これは、このローが年ごとにすべての四半期の注文 (Orders) の合計であることを示します。

注意
CUBE は指数関数的な個数のグループ化セットを生成するため、CUBE を使用して生成される結果セットは非常に大きくなることがあります。このため 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。

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になります。

GROUPING 機能を使用すると、プレースホルダの NULL を基本となるデータによる NULL と区別できます。グループ化セット指定から *group-by-expression* を 1 つ使用して GROUPING 関数を指定した場合、この関数は、プレースホルダの NULL の場合は 1 を返し、そのローの基本となるデータに存在する値 (通常は NULL) を反映している場合は 0 を返します。

たとえば、次のクエリは下の表に示される結果セットを返します。

```
SELECT Employees.EmployeeID AS Employee,
   YEAR( OrderDate ) AS Year,
   COUNT( SalesOrders.ID ) AS Orders,
   GROUPING( Employee ) AS GE,
   GROUPING( Year ) AS GY
FROM Employees LEFT OUTER JOIN SalesOrders
   ON Employees.EmployeeID = SalesOrders.SalesRepresentative
WHERE Employees.Sex IN ( 'F' )
   AND Employees.State IN ( 'TX' , 'NY' )
GROUP BY GROUPING SETS ( ( Year, Employee ), ( Year ), ( ) )
ORDER BY Year, Employee;
```

このクエリは、次の結果を返します。

	Employees	Year	Orders	GE	GY
1	(NULL)	(NULL)	54	1	1
2	(NULL)	(NULL)	0	1	0
3	102	(NULL)	0	0	0
4	390	(NULL)	0	0	0
5	1062	(NULL)	0	0	0
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。

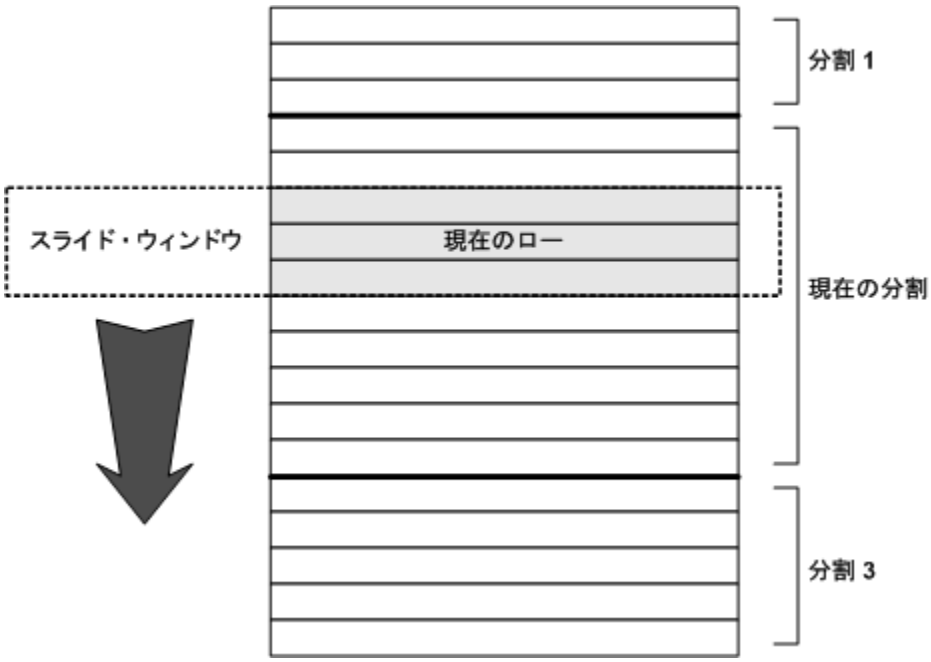
GROUPING 関数の構文の詳細については、「GROUPING 関数 [集合]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Window 関数

OLAP の機能には、入力ローの処理に合わせて入力ローを上から下に移動するスライド「ウィンドウ」の概念が含まれます。ウィンドウの移動とともにウィンドウ内のデータに対する追加の計算を実行できるため、セマンティック上同等なセルフジョイン・クエリや相関サブクエリを使用する方法よりも効率的な方法で、詳細な分析を実行できます。

ウィンドウの境界は、データから抽出しようとする情報を基に設定します。ウィンドウには、ウィンドウ定義内のグループ化指定に従って分割された入力データの、1 つ、複数、またはすべてのローが含まれます。ウィンドウは入力データを上から下に移動し、必須の計算を実行するために必要なローを採用します。

次の図は、入力ローが処理されるのに伴って移動するウィンドウを示しています。データ分割は、ウィンドウ定義に指定された、入力ローのグループ化を反映します。グループ化が指定されていない場合は、すべての入力ローで 1 分割であると見なされます。ウィンドウの長さ (つまりウィンドウに含まれるローの数) と、現在のローと比較したウィンドウのオフセットは、ウィンドウ定義で指定した境界を反映します。



ウィンドウの定義

SQL のウィンドウ拡張を使用して、ウィンドウの境界や、入力ローの分割や順序付けを設定できます。論理的には、GROUP BY 句で定義されたグループが作成された後、最終の SELECT リストの評価とクエリの ORDER BY 句の前に、クエリ仕様の結果を計算するセマンティックの一部として分割が作成されます。したがって、SQL 文における句の評価順は次のようになります。

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. WINDOW
6. DISTINCT
7. ORDER BY

クエリを形成する際には、評価順の影響を考慮する必要があります。たとえば、同じ SELECT クエリ・ブロックにある Window 関数を参照する式を述部にすることはできません。ただし、クエリ・ブロックを派生テーブルに入れることで、派生テーブルを使用して述部を指定できます。次のクエリを実行すると、Window 関数が述部に使用されているという旨のメッセージが表示されて、クエリは失敗します。

```
SELECT DepartmentID, Surname, StartDate, Salary,
     SUM( Salary ) OVER ( PARTITION BY DepartmentID
       ORDER BY StartDate
       RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW ) AS "Sum_Salary"
  FROM Employees
  WHERE State IN ( 'CA', 'UT', 'NY', 'AZ' )
    AND DepartmentID IN ( '100', '200' )
  GROUP BY DepartmentID, Surname, StartDate, Salary
  HAVING Salary > 0 AND "Sum_Salary" > 200
  ORDER BY DepartmentID, StartDate;
```

目的の結果を得るためには、派生テーブル (DT) を使用して述部を指定します。

```
SELECT * FROM ( SELECT DepartmentID, Surname, StartDate, Salary,
                  SUM( Salary ) OVER ( PARTITION BY DepartmentID
                    ORDER BY StartDate
                    RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW ) AS
"Sum_Salary"
                FROM Employees
                WHERE State IN ( 'CA', 'UT', 'NY', 'AZ' )
                  AND DepartmentID IN ( '100', '200' )
                GROUP BY DepartmentID, Surname, StartDate, Salary
                HAVING Salary > 0
                ORDER BY DepartmentID, StartDate ) AS DT
  WHERE DT.Sum_Salary > 200;
```

ウィンドウ分割は GROUP BY 演算子に続くので、分割を実行する計算で、任意の集合関数 (SUM、AVG、VARIANCE など) の結果を利用できます。そのため、クエリの GROUP BY 句や ORDER BY 句だけでなく、ウィンドウを使用することでもグループ化と順序付けの操作を実行できます。

ウィンドウ指定の定義

Window 関数で操作するウィンドウを定義するときは、次の項目を 1 つまたは複数指定します。

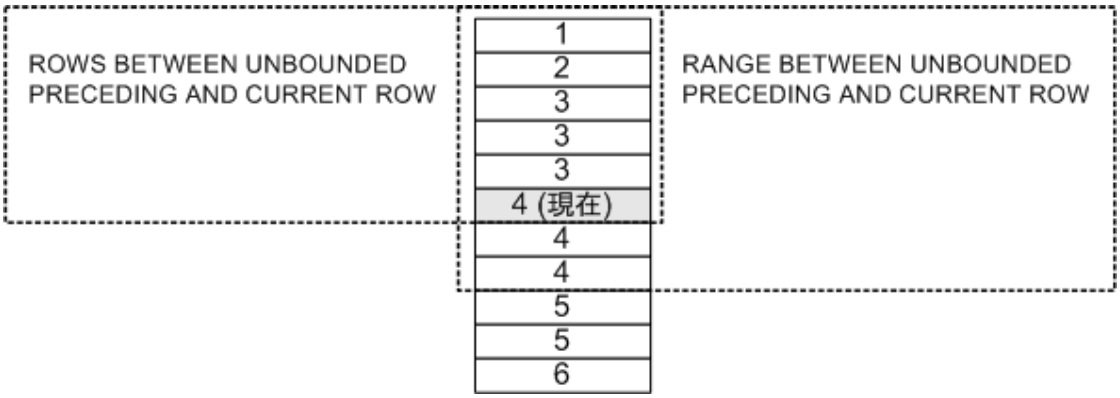
- **分割 (PARTITION BY 句)**　PARTITION BY 句により、入力ローのグループ化方法を定義します。省略すると、入力全体が単一の分割として扱われます。指定した内容に応じて、1 つ、

複数、またはすべての入力ローから分割を作成できます。2 つの分割からのデータが混合することはありません。つまり、ウィンドウが 2 つの分割の境界に達すると、分割内のデータの処理が終了してから、次の分割内のデータの処理が開始されます。ウィンドウの境界の定義方法に応じて、ウィンドウのサイズが分割の先頭と末尾で変化する可能性があります。

- **順序付け (ORDER BY 句)** ORDER BY 句により、Window 関数による処理の前に、入力ローの順序を決める方法を定義します。RANGE 句を使用して境界を指定する場合、またはランキング関数がウィンドウを参照する場合にかぎり、ORDER BY 句が必要です。それ以外の場合、ORDER BY 句はオプションです。省略すると、データベース・サーバが最も効率的な順序で入力ローを処理します。
- **境界 (RANGE 句および ROWS 句)** 現在のローは、ウィンドウの開始ローと終了ローを判断するための参照ポイントになります。ウィンドウ定義の RANGE 句と ROWS 句を使用して、境界を設定できます。RANGE は、現在のローの値からのオフセットとして**データ値の範囲**を指定することでウィンドウを定義します。範囲を計算するためにはデータが順序付けられている必要があるため、RANGE を指定する場合は ORDER BY 句も指定する必要があります。

  ROWS は、現在のローからのオフセットとして**ロー数**を指定することで、ウィンドウを定義します。

  RANGE は、データ値の範囲によってローのセットを定義するため、RANGE ウィンドウに含まれるローは現在のローを越える場合があります。この点は ROWS とは異なります。次の図は、ROWS 句と RANGE 句の違いを示したものです。



ROWS 句と RANGE 句内では、現在のローを基準にしてウィンドウの開始ローと終了ローを (オプションで) 指定できます。これには、PRECEDING 句、BETWEEN 句、および FOLLOWING 句を使用します。これらの句には、式の他に、UNBOUNDED と CURRENT ROW の各キーワードも指定できます。ウィンドウに境界が定義されていない場合、ウィンドウ境界はデフォルトで次のようになります。

- ○ ウィンドウ指定に ORDER BY 句が含まれている場合は、RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW を指定したことと同義になります。
- ○ ウィンドウ指定に ORDER BY 句が含まれていない場合は、ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING を指定したことと同義になります。

次の表は、ウィンドウ境界の例とウィンドウに含まれるローの説明を示したものです。

指定	意味
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW	分割の先頭から開始し、現在のローで終了します。累積の結果 (累積合計など) を計算する場合に使用してください。
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING	分割内のすべてのローを使用します。分割の各ローに対して集合関数の値を同一にする場合に使用してください。
ROWS BETWEEN *x* PRECEDING AND *y* FOLLOWING	現在のローから *x* の距離にある開始ローと *y* の距離にある終了ロー (境界値を含む) からなる固定サイズの移動するウィンドウを作成します。移動平均を計算する場合や隣接するロー間の値の差分を計算する場合には、この例を使用してください。 複数のローからなる移動するウィンドウでは、分割の最初のローや最後のローを計算するときに NULL になります。これは、現在のローが分割のまさに最初または最後のローである場合に、計算で使用できる直前または直後のローが存在しないためです。そのため、代わりに NULL 値が使用されます。
ROWS BETWEEN CURRENT ROW AND CURRENT ROW	1 つのロー (現在のロー) からなるウィンドウ。
RANGE BETWEEN 5 PRECEDING AND 5 FOLLOWING	ロー内の値に基づいてウィンドウを作成します。たとえば、現在のローに対して、ORDER BY 句に指定されたカラムに値 10 が含まれているとします。ウィンドウのサイズを RANGE BETWEEN 5 PRECEDING AND 5 FOLLOWING と指定した場合、最初のローにはそのカラムに 5、最後のローにはそのカラムに 15 がそれぞれ含まれるのに必要な大きさとして、ウィンドウのサイズを指定します。ウィンドウが分割を移動すると、範囲仕様を満たすのに必要なサイズに応じて、ウィンドウのサイズが変化します。

ウィンドウ指定はできるかぎり明示的にしてください。明示的に指定しないと、デフォルトが予期した結果を返さない場合があります。

連続でない値の場合は、RANGE 句を使用して、Window 関数の入力の差に起因する問題を回避します。RANGE 句を使用してウィンドウ境界が設定された場合、データベース・サーバは隣接するローや重複する値を含むローを自動的に処理します。

RANGE では、符号なし整数値を使用します。ORDER BY 式のドメインと RANGE 句で指定した値のドメインに応じて、範囲式のトランケーションが発生することがあります。

ランキング関数やロー番号付け関数を使用するときは、ウィンドウの境界を指定しないでください。

ウィンドウ定義：インラインおよび WINDOW 句

ウィンドウを定義する方法には次の 3 つがあります。

- インライン (Window 関数の OVER 句内)
- WINDOW 句内
- インラインと WINDOW 句内で部分的に指定

ただし、以降の項に示すとおり、方法によっては制限があります。

インライン定義

ウィンドウ定義は、Window 関数の OVER 句に配置できます。このことを、ウィンドウを**インライン**で定義するといいます。

たとえば、次の文は、2001 年 7 月と 8 月に出荷されたすべての製品と、出荷日ごとの累積出荷数量について、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。

```
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

	ID	Description	Quantity	ShipDate	Cumulative_qty
8	501	Plastic Visor	60	2001-07-07	60
9	501	Plastic Visor	12	2001-07-12	72
10	501	Plastic Visor	12	2001-07-22	84
11	601	Zipped Sweatshirt	60	2001-07-19	60
12	700	Cotton Shorts	24	2001-07-26	24

この例では、2 つのテーブルのジョインとクエリの WHERE 句の適用後に、SUM Window 関数の計算が発生します。クエリは次のように処理されます。

1. ProductID の値を基に、入力ローを分割 (グループ化) します。
2. 各分割内で、ShipDate の値を基にローをソートします。
3. 分割内の各ローについて、Quantity の値に対して SUM 関数を評価します。このとき、各分割の最初の (ソートされた) ローからなるスライド・ウィンドウを使用します。

WINDOW 句の定義

前述のクエリの別の構成として、WINDOW 句を使用して、ウィンドウを使用する関数とは別にウィンドウを指定します。次に、各関数の OVER 句からウィンドウを参照します。

この例で、WINDOW 句はウィンドウ Cumulative を作成し、データを ProductID ごとに分割し、ShipDate で並べ替えます。SUM 関数は、その OVER 句でウィンドウを参照し、ROWS 句を使用してウィンドウのサイズを定義します。

```
SELECT p.ID, p.Description, s.Quantity, s.ShipDate,
   SUM( s.Quantity ) OVER ( Cumulative
   ROWS BETWEEN UNBOUNDED PRECEDING
   AND CURRENT ROW ) AS cumulative_qty
FROM SalesOrderItems s
JOIN Products p ON ( s.ProductID = p.ID )
WHERE s.ShipDate BETWEEN '2001-07-01' AND '2001-08-31'
WINDOW Cumulative AS ( PARTITION BY s.ProductID ORDER BY s.ShipDate )
ORDER BY p.ID;
```

WINDOW 句構文を使用するときは、次の制限があります。

- PARTITION BY 句を指定する場合は、WINDOWS 句内に配置する必要があります。
- ROWS 句または RANGE 句を指定する場合は、参照元関数の OVER 句内に配置する必要があります。
- ウィンドウに ORDER BY 句を指定する場合は、WINDOW 句内か参照元関数の OVER 句内に配置できますが、両方には配置できません。
- WINDOW 句は、SELECT 文の ORDER BY 句に先行する必要があります。

インラインおよび WINDOW 句の定義の組み合わせ

ウィンドウ定義の一部をインラインで指定し、残りを WINDOW 句で定義できます。次に例を示します。

```
AVG() OVER ( windowA
        ORDER BY expression )...
...
WINDOW windowA AS ( PARTITION BY expression )
```

この方法でウィンドウ定義を分割すると、次の制限が適用されます。

- Window 関数構文では PARTITION BY 句を使用できません。
- Window 関数構文または WINDOW 句のいずれかで ORDER BY 句を使用することはできますが、両方では使用できません。
- RANGE 句または ROWS 句を WINDOW 句に含めることはできません。

参照

- 「WINDOW 句」 『SQL Anywhere サーバ - SQL リファレンス』
- 「Window 集合関数」 501 ページ
- 「Window ランキング関数」 519 ページ
- 「ウィンドウの定義」 494 ページ

SQL Anywhere の Window 関数

入力ローのセットに対して分析演算を実行できる関数は、Window 関数と呼ばれます。たとえば、すべてのランキング関数、およびほとんどの集合関数は「Window 関数」です。Window 関数を使用すると、データの追加分析を実行できます。この操作を行うには、入力ローを処理前に分割してソートし、次にサイズを設定可能な入力が進むウィンドウ内でローを処理します。

Window 関数には、Window 集合関数、Window ランキング関数、ロー番号付け関数の 3 種類があります。

Window 集合関数

Window 集合関数は、入力内のローの指定されたセットに対する値を返します。たとえば、Window 関数を使用して、指定した期間における会社の販売数の移動平均を計算できます。

Window 集合関数は次の 3 つのカテゴリに分類されます。

- **基本集合関数**　サポートされる基本集合関数のリストを次に示します。

  - SUM
  - AVG
  - MAX
  - MIN
  - FIRST_VALUE
  - LAST_VALUE
  - COUNT

  基本の集合関数の詳細については、「基本集合関数」 502 ページを参照してください。

- **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

  標準偏差関数と平方偏差関数の詳細については、「標準偏差関数と平方偏差関数」 513 ページを参照してください。

- **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

  相関関数と線形回帰関数の詳細については、「相関関数と線形回帰関数」 517 ページを参照してください。

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、数値式の平均値またはユニークな値からなるセットの平均値を返します。
- **MAX 関数**　各ロー・グループで見つかった式の最大値を返します。
- **MIN 関数**　各ロー・グループで見つかった式の最小値を返します。
- **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」 494 ページ

SUM 関数の例

次の例は、Window 関数として使用される SUM 関数を示したものです。クエリは、DepartmentID 別にデータを分割した結果セットを返し、在籍経験が長い人から順に従業員の給与の累積合計 (Sum_Salary) を算出します。結果セットには、カリフォルニア州、ユタ州、ニューヨーク州、またはアリゾナ州に住む従業員のみが含まれます。Sum_Salary のカラムは、従業員の給与の累積合計です。

```
SELECT DepartmentID, Surname, StartDate, Salary,
SUM( Salary ) OVER ( PARTITION BY DepartmentID
   ORDER BY StartDate
   RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
AS "Sum_Salary"
FROM Employees
WHERE State IN ( 'CA', 'UT', 'NY', 'AZ' )
  AND DepartmentID IN ( '100', '200' )
ORDER BY DepartmentID, StartDate;
```

次のテーブルは、クエリからの結果セットを示します。結果セットは、DepartmentID ごとに分割されています。

	DepartmentID	Surname	StartDate	Salary	Sum_Salary
1	100	Whitney	1984-08-28	45700.00	45700.00
2	100	Cobb	1985-01-01	62000.00	107700.00
3	100	Shishov	1986-06-07	72995.00	180695.00
4	100	Driscoll	1986-07-01	48023.69	228718.69
5	100	Guevara	1986-10-14	42998.00	271716.69
6	100	Wang	1988-09-29	68400.00	340116.69
7	100	Soo	1990-07-31	39075.00	379191.69
8	100	Diaz	1990-08-19	54900.00	434091.69
9	200	Overbey	1987-02-19	39300.00	39300.00
10	200	Martel	1989-10-16	55700.00	95000.00
11	200	Savarino	1989-11-07	72300.00	167300.00
12	200	Clark	1990-07-21	45000.00	212300.00
13	200	Goggin	1990-08-05	37900.00	250200.00

DepartmentID 100 の場合、カリフォルニア州、ユタ州、ニューヨーク州、またはアリゾナ州に住む従業員の給与の累積合計は 434,091.69 ドルで、部門 200 の従業員の累積合計は 250,200.00 ドルです。

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)
2	1062	Blaikie	54900.000	55700.000	-800.000
3	249	Guevara	42998.000	54900.000	-11902.000
4	390	Davidson	57090.000	42998.000	14092.000
5	102	Whitney	45700.000	57090.000	-11390.000
6	1507	Wetherby	35745.000	45700.000	-9955.000
7	1751	Ahmed	34992.000	35745.000	-753.000
8	1157	Soo	39075.000	34992.000	4083.000

CurrentRow ウィンドウでは、ウィンドウのサイズが ROWS BETWEEN CURRENT ROW AND CURRENT ROW に設定されているため、SUM は現在のローのみに対して実行されます。同様に、PreviousRow ウィンドウでは、ウィンドウのサイズが ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING に設定されているため、SUM は直前のローのみに対して実行されます。最初のローには先行するローがないため、このローの PreviousRow の値は NULL です。そのため Delta の値も NULL になります。

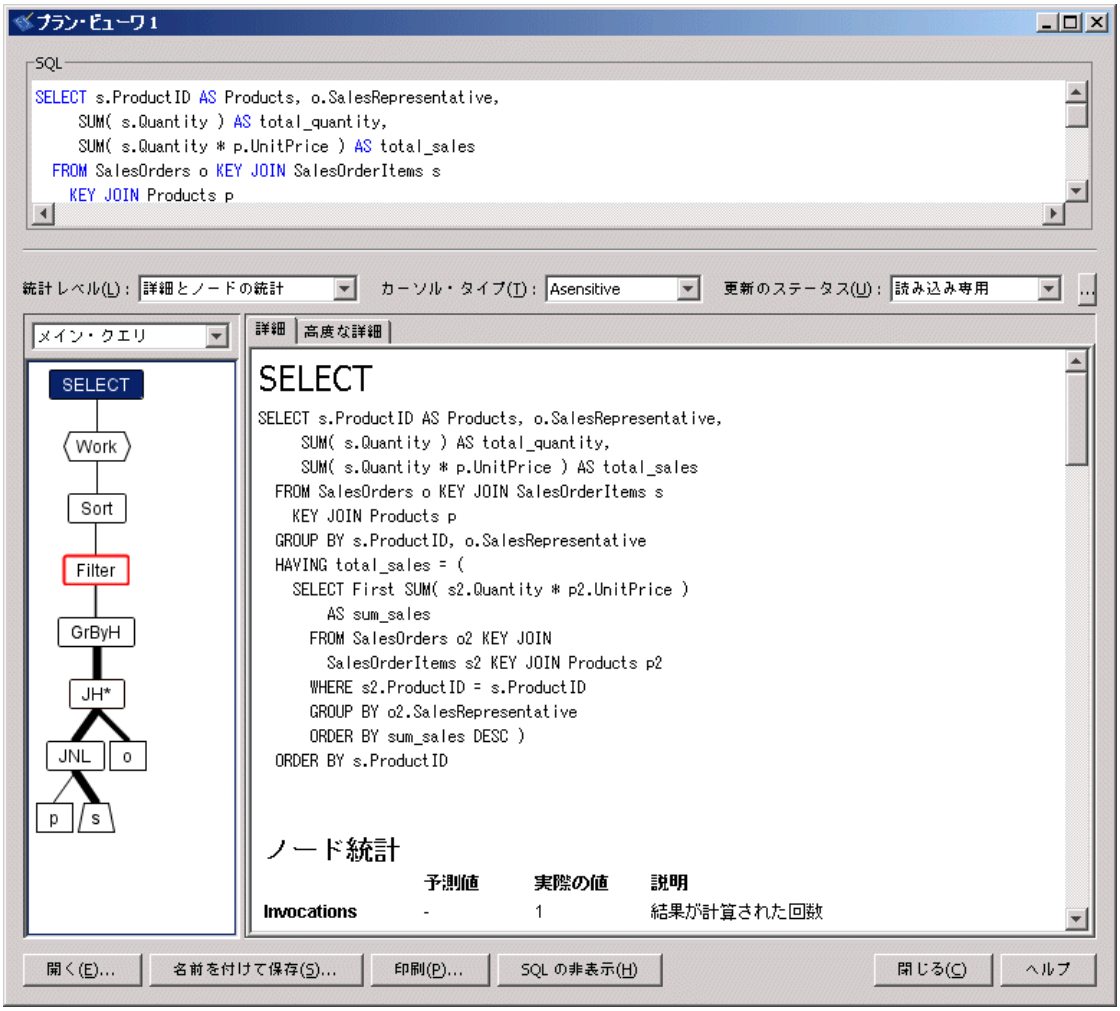
複雑な分析

次のクエリを考えてみます。このクエリは、データベース内で製品ごとに最上位の営業担当者 (総売り上げで定義) をリストします。

```
SELECT s.ProductID AS Products, o.SalesRepresentative,
    SUM( s.Quantity ) AS total_quantity,
    SUM( s.Quantity * p.UnitPrice ) AS total_sales
```



```
FROM SalesOrders o KEY JOIN SalesOrderItems s
  KEY JOIN Products p
GROUP BY s.ProductID, o.SalesRepresentative
HAVING total_sales = (
  SELECT First SUM( s2.Quantity * p2.UnitPrice )
     AS sum_sales
    FROM SalesOrders o2 KEY JOIN
      SalesOrderItems s2 KEY JOIN Products p2
    WHERE s2.ProductID = s.ProductID
    GROUP BY o2.SalesRepresentative
    ORDER BY sum_sales DESC )
ORDER BY s.ProductID;
```



このクエリは、次の結果を返します。

	Products	SalesRepresentative	total_quantity	total_sales
1	300	299	660	5940.00

	Products	SalesRepresentative	total_quantity	total_sales
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
7	501	690	360	2520.00
8	501	949	360	2520.00
9	600	299	612	14688.00
10	601	299	636	15264.00
11	700	299	1008	15120.00

元のクエリは、ProductID をサブクエリの相関外部参照として、ある製品の最高売り上げを判断する相関サブクエリを使用して作成されています。ただし、この場合のようにネストされたクエリを使用すると、コストの高いオプションになることがあります。これは、サブクエリに GROUP BY 句だけでなく、GROUP BY 句内の ORDER BY 句も含まれるからです。そのため、クエリ・オプティマイザは、同じセマンティックを保持したままになり、このネストされたクエリをジョインとして書き換えることができません。したがって、クエリの実行中は、外部ブロック内で計算される派生ローごとにサブクエリが評価されます。

グラフィカル・プランでコストの高い Filter 述部に注意してください。オプティマイザは、クエリの実行コストの 99% がこのプラン演算子に起因するものであると推定します。サブクエリのプランでは、メイン・ブロックのフィルタ演算子のコストが高くなる理由を明確にしています。サブクエリには、ネスト・ループ・ジョインが 2 つと、ハッシュ GROUP BY 演算が 1 つ、ソートが 1 つ含まれます。

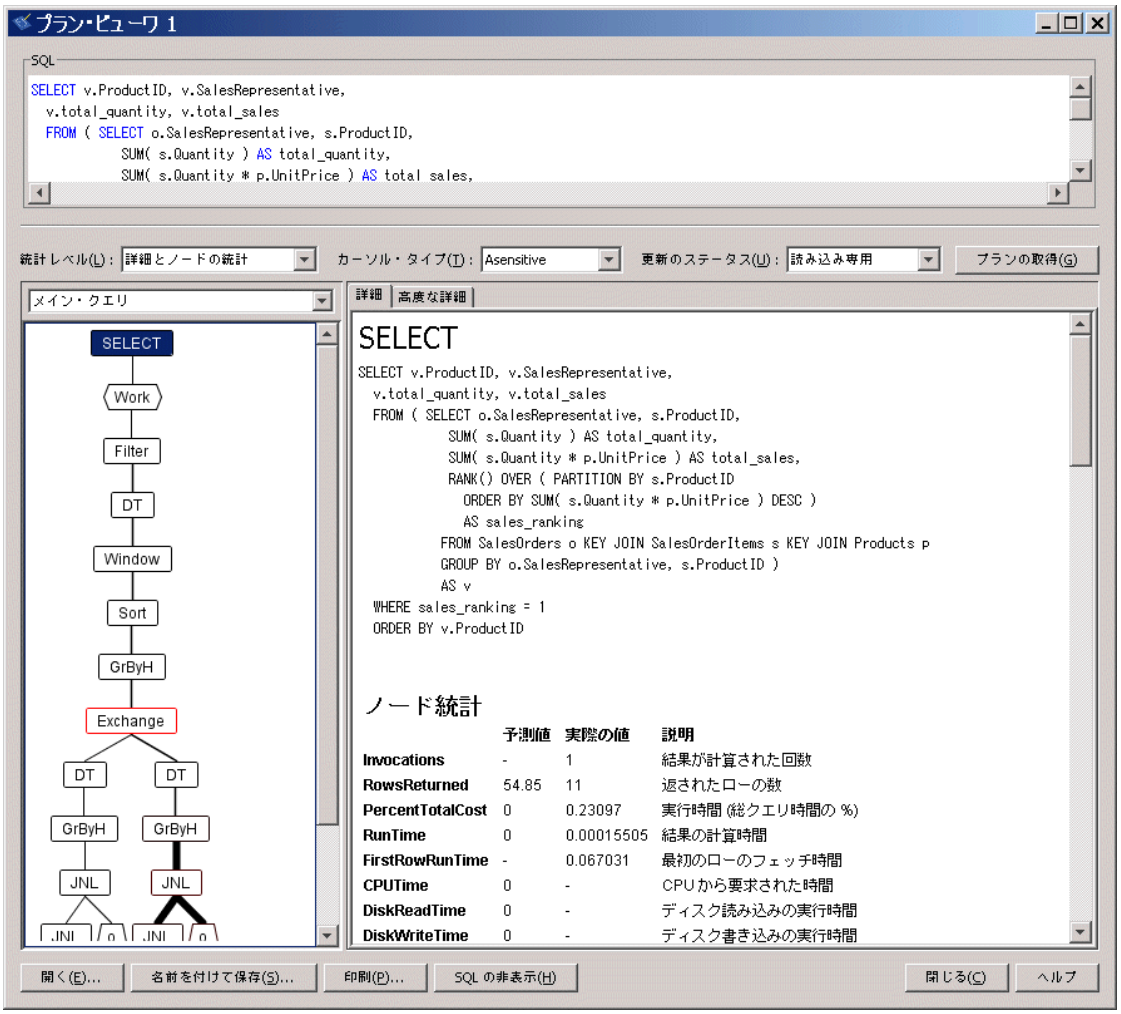
ランキング関数を使用した書き換え

ランキング関数を使用した同じクエリの書き換えでは、同じ結果が非常に効率よく計算されます。

```
SELECT v.ProductID, v.SalesRepresentative,
  v.total_quantity, v.total_sales
  FROM ( SELECT o.SalesRepresentative, s.ProductID,
           SUM( s.Quantity ) AS total_quantity,
           SUM( s.Quantity * p.UnitPrice ) AS total_sales,
           RANK() OVER ( PARTITION BY s.ProductID
             ORDER BY SUM( s.Quantity * p.UnitPrice ) DESC )
             AS sales_ranking
```

```
        FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
        GROUP BY o.SalesRepresentative, s.ProductID )
        AS v
WHERE sales_ranking = 1
ORDER BY v.ProductID;
```

この書き換えられたクエリは、プランがより単純になります。



GROUP BY 句が処理されてから select リスト項目の評価とクエリの ORDER BY 句の処理が行われるまでに、ウィンドウ演算子が計算されます。グラフィカルなプランでわかるように、3 つのテーブルのジョイン後、ジョインされたローは SalesRepresentative 属性と ProductID 属性の組み合わせでグループ化されます。したがって、total_quantity と total_sales の SUM 集合関数は、SalesRepresentative と ProductID の組み合わせごとに計算できます。

GROUP BY 句の評価に続いて、中間結果のローを total_sales の降順にランク付けするために、ウィンドウを使用して RANK 関数が計算されます。WINDOW 指定には PARTITION BY 句が含まれます。それによって、GROUP BY 句の結果が、今度は ProductID ごとに再分割 (再グループ

化) されます。そのため、RANK 関数は、総売り上げの降順で製品ごとにローをランク付けしますが、その製品を販売した営業担当者はすべてまとめられます。このランキングにより、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 ではない派生テーブルのローを拒否するように派生テーブルの結果を制限するだけです。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売上の移動平均を計算するための Window 関数として AVG が使用されています。WINDOW 指定で RANGE 句が使用されているので、ROWS 句の場合のように隣接ローの数で計算されるのではなく、月の値を基にウィンドウ境界が計算されます。ある月に一部またはすべての製品の販売がなかった場合などは、ROWS を使用すると異なる結果が生成されます。

```
SELECT *
 FROM ( SELECT s.ProductID,
       Month( o.OrderDate ) AS julian_month,
       SUM( s.Quantity ) AS sales,
       AVG( SUM( s.Quantity ) )
       OVER ( PARTITION BY s.ProductID
         ORDER BY Month( o.OrderDate ) ASC
         RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING )
       AS average_sales
       FROM SalesOrderItems s KEY JOIN SalesOrders o
       WHERE Year( o.OrderDate ) = 2000
       GROUP BY s.ProductID, Month( o.OrderDate ) )
 AS DT
 ORDER BY 1,2;
```

参照

- 「AVG 関数 [集合]」 『SQL Anywhere サーバ - SQL リファレンス』

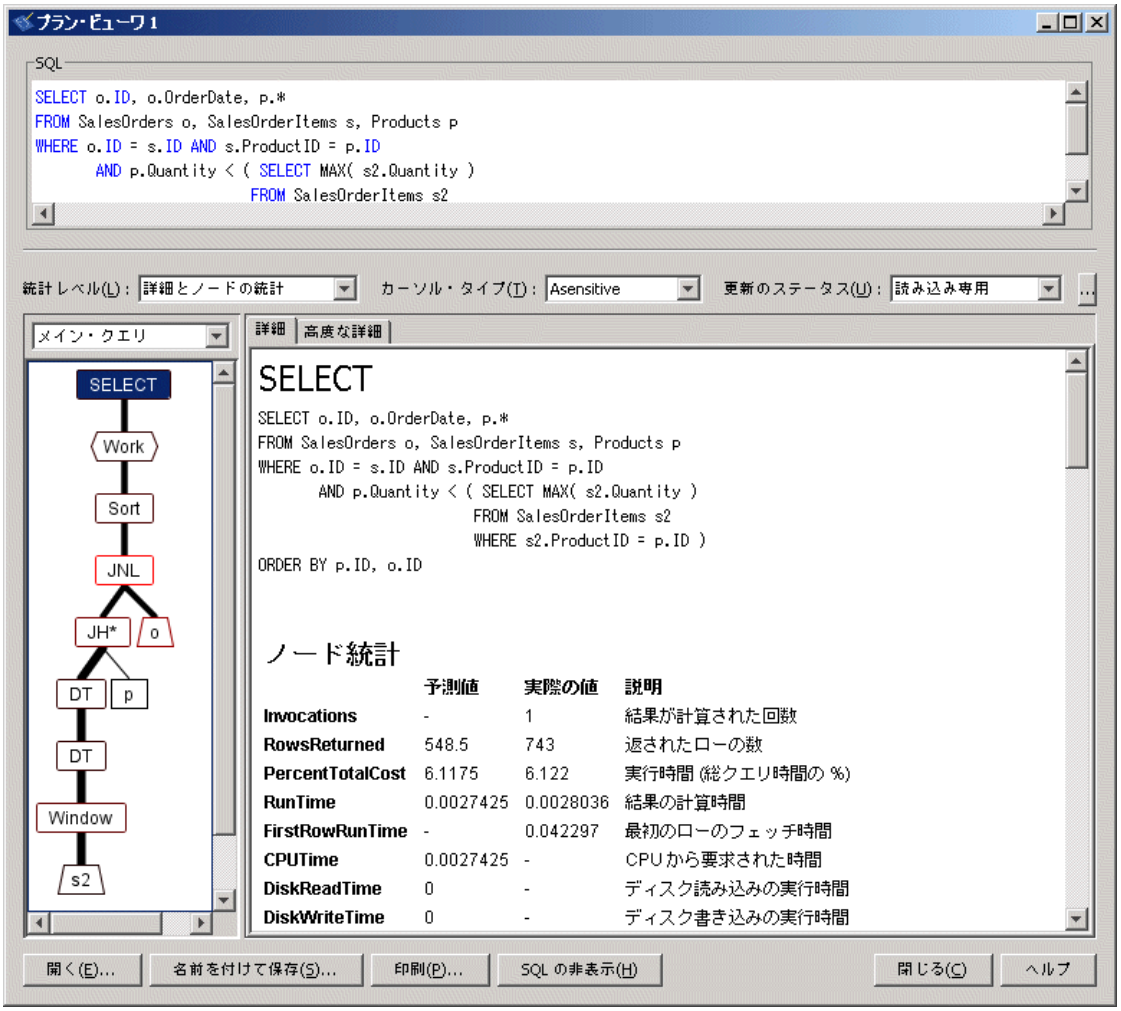
MAX 関数の例

相関サブクエリの削除

場合によっては、特定のカラムの値を最大値や最小値と比較する必要があります。このようなクエリは、相関属性 (外部参照とも呼ばれる) のあるネストされたクエリとして作成されることがよくあります。たとえば、次のクエリでは、その製品の注文 1 回あたりの最大数が製品の在庫数を超えているような製品について、すべての注文を製品情報とともにリストします。

```
SELECT o.ID, o.OrderDate, p.*
FROM SalesOrders o, SalesOrderItems s, Products p
WHERE o.ID = s.ID AND s.ProductID = p.ID
     AND p.Quantity < ( SELECT MAX( s2.Quantity )
                  FROM SalesOrderItems s2
                  WHERE s2.ProductID = p.ID )
ORDER BY p.ID, o.ID;
```

このクエリのグラフィカルなプランは、下の図のようにプラン・ビューワに表示されます。このネストされたクエリが、クエリ・オプティマイザによってどのように変形されたかを確認してください。クエリは、Products テーブルと、Window 関数を含む派生テーブル (相関名 DT) のある SalesOrders テーブルとのジョインに変形されています。



オプティマイザに依存して相関サブクエリを派生テーブルによるジョインに変形するのではなく (セマンティック分析は複雑なので、この方法は単純な場合にしか使用できない)、Window 関数を使用して、このようなクエリを作成できます。

```
SELECT order_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。指定しなければ、最初または最後の値が返されます。

例 1：グループの最初のエントリ

FIRST_VALUE 関数を使用して、一定の順序で並んでいる値グループの最初のエントリを取り出すことができます。次のクエリは、各注文について、注文の最初の品目の ProductID、つまり各注文で LineID が最も小さい品目の ProductID を返します。

クエリでは、DISTINCT キーワードを使用して重複を削除しています。このキーワードを指定しなかった場合、各注文の各品目について重複するローが返されます。

```
SELECT DISTINCT ID,
FIRST_VALUE( ProductID ) OVER ( PARTITION BY ID ORDER BY LineID )
FROM SalesOrderItems
ORDER BY ID;
```

例 2：最大売り上げに対する割合

FIRST_VALUE 関数の一般的な使用方法として、各ローの値を、現在のグループ内の最大値または最小値と比較できます。次のクエリは、各営業担当者の総売り上げを計算してから、その総売り上げと、同じ製品の最大総売り上げを比較します。結果は、最大総売り上げのパーセントで表されます。

```
SELECT s.ProductID AS prod_id, o.SalesRepresentative AS sales_rep,
   SUM( s.Quantity * p.UnitPrice ) AS total_sales,
   100 * total_sales / ( FIRST_VALUE( SUM( s.Quantity * p.UnitPrice ) )
                  OVER Sales_Window ) AS total_sales_percentage
FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
GROUP BY o.SalesRepresentative, s.ProductID
  WINDOW Sales_Window AS ( PARTITION BY s.ProductID
                 ORDER BY SUM( s.Quantity * p.UnitPrice ) DESC )
ORDER BY s.ProductID;
```

例 3：NULL 値を設定することによるデータの高密度化

FIRST_VALUE 関数と LAST_VALUE 関数は、データの密度を高めた後で、NULL の代わりに値を設定したい場合に便利です。たとえば、1 日の総売り上げが最も高い営業担当者が表彰されるとします。次のクエリは、2001 年 4 月の第 1 週にトップ成績を上げた担当者を表示します。

```
SELECT v.OrderDate, v.SalesRepresentative AS rep_of_the_day
FROM ( SELECT o.SalesRepresentative, o.OrderDate,
           RANK() OVER ( PARTITION BY o.OrderDate
                   ORDER BY SUM( s.Quantity *
                    p.UnitPrice ) DESC ) AS sales_ranking
       FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
       GROUP BY o.SalesRepresentative, o.OrderDate ) AS v
WHERE v.sales_ranking = 1
AND v.OrderDate BETWEEN '2001-04-01' AND '2001-04-07'
ORDER BY v.OrderDate;
```

このクエリは、次の結果を返します。

OrderDate	rep_of_the_day
2001-04-01	949
2001-04-02	856
2001-04-05	902
2001-04-06	467
2001-04-07	299

ただし、売り上げがなかった日については結果は返されません。次のクエリは、データの密度を高め、売り上げがなかった日のレコードも作成されるようにします。また、LAST_VALUE 関数を使用して、表彰がなかった日には、成績順位に入れ替わりがあるまで最後にトップ成績を獲得した者の ID を NULL 値の代わりに rep_of_the_day に表示させています。

```
SELECT d.dense_order_date,
              LAST_VALUE( v.SalesRepresentative IGNORE NULLS )
                 OVER ( ORDER BY d.dense_order_date )
                 AS rep_of_the_day
FROM ( SELECT o.SalesRepresentative, o.OrderDate,
           RANK() OVER ( PARTITION BY o.OrderDate
                   ORDER BY SUM( s.Quantity *
                    p.UnitPrice ) DESC ) AS sales_ranking
       FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
       GROUP BY o.SalesRepresentative, o.OrderDate ) AS v
RIGHT OUTER JOIN ( SELECT DATEADD( day, row_num, '2001-04-01' )
                    AS dense_order_date
                FROM sa_rowgenerator( 0, 6 )) AS d
ON v.OrderDate = d.dense_order_date AND sales_ranking = 1
ORDER BY d.dense_order_date;
```

このクエリは、次の結果を返します。

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

前のクエリからの派生テーブル v を、対象日をすべて含む派生テーブル d にジョインすると、1 日ごとにローができます。ただし、この外部ジョインでは、売り上げがなかった日の SalesRepresentative カラムには NULL が含まれます。LAST_VALUE 関数を使用すると、この問題を解決できます。特定のローの rep_of_the_day を、対応する日までの SalesRepresentative の最後の NULL 以外の値と定義します。

参照

- 「FIRST_VALUE 関数 [集合]」 『SQL Anywhere サーバ - SQL リファレンス』
- 「LAST_VALUE 関数 [集合]」 『SQL Anywhere サーバ - SQL リファレンス』
- 「Window 関数」 494 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、真の集合関数であるといえます。MAX や MIN などのその他の基本集合関数と同様に、入力の NULL 値は無視されます。

パフォーマンスを向上させるために、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。

- 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式を確認するには、「集合関数に対応する数式」 528 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	Department	Salary	Average	StandardDeviation
1	Lull	100	87900.00	58736.28	16829.60
2	Scheffield	100	87900.00	58736.28	16829.60

	Employee	Department	Salary	Average	StandardDeviation
3	Scott	100	96300.00	58736.28	16829.60
4	Sterling	200	64900.00	48390.95	13869.60
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。

```
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...

Year	Quarter	Average	Variance
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式の詳細については、「集合関数に対応する数式」 528 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。

  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。

- **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

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。SQL Anywhere でサポートされるランキング関数は次のとおりです。

- CUME_DIST
- DENSE_RANK
- PERCENT_RANK
- RANK

ランキング関数は、SUM 集合関数などと同様の方法で複数の入力ローからの結果を計算しないため、集合関数とは見なされません。これらの関数は、特定の式の値に基づいて、分割内のローのランク (相対的な順序) を計算します。分割内のローの各セットは個別にランク付けされます。そのため OVER 句に PARTITION BY 句が含まれない場合は、入力全体が単一の分割として扱われます。このため、ランキング関数で使用されるウィンドウに対して、ROWS 句または RANGE 句は指定できません。複数のランキング関数を含むクエリを作成し、それぞれの関数が入力ローを異なる状態に分割またはソートするようにできます。

すべてのランキング関数では、ランキング関数が依存する入力ローのソート順序を指定するために ORDER BY 句が必要です。ORDER BY 句に複数の式が含まれる場合は、最初の式によって隣接ローで同じ値になるときに同順の発生を避けるために、2 番目以降の式が使用されます。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
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」 522 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

	Surname	Salary	Sex	Rank
7	Hildebrand	45829.00	F	7
8	Goggin	37900.00	M	8
9	Rebeiro	34576.00	M	9
10	Bigelow	31200.00	F	10
11	Lynch	24903.00	M	11

例 3

データを分割して異なる結果になるように計算できます。例 2 のクエリを使用して、性別で分割することでデータを変更できます。次の例は、従業員を性別ごとに給与の多い順でランクしたものです。

```
SELECT Surname, Salary, Sex,
    RANK ( ) OVER ( PARTITION BY Sex
    ORDER BY Salary DESC ) "Rank"
    FROM Employees
WHERE State IN ( 'UT' );
```

次のテーブルは、クエリからの結果セットを示します。

	Surname	Salary	Sex	Rank
1	Wang	68400.00	M	1
2	Cobb	62000.00	M	2
3	Morris	61300.00	M	3
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)。

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」 519 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、集合関数の値を基にランキングを判断するような複雑な要求を指定できます。

次のクエリは、地域ごとに総売り上げ上位 3 人の営業担当者と、地域ごとの総売り上げを生成します。

```
SELECT *
  FROM ( SELECT o.SalesRepresentative, o.Region,
           SUM( s.Quantity * p.UnitPrice ) AS total_sales,
           DENSE_RANK( ) OVER ( PARTITION BY o.Region,
             GROUPING( o.SalesRepresentative )
             ORDER BY total_sales DESC ) AS sales_rank
         FROM Products p, SalesOrderItems s, SalesOrders o
         WHERE p.ID = s.ProductID AND s.ID = o.ID
         GROUP BY GROUPING SETS( ( o.SalesRepresentative, o.Region ),
           o.Region ) ) AS DT
  WHERE sales_rank <= 3
  ORDER BY Region, sales_rank;
```

このクエリは、次の結果を返します。

	SalesRepresentative	Region	total_sales	sales_rank
1	299	Canada	9312.00	1
2	(NULL)	Canada	24768.00	1
3	1596	Canada	3564.00	2
4	856	Canada	2724.00	3
5	299	Central	32592.00	1
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

	SalesRepresentative	Region	total_sales	sales_rank
18	(NULL)	Western	37632.00	1
19	1596	Western	5076.00	2
20	667	Western	4068.00	3

このクエリは、GROUPING SETS を使用して複数のグループ化を結合します。そのため、ウィンドウの WINDOW PARTITION 句では GROUPING 関数を使用して、特定の営業担当者を表すディテール・ローと、地域全体の総売り上げをリストする小計ローとを区別します。地域ごとの小計のローは、SalesRepresentative 属性に値 NULL がありますが、入力の分割ごとに結果のランキング順序が付けられるため、それぞれの小計ローのランキング値は 1 になります。これにより、ディテール・ローは 1 から適切にランク付けされます。

この例では、DENSE_RANK 関数により、総売り上げの集約について入力がランク付けされています。WINDOW ORDER 句では、エイリアスの設定された select リスト項目が省略形として使用されます。

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。

CUME_DIST 関数

累積分布関数 CUME_DIST は、百分位数の逆数として定義される場合があります。CUME_DIST は、ウィンドウ内の値のセットに関して、特定値の正規化された位置を計算します。関数の範囲は 0 ～ 1 です。

ウィンドウが入力ローを移動するのに合わせて、ウィンドウの ORDER BY 句で指定された式で累積分布が計算されます。ORDER BY 句に複数の式が含まれる場合は、最初の式によって隣接ローで同じ値になるときに同順の発生を避けるために、2 番目以降の式が使用されます。NULL 値はその他の値よりも前にソートされます (昇順の場合)。

次の例は、カリフォルニアに住む従業員の給与に関する累積分布を示す結果セットを返します。

```
SELECT DepartmentID, Surname, Salary,
   CUME_DIST( ) OVER ( PARTITION BY DepartmentID
     ORDER BY Salary DESC ) "Rank"
  FROM Employees
  WHERE State IN ( 'CA' );
```

このクエリは、次の結果を返します。

DepartmentID	Surname	Salary	Rank
200	Savarino	72300.00	0.333333333333333
200	Clark	45000.00	0.666666666666667
200	Overbey	39300.00	1

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)/(-1) として計算されます。

ウィンドウが入力ローを移動するのに合わせて、ウィンドウの ORDER BY 句で指定された式でランクが計算されます。ORDER BY 句に複数の式が含まれる場合は、最初の式によって隣接ローで同じ値になるときに同順の発生を避けるために、2 番目以降の式が使用されます。NULL 値はその他の値よりも前にソートされます (昇順の場合)。

例 1

次の例は、ニューヨークの従業員の給与ランキングを性別ごとに示す結果セットを返します。結果セットは、小数のパーセンテージを使用して降順にランキングされ、性別ごとに分けられます。

```
SELECT DepartmentID, Surname, Salary, Sex,
   PERCENT_RANK( ) OVER ( PARTITION BY Sex
     ORDER BY Salary DESC ) AS PctRank
  FROM Employees
  WHERE State IN ( 'NY' );
```

このクエリは、次の結果を返します。

	DepartmentID	Surname	Salary	Sex	PctRank
1	200	Martel	55700.000	M	0.0
2	100	Guevara	42998.000	M	0.333333333
3	100	Soo	39075.000	M	0.666666667
4	400	Ahmed	34992.000	M	1.0
5	300	Davidson	57090.000	F	0.0
6	400	Blaikie	54900.000	F	0.333333333
7	100	Whitney	45700.000	F	0.666666667
8	400	Wetherby	35745.000	F	1.0

入力は性別 (Sex) で分割されるため、PERCENT_RANK は男性と女性で個別に評価されます。

例 2

次の例は、ユタ州とアリゾナ州の女性従業員のリストを返し、給与の多い順にランクしたものです。PERCENT_RANK 関数は、降順で累積合計を計算するのに使用します。

```
SELECT Surname, Salary,
    PERCENT_RANK ( ) OVER ( ORDER BY Salary DESC ) "Rank"
    FROM Employees
WHERE State IN ( 'UT', 'AZ' ) AND Sex IN ( 'F' );
```

このクエリは、次の結果を返します。

	Surname	Salary	Rank
1	Shishov	72995.00	0
2	Jordan	51432.00	0.25
3	Hildebrand	45829.00	0.5
4	Bigelow	31200.00	0.75
5	Bertrand	29800.00	1

PERCENT_RANK を使用した最上位と最下位の百分位数の検索

PERCENT_RANK 関数を使用して、データ・セット内の最上位または最下位の百分位数を検索できます。次の例では、クエリは給与額についてデータ・セット内で上位 5 % の男性従業員を返します。

```
SELECT *
FROM ( SELECT Surname, Salary,
      PERCENT_RANK ( ) OVER ( ORDER BY Salary DESC ) "Rank"
      FROM Employees
      WHERE Sex IN ( 'M' ) )
      AS DerivedTable ( Surname, Salary, Percent )
WHERE Percent < 0.05;
```

このクエリは、次の結果を返します。

	Surname	Salary	Percent
1	Scott	96300.00	0
2	Sheffield	87900.00	0.025
3	Lull	87900.00	0.025

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

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 では、NUMBER と ROW_NUMBER の 2 つのロー番号付け関数がサポートされています。ROW_NUMBER 関数は、ANSI 標準準拠の関数で、SQL Anywhere の NUMBER(*) 関数と同等の機能の多くを使用できるため、この関数の使用をおすすめします。どちらの関数も同様のタスクを実行しますが、NUMBER 関数には、ROW_NUMBER 関数にはない制限がいくつかあります。

参照

- 「NUMBER 関数 [その他]」『SQL Anywhere サーバ - SQL リファレンス』
- 「ROW_NUMBER 関数」 527 ページ

ROW_NUMBER 関数

ROW_NUMBER 関数は、結果のローにユニークな番号を付けます。この関数はランキング関数ではありませんが、ランキング関数を使用できるどのような状況でも使うことができ、動作はランキング関数に似ています。

たとえば、派生テーブルで ROW_NUMBER を使用して、ROW_NUMBER の値について、ジョインであっても制限を追加できます。

```
SELECT *
FROM ( SELECT Description, Quantity,
      ROW_NUMBER( ) OVER ( ORDER BY ID ASC ) AS RowNum
FROM Products ) AS DT
WHERE RowNum <= 3
ORDER BY RowNum;
```

このクエリは、次の結果を返します。

Description	Quantity	RowNum
Tank Top	28	1
V-neck	54	2
Crew Neck	75	3

ランキング関数の場合と同様に、ROW_NUMBER には ORDER BY 句が必要です。

ウィンドウの ORDER BY 句がユニークでない式で構成される場合は、ROW_NUMBER は非決定的な結果を返すことがあり、同順が発生したときのローの順序は予測できなくなります。

ROW_NUMBER は、分割全体に対して機能するように設計されているため、ROWS 句や RANGE 句を ROW_NUMBER 関数とともに指定することはできません。

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。

集合関数に対応する数式

参考情報として、SQL Anywhere でサポートされるすべての Window 集合関数について、それに等価な数式を次の 2 つの表に示します。

単純な集合関数

Function	Symbol	Formula
SUM(X)		$\sum_{i=1}^{n} x_i$
MAX(X)		$x_i : x_i \geq x_j, i \neq j \; \forall i,j \in n$
MIN(X)		$x_i : x_i \leq x_j, i \neq j \; \forall i,j \in n$
AVG(X)	$\bar{x}$	$\frac{\sum x_i}{n}$
COUNT(*)		n
VAR_SAMP(X)	s_x^2	$\frac{\sum (x_i - \bar{x})^2}{(n-1)}$
VAR_POP(X)	σ_x^2	$\frac{\sum (x_i - \bar{x})^2}{n}$
VARIANCE(X)		identical to VAR_SAMP(X)
STDDEV_SAMP(X)	s_x	$\sqrt{\frac{\sum (x_i - \bar{x})^2}{(n-1)}}$
STDDEV_POP(X)	σ_x	$\sqrt{\frac{\sum (x_i - \bar{x})^2}{n}}$
STDDEV(X)		identical to STDDEV_SAMP(X)

統計集合関数

COVAR_SAMP(Y,X)	*Co-variance*	$s_{xy} = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{(n-1)}$
COVAR_POP(Y,X)	*Co-variance*	$\sigma_{xy} = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{n}$
CORR(Y,X)	*Correlation Coefficient*	$r = \frac{\sum xy - \frac{1}{n}(\sum x)(\sum y)}{(n-1)s_x s_y}$
REGR_AVGX(Y,X)	*Independent mean*	$\bar{x}$
REGR_AVGY(Y,X)	*Dependent mean*	$\bar{y}$
REGR_SLOPE(Y,X)	*Regression Slope*	$b = r\frac{s_y}{s_x}$
REGR_INTERCEPT(Y,X)	*Regression Intercept*	$a = \bar{y} - b\bar{x}$
REGR_R2(Y,X)	*'Goodness-of-fit'*	r^2
REGR_COUNT(Y,X)	*Sample size*	n (non-null (Y, X) pairs)
REGR_SXX(Y,X)	*Sum of squares* (x)	$\sum x^2 - \frac{(\sum x)^2}{n}$
REGR_SYY(Y,X)	*Sum of squares* (y)	$\sum y^2 - \frac{(\sum y)^2}{n}$
REGR_SXY(Y,X)	*Sum of products*	$\sum xy - \frac{(\sum y)(\sum x)}{n}$

サブクエリの使用

目次

リレーショナル・データベースを使用すると、複数のテーブルに関連するデータを保存できます。ジョインを使用して関連するテーブルからデータを抽出できるほか、「サブクエリ」を使用しても抽出できます。サブクエリとは、親の SQL 文の SELECT 句、WHERE 句、HAVING 句内にネストされた SELECT 文です。

サブクエリを使用すると、一部のクエリをジョインより簡単に記述できます。また、サブクエリを使用しないと記述できないクエリがあります。

サブクエリは、次のようなさまざまな方法に分類されます。

- 1 つ以上のローを返すかどうか (単一ローと複数ローのサブクエリ)
- 相関サブクエリか、非相関サブクエリか
- 別のサブクエリ内でネストしているかどうか

単一ローのサブクエリと複数ローのサブクエリ

外部の文に 1 つまたは 0 個のローを返すサブクエリを、「単一ローのサブクエリ」と呼びます。単一ローのサブクエリは、WHERE 句または HAVING 句の中の比較演算子で使用されるサブクエリです。

外部の文に複数のロー (ただしカラムは 1 つだけ) を返すサブクエリを、「複数ローのサブクエリ」と呼びます。複数ローのサブクエリは、IN 句、ANY 句、または ALL 句で使用されるサブクエリです。

例 1 : 単一ローのサブクエリ

テーブル Products に製品だけの情報を、別のテーブル SalesOrdersItems には注文関連の情報を保存します。Products テーブルにはさまざまな製品の情報が入っています。SalesOrdersItems テーブルには、顧客の注文についての情報が入っています。在庫数が 50 未満になったときに製品を再注文する場合、次のクエリで「在庫数が少ない製品は何か」という問い合わせに対する回答を得ることができます。

```
SELECT ID, Name, Description, Quantity
FROM Products
WHERE Quantity < 50;
```

ただし、ほとんど注文されない製品がわずかしかないことよりも頻繁に購入される製品が少ないことの方が関心が高いため、製品が注文される頻度を考慮すると便利です。

サブクエリを使用してある顧客が注文する品目の平均数を決定し、その平均をメイン・クエリに使用して在庫数が少ない製品を検索します。次に示すクエリは、顧客が注文した各タイプの平均品目数の 2 倍未満の製品名とその説明を検索します。

```
SELECT Name, Description
FROM Products WHERE Quantity <  2 * (
  SELECT AVG( Quantity )
  FROM SalesOrderItems
  );
```

WHERE 句では、クエリ結果に含まれる、FROM 句にリストされるテーブルからローを選択するのにサブクエリが役立ちます。HAVING 句では、クエリ結果に含まれる、メイン・クエリの GROUP BY 句で指定されるローのグループを選択するのに役立ちます。

例 2 : 単一ローのサブクエリ

次の単一ローのサブクエリの例では、Products テーブルの製品の平均価格を計算します。そしてその平均は外部クエリの WHERE 句に渡されます。外部クエリは、平均より低い価格のすべての製品の ID、Name、UnitPrice を返します。

```
SELECT ID, Name, UnitPrice
FROM Products
WHERE UnitPrice <
  ( SELECT AVG( UnitPrice ) FROM Products )
ORDER BY UnitPrice DESC;
```


ID	Name	UnitPrice
401	Baseball Cap	10.00
300	Tee Shirt	9.00
400	Baseball Cap	9.00
500	Visor	7.00
501	Visor	7.00

例 3 : IN を使用した複数ローを返す単一のサブクエリ

在庫数が少ない品目を識別し、一方でそれらの品目に対する注文も識別したいとします。次のように、WHERE 句にサブクエリを含む SELECT 文を実行します。

```
SELECT *
FROM SalesOrderItems
WHERE ProductID IN
  (  SELECT ID
     FROM Products
     WHERE Quantity < 20 )
ORDER BY ShipDate DESC;
```

この例では、サブクエリは Products テーブル内の ID カラムにおいて WHERE 句の探索条件を満たすすべての値のリストを作成します。そして一連のローが返されますが、返されるカラムは 1 つだけです。IN キーワードは、それぞれの値をセットのメンバとして扱い、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。

例 4 : IN、ANY、ALL の使用を比較する複数ローのサブクエリ

SQL Anywhere サンプル・データベースには、経理に関するデータを格納するテーブルが 2 つあります。FinancialCodes テーブルは、経理データとこれらの意味についてのさまざまなコードが入っているテーブルです。FinancialData テーブルから歳入項目をリストするには、次のクエリを実行します。

```
SELECT *
FROM FinancialData
WHERE Code IN
   ( SELECT Code
      FROM FinancialCodes
      WHERE type = 'revenue' );
```

Year	Quarter	Code	Amount
1999	Q1	r1	1023
1999	Q2	r1	2033
1999	Q3	r1	2998
1999	Q4	r1	3014

Year	Quarter	Code	Amount
2000	Q1	r1	3114
...	...	...	...

ANY キーワードと ALL キーワードも同様の方法で使用できます。たとえば、次のクエリは前述のクエリと同じ結果を返しますが、ANY キーワードを使用しています。

```
SELECT *
FROM FinancialData
WHERE FinancialData.Code = ANY
  (  SELECT FinancialCodes.Code
     FROM FinancialCodes
     WHERE type = 'revenue' );
```

=ANY 条件は IN 条件とまったく同じですが、ANY を **<** や **>** などの不等号とともに使用するとサブクエリを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' );
```


相関サブクエリと非相関サブクエリ

サブクエリには、親の文に定義されたオブジェクトへの参照を含めることができます。これは「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。相関サブクエリは、外部クエリとは別に評価することはできません。これは、サブクエリが親の文の値を使用するためです。つまり、サブクエリは親の文のローごとに実行されます。したがって、サブクエリの結果は、親の文で評価されるアクティブなローに依存します。

たとえば、次の文のサブクエリは、Products テーブル内のアクティブなローに依存する値を返します。

```
SELECT Name, Description
FROM Products
WHERE Quantity < 2 * (
  SELECT AVG( Quantity )
  FROM SalesOrderItems
  WHERE Products.ID=SalesOrderItems.ProductID );
```

この例では、このサブクエリの Products.ID カラムは外部参照です。このクエリは、在庫数が平均注文数の 2 倍より少ない製品、具体的には、メイン・クエリの WHERE 句によってテストされている製品の、名前と説明を抽出します。サブクエリは SalesOrderItems テーブルをスキャンしてこれを実行します。しかし、サブクエリの WHERE 句にある Products.ID カラムは、サブクエリではなく、**メイン**・クエリの FROM 句に指定されているテーブルのカラムを参照します。データベース・サーバは Products テーブルの各ローの間を移動して、サブクエリの WHERE 句を評価するときに、現在のローの ID 値を使用します。

サブクエリで参照されるカラムが、サブクエリの FROM 句で参照されるテーブルになくても、外部クエリの FROM 句で参照されるテーブルにあれば、クエリはエラーなく実行されます。SQL Anywhere では、サブクエリのカラムが外部クエリのテーブル名で暗黙的に修飾されます。

親の文にオブジェクトへの参照を含まないサブクエリを、「非相関サブクエリ」と呼びます。次の例では、サブクエリは正確に 1 つの値、SalesOrderItems テーブルの平均数を計算します。クエリを評価するときに、データベース・サーバはこの値を一度計算し、その値を Products テーブルの Quantity フィールドにあるそれぞれの値と比較して、対応するローを選択するかどうかを決定します。

```
SELECT Name, Description
FROM Products
WHERE Quantity <  2 * (
  SELECT AVG( Quantity )
  FROM SalesOrderItems );
```

ネストされたサブクエリ

「ネストされたサブクエリ」とは、別のサブクエリの中にネストされたサブクエリです。定義できるサブクエリのネストのレベルに制限はありませんが、3 つ以上のレベルのクエリは、それ以下のレベルのクエリに比べて、実行にかなりの時間がかかります。

次の例では、ネストされたサブクエリを使用して、Fees 部の任意の品目が注文された日に出荷された注文の注文 ID とライン ID を決定します。

```
SELECT ID, LineID
FROM SalesOrderItems
WHERE ShipDate = ANY (
  SELECT OrderDate
  FROM SalesOrders
  WHERE FinancialCode IN (
    SELECT Code
    FROM FinancialCodes
    WHERE ( Description = 'Fees' ) ) );
```

ID	LineID
2001	1
2001	2
2001	3
2002	1
...	...

この例では、最も内側のサブクエリが、"Fees" という説明がある経理コードのカラムを生成します。

```
SELECT Code
FROM FinancialCodes
WHERE ( Description = 'Fees' );
```

次のサブクエリは、最も内側のサブクエリが選択したコードに一致するコードを持つ品目の注文日を検索します。

```
SELECT OrderDate
FROM SalesOrders
WHERE FinancialCode
IN ( subquery-expression );
```

最後に、一番外側のクエリが、サブクエリの検索した日付のいずれかに出荷された注文の注文 ID とライン ID を検索します。

```
SELECT ID, LineID
FROM SalesOrderItems
WHERE ShipDate = ANY ( subquery-expression );
```


ジョインに代わるサブクエリの使用

たとえば、受注と受注先の日付順リストが必要な場合に、Customers ID ではなく会社名を知りたいとします。次のようなジョインを使用して、この結果を得ることができます。

ジョインの使用

2001 年 1 月 1 日以降の受注 ID、日付、各注文を行った会社名をリストするには、次のクエリを実行します。

```
SELECT SalesOrders.ID,
       SalesOrders.OrderDate,
       Customers.CompanyName
FROM SalesOrders
  KEY JOIN Customers
WHERE OrderDate > '2001/01/01'
ORDER BY OrderDate;
```

サブクエリの使用方法

次の SQL 文は、ジョインではなくサブクエリを使用して同じ結果を得ます。

```
SELECT SalesOrders.ID,
  SalesOrders.OrderDate,
  (  SELECT CompanyName FROM Customers
     WHERE Customers.ID = SalesOrders.CustomerID )
FROM SalesOrders
WHERE OrderDate > '2001/01/01'
ORDER BY OrderDate;
```

SalesOrders テーブルがサブクエリの一部でない場合でも、サブクエリは SalesOrders テーブル内の CustomerID カラムを参照します。一方、SalesOrders.CustomerID カラムは SQL 文の本文にある SalesOrders テーブルを参照します。

他のテーブルから要求されるカラムが 1 つだけである場合は、ジョインの代わりにサブクエリを使用できます。サブクエリが返すことができるカラムは 1 つだけです。この例では CompanyName カラムだけを必要としていたので、ジョインをサブクエリに変更することができました。

外部ジョインの使用

ワシントン州在住の顧客名すべてとその顧客の最も最近の受注 ID をリストするには、次のクエリを実行します。

```
SELECT  CompanyName, State,
  ( SELECT MAX( ID )
     FROM SalesOrders
    WHERE SalesOrders.CustomerID = Customers.ID )
FROM Customers
WHERE State = 'WA';
```

CompanyName	State	MAX(SalesOrders.ID)
Custom Designs	WA	2547

CompanyName	State	MAX(SalesOrders.ID)
It's a Hit!	WA	(NULL)

It's a Hit! という会社は何も注文しなかったので、サブクエリはこの顧客については NULL を返します。内部ジョインを使用した場合、発注しなかった会社はリストされません。

外部ジョインを明示的に指定することもできます。その場合は、次のように GROUP BY 句も指定する必要があります。

```
SELECT CompanyName, State,
   MAX( SalesOrders.ID )
FROM Customers
   KEY LEFT OUTER JOIN SalesOrders
WHERE State = 'WA'
GROUP BY CompanyName, State;
```


WHERE 句でのサブクエリ

WHERE 句内のサブクエリは、ロー選択のプロセスの一部として機能します。ローの選択に使用する基準が別のテーブルの結果に依存するときに、WHERE 句内にサブクエリを使用します。

例

在庫数が平均注文数の 2 倍よりも少ない製品を検索します。

```
SELECT Name, Description
FROM Products WHERE Quantity <  2 * (
  SELECT AVG( Quantity )
  FROM SalesOrderItems );
```

このクエリは 2 段階で実行されます。まず、注文ごとに要求される品目の平均数を検索します。次に、どの製品の在庫数がその数の 2 倍より少ないかを検索します。

2 段階のクエリ

要求される品目の**数**は、品目のタイプ、顧客、注文ごとに、SalesOrderItems テーブルの Quantity カラムに格納されます。サブクエリは次のようになります。

```
SELECT AVG( Quantity )
FROM SalesOrderItems;
```

これによって SalesOrderItems テーブルの品目の平均数、25.851413 が返されます。

次のクエリは、前述のクエリで抽出した値の 2 倍よりも少ない在庫数の品目の名前とその説明を返します。

```
SELECT Name, Description
FROM Products
WHERE Quantity < 2*25.851413;
```

サブクエリを使用すると、この 2 つの手順を 1 つのオペレーションにまとめることができます。

WHERE 句でのサブクエリの目的

WHERE 句内でのサブクエリは、探索条件の一部です。WHERE 句内で使用できる簡単な探索条件については、「データのクエリ」 305 ページの章で説明します。

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	62.000000
Shorts	80.000000
Tee Shirt	52.333333

クエリは次のように実行します。

- サブクエリは SalesOrderItems テーブルにある品目の平均数を計算します。
- メイン・クエリは Products テーブルを調べて、製品ごとの平均数を計算し、製品名でグループ化します。
- HAVING 句は、各平均数がサブクエリで検索された数量の 2 倍を超えるかどうかを確認します。超える場合、メイン・クエリはそのロー・グループを返します。超えない場合は返しません。
- SELECT 句は、グループごとに 1 つの計算ローを生成し、各製品の名前と在庫の平均数を示します。

次の例で示すように、HAVING 句には外部参照も使用できます。この例は、前述の例を若干変更したものです。

例

この例では、平均注文数が在庫数の半分よりも多い製品の ID 番号とライン ID 番号を検索します。

```
SELECT ProductID, LineID
FROM SalesOrderItems
GROUP BY ProductID, LineID
HAVING 2* AVG( Quantity ) > (
   SELECT Quantity
```



```
FROM Products
WHERE Products.ID = SalesOrderItems.ProductID );
```

ProductID	LineID
601	3
601	2
601	1
600	2
...	...

この例では、サブクエリは、HAVING 句にテストされるロー・グループに対応する製品の在庫数を生成します。サブクエリは外部参照 SalesOrderItems.ProductID を使用して、その特定製品のレコードを選択します。

比較演算子を持つサブクエリは 1 つの値を返す

このクエリは比較演算子 > を使用するので、サブクエリは 1 つの値を返します。この場合は、1 つの値を返します。Products テーブルの ID フィールドがプライマリ・キーなので、特定の製品 ID に対応する Products テーブルのレコードは 1 つだけになります。

サブクエリのテスト

HAVING 句内で使用できる簡単な探索条件については、「データのクエリ」 305 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。比較テストでは、サブクエリで提供される演算子 (=、<>、<、<=、>、>=) を使用します。
- **限定比較テスト**　サブクエリが生成した値のそれぞれのセットと式の値を比較します。
- **サブクエリ・セット・メンバシップ・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。

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。

サブクエリと IN 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' ) );
```

サブクエリと ANY テスト

ANY テストは、SQL 比較演算子 (=、>、<、>=、<=、!=、<>、!>、!<) のいずれかと組み合せて使用して、1 つの値をサブクエリが生成するデータ値のカラムと比較します。テストを実行するには、SQL は指定された比較演算子を使用して、テスト値をカラムのデータ値のそれぞれと比較します。**いずれかの**比較の結果が TRUE になる場合、ANY テストは TRUE を返します。

ANY を使用するサブクエリは 1 つのカラムを返します。

例

注文番号 2005 の最初の製品が出荷された後に受けた注文の注文 ID と顧客 ID を検索します。

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > ANY (
  SELECT ShipDate
  FROM SalesOrderItems
  WHERE ID=2005 );
```


ID	CustomerID
2006	105
2007	106
2008	107
2009	108
...	...

このクエリを実行すると、メイン・クエリが、注文番号 2005 の**すべての**製品の出荷日に対して、各注文の注文日をテストします。注文日が注文番号 2005 の **1 つ**の出荷の出荷日より後であれば、SalesOrders テーブルの注文 ID と顧客 ID が結果セットに示されます。このように ANY テストは OR 演算子に似ています。前述のクエリは、「この注文は注文番号 2005 の最初の製品が出荷された後に受けたものか、または注文番号 2005 の 2 番目の製品が出荷された後に受けたものか、または ...」というように解釈できます。

ANY 演算子の知識

ANY 演算子はやや複雑な場合があります。このクエリは、「注文番号 2005 の任意の製品が出荷された後に受けた注文を返す」と解釈してしまいがちです。しかし、それでは注文番号 2005 の**すべての**製品が出荷された後に受けた注文の注文 ID と顧客 ID を返すことになり、クエリの動作と異なります。

そうではなく、「注文番号 2005 の**少なくとも 1 つの**製品が出荷された後に受けた注文の注文 ID と顧客 ID を返す」というようにクエリを解釈してみます。キーワード SOME を使用すると、もう少し直感的な方法でクエリを表現できます。次のクエリは前述のクエリと同等です。

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > SOME (
  SELECT ShipDate
  FROM SalesOrderItems
  WHERE ID=2005 );
```

キーワード SOME はキーワード ANY と同等です。

ANY 演算子についての注意

ANY テストには、このほかに 2 つの重要な特徴があります。

- **空のサブクエリの結果セット**　サブクエリが空の結果セットを生成する場合、ANY テストは FALSE を返します。結果がない場合、少なくとも 1 つの結果が比較テストを満たしているというのは真ではないので、これは理にかなっています。
- **サブクエリの結果セットの NULL 値**　サブクエリの結果セットには少なくとも 1 つの NULL 値があることが前提です。結果セットの NULL 以外のすべてのデータ値に対して比較テストが FALSE の場合、ANY は UNKNOWN を返します。これは、比較テストが保持するサブクエリの値があるかどうか、この状況では確定できないためです。値があるかどうかは、結果セットの NULL データの**正確な**値によって異なります。ANY 探索条件の詳細については、

「ANY 探索条件と SOME 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。

サブクエリと ALL テスト

ALL テストは、SQL 比較演算子 (=、>、<、>=、<=、!=、<>、!>、!<) のいずれかと組み合せて使用して、1 つの値をサブクエリが生成するデータ値と比較します。テストを実行するには、SQL は指定された比較演算子を使用して、テスト値を結果セットのデータ値のそれぞれと比較します。すべての比較の結果が TRUE になる場合、ALL テストは TRUE を返します。

例

次の例は、注文番号 2001 のすべての製品が出荷された後に受けた注文の注文 ID と顧客 ID を検索します。

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > ALL (
  SELECT ShipDate
  FROM SalesOrderItems
  WHERE ID=2001 );
```

ID	CustomerID
2002	102
2003	103
2004	104
2005	101
...	...

このクエリを実行すると、メイン・クエリは、注文番号 2001 の**すべての**製品の出荷日に対して、各注文の注文日をテストします。注文日が注文番号 2001 の**すべての**出荷の出荷日より後であれば、SalesOrders テーブルの注文 ID と顧客 ID が結果セットに示されます。このように ALL テストは AND 演算子に似ています。前述のクエリは、「この注文は注文番号 2001 の最初の製品が出荷される前に受けたもので、なおかつ注文番号 2001 の 2 番目の製品が出荷される前に受けたもので、なおかつ ...」というように解釈できます。

ALL 演算子についての注意

ALL テストには、このほかに 3 つの重要な特徴があります。

- **空のサブクエリの結果セット**　サブクエリが空の結果セットを生成した場合、ALL テストは TRUE を返します。結果がない場合、比較テストが結果セットのどの値に対しても適用しているというのは真なので、これは理にかなっています。
- **サブクエリの結果セットの NULL 値**　結果セットのいずれかの値に対する比較テストが FALSE の場合、ALL は FALSE を返します。すべての値が TRUE の場合は TRUE を返しま

す。それ以外の場合は、UNKNOWN を返します。たとえば、サブクエリの結果セットに NULL 値があっても、NULL 以外のすべての値の探索条件が TRUE の場合などです。

- **ALL テストの否定**　次の 2 つの式は同じでは**ありません**。

  ```
  NOT a = ALL (subquery)
  a <> ALL (subquery)
  ```

  このテストの詳細については、「ANY、ALL、または SOME に続くサブクエリ」 551 ページを参照してください。

サブクエリと EXISTS テスト

サブクエリ比較テストとセット・メンバシップ・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、「相関サブクエリと非相関サブクエリ」 535 ページを参照してください。

```
SELECT *
FROM Customers, SalesOrders
WHERE ( OrderDate > '2001-07-13' ) AND
    ( Customers.ID = SalesOrders.CustomerID )

SELECT OrderDate
FROM Customers, SalesOrders
WHERE ( OrderDate > '2001-07-13' ) AND
    ( Customers.ID = SalesOrders.CustomerID );
```

便宜上、"SELECT *" という表記を使用していますが、SalesOrders テーブルのどのカラムが SELECT 文に指定されるかどうかは問題ではありません。

存在テストの否定

EXISTS テストの論理は、NOT EXISTS という形式で否定できます。この場合、テストはサブクエリがローを返さない場合に TRUE を、ローを返す場合に FALSE を返します。

相関サブクエリ

サブクエリには Customers テーブルからの ID カラムへの参照が含まれています。メイン・テーブル内のカラムや式への参照は、「外部参照」と呼ばれます。また、そのサブクエリは「相関」であるといいます。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。

オプティマイザによるサブクエリからジョインへの自動変換

クエリ・オプティマイザは、サブクエリを利用するクエリの多くをジョインとして自動的に書き換えます。変換はユーザによるアクションを必要とすることなく実行されます。この項では、データベースでのクエリのパフォーマンスを理解できるように、どのサブクエリがジョインに変換できるのかを説明します。

マルチレベルのクエリをジョインで作成するために満たす必要がある基準は、演算子のタイプ、クエリの構造、サブクエリの構造によって異なります。サブクエリが WHERE 句内にある場合は、次の形式になることに注意してください。

SELECT *select-list*
FROM *table*
WHERE
[**NOT**] *expression comparison-operator* **(** *subquery-expression* **)**
| [**NOT**] *expression comparison-operator* { **ANY** | **SOME** } **(** *subquery-expression* **)**
| [**NOT**] *expression comparison-operator* **ALL (** *subquery-expression* **)**
| [**NOT**] *expression* **IN (** *subquery-expression* **)**
| [**NOT**] **EXISTS (** *subquery-expression* **)**
GROUP BY *group-by-expression*
HAVING *search-condition*

たとえば、「Mrs. Clarke と Suresh がいつ注文し、どの担当者が注文を受けたか」という要求を考えてみます。これは次のクエリを使用して回答できます。

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID IN (
  SELECT ID
  FROM Customers
  WHERE Surname = 'Clarke' OR GivenName = 'Suresh' );
```

OrderDate	SalesRepresentative
2001-01-05	1596
2000-01-27	667
2000-11-11	467
2001-02-04	195
...	...

サブクエリは WHERE 句に名前がリストされている 2 人の顧客に対応する顧客 ID のリストを生成します。メイン・クエリはこの 2 人の注文に対応する注文日と担当者を検索します。

同じ問い合わせをジョインを使用して応答できます。このクエリの、2 つのテーブルのジョインを使用した代替形式を次に示します。

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders, Customers
```

```
WHERE CustomerID=Customers.ID AND
  ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

この形式のクエリは SalesOrders テーブルを Customers テーブルにジョインして各顧客の注文を検索し、Suresh と Clarke のレコードだけを返します。

サブクエリは有効でもジョインが有効でない場合

サブクエリは有効でも、ジョインが有効でない場合があります。次に例を示します。

```
SELECT Name, Description, Quantity
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

Name	Description	Quantity
Tee Shirt	Tank Top	28
Baseball Cap	Wool cap	12
Visor	Cloth Visor	36
...	...	...

この場合、内部クエリは集計クエリで外部クエリは集計クエリではないので、2 つのクエリを簡単なジョインで組み合わせることはできません。

参照

- 「ジョイン：複数テーブルからのデータ検索」 413 ページ

比較演算子に続くサブクエリ

比較演算子に続くサブクエリ (=、>、<、>=、<=、!=、<>、!>、!<) は、比較と呼ばれます。オプティマイザは、サブクエリが次のような場合に、これらのサブクエリをジョインに変換します。

- メイン・クエリのローごとに値を 1 つずつ返す
- GROUP BY 句を含んでいない
- キーワード DISTINCT を含んでいない
- UNION クエリではない
- 集計クエリではない

例

「Suresh の製品がいつ注文され、どの担当者が注文を受けたか」という要求をサブクエリで表現したとします。


```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID = (
  SELECT ID
  FROM Customers
  WHERE GivenName = 'Suresh' );
```

このクエリは基準を満たすので、ジョインを使用するクエリに変換できます。

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
 ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

しかし、「在庫数が平均注文数の 2 倍よりも少ない製品を検索する」という要求はジョインに変換できません。これは、サブクエリに集計関数 AVG が含まれているためです。

```
SELECT Name, Description
FROM Products
WHERE Quantity <  2 * (
  SELECT AVG( Quantity )
  FROM SalesOrderItems );
```

ANY、ALL、または SOME に続くサブクエリ

キーワード ALL、ANY、SOME のいずれかに続くサブクエリは、限定比較と呼ばれます。オプティマイザは、次のような場合にこれらのサブクエリをジョインに変換します。

- メイン・クエリが GROUP BY 句を含んでおらず、集計クエリでない。または、サブクエリが 1 つの値を返す。
- サブクエリが GROUP BY 句を含んでいない。
- サブクエリがキーワード DISTINCT を含んでいない。
- サブクエリが UNION クエリではない。
- サブクエリが集計クエリではない。
- '*expression comparison-operator* { **ANY** | **SOME** } **(** *subquery-expression* **)**' の部分が否定されていない。
- '*expression comparison-operator* **ALL (** *subquery-expression* **)**' の部分が否定されている。

最初の 4 つの条件は、比較的簡単です。

例

「Mrs. Clarke と Suresh がいつ注文し、どの担当者が注文を受けたか」という要求は、サブクエリの形式で処理できます。

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID = ANY (
  SELECT ID
  FROM Customers
  WHERE Surname = 'Clarke' OR GivenName = 'Suresh' );
```

または、ジョインの形式で表現できます。

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
  ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

しかし、「Mrs. Clarke、Suresh、および顧客でもある従業員が、注文したのはいつか」という要求は UNION クエリとして表現されるので、ジョインには変換できません。

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID = ANY (
  SELECT ID
  FROM Customers
  WHERE Surname = 'Clarke' OR GivenName = 'Suresh'
  UNION
  SELECT EmployeeID
  FROM Employees );
```

同様に、「すべての製品の最初の出荷日の後に出荷されていない注文の注文 ID と顧客 ID を検索する」という要求は、集計クエリで表現されるため、ジョインに変換できません。

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE NOT OrderDate > ALL (
  SELECT FIRST ( ShipDate )
  FROM SalesOrderItems
  ORDER BY ShipDate );
```

ANY と ALL 演算子を使用するサブクエリの否定

5 つ目の条件はやや複雑です。次の形式のクエリがジョインに変換されます。

SELECT *select-list*
FROM *table*
WHERE NOT *expression comparison-operator* **ALL (** *subquery-expression* **)**

SELECT *select-list*
FROM *table*
WHERE *expression comparison-operator* **ANY (** *subquery-expression* **)**

ただし、次のクエリはジョインに変換されません。

SELECT *select-list*
FROM *table*
WHERE *expression comparison-operator* **ALL (** *subquery-expression* **)**

SELECT *select-list*
FROM *table*
WHERE NOT *expression comparison-operator* **ANY (** *subquery-expression* **)**

最初の 2 つのクエリも、後の 2 つのクエリも、それぞれ同等です。すでに説明したように、ANY 演算子は OR 演算子と似ていますが、引数の数が異なります。同様に、ALL 演算子は AND 演算子に似ています。たとえば、次の 2 つの式は同等です。

```
NOT ( ( X > A ) AND ( X > B ) )
( X <= A ) OR ( X <= B )
```


次の 2 つの式も同等です。

```
WHERE NOT OrderDate > ALL (
  SELECT FIRST ( ShipDate )
  FROM SalesOrderItems
  ORDER BY ShipDate )

WHERE OrderDate <= ANY (
  SELECT FIRST ( ShipDate )
  FROM SalesOrderItems
  ORDER BY ShipDate )
```

ANY と ALL の否定

一般に、次の 2 つの式は同等です。

NOT *column-name operator* **ANY (** *subquery-expression* **)**

column-name inverse-operator **ALL (** *subquery-expression* **)**

次の式も、一般に同等です。

NOT *column-name operator* **ALL (** *subquery-expression* **)**

column-name inverse-operator **ANY (** *subquery-expression* **)**

inverse-operator は、次の表に示すように、*operator* を否定することによって取得されます。

operator	inverse-operator
=	<>
<	=>
>	=<
=<	>
=>	<
<>	=

IN に続くサブクエリ

オプティマイザは、IN キーワードが次のような場合に続くサブクエリのみを変換します。

- メイン・クエリが GROUP BY 句を含んでおらず、集計クエリでない。または、サブクエリが 1 つの値を返す。
- サブクエリが GROUP BY 句を含んでいない。
- サブクエリがキーワード DISTINCT を含んでいない。

- サブクエリが UNION クエリではない。
- サブクエリが集計クエリではない。
- 'expression IN (subquery-expression)' の部分が否定されていない。

例

「部長でもある従業員の名前を検索する」という要求は、次のクエリで表現されますが、この要求は条件を満たすため、ジョインされたクエリに変換されます。

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
  SELECT DepartmentHeadID
  FROM Departments
  WHERE ( DepartmentName ='Finance' OR
      DepartmentName = 'Shipping' ) );
```

しかし、「部長か顧客のいずれかである従業員の名前を検索する」という要求は、UNION クエリで表現されているとジョインに変換されません。

IN 演算子に続く UNION クエリは変換できない

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
  SELECT DepartmentHeadID
  FROM Departments
  WHERE ( DepartmentName='Finance' OR
      DepartmentName = 'Shipping' )
  UNION
  SELECT CustomerID
  FROM SalesOrders);
```

同様に、「部長ではない従業員の名前を検索する」という要求は、次に示す否定のサブクエリで表現されますが、変換されません。

```
SELECT GivenName, Surname
FROM Employees
 WHERE NOT EmployeeID IN (
  SELECT DepartmentHeadID
  FROM Departments
  WHERE ( DepartmentName='Finance' OR
      DepartmentName = 'Shipping' ) );
```

IN サブクエリまたは ANY サブクエリがジョインに変換されるために必要な条件は、同じです。これは、2 つの式が論理的には同等であるためです。

ANY 演算子を使用するクエリに変換される、IN 演算子を使用するクエリ

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、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' ) );
```

EXISTS に続くサブクエリ

オプティマイザは次のような場合のみ、EXISTS キーワードに続くサブクエリを変換します。

- メイン・クエリが GROUP BY 句を含んでおらず、集計クエリでない。または、サブクエリが 1 つの値を返す。
- 'EXISTS (subquery)' の部分が否定されていない。
- サブクエリが相関である。つまり、外部参照を含んでいる。

例

「どの顧客が 2001 年 7 月 13 日以降に発注したか」という要求は、外部参照 **Customers.ID** = **SalesOrders.CustomerID** を含む否定されていないサブクエリを持つクエリで表現できるので、次のジョインで表すことができます。

```
SELECT GivenName, Surname
FROM Customers
WHERE EXISTS (
  SELECT *
  FROM SalesOrders
  WHERE ( OrderDate > '2001-07-13' ) AND
      ( Customers.ID = SalesOrders.CustomerID ) );
```

EXISTS キーワードは、空の結果セットをチェックするようデータベース・サーバに通知するものです。内部ジョインが使用されていると、データベース・サーバは、FROM 句内のすべてのテーブルからのデータがあるローのみを自動的に表示します。つまり、次のクエリは、サブクエリを持つクエリが返すものと同じローを返します。

```
SELECT DISTINCT GivenName, Surname
FROM Customers, SalesOrders
WHERE ( SalesOrders.OrderDate > '2001-07-13' ) AND
    ( Customers.ID = SalesOrders.CustomerID );
```

データの追加、変更、削除

目次

データ修正文

データの追加、変更、削除に使う文は「データ修正文」といい、SQL の「データ修正言語」(DML) 部分とも呼ばれます。3 つの主要な DML 文を次に示します。

- **INSERT 文**　テーブルに新規のローを追加。
- **UPDATE 文**　テーブルにある既存のローを変更。
- **DELETE 文**　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。

前述の文以外に、LOAD TABLE 文と TRUNCATE TABLE 文は、特にデータのバルク・ロードや削除を行う場合に便利です。

参照

- 「INSERT 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「UPDATE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「DELETE 文」『SQL Anywhere サーバ - SQL リファレンス』

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、権限、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の使用」 117 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。

auto_commit オプションを On に設定すると、Interactive SQL に対して変更を自動的にコミットするように指定できます。これは Interactive SQL のオプションです。auto_commit を On に設定すると、INSERT 文、UPDATE 文、DELTE 文を実行するたびに Interactive SQL が COMMIT 文を発行します。このため、パフォーマンスが大幅に低下することがあります。このような場合には、auto_commit オプションを Off に設定することをおすすめします。

注意して COMMIT 文を使用すること
このチュートリアルにある例を試してみる場合、データベースの変更を確定しても問題がないと確信するまでは、どのような変更に対してもコミットしないように注意してください。「COMMIT 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

参照

- 「Interactive SQL オプション」『SQL Anywhere サーバ - データベース管理』

変更のキャンセル

コミットされていない変更はすべてキャンセルできます。SQL では、ROLLBACK 文を使用することによって最後のコミット以降に加えた変更をすべて取り消せます。この文は、最後に変更を確定した後にデータベースに対して行われたすべての変更を取り消します。「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

SQL Anywhere は、システム障害や停電が発生した場合にデータベースの整合性を保護します。データベース・サーバをリストアするためのオプションが複数用意されています。たとえば、SQL Anywhere によって別のドライブに保存されたログ・ファイルを使用して、データをリストアできます。リカバリのためにログ・ファイルを使用する場合、SQL Anywhere はデータベースを頻繁に更新する必要がないため、データベース・サーバのパフォーマンスは向上します。

トランザクション処理により、データベース・サーバはデータが一貫性を保っていることを識別できます。トランザクション処理は、なんらかの理由でトランザクションが正常に完了しなかった場合に、トランザクション全体が取り消されるか、ロールバックしたことを確認します。トランザクションが失敗しても、データベースには影響ありません。

SQL Anywhere のトランザクション処理は、トランザクションの途中でシステムがダウンした場合でも、トランザクションの内容は確実に処理されることを保証します。

参照

- 「バックアップとデータ・リカバリ」『SQL Anywhere サーバ - データベース管理』

参照整合性

データを挿入、更新、削除したときに、SQL Anywhere により、データ内に一般的なエラーがあるかどうか自動的に検査が行われます。このタイプの妥当性検査は、データベースに含まれるテーブル内やテーブル間のデータの整合性を検査するため、「参照整合性の確保」と呼ばれます。「エンティティ整合性と参照整合性の確保」 107 ページを参照してください。

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 つの形式があります。VALUES キーワードまたは SELECT 文を使用できます。

値を使う INSERT

VALUES キーワードで新しいロー内の一部、またはすべてのカラムの値を指定します。VALUES キーワードを使った INSERT 文の簡略バージョンの構文は、次のとおりです。

INSERT [**INTO**] *table-name* [(*column-name*, ...)]
VALUES (*expression*, ...)

SELECT * によるクエリを実行した結果に表示される順で、テーブルの各カラムに値を入力すると、カラム名のリストを省略できます。

SELECT からの INSERT

INSERT 文に SELECT 文を使用して、1 つ以上のテーブルから値を引き出せます。データを挿入するテーブルに多数のカラムがある場合は、WITH AUTO NAME を使用して構文を簡単にすることもできます。WITH AUTO NAME を使用する場合、カラム名を指定する必要があるのは、INSERT 文と SELECT 文の両方ではなく、SELECT 文のみです。SELECT 文の名前には、カラム参照かエイリアスの式を指定してください。

SELECT 文を使用した INSERT 文の簡略バージョンの構文は、次のとおりです。

INSERT [**INTO**] *table-name*
[**WITH AUTO NAME**] *select-statement*

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。

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' );
```

DepartmentHeadID にはデフォルトの値はありませんが、NULL. は受け入れます。そのため、NULL は自動的にそのカラムに割り当てられます。

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;
```

指定したカラムの順序はテーブル内のカラムの順序と一致させる必要はありませんが、挿入する値に指定した順序とは一致させてください。

指定されたカラムと指定されていないカラムに挿入される値

値は INSERT 文で指定された内容に従ってローに挿入されます。カラムに値が入力されていない場合、挿入される値はカラム設定 (NULL 値やデフォルト値を挿入するなど) によって異なります。挿入操作が失敗して、エラーが返される場合もあります。次の表は、挿入される値 (該当する場合) とカラム設定に基づいた結果を示しています。

挿入される値	NULL 入力可	NULL 入力不可	NULL 入力可 (DEFAULT 指定)	NULL 入力不可 (DEFAULT 指定)	NULL 入力不可 (DEFAULT AUTOINCREMENT 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。

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。「テーブル制約とカラム制約の使い方」 99 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。「カラム・デフォルトの使い方」 92 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

2 つのテーブルでカラムの順序が同じである場合、どちらのテーブルのカラム名も指定する必要はありません。たとえば、NewProducts というテーブルのスキーマが Products テーブルのスキーマと同じであり、NewProducts テーブルには Products テーブルに追加する製品情報の一部のローが含まれているとします。この場合、次の文を実行できます。

```
INSERT Products
SELECT *
FROM NewProducts;
```

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。「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' ) );
   ```


参照

- 「xp_read_file システム・プロシージャ」 『SQL Anywhere サーバ - SQL リファレンス』
- 「xp_read_file での openxml の使用」 727 ページ
- 「BLOB の格納」 5 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。「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。

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。

DELETE によるデータの削除

DELETE 文は、次のような単純な形式になっています。

DELETE [**FROM**] *table-name*
WHERE *column-name = expression*

次のような、より複雑な形式も使用できます。

DELETE [**FROM**] *table-name*
FROM *table-list*
WHERE *search-condition*

WHERE 句

WHERE 句を使用して削除するローを指定します。WHERE 句がないと、DELETE 文によってテーブルのすべてのローが削除されます。

FROM 句

DELETE 文の 2 番目に位置する FROM 句には、テーブルからデータを選択し、最初に指定されたテーブルから一致するデータを削除する、特別な働きがあります。FROM 句で選択したローに、削除の条件を指定します。「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;
```

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;
```


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。「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。

ベース・テーブルのトランケート操作またはバルク・ロードの実行操作を行うことにより、インデックス (通常のインデックスまたはテキスト・インデックス) 内と従属マテリアライズド・ビュー内のデータが古くなります。最初にインデックスや従属したマテリアライズド・ビューのデータをトランケートし、INPUT 文を実行してから、インデックスとマテリアライズド・ビューを再構築または再表示してください。「TRUNCATE 文」『SQL Anywhere サーバ - SQL リファレンス』と「TRUNCATE TEXT INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

TRUNCATE TABLE 構文

TRUNCATE TABLE 文の構文は次のとおりです。

TRUNCATE TABLE *table-name*

たとえば、SalesOrders テーブルのすべてのデータを削除するには、次のように入力します。

```
TRUNCATE TABLE SalesOrders;
```

TRUNCATE TABLE 文はテーブルで定義されたトリガを呼び出しません。

次の ROLLBACK 文を入力して、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;
```


クエリ処理

この項では、クエリ処理のフェーズ、オプティマイザが使用する方式、オプティマイザのパフォーマンスを最大限に向上させるためのヒントなど、クエリ・オプティマイザとその動作のしくみについて説明します。また、クエリの実行プランの表示方法と分析方法についても説明します。

クエリの最適化と実行

目次

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。

クエリ処理のフェーズ

この項では、文の注釈フェーズから実行完了までの各フェーズについて説明します。また、オプティマイザの設計の基本となる仮定条件について説明し、次に選択性推定、コスト推定、クエリ処理のステップについて説明します。

結果セットのない文 (UPDATE 文や DELETE 文など) も、クエリ処理のフェーズを経由します。

- **注釈フェーズ**　データベース・サーバは、クエリを受け取ると、パーサを使用して文を解析し、クエリの代数表現 (解析ツリー) に変換します。このフェーズでは、「解析ツリー」はセマンティックと構文のチェック (クエリで参照されるオブジェクトがカタログ内に存在することの検証など)、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、非マテリアライズド・ビュー展開などに使用されます。このフェーズの出力は、解析ツリーの形式で書き換えられたクエリで、元のクエリで参照されるすべてのオブジェクトに対する注釈が含まれます。

- **セマンティック変形フェーズ**　このフェーズでは、クエリに対して反復的なセマンティック変形を実行します。クエリが注釈付きの解析ツリーとして表されることに変わりはありませんが、リライト最適化 (ジョインの削除、DISTINCT の削除、述部の正規化など) がこのフェーズで適用されます。このフェーズのセマンティック変形は、解析ツリー表現にヒューリスティックに適用されるセマンティック変形規則に従って実行されます。「セマンティック・クエリ変形」 577 ページを参照してください。

  データベース・サーバによってプランがキャッシュ済みのクエリは、このクエリ処理のフェーズをスキップします。また、単純な文もこのフェーズをスキップする場合があります。たとえば、オプティマイザのバイパスでヒューリスティック・プラン選択を使用する文の多くは、セマンティック変形フェーズで処理されません。このフェーズが文に適用されるかどうかは SQL 文の複雑さによって決まります。「プランのキャッシュ」 601 ページと「クエリ処理のフェーズをスキップするための条件」 575 ページを参照してください。

- **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。「オプティマイザの仕組み」 590 ページを参照してください。

  データベース・サーバによってプランがキャッシュ済みのクエリは、このクエリ処理のフェーズをスキップします。また、単純な文もこのフェーズをスキップする場合があります。「プランのキャッシュ」 601 ページと「クエリ処理のフェーズをスキップするための条件」 575 ページを参照してください。

  このフェーズは、次の 2 つのサブフェーズに分かれています。

  - **最適化前フェーズ**　最適化前フェーズは、後から列挙フェーズで必要になる情報を最適化構造体に設定します。このフェーズでは、クエリを分析し、クエリ・アクセス・プランで使用できる関連するインデックスとマテリアライズド・ビューをすべて検出します。たとえばこのフェーズでは、ビュー・マッチング・アルゴリズムにより、クエリのすべてまたは一部を満たすために使用可能なすべてのマテリアライズド・ビューが特定されます。またオプティマイザは、クエリの述部分析を基にして、クエリのテーブルをジョインするために列挙フェーズで使用可能な代替のジョイン方式を構築します。このフェーズでは、最適なクエリ・アクセス・プランに関する決定は行われません。このフェーズの目的は、列挙フェーズの準備です。

- **列挙フェーズ**　このフェーズでオプティマイザは、最適化前フェーズで生成した構成要素を使用して、クエリの可能なアクセス・プランを列挙します。検索領域は非常に大きいため、オプティマイザは生成に独自の列挙アルゴリズムを使用し、生成されたアクセス・プランを削除します。プランごとにコスト推定が計算されます。コスト推定は、それまでの最適なプランと現在のプランとを比較するために使用されます。この比較時に、コストの高いプランは廃棄されます。コスト推定では、リソースの使用 (ディスクや CPU の操作など)、中間結果のロー数の予測値、最適化ゴール、キャッシュ・サイズなどが考慮されます。列挙プランの出力は、クエリの最適なアクセス・プランです。

- **プラン構築フェーズ**　プラン構築フェーズでは、最適なアクセス・プランを利用して、クエリの実行に使用するクエリ実行プランの対応する最終表現を構築します。Interactive SQL のプラン・ビューワで、プランのグラフィカル・バージョンを表示できます。グラフィカルなプランにはツリー構造があり、各ノードは特定の関係代数演算を実装する物理演算子です。たとえば [ハッシュ・ジョイン] や [順序付けされた Group By] は、それぞれジョイン操作や GROUP BY 操作を実装する物理演算子です。「グラフィカルなプランの解釈」 645 ページを参照してください。

  データベース・サーバによってプランがキャッシュ済みのクエリは、このクエリ処理のフェーズをスキップします。「プランのキャッシュ」 601 ページと「クエリ処理のフェーズをスキップするための条件」 575 ページを参照してください。

- **実行フェーズ**　クエリの結果は、プラン構築フェーズで構築されたクエリ実行プランを使用して計算されます。

クエリ処理のフェーズをスキップするための条件

ほぼすべての文が、すべてのクエリ処理のフェーズを経由します。ただし、例外が主に 2 つあります。「プランのキャッシュ」の恩恵を受けるクエリ (データベース・サーバによってプランがキャッシュ済みのクエリ) と「バイパス・クエリ」です。

- **プランのキャッシュ**　ストアド・プロシージャまたはユーザ定義関数内のクエリの場合、データベース・サーバは再利用できるように実行プランをキャッシュします。このクラスのクエリの場合、クエリ実行プランは実行後にキャッシュされます。次回クエリが実行されると、プランが取得され、実行フェーズまでのすべてのフェーズがスキップされます。「プランのキャッシュ」 601 ページを参照してください。

- **バイパス・クエリ**　バイパス・クエリは、データベース・サーバがオプティマイザをバイパスできると認識する特定の特性を持った、単純なクエリのサブクラスです。最適化をバイパスすることで、実行プランの構築時間を削減できます。

  クエリがバイパス・クエリとして認識されると、コストベースの最適化ではなくヒューリスティックが使用されます。つまり、セマンティック変形と最適化のフェーズはスキップされ、クエリの実行プランはクエリの解析ツリー表現から直接構築されます。

単純なクエリ

単純なクエリとは、次の特性を持った単一クエリ・ブロックの SELECT 文、INSERT 文、DELETE 文、UPDATE 文です。

- クエリ・ブロックに、サブクエリ、UNION や EXCEPT などの追加クエリ・ブロック、および共通テーブル式が含まれていない。
- クエリ・ブロックが 1 つのベース・テーブルまたはマテリアライズド・ビューを参照する。
- クエリ・ブロックには、TOP N 句、FIRST 句、ORDER BY 句、DISTINCT 句を含めることができる。
- クエリ・ブロックには、GROUP BY 句や HAVING 句を含まない集合関数を含めることができる。
- クエリ・ブロックに、Window 関数が含まれていない。
- クエリ・ブロック式に、NUMBER、IDENTITY、サブクエリが含まれていない。
- ベース・テーブルに定義された制約が単純な式である。

セマンティック変形フェーズで、複雑な文は単純な文に変形される場合があります。単純な文に変形されると、クエリをオプティマイザ・バイパスで処理したり、SQL Anywhere サーバでクエリ・プランをキャッシュできます。

最適化の実行と最適化の非実行の強制

プランのキャッシュまたはオプティマイザのバイパスの条件を満たすクエリは、SQL Anywhere オプティマイザでの処理を強制できます。これには、任意の SQL 文で FORCE OPTIMIZATION 句を使用します。

また、文がオプティマイザをバイパスするように強制することもできます。これには、文の FORCE NO OPTIMIZATION 句を使用します。データベース・オプションの設定またはスキーマやクエリの特性などが原因で、文が複雑すぎてオプティマイザをバイパスできない場合、クエリは失敗し、エラーが返されます。

FORCE OPTIMIZATION 句と FORCE NO OPTIMIZATION 句は、次の文の OPTION 句で使用できます。

- 「SELECT 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「UPDATE 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「INSERT 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「DELETE 文」 『SQL Anywhere サーバ - SQL リファレンス』

セマンティック・クエリ変形

SQL Anywhere は操作を効率化するために、セマンティック上は同等で、構文上は異なる形式にユーザのクエリを書き換えます。SQL Anywhere は、さまざまな書き換え操作を実行します。

アクセス・プランを読めば、それが元の文のリテラルな解釈と一致していないことがよくあります。たとえば、SQL 文の効率を高めるために、オプティマイザはジョインを使ってサブクエリを可能なかぎり書き換えようとします。

クエリ書き換えフェーズでは、より効率性と便宜性に優れたクエリの表現方法を検討するために、SQL Anywhere は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。

- 不要な DISTINCT 条件の排除
- サブクエリのネスト解除
- UNION または GROUP 化されたビューや派生テーブルでの述部のプッシュダウンの実行
- OR 述部と IN リスト述部の最適化
- LIKE 述部の最適化
- 外部ジョインの内部ジョインへの変換
- 外部ジョインと内部ジョインの削除
- 述部の推定による利用可能な条件の発見
- 大文字と小文字の不要な変換の排除
- サブクエリを EXISTS 述部として書き換え

注意

カーソルが更新可能な場合、メイン・クエリ・ブロックではクエリ書き換え最適化を実行できないことがあります。カーソルを読み込み専用として宣言すると、最適化を利用できます。「カーソル・タイプの選択」『SQL Anywhere サーバ - プログラミング』と「DECLARE CURSOR 文 [ESQL] [SP]」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

カーソルが更新可能なメイン・クエリ・ブロックで最適化を実行できない例については、「不要な内部ジョインと外部ジョインの削除」 584 ページを参照してください。

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、元の形式も順序が変わった形式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 句の述部の順序を自由に変更できます。

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。一般的に、サブクエリのネスト解除は、FROM 句にテーブルを多くても 1 つだけ含んだ相関サブクエリに対して常に実行されます。これらの相関サブクエリは、ANY、ALL、EXISTS の各述部で使用されます。クエリのセマンティクス上、サブクエリが返すローは多くても 1 つだけであると判断できる場合は、非相関サブクエリ、または FROM 句に複数のテーブルを含んだサブクエリはフラットにされます。

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見なします。

```
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。

```
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が実行される場合、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 化されたビューや派生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派生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、その製品を含む注文の件数と、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;
```


このクエリでは、ID カラムの値が 300 であるローだけに出力を制限しています。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;
```

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、定数だけ、またはクエリ・ブロック 1 回の実行中に定数である値 (外部参照など) だけを IN リストに指定してください。

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>
```

参照

- 「InList アルゴリズム (IN)」 639 ページ

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 に書き換えられる。
- 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。そのため、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。

次のいずれかの条件を満たす場合に、LEFT または RIGHT 外部ジョインが内部ジョインに変換されます。

- NULL 入力テーブルのカラムを参照しており NULL を許容しない述部が、クエリの WHERE 句にある場合。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。
- 外部ジョインの NULL 入力側は、保護された側の各ローについて、ローを 1 つだけ返す。この条件を満たす場合、NULL 入力されるローが存在せず、外部ジョインは内部ジョインと等しくなります。

このリライト最適化を外部ジョイン・クエリに適用できるのは、クエリが外部結合を使用して作成された 1 つ以上のビューを参照する場合です。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。

例 1

次のクエリの場合、ProductID カラムは NOT NULL と宣言されており、SalesOrderItems テーブルは外部キー "FK_ProductID_ID" ("ProductID") REFERENCING "Products" ("ID") を持つため、SalesOrderItems テーブルの各ローには Products テーブルと一致するローが 1 つだけ存在します。

次の SELECT 文は、リライト最適化後の書き換えられたクエリを示しています。

```
SELECT * FROM SalesOrderItems s LEFT OUTER JOIN Products p ON (p.ID = s.ProductID);
SELECT * FROM SalesOrderItems s JOIN Products p ON (p.ID = s.ProductID);
```

例 2

次のクエリは、数量の多い製品とそれに対応する注文をリストします。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。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形式になります。

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';
```

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派生テーブルは、派生の関係で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。
  - 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 入力可の場合は 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 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。そのため、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。

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

非相関サブクエリとは、その外部クエリに含まれる 1 つまたは複数のテーブルを明示的に参照しないサブクエリです。

次の例は、非相関サブクエリを含む一般的な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。この形式では、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、前述のクエリとセマンティック上は同等ですが、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」 635 ページを参照してください。

参照

- 「相関サブクエリと非相関サブクエリ」 535 ページ

ユーザ定義関数のインライン化

単純なユーザ定義関数は、クエリの一部として呼び出されたときにインライン化されることがあります。つまり、元のクエリと同等で、関数定義のないクエリに書き換えられます。テンポラリ関数、再帰関数、NOT DETERMINISTIC 句のある関数はインライン化されません。また、関数がサブクエリを使用して引数として呼び出された場合、または関数がテンポラリ・プロシージャの内部から呼び出された場合は、インライン化されません。

次のいずれかの形式の場合、ユーザ定義関数をインライン化できます。

- 1 つの RETURN 文を伴う関数。次に例を示します。

```
CREATE FUNCTION F1( arg1 INT, arg2 INT )
RETURNS INT
BEGIN
 RETURN arg1 * arg2
END;
```

- 1 つの変数を宣言し、その変数に代入して 1 つの値を返す関数。次に例を示します。

```
CREATE FUNCTION F2( arg1 INT )
RETURNS INT
BEGIN
 DECLARE result INT;
 SET result = ( SELECT ManagerID FROM Employees WHERE EmployeeID=arg1 );
 RETURN result;
END;
```

- 1 つの変数を宣言し、その変数に SELECT INTO して 1 つの値を返す関数。次に例を示します。


```
CREATE FUNCTION F3( arg1 INT )
RETURNS INT
BEGIN
 DECLARE result INT;
 SELECT ManagerID INTO result FROM Employees e1 WHERE EmployeeID=arg1;
 RETURN result;
END;
```

ユーザ定義関数の本文のコピー、呼び出しからの引数の挿入、適切な CAST 関数の挿入によってユーザ定義関数はインライン化され、書き換えられたクエリは元のクエリと同等になります。たとえば、定義済みの関数 F1 と同様の関数を作成し、次のようにクエリの FROM 句でプロシージャを呼び出すとします。

```
SELECT F1( e.EmployeeID, 2.5 ) FROM Employees e;
```

データベース・サーバは、次のようにクエリを書き換える場合があります。

```
SELECT CAST( e.EmployeeID AS INT ) * CAST( 2.5 AS INT ) FROM Employees e;
```

参照

- 「ユーザ定義関数の概要」 882 ページ
- 「CREATE FUNCTION 文 [Web サービス]」 『SQL Anywhere サーバ - SQL リファレンス』
- 「CAST 関数 [データ型変換]」 『SQL Anywhere サーバ - SQL リファレンス』
- 「SELECT 文」 『SQL Anywhere サーバ - SQL リファレンス』

単純なシステム・プロシージャのインライン化

単一の SELECT 文として定義されたシステム・プロシージャは、クエリの FROM 句で呼び出されたときにインライン化される場合があります。つまり、元のクエリと同等でプロシージャ定義のないクエリに書き換えられます。プロシージャがインライン化される場合は、派生テーブルとして書き換えられます。プロシージャがデフォルトの引数を使用する場合、または単一の SELECT 文以外が本文に存在する場合は、プロシージャはインライン化されません。

たとえば、次のプロシージャを作成するとします。

```
CREATE PROCEDURE Test1( arg1 INT )
 BEGIN
  SELECT * FROM Employees WHERE EmployeeID=arg1
 END;
```

ここで、次のようにクエリの FROM 句でこのプロシージャを呼び出すとします。

```
SELECT * FROM Test1( 200 );
```

データベース・サーバは、次のようにクエリを書き換える場合があります。

```
SELECT * FROM ( SELECT * FROM Employees WHERE EmployeeID=CAST( 200 AS INT ) ) AS Test1;
```

参照

- 「ユーザ定義関数の概要」 882 ページ
- 「CREATE FUNCTION 文 [Web サービス]」 『SQL Anywhere サーバ - SQL リファレンス』
- 「SELECT 文」 『SQL Anywhere サーバ - SQL リファレンス』

オプティマイザの仕組み

オプティマイザの役割は、SQL 文を実行する効率的な方法を考案することです。これを行うには、オプティマイザは、クエリの実行プランを判断する必要があります。これには、クエリで参照されるテーブルのアクセス順序、各テーブルに使用されるジョイン演算子とアクセス方式、クエリの各部分の計算でクエリで参照されないマテリアライズド・ビューを使用できるかどうかなどの判断が含まれます。オプティマイザは、クエリで可能なアクセス・プランを生成してコストを計算するときに、ジョイン列挙フェーズ中にクエリを実行するための最適なアクセス・プランを選択します。最適なアクセス・プランでは、オプティマイザの推測値が最短時間と最低コストで望ましい結果セットを返します。オプティマイザは、ディスクへの必要な読み書き回数を推定して列挙された各方式のコストを決定します。

Interactive SQL では、**[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]** ウィンドウを開いて、**[プラン]** タブの設定を変更します。「グラフィカルなプランの解釈」 645 ページと「実行プランの解釈」 642 ページを参照してください。

最初のローを返すコストの最小化

オプティマイザは、汎用的なディスク・アクセス・コスト・モデルを使用して、データベース・ファイルに対するランダム検索と逐次検索の相対的なパフォーマンスの差異を認識します。ALTER DATABASE 文を使用すると、データベースを特定のハードウェア構成に対応させることができます。「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。

セマンティック上等しい構文の使用

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同等ですが、構文はまったく異なる場合があります。ごくわずかな例外はありますが、オプティマイザは、各文のセマンティックだけに基づいて適切なアクセス・プランを考案します。

構文の違いは重要に見えるかもしれませんが、通常はまったく影響ありません。たとえば、クエリ構文で述部、テーブル、属性の順序が違ってもアクセス・プランの選択には影響しません。クエリに非マテリアライズド・ビューが含まれているかどうかによってオプティマイザが影響を受けることもありません。

オプティマイザ・クエリのコスト削減

オプティマイザが実現可能な最も効率の良いアクセス・プランを見つけるのが理想的ですが、通常、これは現実的ではありません。複雑なクエリの場合、さまざまな可能性が存在することがあります。

オプティマイザは効率的ですが、各オプションの分析には、時間とリソースを必要とします。オプティマイザは、これから行う最適化のコストと、これまでに見つけた最高のプランの実行にかかるコストを比較します。相対的にコストの低いプランが考案されると、オプティマイザは停止し、そのプランの実行を進めます。さらに最適化を行うと、すでに見つかっているアクセス・プランを実行する場合よりも多くのリソースを消費する可能性があります。optimization_level オプションの値を高く設定することで、オプティマイザが費やす作業量を指定できます。「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。「カラム統計の更新によるオプティマイザのパフォーマンス向上」 593 ページを参照してください。

オプティマイザによるカラム統計の使用方法

オプティマイザが使用するカラム統計の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。

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。

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。

オプティマイザによるヒューリスティックの使用方法

採用できそうな実行プラン内の各テーブルについて、オプティマイザは結果の一部となるローの数を推定します。ローの数は、テーブルのサイズとクエリの WHERE 句または ON 句に指定された制限によって異なります。

SQL Anywhere は、カラムに対する特定のクエリ述部を満たすローの数を、そのカラムに対するヒストグラムによって推定します。そのためには、指定の述部を満たす値を含むすべての範囲にあるローの数を合算します。クエリの結果セットに部分的に含まれるヒストグラムの値の範囲には、内挿法が使用されます。

多くの場合、オプティマイザはより高度なヒューリスティックを使用します。たとえば、オプティマイザは適切な統計がない場合にかぎってデフォルトの推定値を使用します。また、オプティマイザは、インデックスとキーを使用してロー数の推定精度を上げます。単一のカラムで推定する例を次に示します。

- カラム内である値を持つロー数：そのカラムがユニーク・インデックスを持つかプライマリ・キーである場合、ロー数は 1 つだと推定します。
- インデックス付きカラムで定数と比較したときのロー数：インデックスを調査し、比較条件を満たすローのパーセンテージを推定します。
- 外部キーからプライマリ・キーへのロー数 (キー・ジョイン)：テーブルの相対的サイズを使って推定します。たとえば、5000 ローのテーブルが 1000 ローのテーブルに対して外部キーを持つとき、オプティマイザは 1 つのプライマリ・キー・ローに対して 5 個の外部キー・ローがあると推定します。

参照

カラム値の分散の詳細については、次の各項を参照してください。

- 「ESTIMATE 関数 [その他]」 『SQL Anywhere サーバ - SQL リファレンス』
- 「ESTIMATE_SOURCE 関数 [その他]」 『SQL Anywhere サーバ - SQL リファレンス』

オプティマイザによるプロシージャ統計の使用方法

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、ProCall 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 (PC)」 640 ページを参照してください。

参照

述部の選択性の取得については、次の各項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』

カラム統計の更新によるオプティマイザのパフォーマンス向上

カラム統計は、データベースのシステム・テーブル ISYSCOLSTAT に永久的に格納されます。オプティマイザのパフォーマンスを継続的に向上させるために、データベース・サーバは SELECT 文、INSERT 文、UPDATE 文、DELETE 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に一致するローの数をモニタリングし、その数を推定されたローの数と比較し、必要に応じて既存の統計を更新することで行います。

利用可能なカラム統計の精度が高くなると、それに従ってオプティマイザがより適切に推定を計算できるため、以降のクエリのパフォーマンスが向上します。

データベース・オプションを使用して、カラム統計を更新するかどうかを設定できます。update_statistics データベース・オプションは、クエリの実行中にカラム統計を更新するかどうかを指定します。collect_statistics_on_dml_updates データベース・オプションは、LOAD、INSERT、DELETE、UPDATE などのデータを変更する DML 文の実行中に統計を更新するかどうかを指定します。

統計が現在のカラムの値を正確に反映していないためにパフォーマンスが悪いと考えられる場合は、CREATE STATISTICS 文や DROP STATISTICS 文を実行します。CREATE STATISTICS は古い統計を削除して新しい統計を作成し、DROP STATISTICS は古い統計だけを削除します。

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。

参照

- 「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。「カラム統計の更新によるオプティマイザのパフォーマンス向上」 593 ページを参照してください。

オプティマイザの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要素であり、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。「カラム統計の更新によるオプティマイザのパフォーマンス向上」 593 ページと「インデックスの操作」 75 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。「カラム統計の更新によるオプティマイザのパフォーマンス向上」 593 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探索条件を評価できます。インデックスを使用することで、オプティマイザはデータへのアクセスをスピードアップし、ベース・テーブルから読み込んで処理する情報量を減らします。たとえば、クエリに WHERE column-name=value という探索条件があり、カラムにインデックスが存在する場合、インデックス・スキャンを使って、探索条件を満たすテーブルのローだけを読み込むことができます。

テーブルをジョインする場合にインデックスがあれば、パフォーマンスを大幅に向上させることもできます。

オプティマイザは、可能であれば常に、インデックス専用取得でクエリを処理しようとします。インデックス専用取得では、データベース・サーバでインデックスのデータだけを使用してクエリが処理されるため、テーブル内のローにアクセスする必要がありません。

オプティマイザが使用できるインデックスがない場合は、代わりに逐次テーブル・スキャンが実行されますが、これにはコストがかかります。

オプティマイザは、最高のパフォーマンスが得られるインデックスを判断し、それを使用するように自動的に選択します。ただし、クエリでインデックス・ヒントを使用して、オプティマイザが使用するインデックスを指定することもできます。指定したインデックスが 1 つでも使用できなかった場合はエラーが返されます。インデックス・ヒントによってパフォーマンスが低下する場合があるため、経験のあるユーザだけが実行するよう注意してください。「FROM 句」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

インデックス・コンサルタントを使用して、データベースにインデックスを追加する必要があるかどうかを判別できます。「インデックス・コンサルタント」 199 ページを参照してください。

参照

- 「クエリでの述部の使用」 598 ページ

仮想メモリは貴重なリソースである

オペレーティング・システムといくつかのアプリケーションは、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。

メモリ・ガバナー

SQL Anywhere データベース・サーバはキャッシュ (「バッファ・プール」とも呼ぶ) を使用し、データベース・ページのイメージを一時的に格納 (バッファ) します。これらのページは通常、テーブル・ページとインデックス・ページです。ただし、SQL Anywhere データベースには他の種類の物理ページも格納されます。これらのページに加えて、データベース・サーバは他に 2 つのメモリ・プールのキャッシュを使用します。これらのプールの 1 つは仮想メモリです。接続、文、カーソルを表す構造など、データベース・サーバのデータ構造に使用されます。2 番目のプールは、「クエリ・メモリ」の仮想ストレージとして使用されるキャッシュ・ページで構成されます。

ハッシュ・ジョインやソートなどのクエリ実行アルゴリズムでは、効率良く操作するためにメモリが必要です。SQL Anywhere は「メモリ・ガバナー」を使用して、各文がクエリの実行に使用できるクエリ・メモリの量を決定します。メモリ・ガバナーはクエリ・メモリ・プールを文に割

り付け、負荷を効率的に実行できるようにします。データベース・サーバのプロパティ QueryMemPages に、分散に使用できるクエリ・メモリ・プール内のページ数が表示されます。プール・サイズは、サーバの最大キャッシュ・サイズに対する割合 (キャッシュ・サイズの上限) で設定されます。この値は、-ch サーバ・オプションで制御できます。データベース・サーバ・プロパティ QueryMemPercentOfCache で、クエリ・メモリに設定できる最大キャッシュ・サイズの割合を指定します。この割合は 50% です。

メモリ・ガバナーは選択されたページ数を各文に付与し、文はメモリを大量に消費するクエリ処理アルゴリズムにそのページを使用できます。クエリ処理アルゴリズムがページを使用するまでは、引き続きクエリ・メモリ・プール内のメモリを他の目的 (テーブルのバッファリングやインデックス・ページなど) にも使用できます。クエリ・メモリを使用し、メモリを大量に消費するクエリ処理アルゴリズムには、ハッシュ DISTINCT、ハッシュ GROUP BY、ハッシュ・ジョインなどのあらゆるハッシュベースの演算子や、ソート演算子、ウィンドウ演算子があります。

文が実行を開始すると、メモリ・ガバナーはオプティマイザの推定を使用して、文が使用できるメモリ容量を決定します。この推定は、グラフィカルなプランに QueryMemMaxUseful として表示されます。文のクエリ・メモリは、要求のアクセス・プランで使用される、メモリを大量に消費する特定の演算子間で割り付けられます。交換演算子の下にある、メモリを大量に消費する並列演算子はそれぞれ固有のクエリ・メモリの割り付けを受け取ります。単純な要求の場合は、大容量のメモリによる恩恵を受けません。ただし、ハッシュベースの演算子またはソートを使用する要求は、必要なローをすべてメモリ内に保持できるだけのメモリ容量があると処理効率が向上します。

データベース・サーバのマルチプログラミング・レベルが高くなると、同時実行タスクまたは要求が追加されるたびにデータベース・サーバはクエリ・メモリ容量の予約が必要になり、特定の要求に使用できる容量が減ります。さらにメモリ・ガバナーは、メモリを大量に消費する要求の同時実行数を制限します。この最大値は、データベース・サーバを実行するコンピュータのパフォーマンス特性に応じて選択されます。また、この制限は、サーバ・プロパティ QueryMemActiveMax に表示されます。メモリ・ガバナーは、メモリを大量に消費する要求の同時実行数の推定を継続します。この推定はデータベース・サーバ・プロパティおよびパフォーマンス・モニタの統計に使用できます。メモリ・ガバナーはこの実行数の平均を使用して、クエリ・メモリ・プールからのメモリの割り付け量を決定します。メモリを大量に消費する要求の実行数が少ない場合、より多くのメモリが各要求に割り付けられます。実行数が多い場合は、クエリ・メモリをより均等に共有できるように、各要求への割り付けが少なくなります。このとき、各要求で使用できると推定されるクエリ・メモリ・ページ数が考慮されます。

メモリを大量に消費する文の実行を開始したとき、メモリを大量に消費する要求の最大同時実行数にすでに達している場合は、いずれかの実行中の要求が割り付けられたメモリを解放するまで入力された文は待機します。query_mem_timeout データベース・オプションは、入力された要求にメモリが付与されるまで待つ時間を制御します。デフォルト設定 -1 を使用すると、要求はデータベース・サーバで定義された期間待機します。待機後、使用できるメモリが付与されない場合は文のアクセス・プランは少ないメモリ容量で実行されます。これによって実行が遅くなる場合がありますが、プラン内のメモリを大量に消費する物理演算子に対して実行時のメモリ量が少なくなる方式を使用できる場合は、その方式が使用されます。データベース・サーバ・プロパティおよびパフォーマンス・モニタの統計 QueryMemGrantWaiting に、メモリの割り付けを待つ現在の要求数が表示されます。また、QueryMemGrantWaited に、要求しているメモリが付与されるまで待機する必要がある合計回数が表示されます。

グラフィカルなプランの QueryMemNeedsGrant 値には、単純な要求 (メモリの付与が不要) かメモリを大量に消費する (メモリの付与が必要) かをメモリ・ガバナーが検討した結果が表示されま

す。メモリ・ガバナーによってメモリの付与が不要と分類された要求は、ただちに実行されます。それ以外の場合、要求は一定割合のクエリ・メモリ・プールの使用を求めます。グラフィカルなプランの QueryMemLikelyGrant 値には、要求を実行するために付与されることが予想される推定ページ数が表示されます。

参照

- QueryMemActiveMax プロパティ：「データベース・サーバ・プロパティ」『SQL Anywhere サーバ - データベース管理』
- QueryMemPages プロパティ：「データベース・サーバ・プロパティ」『SQL Anywhere サーバ - データベース管理』
- QueryMemPercentOfCache プロパティ：「データベース・サーバ・プロパティ」『SQL Anywhere サーバ - データベース管理』
- 「query_mem_timeout オプション [データベース]」『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』
- 「統計情報付きのグラフィカルなプラン」 646 ページ
- 「-ch サーバ・オプション」『SQL Anywhere サーバ - データベース管理』

クエリでの述部の使用

「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';
```

クエリを書き換えて検索引数可能にすると、関数を使用しなくても済みます。たとえば、上記のクエリを次のように書き換えることができます。

```
SELECT *
FROM SalesOrders
WHERE OrderDate > '1999-12-31'
AND OrderDate < '2001-01-01';
```

関数値を計算カラムに格納し、このカラムのインデックスを構築すると、関数を使用するクエリが検索引数可能になります。「計算カラム」は、テーブル内の他のカラムから値を取得するカラムです。たとえば、注文の日付を保持するカラム OrderDate がある場合は、OrderDate カラムから抽出した年の値を保持する計算カラム OrderYear を作成できます。

```
ALTER TABLE SalesOrders
ADD OrderYear INTEGER
COMPUTE ( YEAR( OrderDate ) );
```

次に、カラム OrderYear のインデックスを通常どおり追加できます。

```
CREATE INDEX IDX_year
ON SalesOrders ( OrderYear );
```

次の文を実行すると、データベース・サーバは、情報を保持するインデックス・カラムがあることを認識し、そのインデックスを使用してクエリに応答します。

```
SELECT * FROM SalesOrders
WHERE YEAR( OrderDate ) = '2000';
```

計算カラムのドメインは、カラムが置換されるように、COMPUTE 式のドメインと同じにします。上記の例では、YEAR(OrderDate) が integer ではなく string を返した場合には、オプティマイザは式の計算カラムを置換しません。その結果、必要なローの取り出しにインデックス IDX_year を使用できなくなってしまいます。

計算カラムの詳細については、「計算カラムの使用」 31 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

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。この形式では、属性 *x* のインデックスを使用して、一定範囲内の値を検索できます。幸い、SQL Anywhere では、この特殊な変形を自動で行います。

テーブルでのインデックス検索に使用される検索引数可能な述部は、「マッチング」述部です。WHERE 句には、多くのマッチング述部が含まれることがあります。最も適切な述部は、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」 585 ページを参照してください。

MIN 関数と MAX 関数を使用したコスト・ベースの最適化

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。SQL Anywhere サーバは、これらの文を繰り返し最適化するコストを削減するために、次の文に関するプランのキャッシュを検討します。

- ストアド・プロシージャ、ユーザ定義関数、トリガ内で実行されるすべての文。
- 最適化をバイパスするための条件を満たす SELECT 文、INSERT 文、UPDATE 文、DELETE 文。「クエリ処理のフェーズ」 574 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識別する探索条件を含んでいる必要があります。プランをキャッシュする場合は、他の探索条件は指定できません。また、UPDATE 文については、変数の代入を含む SET 句がある文はキャッシュの条件を満たしません。

1 つの接続でこのいずれかの文が複数回実行されたら、オプティマイザはホスト変数の値が不明なまま文の再利用可能なプランを構築します。選択性推定やセマンティック・クエリ変形にホスト変数の値を使用できないため、再利用可能なプランのコストは高くなる場合があります。再利用可能なプランの構造が、文が以前に実行されたときのプランと同じである場合、データベース・サーバは、そのプラン・キャッシュに再利用可能なプランを追加します。最適化を避けることによるコストの削減よりも、実行のたびに最適化することによる利点が勝る場合、実行プランはキャッシュされません。

文で参照されないマテリアライズド・ビューを実行プランで使用し、materialized_view_optimization オプションが Stale 以外に設定されている場合、実行プランはキャッシュされず、ストアド・プロシージャ、ユーザ定義関数、トリガの次回呼び出し時に文が最適化されます。

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、これはすべてのクエリ処理フェーズを経由させて文を処理するよりも、はるかに高速です。使用頻度が低く、キャッシュの使用量が増えない場合、キャッシュされたプランはディスクに格納されます。オプティマイザはクエリを定期的に最適化し直して、キャッシュされたプランが依然として効率的であるかどうかを確認します。

キャッシュできるプランの最大数は、max_plans_cached オプションで指定します。デフォルトは 20 です。プランのキャッシュを無効にするには、このオプションを 0 に設定します。「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

- 「クエリ処理のフェーズをスキップするための条件」 575 ページ
- 「マテリアライズド・ビューによるパフォーマンスの向上」 603 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』

マテリアライズド・ビューによるパフォーマンスの向上

マテリアライズド・ビューとはビューの 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、マテリアライズド・ビューはビューでもあり (クエリ指定がある)、テーブルでもあります (永続的な実体化したローがある)。したがって、テーブルで実行する多くの操作は、マテリアライズド・ビューでも実行できます。たとえば、マテリアライズド・ビューに対して、インデックス構築やアンロードを実行できます。

マテリアライズド・ビューの定義

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、マテリアライズド・ビューを定義することを検討してください。マテリアライズド・ビューは、次のような環境でのパフォーマンスを向上することを目的に設計されています。

- データベースのサイズが大きい。
- 頻繁なクエリにより、大量のデータに対する集約操作やジョイン操作が繰り返し実行される。
- 基本となるデータへの変更は頻度が比較的少ない。
- 最新のデータにアクセスすることは重大な要件ではない。

マテリアライズド・ビュー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、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マテリアライズド・ビューは向いています。

最適化時に送信されるクエリを部分的または全体的に満たす候補として見なされるマテリアライズド・ビューについて、オプティマイザはそのリストを管理します。オプティマイザがクエリの全体または部分を満たす候補となるマテリアライズド・ビューを検出すると、最適化の列挙フェーズ (コストを基に最適なプランを判断する) で行われる推奨の対象にそのビューを含めます。マテリアライズド・ビューをクエリに一致させるためにオプティマイザが使用する処理を「ビュー・マッチング」と呼びます。オプティマイザがマテリアライズド・ビューを検討するには、そのビューが一定の条件を満たす必要があります。つまり、マテリアライズド・ビューがクエリによって明示的に参照されないかぎり、オプティマイザによって使用される保証がありません。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。

マテリアライズド・ビューの使用が許可されていることをオプティマイザが判断すると、候補となるマテリアライズド・ビューのそれぞれが検査されます。マテリアライズド・ビューは、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・サーバでマテリアライズド・ビューを使用できる。「マテリアライズド・ビューの有効化と無効化」 67 ページを参照してください。
- 最適化でマテリアライズド・ビューを使用できる。「オプティマイザによるマテリアライズド・ビューの使用の有効化と無効化」 70 ページを参照してください。
- マテリアライズド・ビューが初期化されている。「マテリアライズド・ビューの初期化」 61 ページを参照してください。

- マテリアライズド・ビューが検討すべきオプティマイザ要件をすべて満たしている。「マテリアライズド・ビューとビュー・マッチング・アルゴリズム」 605 ページを参照してください。
- マテリアライズド・ビューを作成するための重要なオプションの値が、クエリを実行する接続のオプションと一致する。「マテリアライズド・ビューの制限」 57 ページを参照してください。
- マテリアライズド・ビュー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しきい値を超えていない。「オプティマイザでのマテリアライズド・ビューに対する古さのしきい値の設定」 71 ページを参照してください。

マテリアライズド・ビューが上記の基準を満たし、クエリの全体または一部を満たすことがわかった場合、コストベースの最適プランが検出されると、ビュー・マッチング・アルゴリズムは最適化の列挙フェーズ用の推奨にマテリアライズド・ビューを含めます。ただし、それにより最終的にそのマテリアライズド・ビューが最終実行プランで使用されるというわけではありません。たとえばマテリアライズド・ビューを使用しない別のアクセス・プランの方がコストが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マテリアライズド・ビューであっても使用されない可能性があります。

マテリアライズド・ビューの候補リストの決定

オプティマイザが検討する候補となるすべてのマテリアライズド・ビューのリストは、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';
```

オプティマイザはリストの生成時にオプション設定を考慮するため、返されるリストは要求している接続に固有です。接続に指定したオプションとマテリアライズド・ビューの作成時のオプションとが一致しない場合、そのマテリアライズド・ビューは候補と見なされません。一致する必要のあるオプションのリストについては、「マテリアライズド・ビューの制限」 57 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マテリアライズド・ビューのリストを取得するには、クエリを実行する接続から次のクエリを実行します。

```
SELECT * FROM sa_materialized_view_info( ) WHERE AvailForOptimization='O';
```

マテリアライズド・ビューが検討されたかどうかの判断

特定のクエリで使用されるマテリアライズド・ビューのリストは、Interactive SQL でクエリのグラフィカルなプランの **[高度な詳細]** ウィンドウで確認できます。「実行プランの解釈」 642 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マテリアライズド・ビューが検討されたかどうかを判断できます。オプティマイザによって列挙されるアクセス・プランを確認するには、トレーシングをオンにして、OPTIMIZATION_LOGGING トレーシング・タイプを含めるように設定する必要があります。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」 191 ページと「診断トレーシング・レベルの選択」 207 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」 574 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マテリアライズド・ビューを検討しません。

マテリアライズド・ビューとビュー・マッチング・アルゴリズム

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マテリアライズド・ビューを使用できるかどうかを判断します。この判断は、クエリ検査ステップとマテリアライズド・ビュー検査ステップの 2 つのステップで行われます。

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マテリアライズド・ビューのセットに、オプティマイザはそのマテリアライズド・ビューを含めます。

- 含まれるクエリ・ブロックが 1 つだけである。
- 含まれる FROM 句が 1 つだけである。
- 次の構成体や仕様のいずれも含まない。
  - GROUPING SETS
  - CUBE
  - ROLLUP
  - サブクエリ
  - 派生テーブル
  - UNION
  - EXCEPT
  - INTERSECT
  - マテリアライズド・ビュー
  - DISTINCT
  - TOP
  - FIRST
  - セルフジョイン
  - 再帰ジョイン
  - FULL OUTER JOIN

マテリアライズド・ビューの定義には、GROUP BY 句や HAVING 句 (HAVING 句に subselect やサブクエリが含まれない場合) を含めることができます。

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マテリアライズド・ビューのみに適用されます。クエリでマテリアライズド・ビュー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」 642 ページ
- 「クエリ処理のフェーズ」 574 ページ
- 「アプリケーション・プロファイリング」 191 ページ

クエリ検査

クエリ検査では、ビュー・マッチング・アルゴリズムによってクエリが検査されます。次のいずれかの条件に該当する場合、クエリの処理にマテリアライズド・ビューは使用されません。

- クエリで参照されるすべてのテーブルが更新可能である。

  派生の関係で更新可能である SELECT 文や、更新可能なカーソルで明示的に宣言された SELECT 文については、オプティマイザはマテリアライズド・ビューを検討しません。この状況は、Interactive SQL を使用していると発生することがあります。デフォルトで、Interactive SQL は SELECT 文で更新可能なカーソルを利用します。

- 文は、オプティマイザ・バイパスを使用する単純な DML 文であり、ヒューリスティックに最適化される。ただし、OPTION 句で FORCE OPTIMIZATION を使用することにより、任意の SELECT 文でコストベースの最適化を使用できます。「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」 601 ページを参照してください。

マテリアライズド・ビュー検査

マテリアライズド・ビュー検査では、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マテリアライズド・ビューが判断されます。

マテリアライズド・ビュー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 (マテリアライズド・ビュー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。

マテリアライズド・ビュー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。マテリアライズド・ビュー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・マッチング・アルゴリズムによって、ビュー V がクエリ Q に属するクエリ・ブロック QB の一部と一致すると判断され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ている必要があります。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マテリアライズド・ビューの例については、「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」 611 ページを参照してください。

- マテリアライズド・ビュー 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、「マテリアライズド・ビューの制限」 57 ページを参照してください。
- マテリアライズド・ビュー 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しきい値を超えていない。「オプティマイザでのマテリアライズド・ビューに対する古さのしきい値の設定」 71 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 内のテーブル以外には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。

  - COVAR_SAMP(y,x)
  - COVAR_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。

ビュー・マッチング・アルゴリズムの例

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マテリアライズド・ビュー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マテリアライズド・ビュー 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。このマテリアライズド・ビューは 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 マテリアライズド・ビューの 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、このマテリアライズド・ビュー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 マテリアライズド・ビュー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 マテリアライズド・ビューを使用するクエリ 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。オプティマイザは、マテリアライズド・ビューを使用するクエリを書き換えるのではなく、マテリアライズド・ビューを使用するアクセス・プランを生成するということに注意してください。このアクセス・プランは理論上、書き換えられたクエリに対応すると見なすことができます。

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 マテリアライズド・ビューを使用します。

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 マテリアライズド・ビュー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
       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

グループ化されたマテリアライズド・ビューは、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マテリアライズド・ビュー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ます。そのため、マテリアライズド・ビュー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。

以下のマテリアライズド・ビュー 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、クエリ 4 の計算に必要なすべてのデータがビュー V_quantity に含まれているため、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マテリアライズド・ビューを使用できるため、クエリ全体で複数のマテリアライズド・ビューを使用できます。以下のクエリ 5 では、3 つのマテリアライズド・ビューを使用します。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
 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：マテリアライズド・ビュー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。マテリアライズド・ビューで 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 を含むマテリアライズド・ビュー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、マテリアライズド・ビュー 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：

```
 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'
  FOR READ ONLY;
```

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
                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;
```

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・プランの関係代数演算子

アクセス・プランは、関係代数演算子のツリーで構成されます。ツリーのルートで最終的な結果が得られるように、ツリーの最下位から開始して、クエリへの基本入力 (通常はテーブルのロー) を消費し、下から上にローを処理します。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」 642 ページと「グラフィカルなプランの解釈」 645 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・ガバナーのしきい値に近いので、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。

  演算子で使用できるメモリ容量は、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。

  メモリ・ガバナーとマルチプログラミング・レベルの詳細については、次の項を参照してください。

  - 「SQL Anywhere でのスレッド」 『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』
  - 「メモリ・ガバナー」 596 ページ

- 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。

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 (Exchange)」 637 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)。

オプティマイザは、並列処理の追加コスト (ローの追加コピー、作業量の調整のための追加コスト) を推定し、パフォーマンスの向上が期待できる場合のみ並列プランを選択します。

クエリ内並列処理は、priority オプションが background に設定されている接続には使用されません。「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コンピュータの CPU コア数を最近超えた場合は、クエリ内並列処理は使用されません。正確な時間はサーバによって決められ、通常は数秒です。「データベース・サーバ・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、派生テーブルで TOP *n* と ORDER BY を使用するクエリは、ソートがプランの一番上にないので、並列処理が使用される確率が低くなります。

SQL Anywhere では、DB 領域がデフォルトで単一プラッタのディスク・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 (Exchange)」 637 ページ
- 「実行プランの解釈」 642 ページ
- 「ALTER DATABASE 文」 『SQL Anywhere サーバ - SQL リファレンス』

クエリの並列処理

クエリは、クエリが処理するロー数が、返されるロー数よりも多い場合に並列処理が使用される確率が高くなります。この場合、処理されるロー数には、スキャンされるすべてのローのサイズとすべての中間結果のサイズが含まれます。インデックスを使用してテーブルのほとんどがスキップされるので、スキャンされないローは含まれません。理想的なケースは、大きなテーブルに対する単一ローの GROUP BY で、この場合、多数のローがスキャンされ、1 つのローだけが返されます。グループのサイズが大きい場合は、複数グループのクエリも並列処理の確率が高くなります。多数のローを削除する述部またはジョイン条件も高い確率で並列処理されます。

最適化または実行のときにクエリに並列処理が使用されない場合のリストを次に示します。

- サーバのコンピュータに複数のプロセッサがない。
- サーバのコンピュータに、複数のプロセッサを使用するためのライセンスがない。これは、NumLogicalProcessorsUsed サーバ・プロパティで確認できます。ただし、ハイパースレッドのプロセッサはクエリ内並列処理対象として数えられないので、コンピュータでハイパースレッドを使用している場合は、NumLogicalProcessorsUsed の値を 2 で割ります。
- max_query_tasks オプションが 1 に設定されている。
- priority オプションが background に設定されている。
- クエリを含む文が SELECT 文ではない。
- 最近、ActiveReq の値が NumLogicalProcessorsUsed の値以上になったことがある (コンピュータでハイパースレッドを使用している場合はプロセッサ数を 2 で割る)。
- 使用可能なタスク数が不十分である。

参照

- 「クエリ実行時の並列処理」 617 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』
- 「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』
- max_query_tasks、priority、NumLogicalProcessorsUsed、ActiveReq プロパティ：「データベース・サーバ・プロパティ」 『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』

テーブルへのアクセス方法

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。

IndexScan メソッド

IndexScan 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。インデックス・スキャンは、条件に合うローのセットを削減してからテーブルにアクセスするために役立ちます。インデックス・スキャンによって、ローをソートして返すことができます。

IndexScan は、短いプランに 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、テーブルから多数のローを読み込む場合は、インデックス・スキャンは逐次スキャンよりもコストが高くなる場合があります。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。

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。

一致する探索条件がない場合でも、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。

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。

クエリを記述するときに、インデックス・ヒントを指定して、オプティマイザが使用するインデックスとその使用方法を指定することもできます。ただし、インデックス・ヒントはクエリ・オプティマイザの意思決定論理を上書きするため、経験のあるユーザのみ使用してください。インデックス・ヒントを使用すると、次善のアクセス・プランが使用され、パフォーマンスが低下することがあります。「FROM 句」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

IndexOnlyScan 方式 (IO)

オプティマイザで使用されるインデックスに、クエリを満たすために必要な基本となるテーブルのすべてのデータが含まれる場合、基本となるテーブルからの値の読み込みを完全に回避して、

インデックスから直接データを取得できる場合があります。この方法を「インデックス専用取得」と呼びます。インデックス専用取得では、クエリを満たすために必要な I/O およびキャッシュの量が削減され、パフォーマンスが向上します。オプティマイザは、可能な場合は常にインデックス専用取得を実行します。

参照

- 「FROM 句」 『SQL Anywhere サーバ - SQL リファレンス』
- 「クエリでの述部の使用」 598 ページ
- 「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』
- 「実行プランの解釈」 642 ページ

MultipleIndexScan 方式 (MultIdx)

論理演算子 AND または OR で組み合わされた探索条件のセットを含むクエリを満たすために、複数のインデックスを使用できる、または使用する必要がある場合は、MultipleIndexScan を使用します。MultipleIndexScan では、探索条件を満たすために複数の IndexScan 方式を他の演算子と組み合わせます。

AND 演算子で組み合わせた述部の評価に複数のインデックスを使用する場合、MultipleIndexScan はインデックスの論理積操作を実行します。OR 演算子で組み合わせた述部の評価に複数のインデックスを使用する場合、MultipleIndexScan はインデックスの論理和操作を実行します。ただし、MultipleIndexScan は論理和や論理積操作に限定されません。たとえば、MultipleIndexScan は外部ジョインを使用してインデックスの論理和を実行する場合があります。

実行プランを調べると、複合インデックス・スキャンが特定のクエリに使用されているかどうかを確認できます。短いプランでは、複合インデックス・スキャン方式は table-name<MultIdx... のように表示され、使用されたインデックスのリストがその後に続きます。

長いプランおよびグラフィカルなプランでは、複合インデックス・スキャンの使用は MultipleIndexScan ノードで示されます。ノードの下のエントリに、使用されたインデックスとインデックスの結合結果が表示されます。

参照

- 「FROM 句」 『SQL Anywhere サーバ - SQL リファレンス』
- 「クエリでの述部の使用」 598 ページ
- 「実行プランの解釈」 642 ページ

ParallelIndexScan 方式

ParallelIndexScan を使用すると、各 IndexScan 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 IndexScan 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、ページからローを一度に 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、IndexScan 演算子間でページが分割されます。IndexScan 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。

TableScan 方式 (seq)

TableScan 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。これは、逐次テーブル・スキャンと呼ばれます。

逐次テーブル・スキャンは、短いテキスト・プランと長いテキスト・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。

逐次テーブル・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すべてのローにロックをかけますが、インデックス・スキャンは探索条件と一致するローだけにロックをかけます。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。

ParallelTableScan 方式

ParallelTableScan を使用すると、各 TableScan 演算子は交換演算子とともに使用することで、テーブル・スキャンを並列に実行できます。各 TableScan 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、ページからローを一度に 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、TableScan 演算子間でページが分割されます。並列 TableScan 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。

HashTableScan 方式 (HTS)

HashTableScan は、ハッシュ・ジョインの構築側をイン・メモリ・テーブルのようにスキャンします。これを使用して、次の 1 番目の構造を持つプランを、2 番目の構造のプランに変換できます。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>*JH (<operator>... (table2<seq>))

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、ハッシュ・テーブル・スキャンによって、他の演算子による処理が必要な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。

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しきい値の計算と同様の方法でしきい値の構築サイズを計算します。構築側のローの数がこのしきい値を超えると、HashTableScan が中止され、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。

RowIdScan 方式 (ROWID)

RowIdScan 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ローを検索するために使用されます。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 スキャンは、短いテキスト・プランと長いテキスト・プランに correlation-name<ROWID> 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。

RowIdScan 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、RowIdScan は空の集合を返します。

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』

アルゴリズムの種類

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。

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」と呼びます。

  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 (ほとんどの場合、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イン・メモリ・ハッシュ・テーブルを作成してから、大きい方の入

力を読み込みます。次に、イン・メモリ・ハッシュ・テーブルを調査して一致するローを検索します。

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イン・メモリ・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。

HashJoin アルゴリズム (JH、JHSP、JHFO、JHAP、JHO、JHPO)

HashJoin は、2 つの入力のうち小さい方のイン・メモリ・ハッシュ・テーブルを作成してから、大きい方の入力を読み込みます。次に、イン・メモリ・ハッシュ・テーブルを調査して一致するローを検索します。見つかったローはワーク・テーブルに書き込まれます。小さい方の入力がメモリに収まらない場合は、HashJoin によって両方の入力が小さなワーク・テーブルに分割されます。これらの小さくなったワーク・テーブルは、小さい方の入力がメモリに収まるようになるまで再帰的に処理されます。

また、HashJoin には次のような特性があります。

- 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。

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、HashJoin 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 HashJoin を実行した場合、HashJoin は完了できません。この場合、HashJoin 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 NestedLoopsJoin 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。

HashJoin 演算子で使用できるメモリ容量は、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。

HashJoin はメモリ不足状態の Windows Mobile では無効になります。

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

詳細については、「接続プロパティ」『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。

RecursiveHashJoin アルゴリズム (JHR)

RecursiveHashJoin は HashJoin の変形で、再帰的な UNION クエリに使用されます。詳細については、「HashJoin アルゴリズム (JH、JHSP、JHFO、JHAP、JHO、JHPO)」 625 ページと「再帰共通テーブル式」 469 ページを参照してください。

RecursiveLeftOuterHashJoin アルゴリズム (JHRO)

RecursiveLeftOuterHashJoin は HashJoin の変形で、特定の再帰的な UNION クエリに使用されます。詳細については、「HashJoin アルゴリズム (JH、JHSP、JHFO、JHAP、JHO、JHPO)」 625 ページと「再帰共通テーブル式」 469 ページを参照してください。

HashSemijoin アルゴリズム (JHS)

HashSemijoin は、左側と右側の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。HashSemijoin では、右側のローを読み込んでイン・メモリ・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。一致するローが見つかると左側のローが結果に出力され、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 HashSemijoin を考慮しません。NestedLoopsSemijoin 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、HashSemijoin 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、HashSemijoin の方が NestedLoopsSemijoin よりパフォーマンスに優れています。

HashJoin 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、HashSemijoin 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。

HashSemijoin 演算子で使用できるメモリ容量は、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 Mobile では使用できません。

詳細については、「接続プロパティ」『SQL Anywhere サーバ - データベース管理』の「QueryLowMemoryStrategy」を参照してください。

HashAntisemijoin アルゴリズム (JHA)

HashAntisemijoin 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。HashAntisemijoin では、右側のローを読み込んでイン・メモリ・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。HashAntisemijoin 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、HashAntisemijoin 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。

HashJoin 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、HashAntisemijoin 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。

HashAntisemijoin 演算子で使用できるメモリ容量は、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 Mobile では使用できません。

詳細については、「接続プロパティ」『SQL Anywhere サーバ - データベース管理』の「QueryLowMemoryStrategy」を参照してください。

MergeJoin アルゴリズム (JM、JMFO、JMO)

MergeJoin は 2 つの入力を読み込みます。このとき、2 つの入力はどちらもジョイン属性で順序付けされています。左側の入力のローごとに、右側の入力のローにソート順にアクセスすることで、一致する右側のローをすべて読み込みます。

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。

MergeJoin が HashJoin 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 HashJoin より MergeJoin を選択します。

NestedLoopsJoin アルゴリズム (JNL、JNLFO、JNLO)

NestedLoopsJoin では、左側のローごとに右側全体が読み込まれ、左側と右側のジョインが計算されます(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・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 NestedLoopsJoin が選択される可能性があります。

NestedLoopsJoin 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。ただし、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、NestedLoopsJoin の効率は、他のジョイン方式より低くなります。ただし、NestedLoopsJoin 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、NestedLoopsJoin だけです。つまり、ジョイン上の sensitive カーソルは、NestedLoopsJoin を使用した場合のみ実行できます。

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』

NestedLoopsSemijoin アルゴリズム (JNLS)

NestedLoopsJoin と同様に、NestedLoopsSemijoin は左側のローごとに右側をスキャンして、入力をジョインします。NestedLoopsJoin 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。

NestedLoopsSemijoin は 2 つの点で NestedLoopsJoin と異なります。第 1 に、NestedLoopsSemijoin 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、NestedLoopsSemijoin では、最初に一致するローが見つかった時点で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、NestedLoopsSemijoin を使用できます。

NestedLoopsAntisemijoin アルゴリズム (JNLA)

NestedLoopsJoin アルゴリズムと同様に、NestedLoopsAntisemijoin は左側のローごとに右側をスキャンして、入力をジョインします。NestedLoopsJoin 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。NestedLoopsAntisemijoin が NestedLoopsJoin と異なる点は、左側の値のみを出力することです。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。具体的には、右側に対応する値が存在しない左側の値だけが表示されます。

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、「HashDistinct アルゴリズム (DistH)」 629 ページと「OrderedDistinct アルゴリズム (DistO)」 630 ページを参照してください。

HashDistinct アルゴリズム (DistH)

HashDistinct は 1 つの入力を受け取り、すべての排他ローを返します。HashDistinct は、これを入力を読み込んだ時に行い、イン・メモリ・ハッシュ・テーブルを構築します。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。入力全体がイン・メモリ・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。

また、HashDistinct には次のような特性があります。

- 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する。
- ワーク・テーブルを使用するので、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る。
- 前に返されなかったローを見つけるとそのローを返す。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する。

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環境で HashDistinct を実行した場合、HashDistinct は完了できません。この場合、HashDistinct では中間結果が廃棄され、代わりに内部メモリ不足時方式が使用されます。

HashDistinct 演算子で使用できるメモリ容量は、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。

OrderedDistinct アルゴリズム (DistO)

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、OrderedDistinct 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。OrderedDistinct 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、OrderedDistinct 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、「HashGroupBy アルゴリズム (GrByH)」 630 ページ、「OrderedGroupBy アルゴリズム (GrByO)」 632 ページ、「SingleRowGroupBy アルゴリズム (GrByS)」 632 ページを参照してください。

HashGroupBy アルゴリズム (GrByH)

HashGroupBy では、グループごとに 1 つのローで構成されるイン・メモリ・ハッシュ・テーブルが構築されます。入力ローが読み込まれると、ワーク・テーブル内で関連グループが検索されます。集合関数が更新され、グループ・ローがワーク・テーブルに再び書き込まれます。グループ・レコードが見つからなければ、新しいグループ・レコードが初期化され、ワーク・テーブルに挿入されます。

HashGroupBy 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。

HashGroupBy 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 HashGroupBy 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、HashGroupBy 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 HashGroupBy からの中間結果を廃棄し、代わりに内部メモリ不足時方式を使用します。

HashGroupBy 演算子で使用できるメモリ容量は、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。

ClusteredHashGroupBy アルゴリズム (GrByHClust)

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る傾向があります。ClusteredHashGroupBy 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 ClusteredHashGroupBy 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。

ClusteredHashGroupBy 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、ClusteredHashGroupBy 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 ClusteredHashGroupBy 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、ClusteredHashGroupBy 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 負荷の詳細については、「OLAP のサポート」 481 ページを参照してください。

HashGroupBySets アルゴリズム (GrByHSets)

GROUPING SETS クエリを実行する場合、HashGroupBy の変形である HashGroupBySets が使用されます。

HashGroupBySets は並列実行できます。

詳細については、「HashGroupBy アルゴリズム (GrByH)」 630 ページを参照してください。

OrderedGroupBy アルゴリズム (GrByO)

OrderedGroupBy では、グループ化カラムによって順序付けされた入力が読み込まれます。各ローは、読み込まれるたびに前のローと比較されます。グループ化カラムが一致すると、現在のグループが更新されます。それ以外の場合は、現在のグループが出力され、新しいグループが開始されます。

OrderedGroupBySets アルゴリズム (GrByOSets)

GROUPING SETS クエリを実行する場合、OrderedGroupBy の変形である OrderedGroupBySets が使用されます。このアルゴリズムでは、入力がグループ化カラムでソートされている必要があります。「OrderedGroupBy アルゴリズム (GrByO)」 632 ページを参照してください。

SingleRowGroupBy アルゴリズム (GrByS)

GROUP BY を指定しなければ、単一ローの集合の生成に SingleRowGroupBy が使用されます。各入力ローに対して単一のグループ・ローがメモリに格納され、更新されます。

SortedGroupBySets アルゴリズム (GrBySSets)

SortedGroupBySets は、GROUPING SETS を含む OLAP クエリを処理する場合に使用されます。

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 (MergeExcept と HashExcept)
- INTERSECT アルゴリズム (MergeIntersect と HashIntersect)
- UNION アルゴリズム (UNION、UNION ALL、再帰 UNION)

Except アルゴリズム (EAH、EAM、EH、EM)

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 MergeExcept (EM) とハッシュベースの変形 HashExcept (EH) の 2 つの物理的な実装から選択します。

MergeExcept では、MergeJoin 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、HashExcept では、HashAntisemijoin 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)。

メモリ不足が検出されると、HashExcept 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。

HashExcept はメモリ不足状態の Windows Mobile では無効になります。

EXCEPT の場合、MergeExcept と HashExcept 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、HashExceptAll と MergeExceptAll は、結果内の重複するローの正しい数を計算する RowReplicate と組み合わせられます。

参照

- 「EXCEPT 句」『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」 407 ページ
- QueryLowMemoryStrategy 接続プロパティ：「接続プロパティ」『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・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」 237 ページ

Intersect アルゴリズム (IH、IM、IAH、IAM)

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 MergeIntersect (IM) とハッシュベースの変形 HashIntersect (IH) の 2 つの物理的な実装から選択します。

MergeIntersect では、MergeJoin 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、HashIntersect では、HashJoin 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必要に応じて、HashIntersect 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。

HashIntersect はメモリ不足状態の Windows Mobile では無効になります。

INTERSECT の場合、MergeIntersect または HashIntersect 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 の場合、MergeIntersectAll と HashIntersectAll は、結果内の重複するローの正しい数を計算する RowReplicate と組み合わせられます。

参照

- 「INTERSECT 句」 『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」 407 ページ
- QueryLowMemoryStrategy 接続プロパティ：「接続プロパティ」 『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・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」 237 ページ

RecursiveTable アルゴリズム (RT)

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。「共通テーブル式」 461 ページを参照してください。

RecursiveUnion アルゴリズム (RU)

RecursiveUnion は、再帰的な UNION クエリの実行中に使用されます。「再帰共通テーブル式」 469 ページを参照してください。

RowReplicate アルゴリズム (RR)

RowReplicate は、EXCEPT ALL や INTERSECT ALL などの集合操作の実行時に使用されます。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 は、結果セット内のロー数が正しいことを保証します。「UNION、INTERSECT、EXCEPT を使用した、クエリ結果に対する集合操作の実行」 407 ページを参照してください。

UnionAll アルゴリズム (UA)

UnionAll では、重複に関係なく各入力からローが読み込まれ、出力されます。このアルゴリズムは、UNION 句と UNION ALL 句を実装するために使用されます。UNION の場合は、UnionAll によって生成される重複を削除するための、HashDistinct や OrderedDistinct などの重複排除アルゴリズムが必要です。

「HashDistinct アルゴリズム (DistH)」 629 ページと「OrderedDistinct アルゴリズム (DistO)」 630 ページを参照してください。

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 (Sort)」 635 ページと「UnionAll アルゴリズム (UA)」 634 ページを参照してください。

ソート・アルゴリズム (Sort)

Sort 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 Sort を実行すると、Sort が完了できない場合があります。この場合、Sort 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、Sort 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。

Sort 演算子で使用できるメモリ容量は、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。Sort では、出力ローがワーク・テーブルに書き込まれませんが、Sort 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。

SortTopN アルゴリズム (SrtN)

SortTopN 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。

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見なします。

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 [Web サービス]」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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。

DecodePostings (DP)

テキスト・インデックスは、テーブル内の圧縮チャンクに格納されます。DecodePostings は、テキスト・インデックス内の単語の位置情報を復号化します。

「全文検索」 338 ページを参照してください。

DerivedTable アルゴリズム (DT)

派生テーブルは、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派生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、派生テーブルの名前と、計算されたカラムのリストが表示されます。

派生テーブルには、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。派生テーブルは、元の文で指定されている派生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。

派生テーブルの詳細については、「FROM 句：テーブルの指定」 317 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派生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 (Exchange)

Exchange 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。

Exchange 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・アルゴリズム (Filter、PreFilter)

フィルタでは、任意のタイプの述部、subselect を含む比較、EXISTS と NOT EXISTS の各サブクエリ (その他の形式の限定サブクエリ) などの探索条件が適用されます。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 Filter がない場合も多くあります。たとえば、IndexScan 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。たとえば、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 Filter 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。

PreFilter 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、Filter 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」 319 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 (HF、HFP)

ハッシュ・フィルタは、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・サーバは、(ハッシュの競合の有無に応じて) その値の一致するローがあるかどうかを判断できます。

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 にジョインします。データベース・サーバは、R のローをすべて読み込んでから、T のローを読み込みます。ハッシュ・フィルタがインデックス・スキャンによって返される R のローを使用して構築されている場合、データベース・サーバは、R とジョインできない T のローを直ちに拒否できます。このようにして、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。
- 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操作が行われ、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。

InList アルゴリズム (IN)

InList は、インデックスを使用して IN リスト述部を満たすことができる場合に使用されます。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 );
```

これを行うために、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・テーブルからフェッチされ、Employees テーブルの調査に使用されます。

InList を使用するには、IN リスト述部内の各要素が定数であるか、または最適化時に定数と評価される値 (CURRENT DATE、CURRENT TIMESTAMP、非決定的システム関数、ユーザ定義関数) や、クエリ・ブロック 1 回の実行中に定数である値 (外部参照) であることが必要です。たとえば、次のクエリでは InList を使用できます。

```
SELECT *, (
   SELECT FIRST GivenName
    FROM Employees e
    WHERE e.DepartmentID IN ( 500, d.DepartmentID )
    ORDER BY e.DepartmentID )
FROM Departments d;
```

同じインデックスを使用して、複数の IN リスト述部の条件を満たすことができます。

OpenString アルゴリズム (OpenString)

OpenString は、OPENSTRING 句を含む SELECT 文の FROM 句で使用されます。ローは、OPENSTRING 句で指定された BLOB またはファイルからフェッチされます。「FROM 句」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

OpenString 演算子は LOAD TABLE 文のプランにも表示されます。

ProcCall アルゴリズム (PC)

ProcCall は FROM 句内のプロシージャに使用され、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 が呼び出されるたびに、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』
- 「オプティマイザによるプロシージャ統計の使用方法」 592 ページ

RowConstructor アルゴリズム (ROWS)

RowConstructor は、他のアルゴリズムへの入力として使用できる仮想ローを作成する特殊な演算子です。RowConstructor は、次の 2 つ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 RowConstructor を使用するように変換され、SYS.DUMMY のテーブル・スキャンの要件に置き換わり、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、SYS.DUMMY のテーブル・スキャンを実行する代わりに RowConstructor が使用されても、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」 642 ページ

RowLimit アルゴリズム (RL)

RowLimit は、入力の最初の *n* 個のローを返し、残りのローは無視します。ローの制限は、SELECT 文の TOP n または FIRST 句によって設定されます。「SELECT 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Termbreaker アルゴリズム (TermBreak)

単語の区切りアルゴリズムは全文検索に使用されます。単語区切りの使用方法については、「テキスト設定オブジェクト」 339 ページを参照してください。

Window アルゴリズム (Window)

Window は、Window 関数を使用する OLAP クエリを評価する場合に使用します。「Window 関数」 494 ページを参照してください。

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ステップのセットです。最適化されたばかりかどうか、オプティマイザをバイパスしたかどうか、プランが以前の実行からキャッシュされたかどうかなどに関係なく、文の実行プランの保存と確認が可能です。クエリの実行プランは、元の文で使用される構文と正確に対応するとは限りません。クエリで明示的に指定したベース・テーブルの代わりにマテリアライズド・ビューを使用する場合があります。ただし、実行プランに記述された操作は、セマンティック上は元のクエリと同等です。

Interactive SQL 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。

- 短いテキスト・プラン
- 長いテキスト・プラン
- グラフィカルなプラン
- グラフィカルなプラン (ルート統計あり)
- グラフィカルなプラン (全統計あり)
- Ultra Light (短い、長い、またはグラフィカル)

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。

その他の解釈

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」 574 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、次の項を参照してください。

- 「セマンティック・クエリ変形」 577 ページ
- 「サブクエリを EXISTS 述部として書き換える」 587 ページ
- 「マテリアライズド・ビューによるパフォーマンスの向上」 603 ページ

オプティマイザがクエリの実装に使用するアルゴリズムと方式については、「クエリ実行アルゴリズム」 616 ページを参照してください。

グラフィカルなプランへのアクセス方法の詳細については、「グラフィカルなプランの表示」 653 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・プランの解釈」 643 ページと「グラフィカルなプランの解釈」 645 ページを参照してください。

テキスト・プランの解釈

クエリ実行プランのテキスト表示には、短いプランと長いプランの 2 種類があります。SQL 関数を使用して、テキスト・プランにアクセスできます。「短いテキスト・プランと長いテキスト・プランの表示」 644 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の解釈」 645 ページを参照してください。

短いテキスト・プラン

短いテキスト・プランは、プランを短時間で比較する場合に便利です。短いプランでは、すべてのプラン・フォーマットの最低限の情報が 1 行で表示されます。

次の例では、ORDER BY 句によって結果セット全体がソートされるため、プランは Work[Sort で始まります。Customers テーブルは、プライマリ・キー・インデックス CustomersKey によってアクセスされます。カラム Customers.ID がプライマリ・キーのため、インデックス・スキャンを使用して探索条件が満たされます。省略形 JNL は、Customers と SalesOrders の間のジョインを処理するためにオプティマイザでマージ・ジョインが選択されたことを示します。最後に、外部キー・インデックス FK_CustomerID_ID を使用して SalesOrders テーブルがアクセスされ、Customers テーブル内で CustomerID が 100 未満であるローが検索されます。

```
SELECT EXPLANATION ('SELECT GivenName, Surname, OrderDate
FROM Customers JOIN SalesOrders
WHERE CustomerID < 100
ORDER BY OrderDate');
```

```
Work[ Sort[ Customers<CustomersKey> JNL SalesOrders<FK_CustomerID_ID> ] ]
```

プランに使用されるコード・ワードの詳細については、「実行プランの省略形」 658 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 EXPLANATION ('SELECT *
FROM SalesOrders AS o
  KEY JOIN SalesOrderItems AS I
WHERE EXISTS
  (  SELECT *
     FROM Products p
     WHERE p.ID = 300 )');
```

```
o<seq> JNL i<FK_ID_ID> : p<ProductsKey>
```

各クエリ・ブロックのジョイン方式はコロンで区切られます。短いプランでは常に、メイン・ブロックのジョイン方式が先にリストされます。その後に、他のクエリ・ブロックのジョイン方式がリストされます。このような他のクエリ・ブロックのジョイン方式の順序は、文におけるクエリ・ブロックの順序やその実行順序とは一致しない場合があります。

プランに使用される省略形の詳細については、「実行プランの省略形」 658 ページを参照してください。

長いテキスト・プラン

長いテキスト・プランでは、短いテキスト・プランより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、長いテキスト・プランの最初の行は Plan[Total Cost Estimate: 6.46e-005] 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。Estimated Cache Pages は、文の処理に使用できる現在の予測キャッシュ・サイズです。

このプランは、結果がソートされ、ネスト・ループ・ジョインが使用されることを示します。ジョイン演算子と同じ行に、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。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 PLAN ('SELECT GivenName, Surname, OrderDate, Region, Country
FROM Customers JOIN SalesOrders ON ( SalesOrders.CustomerID = Customers.ID )
WHERE CustomerID < 100 AND ( Region LIKE "Eastern"
     OR Country LIKE "Canada" )
ORDER BY OrderDate');

( Plan [ Total Cost Estimate: 6.46e-005, Costed Best Plans: 1, Costed Plans: 10, Optimization Time:
0.0011462,
Estimated Cache Pages: 348 ]
  ( WorkTable
    ( Sort
      ( NestedLoopsJoin
        ( IndexScan Customers CustomersKey[ Customers.ID < 100 : 0.0001% Index | Bounded ] )
       ( IndexScan SalesOrders FK_CustomerID_ID[ Customers.ID = SalesOrders.CustomerID : 0.79365%
Statistics ]
          [  ( SalesOrders.CustomerID < 100 : 0.0001% Index | Bounded )
          AND  ( ( ((Customers.Country LIKE 'Canada' : 100% Computed)
          AND (Customers.Country = 'Canada' : 5% Guess))
          OR ((SalesOrders.Region LIKE 'Eastern' : 100% Computed)
          AND (SalesOrders.Region = 'Eastern' : 5% Guess)) ) : 100% Guess )  ] )
      )
    )
  )
)
```

プランに使用される省略形の詳細については、「実行プランの省略形」 658 ページを参照してください。

短いテキスト・プランと長いテキスト・プランの表示

♦ **短いテキスト・プランを表示するには、次の手順に従います (SQL の場合)。**

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. EXPLANATION 関数を実行します。「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。

♦ **長いテキスト・プランを表示するには、次の手順に従います (SQL の場合)。**

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. PLAN 関数を実行します。「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の解釈

Interactive SQL のグラフィカルなプラン機能を使用すると、クエリの実行プランを **[プラン・ビューワ]** ウィンドウに表示できます。実行プランは、関係代数演算子のツリーで構成されます。ツリーのルートで最終的な結果が得られるように、ツリーの最下位から開始して、クエリへの基本入力 (通常はテーブルのロー) を消費し、下から上にローを処理します。このツリーのノードは特定の代数演算子に対応しています。ただし、サーバで実行されるすべてのクエリ検査がノードで表示されるとは限りません。たとえば、サブクエリと関数のキャッシュの影響は、グラフィカルなプランには直接表示されません。

グラフィカルなプランでは、次のようにノードをさまざまな形で表示して、実行される操作の種類を示します。

- データを実体化する操作は六角形。
- インデックス・スキャンは台形。
- テーブル・スキャンは長方形。
- その他の操作は角丸四角形。

グラフィカルなプランを使用して、特定のクエリのパフォーマンス上の問題を診断できます。たとえば、プラン内の情報に基づいて、この特定のクエリのパフォーマンスを向上させるためにテーブルにインデックスが必要かを判断できます。**[プラン・ビューワ]** で **[保存]** ボタンを押すと、クエリのグラフィカルなプランを保存して後で参照できます。SQL Anywhere のグラフィカルなプランは、拡張子 *.saplan* で保存されます。

パフォーマンスに問題がある可能性がある場合は、グラフィカルなプラン内で太い線と赤い枠線で示されます。次に例を示します。

- 処理対象のロー数が増加するに従って、プラン内のノード間の線が太くなります。テーブル・スキャンに太い線が表示されている場合は、インデックスの作成が必要である可能性を示していることがあります。
- ノードの枠線が赤い場合は、実行プラン内の他の演算子に比べて、その演算子のコストが高かったことを示しています。

プランのノードの形とその他のグラフィカルなコンポーネントは、Interactive SQL 内でカスタマイズできます。「グラフィカルなプランの表示のカスタマイズ」 652 ページを参照してください。

グラフィカルなプラン、概要付きのグラフィカルなプラン、または統計情報付きのグラフィカルなプランのいずれかを表示できます。3 つのプランはすべて、プランの中で最も高コストとして推定された部分を表示できます。統計情報付きのグラフィカルなプランの生成は、クエリの実行時にデータベース・サーバがモニタしている実際のクエリ実行統計が表示されるため高コストです。統計情報付きのグラフィカルなプランでは、クエリ・オプティマイザがアクセス・プランの作成時に使用する推定を、実行中にモニタされた実際の統計と直接比較できます。ただし、オプティマイザはクエリのコストを正確に推定できないことが多いので、推定値と実際の統計値には違いがあると想定してください。

グラフィカルなプランを表示する方法については、「グラフィカルなプランの表示」 653 ページを参照してください。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」 191 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・プランの解釈」 643 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

グラフィカルなプランは、短いテキスト・プランや長いテキスト・プランよりも多くの情報を提供します。統計情報付きのグラフィカルなプランの生成は高コストですが、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。推定と実際の統計が大きく異なる場合は、情報不足のためにオプティマイザが正確にクエリのコストを推定できず、その結果非効率な実行プランになっていることがあります。

統計情報付きのグラフィカルなプランを生成するには、データベース・サーバで文を実行する必要があります。実行時間が長い文のグラフィカルなプランを生成すると、かなり時間がかかる場合があります。UPDATE、INSERT、または DELETE 文の場合は、文の読み込み専用の部分だけが実行されます。テーブルの変更は実行されません。ただし、文にユーザ定義関数が含まれる場合は、クエリの一部としてユーザ定義関数が実行されます。ユーザ定義関数に副次的影響 (たとえば、ローの変更、テーブルの作成、コンソールへのメッセージ送信など) がある場合、統計情報付きのグラフィカルなプランを取得するときにこれらの変更が実行されます。場合によっては、統計情報付きのグラフィカルなプランの取得後に ROLLBACK 文を実行して、これらの副次的影響を元に戻すことができます。「ROLLBACK 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

統計情報付きのグラフィカルなプランによるパフォーマンス分析

統計情報付きのグラフィカルなプランを使用して、データベースのパフォーマンス上の問題を識別できます。統計情報付きのグラフィカルなプランのフィールドの詳細については、「[ノード統計] フィールドの説明」 654 ページと「[オプティマイザ統計] フィールドの説明」 656 ページを参照してください。

クエリの実行上の問題の識別

クエリの実行に影響するデータベース・オプションやその他のグローバルな設定は、ルート演算子ノードについてのみ表示できます。

選択性のパフォーマンスの確認

述部の選択性 (条件式) は、条件を満たすローの割合のことです。推定される述部の選択性が提供する情報に基づいて、オプティマイザはコストの推定を行います。オプティマイザの正常な処理には、正確な選択性推定が不可欠です。たとえば、オプティマイザが述部の選択性が高い (5% の選択性など) と誤って推定しても、実際には述部の選択性がかなり低い (50% など) 場合は、パフォーマンスが低下することがあります。選択性推定は正確ではない場合がありますが、極端に大きな誤差は問題がある可能性を示しています。

クエリの重要な部分の選択性情報が不正確であると判断した場合、CREATE STATISTICS を使用してカラムの新しい統計値セットを生成できます。まれに、明示的な選択性推定を指定したい場合があります。ただし、この方法では、後で統計を更新するときに問題が発生する可能性があります。

クエリがバイパス・クエリだと判断された場合、選択性統計値は表示されません。バイパス・クエリの詳細については、「オプティマイザの仕組み」 590 ページと「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。

次のような場合に不適切な選択性の兆候が見られます。

- **RowsReturned の実際値と推定値**　**RowsReturned** は結果セット内のロー数です。**RowsReturned** 統計値は、ツリーの先頭にあるルート・ノードのテーブル内に表示されます。推定ロー・カウントが実際のロー・カウントと大きく異なる場合は、このノードまたはサブツリーに接続された述部の選択性が正しくない可能性があります。
- **述部選択性の実際値と推定値**　「**述部**」というサブヘッダを検索して、述部の選択性を確認します。述部情報の解釈については、「グラフィカル・プラン内の選択性の表示」 651 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、**[統計情報]** ウィンドウ枠の述部サブヘッダの下にリストされています。

  述部選択性推定のソースが **Guess** の場合、オプティマイザには述部のフィルタリングの特性の判断に使用できる情報がないため、ヒストグラムがないなどの問題を示す可能性があります。推定ソースが **Index** で、選択性推定が正しくない場合は、インデックスが偏っていることが問題である可能性があります。REORGANIZE TABLE 文を使用してインデックスの断片化を解除すると改善できる場合があります。「REORGANIZE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

キャッシュのパフォーマンスの確認

キャッシュの読み込み数 (**[CacheRead]** フィールド) とヒット数 (**[CacheHits]** フィールド) が同じである場合は、この SQL 文のために処理されるすべてのオブジェクトがキャッシュに保持されています。キャッシュの読み込み数がヒット数よりも多い場合は、テーブルまたはインデックス・ページがサーバのキャッシュに存在しないため、データベース・サーバがこれらをディスクから読み込んでいることを示します。これは、ハッシュ・ジョインなどの場合に予想されます。ネスト・ループ・ジョインなどの場合、キャッシュ・ヒット率が低いときは、クエリを効率的に実行するためのキャッシュ (バッファ・プール) の不足を示している可能性があります。この場合、サーバのキャッシュ・サイズを増やすと改善できることがあります。

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」 255 ページを参照してください。

無効なインデックスの識別

クエリ実行プランからは、インデックスがパフォーマンスの向上に役立っているかどうかが不明な場合がよくあります。SQL Anywhere で使用されているスキャンベースの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」 268 ページと「インデックス・コンサルタント」 199 ページを参照してください。

データの断片化の問題の識別

Runtime および **FirstRowRunTime** の実際値と推定値は、ルート・ノード統計値で提供されます。ノードに **RunTime** が存在する場合、**[サブツリー統計]** セクションにはこの値だけが表示されます。

RunTime の解釈は、表示される統計セクションによって異なります。**[ノード統計]** の場合、**RunTime** はこのノードだけを実行している間に、対応する演算子が費やした累積時間です。**[サブツリー統計]** の場合、**RunTime** はこのノードの直下にある演算子のサブツリー全体に対して費やされた合計実行時間を表します。したがって、ほとんどの演算子では **RunTime** と **FirstRowRunTime** は独立した測定値で、個別に分析する必要があります。

FirstRowRunTime は、このノードの中間結果の最初のローを作成するために要した時間です。

テーブル・スキャンまたはインデックス・スキャンで、ノードの **RunTime** が予想よりも大きい場合、REORGANIZE TABLE 文を実行するとパフォーマンスを向上できることがあります。sa_table_fragmentation() システム・プロシージャと sa_index_density() システム・プロシージャを使用して、テーブルまたはインデックスが断片化されているかどうかを判断できます。

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」 261 ページを参照してください。

プランに使用されるコード・ワードの詳細については、「実行プランの省略形」 658 ページを参照してください。

グラフィカルなプランのノードの詳細情報の表示

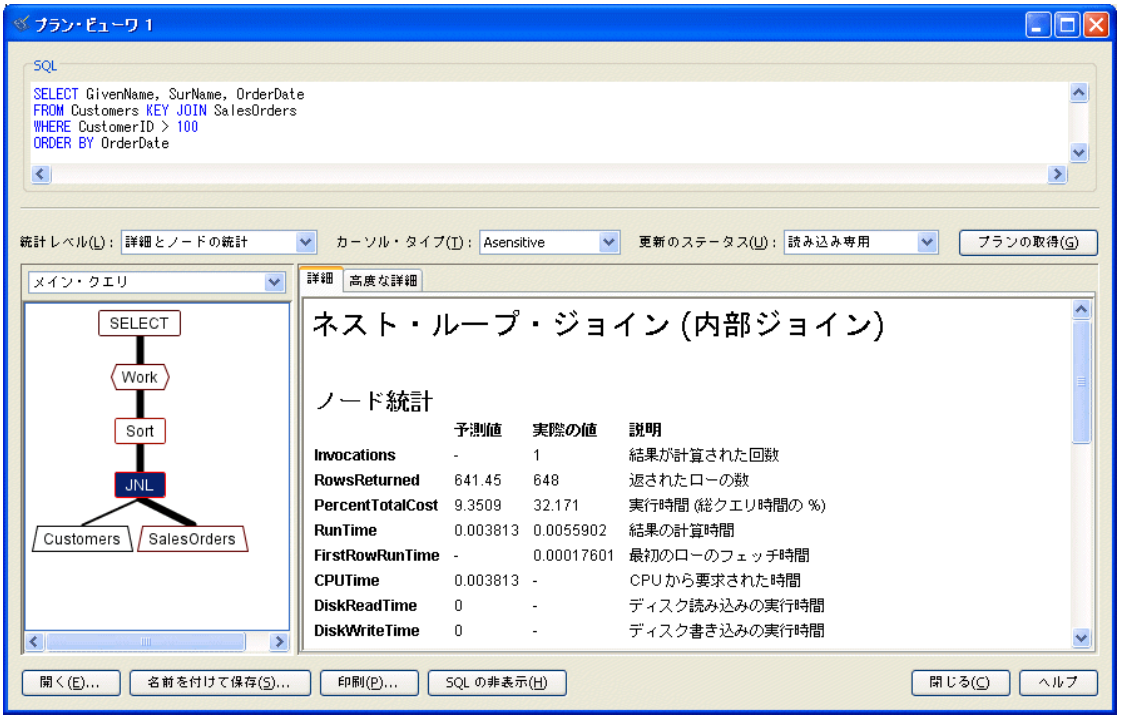
グラフィカルなプランでノードの詳細情報を表示するには、グラフィカルな図の左ウィンドウ枠でノードをクリックします。右側の **[詳細]** ウィンドウ枠と **[高度な詳細]** ウィンドウ枠にノードに関する詳細情報が表示されます。**[詳細]** ウィンドウ枠の次の 3 つのセクションにノードの統計値が表示されます。

- **[ノード統計]**
- **[サブツリー統計]**
- **[オプティマイザ統計]**

ノード統計は、特定のノードの実行に関する統計です。リーフ・ノードには **[詳細]** ウィンドウ枠があり、演算子に対する推定と実際の統計が表示されます。親ノードの右側にリーフ・ノードが表示されると、親演算子からローを複数回フェッチできます。たとえば、ネスト・ループ・ジョインを使用すると、リーフ・ノード (逐次、インデックス、または RowID スキャン・ノード) には実行ごとの (平均の) 統計と実行時の実際の累積統計の両方が含まれます。

ノードがリーフ・ノードではない場合、ノードは他のノードの中間結果を消費し、**[詳細]** ウィンドウ枠の **[サブツリー統計]** セクションには、このノードのサブツリー全体に対する推定と実際の累積統計が表示されます。SQL 要求全体を表すオプティマイザ統計情報は、ルート・ノードだけに存在します。オプティマイザ統計値は特に文の最適化に関連しており、最適化のゴール設定、最適化レベル設定、検討するプラン数などの値があります。

次の例では、 **nested loops join (JNL)** ノードが選択され、このノードに関連する情報だけが右ウィンドウ枠に表示されています。たとえば、**[述部]** の説明は TRUE となっています。これは、述部が適用されていないことを示しています。**Customers** 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%. に変わります。



[高度な詳細] ウィンドウ枠に表示される情報は、各演算子によって異なります。ルート・ノードの場合、**[高度な詳細]** ウィンドウ枠に、クエリが最適化されたときに有効になっていたすべての接続オプションの設定が表示されます。他の種類のノードでは、**[高度な詳細]** ウィンドウ枠に、特定のノードの処理で検討されたインデックスまたはマテリアライズド・ビューに関する情報が表示される場合があります。

グラフィカルなプランの各ノードのコンテキスト別ヘルプを表示するには、ノードを右クリックして **[ヘルプ]** を選択します。

プランに使用される省略形の詳細については、「実行プランの省略形」 658 ページを参照してください。

注意
クエリがバイパス・クエリとして認識されている場合、一部の最適化ステップがバイパスされ、**[クエリ・オプティマイザ]** セクションと **[述部]** セクションはどちらもグラフィカルなプランに表示されません。バイパス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」 590 ページを参照してください。

参照

- 「グラフィカルなプランの解釈」 645 ページ
- 「グラフィカルなプランの表示」 653 ページ
- 「実行プランの解釈」 642 ページ
- 「[ノード統計] フィールドの説明」 654 ページ
- 「[オプティマイザ統計] フィールドの説明」 656 ページ

グラフィカル・プラン内の選択性の表示

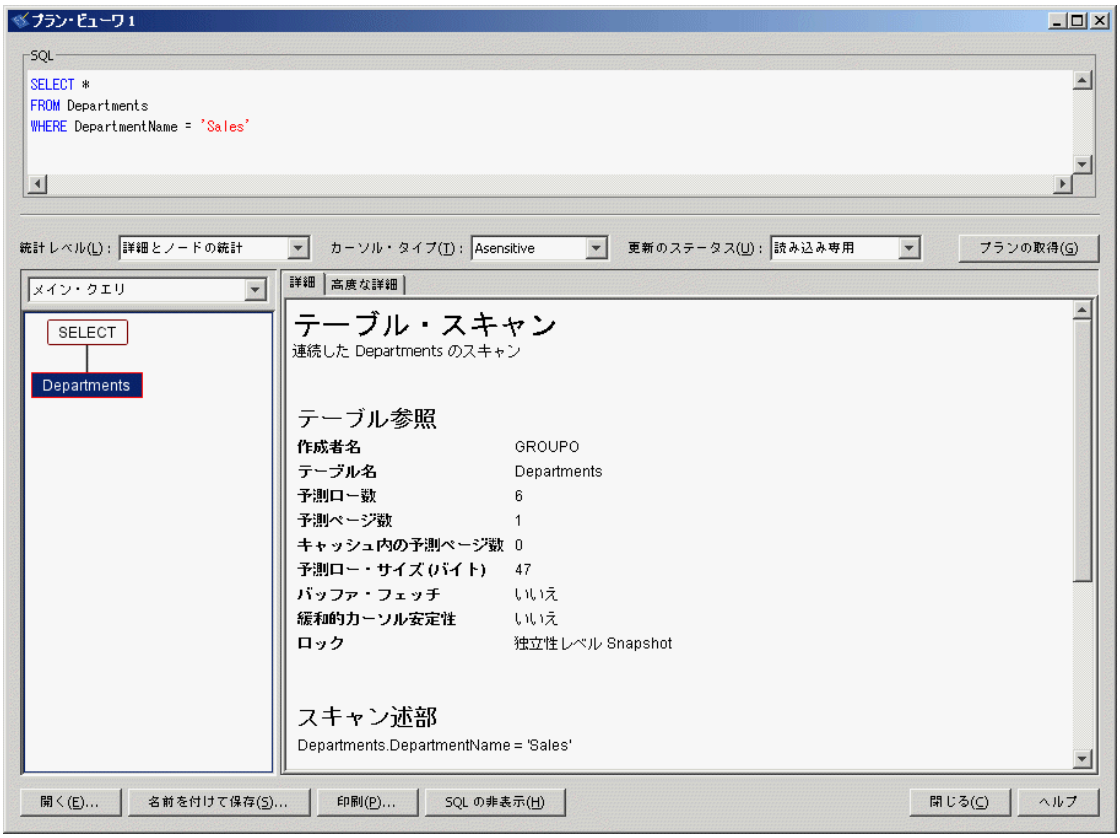
次の例では、選択されたノードが Departments テーブルのスキャンを表し、統計情報ウィンドウ枠には探索条件、選択性推定、実際の選択性として **[述部]** が表示されています。

[詳細] ウィンドウ枠の **[ノード統計]**、**[サブツリー統計]**、**[オプティマイザ統計]** の 3 つのセクションに、各ノードに関する統計値が表示されます。

ノード統計は、特定のノードの実行に関連しています。プラン内のノードがリーフ・ノードではなく、他のノードの中間結果を消費する場合、**[詳細]** ウィンドウ枠の **[サブツリー統計]** セクションには、対象ノードのサブツリー全体に対する推定と実際の累積統計が表示されます。オプティマイザ統計情報はルート・ノードだけに存在し、SQL 要求全体を表します。

バイパス・クエリには、選択性の情報が表示されない場合があります。バイパス・クエリの詳細については、「オプティマイザの仕組み」 590 ページを参照してください。

アクセス・プランは、データベース内で使用可能な統計値によって決まります。この統計値は、どのクエリが実行済みかによります。ここでは、各種の統計値やプランを確認できます。



この述部記述は次のとおりです。

```
Departments.DepartmentName = 'Sales' : 20% Column; true 1/5 20%
```

この述部は次のように解釈できます。

- Departments.DepartmentName = 'Sales' は述部です。
- 20% は、オプティマイザによる選択性推定です。つまり、オプティマイザは 20% のローが述部を満たすという推定に基づいてクエリ・アクセスを選択しています。

  これは、ESTIMATE 関数で得られる出力と同じです。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。
- Column は推定ソースです。これは、ESTIMATE_SOURCE 関数で得られる出力と同じです。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。

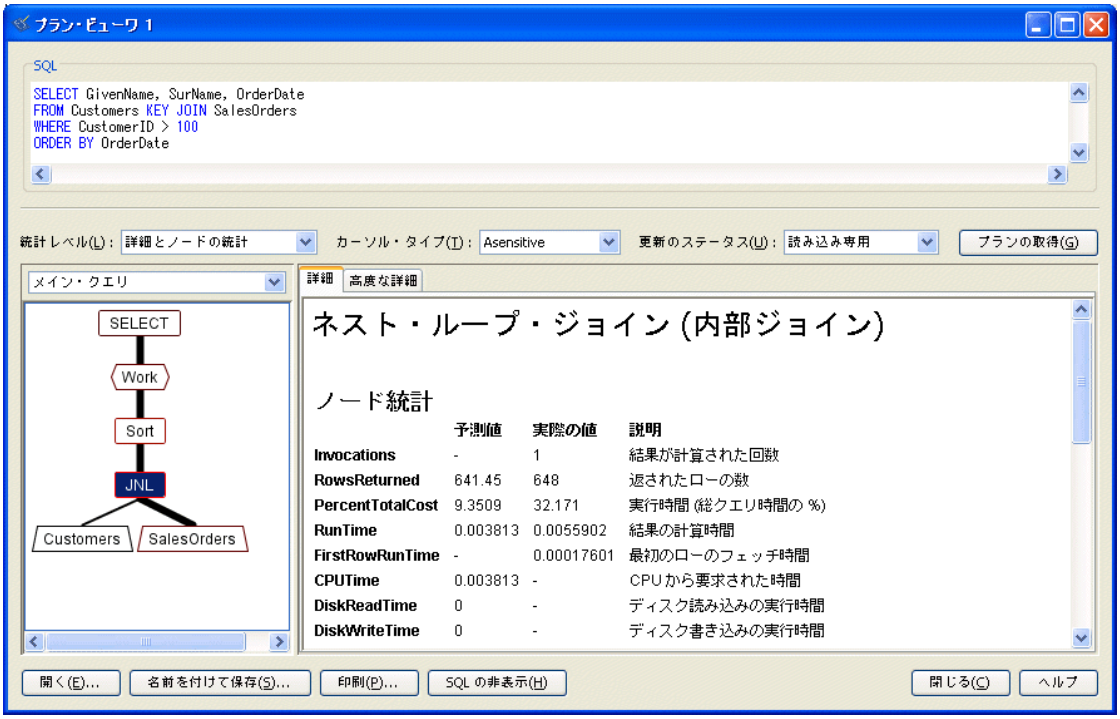
注意
[統計情報付きのグラフィカルなプラン] ではなく、[グラフィカルなプラン] を選択すると、最後の 2 つの統計情報は表示されません。

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の実行後に、プラン内の項目の表示を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 [プラン・ビューワ] の左下にある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。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。



グラフィカルなプランの表示

Interactive SQL または GRAPHICAL_PLAN 関数を使用して、グラフィカルなプランを表示できます。テキスト・プランにアクセスする方法については、「テキスト・プランの解釈」 643 ページを参照してください。

グラフィカルなプランの表示

♦ **グラフィカルなプランを表示するには、次の手順に従います (Interactive SQL の場合)。**

1. Interactive SQL を起動して、SQL Anywhere データベースに接続します。
2. **[SQL 文]** ウィンドウ枠に文を入力します。
3. **[ツール] - [プラン・ビューワを開く]** を選択します。
4. [統計レベル]、[カーソル・タイプ]、[更新のステータス] を選択して、**[プランの取得]** をクリックします。

♦ **グラフィカルなプランを表示するには、次の手順に従います (SQL の場合)。**

GRAPHICAL_PLAN 関数を使用して、グラフィカルなプランを XML フォーマットの文字列で表示できます。

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. GRAPHICAL_PLAN 関数を実行します。「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。

参照：

- 「実行プランの省略形」 658 ページ

[ノード統計] フィールドの説明

次に、グラフィカルなプランの **[ノード統計]** セクションに表示されるフィールドについて説明します。

フィールド	説明
CacheHits	この演算子によるキャッシュ読み込み要求で、バッファ・プールで条件が満たされ、ディスク読み込み操作が不要になった要求の合計数。
CacheRead	この演算子がデータベース・ファイルのページ (通常、テーブル・ページやインデックス・ページなど) を読み込もうとした合計回数。
CPUTime	このノードが表す処理アルゴリズムによって発生した CPU 時間。
DiskRead	このノードの処理の結果として、ディスクから読み込まれた累積ページ数。
DiskReadTime	このノードの処理に必要なデータベース・ページをディスクから読み込むために要した累積時間。
DiskWrite	このノードの処理の結果として、ディスクに書き込まれた累積ページ数。
DiskWriteTime	このノードのアルゴリズムの処理に必要なデータベース・ページをディスクに書き込むために要した累積時間。
FirstRowRunTime	**FirstRowRunTime** 値は、このノードの中間結果の最初のローを作成するために要した時間です。
Invocations	ノードが結果を計算するために呼び出された回数。親ノードに結果を返します。ほとんどのノードは 1 回だけ呼び出されます。ただし、スキャン・ノードの親がネスト・ループ・ジョインで、ノードが複数回実行される可能性がある場合、各呼び出し後に異なるロー・セットを返すことがあります。
PercentTotalCost	特定のノード内で結果の計算に費やす **RunTime**。文の合計 RunTime に対する割合で表します。

フィールド	説明
QueryMemMaxUseful	この特定の演算子での使用が予想されるクエリ・メモリの推定容量。**Actual** 統計値で報告されるクエリ・メモリの実際の使用量と大幅に異なる場合、クエリ・オプティマイザによる結果セットのサイズの推定に潜在的な問題がある可能性があります。この推定のエラーの原因は、述部選択性推定が不正確であるか存在しないことが考えられます。
RowsReturned	要求の処理の結果として親ノードに返されたローの数。**RowsReturned** は、このノードで表されるオブジェクト (派生オブジェクトの場合がある) 内のローの数と同一であることがよくありますが、必ず同じとは限りません。ベース・テーブル・スキャンを表すリーフ・ノードを考慮してください。**RowsReturned** 値がテーブル内のローの数よりも小さい、または大きい場合があります。最終結果の計算で、親ノードがテーブルのすべてのローを要求できない場合、**RowsReturned** 値は小さくなります。GROUP BY GROUPING SETS クエリなど、親のハッシュ GROUP BY GROUPING SETS ノードが異なるグループを計算するために入力の受け渡しを複数回要求する場合、**RowsReturned** は大きくなります。 返された推定ローと返された実際の数とが大幅に食い違っている場合、オプティマイザが不正確な選択性情報に基づいて操作していることを示している可能性があります。
RunTime	実際の時間の計測値。入出力の待機、ローのロック、テーブルのロック、 内部サーバの同時制御メカニズム、実際の処理実行時間が含まれます。**RunTime** の解釈は、表示される統計セクションによって異なります。[ノード統計] の場合、**RunTime** はこのノードだけを実行している間に、ノードの対応する演算子が費やした累積時間です。[ノード統計] セクションには、この統計の推定値と実際値の両方が表示されます。 テーブル・スキャンまたはインデックス・スキャンで、ノードの **RunTime** が予想よりも大きい場合、さらに分析を進めると問題の特定に役立つ場合があります。クエリで共有リソースに対する競合が発生し、その結果ブロックした可能性があります。sa_locks() システム・プロシージャを使用して、ブロックされた接続をモニタできます。別の例として、ディスクのデータベース・ページ・レイアウトが最適化されていない場合や、テーブルが内部ページの断片化による影響を受けている場合があります。REORGANIZE TABLE 文を実行するとパフォーマンスを向上できることがあります。sa_table_fragmentation() システム・プロシージャと sa_index_density() システム・プロシージャを使用して、テーブルまたはインデックスが断片化されているかどうかを判断できます。

[オプティマイザ統計] フィールドの説明

次に、グラフィカルなプランの **[オプティマイザ統計]** セクションに表示されるフィールドについて説明します。**[オプティマイザ統計]** には、データベース・サーバの状態および選択された文の最適化に関する情報が表示されます。

フィールド	説明
最適化方法	実行方式の選択に使用されたアルゴリズム。戻り値は次のとおりです。 ● バイパス (クエリ書き換え) ● バイパス (簡易クエリ書き換え) ● バイパス (ヒューリスティック) ● バイパス後、最適化 ● 最適化 ● 再利用 ● 再利用 (単純)
見積り済み最良プラン	クエリ・オプティマイザが異なるクエリ実行方式を列挙するとき、現在の方式の前に検出された最良な方式よりも推定コストが安い方式が検出された回数を追跡します。特定のクエリに対してこの発生回数を予測することは困難ですが、この数値が小さい場合、オプティマイザのアルゴリズムによって検索領域が大幅に削減され、通常は、最適化時間が短縮されることを示します。オプティマイザは特定の文のクエリ・ブロックごとに 1 回以上の列挙プロセスを起動するため、 **[見積り済み最良プラン]** は累積回数を表します。「オプティマイザの仕組み」 590 ページを参照してください。 **[見積り済み最良プラン]**、**[見積り済みプラン]**、**[最適化時間]** の値が 0 の場合、SQL Anywhere オプティマイザによって文は最適化されていません。データベース・サーバはこの文をバイパスし、文の最適化を行わずに実行プランを生成したか、または文のプランはキャッシュされました。「クエリ処理のフェーズ」 574 ページを参照してください。
見積り済みプラン	コストを部分的または完全に推定したこの要求に対して、オプティマイザが検討した異なるアクセス・プランの数。**[見積り済み最良プラン]** と同様、この値が小さい場合は通常、最適化時間の短縮を示し、この値が大きい場合は SQL クエリがより複雑であることを示します。 **[見積り済み最良プラン]**、**[見積り済みプラン]**、**[最適化時間]** の値が 0 の場合、文は最適化されていません。データベース・サーバは文をバイパスし、文の最適化を行わず実行プランを生成しています。「クエリ処理のフェーズ」 574 ページを参照してください。

フィールド	説明
最適化時間	文の最適化に要した時間。 **[見積り済み最良プラン]**、**[見積り済みプラン]**、**[最適化時間]** の値が 0 の場合、文は最適化されていません。データベース・サーバは文をバイパスし、文の最適化を行わず実行プランを生成しています。「クエリ処理のフェーズ」 574 ページを参照してください。
予測キャッシュ・ページ数	文の処理に使用できる現在の推定キャッシュ・サイズ。 非効率なアクセス・プランを削減するため、オプティマイザは現在のキャッシュ・サイズの半分は選択された文の処理に使用できると想定します。
CurrentCacheSize	最適化時のデータベース・サーバのキャッシュ・サイズ (キロバイト単位)。
QueryMemMaxUseful	この要求に使用できるクエリ・メモリのページ数。この数値が 0 の場合、文の実行プランにはメモリを大量に消費する演算子がなく、サーバのメモリ・ガバナーの制御対象ではありません。「メモリ・ガバナー」 596 ページを参照してください。
QueryMemNeedsGrant	この要求の実行方式に存在し、メモリを大量に消費する 1 つ以上のクエリ実行演算子に対して、メモリ・ガバナーがメモリを付与する必要があるかどうかを示します。「メモリ・ガバナー」 596 ページを参照してください。
QueryMemLikelyGrant	この文がすぐに実行される場合、この文に付与されるクエリ・メモリ・プールの推定ページ数。この推定値は、プラン内のメモリを大量に消費する演算子の数、データベース・サーバのマルチプログラミング・レベル、メモリを大量に消費する同時実行の要求数によって異なります。「メモリ・ガバナー」 596 ページを参照してください。
QueryMemPages	すべての接続に対して、メモリを大量に消費するクエリ実行アルゴリズムで使用できるクエリ・メモリ・プール内のメモリの合計容量を表すページ数。「メモリ・ガバナー」 596 ページを参照してください。
QueryMemActiveMax	特定の時点でクエリ・メモリをアクティブに使用できるタスクの最大数。「メモリ・ガバナー」 596 ページを参照してください。
QueryMemActiveEst	安定した状態のデータベース・サーバで、クエリ・メモリをアクティブに使用するタスクの推定平均数。「メモリ・ガバナー」 596 ページを参照してください。

フィールド	説明
isolation_level	文の独立性レベル。文の独立性レベルは、同じトランザクション内の他の文と異なる場合があります。また、特定のベース・テーブルに対して FROM 句のヒントを使用して優先される可能性があります。「isolation_level オプション [データベース] [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_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_level	クエリ・オプティマイザがアクセス・プランの検索に費やす作業量を制御します。「optimization_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。
optimization_workload	optimization_workload 設定値 (**Mixed** または **OLAP**)。「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。
max_query_tasks	単一のクエリの並列実行プランで使用される可能性があるタスクの最大数。「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』を参照してください。
user_estimates	クエリ・テキストの各述部で指定されたユーザ推定を尊重するか無視するかを制御します。「user_estimates 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。

実行プランの省略形

次に、実行プランに表示される省略形を示します。

短いテキスト・プラン	長いテキスト・プラン	その他の情報
	見積り済み最良プラン	オプティマイザは、特定のクエリのアクセス・プランを生成してコストを計算します。この処理の最中、現在の最良プランが、より低いコストが推定される新しい最良プランで置き換えられる場合があります。最後の最良プランが、文の実行に使用する実行プランになります。見積り済み最良プランは、現在の最良プランよりも優れたプランをオプティマイザが検出した回数を示しています。数字が小さい場合は、最良プランが列挙プロセスの早い段階で決定されたことを示しています。オプティマイザは特定の文のクエリ・ブロックごとに 1 回以上の列挙プロセスを起動するため、見積り済み最良プランは累積回数を表します。「オプティマイザの仕組み」 590 ページを参照してください。
	見積り済みプラン	オプティマイザによって生成される多数のプランが、それまでに検出された最良プランと比較してコストが高いと判断されます。見積り済みプランは、指定された文の列挙プロセスの最中にオプティマイザが検討した一部のプランまたは完全なプランの数を表します。
DELETE	削除	削除操作のルート・ノード。「DELETE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
DistH	HashDistinct	「HashDistinct アルゴリズム (DistH)」 629 ページを参照してください。
DistO	OrderedDistinct	「OrderedDistinct アルゴリズム (DistO)」 630 ページを参照してください。
DP	DecodePostings	「DecodePostings (DP)」 637 ページを参照してください。
DT	DerivedTable	「DerivedTable アルゴリズム (DT)」 637 ページを参照してください。
EAH	HashExceptAll	「Except アルゴリズム (EAH、EAM、EH、EM)」 632 ページを参照してください。
EAM	MergeExceptAll	「Except アルゴリズム (EAH、EAM、EH、EM)」 632 ページを参照してください。

短いテキスト・プラン	長いテキスト・プラン	その他の情報
EH	HashExcept	「Except アルゴリズム (EAH、EAM、EH、EM)」 632 ページを参照してください。
EM	MergeAccept	「Except アルゴリズム (EAH、EAM、EH、EM)」 632 ページを参照してください。
Exchange	Exchange	「交換アルゴリズム (Exchange)」 637 ページを参照してください。
フィルタ	フィルタ	「フィルタ・アルゴリズム (Filter、PreFilter)」 638 ページを参照してください。
GrByH	HashGroupBy	「HashGroupBy アルゴリズム (GrByH)」 630 ページを参照してください。
GrByHClust	HashGroupByClustered	「ClusteredHashGroupBy アルゴリズム (GrByHClust)」 631 ページを参照してください。
GrByHSets	HashGroupBySets	「HashGroupBySets アルゴリズム (GrByHSets)」 631 ページを参照してください。
GrByO	OrderedGroupBy	「OrderedGroupBy アルゴリズム (GrByO)」 632 ページを参照してください。
GrByOSets	OrderedGroupBySets	「OrderedGroupBySets アルゴリズム (GrByOSets)」 632 ページを参照してください。
GrByS	SingleRowGroupBy	「SingleRowGroupBy アルゴリズム (GrByS)」 632 ページを参照してください。
GrBySSets	SortedGroupBySets	「SortedGroupBySets アルゴリズム (GrBySSets)」 632 ページを参照してください。
HF	HashFilter	「ハッシュ・フィルタ・アルゴリズム (HF、HFP)」 638 ページを参照してください。
HFP	ParallelHashFilter	「ハッシュ・フィルタ・アルゴリズム (HF、HFP)」 638 ページを参照してください。
HTS	HashTableScan	「HashTableScan 方式 (HTS)」 623 ページを参照してください。
IAH	HashIntersectAll	「Intersect アルゴリズム (IH、IM、IAH、IAM)」 633 ページを参照してください。

短いテキスト・プラン	長いテキスト・プラン	その他の情報
IAM	MergeIntersectAll	「Intersect アルゴリズム (IH、IM、IAH、IAM)」 633 ページを参照してください。
IH	HashIntersect	「Intersect アルゴリズム (IH、IM、IAH、IAM)」 633 ページを参照してください。
IM	MergeIntersect	「Intersect アルゴリズム (IH、IM、IAH、IAM)」 633 ページを参照してください。
IN	InList	「InList アルゴリズム (IN)」 639 ページを参照してください。
table-name<index-name>	IndexScan、ParallelIndexScan	グラフィカルなプランで、インデックス・スキャンの場合は台形の中にインデックス名が表示されます。「IndexScan メソッド」 620 ページを参照してください。
INSENSITIVE	大文字小文字の区別	「Intersect アルゴリズム (IH、IM、IAH、IAM)」 633 ページを参照してください。
INSERT	挿入	挿入操作のルート・ノード。「INSERT 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
IO	IndexOnlyScan、ParallelIndexOnlyScan	「IndexOnlyScan 方式 (IO)」 620 ページと「ParallelIndexScan 方式」 621 ページを参照してください。
JH	HashJoin	「HashJoin アルゴリズム (JH、JHSP、JHFO、JHAP、JHO、JHPO)」 625 ページを参照してください。
JHS	HashSemijoin	「HashSemijoin アルゴリズム (JHS)」 626 ページを参照してください。
JHSP	ParallelHashSemijoin	「HashJoin アルゴリズム (JH、JHSP、JHFO、JHAP、JHO、JHPO)」 625 ページを参照してください。
JHFO	Full Outer HashJoin	「HashJoin アルゴリズム (JH、JHSP、JHFO、JHAP、JHO、JHPO)」 625 ページを参照してください。
JHA	HashAntisemijoin	「HashAntisemijoin アルゴリズム (JHA)」 627 ページを参照してください。

短いテキスト・プラン	長いテキスト・プラン	その他の情報
JHAP	ParallelHashAntisemijoin	「HashJoin アルゴリズム (JH、JHSP、JHFO、JHAP、JHO、JHPO)」 625 ページを参照してください。
JHO	Left Outer HashJoin	「HashJoin アルゴリズム (JH、JHSP、JHFO、JHAP、JHO、JHPO)」 625 ページを参照してください。
JHP	ParallelHashJoin	「HashJoin アルゴリズム (JH、JHSP、JHFO、JHAP、JHO、JHPO)」 625 ページを参照してください。
JHPO	ParallelLeftOuterHashJoin	「HashJoin アルゴリズム (JH、JHSP、JHFO、JHAP、JHO、JHPO)」 625 ページを参照してください。
JHR	RecursiveHashJoin	「RecursiveHashJoin アルゴリズム (JHR)」 626 ページを参照してください。
JHRO	RecursiveLeftOuterHashJoin	「RecursiveLeftOuterHashJoin アルゴリズム (JHRO)」 626 ページを参照してください。
JM	MergeJoin	「MergeJoin アルゴリズム (JM、JMFO、JMO)」 628 ページを参照してください。
JMFO	Full Outer MergeJoin	「MergeJoin アルゴリズム (JM、JMFO、JMO)」 628 ページを参照してください。
JMO	Left Outer MergeJoin	「MergeJoin アルゴリズム (JM、JMFO、JMO)」 628 ページを参照してください。
JNL	NestedLoopsJoin	「NestedLoopsJoin アルゴリズム (JNL、JNLFO、JNLO)」 628 ページを参照してください。
JNLA	NestedLoopsAntisemijoin	「NestedLoopsAntisemijoin アルゴリズム (JNLA)」 629 ページを参照してください。
JNLFO	Full Outer NestedLoopsJoin	「NestedLoopsJoin アルゴリズム (JNL、JNLFO、JNLO)」 628 ページを参照してください。
JNLO	Left Outer NestedLoopsJoin	「NestedLoopsJoin アルゴリズム (JNL、JNLFO、JNLO)」 628 ページを参照してください。
JNLS	NestedLoopsSemijoin	「NestedLoopsSemijoin アルゴリズム (JNLS)」 628 ページを参照してください。

短いテキスト・プラン	長いテキスト・プラン	その他の情報
KEYSET	キーセット	キーセット駆動型カーソルを指定します。「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。
LOAD	ロード	ロード操作のルート・ノード。「LOAD TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
MultiIdx	MultipleIndexScan	「MultipleIndexScan 方式 (MultIdx)」 621 ページを参照してください。
OpenString	OpenString	「OpenString アルゴリズム (OpenString)」 639 ページを参照してください。
	最適化時間	オプティマイザが指定された文の列挙プロセスに費やした合計時間。
PC	ProcCall	プロシージャ・コール (テーブル関数)。「ProcCall アルゴリズム (PC)」 640 ページを参照してください。
PreFilter	PreFilter	「フィルタ・アルゴリズム (Filter、PreFilter)」 638 ページを参照してください。
RL	RowLimit	「RowLimit アルゴリズム (RL)」 641 ページを参照してください。
ROWID	RowIdScan	グラフィカルなプランで、ロー ID スキャンの場合は長方形の中にテーブル名が表示されます。「RowIdScan 方式 (ROWID)」 623 ページを参照してください。
ROWS	RowConstructor	「RowConstructor アルゴリズム (ROWS)」 640 ページを参照してください。
RR	RowReplicate	「RowReplicate アルゴリズム (RR)」 634 ページを参照してください。
RT	RecursiveTable	「RecursiveTable アルゴリズム (RT)」 634 ページを参照してください。
RU	RecursiveUnion	「RecursiveUnion アルゴリズム (RU)」 634 ページを参照してください。

短いテキスト・プラン	長いテキスト・プラン	その他の情報
SELECT	選択	選択操作のルート・ノード。「SELECT 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
seq	TableScan、ParallelTableScan	グラフィカルなプランで、テーブル・スキャンの場合は長方形の中にテーブル名が表示されます。「TableScan 方式 (seq)」 622 ページと「ParallelTableScan 方式」 622 ページを参照してください。
Sort	Sort	インデックス・ソートまたはマージ・ソート。「ソート・アルゴリズム (Sort)」 635 ページを参照してください。
SrtN	SortTopN	「SortTopN アルゴリズム (SrtN)」 635 ページを参照してください。
TermBreak	TermBreak	全文検索の単語の区切りアルゴリズム。「テキスト・インデックスの変更」 358 ページを参照してください。
UA	UnionAll	「UnionAll アルゴリズム (UA)」 634 ページを参照してください。
UPDATE	Update	更新操作のルート・ノード。「UPDATE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
Window	Window	「Window アルゴリズム (Window)」 641 ページを参照してください。
Work	ワーク・テーブル	中間結果を表す内部ノード。

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。
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。

統計情報	説明
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。
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。

統計情報	説明
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。

項目	説明
Materialized views	オプティマイザによって検討されるマテリアライズド・ビューのリスト。リスト内の各エントリは view-name [view-matching-outcome] [table-list] という形式の組です。ここで *view-matching-outcome* はマテリアライズド・ビューの使用法を示します。この値が COSTED の場合、ビューは列挙時に使用されています。*table-list* は、このビューで置き換えられる可能性のあったクエリ・テーブルのリストです。 *view-matching-outcome* の値は、次のとおりです。 ● ベース・テーブルが一致しません ● 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マテリアライズド・ビューの使用を妨げる制約と条件の詳細については、「マテリアライズド・ビューによるパフォーマンスの向上」 603 ページと「マテリアライズド・ビューの制限」 57 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。

項目	説明
Estimated pages	テーブルの推定ページ数。
Estimated row size	テーブルの推定ロー・サイズ。
Page maps	複数ページの読み込みにページ・マップが使用される場合は YES。

インデックス・スキャンに関連するプランの項目

項目	説明
Selectivity	範囲バウンドと一致する推定ロー数。
Index name	インデックスの名前。
Key type	PRIMARY KEY、FOREIGN KEY、CONSTRAINT (一意性制約)、UNIQUE (ユニーク・インデックス) のいずれか。インデックスがユニークでないセカンダリ・インデックスの場合、キー・タイプは表示されません。
Depth	インデックスの高さ。「テーブルとページのサイズ」 672 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。
Sequential Transitions	インデックスがどれだけクラスタ化されているかを示す、各物理インデックスの統計情報。
Random Transitions	インデックスがどれだけクラスタ化されているかを示す、各物理インデックスの統計情報。
Key Values	インデックス内のユニークなエントリの数。
Cardinality	推定ロー数と異なる場合の、インデックスのカーディナリティ。バージョン 6.0.0 以前の SQL Anywhere データベースにのみ適用されます。
Direction	FORWARD または BACKWARD。
Range bounds	範囲バウンドは、リスト (col_name=value) または col_name IN [low, high] として表示されます。
Primary Key Table	外部キー・インデックス・スキャンのプライマリ・キー・テーブル名。

項目	説明
Primary Key Table Estimated Rows	外部キー・インデックス・スキャンのプライマリ・キー・テーブル内のロー数。
Primary Key Column	外部キー・インデックス・スキャンのプライマリ・キー・カラム名。

ジョイン、フィルタ、事前フィルタに関連するプランの項目

項目	説明
Predicate	このノードで評価される探索条件、選択性推定、測定値。「グラフィカル・プラン内の選択性の表示」 651 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。
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。

クエリ・パフォーマンスの向上

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。

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。

いくつかのローを削除し、空き領域を使用できる程度の小さなローを新しく挿入しなかった場合、時間の経過とともにデータベース内の情報がまばらになることがあります。その場合は、テーブルを再ロードするか、REORGANIZE TABLE 文を使用してテーブルの断片化を解除します。

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」と呼ばれるこの方法で、ディスクへの I/O 操作の合計数が減り、パ

フォーマンスが向上します。ユーザは、ビットマップの作成や使用に関するデータベース・サーバの条件を制御できません。

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 Mobile デバイスで SQL Anywhere を使用する場合に重要です。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・サイズを選択することが重要です。

参照

- 「適切なページ・サイズの使用」 265 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。

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」 199 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」 676 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、ISYSPHYSIDX に記録されず、使用後に廃棄されます。また、テンポラリ・テーブルの物理インデックスは共有されません。

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、すべてのインデックスを削除してから再作成する方法もあります。

物理インデックスが共有されているテーブルの特定

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

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

このクエリの結果セットの例を次に示します。

table_name	table_id	phys_index_id	COUNT(*)
ISYSCHECK	57	0	2
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。

参照

- 「インデックスの再構築」 80 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、単純な計算式では判断できません。インデックス検索の利点と、そのインデックスの管理に伴うオーバヘッドのトレードオフを考慮してください。次の要素を考慮して、インデックスの作成が必要かどうかを判断できます。

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

  詳細については、「複合インデックス」 678 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」 199 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、「インデックスの操作」 75 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 Mobile では使用できません。

さらに、完全比較の数は、統計情報付きのグラフィカルなプランの形式で提供されます。詳細については、「プランに使用される一般的な統計」 664 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・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、ツリーのようにいくつかのレベルで編成されています。インデックスの最初のページはルート・ページと呼ばれ、その次の下位レベルの 1 つ以上のページへと分岐して、さらにそれが分岐して、インデックスの最下位レベルに達します。最下位レベルのインデックス・ページは、リーフ・ページと呼ばれます。*n* レベルを持つインデックスの場合、特定のローを探すにはインデックス・ページを *n* 回読み込む必要があり、実際のローを含むデータ・ページを 1 回読み込む必要があります。通常、使用頻度の高いインデックス・ページはキャッシュに格納される傾向があるため、ディスクからの読み込みは *n* 回よりも少なくて済みます。

「インデックス・ファンアウト」は、1 ページに格納されるインデックス・エントリの数です。ファンアウトが大きなインデックスは、ファンアウトが小さなインデックスよりレベル数が少なくなります。したがって、通常はインデックス・ファンアウトが大きいほど、インデックスのパフォーマンスが向上します。データベースに適切なページ・サイズを選択すると、インデックス・ファンアウトを向上できます。「テーブルとページのサイズ」 672 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;
```

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」 143 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。

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。

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、ファンアウトを向上させるために、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。

参照

- 「クラスタード・インデックスの使用」 77 ページ
- 「ALTER INDEX 文」 『SQL Anywhere サーバ - SQL リファレンス』

SQL のダイアレクトと互換性

この項では、Transact-SQL の互換性と、他の SQL ソフトウェアにはない SQL Anywhere の機能について説明します。

SQL ダイアレクト

目次

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。

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も確認できます。また、SQL FLAGGER は、SQL Anywhere で Ultra Light アプリケーションのプロトタイプを作成するときに、使用している SQL が Ultra Light でサポートされていることを確認するためにも使用できます。

SQL FLAGGER では、SQL 文の構文とセマンティックの両要素が分析の対象ですが、準拠はコンパイル時に静的にチェックされます。構文の準拠のテスト例として、INSERT 文にオプションの INTO キーワードがない場合を考えます (たとえば、INSERT Products VALUES(...))。これは、SQL Anywhere による SQL 言語の文法上の拡張です。ANSI SQL/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。

キーのジョインもベンダ拡張として通知されます。ON 句を使用しないで、キーワード JOIN を使用する場合、キー・ジョインがデフォルトで使用されます。キー・ジョインは、既存の外部キー関係を使用して、テーブルをジョインします。キー・ジョインは Ultra Light ではサポートされていません。たとえば、次のクエリは、Products テーブルと SalesOrderItems テーブル間の暗黙的なジョイン条件を指定しています。このクエリは、SQL FLAGGER でベンダ拡張として通知されます。

```
SELECT * FROM Products JOIN SalesOrderItems;
```

SQL FLAGGER 機能は、SQL 文の実行に依存しません。すべての通知論理は単に静的なコンパイル時の処理として行われます。

参照

- 「SQLFLAGGER 関数 [その他]」 『SQL Anywhere サーバ - SQL リファレンス』
- 「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」 448 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」 896 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 要素の準拠状態を確認するために、SQL FLAGGER では次の ANSI SQL 標準を使用します。

- 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、マテリアライズド・ビュー、パブリケーション、サブスクリプション、プロキシ・テーブルを作成する文を含む) は、ANSI SQL 標準のベンダ拡張なので、準拠しないと通知されます。

SET OPTION 文とそのオプション要素は、どの SQL 標準に対しても、Ultra Light との互換性についても、準拠しないと通知されません。

参照

- 「Ultra Light SQL 要素」『Ultra Light データベース管理とリファレンス』
- 「SET OPTION 文」『SQL Anywhere サーバ - SQL リファレンス』

他の SQL ソフトウェアにはない機能

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。

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換に使用できます。「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。

Watcom-SQL

SQL Anywhere がサポートしている SQL ダイアレクトは Watcom-SQL と呼ばれています。SQL Anywhere は、最初にリリースされた 1992 年当時には Watcom SQL と呼ばれていました。SQL Anywhere がサポートする SQL ダイアレクトについては、現在でも Watcom-SQL という用語を使用しています。

SQL Anywhere は、Sybase Adaptive Server Enterprise がサポートする SQL ダイアレクトである Transact-SQL の大部分のサブセットもサポートしています。「Transact-SQL との互換性」 691 ページを参照してください。

Transact-SQL との互換性

SQL Anywhere は、Sybase Adaptive Server Enterprise がサポートする SQL ダイアレクトである Transact-SQL の大部分のサブセットをサポートしています。この項では、SQL Anywhere と Adaptive Server Enterprise 間の SQL の互換性について説明します。

目的

次に、SQL Anywhere の Transact-SQL に対するサポートの目的を示します。

- **アプリケーションの移植性**　アプリケーション、ストアド・プロシージャ、バッチ・ファイルの多くは、Adaptive Server Enterprise と SQL Anywhere データベースの両方で使用できるように作成できます。
- **データの移植性**　SQL Anywhere データベースと Adaptive Server Enterprise データベースの間では、最小限の作業でデータの交換とレプリケートができます。

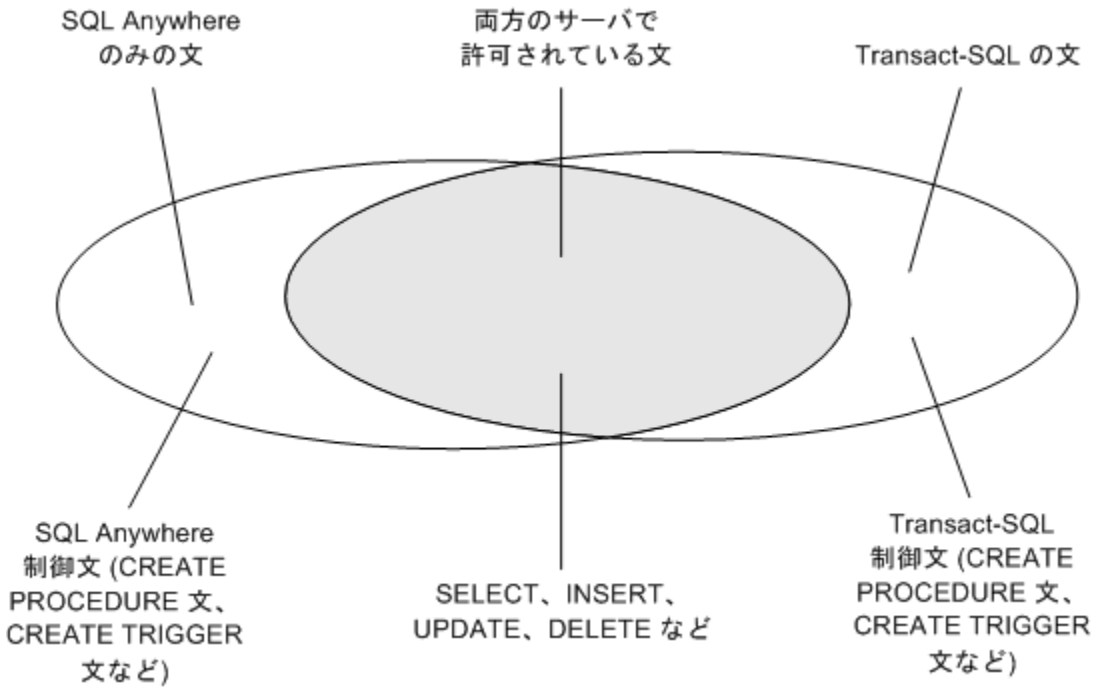
目的は、Adaptive Server Enterprise と SQL Anywhere の両方で動作するアプリケーションを作成することです。既存の Adaptive Server Enterprise アプリケーションの多くは、SQL Anywhere データベースで実行するには多少の変更が必要です。

Transact-SQL のサポート方法

次に、SQL Anywhere の Transact-SQL に対するサポート方法を示します。

- SQL 文の多くは、SQL Anywhere と Adaptive Server Enterprise 間で互換性があります。
- 特にプロシージャ、トリガ、バッチで使用されるプロシージャ言語で書かれた文のいくつかでは、これまでの SQL Anywhere のバージョンでサポートされる構文とともに、別の Transact-SQL 文がサポートされます。このような文については、SQL Anywhere では 2 種類の SQL ダイアレクトがサポートされています。これらのダイアレクトは Transact-SQL (Adaptive Server Enterprise のダイアレクト) および Watcom-SQL (SQL Anywhere のダイアレクト) と呼ばれています。
- プロシージャ、トリガ、バッチは Transact-SQL または Watcom-SQL ダイアレクトのどちらでも実行されます。バッチまたはプロシージャ内では、1 つのダイアレクトの制御文だけを通して使用します。たとえば、ダイアレクトごとに異なったフロー制御文があります。

2 つのダイアレクトが重なり合う状況を、次の図に示します。



類似点と相違点

SQL Anywhere は、既存のデータを処理するときの Transact-SQL 言語の要素、関数、文のほとんどをサポートします。たとえば SQL Anywhere は、すべての数値関数、集合関数、日付と時刻関数をサポートし、1 つを除くすべての文字列関数をサポートしています。別の例として、ジョインを使う拡張された DELETE 文と UPDATE 文がサポートされます。

さらに、SQL Anywhere では、Transact-SQL のストアド・プロシージャ言語 (CREATE PROCEDURE 文、CREATE TRIGGER 文、制御文など) 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。

Adaptive Server Enterprise のアーキテクチャ

Adaptive Server Enterprise と SQL Anywhere は、それぞれの明確な目的に合わせたアーキテクチャを持つ、相互に補完的な製品です。

この項では、Adaptive Server Enterprise と SQL Anywhere のアーキテクチャの違いについて説明します。また、互換性のあるデータベース管理のために SQL Anywhere に含まれている、Adaptive Server Enterprise に似たツールについても説明します。

サーバとデータベース

サーバとデータベースの関係は、Adaptive Server Enterprise と SQL Anywhere では異なります。

Adaptive Server Enterprise では各データベースはサーバ内に存在し、各サーバは複数のデータベースを持つことができます。ユーザはサーバに対するログイン権限を持っている場合、サーバに接続できます。サーバに接続すると、ユーザはサーバ上のパーミッションを持つ各データベースを使用できます。master データベースに保持されている、システム全体に適用されるシステム・テーブルは、サーバ上のすべてのデータベースに共通な情報を含んでいます。

SQL Anywhere に master データベースがない

SQL Anywhere では、Adaptive Server Enterprise の master データベースに対応するレベルはありません。その代わり、それぞれのデータベースが独立したエンティティであり、自分のシステム・テーブルを持っています。ユーザは、サーバに対してではなく 1 つのデータベースに対する接続権限を持ちます。ユーザが接続する場合、個々のデータベースに対する接続となります。master データベース・レベルで維持されているシステム全体にわたる一連のシステム・テーブルはありません。SQL Anywhere の各データベース・サーバは、複数のデータベースを動的にロード、アンロードできます。ユーザはそれぞれのデータベースに対する独立した接続を維持できます。

SQL Anywhere は、Transact-SQL のサポートと Open Server のサポートに対して、Adaptive Server Enterprise に似た方法でタスクを実行するツールを提供します。たとえば、SQL Anywhere は、Adaptive Server Enterprise の sp_addlogin システム・プロシージャの実装を提供し、同等の処理、つまりデータベースへのユーザの追加を実行します。「Open Server としての SQL Anywhere 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。

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。

SQL Anywhere では、ドメインはデフォルト値とそれに対応した CHECK 条件を持つことができ、その CHECK 条件はそのデータ型に基づいて定義されたすべてのカラムに適用されます。ドメインを作成するには、CREATE DOMAIN 文を使用します。

デフォルト値とルール、または CHECK 条件は、個々のカラムについて、CREATE TABLE 文または ALTER TABLE 文を使用して定義できます。

Adaptive Server Enterprise では、CREATE DEFAULT 文は名前付きデフォルトを作成します。このデフォルトは、特定のカラムにバインドすることによって、カラムのデフォルト値として使用できます。また、sp_bindefault システム・プロシージャを使用してドメインのカラムにバインドすると、ドメインのすべてのカラムのデフォルト値として使用できます。CREATE RULE 文は、名前のついたルールを作成します。このルールは、特定のカラムにバインドすることによって、さまざまなカラムのドメイン定義に使用でき、データ型にバインドすると、そのドメインのすべてのカラムのルールとして使用できます。ルールをデータ型やカラムにバインドするには、sp_bindrule システム・プロシージャを使用します。

参照

- 「CREATE DOMAIN 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「CREATE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「ALTER TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「探索条件」『SQL Anywhere サーバ - SQL リファレンス』

システム・テーブル

SQL Anywhere には、独自のシステム・テーブルのほかに、Adaptive Server Enterprise のシステム・テーブルに対応する箇所をシミュレートするビューがあります。

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。

Adaptive Server Enterprise では、データベース所有者 (ユーザ dbo) がシステム・テーブルを所有します。SQL Anywhere では、システム所有者 (ユーザ SYS) がシステム・テーブルを所有します。ユーザ dbo は、SQL Anywhere が提供する Adaptive Server Enterprise と互換性のあるシステム・ビューを所有します。

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 DBA です。
- RESOURCE 権限は、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。

参照

- 「データベースのパーミッションと権限の概要」『SQL Anywhere サーバ - データベース管理』
- 「DBA 権限」『SQL Anywhere サーバ - データベース管理』
- 「RESOURCE 権限」『SQL Anywhere サーバ - データベース管理』

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、ユーザはサーバに接続します。各ユーザには、サーバのログイン ID とパスワード、およびそのサーバ上でアクセスする各データベースのユーザ ID が必要になります。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。「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。

システム・プロシージャ	説明
sp_adduser	Adaptive Server Enterprise と SQL Anywhere の両方で、ユーザをデータベースに追加する。Adaptive Server Enterprise では、これは sp_addlogin とは異なるタスクですが、SQL Anywhere では同じです。
sp_addgroup	グループをデータベースに追加する。
sp_changegroup	ユーザをグループに追加するか、ユーザをあるグループから別のグループに移動する。
sp_droplogin	Adaptive Server Enterprise では、ユーザをサーバから削除する。SQL Anywhere では、ユーザをデータベースから削除します。
sp_dropuser	ユーザをデータベースから削除する。
sp_dropgroup	グループをデータベースから削除する。

Adaptive Server Enterprise では、ログイン ID はサーバ全体に適用されます。SQL Anywhere では、ユーザは個々のデータベースに属します。

データベース・オブジェクト・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。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。「ユーザとグループ」 697 ページを参照してください。

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。

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 11] - [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)。

- 次の dbinit コマンドを実行します。

  ```
  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見なします。

```
'ignore the trailing blanks   '
'ignore the trailing blanks'
```

このオプションを選択しないと、SQL Anywhere ではこの 2 つの文字列は異なっていると見なします。

このオプションを選択すると、クライアント・アプリケーションでフェッチした文字列にブランクが埋め込まれます。

古いシステム・ビューの削除

SQL Anywhere の古いバージョンでは 2 つのシステム・ビューを使用していましたが、互換性を確保しようとすると、それらの名前は Adaptive Server Enterprise のシステム・ビューと矛盾します。その 2 つのビューは SYSCOLUMNS と SYSINDEXES です。Open Client または JDBC のインタフェースを使用している場合は、これらのビューを除外してデータベースを作成します。この処理には dbinit -k オプションを使用します。

データベースの作成時にこのオプションを使用しなかった場合、SELECT * FROM SYSCOLUMNS; 文を実行すると、「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。

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 標準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 標準に一致する SQL Anywhere とは異なります。

quoted_identifier オプションは、Adaptive Server Enterprise と SQL Anywhere の両方で使用できます。識別子と文字列が互換性を持って処理されるように、両方のデータベースでこのオプションが同じ値に設定されていることを確認します。「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。

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。

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 [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。

「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。

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。

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 カラムは、送り状番号や従業員番号のような連続番号を格納します。このカラムは、自動生成されます。IDENTITY カラムの値はテーブルの各ローをユニークに識別します。

Adaptive Server Enterprise では、データベースの各テーブルは IDENTITY カラムを 1 つだけ持つことができます。データ型は numeric、位取りは 0 (ゼロ)、NULL 値は許可されません。

SQL Anywhere では、IDENTITY カラムはカラムのデフォルトとして設定されます。連続番号でない値も、INSERT 文を使用してカラムに明示的に挿入できます。Adaptive Server Enterprise は、identity_insert オプションが **on** に設定されないかぎり、identity カラムへの INSERT を許可しません。SQL Anywhere では、NOT NULL プロパティ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 カラムを作成するには、次の CREATE TABLE 構文を使用します。ただし、*n* にはテーブルに挿入される可能性のある最大ロー数の値を保持できる大きさである必要があります。

```
CREATE TABLE table-name (
  ...
  column-name numeric(n,0) IDENTITY NOT NULL,
  ...
)
```

@@identity を使用する IDENTITY カラム値の検索

初めてローを 1 つテーブルに挿入すると、IDENTITY カラムに 1 という値が割り当てられます。その後は挿入するたびに、カラムの値が 1 ずつ増えます。最後に IDENTITY カラムに挿入した値は、@@identity グローバル変数として使用できます。

@@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。

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 と呼ばれます。「SQL FLAGGER を使用した SQL 準拠のテスト」 685 ページを参照してください。

次に、互換性のある SQL を記述するときの一般的なガイドラインを示します。

- デフォルトの動作を使用せずに、使用可能なオプションをすべて含めます。
- 演算子の優先順位のデフォルトが同じであるとするのではなく、文中の実行の順序をカッコを使用して明確にします。
- Adaptive Server Enterprise に移植できるように、変数名には @ をプレフィクスとして付ける Transact-SQL の規則に従います。
- プロシージャ、トリガ、バッチでは、BEGIN 文の直後で変数とカーソルを宣言します。Adaptive Server Enterprise では、プロシージャ、トリガ、バッチ内のどこででも宣言できますが、SQL Anywhere では、BEGIN 文の直後で行う必要があります。
- データベース内の識別子として、Adaptive Server Enterprise または SQL Anywhere の予約語を使用しないでください。
- 大きいネームスペースを想定します。たとえば、各インデックスにはユニークな名前を持たせるようにします。

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。

このオプションの詳細については、「Transact-SQL との互換性を維持するためのオプション設定」 702 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 文のテーブル名の先頭にシャープ記号 (#) を付けます。このようなテンポラリ・テーブルは、SQL Anywhere の宣言テンポラリ・テーブルであり、現在の接続でしか使用できません。

テーブルの物理的配置は、Adaptive Server Enterprise と SQL Anywhere では実行方法が異なります。SQL Anywhere は **ON** *segment-name* 句をサポートしますが、*segment-name* は SQL Anywhere の DB 領域を参照します。

参照

- 「SQL FLAGGER を使用した SQL 準拠のテスト」 685 ページ
- 「DECLARE LOCAL TEMPORARY TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「CREATE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』

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*"

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 句
- 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。
- テーブル仕様のパフォーマンス・パラメータ部分は解析されますが、効果はありません。
- HOLDLOCK キーワードは SQL Anywhere によってサポートされます。HOLDLOCK の場合、データ・ページが不要になっても共有ロックが解放されないため、指定されたテーブルやビューの共有ロックの制限はより厳しくなります。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。

```
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。

参照

- 「SQL FLAGGER を使用した SQL 準拠のテスト」 685 ページ
- 「SELECT 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「OLAP のサポート」 481 ページ

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 標準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」 413 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 =*)」 432 ページを参照してください。

参照

- 「SQL FLAGGER を使用した SQL 準拠のテスト」 685 ページ

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 標準を基準にしており、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 は、文レベルの AFTER トリガをサポートしています。つまり、このトリガは、トリガを起動する文が完了してから実行されます。SQL Anywhere は、ローレベルの BEFORE、AFTER、INSTEAD OF トリガと、文レベルの AFTER および INSTEAD OF トリガをサポートしています。「トリガの概要」 886 ページを参照してください。

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」 873 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。トリガとなる動作とそのトリガがともにアトミック・ステートメントを構成するためです。

  SQL Anywhere は、Adaptive Server Enterprise と互換性のある ROLLBACK TRIGGER 文を提供します。この文は、トリガ内で動作を取り消すために使用します。「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。「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 を使用したストアド・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/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。

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」 905 ページを参照してください。

Transact-SQL プロシージャの中の変数

SQL Anywhere は SET 文を使用して、プロシージャ内の変数に値を割り当てます。Transact-SQL では、空のテーブル・リストの SELECT 文、または SET 文を使用して値を割り当てます。次のプロシージャは、Transact-SQL 構文の働きを示します。

```
CREATE PROCEDURE multiply
        @mult1 int,
        @mult2 int,
        @result int output
AS
SELECT @result = @mult1 * @mult2;
```

このプロシージャを呼び出すには、次のようにします。

```
CREATE VARIABLE @product int
go
EXECUTE multiply 5, 6, @product OUTPUT
go
```

変数 @product の値は、プロシージャの実行後 30 になります。

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」 707 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。

Transact-SQL プロシージャでのエラー処理

デフォルトでのプロシージャのエラー処理は、Watcom-SQL と Transact-SQL とでは違います。Watcom-SQL のデフォルトでは、エラーが起こった場合にプロシージャは終了し、呼び出した環境に SQLSTATE 値と SQLCODE 値を返します。

EXCEPTION 文を使って、Watcom-SQL ストアド・プロシージャに明示的なエラー処理を組み込むことができます。または、ON EXCEPTION RESUME 文を使って、エラーが起こった次の文から実行を再開するよう、プロシージャに指示することもできます。

Transact-SQL のプロシージャでエラーが起こった場合は、次の文から実行が継続されます。最後に実行された文のエラー・ステータスは、グローバル変数 @@error に保存されます。文の後ろにあるこの変数をチェックして、プロシージャから強制的に返すことができます。たとえば、次の文は、エラーが起こると終了させます。

```
IF @@error != 0 RETURN
```

プロシージャが実行を終了したときの戻り値で、プロシージャが成功したかどうかがわかります。この戻り値は整数で、次のように指定してアクセスできます。

```
DECLARE @Status INT
EXECUTE @Status = proc_sample
IF @Status = 0
  PRINT 'procedure succeeded'
ELSE
  PRINT 'procedure failed'
```

次のテーブルは、組み込みプロシージャの戻り値とその意味を示します。

値	定義	SQL Anywhere SQLSTATE
0	プロシージャはエラーを起こすことなく実行された	
-1	オブジェクトが見つからない	42W33、52W02、52003、52W07、42W05
-2	データ型エラー	53018
-3	プロセスはデッドロックの対象となった	40001、40W06
-4	パーミッション・エラー	42501
-5	構文エラー	42W04
-6	その他のユーザ・エラー	
-7	リソース・エラー、たとえば領域が足りない	08W26

値	定義	SQL Anywhere SQLSTATE
-10	致命的な内部の矛盾	40W01
-11	致命的な内部の矛盾	40000
-13	データベースが壊れている	WI004
-14	ハードウェア・エラー	08W17、40W03、40W04

SQL Anywhere SQLSTATE が該当しない場合、デフォルト値の -6 が返されます。

RETURN 文は、この表の値以外の、ユーザが意味を定義した整数値を返すこともできます。

プロシージャ内での RAISERROR 文の使用

RAISERROR 文を使用してユーザ定義エラーを作成します。RAISERROR 文は、SIGNAL 文と同じように機能します。「RAISERROR 文」 『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。on_tsql_error オプションをデフォルト値 (Conditional) に設定した場合、continue_after_raiserror オプションは、RAISERROR 文の実行後の動作を制御します。on_tsql_error オプションを Stop または Continue に設定した場合、その設定は continue_after_raiserror の設定より優先されます。

中間レベルの RAISERROR ステータスとコードは、プロシージャが終了すると失われます。結果が返されるときに RAISERROR とともにエラーが発生した場合は、エラー情報が返され、RAISERROR 情報は失われます。アプリケーションでは、別の実行ポイントでグローバル変数 @@error を検査して、中間の RAISERROR ステータスを問い合わせることができます。

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。

データベースにおける XML

この項では、データベースで XML を使用する方法について説明します。

データベースにおける XML の使用

目次

Extensible Markup Language (XML) は、構造化データをテキスト形式で表します。XML は、大規模な電子出版の課題を満たすために設計されました。

XML は、HTML のように単純なマークアップ言語ですが、SGML のように柔軟性があります。XML は、階層型で、その主な目的は、人間とコンピュータの両方が作成し、読むことのできるデータの構造を記述することです。

XML では、さまざまな形式のデータを記述する、一連の静的な要素は提供されておらず、ユーザが要素を定義できます。そのため、XML を使用して多くの種類の構造化データを記述できます。XML 文書では、オプションとして文書型定義 (DTD) または XML スキーマを使用し、XML ファイルで使用される構造、要素、属性を定義できます。

SQL Anywhere で XML を使用する方法はいくつかあります。

- データベースに XML 文書を格納する
- リレーショナル・データを XML としてエクスポートする
- データベースに XML をインポートする
- リレーショナル・データを XML として問い合わせる

XML の詳細については、http://www.w3.org/XML/ を参照してください。

リレーショナル・データベースにおける XML 文書の格納

SQL Anywhere は、データベースで XML 文書を格納するために使用できる 2 つのデータ型、XML データ型と LONG VARCHAR データ型をサポートしています。これらのデータ型は、いずれも XML 文を文字列としてデータベースに格納します。

XML データ型は、データベース・サーバの文字セット・エンコードを使用します。XML エンコード属性は、データベース・サーバが使用するエンコードと一致する必要があります。XML エンコード属性は、自動文字セット変換の実行方法を指定しません。

XML データ型は、文字列と相互にキャスト可能なすべてのデータ型と、相互にキャストできます。文字列が XML にキャストされるときに、整形式かどうかはチェックされ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」 734 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」 724 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。

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。
- **//** ルートのすべての子孫を示します。ルート・ノードも含まれます。
- **. (単一のピリオド)** XML 文書のカレント・ノードを示します。
- **.//** カレント・ノードのすべての子孫を示します。カレント・ノードも含まれます。
- **..** カレント・ノードの親ノードを示します。
- ***./@attributename*** *attributename* という名前を持つ、カレント・ノードの属性を示します。
- ***./childname*** カレント・ノードの子で、*childname* という名前を持つ要素を示します。

次の XML 文書を考えてみます。

```
<inventory>
  <product ID="301" size="Medium">Tee Shirt
    <quantity>54</quantity>
  </product>
  <product ID="302" size="One Size fits all">Tee Shirt
    <quantity>75</quantity>
  </product>
  <product ID="400" size="One Size fits all">Baseball Cap
    <quantity>112</quantity>
  </product>
</inventory>
```

<inventory> 要素は、ルート・ノードです。この要素は、次の XPath 式を使用して参照できます。

```
/inventory
```


カレント・ノードが <quantity> 要素であると仮定します。このノードは、次の XPath 式を使用して参照できます。

```
.
```

<inventory> 要素の子である <product> 要素をすべて検出するには、次の XPath 式を使用します。

```
/inventory/product
```

カレント・ノードが <product> 要素のときに、size 属性を参照したい場合は、次の XPath 式を使用します。

```
./@size
```

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。

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」 737 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>
    <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。メタプロパティ名では大文字と小文字が区別されます。そのため、ID や LOCALNAME はメタプロパティ名として使用できません。

```
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>
```

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、ラテラル派生テーブルを使用します。

次の文は、ManagerID と Reports という 2 つのカラムを持つテーブルを作成します。Reports カラムには、Employees テーブルから生成された XML データが含まれます。

```
CREATE TABLE 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>

ManagerID	Reports
902	<reports> <e>129</e> <e>195</e> <e>299</e> <e>467</e> ... </reports>
1293	<reports> <e>148</e> <e>390</e> <e>586</e> <e>757</e> ... </reports>
...	...

次のクエリは、ラテラル派生テーブルを使用して、2 つのカラムを持つ結果セットを生成しています。1 つのカラムは、各マネージャの ID をリストします。もう 1 つのカラムは、そのマネージャに報告を行う各従業員の ID をリストします。

```
SELECT ManagerID, EmployeeID
FROM 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
...	...

ラテラル派生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。

SQL Anywhere .NET データ・プロバイダを使用して、DataSet を基に 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。

デフォルトの XML ネームスペースの定義

デフォルトのネームスペースは、xmlns="URI" の形式の属性で、XML 文書の要素に定義します。次の例では、文書には http://www.iAnywhere.com/EmployeeDemo という URI にバインドされるデフォルトのネームスペースがあります。

```
<x xmlns="http://www.iAnywhere.com/EmployeeDemo"/>
```

要素の名前にプレフィクスがない場合は、その要素およびその要素のすべての子孫要素にデフォルトのネームスペースが適用されます。コロンによって、プレフィクスと残りの要素名は区切られます。たとえば、<x/> にはプレフィクスがないが、<p:x/> にはプレフィクス p があります。xmlns:prefix="URI" の形式の属性によって、プレフィクスにバインドされるネームスペースを定義します。次の例では、文書がプレフィクス p を前述の例と同じ URI にバインドします。

```
<x xmlns:p="http://www.iAnywhere.com/EmployeeDemo"/>
```

デフォルトのネームスペースが属性に適用されることはありません。プレフィクスがある場合を除き、属性は常に NULL ネームスペース URI にバインドされます。次の例では、ルート要素と子要素には iAnywhere1 ネームスペースがあり、x 属性には NULL ネームスペース URI、y 要素には iAnywhere2 ネームスペースがあります。

```
<root xmlns="iAnywhere1" xmlns:p="iAnywhere2">
<child x='1' p:y='2' />
</root>
```

XML 文書を、openxml クエリの *namespace-declaration* 引数として渡すと、文書のルート要素に定義されたネームスペースはクエリに適用されます。ルート要素以降の残りの文書はすべて無視されます。次の例では、p1 は文書では iAnywhere1 にバインドされ、*namespace-declaration* 引数では p2 にバインドされます。クエリはプレフィクス p2 を使用できます。

```
SELECT *
FROM openxml( '<p1:x xmlns:p1="iAnywhere1"> 1 </x>', '/p2:x', 1, '<root xmlns:p2="iAnywhere1"/>' )
WITH ( c1 int '.' );
```

要素を一致させる場合、プレフィクスがバインドされる URI を正確に指定する必要があります。上の例では、xpath クエリの x 名 と文書の x 要素は、どちらも iAnywhere1 ネームスペースがあるため、一致となります。

openxml システム・プロシージャの *namespace-declaration* では、デフォルトのネームスペースを使用しないでください。NULL ネームスペースを含む任意の URI にバインドされる x 要素と一致する、/*:x 形式のワイルドカード・クエリを使用してください。または、特定のプレフィクスに必要な URI をバインドし、それをクエリで使用します。XML 文書から結果セットを生成する方法の詳細については、「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。

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」 733 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 標準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」 750 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 通り提供されます。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。

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」 735 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」 736 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」 737 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 の形式を制御できます。

  詳細については、「FOR XML EXPLICIT の使用」 740 ページを参照してください。

次の項では、FOR XML 句の 3 つのモードに共通する、バイナリ・データ、NULL 値、無効な XML 名に関連する動作について説明します。また、FOR XML 句の使用例も示します。

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。
- 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』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
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;amp; D</DepartmentName>
</row>
<row>
  <EmployeeID>105</EmployeeID>
  <DepartmentName>R &amp;amp; D</DepartmentName>
</row>
<row>
  <EmployeeID>160</EmployeeID>
  <DepartmentName>R &amp;amp; D</DepartmentName>
</row>
<row>
  <EmployeeID>243</EmployeeID>
  <DepartmentName>R &amp;amp; D</DepartmentName>
</row>
...
```

使用法

BINARY、LONG BINARY、IMAGE、VARBINARY カラムのデータは、FOR XML RAW を含むクエリを実行すると、自動的に Base64 エンコード・フォーマットで返されます。

デフォルトでは、NULL 値は、結果から省略されます。for_xml_null_treatment オプションを使用すると、この動作を制御できます。

FOR XML 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」 734 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;amp; D"/>
<row EmployeeID="105" DepartmentName="R &amp;amp; D"/>
<row EmployeeID="160" DepartmentName="R &amp;amp; D"/>
<row EmployeeID="243" DepartmentName="R &amp;amp; D"/>
...
```

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;amp; D</DepartmentName>
  </Departments>
</Employees>
<Employees>
  <EmployeeID>105</EmployeeID>
  <Departments>
    <DepartmentName>R &amp;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。

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、必要なネストが得られないことがあります。

FOR XML AUTO は、整形式 XML 文書を返しません。これは、文書に単一のルート・ノードが含まれないためです。<root> 要素が必要な場合は、1 つの方法として、XMLELEMENT 関数を使用して挿入できます。次に例を示します。

```
SELECT XMLELEMENT( NAME root,
            (SELECT EmployeeID AS id, GivenName AS name
            FROM Employees FOR XML AUTO ) );
```

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 Employees.DepartmentID=Departments.DepartmentID
ORDER BY EmployeeID
FOR XML AUTO;
```

前述のクエリによって、次の XML が生成されます。

```
<employee EmployeeID="102">
  <department DepartmentName="R &amp;amp; D"/>
</employee>
<employee EmployeeID="105">
  <department DepartmentName="R &amp;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 Employees.DepartmentID=Departments.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、必要なネストに関する情報がクエリ結果内で指定されるようにしてください。FOR XML EXPLICIT がサポートするオプションのディレクティブを使用すると、個別のカラムの扱いを設定できます。たとえば、あるカラムが要素内容と属性内容のどちらとして表示されるかを制御できます。また、あるカラムが生成された XML に含まれるのではなく、結果の順序付けのみに使用されるように制御できます。

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」 742 ページを参照してください。

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」 741 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」 742 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。

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」 747 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」 746 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」 748 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」 749 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」 734 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。

```
<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,
       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」 740 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;amp; D"/>
</employee>
<employee ID="105" name="Matthew">
  <department name="R &amp;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;amp; D</name>
  </department>
</employee>
<employee>
  <id>105</id>
  <name>Matthew</name>
  <department>
    <name>R &amp;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;
```

このクエリは、次の結果を返します。

```
<employee name="Fran">
  <department name="R &amp;amp; D"/>
</employee>
<employee name="Matthew">
  <department name="R &amp;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」 734 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;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。

```
<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、ドラフト段階の標準です。SQL/XML は、XML とともに SQL を使用する方法を定めています。サポートされる関数を使用して、リレーショナル・データから XML 文書を構成するクエリを記述できます。

無効な名前と SQL/XML

SQL/XML では、スペースを含む式など、有効な XML 名でない式は、FOR XML 句と同様にエスケープされます。XML 型の要素の内容は、引用されません。

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」 734 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」 722 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>

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。

```
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。

ID	product_info
401	<item_description> <name>Baseball Cap</name> <color>White</color> <price>10.00</price> </item_description>
500	<item_description> <name>Visor</name> <color>White</color> <price>7.00</price> </item_description>
501	<item_description> <name>Visor</name> <color>Black</color> <price>7.00</price> </item_description>
...	...

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。

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_info
<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」 750 ページを参照してください。

XML 文書のヘッダ情報の指定

SQL Anywhere がサポートする FOR XML 句と SQL/XML 関数は、生成する XML 文書にバージョン宣言情報を含めません。XMLGEN 関数を使用すると、ヘッダ情報を生成できます。

```
SELECT XMLGEN( '<?xml version="1.0"
               encoding="ISO-8859-1" ?>
               <r>{$x}</r>',
               (SELECT GivenName, Surname
      FROM Customers FOR XML RAW) AS x );
```

このクエリは、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。

リモート・データとバルク・オペレーション

この項では、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。

データのインポートとエクスポート

目次

「バルク・オペレーション」という用語は、データのインポートやエクスポートのプロセスを説明するために使用されます。バルク・オペレーションは、DBA 権限のある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・ログに影響する可能性があるため、ユーザがデータベースに接続していないときに実行する必要があります。

データをインポートおよびエクスポートする際の、一般的な状況を次に示します。

- 新しいデータベースに最初のデータ・セットをインポートする
- データベースの構造を修正した場合などに、新しいデータベースを構築する
- スプレッドシートなど、他のアプリケーションで使うためにデータベースからデータをエクスポートする
- レプリケーションまたは同期に使用するデータベースの抽出を作成する
- 破損したデータベースを修復する
- データベースを再構築してパフォーマンスを向上させる
- 新しいバージョンのデータベース・ソフトウェアを入手し、ソフトウェア・アップグレードを完了する

バルク・オペレーションのパフォーマンスの側面

バルク・オペレーションのパフォーマンスは、オペレーションがデータベース・サーバの内部と外部のどちらに対するものかなどの要因によって決まります。

内部バルク・オペレーション

内部バルク・オペレーションは、LOAD TABLE および UNLOAD 文を使用してデータベース サーバによって実行されるインポートおよびエクスポート操作であり、**サーバ側**バルク・オペレーションとも呼ばれます。

内部バルク・オペレーションを実行する場合は、ASCII テキスト・ファイルまたは Adaptive Server Enterprise の BCP ファイルからロードしたり、これらのファイルにアンロードすることができます。これらのファイルは、データベース サーバと同じコンピュータに存在することも、クライアント・コンピュータに存在することもできます。書き込みまたは読み込み対象のファイルのパスは、データベース・サーバからの相対パスとして指定します。内部バルク・オペレーションは、データベースのデータのインポートおよびエクスポート方法としては最速です。

外部バルク・オペレーション

外部バルク・オペレーションは、INPUT および OUTPUT 文を使用して Interactive SQL などのクライアントによって実行されるインポートおよびエクスポート操作であり、**クライアント側**バルク・オペレーションとも呼ばれます。クライアントが INPUT 文を発行すると、INPUT 文で指定されたファイルの処理時に読み込まれた各ローに対して、トランザクション・ログに INSERT 文が記録されます。このため、クライアント側ロードはサーバ側ロードよりかなり遅くなります。また、INPUT 中に INSERT トリガが起動されます。

OUTPUT 文により、SELECT 文の結果セットを複数の異なるファイル・フォーマットで書き出すことが可能になります。

外部バルク・オペレーションの場合、読み込みまたは書き込み対象のファイルのパスは、クライアント・アプリケーションが実行されているコンピュータからの相対パスとして指定します。

参照

- 「LOAD TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「UNLOAD 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「INPUT 文 [Interactive SQL]」 『SQL Anywhere サーバ - SQL リファレンス』
- 「OUTPUT 文 [Interactive SQL]」 『SQL Anywhere サーバ - SQL リファレンス』
- 「データのインポートのパフォーマンスに関するヒント」 764 ページ
- 「-b サーバ・オプション」 『SQL Anywhere サーバ - データベース管理』

バルク・オペレーションのデータ・リカバリの問題

バルク・オペレーション・モード (-b サーバ・オプション) でデータベース・サーバを実行できます。このオプションを使用した場合、データベース・サーバでは一部の重要な機能を実行しません。具体的には、次のとおりです。

関数	影響
トランザクション・ログの管理	変更の記録がありません。COMMIT を実行するたびにチェックポイントが設定されます。
レコードのロック	重大な影響はありません。

また、バルク・ロードしたデータをリカバリ時にも使用できるようにすることが必要な場合もあります。そのためには、元のデータ・ソースを元の場所にそのまま残します。また、LOAD TABLE 文の一部のロギング・オプションを使用して、バルク・ロードしたデータをトランザクション・ログに記録することもできます。「LOAD TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

警告
バルク・オペレーション・モードでは、メディア障害に対してデータベースが防御されないので、このモードの使用前後には、データベースのバックアップを行ってください。

参照

- 「-b サーバ・オプション」 『SQL Anywhere サーバ - データベース管理』

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 を使用して、次の作業を実行できます。

- テキスト・ファイルからテーブル全体またはテーブルの一部をインポートする
- 変数からデータをインポートする
- スクリプトでインポート手順を自動化して、複数のテーブルを連続的にインポートする
- テーブルにデータを挿入または追加する
- テーブル内のデータを置換する
- インポートの前またはインポート中にテーブルを作成する
- クライアント・コンピュータにあるファイルからデータをロードする
- BCP フォーマット句を使用して、SQL Anywhere と Adaptive Server Enterprise の間でファイルを転送する

まったく新しいデータベースを作成する場合、パフォーマンスを最適化するには、LOAD TABLE を使用してデータをロードしてください。

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」 796 ページを参照してください。

参照

- 「LOAD TABLE 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「UNLOAD 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「INPUT 文 [Interactive SQL]」 『SQL Anywhere サーバ - SQL リファレンス』
- 「OUTPUT 文 [Interactive SQL]」 『SQL Anywhere サーバ - SQL リファレンス』
- 「データのインポートのパフォーマンスに関するヒント」 764 ページ
- 「バルク・オペレーションのパフォーマンスの側面」 762 ページ
- 「-b サーバ・オプション」 『SQL Anywhere サーバ - データベース管理』
- 「インポート用のテーブル構造」 780 ページ
- 「クライアント・コンピュータ上のデータへのアクセス」 793 ページ

データのインポートのパフォーマンスに関するヒント

大量のデータをインポートするには時間がかかる場合があります。時間を節約するには次のような方法があります。

- データ・ファイルとデータベースは、物理的に別のディスク・ドライブに置く。ロード中のディスク・ヘッドの動きを削減できます。
- データベースのサイズを拡張する。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。また、大量のデータをロードする場合のパフォーマンスを改善でき、ファイル・システム内

でデータベースの断片化を防ぐことができます。「ALTER DBSPACE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- データのロードにテンポラリ・テーブルを使用する。ローカルまたはグローバルのテンポラリ・テーブルは、データ・セットを繰り返しロードする必要がある場合、または異なる構造を持つテーブルをマージする必要がある場合に役立ちます。
- LOAD TABLE 文を使用している場合は、-b オプション (バルク・オペレーション・モード) を指定せずにデータベース・サーバを起動する。
- INPUT 文または OUTPUT 文を使用している場合は、データベース・サーバと同じコンピュータ上で Interactive SQL またはクライアント・アプリケーションを実行する。ネットワークを介してデータをロードすると、通信のために余分な負荷がかかります。新しいデータは、データベース・サーバがビジー状態ではないときに徐々にロードすることをおすすめします。

参照

- 「LOAD TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「INPUT 文 [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』
- 「OUTPUT 文 [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』
- 「-b サーバ・オプション」『SQL Anywhere サーバ - データベース管理』

インポート・ウィザードを使用したデータのインポート

Interactive SQL の **インポート・ウィザード**は、データのソース、フォーマット、インポート先テーブルを選択するために使用します。データは TEXT および FIXED フォーマットのファイルからインポートできます。データは既存のテーブルまたは新しいテーブルにインポートできます。また、**インポート・ウィザード**を使用すると、次の間でデータのインポートを行うこともできます。

- SQL Anywhere データベースと Ultra Light データベースなど、タイプの異なるデータベース。
- SQL Anywhere バージョン 11.0.0 データベースと SQL Anywhere バージョン 10.0.0 データベースなど、バージョンの異なるデータベース (各データベースの ODBC ドライバがある場合)。

次の場合は、Interactive SQL の**インポート・ウィザード**を使用します。

- データのインポートと同時にテーブルを作成する場合
- テキスト以外のフォーマットでのデータのインポートにポイント・アンド・クリック・インタフェースを使用する場合

♦ **データをインポートするには、次の手順に従います。**

1. Interactive SQL で、**[データ] - [インポート]** を選択します。
2. **インポート・ウィザード**の指示に従います。

♦ データをファイルから SQL Anywhere サンプル・データベースにインポートするには、次の手順に従います。

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 で、**[データ] - [インポート]** を選択します。
3. **[テキスト・ファイル]** を選択して **[次へ]** をクリックします。
4. **[ファイル名]** フィールドに **import.txt** と入力します。
5. **[既存のテーブル]** を選択します。
6. **[Employees]** を選択して **[次へ]** をクリックします。
7. **[フィールド・セパレータ]** リストで、**[カンマ (,)]** を選択します。**[次へ]** をクリックします。
8. **[インポート]** をクリックします。
9. **[閉じる]** をクリックします。

   インポートが完了すると、ウィザードで作成した SQL 文がコマンド履歴に保存されます。

   [SQL] - [前の SQL] を選択すると、生成した SQL 文を表示できます。

   インポート・ウィザードによって生成された IMPORT 文が **[SQL 文]** ウィンドウ枠に次のように表示されます。

```
-- Generated by the Import Wizard
INPUT INTO "GROUPO"."Employees" from 'C:¥¥Tobedeleted¥¥import.txt'
FORMAT TEXT ESCAPES ON ESCAPE CHARACTER '¥¥' DELIMITED BY ',' ENCODING 'Cp1252'
```

♦ データを SQL Anywhere サンプル・データベースから Ultra Light データベースにインポートするには、次の手順に従います。

1. Ultra Light データベース (*C:¥Documents and Settings¥All Users¥Documents¥SQL Anywhere 11¥Samples¥UltraLite¥CustDB¥custdb.udb* など) に接続します。
2. Interactive SQL で、**[データ] - [インポート]** を選択します。
3. **[データベース]** をクリックします。**[次へ]** をクリックします。
4. **[データベース・タイプ]** リストで、**[SQL Anywhere]** を選択します。
5. **[ID]** タブで、**[ODBC データ・ソース名]** をクリックし、**SQL Anywhere 11 Demo** と入力します。**[次へ]** をクリックします。
6. **[テーブル名]** リストで、**[Customers]** を選択します。**[次へ]** をクリックします。
7. **[新しいテーブル]** をクリックします。
8. **[所有者]** リストで、**dba** を選択します。

9. **[テーブル名]** フィールドに **SQLAnyCustomers** と入力します。
10. **[インポート]** をクリックします。
11. **[閉じる]** をクリックします。
12. 生成された SQL 文を表示するには、**[SQL]** - **[前の SQL]** を選択します。

    インポート・ウィザードによって作成および使用された IMPORT 文が **[SQL 文]** ウィンドウ枠に表示されます。

```
-- Generated by the Import Wizard
INPUT USING 'DSN=SQL Anywhere 11 Demo;CON=""'
FROM "GROUPO.Customers" INTO "dba"."SQLAnyCustomers"
CREATE TABLE ON
```

INPUT 文を使用したデータのインポート

INPUT 文は、異なるファイル・フォーマットのデータを既存のテーブルや新しいテーブルにインポートするために使用します。データベースの ODBC ドライバが存在する場合は、USING 句を使用すると、異なる種類のデータベースや異なるバージョンの SQL Anywhere データベースからデータをインポートできます。

INPUT 文を使用すると、TEXT および FIXED フォーマットのデータをインポートできます。他のファイル・フォーマットのデータをインポートするには、USING 句と ODBC データ・ソースを使用します。

デフォルトの入力フォーマットを使用したり、INPUT 文ごとにファイル・フォーマットを指定することができます。INPUT 文は Interactive SQL 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。

INPUT 文は、ファイルまたは他のデータベースからデータをインポートする場合に使用します。

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。

マテリアライズド・ビューに関する考慮事項

即時ビューでは、基本となるテーブルにデータをバルク・ロードしようとするとエラーが返されます。最初にビューのデータをトランケートしてから、バルク・ロード・オペレーションを実行する必要があります。

手動ビューでは、基本となるテーブルにデータをバルク・ロードできます。ただし、ビュー内のデータは次回のリフレッシュ時までは古いままです。

テーブルへのバルク・ロード・オペレーション (INPUT など) を実行する際は、先に従属するマテリアライズド・ビューのデータをトランケートすることを検討してください。データをロードしたら、ビューをリフレッシュします。「TRUNCATE 文」『SQL Anywhere サーバ - SQL リファレンス』と「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

テキスト・インデックスに関する考慮事項

即時テキスト・インデックスでは、基本となるテーブルで INPUT などのバルク・ロード・オペレーションを実行してからテキスト・インデックスを更新すると、自動更新にもかかわらず時間がかかる場合があります。手動テキスト・インデックスでは、リフレッシュにも時間がかかる場合があります。

テーブルへのバルク・ロード・オペレーション (INPUT など) を実行する際は、先に従属するテキスト・インデックスを削除することを検討してください。データをロードしたら、テキスト・インデックスを再作成します。「DROP TEXT INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』と「CREATE TEXT INDEX 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

INPUT 文を使用すると、変更内容はトランザクション・ログに記録されます。メディア障害が発生した場合は、詳細な変更の記録があります。ただし、この方法ではすべてのローがトランザクション・ログに書き込まれるので、この方法で大量のデータをインポートすると、パフォーマンスに影響が及びます。

対照的に、LOAD TABLE 文では各ローがトランザクション・ログに保存されないため、INPUT 文よりも高速な可能性があります。ただし、サポートされるデータベースやファイル・フォーマットの面では、INPUT 文の方が柔軟性があります。

♦ データをインポートするには、次の手順に従います (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. Interactive SQL を開き、SQL Anywhere 11 Demo データベースに接続します。
3. **[SQL 文]** ウィンドウ枠に INPUT 文を入力します。

   ```
   INPUT INTO Employees
   FROM c:¥new_employees.txt
   FORMAT TEXT;
   SELECT * FROM Employees;
   ```

   この文では、SQL Anywhere 11 Demo データベース内のインポート先テーブルの名前は Employees 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理由を示すメッセージが表示されます。

♦ **データを Microsoft Excel スプレッドシートから SQL Anywhere 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. Microsoft Excel でスプレッドシートを開きます。
2. Microsoft Excel で、インポートするセルを選択し、**[挿入]** - **[名前]** - **[定義]** を選択します。

   選択したセルに **myData** などの名前を入力します。
3. **[OK]** をクリックします。
4. スプレッドシートを保存して閉じます。
5. スプレッドシート用の ODBC データ・ソースを作成します。
   - **[スタート]** - **[プログラム]** - **[SQL Anywhere 11]** - **[ODBC アドミニストレータ]** を選択します。
   - **[ユーザー DSN]** タブを選択して現在のユーザー用の DSN を作成するか、**[システム DSN]** タブを選択してシステム全体にわたる DSN を作成します。
   - **[追加]** をクリックします。

     ドライバのリストから **Microsoft Excel Driver** を選択し、**[完了]** をクリックします。
   - 必要なパラメータを指定し、**[OK]** をクリックしてウィンドウを閉じ、データ・ソースを作成します。

     たとえば、**[データ ソース名]** に **myExcelFile** と入力します。**[ブックの選択]** をクリックし、Excel スプレッドシート・ファイルを参照して探します。
   - **[OK]** をクリックして DSN を保存します。
6. Interactive SQL を開き、SQL Anywhere データベースに接続します。
7. 次の INPUT 文を実行して Excel スプレッドシートからデータをインポートし、t という名前の新しいテーブルに保存します。

```
INPUT USING 'dsn=myExcelFile;DSN=myExcelFile'
FROM "myData" INTO "t"
CREATE TABLE ON
```

LOAD TABLE 文を使用したデータのインポート

LOAD TABLE 文は、データベース・サーバまたはクライアント・コンピュータにあるデータを、テキスト・フォーマットや ASCII フォーマットで既存のテーブルにインポートするために使用します。

また、LOAD TABLE 文を使用すると、別のテーブルのカラムや値の式 (関数やシステム・プロシージャの結果など) からデータをインポートすることもできます。

LOAD TABLE 文はテーブルにローを追加しますが、置き換えることはしません。

LOAD TABLE 文 (WITH ROW LOGGING および WITH CONTENT LOGGING オプションなし) を使用すると、INPUT 文を使用するよりもずっと早くデータをロードできます。

LOAD TABLE 文を使用してロードされたデータに対してはトリガは起動しません。

マテリアライズド・ビューに関する考慮事項

即時ビューでは、基本となるテーブルにデータをバルク・ロードしようとするとエラーが返されます。最初にビューのデータをトランケートしてから、バルク・ロード・オペレーションを実行する必要があります。

手動ビューでは、基本となるテーブルにデータをバルク・ロードできます。ただし、ビュー内のデータは次回のリフレッシュ時までは古いままです。

テーブルへのバルク・ロード・オペレーション (LOAD TABLE など) を実行する際は、先に従属するマテリアライズド・ビューのデータをトランケートすることを検討してください。データをロードしたら、ビューをリフレッシュします。「TRUNCATE 文」『SQL Anywhere サーバ - SQL リファレンス』と「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

テキスト・インデックスに関する考慮事項

即時テキスト・インデックスでは、基本となるテーブルで LOAD TABLE などのバルク・ロード・オペレーションを実行してからテキスト・インデックスを更新すると、自動更新にもかかわらず時間がかかる場合があります。手動テキスト・インデックスでは、リフレッシュにも時間がかかる場合があります。

テーブルへのバルク・ロード・オペレーション (LOAD TABLE など) を実行する際は、先に従属するテキスト・インデックスを削除することを検討してください。データをロードしたら、テキスト・インデックスを再作成します。「DROP TEXT INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』と「CREATE TEXT INDEX 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・リカバリと同期の考慮事項

デフォルトでは、データをファイルからロードする場合 (LOAD TABLE table-name FROM filename; など)、LOAD TABLE 文のみがトランザクション・ログに記録され、ロードされる実際のデータのローは記録されません。このため、元のロード・ファイルの変更、移動、削除後にトランザクション・ログを使用してデータベースのリカバリを行おうとすると、問題が発生します。また、同期やレプリケーションを行うデータベースが新しいデータを取得できません。

リカバリや同期の考慮事項に対応するため、LOAD TABLE 文では 2 つのロギング・オプションを使用できます。WITH ROW LOGGING を使用すると、ロードされたローごとに、トランザクション・ログに INSERT 文を作成します。WITH CONTENT LOGGING を使用すると、ロードされたローをグループ化してチャンクにし、チャンクをトランザクション・ログに記録します。これらのオプションにより、ロード・データのソースがなくなっても、ロード操作を繰り返すことができます。「LOAD TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

データベースのミラーリングの考慮事項

データベースでミラーリングを行っている場合は、LOAD TABLE 文の使用に注意してください。たとえば、ファイルからデータをロードする場合、ミラー・サーバでファイルをロードできるか

どうかや、ミラー・データベースがロードを処理するまでにロード元のソースのデータが変更されるかどうかに注意してください。これらのいずれかのリスクが存在する場合は、LOAD TABLE 文のロギング・レベルとして WITH ROW LOGGING または WITH CONTENT LOGGING のいずれかを指定してください。これにより、ミラー・データベースにロードされるデータが、ミラーリングされているデータベースにロードされたデータと同一になります。「LOAD 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

- 「クライアント・コンピュータ上のデータへのアクセス」 793 ページ
- 「データベース・ミラーリングの概要」『SQL Anywhere サーバ - データベース管理』
- 「INPUT 文 [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』
- 「LOAD TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』

INSERT 文を使用したデータのインポート

INSERT 文は、データベースにローを追加するために使用します。INSERT 文にはインポート先テーブルにインポートするデータが含まれているため、この文は対話型入力と見なされます。リモート・データ・アクセスに INSERT 文を使用して、ファイルではなく別のデータベースからデータをインポートすることもできます。

次の場合は、INSERT 文を使用してデータをインポートします。

- 1 つのテーブルに少量のデータをインポートする場合
- ファイル・フォーマットが柔軟な場合
- ファイルではなく外部データベースからリモート・データをインポートする場合

INSERT 文では、ON EXISTING 句を使用して、挿入するローがすでに挿入先のテーブルに存在する場合に実行するアクションを指定できます。ただし、ON EXISTING 条件を満たすローが数多く存在する可能性がある場合は、代わりに MERGE 文を使用してください。MERGE 文を使用すると、一致するローに対して実行するアクションをより詳細に制御できます。また、一致を定義するためのより高度な構文も使用できます。「MERGE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

マテリアライズド・ビューに関する考慮事項

即時ビューでは、基本となるテーブルにデータをバルク・ロードしようとするとエラーが返されます。最初にビューのデータをトランケートしてから、バルク・ロード・オペレーションを実行する必要があります。

手動ビューでは、基本となるテーブルにデータをバルク・ロードできます。ただし、ビュー内のデータは次回のリフレッシュ時までは古いままです。

テーブルへのバルク・ロード・オペレーション (INSERT など) を実行する際は、先に従属するマテリアライズド・ビューのデータをトランケートすることを検討してください。データをロードしたら、ビューをリフレッシュします。「TRUNCATE 文」『SQL Anywhere サーバ - SQL リファレンス』と「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

テキスト・インデックスに関する考慮事項

即時テキスト・インデックスでは、基本となるテーブルで INSERT などのバルク・ロード・オペレーションを実行してからテキスト・インデックスを更新すると、自動更新にもかかわらず時間がかかる場合があります。手動テキスト・インデックスでは、リフレッシュにも時間がかかる場合があります。

テーブルへのバルク・ロード・オペレーション (INSERT など) を実行する際は、先に従属するテキスト・インデックスを削除することを検討してください。データをロードしたら、テキスト・インデックスを再作成します。「DROP TEXT INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』と「CREATE TEXT INDEX 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。

♦ 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)。

次の例では、SQL Anywhere サンプル・データベースの Departments テーブルにデータが追加されます。

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。

参照

- 「トランザクション・ログ」『SQL Anywhere サーバ - データベース管理』
- 「INSERT 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「LOAD TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「INPUT 文 [Interactive SQL]」『SQL Anywhere サーバ - SQL リファレンス』

MERGE 文を使用したデータのインポート

MERGE 文は、更新操作を実行し、大量のテーブル・データを更新するために使用します。データのマージ時に、ソース・データのローがターゲット・データのローに一致する場合または一致しない場合に実行するアクションを指定できます。

マージ動作の定義

次に示すのは、説明のために簡略化した MERGE 文の構文です。MERGE 文の完全な構文については、「MERGE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

MERGE INTO *target-object*
USING *source-object*
ON *merge-search-condition*
{ **WHEN MATCHED** | **WHEN NOT MATCHED** } [...]

データベースによってマージ操作が実行されると、*source-object* のローと *target-object* のローが比較され、ON 句に含まれる定義に基づいて一致する行または一致しない行が検索されます。*merge-search-condition* が true になるローが少なくとも 1 つ *target-table* に存在する場合、*source-object* のローは一致と見なされます。

source-object は、ベース・テーブル、ビュー、マテリアライズド・ビュー、派生テーブル、プロシージャの結果のいずれかです。*target-object* には、これらのオブジェクトのうち、マテリアライズド・ビューとプロシージャ以外の任意のオブジェクトを指定できます。これらのオブジェクト型に関するさらなる制限については、「MERGE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

ANSI SQL/2003 標準では、マージ操作中に *target-object* のローを *source-object* の複数のローで更新することは許可されません。

source-object のローが一致または不一致と見なされると、それぞれ一致の場合と不一致の場合の WHEN 句 (WHEN MATCHED または WHEN NOT MATCHED) に対して評価されます。WHEN MATCHED 句では、*target-object* のローに対して実行するアクションを定義します (たとえば、WHEN MATCHING ... UPDATE は、*target-object* のローを更新するよう指定します)。WHEN NOT MATCHED 句では、*source-object* の一致しないローを使用して、*target-object* に対して実行するアクションを定義します。

WHEN 句は無制限に指定でき、指定した順序で処理されます。また、WHEN 句内で AND 句を使用し、ローのサブセットに対するアクションを指定することもできます。たとえば、次の WHEN 句では、一致したローの Quantity カラムの値に応じて、異なるアクションを実行するよう定義しています。

```
WHEN MATCHED AND myTargetTable.Quantity<=500 THEN SKIP
WHEN MATCHED AND myTargetTable.Quantity>500 THEN UPDATE SET myTargetTable.Quantity=500
```

マージ操作における分岐

一致するローおよび一致しないローをアクションごとにグループ化することを「分岐化」と呼び、各グループを「分岐」と呼びます。「分岐」は、単一の WHEN MATCHED 句または WHEN NOT MATCHED 句と同等です。たとえば、ある分岐に *source-object* の一致しないローのセットが含まれている場合は、それらのローを挿入する必要があります。分岐アクションの実行は、すべての分岐アクティビティを完了してから (*source-object* のすべてのローを評価してから) 開始されます。データベース・サーバは、WHEN 句が指定された順序に従って、分岐アクションの実行を開始します。

source-object の一致しないロー、または *source-object* と *target-object* の一致するローのペアが分岐に入ると、後続の分岐に対して評価されません。したがって、WHEN 句を指定する順序は重要です。

一致または不一致と見なされる *source-object* のローのうち、いずれの分岐にも属さない (つまり、いずれの WHEN 句も満たさない) ものは無視されます。これは、WHEN 句に AND 句が含まれていて、ローがいずれの AND 句の条件も満たさない場合に発生します。この場合、アクションが定義されていないため、ローは無視されます。

データを変更するアクションは、個々の INSERT、UPDATE、DELETE 文としてトランザクション・ログに記録されます。

ターゲット・テーブルに定義されたトリガ

通常、マージ操作中に各 INSERT、UPDATE、DELETE 文を実行すると、トリガが起動されます。たとえば、UPDATE アクションが定義された分岐の処理時に、データベース・サーバは次の内容を実行します。

1. すべての BEFORE UPDATE トリガを起動する
2. ローの候補セットに対して UPDATE 文を実行すると同時に、すべてのロー・レベルの UPDATE トリガを起動する
3. AFTER UPDATE トリガを起動する

target-table に対してトリガを起動すると、別の分岐で更新されるローに影響が及ぶ場合、マージ操作で競合が発生する可能性があります。たとえば、ロー B を削除するトリガを起動するアクションをロー A に対して実行するとします。しかし、ロー B には独自のアクションが定義されており、まだ実行されていません。ローに対してアクションを実行できないと、マージ操作は失敗し、すべての変更がロールバックされ、エラーが返されます。

複数のトリガ・アクションが定義されたトリガは、同じトリガに各トリガ・アクションを 1 つずつ指定したものと見なされます (つまり、各トリガに 1 つのトリガ・アクションを指定して、別個のトリガを定義したのと同等になります)。

即時マテリアライズド・ビューに関する考慮事項

MERGE 文によって多数のローを更新すると、データベース・サーバのパフォーマンスに影響する場合があります。多数のローを更新する場合は、従属する即時マテリアライズド・ビューのデータをトランケートしてから、テーブルで MERGE 文を実行することを検討してください。MERGE 文を実行したら、REFRESH MATERIALIZED VIEW 文を実行します。「REFRESH MATERIALIZED VIEW 文」『SQL Anywhere サーバ - SQL リファレンス』と「TRUNCATE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

テキスト・インデックスに関する考慮事項

MERGE 文によって多数のローを更新すると、データベース・サーバのパフォーマンスに影響する場合があります。テーブルで MERGE 文を実行する際は、先に従属するテキスト・インデックスを削除することを検討してください。MERGE 文を実行したら、テキスト・インデックスを再作成します。「DROP TEXT INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』と「CREATE TEXT INDEX 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

例 1

ジャケットとセーターを販売する小さい会社を経営しているとします。ジャケットの素材の価格が 5% 上昇したため、それに合わせて価格を調整したいとします。次の CREATE TABLE 文を使用して、販売するジャケットとセーターの現在の価格情報を保持する myProducts という小さいテーブルを作成します。その後の INSERT 文で、myProducts にデータを入力します。

```
CREATE TABLE myProducts (
  product_id    NUMERIC(10),
  product_name  CHAR(20),
  product_size  CHAR(20),
  product_price NUMERIC(14,2));
INSERT INTO myProducts VALUES (1, 'Jacket', 'Small', 29.99);
INSERT INTO myProducts VALUES (2, 'Jacket', 'Medium', 29.99);
INSERT INTO myProducts VALUES (3, 'Jacket', 'Large', 39.99);
INSERT INTO myProducts VALUES (4, 'Sweater', 'Small', 18.99);
INSERT INTO myProducts VALUES (5, 'Sweater', 'Medium', 18.99);
INSERT INTO myProducts VALUES (6, 'Sweater', 'Large', 19.99);
SELECT * FROM myProducts;
```

product_id	product_name	product_size	product_price
1	Jacket	Small	29.99
2	Jacket	Medium	29.99
3	Jacket	Large	39.99
4	Sweater	Small	18.99
5	Sweater	Medium	18.99
6	Sweater	Large	19.99

さらに、次の文を使用して、ジャケットの価格変更に関する情報を保持する myPrices という別のテーブルを作成します。マージ操作を実行する前に myPrices テーブルの内容を確認できるように、末尾に SELECT 文を追加します。

```
CREATE TABLE myPrices (
  product_id    NUMERIC(10),
  product_name  CHAR(20),
  product_size  CHAR(20),
  product_price NUMERIC(14,2),
  new_price     NUMERIC(14,2));
INSERT INTO myPrices (product_id) VALUES (1);
INSERT INTO myPrices (product_id) VALUES (2);
INSERT INTO myPrices (product_id) VALUES (3);
INSERT INTO myPrices (product_id) VALUES (4);
INSERT INTO myPrices (product_id) VALUES (5);
INSERT INTO myPrices (product_id) VALUES (6);
COMMIT;
SELECT * FROM myPrices;
```

product_id	product_name	product_size	product_price	new_price
1	(NULL)	(NULL)	(NULL)	(NULL)

product_id	product_name	product_size	product_price	new_price
2	(NULL)	(NULL)	(NULL)	(NULL)
3	(NULL)	(NULL)	(NULL)	(NULL)
4	(NULL)	(NULL)	(NULL)	(NULL)
5	(NULL)	(NULL)	(NULL)	(NULL)
6	(NULL)	(NULL)	(NULL)	(NULL)

次の MERGE 文を使用して、myProducts テーブルのデータを myPrices テーブルにマージします。*source-object* は、product_name が Jacket のローのみを含むようフィルタリングされた派生テーブルです。また、ON 句では、*target-object* と *source-object* の product_id カラムの値が一致する場合にローが一致するよう指定しています。

```
MERGE INTO myPrices p
USING ( SELECT
    product_id,
    product_name,
    product_size,
    product_price
  FROM myProducts
  WHERE product_name='Jacket') pp
ON (p.product_id = pp.product_id)
WHEN MATCHED THEN
  UPDATE SET
    p.product_id=pp.product_id,
    p.product_name=pp.product_name,
    p.product_size=pp.product_size,
    p.product_price=pp.product_price,
    p.new_price=pp.product_price * 1.05;
SELECT * FROM myPrices;
```

product_id	product_name	product_size	product_price	new_price
1	Jacket	Small	29.99	31.49
2	Jacket	Medium	29.99	31.49
3	Jacket	Large	39.99	41.99
4	(NULL)	(NULL)	(NULL)	(NULL)
5	(NULL)	(NULL)	(NULL)	(NULL)
6	(NULL)	(NULL)	(NULL)	(NULL)

product_id が 4、5、6 のカラムの値は、NULL のままになります。これは、myProducts テーブルで製品が (product_name='Jacket') であるいずれのローとも一致しなかったためです。

例 2

次の例では、myTargetTable のプライマリ・キー値を使用してローを一致させ、mySourceTable テーブルと myTargetTable テーブルのローをマージします。mySourceTable のローに、myTargetTable のプライマリ・キー・カラムと同じ値が含まれている場合、ローは一致していると見なされます。

```
MERGE INTO myTargetTable
  USING mySourceTable ON PRIMARY KEY
  WHEN NOT MATCHED THEN INSERT
  WHEN MATCHED THEN UPDATE;
```

WHEN NOT MATCHED THEN INSERT 句は、mySourceTable にはあって myTargetTable にはないローを、myTargetTable に追加することを指定します。WHEN MATCHED THEN UPDATE 句は、myTargetTable の一致するローを mySourceTable の値に更新することを指定します。

次の構文は前述の構文と同等です。myTargetTable にはカラム (I1, I2, .. I*n*) があり、プライマリ・キーがカラム (I1, I2) に定義されていることを前提としています。mySourceTable にはカラム (U1, U2, .. U*n*) があります。

```
MERGE INTO myTargetTable ( I1, I2, .. ., In )
  USING mySourceTable ON myTargetTable.I1 = mySourceTable.U1
    AND myTargetTable.I2 = mySourceTable.U2
  WHEN NOT MATCHED
    THEN INSERT ( I1, I2, .. In )
      VALUES ( mySourceTable.U1, mySourceTable.U2, ..., mySourceTable.Un )
  WHEN MATCHED
    THEN UPDATE SET
    myTargetTable.I1 = mySourceTable.U1,
    myTargetTable.I2 = mySourceTable.U2,
    ...
    myTargetTable.In = mySourceTable.Un;
```

RAISERROR アクションの使用

一致または不一致のアクションに指定できるアクションの 1 つが、RAISERROR です。RAISERROR を使用すると、WHEN 句の条件を満たした場合に、マージ操作を失敗させることができます。

RAISERROR を指定すると、データベース・サーバはデフォルトで SQLSTATE 23510 および SQLCODE -1254 を返します。必要に応じて、RAISERROR キーワードの後に *error_number* パラメータを指定し、返される SQLCODE をカスタマイズできます。「MERGE 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

カスタム SQLCODE の指定は、後でエラーの発生した状況を特定する場合に便利です。

カスタム SQLCODE には 17000 よりも大きい正の整数を指定してください。数または変数のいずれとしても指定できます。

次の文で、SQLCODE のカスタマイズによって返される内容にどのような影響が出るかを簡単に示します。

次のように targetTable を作成します。

```
CREATE TABLE targetTable( c1 int );
INSERT INTO targetTable VALUES( 1 );
COMMIT;
```

次の文は、SQLSTATE = '23510' および SQLCODE = -1254 のエラーを返します。

```
MERGE INTO targetTable
  USING (SELECT 1 c1 ) AS sourceData
  ON targetTable.c1 = sourceData.c1
  WHEN MATCHED THEN RAISERROR;
SELECT sqlstate, sqlcode;
```

次の文は、SQLSTATE = '23510' および SQLCODE = -17001 のエラーを返します。

```
MERGE INTO targetTable
  USING (SELECT 1 c1 ) AS sourceData
  ON targetTable.c1 = sourceData.c1
  WHEN MATCHED THEN RAISERROR 17001
  WHEN NOT MATCHED THEN RAISERROR 17002;
SELECT sqlstate, sqlcode;
```

次の文は、SQLSTATE = '23510' および SQLCODE = -17002 のエラーを返します。

```
MERGE INTO targetTable
  USING (SELECT 2 c1 ) AS sourceData
  ON targetTable.c1 = sourceData.c1
  WHEN MATCHED THEN RAISERROR 17001
  WHEN NOT MATCHED THEN RAISERROR 17002;
SELECT sqlstate, sqlcode;
```

プロキシ・テーブルを使用したデータのインポート

プロキシ・テーブルは、メタデータを含むローカル・テーブルです。リモート・データベース・サーバのテーブルに、ローカル・テーブルであるかのようにアクセスするときに使用します。これによって、データを直接インポートできます。

次の場合は、プロキシ・テーブルを使用してデータをインポートします。

- リモート・データにアクセスできる場合
- データを別のデータベースから直接インポートする場合

データベースに対する影響

プロキシ・テーブルを使用すると、変更内容はトランザクション・ログに記録されます。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」 815 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、無効な日付や数字が含まれている可能性があります。conversion_error データベース・オプションを使用すると、変換エラーを無視し、無効な値を NULL 値に変換できます。

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

♦ **テーブルをインポートするには、次の手順に従います。**

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. Interactive SQL の **[データ]** - **[インポート]** を選択します。
3. **[テキスト・ファイル]** を選択して **[次へ]** をクリックします。
4. **[ファイル名]** フィールドで **[参照]** をクリックしてファイルを追加します。
5. **[既存のテーブル]** を選択します。
6. **[次へ]** をクリックします。
7. ASCII ファイルの場合は、ASCII ファイルを読み込む方法を指定して **[次へ]** をクリックします。
8. **[インポート]** をクリックします。
9. **[閉じる]** をクリックします。

♦ **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) );
```

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トリガの起動や参照整合性に関わるアクションの実行を行うことはありません。

参照

- 「CREATE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「LOAD TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』

インポート用のテーブル構造

ソース・データの構造は、インポート先テーブル自体の構造と一致する必要はありません。たとえば、カラムのデータ型が異なる、順序が異なる、またはロード先のテーブルのカラム数を超えるインポート・データの値があるなどです。

テーブルまたはデータの並べ替え

インポートするデータの構造がインポート先テーブルの構造と一致しないことがわかっている場合は、次の操作を行うことができます。

- LOAD TABLE 文でロードするカラムの名前リストを入力します。
- INSERT 文の一種とグローバル・テンポラリ・テーブルを使用して、インポート・データをテーブルに合うように並べ替えることができます。
- INPUT 文を使用して、カラムの特定のセットまたは順序を指定できます。

カラムに NULL 値を入力できるようにする

インポート中のファイルにテーブルのカラムのサブセットへのデータがある場合、またはカラムの順序が異なる場合は、LOAD TABLE 文の DEFAULTS オプションを使用して、ブランクを埋めて一致しないテーブル構造をマージすることもできます。

- DEFAULTS オプションが OFF の場合は、カラム・リストにないカラムすべてに NULL が割り当てられます。DEFAULTS オプションが OFF で、NULL 入力不可のカラムがカラム・リストから省かれている場合は、データベース・サーバは、空の文字列をカラムの型に変換しようとします。
- DEFAULTS オプションが ON で、カラムにデフォルト値が入っている場合は、その値が使用されます。

たとえば、Customers テーブルの City カラムにデフォルト値を定義してから、次のような LOAD TABLE 文を使用して new_customers.txt という架空のファイルから新しいローを Customers テーブルにロードできます。

```
ALTER TABLE Customers
ALTER City DEFAULT 'Waterloo';
```

```
LOAD TABLE Customers ( Surname, GivenName, Street, State, Phone )
FROM 'new_customers.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. **[SQL 文]** ウィンドウ枠で、入力ファイルと構造が一致するグローバル・テンポラリ・テーブルを作成します。

   CREATE TABLE 文を使用して、グローバル・テンポラリ・テーブルを作成できます。

2. LOAD TABLE 文を使用して、作成したグローバル・テンポラリ・テーブルにデータをロードします。

   データベース接続を閉じると、グローバル・テンポラリ・テーブル内のデータは消去されます。ただし、テーブル定義は残ります。この定義は、次にデータベースに接続するときに使用できます。

3. INSERT 文と SELECT 句を使用し、テンポラリ・テーブルからデータを抽出して要約し、データベースの 1 つまたは複数の永久テーブルにコピーします。

参照

- 「CREATE TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「LOAD TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』

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」 564 ページを参照してください。

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。データのエクスポートは、データベースの大部分を共有する必要がある場合、または特定の基準に従ってデータベースの一部を抽出する必要がある場合に便利です。SQL Anywhere を使用して、次の作業を実行できます。

- 個々のテーブル、クエリ結果、またはテーブル・スキーマをエクスポートする
- 複数のテーブルを連続的にエクスポートできるようにするために、エクスポートを自動化するスクリプトを作成する
- 多くの異なるファイル・フォーマットにエクスポートする
- クライアント・コンピュータにあるファイルにデータをエクスポートする
- BCP FORMAT 句を使用して、SQL Anywhere と Adaptive Server Enterprise の間でファイルをエクスポートする

データをエクスポートする前に、所有しているリソースの種類、およびデータベースからエクスポートする情報の種類を判断します。

データベース全体をエクスポートする場合は、パフォーマンスを考慮して、データをエクスポートするのではなくデータベースをアンロードしてください。「データベースの再構築」 796 ページを参照してください。

参照

- 「UNLOAD 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「OUTPUT 文 [Interactive SQL]」 『SQL Anywhere サーバ - SQL リファレンス』
- 「バルク・オペレーションのパフォーマンスの側面」 762 ページ
- 「OUTPUT 文を使用した NULL の出力」 790 ページ
- 「クライアント・コンピュータ上のデータへのアクセス」 793 ページ

エクスポート・ウィザードを使用したデータのエクスポート

エクスポート・ウィザードは、特定のフォーマットのクエリ結果をファイルやデータベースにエクスポートするために使用します。

♦ 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. クエリを実行します。
2. Interactive SQL で、**[データ] - [エクスポート]** を選択します。
3. **エクスポート・ウィザード**の指示に従います。

♦ **Interactive SQL を使用して結果セット・データを Ultra Light データベースに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. **[データ]** - **[エクスポート]** を選択します。
3. **[データベース]** をクリックします。
4. **[データベース・タイプ]** リストで、**[Ultra Light]** を選択します。
5. **[ユーザ ID]** フィールドに **dba** と入力します。
6. **[パスワード]** フィールドに **sql** と入力します。
7. **[データベース]** タブをクリックします。
8. **[データベース・ファイル]** フィールドに *C:¥Documents and Settings¥All Users¥Documents¥SQL Anywhere 11¥Samples¥UltraLite¥CustDB¥custdb.udb* と入力します。
9. **[次へ]** をクリックします。
10. **[新しいテーブル]** をクリックします。
11. **[所有者]** リストで、**dba** を選択します。
12. **[テーブル名]** フィールドに **NewTable** と入力します。
13. **[エクスポート]** をクリックします。
14. **[SQL]** - **[前の SQL]** を選択します。

    インポート・ウィザードによって作成および使用された IMPORT 文が **[SQL 文]** ウィンドウ枠に表示されます。

    ```
    -- Generated by the Export Wizard
    OUTPUT USING 'driver=UltraLite 11;UID=dba;PWD=sql;
    DBF=C:¥Documents and Settings¥All Users¥Documents¥SQL Anywhere 11¥Samples¥UltraLite
    ¥CustDB¥custdb.udb'
    INTO "dba"."NewTable"
    CREATE TABLE ON
    ```

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することができます。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、Interactive SQL の **[オプション]** ウィンドウ (Interactive SQL で **[ツール]** - **[オプション]** を選択) 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 TEXT;
```

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、*mydatabase.db* という名前の SQL Anywhere データベースの新しいテーブルにエクスポートします。

```
SELECT *
FROM Employees;
OUTPUT USING 'driver=SQL Anywhere 11;UID=dba;PWD=sql;DBF=C:¥Tobedeleted
¥mydatabase.db;CON=""'
INTO "dba"."newcustomers"
CREATE TABLE ON
```

UNLOAD TABLE 文を使用したデータのエクスポート

UNLOAD TABLE 文を使用すると、テキスト・フォーマットだけでデータを効率的にエクスポートできます。UNLOAD TABLE 文では、1 行に 1 ローずつ、値をカンマで区切ってエクスポートします。再ロードを速くするために、データはプライマリ・キー値の順にエクスポートされます。

次の場合は、UNLOAD TABLE 文を使用します。

- テキスト・フォーマットでテーブル全体をエクスポートする場合
- データベース・パフォーマンスを考慮する場合
- クライアント・コンピュータにあるファイルにデータをエクスポートする場合

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。

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。

参照

- 「クライアント・コンピュータ上のデータへのアクセス」 793 ページ
- 「UNLOAD 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「OUTPUT 文 [Interactive SQL]」 『SQL Anywhere サーバ - SQL リファレンス』

UNLOAD 文を使用したデータのエクスポート

UNLOAD 文は、クエリ結果をファイルにエクスポートするという点で OUTPUT 文に似ています。ただし、UNLOAD 文はテキスト・フォーマットでより効率的にデータをエクスポートします。UNLOAD 文は、1 行に 1 ローずつ、値をカンマで区切ってエクスポートします。

次の場合は、UNLOAD 文を使用してデータをアンロードします。

- パフォーマンスが問題で、クエリ結果をエクスポートする場合
- テキスト・フォーマットで出力データを格納する場合
- アプリケーションにエクスポート・コマンドを埋め込む場合
- クライアント・コンピュータにあるファイルにデータをエクスポートする場合

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。

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。

参照

- 「クライアント・コンピュータ上のデータへのアクセス」 793 ページ
- 「UNLOAD 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「OUTPUT 文 [Interactive SQL]」 『SQL Anywhere サーバ - SQL リファレンス』

dbunload ユーティリティを使用したデータのエクスポート

dbunload ユーティリティは、1 つ、複数、またはすべてのデータベース・テーブルをエクスポートするために使用します。テーブル・データとテーブル・スキーマをエクスポートできます。データベースのテーブルを配置し直す場合にも dbunload ユーティリティを使用でき、必要なコマンド・ファイルを作成して必要に応じて修正します。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。

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。

データベース・アンロード・ウィザードを使用したデータのエクスポート

データベース・アンロード・ウィザードは、既存のデータベースを新しいデータベースにアンロードするために使用します。

データベース・アンロード・ウィザードを使用してデータベースをアンロードする際、データベース内のすべてのオブジェクトをアンロードするのか、またはデータベースからテーブルのサブセットのみをアンロードするのかを選択できます。**[所有者フィルタの設定]** ウィンドウ内で選択したユーザ用のテーブルのみが、**データベース・アンロード・ウィザード**に表示されます。特定のデータベース・ユーザに属するテーブルを表示させたい場合、アンロードするデータベースを右クリックし、**[所有者フィルタの設定]** を選択してから、表示されるウィンドウで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。

データベース・アンロード・ウィザードを使用すると、データベース全体をカンマ区切りのテキスト・フォーマットでアンロードし、データベース全体の再作成に必要な 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**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] - [SQL Anywhere 11] - [データベースのアンロード]** を選択します。
2. **データベース・アンロード・ウィザード**の指示に従います。

[データのアンロード] ウィンドウを使用したデータのエクスポート

Sybase Central の **[データのアンロード]** ウィンドウを使用して、データベース内の 1 つ以上のテーブルをアンロードできます。この機能は、**データベース・アンロード・ウィザード**またはアンロード・ユーティリティ (dbunload) でも使用できますが、このウィンドウを使用すると、**データベース・アンロード・ウィザード**全体を実行する代わりに 1 ステップでテーブルをアンロードできます。

♦ [データのアンロード] ウィンドウ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. 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. **[テーブル]** をダブルクリックします。
3. データをエクスポートするテーブルを右クリックして、**[データのアンロード]** を選択します。
4. **[データのアンロード]** ウィンドウの入力を完了します。**[OK]** をクリックします。

クエリ結果のエクスポート

[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 句は、SQL Anywhere と Adaptive Server Enterprise 間でファイルのインポートとエクスポートを実行するために使用します。詳細については、「Adaptive Server Enterprise の互換性」 814 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]** - **[実行]** を選択します。
3. **[データ]** - **[エクスポート]** を選択します。
4. 結果の出力場所を指定して **[次へ]** をクリックします。
5. テキスト、HTML、XML ファイルの場合は、**[ファイル名]** フィールドにファイル名を入力して **[エクスポート]** をクリックします。

   ODBC データベースの場合は、次の手順を実行します。

   a. データベースを選択して **[次へ]** をクリックします。

   b. データの保存場所を指定して **[エクスポート]** をクリックします。
6. **[閉じる]** をクリックします。

♦ 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. クエリ結果をエクスポートして、別のファイルに結果を追加するには、APPEND 句を使用します。

   ```
   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には、VERBOSE 句を使用します。

   ```
   SELECT * FROM Employees;
   OUTPUT TO 'employees.txt'
   VERBOSE;
   ```

4. **[SQL] - [実行]** を選択します。

   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。

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** と入力します。

OUTPUT 文の APPEND 句と VERBOSE 句は、Interactive SQL の以前のバージョンの演算子 >#、>>#、>&、>>& と同じです。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] - [実行]** を選択します。

   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 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)';
  ```

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。

♦ NULL 値の代わりに表示される値を変更するには、次の手順に従います (Interactive SQL の [結果] ウィンドウ枠の場合)。

1. **[ツール] - [オプション]** を選択します。
2. **[SQL Anywhere]** をクリックします。
3. **[結果]** タブをクリックします。
4. **[NULL 値の代替文字]** フィールドで、**Value** と入力します。
5. **[OK]** をクリックします。

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] - [SQL Anywhere 11] - [データベースのアンロード]** を選択します。
2. **データベース・アンロード・ウィザード**の指示に従います。

♦ 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)。

- dunload ユーティリティを実行し、-c オプションを使用して接続パラメータを指定します。

  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* にアンロードするには、次の手順に従います。

  ```
  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。

  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

テーブル内のすべてのデータを選択して、クエリ結果をエクスポートすることにより、テーブルをエクスポートできます。「クエリ結果のエクスポート」 788 ページを参照してください。

同じ手順を使用して、ビューもエクスポートできます。

♦ **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)。**

● 次のコマンドを実行します。

```
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。

参照

● 「クエリ結果のエクスポート」 788 ページ
● 「アンロード・ユーティリティ (dbunload)」 『SQL Anywhere サーバ - データベース管理』
● 「UNLOAD 文」 『SQL Anywhere サーバ - SQL リファレンス』

クライアント・コンピュータ上のデータへのアクセス

SQL Anywhere では、ファイルをデータベース・サーバ・コンピュータにコピーすることなく、SQL の文と関数を使用して、クライアント・コンピュータ上のファイルからデータをロードしたり、ファイルにデータをアンロードすることができます。これを実行するために、データベース・サーバは Command Sequence 通信プロトコル (CmdSeq) ファイル・ハンドラを使用して転送を開始します。CmdSeq ファイル・ハンドラは、データベース・サーバがクライアント・コンピュータとのデータの転送を必要としているクライアント・アプリケーションから要求を受信した後、応答を送信する前に呼び出されます。このファイル・ハンドラは、クライアントの複数のファイルの同時転送およびインターリーブ転送を任意の時点でサポートしています。たとえば、クライアント・アプリケーションにより実行された文で複数のファイルの同時転送が必要な場合、データベース・サーバはそれを開始することができます。

CmdSeq ファイル・ハンドラを使用してクライアント・データの転送を実行することにより、アプリケーションは特殊なコードを使用する必要がなく、次に示す SQL コンポーネントを使用した機能をすぐに利用することができます。

- **READ_CLIENT_FILE 関数**　READ_CLIENT_FILE 関数は、クライアント・コンピュータ上の指定したファイルからデータを読み込み、ファイルの内容を表す LONG BINARY 値を返します。この関数は、BLOB が使用可能な SQL コードの任意の位置で使用できます。可能な場合は、文で実体化の実行を明示的に指定していないかぎり、READ_CLIENT_FILE 関数によって返されるデータはメモリ内で実体化されません。たとえば、LOAD TABLE 文は実体化を行わずにクライアント・ファイルからデータをストリーミングします。READ_CLIENT_FILE 関数によって返された値を接続変数に割り当てると、データベース・サーバはクライアント・ファイルの内容を取得して実体化します。「READ_CLIENT_FILE 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。
- **WRITE_CLIENT_FILE 関数**　WRITE_CLIENT_FILE 関数は、クライアント・コンピュータにある、指定したファイルにデータを書き込みます。「WRITE_CLIENT_FILE 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。
- **READCLIENTFILE 権限**　READCLIENTFILE 権限を使用すると、ユーザはクライアント・コンピュータにあるファイルを読み込むことができます。「READCLIENTFILE 権限」『SQL Anywhere サーバ - データベース管理』を参照してください。
- **WRITECLIENTFILE 権限**　WRITECLIENTFILE 権限を使用すると、ユーザはクライアント・コンピュータにあるファイルに書き込むことができます。「WRITECLIENTFILE 権限」『SQL Anywhere サーバ - データベース管理』を参照してください。
- **LOAD TABLE ... USING CLIENT FILE 句**　USING CLIENT FILE 句を使用すると、クライアント・コンピュータにあるファイル内のデータを使用してテーブルをロードできます。たとえば、LOAD TABLE ... USING CLIENT FILE 'my-file.txt'; と指定すると、クライアント・コンピュータから *my-file.txt* というファイルがロードされます。「LOAD TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
- **LOAD TABLE ... USING VALUE 句**　USING VALUE 句を使用すると、BLOB 式を値として指定できます。BLOB 式では、READ_CLIENT_FILE 関数を使用してクライアント・コンピュータ上のファイルから BLOB をロードできます。たとえば、LOAD TABLE ... USING VALUE READ_CLIENT_FILE('my-file') というように指定します。*my-file* はクライアント・コンピュー

タ上のファイルです。「LOAD TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

- **UNLOAD TABLE ... INTO CLIENT FILE 句** INTO CLIENT FILE 句を使用すると、データのアンロード先としてクライアント・コンピュータにあるファイルを指定できます。「UNLOAD 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
- **UNLOAD TABLE ... INTO VARIABLE 句** INTO VARIABLE 句を使用すると、データのアンロード先として変数を指定できます。「UNLOAD 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
- **read_client_file および write_client_file 保護機能** read_client_file および write_client_file 保護機能は、クライアント・ファイルの読み込みまたは書き込みを行う文の使用を制御します。「保護された機能の指定」『SQL Anywhere サーバ - データベース管理』と「-sf 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。

クライアント側データ・セキュリティ

SQL Anywhere は、クライアント・ファイルの転送において、データベース・サーバ・コンピュータとは異なる場所に存在することの多いクライアント・コンピュータ上のデータを不正に転送できないようにする手段を提供しています。

そのため、データベース・サーバは各文の実行元を追跡し、文がクライアント・アプリケーションから直接受信されたものかどうかを判断します。クライアントからの新しいファイルの転送を開始する際に、データベース・サーバは文の実行元に関する情報を付加します。文がクライアント・アプリケーションから直接送信された場合は、CmdSeq ファイル・ハンドラによってファイルの転送が許可されます。文がクライアント・アプリケーションから直接送信されたものではない場合、アプリケーションは検証のコールバックを登録する必要があります。コールバックが登録されないと、転送が拒否されて文が失敗し、エラーが発生します。

また、接続が正常に確立されるまで、クライアント・データの転送は許可されません。この制限により、接続文字列やログイン・プロシージャを使用した不正アクセスを防止できます。

許可されたユーザを装ってシステムへのアクセスを試みるユーザからの保護を実現するには、転送データの暗号化を行ってください。

また、SQL Anywhere では、さまざまなレベルでアクセスを制御する次のセキュリティ・メカニズムも提供しています。

- **サーバ・レベルのセキュリティ** read_client_file および write_client_file 保護機能により、サーバワイドでクライアント側の転送を無効にできます。「-sf 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。
- **アプリケーション・レベルおよび DBA レベルのセキュリティ** allow_read_client_file および allow_write_client_file データベース・オプションにより、データベース、ユーザ、接続レベルでのアクセス制御を実現します。たとえば、アプリケーションの接続後にこのデータベース・オプションを OFF に設定することにより、アプリケーションがクライアント側の転送に使用されるのを防止できます。「allow_read_client_file 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。

- **ユーザ・レベルのセキュリティ**　READCLIENTFILE および WRITECLIENTFILE 権限は、それぞれクライアント・コンピュータからのデータの読み込みまたはクライアント・コンピュータへのデータの書き込みに対して、ユーザ・レベルのアクセス制御を提供します。「READCLIENTFILE 権限」『SQL Anywhere サーバ - データベース管理』と「WRITECLIENTFILE 権限」『SQL Anywhere サーバ - データベース管理』を参照してください。

クライアント側データのロード時のリカバリの計画

トランザクション・ログから LOAD TABLE 文のリカバリが必要になった場合、データのロードに使用したクライアント・コンピュータ上のファイルは SQL Anywhere ですでに使用できないか、変更されている可能性が高いため、元のデータが使用できなくなります。このような状況が発生するのを防ぐため、ロギングがオフになっていないことを確認してください。また、データのロード時に WITH ROW LOGGING または WITH CONTENT LOGGING 句のいずれかを指定してください。これらの句を指定することにより、ロードしているデータがトランザクション・ログに記録されるため、リカバリ時にリプレイが可能になります。

WITH ROW LOGGING では、挿入された各ローがトランザクション・ログに INSERT 文として記録されます。WITH CONTENT LOGGING では、挿入されたデータが、データベース・サーバがリカバリ時に処理するためにチャンク単位でトランザクション・ログに記録されます。どちらの方法も、リカバリ時にクライアント側データをロードできるようにするには適しています。ただし、同期を行っているデータベースにデータをロードする場合は、WITH CONTENT LOGGING は使用できません。

次のいずれかの LOAD TABLE 文を指定する際に、ロギング・レベルが指定されていない場合は、WITH CONTENT LOGGING がデフォルトの動作となります。

- LOAD TABLE ...USING CLIENT FILE *client-filename-expression*
- LOAD TABLE ...USING VALUE *value-expression*
- LOAD TABLE ...USING COLUMN *column-expression*

ロード操作中にロードしたデータをトランザクション・ログに記録する方法の詳細については、「LOAD TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

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 11 - 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。

インポートとエクスポートの場合、データの送信先はデータベース内またはデータベース外になります。インポートでは、データはデータベースに読み込まれます。エクスポートでは、データはデータベースから書き出されます。情報を、SQL Anywhere 以外の別のデータベースから受け取ったり、別のデータベースに送信したりすることがよくあります。

暗号化オプション -ek、-ep または -et を指定する場合は、*reload.sql* ファイルの LOAD TABLE 文に暗号化キーを含めます。ハードコーディングされたキーはセキュリティの低下を招くため、*reload.sql* ファイルのパラメータには暗号化キーが指定されます。Interactive SQL 内で *reload.sql* ファイルを実行する際に、パラメータとして暗号化キーを指定してください。READ 文にキーを指定していない場合、Interactive SQL でキーの入力を要求するプロンプトが表示されます。「Interactive SQL ユーティリティ (dbisql)」 『SQL Anywhere サーバ - データベース管理』を参照してください。

ロードとアンロードでは、SQL Anywhere のデータベースからデータとスキーマを取り出し、それらを別の SQL Anywhere データベースに入れ直します。アンロード・プロシージャでは、データ・ファイルと、テーブルを正確に再作成するために必要なテーブル定義を含む *reload.sql* ファイルが生成されます。*reload.sql* スクリプトを実行すると、テーブルが再作成され、そこに元のデータがロードされます。

データベースの再構築には時間がかかる可能性があり、大量のディスク領域が必要になることがあります。また、データベースのアンロードと再ロード中は、そのデータベースを使用できません。このため、明確な目的がないかぎり、運用環境でデータベースを再構築しないでください。

特定の SQL Anywhere データベースから別の SQL Anywhere データベースへ

再構築では、通常 SQL Anywhere データベースからデータをコピーし、それを別の SQL Anywhere データベースに再ロードします。アンロードと再ロードは関連しています。通常はどちらか一方ではなく、両方を行います。

再構築とエクスポート

再構築がエクスポートとは異なる点は、再構築では、データの他にテーブル定義とスキーマのエクスポートとインポートが行われることです。再構築処理のアンロード部分では、テキスト・フォーマットのデータ・ファイルと、テーブルとその他の定義が含まれる *reload.sql* ファイルが生成されます。*reload.sql* スクリプトを実行すると、テーブルが再作成され、そこにデータがロードされます。

詳細については、「内部アンロードと外部アンロード、内部再ロードと外部再ロード」『SQL Anywhere サーバ - データベース管理』を参照してください。

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」 805 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。

参照

- 「データベース再構築時のダウン時間の最短化」 803 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」 799 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」 798 ページ
- 「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』
- 「手動ビューのリフレッシュ」 62 ページ

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。新しい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を判断する方法については、「SQL Anywhere 11 へのアップグレード」『SQL Anywhere 11 - 変更点とアップグレード』を参照してください。

  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 より前からアップグレードする場合は、データベース・ファイルを再構築する必要があります。バージョン 10.0.0 以降からアップグレードする場合は、アップグレード・ユーティリティを使用するか、データベースを再構築します。

デ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 11 - 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 11 - 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』

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. 次のいずれかのオプションを指定して 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 ユーティリティには、アンロード作業をチューニングするための追加オプションがあります。また、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

この項の内容は、SQL Anywhere の Mobile Link クライアント (dbmlsync を使用するクライアント)、SQL Remote、および 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に参加しているときは、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。2 つ目の方法では、手動で同じタスクを実行します。

すべてのサブスクリプションを同期してから、Mobile Link 同期に参加するデータベースを再構築する必要があります。

♦ 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. 次 dbunload のコマンドを実行して、データベースを再構築します。

   ```
   dbunload -c connection-string -ar directory
   ```

   connection-string は 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 ユーティリティには、アンロード作業をチューニングするための追加オプションがあります。また、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」 796 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 0000698242 -il -ir -is database-name.db
   ```
9. Message Agent を実行するときは、コマンド・ラインに元のオフライン・ディレクトリのパスを入力します。
10. データベースを起動します。これで、再ロードされたデータベースにアクセスできます。

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テキスト・フォーマットでアンロードし、データベース全体の再作成に必要な 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**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」 798 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」 799 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、特定の文字コードでデータを効率的にエクスポートできます。テキスト・フォーマットでデータをエクスポートする場合は、UNLOAD TABLE 文を使用してデータベースを再構築してください。

データベースに対する影響

UNLOAD TABLE 文は、テーブル全体に排他ロックを配置します。

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。

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指定したテーブルの再ロードに必要な文は、現在のローカル・ディレクトリの *reload.sql* に記述されています。

♦ **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)。**

- dbunload コマンドを実行し、-c オプションを使用して接続パラメータを指定し、-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かどうかを指定します。

  たとえば、Employees テーブルからデータをエクスポートするには、次のコマンドを実行します。

  ```
  dbunload -c "DBN=demo;UID=DBA;PWD=sql" -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)。**

- dbunload コマンドを実行し、-c オプションを使用して接続パラメータを指定し、-t オプションを使用してデータをエクスポートするテーブルを指定し、-n オプションを指定してスキーマのみをアンロードするかどうかを指定します。

  たとえば、Employees テーブルからスキーマ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. dbinit ユーティリティを実行して空の新しいデータベース・ファイルを作成します。

   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
   ```

2. *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
   ```

データベース再構築時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 ～ 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。

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。

- 「抽出ユーティリティ (dbxtract)」『SQL Remote』
- 「リモート・データベースの抽出」『SQL Remote』
- 「リモート・データベースの配備」『Mobile Link - クライアント管理』

SQL Anywhere へのデータベースの移行

sa_migrate システム・プロシージャまたは**データベース移行ウィザード**を使用して、次のソースからテーブルをインポートします。

- SQL Anywhere
- Ultra Light
- Sybase ASE
- IBM DB/2
- Microsoft SQL Server
- Microsoft Access
- Oracle
- MySQL
- Advantage Database Server
- リモート・サーバに接続する汎用 ODBC ドライバ

データベース移行ウィザードやシステム・プロシージャの sa_migrate セットを使用してデータを移行する前に、まず **target database** 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」 『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ための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. 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. **[ツール] - [SQL Anywhere 11] - [データベースの移行]** を選択します。
3. **[次へ]** をクリックします。
4. 対象のデータベースを選択して **[次へ]** をクリックします。
5. リモート・データベースへの接続に使用するリモート・サーバを選択し、**[次へ]** をクリックします。

   リモート・サーバ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**の指示に従います。リモート・サーバの作成の詳細については、「CREATE SERVER 文を使用して、リモート・サーバを作成します。」 820 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。

6. 移行するテーブルを選択し、**[次へ]** をクリックします。

   システム・テーブルは移行できないので、このリストにはシステム・テーブルは表示されません。

7. ターゲット・データベース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**ユーザ作成ウィザード**の指示に従います。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。

8. リモート・テーブルからデータと外部キーを移行するかどうか、また移行プロセスのために作成されたプロキシ・テーブルを保持するかどうかを選択し、**[次へ]** をクリックします。

9. **[完了]** をクリックします。

sa_migrate システム・プロシージャの使用

sa_migrate システム・プロシージャを使用して、リモート・データを移行します。テーブルや外部キーのマッピングを削除する場合は、拡張メソッドを使用します。

sa_migrate システム・プロシージャを使用したすべてのテーブルの移行

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・ユーザのすべてのテーブルの移行

1. ターゲット・データベースを作成します。「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。

2. Interactive SQL からターゲット・データベースに接続します。

3. 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。「CREATE SERVER 文を使用して、リモート・サーバを作成します。」 820 ページを参照してください。

4. 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。「外部ログインの作成」 829 ページを参照してください。

5. 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。「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。
6. **[SQL 文]** ウィンドウ枠で、sa_migrate システム・プロシージャを実行します。次に例を示します。

   ```
   CALL sa_migrate( 'local_user1', 'rmt_server1', NULL, 'remote_user1', 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 remote_user1 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 システム・プロシージャを使用した個別のテーブルの移行

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. ターゲット・データベースを作成します。「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。
2. Interactive SQL からターゲット・データベースに接続します。
3. 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。「CREATE SERVER 文を使用して、リモート・サーバを作成します。」 820 ページを参照してください。
4. 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。「外部ログインの作成」 829 ページを参照してください。
5. 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。「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。
6. sa_migrate_create_remote_table_list システム・プロシージャを実行します。次に例を示します。

   ```
   CALL sa_migrate_create_remote_table_list( 'rmt_server1',
      NULL, 'remote_user1', 'mydb' );
   ```

   リモート・サーバの名前を指定してください。

   これにより、移行するリモート・テーブルのリストが dbo.migrate_remote_table_list テーブルに設定されます。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。

7. sa_migrate_create_tables システム・プロシージャを実行します。次に例を示します。

   ```
   CALL sa_migrate_create_tables( 'local_user1' );
   ```

   このプロシージャは、dbo.migrate_remote_table_list からリモート・テーブルのリストを取り出し、リストにあるテーブルごとにプロキシ・テーブルとベース・テーブルを作成します。また、移行したテーブルのプライマリ・キー・インデックスもすべて作成します。

   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。

8. リモート・テーブルからターゲット・データベース上のベース・テーブルにデータを移行する場合は、sa_migrate_data システム・プロシージャを実行します。次に例を示します。

   次の文を実行します。

   ```
   CALL sa_migrate_data( 'local_user1' );
   ```

   このプロシージャは、各リモート・テーブルから、sa_migrate_create_tables プロシージャで作成されたベース・テーブルにデータを移行します。

   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 10 に進みます。

9. sa_migrate_create_remote_fks_list システム・プロシージャを実行します。次に例を示します。

   ```
   CALL sa_migrate_create_remote_fks_list( 'rmt_server1' );
   ```

   このプロシージャは、dbo.migrate_remote_table_list にリストされている各リモート・テーブルに関連した外部キーのリストを、テーブル dbo.migrate_remote_fks_list に設定します。

   再作成しない外部キー・マッピングを、ローカルのベース・テーブルから削除できます。

   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。

10. sa_migrate_create_fks システム・プロシージャを実行します。次に例を示します。

    ```
    CALL sa_migrate_create_fks( 'local_user1' );
    ```

    このプロシージャは、dbo.migrate_remote_fks_list に定義されている外部キー・マッピングをベース・テーブル上に作成します。

    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。

11. 移行用に作成されたプロキシ・テーブルを削除する場合は、sa_migrate_drop_proxy_tables システム・プロシージャを実行します。次に例を示します。

    ```
    CALL sa_migrate_drop_proxy_tables( 'local_user1' );
    ```

    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。**Command files** 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、**scripts** ともいいます。「コメント」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。

Interactive SQL で SQL コマンド・ファイルを開く

Windows オペレーティング・システムでは、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。このように設定すると、ファイルをダブルクリックするだけで、Interactive SQL の **[SQL 文]** ウィンドウ枠にファイルの内容が表示されます。

詳細については、「Interactive SQL を .sql ファイルのデフォルト・エディタに設定する」 『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 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。

- **[SQL 文]** ウィンドウ枠にロードすることなく、コマンド・ファイルを実行する。

  ♦ 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 で、**[ファイル]** - **[スクリプトの実行]** を選択します。
  2. ファイルを検索し、**[開く]** をクリックします。

     指定されたファイルの内容がすぐに実行されます。実行の進行状況を示す **[ステータス]** ウィンドウが表示されます。

     [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。

- Interactive SQL の READ 文を使用して、**[SQL 文]** ウィンドウ枠にロードすることなく、コマンド・ファイルを実行する。

  ♦ Interactive SQL の READ 文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。

  - **[SQL 文]** ウィンドウ枠で、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。

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。

- 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。

  ♦ 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 (コマンド・プロンプトの場合)。

  - dbisql ユーティリティを実行し、コマンド・ライン引数としてコマンド・ファイルを指定します。

    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 11 Demo" myscript.sql
    ```

- 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。

  ♦ 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] - [開く]** を選択します。
  2. ファイルを検索し、**[開く]** をクリックします。

     コマンドが **[SQL 文]** ウィンドウ枠に表示され、読み込み、編集、実行を行うことができます。

  Windows プラットフォームでは、Interactive SQL を *.sql* コマンド・ファイルのデフォルト・エディタにできます。このように設定すると、ファイルをダブルクリックするだけで、Interactive SQL の **[SQL 文]** ウィンドウ枠に内容が表示されます。「Interactive SQL を .sql ファイルのデフォルト・エディタに設定する」『SQL Anywhere サーバ - データベース管理』を参照してください。

- コマンド・ファイルをお気に入りから **[SQL 文]** ウィンドウ枠にロードする。

  「お気に入りの使用」『SQL Anywhere サーバ - データベース管理』を参照してください。

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 Employees;
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。

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テキスト・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 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・データへのアクセス

目次

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、Microsoft 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実行することができます。データを取り出してから操作することによって、リモート・データ・ソースではサポートされていない機能を SQL Anywhere が補います。
- 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
- Advantage Database Server
- Oracle
- IBM DB2
- Microsoft SQL Server
- Microsoft Access
- MySQL
- Ultra Light
- 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、http://www.ianywhere.jp/developers/technotes/os_components_1101.html を参照してください。

リモート・テーブルのマッピング

クライアント・アプリケーション側から見ると、接続している SQL Anywhere 内にすべてのテーブルがあるかのように見えます。リモート・テーブルに関わるクエリが実行されると、内部では対象となるデータ・ソースを割り出し、外部にあるそのデータ・ソースにアクセスしてデータを取り出します。

リモート・テーブル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」 820 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ピングします。「外部ログインの使用」 829 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・テーブルの操作」 831 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
- **ulodbc** UltraLite
- **aseodbc** Sybase SQL Server と Adaptive Server Enterprise (バージョン 10 以降)
- **adsodbc** Advantage Database Server
- **db2odbc** IBM DB2
- **mssodbc** Microsoft SQL Server
- **oraodbc** Oracle サーバ (バージョン 8.0 以降)
- **mysqlodbc** MySQL
- **msaccessodbc** Microsoft Access
- **odbc** その他の ODBC データ・ソース

注意
リモート・データ・アクセスを使用する際に、Unicode をサポートしていない ODBC ドライバを使用すると、その ODBC ドライバから受け取るデータに対して、文字セット変換が実行されません。

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)

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、ODBC クラスを使用できない場合にのみ使用してください。

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」 851 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 に設定すると、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]** ウィンドウの **[トランザクション]** タブを開いて [Retrieve ブロック化係数] を 1 に設定すると、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"
```

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。

CREATE SERVER 文を使用して、リモート・サーバを作成します。

CREATE SERVER 文を使用して、リモート・サーバ定義を設定します。Sybase Central を使用したリモート・サーバ定義の作成方法については、「Sybase Central を使用したリモート・サーバの作成」 821 ページを参照してください。

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・サーバが存在するコンピュータの名前と 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
- **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 11;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・サーバを作成するには、次の手順に従います (Sybase Central の場合)。**

1. DBA 権限のあるユーザとしてホスト・データベースに接続します。
2. 左ウィンドウ枠で、**[リモート・サーバ]** をダブルクリックします。
3. **[ファイル]** - **[新規]** - **[リモート・サーバ]** を選択します。
4. **[新しいリモート・サーバの名前を指定してください。]** フィールドにリモート・サーバの名前を入力し、**[次へ]** をクリックします。
5. リモート・サーバのタイプを選択して、**[次へ]**をクリックします。
6. 接続タイプを選択し、**[接続情報を指定してください。]** フィールドに次に示す接続情報を入力します。

- ODBC 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、*computer-name*:*port-number* の形式で URL を指定します。

データ・アクセス・メソッド (JDBC か ODBC) は、SQL Anywhere がリモート・データベースへのアクセスに使用するためのものです。Sybase Central が Adaptive Server Anywhere に接続する方法をこれで決めているわけではありません。

7. **[次へ]** をクリックします。
8. リモート・サーバを読み込み専用にするかどうかを指定し、**[次へ]** をクリックします。
9. Click **[現在のユーザの外部ログインを作成する]** をクリックし、必須フィールドの入力を完成させます。

   デフォルトでは、SQL Anywhere は、現在のユーザに代わってリモート・サーバに接続する場合に、常にそのユーザのユーザ ID とパスワードを使用します。ただし、リモート・サーバに現在のユーザと同じユーザ ID とパスワードで定義されたユーザがいない場合は、外部ログインを作成する必要があります。外部ログインは、現在のユーザ用に代替ログイン名とパスワードを割り当ててリモート・サーバに接続できるようにします。「CREATE EXTERNLOGIN 文」 『SQL Anywhere サーバ - SQL リファレンス』を参照してください。
10. **[接続テスト]** をクリックしてリモート・サーバ接続をテストします。
11. **[完了]** をクリックします。

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、次の手順に従います (Sybase Central の場合)。**

1. DBA 権限のあるユーザとしてホスト・データベースに接続します。
2. 左ウィンドウ枠で、**[リモート・サーバ]** をダブルクリックします。
3. リモート・サーバを選択し、**[ファイル]** - **[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. DBA 権限のあるユーザとしてホスト・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

リモート・サーバの設定の変更は、次回リモート・サーバに接続する際に有効になります。

♦ リモート・サーバのプロパティを変更するには、次の手順に従います (Sybase Central の場合)。

1. RESOURCE 権限のあるユーザとしてホスト・データベースに接続します。
2. 左ウィンドウ枠で、**[リモート・サーバ]** をダブルクリックします。
3. リモート・サーバを選択し、**[ファイル]** - **[プロパティ]** を選択します。
4. リモート・サーバの設定を変更し、**[OK]**をクリックします。

♦ 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. DBA 権限のあるユーザとしてホスト・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
[, @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 ワークシートの中から、excel という名前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 の運用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 はこの機能の情報を使用して、リモート・サーバに渡すことができる SQL 文の量を判断します。

また、SYSCAPABILITY と SYSCAPABILITYNAME のシステム・ビューを問い合わせると、リモート・サーバの機能情報も参照できます。これらのシステム・ビューは、SQL Anywhere が最初にリモート・サーバに接続するまでは空の状態です。

sp_servercaps システム・プロシージャを使用する場合は、*server-name* には CREATE SERVER 文で使用した *server-name* と同じ名前を指定してください。

次のように、sp_servercaps ストアド・プロシージャを実行します。

```
CALL sp_servercaps  server-name;
```

参照

- 「sp_servercaps システム・プロシージャ」『SQL Anywhere サーバ - SQL リファレンス』
- 「SYSCAPABILITY システム・ビュー」『SQL Anywhere サーバ - SQL リファレンス』
- 「SYSCAPABILITYNAME システム・ビュー」『SQL Anywhere サーバ - SQL リファレンス』
- 「CREATE SERVER 文」『SQL Anywhere サーバ - SQL リファレンス』

ディレクトリ・アクセス・サーバの使用

「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 文または**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. 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. 左ウィンドウ枠で、**[ディレクトリ・アクセス・サーバ]** をダブルクリックします。
3. **[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。
4. **ディレクトリ・アクセス・サーバの作成ウィザード**の指示に従います。

♦ 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. DBA 権限のあるユーザとしてホスト・データベースに接続します。
2. 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。

```
CREATE SERVER my_dir_tree
CLASS 'directory'
USING 'root=c:¥Program Files';
```

3. 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;
   ```

4. 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. 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. 左ウィンドウ枠で、**[ディレクトリ・アクセス・サーバ]** をダブルクリックします。
3. ディレクトリ・アクセス・サーバを選択し、**[編集]** - **[削除]** を選択します。

♦ **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)。**

1. DBA 権限のあるユーザとしてホスト・データベースに接続します。
2. DROP SERVER 文を実行します。

   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. 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. 左ウィンドウ枠で、**[ディレクトリ・アクセス・サーバ]** をダブルクリックします。
3. 右ウィンドウ枠で、**[プロキシ・テーブル]** タブをクリックします。
4. プロキシ・テーブルを選択し、**[編集]** - **[削除]** を選択します。
5. **[はい]** をクリックします。

♦ **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)。**

1. DBA 権限のあるユーザとしてホスト・データベースに接続します。
2. 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 TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』

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

外部ログインを作成するには、次のいずれかの手順を使用します。

- **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. DBA 権限のあるユーザとして、または外部ログインの所有者として、ホスト・データベースに接続します。
2. 左ウィンドウ枠で、**[リモート・サーバ]** をダブルクリックします。
3. リモート・サーバを選択し、右ウィンドウ枠で **[外部ログイン]** タブをクリックします。
4. **[ファイル]** - **[新規]** - **[外部ログイン]** を選択します。
5. **外部ログイン作成ウィザード**の指示に従います。

- **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. DBA 権限のあるユーザとして、または外部ログインの所有者として、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

SQL Anywhere のシステム・テーブルから外部ログインを削除するには、次のいずれかの手順を使用します。

♦ **外部ログインを削除するには、次の手順に従います (Sybase Central の場合)。**

1. DBA 権限のあるユーザとして、または外部ログインの所有者として、ホスト・データベースに接続します。
2. 左ウィンドウ枠で、**[リモート・サーバ]** をダブルクリックします。
3. リモート・サーバを選択し、右ウィンドウ枠で **[外部ログイン]** タブをクリックします。
4. 外部ログインを選択し、**[ファイル]** - **[削除]** を選択します。
5. **[はい]** をクリックします。

♦ **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. DBA 権限のあるユーザとして、または外部ログインの所有者として、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』

プロキシ・テーブルの操作

リモート・オブジェクトに対応するローカルの「プロキシ・テーブル」を作成すると、意識することなくリモート・データにアクセスできるようになります。プロキシ・テーブルを使用すると、リモート・データベースがプロキシ・テーブルの候補としてエクスポートする任意のオブジェクト (テーブル、ビュー、マテリアライズド・ビューを含む) にアクセスできます。プロキシ・テーブルを作成するには、次の文のいずれかを使用します。

- リモート・サーバにすでにテーブルが存在する場合は、CREATE EXISTING TABLE 文を使用します。この文は、リモート・サーバの既存のテーブルのプロキシ・テーブルを定義します。
- リモート・サーバにテーブルが存在しない場合は、CREATE TABLE 文を使用します。この文はリモート・サーバに新しいテーブルを作成して、そのテーブルのプロキシ・テーブルを定義します。

注意
セーブポイント内では、プロキシ・テーブルを変更することはできません。「トランザクション内のセーブポイント」 122 ページを参照してください。

プロキシ・テーブルでトリガを起動する場合は、プロキシ・テーブルの所有者の権限ではなく、トリガを起動するユーザの権限を使用します。

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。

- **owner**　データベースが所有者の概念をサポートしている場合、所有者名を表します。このフィールドは、何人かの所有者が同じ名前でテーブルを所有する場合にだけ必要です。
- **table-name**　このフィールドはテーブルの名前を指定します。Excel スプレッドシートの場合、これはブックの「シート」の名前になります。*table-name* を入力しない場合、リモート・テーブル名はローカルのプロキシ・テーブル名と同じであると見なされます。

例

以下はロケーション文字列の使用例です。

- SQL Anywhere：

  'RemoteSA..GROUPO.Employees'

- Adaptive Server Enterprise：

  'RemoteASE.pubs2.dbo.publishers'

- Excel：

  '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・テーブルを作成するには、次のいずれかの手順を使用します。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. DBA 権限のあるユーザとしてホスト・データベースに接続します。
2. 左ウィンドウ枠で、**[リモート・サーバ]** をダブルクリックします。
3. リモート・サーバを選択し、右ウィンドウ枠で **[プロキシ・テーブル]** タブを選択します。
4. **[ファイル]** - **[新規]** - **[プロキシ・テーブル]** を選択します。
5. **プロキシ・テーブル作成ウィザード**の指示に従います。

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)。

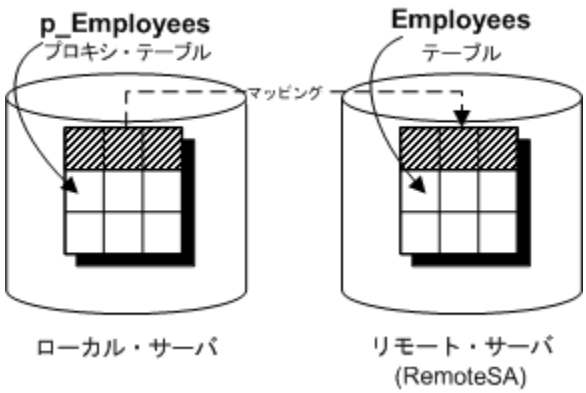
1. DBA 権限のあるユーザとしてホスト・データベースに接続します。
2. CREATE EXISTING TABLE 文を実行します。

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* にマッピングします。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。

```
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・テーブルにマッピングする Members というプロキシ・テーブルを作成します。

```
CREATE TABLE Members
( membership_id INTEGER NOT NULL,
member_name CHAR( 30 ) NOT NULL,
office_held CHAR( 20 ) NULL )
AT 'RemoteSA..GROUPO.Employees';
```

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。

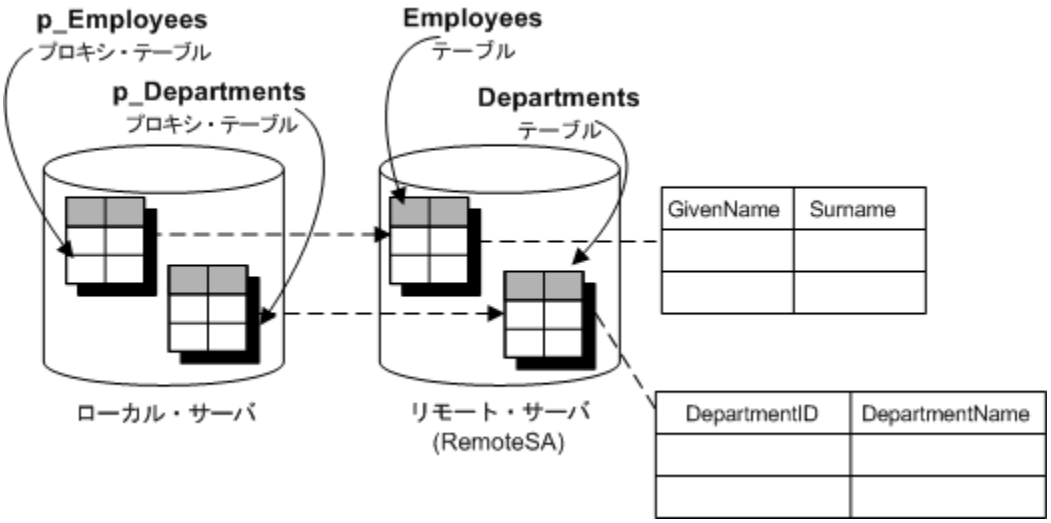
たとえば、asetest という名前の Adaptive Server Enterprise サーバの production データベースにある sysobjects テーブルのカラムのリストを取得するには、次のように指定します。

```
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。

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・テーブルにマッピングされた、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。

   ```
   dbeng11 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 11 Demo を参照します。

   ```
   CREATE SERVER RemoteSA
   CLASS 'saodbc'
   USING 'SQL Anywhere 11 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;
```


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」 867 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';
   ```

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;
```

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・コールを発行するには、次のいずれかの手順を使用します。

リモート・プロシージャは、最長 254 バイトの入力パラメータを受け入れ、最大 254 文字までの出力変数を返します。

リモート・プロシージャが結果セットを返すことができる場合は、たとえすべてのケースで結果セットを返せるわけではなくても、ローカル・プロシージャ定義には RESULT 句を含めてください。

♦ リモート・プロシージャを作成するには、次の手順に従います (Sybase Central の場合)。

1. DBA 権限のあるユーザとしてホスト・データベースに接続します。
2. 左ウィンドウ枠で、**[リモート・サーバ]** をダブルクリックします。
3. リモート・サーバを選択し、右ウィンドウ枠で **[リモート・プロシージャ]** タブをクリックします。
4. **[ファイル]** - **[新規]** - **[リモート・プロシージャ]** を選択します。
5. **リモート・プロシージャ作成ウィザード**の指示に従います。

♦ 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 (SQL の場合)。

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. 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';
```

詳細については、「CREATE PROCEDURE 文 [Web サービス]」 『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・コールを発行するには、次の手順に従います (SQL の場合)。**

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. 次のようにプロシージャを実行します。

```
CALL remotewho();
```

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

リモート・プロシージャを削除するには、次のいずれかの手順を使用します。

♦ **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 (Sybase Central の場合)。**

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. 左ウィンドウ枠で、**[リモート・サーバ]** をダブルクリックします。
3. リモート・サーバを選択し、右ウィンドウ枠で **[リモート・プロシージャ]** タブをクリックします。
4. リモート・プロシージャを選択し、**[ファイル]** - **[削除]** を選択します。
5. **[はい]** をクリックします。

♦ **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. 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. DROP PROCEDURE 文を実行します。

詳細については、「DROP PROCEDURE 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の使用」 117 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。
- ネストされた BEGIN TRANSACTION 文と COMMIT TRANSACTION 文がリモート・サーバに関わるトランザクションに含まれている場合は、一番外側の組の文だけが処理されます。

BEGIN TRANSACTION 文と COMMIT TRANSACTION 文を含む一番内側の組は、リモート・サーバに転送されません。

内部オペレーション

この項では、クライアント・アプリケーションの裏側で行われている SQL Anywhere のリモート・サーバに対するオペレーションについて説明します。

クエリの解析

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。

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・サーバは特定の SQL 文の全機能をサポートしているとはかぎらないので、SQL Anywhere では、クエリがリモート・サーバに対応できるようになるまで、文を単純なコンポーネントに分割する必要があります。リモート・サーバに渡されない SQL 機能は、SQL Anywhere 自身が評価します。

たとえば、あるクエリに ORDER BY 文があるとします。リモート・サーバが ORDER BY を実行できない場合、ORDER BY を除いて、文がリモート・サーバに送信されます。SQL Anywhere は、結果が返されると、ORDER BY を実行してから結果をユーザに返します。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

効率性を考慮して、SQL Anywhere では、文に含まれるできるだけ多くの要素をリモート・サーバに渡します。多くの場合、これは元々 SQL Anywhere に指定された完全な文です。

SQL Anywhere は、次のような場合に完全な文を渡します。

- 文内のテーブルがどれも同じリモート・サーバに存在している。
- リモート・サーバが文内のすべての構文を処理できる。

まれに、リモート・サーバが作業を行うよりも、SQL Anywhere がいくつかの作業を行った方が効率が良い場合があります。たとえば、SQL Anywhere のソート・アルゴリズムの方が優れていることがあります。この場合は、ALTER SERVER 文を使用して、リモート・サーバの機能の変更を検討します。

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・サーバではサポートされていない SQL 機能を文が使用している場合は、クエリは複数の単純な部分に分解されます。

SELECT

引き渡すことのできない部分が取り除かれながら、SELECT 文は分解されていきます。取り除かれた部分の処理は SQL Anywhere によって実行されます。たとえば、次の文内にある ATAN2 関数をリモート・サーバが処理できないとします。

```
SELECT a,b,c
WHERE ATAN2( b, 10 ) > 3
AND c = 10;
```

リモート・サーバに送信される文は、次のように変換されます。

```
SELECT a,b,c WHERE c = 10;
```

次に、SQL Anywhere がローカルで WHERE ATAN2(b, 10) > 3 を中間結果セットに適用します。

ジョイン

2 つのテーブルがジョインされると、1 つは外部テーブルとなります。外部テーブルは、テーブルに適用する WHERE 条件に基づいてスキャンされます。検出され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。

テンポラリ・テーブルは更新できない
中間テンポラリ・テーブルが必要な場合は、UPDATE または DELETE を実行できません。これは ORDER BY を持つクエリと、サブクエリを持つクエリのいくつかで発生します。

リモート・データ・アクセスのトラブルシューティング

この項では、リモート・サーバへのアクセスのトラブルシューティングのヒントをいくつか説明します。

リモート・データに対してサポートされない機能

次の SQL Anywhere 機能はリモート・データではサポートされていません。

- リモート・テーブルに対する 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・データに対して開かれたカーソルの後方スクロール。フェッチ文は NEXT または RELATIVE 1 である必要があります。
- プロキシ・テーブルを参照する式を含む関数の呼び出し
- リモート・テーブルのカラムにリモート・サーバに関するキーワードがある場合、そのカラムのデータにはアクセスできない。CREATE EXISTING TABLE 文を実行して定義をインポートすることはできますが、そのカラムを選択することはできません。

大文字と小文字の区別

SQL Anywhere データベースの大文字と小文字の区別の設定は、アクセスするリモート・サーバが使用する設定に合わせてください。

デフォルトでは、SQL Anywhere データベースは大文字と小文字を区別しないで作成されます。この設定では、大文字と小文字を区別するデータベースから選択を行ったときに、予期しない結果が発生することがあります。ORDER BY または文字列比較がリモート・サーバに送信されるか、それともローカルの SQL Anywhere によって評価されるかによって、発生する結果が異なります。

接続のテスト

次の手順で、リモート・サーバに接続できることを確認します。

- Interactive SQL などのクライアント・ツールを使用してリモート・サーバに接続できることを確認してから、SQL Anywhere を設定します。
- リモート・サーバに対して簡単なパススルー文を実行して、接続とリモート・ログインの設定を確認します。次に例を示します。

  ```
  FORWARD TO RemoteSA {SELECT @@version};
  ```

- リモート・サーバとの対話をトレースするために、リモート・トレーシングを有効にします。次に例を示します。

  ```
  SET OPTION cis_option = 7;
  ```

  リモート・トレーシングを有効にすると、データベース・サーバ・メッセージ・ウィンドウにトレーシング情報が表示されます。この出力をファイルに記録するには、データベース・サーバの起動時に -o サーバ・オプションを指定します。

  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、リモート・トレーシングやクエリの実行プランを表示できます。

```
SET OPTION cis_option = 7;
```

リモート・トレーシングを有効にすると、データベース・サーバ・メッセージ・ウィンドウにトレーシング情報が表示されます。この出力をファイルに記録するには、データベース・サーバの起動時に -o サーバ・オプションを指定します。

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」 139 ページを参照してください。

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。

リモート・データ・アクセスのサーバ・クラス

目次

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。

- ODBC ベースのサーバ・クラス
- JDBC 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。

サーバ・クラスのカテゴリ (JDBC ベースか ODBC ベース) 全般についての情報と、個々のサーバ・クラスに固有の情報の両方を参照する必要があります。

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
- msaccessodbc
- mysqlodbc
- ulodbc
- adsodbc
- odbc

注意
リモート・データ・アクセスを使用する際に、Unicode をサポートしていない ODBC ドライバを使用すると、その ODBC ドライバから受け取るデータに対して、文字セット変換が実行されません。

ODBC 外部サーバの定義

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。これを実行するため、ODBC アドミニストレータを使用して、データ・ソースを作成することができます。

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、データ・ソース名も 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';
```

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。

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 11;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が TCP/IP です。

参照

各 ODBC サーバ・クラスに固有の情報については、次を参照してください。

- 「サーバ・クラス saodbc」 853 ページ
- 「サーバ・クラス ulodbc」 853 ページ
- 「サーバ・クラス aseodbc」 853 ページ
- 「サーバ・クラス db2odbc」 856 ページ
- 「サーバ・クラス oraodbc」 858 ページ
- 「サーバ・クラス mssodbc」 860 ページ
- 「サーバ・クラス msaccessodbc」 863 ページ
- 「サーバ・クラス mysqlodbc」 861 ページ
- 「サーバ・クラス adsodbc」 856 ページ
- 「サーバ・クラス odbc」 864 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。「CREATE SERVER 文の USING パラメータ」 867 ページを参照してください。

サーバ・クラス ulodbc

サーバ・クラスが ulodbc のサーバは Ultra Light データベースです。Ultra Light データベースへの接続を定義する ODBC データ・ソース名を作成します。ODBC データ・ソース名の CREATE SERVER 文を発行します。

Ultra Light は、SQL Anywhere で使用可能なデータ型のサブセットをサポートしているため、Ultra Light のデータ型と SQL Anywhere のデータ型には 1 対 1 のマッピングが存在します。「Ultra Light のデータ型」 『Ultra Light データベース管理とリファレンス』を参照してください。

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が、パフォーマンスは、asejdbc クラスの方が優れています。

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。
- quoted_identifiers オプションのローカル設定は、Adaptive Server Enterprise の引用符付き識別子の使用を制御します。たとえば、quoted_identifiers オプションをローカルで Off に設定すると、Adaptive Server Enterprise に対して引用符付き識別子がオフになります。
- **[Configuration Manager]** でユーザ・データソースを次の属性で設定します。
  - **[General] タブ**　**[Data Source Name]** に任意の値を入力します。この値は、CREATE SERVER 文の USING 句に使用されます。

    サーバ名は Sybase interfaces ファイルにあるサーバ名と一致させてください。

    interfaces ファイルの詳細については、「interfaces ファイル」 『SQL Anywhere サーバ - データベース管理』を参照してください。
  - **[Advanced] タブ**　**[Application Using Threads]** オプションと **[Enable Quoted Identifiers]** オプションを選択します。
  - **[Connection] タブ**　[charset] フィールドを、SQL Anywhere の文字セットに一致するように設定します。

    [language] フィールドを、エラー・メッセージを表示したい言語に設定します。
  - **[Performance] タブ**　**[Prepare Method]** を 「2-Full」 に設定します。

    最高のパフォーマンスを得るには、**[Fetch Array Size]** をできるだけ大きな値に設定します。これはメモリ内にキャッシュされるローの数なので、この値を大きくすると必要なメモリ量が増大します。Adaptive Server Enterprise では 100 の値を使用することをおすすめします。

    [Select Method] を 「0-Cursor」 に設定します。

    [Packet Size] をできるだけ大きな値に設定します。Adaptive Server Enterprise では -1 の値を使用することをおすすめします。

    [Connection Cache] を 1 に設定します。

データ型変換：ODBC と Adaptive Server Enterprise

CREATE TABLE 文を発行するときに、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*)
CHARACTER(*n*)	varchar(*n*)
VARCHAR(*n*)	varchar(*n*)
CHARACTER VARYING(*n*)	varchar(*n*)
LONG VARCHAR	text

SQL Anywhere データ型	Adaptive Server Enterprise のデフォルト・データ型
TEXT	text
BINARY(*n*)	binary(*n*)
LONG BINARY	image
IMAGE	image
BIGINT	numeric(20,0)

サーバ・クラス adsodbc

CREATE TABLE 文を発行するときに、SQL Anywhere は、次のデータ型変換を使用して、データ型を対応する Advantage Database Server のデータ型に自動的に変換します。

SQL Anywhere データ型	ADS のデフォルト・データ型
BIT	Logical
TINYINT、SMALLINT、INT、INTEGER	Integer
BIGINT	Numeric(32)
DECIMAL(p,s)、NUMERIC(p,s)	Numeric(p+3)
DATE	Date
TIME	Time
DATETIME、TIMESTAMP	TimeStamp
MONEY、SMALLMONEY	Money
FLOAT、REAL	Double
CHAR(*n*)、VARCHAR(*n*)、LONG VARCHAR	Char(*n*)
BINARY(*n*)、VARBINARY(*n*)、LONG BINARY	Blob

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
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

SQL Anywhere データ型	DB2 のデフォルト・データ型
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
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 ドライバの制限の結果として、システム・テーブルの CREATE EXISTING TABLE 文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、次のデータ型変換を使用して、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。

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)
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

SQL Anywhere データ型	Oracle のデータ型
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';
```

- quoted_identifiers オプションのローカル設定は、Microsoft SQL Server の引用符付き識別子の使用を制御します。たとえば、quoted_identifiers オプションをローカルで Off に設定すると、Microsoft SQL Server に対して引用符付き識別子がオフになります。

データ型変換：Microsoft SQL Server

CREATE TABLE 文を発行するときに、SQL Anywhere は、次のデータ型変換を使用して、データ型を対応する Microsoft SQL Server のデータ型に自動的に変換します。

SQL Anywhere データ型	Microsoft SQLServer のデフォルト・データ型
BIT	bit
TINYINT	tinyint
SMALLINT	smallint
INT	int

SQL Anywhere データ型	Microsoft SQLServer のデフォルト・データ型
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
DOUBLE	float
UNIQUEIDENTIFIERSTR	uniqueidentifier

サーバ・クラス mysqlodbc

CREATE TABLE 文を発行するときに、SQL Anywhere は、次のデータ型変換を使用して、データ型を対応する MySQL のデータ型に自動的に変換します。

SQL Anywhere データ型	MySQL のデフォルト・データ型
BIT	bit(1)
TINYINT	tinyint unsigned
SMALLINT	smallint
INT、INTEGER	int
BIGINT	bigint
DECIMAL(p,s)、NUMERIC(p,s)	decimal(p,s)
DATE	date
TIME	time
DATETIME、TIMESTAMP	datetime
MONEY	decimal(19,4)
SMALLMONEY	decimal(10,4)
FLOAT	float
REAL	real
CHAR(*n*)	*n* が 254 未満の場合 char(*n*) *n* が 254 以上、4000 未満の場合 varchar(*n*) *n* が 4000 以上の場合 longtext
VARCHAR(*n*)	*n* が 4000 未満の場合 varchar(*n*) *n* が 4000 以上の場合 longtext
LONG VARCHAR	longtext
BINARY(*n*)、VARBINARY(*n*)	*n* が 4000 未満の場合 varbinary(*n*) *n* が 4000 以上の場合 longblob
LONG BINARY	longblob

サーバ・クラス msaccessodbc

Access データベースは *.mdb* ファイルに格納されます。ODBC マネージャを使用して、ODBC データ・ソースを作成し、これらのファイルの 1 つにマッピングします。新しい *.mdb* ファイルは、ODBC マネージャを使って作成できます。SQL Anywhere でテーブルを作成するときにデフォルトを指定しないと、このデータベース・ファイルがデフォルトになります。

ODBC データ・ソースが access という名前であると仮定した場合、次のいずれかの文を使用してデータにアクセスできます。

- CREATE TABLE tab1 (a int, b char(10))
  AT 'access...tab1';
- CREATE TABLE tab1 (a int, b char(10))
  AT 'access;d:¥pcdb¥data.mdb;;tab1';
- CREATE EXISTING TABLE tab1
  AT 'access;d:¥pcdb¥data.mdb;;tab1';

Access では所有者名の修飾をサポートしないので、これはブランクのままにしてください。

データ型変換：Microsoft Access

SQL Anywhere データ型	Microsoft Access のデフォルト・データ型
BIT、TINYINT	TINYINT
SMALLINT	SMALLINT
INT、INTEGER	INTEGER
BIGINT	DECIMAL(19,0)
DECIMAL(p,s)、NUMERIC(p,s)	DECIMAL(p,s)
DATE、TIME、DATETIME、TIMESTAMP	DATETIME
MONEY、SMALLMONEY	MONEY
FLOAT	FLOAT
REAL	REAL
CHAR(*n*)、VARCHAR(*n*)	*n* が 254 未満の場合 CHARACTER(*n*) *n* が 254 以上の場合 TEXT
LONG VARCHAR	TEXT
BINARY、VARBINARY	*n* が 4000 未満の場合 BINARY(*n*) *n* が 4000 以上の場合 IMAGE

SQL Anywhere データ型	Microsoft Access のデフォルト・データ型
LONG BINARY	IMAGE

サーバ・クラス odbc

独自のサーバ・クラスを持たない ODBC データ・ソースでは、サーバ・クラス **odbc** を使用します。任意の ODBC ドライバを使用できます。iAnywhere は次の ODBC データ・ソースの使用を確認しています。

- 「Microsoft Excel (Microsoft 3.51.171300)」 864 ページ
- 「Microsoft FoxPro (Microsoft 3.51.171300)」 865 ページ
- 「Lotus Notes SQL 2.0」 865 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、Microsoft ダウンロード・センターからダウンロードできます。MDAC 2.0 には、上に示すバージョンの Microsoft ドライバが含まれています。

Microsoft Excel (Microsoft 3.51.171300)

Excel で、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ピングされます。ODBC ドライバ・マネージャで ODBC データ・ソース名を設定する場合は、そのデータ・ソースに関連付けられたデフォルトのブック名を指定します。ただし、CREATE TABLE 文を発行する場合には、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';
```

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、マッピングするカラムとローの範囲を明示的に指定する必要があります。次に例を示します。

```
CREATE EXISTING TABLE mywork
AT 'excel;d:¥work1;;mywork$';
```


シート名に $ を追加すると、ワークシート全体が選択されることを示します。

AT で指定するロケーションの文字列で、フィールド・セパレータとしてピリオドの代わりにセミコロンが使用されていることに注意してください。これは、ファイル名にピリオドが使用されるためです。Excel では所有者名フィールドをサポートしないので、これはブランクのままにしてください。

削除はサポートされていません。また、Excel ドライバは位置付け更新をサポートしないため、更新も可能でない場合があります。

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';
```

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 http://www.lotus.com/ 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ピングされるかについては、ドライバに付属のマニュアルを参照してください。SQL Anywhere テーブルは、Notes フォームに簡単にマッピングできます。

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 テーブルにマッピングします。

  ```
  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。

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)

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、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 6.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。
- JDBC リモート・サーバ接続が切断されるか、失われた場合、JDBC リモート・サーバを使用して、プロキシ・テーブルやプロキシ・プロシージャなどのプロキシ・オブジェクトにアクセスしようとした場合にのみ、サーバが使用できないことがわかります。ODBC にはこの制限がありません。

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

アクセスする SQL Anywhere データベースごとに、個別の CREATE SERVER 文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 データベース・サーバを設定します。

```
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';
```

/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。

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。

注意

- quoted_identifiers オプションのローカル設定は、Adaptive Server Enterprise の引用符付き識別子の使用を制御します。たとえば、quoted_identifiers オプションをローカルで Off に設定すると、Adaptive Server Enterprise に対して引用符付き識別子がオフになります。

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、次のデータ型変換を使用して、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。

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

SQL Anywhere データ型	Adaptive Server Enterprise のデフォルト・データ型
SMALLMONEY	numeric(10,4)
MONEY	numeric(19,4)
DATE	datetime
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)

ストアド・プロシージャとトリガ

この項では、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。

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

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。

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」 927 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」 227 ページを参照してください。

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、いくつかの利点があります。

標準化

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

効率化

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

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

セキュリティ

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

プロシージャの作成

Sybase Central では、**プロシージャ作成ウィザード**に、プロシージャ・テンプレートを使用するオプションもあります。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。プロシージャを作成するには DBA または RESOURCE 権限が必要です。

♦ **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 権限のあるユーザとしてデータベースに接続します。
2. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックします。
3. **[ファイル]** - **[新規]** - **[プロシージャ]** を選択します。
4. **プロシージャ作成ウィザード**の指示に従います。
5. 右ウィンドウ枠で、**[SQL]** タブをクリックして、プロシージャ・コードを完了します。

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

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,
  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。

テンポラリ・プロシージャ

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。「CREATE PROCEDURE 文 [Web サービス]」 『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

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。次の項を参照してください。

- 「リモート・プロシージャの作成」 839 ページ
- 「Sybase Central を使用したリモート・サーバの作成」 821 ページ

参照

- 「SQL Anywhere データベース接続」 『SQL Anywhere サーバ - データベース管理』
- 「CREATE PROCEDURE 文 [Web サービス]」 『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」 900 ページ
- 「リモート・プロシージャの作成」 839 ページ

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。

♦ **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. DBA 権限または RESOURCE 権限のあるユーザとしてデータベースに接続します。
2. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックします。
3. プロシージャを選択します。
4. 次のいずれかの方法を使用して、プロシージャを編集します。
   - 右ウィンドウ枠で、**[SQL]** タブをクリックします。
   - プロシージャを右クリックして、**[新しいウィンドウで編集]** を選択します。

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

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

  データベース・ドキュメント・ジェネレータを使用して、SQL Anywhere データベースをドキュメント化する場合、これらのコメントを出力に含めるオプションがあります。「データベースのドキュメント化」『SQL Anywhere サーバ - データベース管理』を参照してください。

参照

- 「データベース・オブジェクトのプロパティの設定」 16 ページ
- 「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』
- 「ユーザのパーミッションと権限の取り消し」『SQL Anywhere サーバ - データベース管理』
- 「ALTER PROCEDURE 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「CREATE PROCEDURE 文 [Web サービス]」『SQL Anywhere サーバ - SQL リファレンス』
- 「プロシージャの作成」 876 ページ
- 「Sybase Central を使用したストアド・プロシージャの変換」 713 ページ

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。

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。

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;
```

参照

- 「データベースのパーミッションと権限の概要」『SQL Anywhere サーバ - データベース管理』
- 「CALL 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「GRANT 文」『SQL Anywhere サーバ - SQL リファレンス』
- 「FROM 句」『SQL Anywhere サーバ - SQL リファレンス』

Sybase Central におけるプロシージャのコピー

Sybase Central でデータベース間でプロシージャをコピーするには、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]** 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。

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、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 権限のあ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. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックします。
3. プロシージャを選択して、**[編集]** - **[削除]** を選択します。
4. **[はい]** をクリックします。

♦ **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 権限のあるユーザ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;
```

参照

- 「SQL Anywhere データベース接続」『SQL Anywhere サーバ - データベース管理』
- 「DROP PROCEDURE 文」『SQL Anywhere サーバ - SQL リファレンス』

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。

♦ プロシージャを作成して実行し、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。
2. [SQL 文] ウィンドウ枠で、次のように入力し、従業員の平均給与を OUT パラメータとして返すプロシージャ (AverageSalary) を作成します。

```
CREATE PROCEDURE AverageSalary( OUT avgsal NUMERIC(20,3) )
BEGIN
  SELECT AVG( Salary )
  INTO avgsal
  FROM Employees;
END;
```

3. プロシージャの結果を格納する変数を作成します。ここでは、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。

参照

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

プロシージャ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。Interactive SQL 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 );
```

Employee ID	Salary
102	45700.000
105	62000.000
160	57490.000
243	72995.000
...	...

[オプション] ウィンドウの **[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。

参照

- 「プロシージャから複数の結果セットを返す」 908 ページ

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。

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

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;
```

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。

Sybase Central からユーザ定義関数を作成することもできます。

♦ **ユーザ定義関数を作成するには、次の手順に従います (Sybase Central の場合)。**

1. DBA 権限または RESOURCE 権限のあるユーザとしてデータベースに接続します。
2. 左ウィンドウ枠で、**[プロシージャとファンクション]** をクリックします。
3. **[ファイル]** - **[新規]** - **[ファンクション]** を選択します。
4. **ファンクション作成ウィザード**の指示に従います。
5. 右ウィンドウ枠で、**[SQL]** タブをクリックして、プロシージャ・コードを完了します。

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

参照

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

ユーザ定義関数の呼び出し

ユーザ定義関数は、集合関数以外の組み込み関数が使われていればどこでも使用できますが、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
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、カナダとアメリカの顧客が含まれます。ユーザ定義関数 Nationality は、Country カラムの入力データに基づいて 3 文字の国コードを生成します。

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

```
        ELSE
           SET nation_string = 'OTH';
        END IF;
     END IF;
RETURN ( nation_string );
END;
```

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

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

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

カーソルと例外の宣言については後で説明します。

注意

この関数は説明には役立ちますが、多数のローを含む SELECT に使用する場合は、性能が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 権限のあるユーザのみが、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;
```

「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「べき等」として扱われます。べき等関数は、同じパラメータに対して一貫した結果を返し、副次効果はありません。同じパラメータを持つべき等関数が連続して 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」 636 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」 85 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。

注意
その後のトリガが起動しない LOAD TABLE、TRUNCATE、WRITETEXT の 3 つの特別な文があります。「LOAD TABLE 文」『SQL Anywhere サーバ - SQL リファレンス』、「TRUNCATE 文」『SQL Anywhere サーバ - SQL リファレンス』、「WRITETEXT 文 [T-SQL]」『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

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。

トリガが起動しないようにするには、-gf サーバ・オプションを指定するか、または fire_triggers オプションを設定します。次の項を参照してください。

- 「-gf サーバ・オプション」『SQL Anywhere サーバ - データベース管理』
- 「fire_triggers オプション [互換性]」『SQL Anywhere サーバ - データベース管理』

トリガのタイプ

SQL Anywhere では、次のトリガのタイプがサポートされています。

- **BEFORE トリガ** BEFORE トリガは、トリガ元アクションが実行される前に実行されます。BEFORE トリガはテーブルに定義できますが、ビューには定義できません。
- **AFTER トリガ** AFTER トリガは、トリガ元アクションが完了した後に実行されます。AFTER トリガはテーブルに定義できますが、ビューには定義できません。
- **INSTEAD OF トリガ** INSTEAD OF トリガは、トリガ元アクションの代わりに実行される条件付きのトリガです。INSTEAD OF トリガはテーブルとビューに定義できます (マテリアライズド・ビューを除く)。「INSTEAD OF トリガ」 893 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に新しいローが挿入されたときに、トリガが起動される。
DELETE	トリガの関連するテーブル内のローが削除されたときに、トリガが起動される。
UPDATE	トリガの関連するテーブル内のローが更新されたときに、トリガが起動される。
UPDATE OF *column-list*	トリガの関連するテーブル内のローが、**column-list** 中のカラムが変更されるなどして更新されたときに、トリガが起動される。

処理が必要なイベントごとにトリガを個別に作成できます。または、共有するアクションや、イベントに応じたアクションが複数ある場合は、すべてのイベントに対して 1 つのトリガを作成し、IF 文を使用して実行するアクションを区別できます。「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。SQL Anywhere では、文レベル BEFORE トリガはサポートされていません。

トリガ実行のタイミングは柔軟に設定できるので、実行に応じてカスケード更新または削除の実行が決まるような、参照整合性に依存するトリガに対して有効です。

トリガの実行中にエラーが発生すると、トリガを起動した操作そのものがエラーになります。INSERT、UPDATE、DELETE はアトミック・オペレーションです。これらがエラーになると、トリガの結果とトリガが起動したプロシージャを含め、その文のすべての結果がキャンセルされます。「アトミックな複合文」 900 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. DBA 権限または RESOURCE 権限のあるユーザとしてデータベースに接続します。
2. 左ウィンドウ枠で、**[トリガ]** をクリックします。
3. **[ファイル] - [新規] - [トリガ]** を選択します。
4. **トリガ作成ウィザード**の指示に従います。
5. コードを完了するには、右ウィンドウ枠で、**[SQL]** タブをクリックします。

♦ 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. DBA 権限のあるユーザとしてデータベースに接続します。トリガに関連付けられているテーブルに対して ALTER パーミッションも必要です。
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ることを示すエラーが表示されます。

このトリガは、Employees テーブルに新しいローが追加されると起動されます。2001 年 6 月 6 日以降の生年月日に対応する新しいローを検知し、エラーにします。

フレーズ REFERENCING NEW AS new_employee は、トリガ・コード中の文がエイリアス new_employee を使用して、新しいローのデータを参照できるようにします。

エラーが発生すると、トリガ元の文、およびトリガによる前の変更内容がすべて取り消されます。

Employees テーブルに複数のローを追加する INSERT 文の場合は、新しいローごとに check_birth_date トリガが起動されます。どれか 1 つのローでトリガが失敗すると、INSERT 文のすべての結果がロールバックされます。

ローを追加した後でなく、追加する前にトリガが起動されるようにするには、例文の 2 行目を次のように変更します。

```
BEFORE INSERT ON Employees
```

REFERENCING NEW 句は追加されるローの値を参照します。この句はトリガが起動されるタイミング (BEFORE と AFTER) には影響されません。

トリガではなく、宣言参照整合性を使用したり、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。

参照

- 「SQL Anywhere データベース接続」 『SQL Anywhere サーバ - データベース管理』
- 「COMMIT 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「ROLLBACK TO SAVEPOINT 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「CREATE TRIGGER 文」 『SQL Anywhere サーバ - SQL リファレンス』
- 「複合文の使用」 900 ページ

トリガの実行

トリガは指定したテーブルに INSERT、UPDATE、DELETE が行われたときに、自動的に実行されます。ロー・レベル・トリガは、ローが影響を受けるごとに起動され、文レベル・トリガは、文全体が一度に起動されます。

INSERT、UPDATE、DELETE がトリガを起動すると、トリガのタイプ (BEFORE または AFTER) によって、次の順序で操作が行われます。

1. BEFORE トリガが起動します。
2. 追加などの操作そのものが実行されます。
3. 参照動作を行います。
4. AFTER トリガが起動します。

注意
CREATE TRIGGER 文を使用してトリガを作成するときにトリガのタイプを指定しなかった場合は、デフォルトで AFTER になります。

手順の途中で、プロシージャまたはトリガの内部で処理されないエラーが発生すると、それより以前の手順は取り消され、それ以降の手順は実行されません。トリガを起動した操作そのものも失敗となります。

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。

♦ **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. DBA 権限のあるユーザまたはトリガ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、**[トリガ]** をダブルクリックします。
3. トリガを選択します。
4. 次のいずれかの方法を使用して、トリガを変更します。
   - 右ウィンドウ枠で、**[SQL]** タブをクリックします。
   - トリガを右クリックして、**[新しいウィンドウで編集]** を選択します。

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

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

     データベース・ドキュメント・ジェネレータを使用して、SQL Anywhere データベースをドキュメント化する場合、これらのコメントを出力に含めるオプションがあります。「データベースのドキュメント化」『SQL Anywhere サーバ - データベース管理』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. DBA 権限のあるユーザまたはトリガの所有者として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。

参照

- 「データベース・オブジェクトのプロパティの設定」 16 ページ
- 「SQL Anywhere データベース接続」『SQL Anywhere サーバ - データベース管理』
- 「Sybase Central を使用したストアド・プロシージャの変換」 713 ページ
- 「ALTER TRIGGER 文」『SQL Anywhere サーバ - SQL リファレンス』

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. DBA 権限のあるユーザまたはトリガ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、**[トリガ]** をダブルクリックします。
3. トリガを選択して、**[編集]** - **[削除]** を選択します。
4. **[はい]** をクリックします。

♦ **トリガを削除するには、次の手順に従います (SQL の場合)。**

1. DBA 権限のあるユーザまたはトリガの所有者として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、mytrigger トリガをデータベースから削除する文を示します。

```
DROP TRIGGER mytrigger;
```

参照

- 「SQL Anywhere データベース接続」『SQL Anywhere サーバ - データベース管理』
- 「DROP TRIGGER 文」『SQL Anywhere サーバ - SQL リファレンス』

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、データベース・サーバがテーブルが見つからないことを示すメッセージを返します。

また、user_1 はトリガに指定された操作を実行するためのパーミッションを持っていなければなりません。

参照

- 「データベースのパーミッションと権限の概要」『SQL Anywhere サーバ - データベース管理』

トリガに関する詳細情報

トリガの理解しにくい一面として、複数のトリガが同じトリガ元アクションの影響を受ける場合に、これらのトリガが実行される順序があります。競合するトリガが実行されるかどうか、また実行される場合の順序は、トリガのタイプ (BEFORE、INSTEAD OF、または AFTER) とトリガのスコープ (ローレベルまたは文レベル) の 2 点で決まります。

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はスキップされ、代わりに指定されたアクションが実行される点で BEFORE トリガや AFTER トリガと異なります。

INSTEAD OF トリガに固有の機能や制限を次に示します。

- INSTEAD OF トリガは、特定のテーブルのトリガ・イベントごとに 1 つだけ指定できます。
- INSTEAD OF トリガはテーブルまたはビューに定義できます。ただし、INSTEAD OF トリガはマテリアライズド・ビューには定義できません。マテリアライズド・ビューには INSERT 文、DELETE 文、UPDATE 文などの DML 操作を実行できないからです。
- INSTEAD OF トリガを定義するときは、ORDER 句または WHEN 句は指定できません。
- INSTEAD OF トリガは、UPDATE OF *column-list* トリガ・イベントには定義できません。「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 トリガが再帰を実行するかどうかは、トリガのターゲットがベース・テーブルであるか、ビューであるかで異なります。ビューの場合は再帰が発生しますが、ベース・テーブルの場合は発生しません。たとえば、INSTEAD OF トリガによって、トリガが定義されているベース・テーブルに対して DML 操作が実行されても、これらの操作でトリガは実行されません (BEFORE トリガや AFTER トリガを含む)。ターゲットがビューの場合は、ビューに対して実行された操作ですべてのトリガが実行されます。
- テーブルに INSTEAD OF トリガが定義されている場合、ON EXISTING 句を含む INSERT 文をテーブルに対して実行できません。実行しようとすると、SQLE_INSTEAD_TRIGGER エラーが返されます。
- WITH CHECK OPTION を指定して定義されているか、WITH CHECK OPTION を指定して定義されている別のビューにネストされており、INSTEAD OF INSERT トリガが定義されているビューに対して 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、派生の関係で更新不可能なビューに対して 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 INSERT、UPDATE、DELETE 文を実行する意味を定義します。たとえば、次のビューを作成するとします。

```
CREATE VIEW V1 ( Surname, GivenName, State )
  AS SELECT DISTINCT Surname, GivenName, State
     FROM Contacts;
```

DISTINCT キーワードによって、派生の関係で V1 が更新不可能になるので、V1 のローは削除できません。つまり、データベース・サーバでは、V1 からローを削除する意味を明確に特定できません。ただし、V1 への削除操作を実装する INSTEAD OF DELETE トリガを定義することはできます。たとえば、次のトリガでは、指定した Surname、GivenName、State のローが Contacts から削除されるときに、そのすべてのローが削除されます。

```
CREATE TRIGGER V1_Delete
  INSTEAD OF DELETE ON V1
  REFERENCING OLD AS old_row
  FOR EACH ROW
BEGIN
    DELETE FROM Contacts
      WHERE Surname = old_row.Surname
        AND GivenName = old_row.GivenName
        AND State = old_row.State
END;
```

V1_Delete トリガを定義すると、V1 からローを削除できます。さらに、V1 で INSERT 文と UPDATE 文を実行させる他の INSTEAD OF トリガを定義することもできます。

INSTEAD OF DELETE トリガが定義されたビューが別のビューにネストされている場合は、DELETE に対する更新可能性を確認するためにベース・テーブルのように処理されます。INSERT 操作と UPDATE 操作も同様です。前の例の続きで、別のビューを作成します。

```
CREATE VIEW V2 ( Surname, GivenName ) AS
  SELECT Surname, GivenName from V1;
```

V1_Delete トリガがなければ、V2 からローを削除することはできません。これは、派生の関係で V1 は更新不可能なので、V2 も更新不可能になるからです。しかし、V1 に INSTEAD OF DELETE トリガを定義すれば、V2 からローを削除できます。V2 からローが削除されるたびに V1 からローが削除され、V1_Delete トリガが実行されます。

INSTEAD OF トリガをネストされているビューに定義する場合は、これらのトリガが実行されるときに、意図しない影響がある可能性があるため、注意してください。意図する動作を明示的にするには、ネストされているビューを参照するすべてのビューに INSTEAD OF トリガを定義します。

次のトリガを V2 に定義し、DELETE 文の意図した動作を実行できます。

```
CREATE TRIGGER V2_Delete
  INSTEAD OF DELETE ON V2
  REFERENCING OLD AS old_row
  FOR EACH ROW
BEGIN
    DELETE FROM Contacts
      WHERE Surname = old_row.Surname
```

```
        AND GivenName = old_row.GivenName
END;
```

V2_Delete トリガによって、V1 に対する INSTEAD OF DELETE トリガが削除または変更されても、V2 に対する削除操作の動作は変わりません。

バッチの概要

バッチは一緒に送信されてグループとして次々に実行される一連の SQL 文です。プロシージャ (CASE、IF、LOOP など) で使われる制御文はバッチでも使えます。バッチが BEGIN/END で囲まれた複合文で構成される場合、ホスト変数、変数のローカル宣言、カーソル、テンポラリ・テーブル、例外を含めることもできます。ホスト変数参照は、次の制限付きでバッチ内で使用できます。

- ホスト変数を参照できるのはバッチ内の 1 文だけです。
- ホスト変数を使用する文の前に、結果セットを返す文を入れることはできません。

バッチの使用を明確に示すために、BEGIN/END を使うことをおすすめします。

バッチ内の文はセミコロンで区切ることができます。その場合、バッチは Watcom-SQL 構文に準拠しています。文を区切るためにセミコロンを使用しない複数文のバッチは、Transact-SQL 構文に準拠します。バッチの構文によって、バッチ内で使用できる文とバッチ内でのエラーの処理方法が決まります。Transact-SQL バッチの詳細については、「Transact-SQL のバッチの概要」 712 ページを参照してください。

多くの点で、バッチはストアド・プロシージャに似ていますが、いくつかの違いがあります。

- バッチには名前がありません。
- バッチにはパラメータを使用できません。
- バッチは永続的にデータベースに保存されません。
- バッチは異なる接続で共有できません。

簡単なバッチは、デリミタのない SQL 文のセットで、次の行に go という単語が続きます。次の例では、Eastern Sales という部署を作成し、Massachusetts のすべての営業担当者を Eastern Sales に転送します。これは Transact-SQL バッチの例です。

```
INSERT
INTO Departments ( DepartmentID, DepartmentName )
VALUES ( 220, 'Eastern Sales' )

UPDATE Employees
SET DepartmentID = 220
WHERE DepartmentID = 200
AND State = 'MA'

COMMIT
go
```

go という単語は Interactive SQL によって認識され、前の文は 1 つのバッチとしてサーバに送信されます。「複数の SQL 文の実行」 『SQL Anywhere サーバ - データベース管理』を参照してください。

次の例は見た目は似ていますが、Interactive SQL での処理はまったく異なります。この例では、Transact-SQL 構文を使用しません。各文はセミコロンで区切られています。Interactive SQL はセミコロンで区切られた各文を個別にサーバに送信します。この場合は、バッチとしては処理されません。


```
INSERT
INTO Departments ( DepartmentID, DepartmentName )
VALUES ( 220, 'Eastern Sales' );

UPDATE Employees
SET DepartmentID = 220
WHERE DepartmentID = 200
AND State = 'MA';

COMMIT;
```

Interactive SQL でバッチとして処理するには、BEGIN ... END を使用して、複合文に変更します。次の構文は、前の例を修正したものです。複合文に含まれる 3 つの文は、バッチとしてサーバに送信されます。

```
BEGIN
  INSERT
  INTO Departments ( DepartmentID, DepartmentName )
  VALUES ( 220, 'Eastern Sales' );

  UPDATE Employees
  SET DepartmentID = 220
  WHERE DepartmentID = 200
  AND State = 'MA';

  COMMIT;
END
```

この例の場合、サーバがバッチと個別のどちらで文を実行しても結果は同じになります。ただし、結果が異なる場合もあります。次の例を考えます。

```
DECLARE @CurrentID INTEGER;
SET @CurrentID = 207;
SELECT Surname FROM Employees
  WHERE EmployeeID=@CurrentID;
```

Interactive SQL を使用してこの例を実行すると、データベース・サーバが変数が見つからないことを示すエラーを返します。このエラーは、Interactive SQL が 3 つの文を個別にサーバに送信するために発生します。これらの文はバッチとしては実行されません。このようなエラーに対処するには、複合文を使用して Interactive SQL が強制的に 3 つの文をバッチとしてサーバに送信するようにします。次の例では、複合文を使用しています。

```
BEGIN
  DECLARE @CurrentID INTEGER;
  SET @CurrentID = 207;
  SELECT Surname FROM Employees
    WHERE EmployeeID=@CurrentID;
END
```

一連の文を BEGIN と END で囲んだ場合、Interactive SQL は強制的に文をバッチとして処理します。

IF 文は複合文の一例です。Interactive SQL は、次の文を 1 つのバッチとしてサーバに送信します。

```
IF EXISTS(   SELECT *
          FROM SYSTAB
          WHERE table_name='Employees' )
THEN
```

```
    SELECT   Surname AS LastName,
             GivenName AS FirstName
    FROM Employees;
    SELECT Surname, GivenName
    FROM Customers;
    SELECT Surname, GivenName
    FROM Contacts;
  ELSE
    MESSAGE 'The Employees table does not exist'
    TO CLIENT;
  END IF
```

別の方法で SQL 文を作成して実行した場合、この例は適用されません。たとえば、ODBC を使用するアプリケーションでは、セミコロンで区切られた文をバッチとして作成および実行できます。

Interactive SQL の文とサーバ向けの SQL 文が混在している場合は、注意が必要です。次の例は、Interactive SQL の文と SQL 文を混合する場合の問題を示します。この例では、Interactive SQL の OUTPUT 文が複合文に組み込まれているので、その他のすべての文と一緒にバッチとしてサーバに送信され、構文エラーが発生します。

```
  IF EXISTS(   SELECT *
               FROM SYSTAB
               WHERE table_name='Employees' )
  THEN
    SELECT   Surname AS LastName,
             GivenName AS FirstName
    FROM Employees;
    SELECT Surname, GivenName
    FROM Customers;
    SELECT Surname, GivenName
    FROM Contacts;
    OUTPUT TO 'c:¥¥temp¥¥query.txt';
  ELSE
    MESSAGE 'The Employees table does not exist'
    TO CLIENT;
  END IF
```

正しい OUTPUT 文の例は、次のとおりです。

```
  IF EXISTS(   SELECT *
               FROM SYSTAB
               WHERE table_name='Employees' )
  THEN
    SELECT   Surname AS LastName,
             GivenName AS FirstName
    FROM Employees;
    SELECT Surname, GivenName
    FROM Customers;
    SELECT Surname, GivenName
    FROM Contacts;
  ELSE
    MESSAGE 'The Employees table does not exist'
    TO CLIENT;
  END IF;
  OUTPUT TO 'c:¥¥temp¥¥query.txt';
```


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複数あります。使用可能な制御文は、次のとおりです。

制御文	構文
複合文 「BEGIN 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, ...)

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」な文とは、完全に実行されるか、まったく実行されない文のことです。たとえば、何千ものローを更新する UPDATE 文では、たくさんのローを更新した後にエラーが発生することがあります。文が完了しないと、変更されたすべてのローが元の状態に戻ります。したがって、UPDATE 文はアトミックです。

複合文でないすべての SQL 文はアトミックです。BEGIN キーワードの後にキーワード ATOMIC を追加して、複合文をアトミックにすることができます。

```
BEGIN ATOMIC
  UPDATE Employees
  SET ManagerID = 501
  WHERE EmployeeID = 467;
   UPDATE Employees
  SET BirthDate = 'bad_data';
END
```

この例の 2 つの UPDATE 文は、アトミックな複合文の一部です。これら 2 つの文は、1 つの文として更新を完了するか、両方ともエラーになります。最初の UPDATE 文はエラーなしで完了するとします。次の UPDATE 文は BirthDate カラムに割り当てた値を日付に変換できないため、エラーになります。

このアトミックな複合文はエラーになり、UPDATE 文の結果は両方とも取り消されます。現在実行中のトランザクションがコミットされても、この複合文中の文は両方ともその効果をもたらしません。

アトミックな複合文が成功すると、現在実行中のトランザクションがコミットされた場合のみ、複合文中で実行された変更は有効になります。アトミックな複合文が成功しても、その文で発生したトランザクションがロールバックされた場合は、アトミックな複合文もロールバックされます。アトミックな複合文の開始時に、セーブポイントが設定されます。文でエラーが発生すると、そのセーブポイントにロールバックされます。

アトミックな複合文がオートコミット (非連鎖) モードで実行されると、文の実行が完了するまでコミット・モードが手動 (連鎖) に変更されます。手動モードでは、アトミックな複文内で DML 文を実行しても、即座に COMMIT や ROLLBACK は実行されません。アトミックな複合文が正常に完了すると、COMMIT 文が実行されます。正常に完了しない場合は ROLLBACK 文が実行されます。オートコミット動作の詳細については、「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』と「オートコミットの動作を制御する」『SQL Anywhere サーバ - プログラミング』を参照してください。

COMMIT 文と ROLLBACK 文、および一部の ROLLBACK TO SAVEPOINT 文は、アトミックな複合文内で使用できません。「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」 921 ページを参照してください。

アトミックな複合文内の一部の文しか実行されない場合もあります。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」 917 ページを参照してください。

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」 900 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();
```

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 通りの形式のどちらでも使用できます。

CREATE PROCEDURE 文の引数リストの末尾にオプションのパラメータがある場合、これらは CALL 文で省略できます。次に示すのは、INOUT パラメータを 3 つ持つプロシージャの例です。

```
CREATE PROCEDURE SampleProcedure(
    INOUT var1 INT DEFAULT 1,
            INOUT var2 int DEFAULT 2,
            INOUT var3 int DEFAULT 3 )
...
```

この例では、プロシージャを呼び出す環境で、プロシージャに渡す数値を格納するための変数を 3 つ設定してあるものと想定しています。

```
CREATE VARIABLE V1 INT;
CREATE VARIABLE V2 INT;
CREATE VARIABLE V3 INT;
```

次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」 882 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。**

● Interactive SQL で、次のように入力します。

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

Name
Fran Whitney
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」 876 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 文を使用して OUT パラメータに値を割り当てて返します。

```
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;
```

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」 907 ページを参照してください。

クエリの結果、選択されたローが存在しない場合は、警告が返ります。

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。

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 句のリスト内の変数の数は、SELECT 文のリスト内の変数の数に一致しなければなりません。データ型が一致しない場合は、可能であれば自動的にデータ型の変換が行われます。
- RESULT 句は CREATE PROCEDURE 文の一部で、コマンド・デリミタは付きません。
- SELECT 文のリスト内の変数の名前は、RESULT 句のリスト内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]** ウィンドウの **[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。
- 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]** ウィンドウの **[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 (Sybase Central の場合)。**

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. 左ウィンドウ枠でデータベースを選択し、**[ファイル]** - **[Interactive SQL を開く]** を選択します。
3. Interactive SQL で、**[ツール]** - **[オプション]** を選択します。
4. **[SQL Anywhere]** をクリックします。
5. **[結果]** タブで、**[すべての結果セットを表示]** を選択します。
6. **[OK]** をクリックします。

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 ();
```

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。

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」 907 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 )
      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。

- 「**ローが見つかりません。**」という例外が宣言されます。この例外は、プロシージャの後でクエリの結果のループが完了するときに通知されます。

  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」 913 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。

この TopCompanyValue プロシージャでの LOOP 文は標準的な形式で、最後のローを処理して終了します。FOR ループを使うと、このプロシージャはさらに簡潔になります。FOR 文は 1 つの文に、上記のプロシージャのいくつかの要素を組み込みます。

```
CREATE PROCEDURE TopCustomerValue2(
    OUT TopCompany CHAR(36),
    OUT TopValue INT )
```

```
BEGIN
  -- 1. Initialize the TopValue variable
  SET TopValue = 0;
  -- 2. Do the For Loop
  FOR CompanyFor AS ThisCompany
    CURSOR FOR
    SELECT CompanyName AS ThisName,
      CAST( sum( SalesOrderItems.Quantity *
          Products.UnitPrice ) AS INTEGER )
      AS ThisValue
    FROM Customers
      INNER JOIN SalesOrders
      INNER JOIN SalesOrderItems
      INNER JOIN Products
    GROUP BY ThisName
  DO
    IF ThisValue > TopValue THEN
      SET TopCompany = ThisName;
      SET TopValue = ThisValue;
    END IF;
  END FOR;
END;
```

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
      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 );
```

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、およびその意味の詳細については、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。この特別な値は、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」 910 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」 917 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」 916 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 通りの方法があります。

- **デフォルトのエラー処理**　プロシージャかトリガがエラーを起こしたときに、呼び出しを行った環境にエラー・コードが返されます。
- **ON EXCEPTION RESUME**　CREATE PROCEDURE 文に ON EXCEPTION RESUME 句が含まれていれば、プロシージャはエラーを起こした箇所の次の文から実行を再開します。

  ON EXCEPTION RESUME を使用するプロシージャの正確な動作は、on_tsql_error オプション設定によって指定します。「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;

CALL OuterProc();
```

Interactive SQL の [メッセージ] タブに次のメッセージが表示されます。

```
Hello from OuterProc.
Hello from InnerProc.
```

InnerProc 内の DECLARE 文は、サーバが認識しているエラー条件に関連して事前に定義された SQLSTATE 値のうち、1 つの記号名を宣言します。

MESSAGE 文は Interactive SQL の [メッセージ] タブにメッセージを送ります。

SIGNAL 文は InnerProc プロシージャ内から、エラーであることを外部に知らせる役割を持ちます。

InnerProc の SIGNAL 文の後の文は実行されず、InnerProc はすぐに呼び出しを行った環境 (この場合はプロシージャ OuterProc) に制御を戻します。OuterProc の CALL 文の後に続く文は実行されません。エラーは呼び出しを行った環境に戻され、処理されます。たとえば、Interactive SQL はエラー・メッセージをメッセージ・ウィンドウに表示してエラーの処理を行います。

TRACEBACK 関数はエラーが起きたときに実行していた文をリストします。Interactive SQL から TRACEBACK を使うには、次の文を入力します。

```
SELECT TRACEBACK();
```

ON EXCEPTION RESUME を使ったエラー処理

ON EXCEPTION RESUME 文が CREATE PROCEDURE 文に含まれていた場合、エラーが起きると、次の文が検査されます。その文がエラーを処理する場合、プロシージャの実行が続行され、エラーが発生した文の次の文を実行します。エラーが発生したとき、呼び出しを行った環境に制御を戻しません。

on_tsql_error オプション設定を使用して、ON EXCEPTION RESUME を使用するプロシージャの動作を変更できます。「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

次に示すプロシージャは、アプリケーションからプロシージャ OuterProc を呼び出し、OuterProc が InnerProc を呼び出して、そこでエラーが発生した場合の処理を示します。例文は、この項の最初で使用したプロシージャを基にしています。

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE OuterProc()
ON EXCEPTION RESUME
BEGIN
  DECLARE res CHAR(5);
  MESSAGE 'Hello from OuterProc.' TO CLIENT;
  CALL InnerProc();
  SET res=SQLSTATE;
  IF res='52003' THEN
    MESSAGE 'SQLSTATE set to ',
      res, ' in OuterProc.' TO CLIENT;
  END IF
END;

CREATE PROCEDURE InnerProc()
ON EXCEPTION RESUME
BEGIN
  DECLARE column_not_found
    EXCEPTION FOR SQLSTATE '52003';
  MESSAGE 'Hello from InnerProc.' TO CLIENT;
  SIGNAL column_not_found;
  MESSAGE 'SQLSTATE set to ',
  SQLSTATE, ' in InnerProc.' TO CLIENT;
END;

CALL OuterProc();
```

Interactive SQL の [メッセージ] タブに、次のメッセージが表示されます。

```
Hello from OuterProc.
Hello from InnerProc.
SQLSTATE set to 52003 in OuterProc.
```

実行パスを次に示します。

1. OuterProc は InnerProc を実行して呼び出します。

2. InnerProc では、SIGNAL 文がエラーを通知します。
3. MESSAGE 文はエラー処理文ではないので、制御は OuterProc に返され、メッセージは表示されません。
4. OuterProc では、エラーに続く文が SQLSTATE の値を **res** という変数に割り当てます。これはエラー処理文なので、実行は継続され、OuterProc メッセージが表示されます。

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。

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」 913 ページで使用したプロシージャを基にしています。

この場合、SIGNAL 文はエラーではなく、「**ローが見つかりません。**」という警告を生成します。

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE OuterProc()
BEGIN
  MESSAGE 'Hello from OuterProc.' TO CLIENT;
  CALL InnerProc();
  MESSAGE 'SQLSTATE set to ',
    SQLSTATE,' in OuterProc.' TO CLIENT;
END;
CREATE PROCEDURE InnerProc()
BEGIN
  DECLARE row_not_found
    EXCEPTION FOR SQLSTATE '02000';
  MESSAGE 'Hello from InnerProc.' TO CLIENT;
  SIGNAL row_not_found;
  MESSAGE 'SQLSTATE set to ',
  SQLSTATE, ' in InnerProc.' TO CLIENT;
END;

CALL OuterProc();
```

Interactive SQL の [メッセージ] タブに、次のメッセージが表示されます。

```
Hello from OuterProc.
Hello from InnerProc.
SQLSTATE set to 02000 in InnerProc.
SQLSTATE set to 00000 in OuterProc.
```

両方のプロシージャとも、警告によって SQLSTATE に値 (02000) が設定された後も実行を続けました。

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 部分で定義します。「複合文の使用」 900 ページを参照してください。

複合文でエラーが起きた場合、例外ハンドラが実行されます。警告では、例外ハンドラは実行されません。ネストされた複合文の中でエラーが起きた場合、また、複合文の中から起動されたプロシージャやトリガの中でエラーが起きた場合は、例外処理コードが実行されます。

中断エラー SQL_INTERRUPT、SQLSTATE 57014 の例外ハンドラには、ROLLBACK や ROLLBACK TO SAVEPOINT などの中断のできない文だけを含めます。例外ハンドラに、接続の中断時に呼び出される中断可能な文を含めると、データベース・サーバは最初の中断可能な文で例外ハンドラを停止し、中断エラーを返します。

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」 913 ページで使用したプロシージャを基にして、例外ハンドラの処理を示します。

この例では、InnerProc プロシージャ中の「**カラムが見つかりません。**」というエラーを処理するために、コードが追加されています。

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE OuterProc()
BEGIN
  MESSAGE 'Hello from OuterProc.' TO CLIENT;
  CALL InnerProc();
  MESSAGE 'SQLSTATE set to ',
    SQLSTATE,' in OuterProc.' TO CLIENT
END;
CREATE PROCEDURE InnerProc()
BEGIN
  DECLARE column_not_found
   EXCEPTION FOR SQLSTATE '52003';
  MESSAGE 'Hello from InnerProc.' TO CLIENT;
 SIGNAL column_not_found;
 MESSAGE 'Line following SIGNAL.' TO CLIENT;
  EXCEPTION
    WHEN column_not_found THEN
    MESSAGE 'Column not found handling.' TO CLIENT;
      WHEN OTHERS THEN
    RESIGNAL ;
END;

CALL OuterProc();
```

Interactive SQL の **[メッセージ]** タブに、次のメッセージが表示されます。

```
Hello from OuterProc.
Hello from InnerProc.
Column not found handling.
SQLSTATE set to 00000 in OuterProc.
```

EXCEPTION 句は例外ハンドラを宣言します。これ以降の文はエラーが起きないかぎり実行されません。WHEN 句は例外名 (DECLARE 文で宣言) と、その例外が起こったときに実行する文を

定義します。WHEN OTHERS THEN 句はその前の WHEN 句以外で例外が起こったときに実行する文を定義します。

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。

追加の注意事項

- InnerProc の SIGNAL 文に続く行ではなく、EXCEPTION ハンドラが実行されます。
- 「**カラムが見つかりません。**」というエラーが発生したため、エラー処理のための MESSAGE 文が実行され、SQLSTATE は 0 にリセットされます (エラー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。**」という例外に対するエラー処理コードが単なる RESIGNAL 文である場合、制御は OuterProc に戻され、SQLSTATE の値は 52003 に設定されたままになります。これは、InnerProc にはエラー処理コードがないのと同じです。OuterProc にはこれ以外のエラー処理コードはないため、プロシージャはエラーになります。

例外処理とアトミックな複合文

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、アトミックな複合文内の一部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。

次の例は、ネストされた複合文をどのように使用してフローを制御するかを示します。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」 913 ページの例に使用したものに基づいています。

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE InnerProc()
BEGIN
  BEGIN
    DECLARE column_not_found
    EXCEPTION FOR SQLSTATE VALUE '52003';
    MESSAGE 'Hello from InnerProc' TO CLIENT;
    SIGNAL column_not_found;
    MESSAGE 'Line following SIGNAL' TO CLIENT
```



```
    EXCEPTION
      WHEN column_not_found THEN
        MESSAGE 'Column not found handling' TO
        CLIENT;
      WHEN OTHERS THEN
        RESIGNAL;
  END;
  MESSAGE 'Outer compound statement' TO CLIENT;
END;

CALL InnerProc();
```

Interactive SQL の [メッセージ] タブに、次のメッセージが表示されます。

```
Hello from InnerProc
Column not found handling
Outer compound statement
```

エラーを引き起こした SIGNAL 文が検出されると、制御は複合文の例外ハンドラに渡されて、「**カラムが見つかりません。**」というメッセージが出力されます。次に制御は外部複合文に渡され、「Outer compound statement」メッセージが出力されます。

内部複合文で「**カラムが見つかりません。**」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。

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内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 [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内の SQL 文は現在のトランザクションの一部です。1 つのトランザクション内で複数のプロシージャを呼び出すことや、1 つのプロシージャ内に複数のトランザクションを持つことができます。

アトミックな文内では COMMIT と ROLLBACK は許可されません。トリガはアトミックな文である INSERT、UPDATE、DELETE によって起動さ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内では許可されません。

プロシージャまたはトリガではセーブポイントを使用できますが、ROLLBACK TO SAVEPOINT 文はアトミック・オペレーションが開始される以前のセーブポイントを参照することはできません。また、アトミック・オペレーション内のすべてのセーブポイントは、その操作が終了したときに解除されます。

参照

- 「トランザクションと独立性レベルの使用」 117 ページ
- 「アトミックな複合文」 900 ページ
- 「トランザクション内のセーブポイント」 122 ページ

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性のチェック

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

プロシージャ内の各文はセミコロンで終わります。最後の文はセミコロンがなくてもかまいませんが、各文の後ろにはセミコロンを付けることを習慣にしてください。

CREATE PROCEDURE 文は本体である複合文と、RESULT 指定の両方を含みます。キーワード BEGIN または END の後と、RESULT 句の後には、セミコロンは必要ありません。

プロシージャ内のテーブル名

プロシージャがテーブルを参照する場合は、テーブル名に必ず所有者 (作成者) の名前をプレフィクスとして付けてください。

プロシージャがテーブルを参照するときは、プロシージャ作成者のグループ・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 (直接的または間接的な) 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」 931 ページを参照してください。

プロシージャ、トリガ、イベント、バッチで使用できる文

バッチにはほとんどの SQL 文を使用できますが、次の文は使用できません。

- ALTER DATABASE (構文 3 および 4)
- CONNECT
- CREATE DATABASE
- CREATE DECRYPTED FILE
- CREATE ENCRYPTED FILE
- DISCONNECT
- DROP CONNECTION
- DROP DATABASE
- FORWARD TO
- INPUT、OUTPUT などの Interactive SQL 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」 921 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、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、関連オブジェクトを使用可能な状態に保ちながら、その内容を難読化して読み取れないようにします。また、アンロードして、別のデータベースに再ロードすることもできます。

修正を元に戻すことはできません。修正すると、オブジェクトの元のテキストが削除されます。オブジェクトの元のソースをデータベースの外部に保存しておく必要があります。

デバッガによるデバッグでは、プロシージャ定義が表示されず、プロシージャ・プロファイリングにもソースが表示されません。

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.SYSUSER u
      WHERE p.creator = u.user_id
      AND p.creator NOT IN (0,1,3)
   DO
      MESSAGE 'altering ' || proc_name;
      EXECUTE IMMEDIATE 'ALTER PROCEDURE "' ||
         user_name || '"."' || proc_name
         || '" SET HIDDEN'
   END FOR
END;
```

参照

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

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

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**　SQL プロシージャのブレークポイントで実行が停止したときに、クエリを実行できます。これによって、テンポラリ・テーブルに保持される中間結果を参照したり、ベース・テーブルの値をチェックしたり、クエリ実行プランを参照したりできます。

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。

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。

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。**

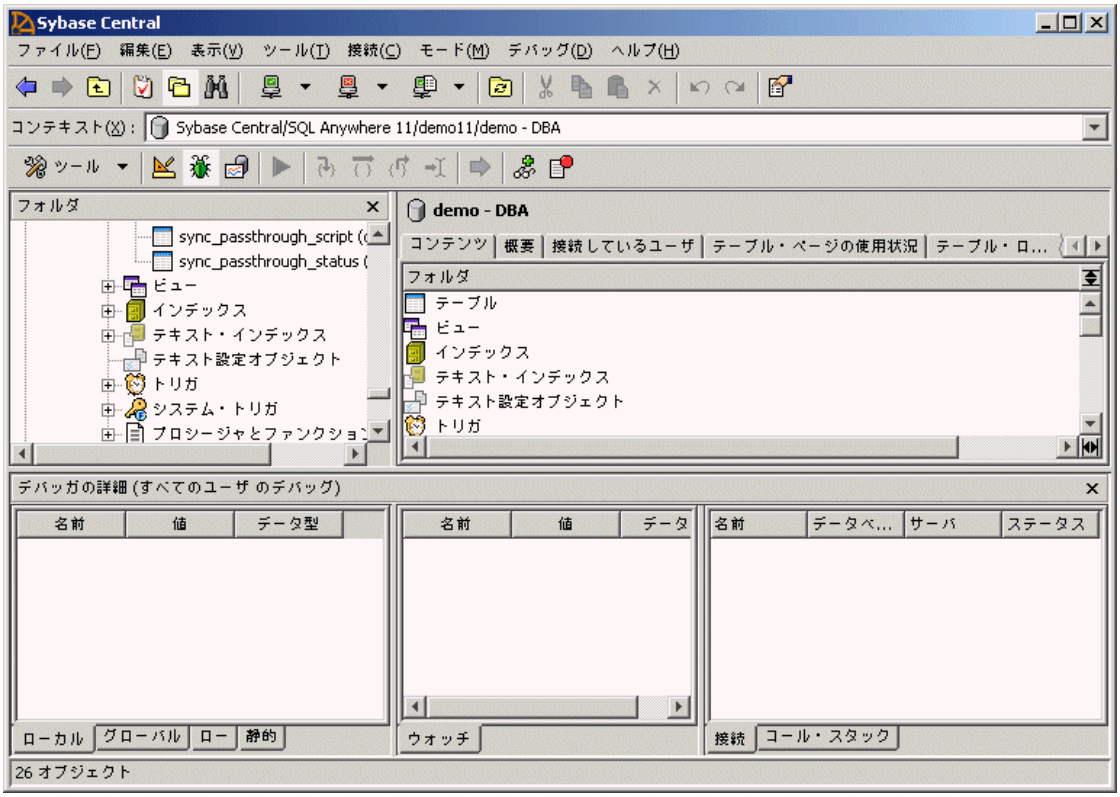
1. このチュートリアルで使用するサンプル・データベースのコピーを保存するディレクトリ (*c:¥demodb* など) を作成します。
2. サンプル・データベースを *samples-dir¥demo.db* から *c:¥demodb* にコピーし、名前を変更します。

   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。
3. **[スタート] - [プログラム] - [SQL Anywhere 11] - [Sybase Central]** をクリックします
4. **[接続] - [SQL Anywhere 11 に接続]** を選択します。
5. **[ODBC データ・ソース名]** フィールドで、サンプル・データベース名を入力します。
6. **[OK]** をクリックします。
7. **[モード] - [デバッグ]** を選択します。
8. **[デバッグするユーザを指定してください]** フィールドに * と入力し、**[OK]** をクリックします。

   別のユーザをデバッグする場合は、[デバッグ] モードを終了してから、もう一度 [デバッグ] モードに入ってください。

   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。

   ユーザ名を入力すると、そのユーザ名を使用した接続に関する情報が取得され、**[接続]** タブに表示されます。



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。

♦ **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]** をダブルクリックします。
2. **[Debugger_Tutorial (GROUPO)]** を右クリックし、**[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、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. **[モード]** - **[デバッグ]** を選択します。
2. 右ウィンドウ枠で、**[Debugger_Tutorial (GROUPO)]** をダブルクリックします。
3. 右ウィンドウ枠で、次の文を検索します。

   ```
   OPEN cursor_this_customer;
   ```
4. ブレークポイントを追加するには、左側にあるグレーの縦線の領域をクリックします。ブレークポイントは、赤い円で表示されます。
5. 左ウィンドウ枠で、**[Debugger_Tutorial (GROUPO)]** を右クリックし、**[Interactive SQL から実行]** を選択します。

   Sybase Central の **[接続]** タブに、ブレークポイントを示す黄色の矢印が表示されます。
6. **[デバッガの詳細]** ウィンドウで **[ローカル]** タブをクリックし、プロシージャ内のローカル変数とその現在の値とデータ型のリストを表示します。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。
7. [F11] キーを押して、プロシージャをスクロールします。次の行に到達すると、変数の値が変化します。

   ```
   IF this_value > top_value THEN
   ```
8. [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. **[デバッガの詳細]** ウィンドウで、**[ローカル]** タブをクリックします。
2. **[Top_Value]** 変数をクリックし、**[値]** フィールドに **3000** と入力します。
3. **[This_Value]** 変数の **[値]** フィールドが 3000 よりも大きな値になるまで [F11] キーを繰り返し押します。
4. ブレークポイントをクリックして、グレーにします。
5. [F5] キーを押して、プロシージャを実行します。

   [Interactive SQL] ウィンドウが再び表示されます。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. 右ウィンドウ枠で、次の文を検索します。

   ```
   OPEN cursor_this_customer;
   ```

3. top_value 変数を初期化する新しい行を入力します。

   ```
   SET top_value = 0;
   ```

4. **[ファイル]** - **[保存]** を選択します。
5. プロシージャを再度実行し、Interactive SQL に正しい結果が表示されることを確認します。

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。

ブレークポイントの活用

ブレークポイントでは、デバッガがソース・コードの実行を中断するタイミングを制御します。

[デバッグ] モードで実行中に、接続がブレークポイントに到達した場合、その動作は選択した接続によって変わります。

- 接続を選択していない場合は、接続は自動的に選択され、プロシージャのソースが表示されてデバッグされます。
- 接続が選択済みで、その接続がブレークポイントに到達した接続と同じである場合は、プロシージャのソースが表示されてデバッグされます。
- 接続は選択済みであるが、その接続がブレークポイントに到達した接続とは異なる場合、ウィンドウが開き、ブレークポイントに到達した接続に変更するように求めるメッセージが表示されます。

ブレークポイントの設定

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. 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. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックします。
3. プロシージャを選択します。
4. **[モード]** - **[デバッグ]** を選択します。
5. **[デバッグするユーザを指定してください]** フィールドに * と入力してすべてのユーザをデバッグするか、デバッグ対象となるデータベース・ユーザの名前を入力します。
6. 右ウィンドウ枠で、ブレークポイントを挿入する行をクリックします。

   クリックした行にカーソルが表示されます。
7. [F9] キーを押します。

   コード行の左側に赤い円が表示されます。

♦ **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]** - **[ブレークポイント]** を選択します。
2. **[新規]** をクリックします。
3. **[プロシージャ]** リストからプロシージャを選択します。
4. 必要に応じて、**[条件]** フィールドと **[カウント]** フィールドにも入力します。

条件とは、ブレークポイントが処理を中断するために TRUE に評価されなければならない SQL 式です。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、実行が停止するまでの、ブレークポイントのヒット数です。0 を指定した場合、そのブレークポイントで常に実行が停止されます。

5. **[OK]** をクリックします。ブレークポイントが、プロシージャ内の最初の実行可能な文に設定されます。

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]** ウィンドウ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. 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. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックします。
3. プロシージャを選択します。
4. **[モード]** - **[デバッグ]** を選択します。
5. 右ウィンドウ枠で、編集する行の左側にあるブレークポイント・インジケータをクリックします。ブレークポイントがアクティブから非アクティブに変わります。

♦ **ブレークポイント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] ウィンドウの場合)。**

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. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックします。
3. プロシージャを選択します。
4. **[モード]** - **[デバッグ]** を選択します。
5. **[デバッグ]** - **[ブレークポイント]** を選択します。
6. ブレークポイントを選択し、**[編集]**、**[無効にする]**、または **[削除]** をクリックします。
7. **[閉じる]** をクリックします。

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. 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. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックします。
3. プロシージャを選択します。
4. **[モード]** - **[デバッグ]** を選択します。
5. **[デバッグ]** - **[ブレークポイント]** を選択します。
6. 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。
7. **[条件]** リストで、条件をクリックします。たとえば、特定のユーザ ID からの接続にのみ適用するようにブレークポイントを設定するには、次の条件を入力します。

   ```
   CURRENT USER='user-name'
   ```

   user-name は、ブレークポイントをアクティブにする対象のユーザ ID です。
8. **[OK]** をクリックしてから、**[閉じる]** をクリックします。

変数の活用

デバッガでは、コードをステップしながら変数の動作を表示して編集できます。デバッガには、ストアド・プロシージャで使用するさまざまな種類の変数を表示する **[デバッガの詳細]** ウィンドウ枠が用意されています。**[デバッガの詳細]** ウィンドウ枠は、Sybase Central をデバッグ・モードで実行しているときに、Sybase Central の最下部に表示されます。

変数値の表示

♦ **変数の値を表示するには、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. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックします。
3. プロシージャを選択します。
4. **[モード]** - **[デバッグ]** を選択します。
5. **[デバッグするユーザを指定してください]** フィールドに * と入力してすべてのユーザをデバッグするか、デバッグ対象となるデータベース・ユーザの名前を入力します。
6. 右ウィンドウ枠で、ブレークポイントを挿入する行をクリックします。

   クリックした行にカーソルが表示されます。
7. [F9] キーを押します。

   コード行の左側に赤い円が表示されます。
8. **[デバッガの詳細]** ウィンドウ枠で、**[ローカル]** タブをクリックします。
9. 左ウィンドウ枠で、プロシージャを右クリックし、**[Interactive SQL から実行]** を選択します。**[ローカル]** タブに、変数とその値が表示されます。

グローバル変数の表示

グローバル変数は SQL Anywhere によって定義され、現在の接続、データベース、その他の設定に関する情報がグローバル変数に格納されます。グローバル変数は、**[グローバル]** タブの **[デバッガの詳細]** ウィンドウ枠に表示されます。

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、**[ロー]** タブの **[デバッガの詳細]** ウィンドウ枠に表示されます。

トリガの詳細については、「トリガの概要」 886 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。

呼び出しスタックの表示

ネストされたプロシージャをデバッグするときに、呼び出しの順番を検査すると便利なことがあります。プロシージャのリストは、**[コール・スタック]** タブで参照できます。

♦ **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. 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. 左ウィンドウ枠で、**[プロシージャとファンクション]** をダブルクリックします。
3. プロシージャを選択します。
4. **[モード]** - **[デバッグ]** を選択します。
5. **[デバッグするユーザを指定してください]** フィールドに * と入力してすべてのユーザをデバッグするか、デバッグ対象となるデータベース・ユーザの名前を入力します。
6. 右ウィンドウ枠で、ブレークポイントを挿入する行をクリックします。

   クリックした行にカーソルが表示されます。
7. [F9] キーを押します。

   コード行の左側に赤い円が表示されます。
8. **[デバッガの詳細]** ウィンドウ枠で、**[ローカル]** タブをクリックします。
9. 左ウィンドウ枠で、プロシージャを右クリックし、**[Interactive SQL から実行]** を選択します。
10. **[デバッガの詳細]** ウィンドウ枠で、**[コール・スタック]** タブをクリックします。

    [コール・スタック] タブにプロシージャの名前が表示されます。リストの一番上に現在のプロシージャが表示されます。そのプロシージャを呼び出したプロシージャがそのすぐ下に表示されます。

接続の活用

[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。

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」 935 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。

用語解説

用語解説

Adaptive Server Anywhere (ASA)

SQL Anywhere Studio のリレーショナル・データベース・サーバ・コンポーネントであり、主に、モバイル環境と埋め込み環境、または小規模および中規模のビジネス用のサーバとして使用されます。バージョン 10.0.0 で、Adaptive Server Anywhere は SQL Anywhere サーバに、SQL Anywhere Studio は SQL Anywhere にそれぞれ名前が変更されました。

参照：「SQL Anywhere」 948 ページ。

Carrier

Mobile Link システム・テーブルまたは Notifier プロパティ・ファイルに保存される Mobile Link オブジェクトで、システム起動同期で使用される通信業者に関する情報が含まれます。

参照：「サーバ起動同期」 953 ページ。

DB 領域

データ用の領域をさらに作成する追加のデータベース・ファイルです。1 つのデータベースは 13 個までのファイルに保管されます (初期ファイル 1 つと 12 の DB 領域)。各テーブルは、そのインデックスとともに、単一のデータベース・ファイルに含まれている必要があります。CREATE DBSPACE という SQL コマンドで、新しいファイルをデータベースに追加できます。

参照：「データベース・ファイル」 957 ページ。

DBA 権限

ユーザに、データベース内の管理作業を許可するレベルのパーミッションです。DBA ユーザにはデフォルトで DBA 権限が与えられています。

参照：「データベース管理者 (DBA)」 957 ページ。

EBF

Express Bug Fix の略です。Express Bug Fix は、1 つ以上のバグ・フィックスが含まれる、ソフトウェアのサブセットです。これらのバグ・フィックスは、更新のリリース・ノートにリストされます。バグ・フィックス更新を適用できるのは、同じバージョン番号を持つインストール済みのソフトウェアに対してだけです。このソフトウェアについては、ある程度のテストが行われているとはいえ、完全なテストが行われたわけではありません。自分自身でソフトウェアの妥当性を確かめるまでは、アプリケーションとともにこれらのファイルを配布しないでください。

Embedded SQL

C プログラム用のプログラミング・インタフェースです。SQL Anywhere の Embedded SQL は ANSI と IBM 規格に準拠して実装されています。

FILE

SQL Remote のレプリケーションでは、レプリケーション・メッセージのやりとりのために共有ファイルを使うメッセージ・システムのことです。これは特定のメッセージ送信システムに頼らずにテストやインストールを行うのに便利です。

参照：「レプリケーション」 965 ページ。

grant オプション

他のユーザにパーミッションを許可できるレベルのパーミッションです。

iAnywhere JDBC ドライバ

iAnywhere JDBC ドライバでは、pure Java である jConnect JDBC ドライバに比べて何らかの有利なパフォーマンスや機能を備えた JDBC ドライバが提供されます。ただし、このドライバは pure Java ソリューションではありません。iAnywhere JDBC ドライバは一般に推奨されるドライバです。

参照：

- 「JDBC」 945 ページ
- 「jConnect」 945 ページ

InfoMaker

レポート作成とデータ管理用のツールです。洗練されたフォーム、レポート、グラフ、クロスタブ、テーブルを作成できます。また、これらを基本的な構成要素とするアプリケーションも作成できます。

Interactive SQL

データベース内のデータの変更や問い合わせ、データベース構造の修正ができる、SQL Anywhere のアプリケーションです。Interactive SQL では、SQL 文を入力するためのウィンドウ枠が表示されます。また、クエリの進捗情報や結果セットを返すウィンドウ枠も表示されます。

JAR ファイル

Java アーカイブ・ファイルです。Java のアプリケーションで使用される 1 つ以上のパッケージの集合からなる圧縮ファイルのフォーマットです。Java プログラムをインストールしたり実行したりするのに必要なリソースが 1 つの圧縮ファイルにすべて収められています。

Java クラス

Java のコードの主要な構造単位です。これはプロシージャや変数の集まりで、すべてがある一定のカテゴリに関連しているためグループ化されたものです。

jConnect

JavaSoft JDBC 標準を Java で実装したものです。これにより、Java 開発者は多層／異機種環境でもネイティブなデータベース・アクセスができます。iAnywhere JDBC ドライバは一般に推奨されるドライバです。

参照：

- 「JDBC」 945 ページ
- 「iAnywhere JDBC ドライバ」 944 ページ

JDBC

Java Database Connectivity の略です。Java アプリケーションからリレーショナル・データにアクセスすることを可能にする SQL 言語プログラミング・インタフェースです。推奨 JDBC ドライバは、iAnywhere JDBC ドライバです。

参照：

- 「jConnect」 945 ページ
- 「iAnywhere JDBC ドライバ」 944 ページ

Listener

Mobile Link サーバ起動同期に使用される、dblsn という名前のプログラムです。Listener はリモート・デバイスにインストールされ、Push 通知を受け取ったときにデバイス上でアクションが開始されるように設定されます。

参照：「サーバ起動同期」 953 ページ。

LTM

LTM (Log Transfer Manager) は、Replication Agent とも呼ばれます。Replication Server と併用することで、LTM はデータベース・トランザクション・ログを読み込み、コミットされた変更を Sybase Replication Server に送信します。

参照：「Replication Server」 948 ページ。

Mobile Link

Ultra Light と SQL Anywhere のリモート・データベースを統合データベースと同期させるために設計された、セッションベース同期テクノロジです。

参照：

- 「統合データベース」 972 ページ
- 「同期」 972 ページ
- 「Ultra Light」 949 ページ

Mobile Link クライアント

2 種類の Mobile Link クライアントがあります。SQL Anywhere リモート・データベース用の Mobile Link クライアントは、dbmlsync コマンド・ライン・ユーティリティです。Ultra Light リモート・データベース用の Mobile Link クライアントは、Ultra Light ランタイム・ライブラリに組み込まれています。

Mobile Link サーバ

Mobile Link 同期を実行する、mlsrv11 という名前のコンピュータ・プログラムです。

Mobile Link システム・テーブル

Mobile Link の同期に必要なシステム・テーブルです。Mobile Link 設定スクリプトによって、Mobile Link 統合データベースにインストールされます。

Mobile Link モニタ

Mobile Link の同期をモニタするためのグラフィカル・ツールです。

Mobile Link ユーザ

Mobile Link ユーザは、Mobile Link サーバに接続するのに使用されます。Mobile Link ユーザをリモート・データベースに作成し、統合データベースに登録します。Mobile Link ユーザ名はデータベース・ユーザ名から完全に独立しています。

Notifier

Mobile Link サーバ起動同期に使用されるプログラムです。Notifier は Mobile Link サーバに統合されており、統合データベースに Push 要求がないか確認し、Push 通知を送信します。

参照：

- 「サーバ起動同期」 953 ページ
- 「Listener」 945 ページ

ODBC

Open Database Connectivity の略です。データベース管理システムに対する Windows の標準的なインタフェースです。ODBC は、SQL Anywhere がサポートするインタフェースの 1 つです。

ODBC アドミニストレータ

Windows オペレーティング・システムに付属している Microsoft のプログラムです。ODBC データ・ソースの設定に使用します。

ODBC データ・ソース

ユーザが ODBC からアクセスするデータと、そのデータにアクセスするために必要な情報の仕様です。

PDB

Palm のデータベース・ファイルです。

PowerDesigner

データベース・モデリング・アプリケーションです。これを使用すると、データベースやデータ・ウェアハウスの設計に対する構造的なアプローチが可能となります。SQL Anywhere には、PowerDesigner の Physical Data Model コンポーネントが付属します。

PowerJ

Java アプリケーション開発に使用する Sybase 製品です。

Push 通知

QAnywhere では、メッセージ転送を開始するよう QAnywhere クライアントに対して指示するために、サーバから QAnywhere クライアントに配信される特殊なメッセージです。Mobile Link サーバ起動同期では、Push 要求データや内部情報を含むデバイスに Notifer から配信される特殊なメッセージです。

参照：

- 「QAnywhere」 947 ページ
- 「サーバ起動同期」 953 ページ

Push 要求

Mobile Link サーバ起動同期において、Push 通知をデバイスに送信する必要があるかどうかを判断するために Notifier が確認する、結果セット内の値のローです。

参照：「サーバ起動同期」 953 ページ。

QAnywhere

アプリケーション間メッセージング (モバイル・デバイス間メッセージングやモバイル・デバイスとエンタープライズの間のメッセージングなど) を使用すると、モバイル・デバイスや無線デバイスで動作しているカスタム・プログラムと、集中管理されているサーバ・アプリケーションとの間で通信できます。

QAnywhere Agent

QAnywhere では、クライアント・デバイス上で動作する独立のプロセスのことです。クライアント・メッセージ・ストアをモニタリングし、メッセージを転送するタイミングを決定します。

REMOTE DBA 権限

SQL Remote では、Message Agent (dbremote) で必要なパーミッションのレベルを指します。Mobile Link では、SQL Anywhere 同期クライアント (dbmlsync) で必要なパーミッションのレベルを指します。Message Agent (dbremote) または同期クライアントがこの権限のあるユーザとして接続した場合、DBA のフル・アクセス権が与えられます。Message Agent (dbremote) または同期クライアント (dbmlsync) から接続しない場合、このユーザ ID にはパーミッションは追加されません。

参照：「DBA 権限」 943 ページ。

Replication Agent

参照：「LTM」 945 ページ。

Replication Server

SQL Anywhere と Adaptive Server Enterprise で動作する、Sybase による接続ベースのレプリケーション・テクノロジです。Replication Server は、少数のデータベース間でほぼリアルタイムのレプリケーションを行うことを目的に設計されています。

参照：「LTM」 945 ページ。

SQL

リレーショナル・データベースとの通信に使用される言語です。SQL は ANSI により標準が定義されており、その最新版は SQL-2003 です。SQL は、公認されてはいませんが、Structured Query Language の略です。

SQL Anywhere

SQLAnywhere のリレーショナル・データベース・サーバ・コンポーネントであり、主に、モバイル環境と埋め込み環境、または小規模および中規模のビジネス用のサーバとして使用されます。SQL Anywhere は、SQL Anywhere RDBMS、Ultra Light RDBMS、Mobile Link 同期ソフトウェア、その他のコンポーネントを含むパッケージの名前でもあります。

SQL Remote

統合データベースとリモート・データベース間で双方向レプリケーションを行うための、メッセージベースのデータ・レプリケーション・テクノロジです。統合データベースとリモート・データベースは、SQL Anywhere である必要があります。

SQL ベースの同期

Mobile Link では、Mobile Link イベントを使用して、テーブル・データを Mobile Link でサポートされている統合データベースに同期する方法のことです。SQL ベースの同期では、SQL を直接使用したり、Java と .NET 用の Mobile Link サーバ API を使用して SQL を返すことができます。

SQL 文

DBMS に命令を渡すために設計された、SQL キーワードを含む文字列です。

参照：

- 「スキーマ」 955 ページ
- 「SQL」 948 ページ
- 「データベース管理システム (DBMS)」 957 ページ

Sybase Central

SQL Anywhere データベースのさまざまな設定、プロパティ、ユーティリティを使用できる、グラフィカル・ユーザ・インタフェースを持つデータベース管理ツールです。Mobile Link などの他の iAnywhere 製品を管理する場合にも使用できます。

SYS

システム・オブジェクトの大半を所有する特別なユーザです。一般のユーザは SYS でログインできません。

Ultra Light

小型デバイス、モバイル・デバイス、埋め込みデバイス用に最適化されたデータベースです。対象となるプラットフォームとして、携帯電話、ポケットベル、パーソナル・オーガナイザなどが挙げられます。

Ultra Light ランタイム

組み込みの Mobile Link 同期クライアントを含む、インプロセス・リレーショナル・データベース管理システムです。Ultra Light ランタイムは、Ultra Light の各プログラミング・インタフェースで使用されるライブラリと、Ultra Light エンジンの両方に含まれます。

Windows

Windows Vista、Windows XP、Windows 200x などの、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 CE

「Windows Mobile」 949 ページを参照してください。

Windows Mobile

Microsoft がモバイル・デバイス用に開発したオペレーティング・システムのファミリです。

アーティクル

Mobile Link または SQL Remote では、テーブル全体もしくはテーブル内のカラムとローのサブセットを表すデータベース・オブジェクトを指します。アーティクルの集合がパブリケーションです。

参照：

- 「レプリケーション」 965 ページ
- 「パブリケーション」 960 ページ

アップロード

同期中に、リモート・データベースから統合データベースにデータが転送される段階です。

アトミックなトランザクション

完全に処理されるかまったく処理されないことが保証される 1 つのトランザクションです。エラーによってアトミックなトランザクションの一部が処理されなかった場合は、データベースが一貫性のない状態になるのを防ぐために、トランザクションがロールバックされます。

アンロード

データベースをアンロードすると、データベースの構造かデータ、またはその両方がテキスト・ファイルにエクスポートされます (構造は SQL コマンド・ファイルに、データはカンマ区切りの ASCII ファイルにエクスポートされます)。データベースのアンロードには、アンロード・ユーティリティを使用します。

また、UNLOAD 文を使って、データから抜粋した部分だけをアンロードできます。

イベント・モデル

Mobile Link では、同期を構成する、begin_synchronization や download_cursor などの一連のイベントのことです。イベントは、スクリプトがイベント用に作成されると呼び出されます。

インクリメンタル・バックアップ

トランザクション・ログ専用のバックアップです。通常、フル・バックアップとフル・バックアップの間に使用します。

参照：「トランザクション・ログ」 959 ページ。

インデックス

ベース・テーブルにある 1 つ以上のカラムに関連付けられた、キーとポインタのソートされたセットです。テーブルの 1 つ以上のカラムにインデックスが設定されていると、パフォーマンスが向上します。

ウィンドウ

分析関数の実行対象となるローのグループです。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、入力の現在のローについて計算を実行する必要があるローの数や範囲を含むように移動します。ウィンドウ構成の主な利点は、追加のクエリを実行しなくても、結果をグループ化して分析する機会が増えることです。

エージェント ID

参照：「クライアント・メッセージ・ストア ID」 952 ページ。

エンコード

文字コードとも呼ばれます。エンコードは、文字セットの各文字が情報の 1 つまたは複数のバイトにマッピングされる方法のことで、一般的に 16 進数で表現されます。UTF-8 はエンコードの例です。

参照：

- 「文字セット」 973 ページ
- 「コード・ページ」 953 ページ
- 「照合」 970 ページ

オブジェクト・ツリー

Sybase Central では、データベース・オブジェクトの階層を指します。オブジェクト・ツリーの最上位には、現在使用しているバージョンの Sybase Central がサポートするすべての製品が表示されます。それぞれの製品を拡張表示すると、オブジェクトの下位ツリーが表示されます。

参照：「Sybase Central」 949 ページ。

カーソル

結果セットへの関連付けに名前を付けたもので、プログラミング・インタフェースからローにアクセスしたり更新したりするときに使用します。SQL Anywhere では、カーソルはクエリ結果内で前方や後方への移動をサポートします。カーソルは、カーソル結果セット (通常 SELECT 文で定義される) とカーソル位置の 2 つの部分から構成されます。

参照：

- 「カーソル結果セット」 952 ページ
- 「カーソル位置」 951 ページ

カーソル位置

カーソル結果セット内の 1 つのローを指すポインタ。

参照：

- 「カーソル」 951 ページ
- 「カーソル結果セット」 952 ページ

カーソル結果セット

カーソルに関連付けられたクエリから生成されるローのセットです。

参照：

- 「カーソル」 951 ページ
- 「カーソル位置」 951 ページ

クエリ

データベースのデータにアクセスしたり、そのデータを操作したりする SQL 文や SQL 文のグループです。

参照：「SQL」 948 ページ。

クライアント／サーバ

あるアプリケーション (クライアント) が別のアプリケーション (サーバ) に対して情報を送受信するソフトウェア・アーキテクチャのことです。通常この 2 種類のアプリケーションは、ネットワークに接続された異なるコンピュータ上で実行されます。

クライアント・メッセージ・ストア

QAnywhere では、メッセージを保管するリモート・デバイスにある SQL Anywhere データベースのことです。

クライアント・メッセージ・ストア ID

QAnywhere では、Mobile Link リモート ID のことです。これによって、クライアント・メッセージ・ストアがユニークに識別されます。

グローバル・テンポラリ・テーブル

明示的に削除されるまでデータ定義がすべてのユーザに表示されるテンポラリ・テーブルです。グローバル・テンポラリ・テーブルを使用すると、各ユーザが、1 つのテーブルのまったく同じインスタンスを開くことができます。デフォルトでは、コミット時にローが削除され、接続終了時にもローが削除されます。

参照：

- 「テンポラリ・テーブル」 958 ページ
- 「ローカル・テンポラリ・テーブル」 966 ページ

ゲートウェイ

Mobile Link システム・テーブルまたは Notifier プロパティ・ファイルに保存される Mobile Link オブジェクトで、システム起動同期用のメッセージの送信方法に関する情報が含まれます。

参照：「サーバ起動同期」 953 ページ。

コード・ページ

コード・ページは、文字セットの文字を数値表示 (通常 0 ～ 255 の整数) にマッピングするエンコードです。Windows Code Page 1252 などのコード・ページがあります。このマニュアルの目的上、コード・ページとエンコードは同じ意味で使用されます。

参照：

- 「文字セット」 973 ページ
- 「エンコード」 951 ページ
- 「照合」 970 ページ

コマンド・ファイル

SQL 文で構成されたテキスト・ファイルです。コマンド・ファイルは手動で作成できますが、データベース・ユーティリティによって自動的に作成することもできます。たとえば、dbunload ユーティリティを使うと、指定されたデータベースの再構築に必要な SQL 文で構成されたコマンド・ファイルを作成できます。

サーバ・メッセージ・ストア

QAnywhere では、サーバ上のリレーショナル・データベースです。このデータベースは、メッセージを、クライアント・メッセージ・ストアまたは JMS システムに転送されるまで一時的に格納します。メッセージは、サーバ・メッセージ・ストアを介して、クライアント間で交換されます。

サーバ管理要求

XML 形式の QAnywhere メッセージです。サーバ・メッセージ・ストアを管理したり、QAnywhere アプリケーションをモニタリングするために QAnywhere システム・キューに送信されます。

サーバ起動同期

Mobile Link サーバから Mobile Link 同期を開始する方法です。

サービス

Windows オペレーティング・システムで、アプリケーションを実行するユーザ ID がログオンしていないときにアプリケーションを実行する方法です。

サブクエリ

別の SELECT 文、INSERT 文、UPDATE 文、DELETE 文、または別のサブクエリの中にネストされた SELECT 文です。

相関とネストの 2 種類のサブクエリがあります。

サブスクリプション

Mobile Link 同期では、パブリケーションと Mobile Link ユーザ間のクライアント・データベース内のリンクであり、そのパブリケーションが記述したデータの同期を可能にします。

SQL Remote レプリケーションでは、パブリケーションとリモート・ユーザ間のリンクのことで、これによりリモート・ユーザはそのパブリケーションの更新内容を統合データベースとの間で交換できます。

参照：

- 「パブリケーション」 960 ページ
- 「Mobile Link ユーザ」 946 ページ

システム・オブジェクト

SYS または dbo が所有するデータベース・オブジェクトです。

システム・テーブル

SYS または dbo が所有するテーブルです。メタデータが格納されています。システム・テーブル (データ辞書テーブルとしても知られています) はデータベース・サーバが作成し管理します。

システム・ビュー

すべてのデータベースに含まれているビューです。システム・テーブル内に格納されている情報をわかりやすいフォーマットで示します。

ジョイン

指定されたカラムの値を比較することによって 2 つ以上のテーブルにあるローをリンクする、リレーショナル・システムでの基本的な操作です。

ジョイン・タイプ

SQL Anywhere では、クロス・ジョイン、キー・ジョイン、ナチュラル・ジョイン、ON 句を使ったジョインの 4 種類のジョインが使用されます。

参照：「ジョイン」 954 ページ。

ジョイン条件

ジョインの結果に影響を及ぼす制限です。ジョイン条件は、JOIN の直後に ON 句か WHERE 句を挿入して指定します。ナチュラル・ジョインとキー・ジョインについては、SQL Anywhere がジョイン条件を生成します。

参照：

- 「ジョイン」 954 ページ
- 「生成されたジョイン条件」 971 ページ

スキーマ

テーブル、カラム、インデックス、それらの関係などを含んだデータベース構造です。

スクリプト

Mobile Link では、Mobile Link のイベントを処理するために記述されたコードです。スクリプトは、業務上の要求に適合するように、データ交換をプログラム的に制御します。

参照：「イベント・モデル」 950 ページ。

スクリプト・バージョン

Mobile Link では、同期を作成するために同時に適用される、一連の同期スクリプトです。

スクリプトベースのアップロード

Mobile Link では、ログ・ファイルを使用した方法の代わりとなる、アップロード処理のカスタマイズ方法です。

ストアド・プロシージャ

ストアド・プロシージャは、データベースに保存され、データベース・サーバに対する一連の操作やクエリを実行するために使用される SQL 命令のグループです。

スナップショット・アイソレーション

読み込み要求を発行するトランザクション用のデータのコミットされたバージョンを返す、独立性レベルの種類です。SQL Anywhere では、スナップショット、文のスナップショット、読み込み専用文のスナップショットの 3 つのスナップショットの独立性レベルがあります。スナップショット・アイソレーションが使用されている場合、読み込み処理は書き込み処理をブロックしません。

参照：「独立性レベル」 973 ページ。

セキュア機能

データベース・サーバが起動されたときに、そのデータベース・サーバで実行されているデータベースでは使用できないように -sf オプションによって指定される機能です。

セッション・ベースの同期

統合データベースとリモート・データベースの両方でデータ表現の一貫性が保たれる同期です。Mobile Link はセッション・ベースです。

ダイレクト・ロー・ハンドリング

Mobile Link では、テーブル・データを Mobile Link でサポートされている統合データベース以外のソースに同期する方法のことです。アップロードとダウンロードの両方をダイレクト・ロー・ハンドリングで実装できます。

参照：

- 「統合データベース」 972 ページ
- 「SQL ベースの同期」 949 ページ

ダウンロード

同期中に、統合データベースからリモート・データベースにデータが転送される段階です。

チェックサム

データベース・ページを使用して記録されたデータベース・ページのビット数の合計です。チェックサムを使用すると、データベース管理システムは、ページがディスクに書き込まれるときに数が一致しているかを確認することで、ページの整合性を検証できます。数が一致した場合は、ページが正常に書き込まれたとみなされます。

チェックポイント

データベースに加えたすべての変更内容がデータベース・ファイルに保存されるポイントです。通常、コミットされた変更内容はトランザクション・ログだけに保存されます。

データ・キューブ

同じ結果を違う方法でグループ化およびソートされた内容を各次元に反映した、多次元の結果セットです。データ・キューブは、セルフジョイン・クエリと相関サブクエリを必要とするデータの複雑な情報を提供します。データ・キューブは OLAP 機能の一部です。

データベース

プライマリ・キーと外部キーによって関連付けられているテーブルの集合です。これらのテーブルでデータベース内の情報が保管されます。また、テーブルとキーによってデータベースの構造が定義されます。データベース管理システムでこの情報にアクセスします。

参照：

- 「外部キー」 967 ページ
- 「プライマリ・キー」 962 ページ
- 「データベース管理システム (DBMS)」 957 ページ
- 「リレーショナル・データベース管理システム (RDBMS)」 965 ページ

データベース・オブジェクト

情報を保管したり受け取ったりするデータベース・コンポーネントです。テーブル、インデックス、ビュー、プロシージャ、トリガはデータベース・オブジェクトです。

データベース・サーバ

データベース内にある情報へのすべてのアクセスを規制するコンピュータ・プログラムです。SQL Anywhere には、ネットワーク・サーバとパーソナル・サーバの 2 種類のサーバがあります。

データベース・ファイル

データベースは 1 つまたは複数のデータベース・ファイルに保持されます。まず、初期ファイルがあり、それに続くファイルは DB 領域と呼ばれます。各テーブルは、それに関連付けられているインデックスとともに、単一のデータベース・ファイルに含まれている必要があります。

参照：「DB 領域」 943 ページ。

データベース管理システム (DBMS)

データベースを作成したり使用したりするためのプログラムの集合です。

参照：「リレーショナル・データベース管理システム (RDBMS)」 965 ページ。

データベース管理者 (DBA)

データベースの管理に必要なパーミッションを持つユーザです。DBA は、データベース・スキーマのあらゆる変更や、ユーザやグループの管理に対して、全般的な責任を負います。データベース管理者のロールはデータベース内に自動的に作成されます。その場合、ユーザ ID は DBA であり、パスワードは sql です。

データベース所有者 (dbo)

SYS が所有しないシステム・オブジェクトを所有する特別なユーザです。

参照：

- 「データベース管理者 (DBA)」 957 ページ
- 「SYS」 949 ページ

データベース接続

クライアント・アプリケーションとデータベース間の通信チャネルです。接続を確立するためには有効なユーザ ID とパスワードが必要です。接続中に実行できるアクションは、そのユーザ ID に付与された権限によって決まります。

データベース名

サーバがデータベースをロードするとき、そのデータベースに指定する名前です。デフォルトのデータベース名は、初期データベース・ファイルのルート名です。

参照：「データベース・ファイル」 957 ページ。

データ型

CHAR や NUMERIC などのデータのフォーマットです。ANSI SQL 規格では、サイズ、文字セット、照合に関する制限もデータ型に組み込めます。

参照：「ドメイン」 958 ページ。

データ操作言語 (DML)

データベース内のデータの操作に使う SQL 文のサブセットです。DML 文は、データベース内のデータを検索、挿入、更新、削除します。

データ定義言語 (DDL)

データベース内のデータの構造を定義するときに使う SQL 文のサブセットです。DDL 文は、テーブルやユーザなどのデータベース・オブジェクトを作成、変更、削除できます。

デッドロック

先へ進めない場所に一連のトランザクションが到達する状態です。

デバイス・トラッキング

Mobile Link サーバ起動同期において、デバイスを特定する Mobile Link のユーザ名を使用して、メッセージのアドレスを指定できる機能です。

参照：「サーバ起動同期」 953 ページ。

テンポラリ・テーブル

データを一時的に保管するために作成されるテーブルです。グローバルとローカルの 2 種類があります。

参照：

- 「ローカル・テンポラリ・テーブル」 966 ページ
- 「グローバル・テンポラリ・テーブル」 952 ページ

ドメイン

適切な位置に精度や小数点以下の桁数を含み、さらにオプションとしてデフォルト値や CHECK 条件などを含んでいる、組み込みデータ型のエイリアスです。ドメインには、通貨データ型のように SQL Anywhere が事前に定義したものもあります。ユーザ定義データ型とも呼ばれます。

参照：「データ型」 958 ページ。

トランザクション

作業の論理単位を構成する一連の SQL 文です。1 つのトランザクションは完全に処理されるかまったく処理されないかのどちらかです。SQL Anywhere は、ロック機能のあるトランザクション処理をサポートしているので、複数のトランザクションが同時にデータベースにアクセスしてもデータを壊すことがありません。トランザクションは、データに加えた変更を永久なものにする COMMIT 文か、トランザクション中に加えられたすべての変更を元に戻す ROLLBACK 文のいずれかで終了します。

トランザクション・ログ

データベースに対するすべての変更内容が、変更された順に格納されるファイルです。パフォーマンスを向上させ、データベース・ファイルが破損した場合でもデータをリカバリできます。

トランザクション・ログ・ミラー

オプションで設定できる、トランザクション・ログ・ファイルの完全なコピーのことで、トランザクション・ログと同時に管理されます。データベースの変更がトランザクション・ログへ書き込まれると、トランザクション・ログ・ミラーにも同じ内容が書き込まれます。

ミラー・ファイルは、トランザクション・ログとは別のデバイスに置いてください。一方のデバイスに障害が発生しても、もう一方のログにリカバリのためのデータが確保されます。

参照：「トランザクション・ログ」 959 ページ。

トランザクション単位の整合性

Mobile Link で、同期システム全体でのトランザクションの管理を保証します。トランザクション全体が同期されるか、トランザクション全体がまったく同期されないかのどちらかになります。

トリガ

データを修正するクエリをユーザが実行すると、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。

参照：

- 「ロー・レベルのトリガ」 966 ページ
- 「文レベルのトリガ」 973 ページ
- 「整合性」 970 ページ

ネットワーク・サーバ

共通ネットワークを共有するコンピュータからの接続を受け入れるデータベース・サーバです。

参照：「パーソナル・サーバ」 960 ページ。

ネットワーク・プロトコル

TCP/IP や HTTP などの通信の種類です。

パーソナル・サーバ

クライアント・アプリケーションが実行されているコンピュータと同じマシンで実行されているデータベース・サーバです。パーソナル・データベース・サーバは、単一のコンピュータ上で単一のユーザが使用しますが、そのユーザからの複数の同時接続をサポートできます。

パッケージ

Java では、それぞれが互いに関連のあるクラスの集合を指します。

ハッシュ

ハッシュは、インデックスのエントリをキーに変換する、インデックスの最適化のことです。インデックスのハッシュの目的は、必要なだけの実際のロー・データをロー ID に含めることで、インデックスされた値を特定するためのローの検索、ロード、アンパックという負荷の高い処理を避けることです。

パフォーマンス統計値

データベース・システムのパフォーマンスを反映する値です。たとえば、CURRREAD 統計値は、データベース・サーバが要求したファイル読み込みのうち、現在まだ完了していないものの数を表します。

パブリケーション

Mobile Link または SQL Remote では、同期されるデータを識別するデータベース・オブジェクトのことです。Mobile Link では、クライアント上にのみ存在します。1 つのパブリケーションは複数のアーティクルから構成されています。SQL Remote ユーザは、パブリケーションに対してサブスクリプションを作成することによって、パブリケーションを受信できます。Mobile Link ユーザは、パブリケーションに対して同期サブスクリプションを作成することによって、パブリケーションを同期できます。

参照：

- 「レプリケーション」 965 ページ
- 「アーティクル」 950 ページ
- 「パブリケーションの更新」 960 ページ

パブリケーションの更新

SQL Remote レプリケーションでは、単一のデータベース内の 1 つまたは複数のパブリケーションに対して加えられた変更のリストを指します。パブリケーションの更新は、レプリケーション・メッセージの一部として定期的にリモート・データベースへ送られます。

参照：

- 「レプリケーション」 965 ページ
- 「パブリケーション」 960 ページ

パブリッシャ

SQL Remote レプリケーションでは、レプリケートできる他のデータベースとレプリケーション・メッセージを交換できるデータベースの単一ユーザを指します。

参照：「レプリケーション」 965 ページ。

ビジネス・ルール

実世界の要求に基づくガイドラインです。通常ビジネス・ルールは、検査制約、ユーザ定義データ型、適切なトランザクションの使用により実装されます。

参照：

- 「制約」 970 ページ
- 「ユーザ定義データ型」 964 ページ

ヒストグラム

ヒストグラムは、カラム統計のもっとも重要なコンポーネントであり、データ分散を表します。SQL Anywhere は、ヒストグラムを維持して、カラムの値の分散に関する統計情報をオプティマイザに提供します。

ビット配列

ビット配列は、一連のビットを効率的に保管するのに使用される配列データ構造の種類です。ビット配列は文字列に似てますが、使用される要素は文字ではなく 0 (ゼロ) と 1 になります。ビット配列は、一般的にブール値の文字列を保持するのに使用されます。

ビュー

データベースにオブジェクトとして格納される SELECT 文です。ビューを使用すると、ユーザは 1 つまたは複数のテーブルのローやカラムのサブセットを参照できます。ユーザが特定のテーブルやテーブルの組み合わせのビューを使うたびに、テーブルに保持されているデータから再計算されます。ビューは、セキュリティの目的に有用です。またデータベース情報の表示を調整して、データへのアクセスが簡単になるようにする場合も役立ちます。

ファイルベースのダウンロード

Mobile Link では、ダウンロードがファイルとして配布されるデータの同期方法であり、同期変更のオフライン配布を可能にします。

ファイル定義データベース

Mobile Link では、ダウンロード・ファイルの作成に使用される SQL Anywhere データベースのことです。

参照：「ファイルベースのダウンロード」 961 ページ。

フェールオーバ

アクティブなサーバ、システム、またはネットワークで障害や予定外の停止が発生したときに、冗長な (スタンバイ) サーバ、システム、またはネットワークに切り替えることです。フェールオーバは自動的に発生します。

プライマリ・キー

テーブル内のすべてのローをユニークに識別する値を持つカラムまたはカラムのリストです。

参照：「外部キー」 967 ページ。

プライマリ・キー制約

プライマリ・キーのカラムに対する一意性制約です。テーブルにはプライマリ・キー制約を 1 つしか設定できません。

参照：

- 「制約」 970 ページ
- 「検査制約」 969 ページ
- 「外部キー制約」 968 ページ
- 「一意性制約」 967 ページ
- 「整合性」 970 ページ

プライマリ・テーブル

外部キー関係でプライマリ・キーを含むテーブルです。

プラグイン・モジュール

Sybase Central で、製品にアクセスしたり管理したりする方法です。プラグインは、通常、インストールすると Sybase Central にもインストールされ、自動的に登録されます。プラグインは、多くの場合、Sybase Central のメイン・ウィンドウに最上位のコンテナとして、その製品名 (たとえば SQL Anywhere) で表示されます。

参照：「Sybase Central」 949 ページ。

フル・バックアップ

データベース全体をバックアップすることです。オプションでトランザクション・ログのバックアップも可能です。フル・バックアップには、データベース内のすべての情報が含まれており、システム障害やメディア障害が発生した場合の保護として機能します。

参照：「インクリメンタル・バックアップ」 950 ページ。

プロキシ・テーブル

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。

参照：「メタデータ」 963 ページ。

ベース・テーブル

データを格納する永久テーブルです。テーブルは、テンポラリ・テーブルやビューと区別するために、「ベース・テーブル」と呼ばれることがあります。

参照：

- 「テンポラリ・テーブル」 958 ページ
- 「ビュー」 961 ページ

ポーリング

Mobile Link サーバ起動同期において、Mobile Link Listerner などのライト・ウェイト・ポーラが Notifier から Push 通知を要求する方法です。

参照：「サーバ起動同期」 953 ページ。

ポリシー

QAnywhere では、メッセージ転送の発生時期を指定する方法のことです。

マテリアライズド・ビュー

計算され、ディスクに保存されたビューのことです。マテリアライズド・ビューは、ビュー (クエリ指定を使用して定義される) とテーブル (ほとんどのテーブルの操作をそのテーブル上で実行できる) の両方の特性を持ちます。

参照：

- 「ベース・テーブル」 963 ページ
- 「ビュー」 961 ページ

ミラー・ログ

参照：「トランザクション・ログ・ミラー」 959 ページ。

メタデータ

データについて説明したデータです。メタデータは、他のデータの特質と内容について記述しています。

参照：「スキーマ」 955 ページ。

メッセージ・システム

SQL Remote のレプリケーションでは、統合データベースとリモート・データベースの間でのメッセージのやりとりに使用するプロトコルのことです。SQL Anywhere では、FILE、FTP、SMTP のメッセージ・システムがサポートされています。

参照：

- 「レプリケーション」 965 ページ
- 「FILE」 944 ページ

メッセージ・ストア

QAnywhere では、メッセージを格納するクライアントおよびサーバ・デバイスのデータベースのことです。

参照：

- 「クライアント・メッセージ・ストア」 952 ページ
- 「サーバ・メッセージ・ストア」 953 ページ

メッセージ・タイプ

SQL Remote のレプリケーションでは、リモート・ユーザと統合データベースのパブリッシャとの通信方法を指定するデータベース・オブジェクトのことを指します。統合データベースには、複数のメッセージ・タイプが定義されていることがあります。これによって、リモート・ユーザはさまざまなメッセージ・システムを使って統合データベースと通信できることになります。

参照：

- 「レプリケーション」 965 ページ
- 「統合データベース」 972 ページ

メッセージ・ログ

データベース・サーバや Mobile Link サーバなどのアプリケーションからのメッセージを格納できるログです。この情報は、メッセージ・ウィンドウに表示されたり、ファイルに記録されたりすることもあります。メッセージ・ログには、情報メッセージ、エラー、警告、MESSAGE 文からのメッセージが含まれます。

メンテナンス・リリース

メンテナンス・リリースは、同じメジャー・バージョン番号を持つ旧バージョンのインストール済みソフトウェアをアップグレードするための完全なソフトウェア・セットです (バージョン番号のフォーマットは、*メジャー.マイナー.パッチ.ビルド*です)。バグ・フィックスとその他の変更については、アップグレードのリリース・ノートにリストされます。

ユーザ定義データ型

参照：「ドメイン」 958 ページ。

ライト・ウェイト・ポーラ

Mobile Link サーバ起動同期において、Mobile Link サーバからの Push 通知をポーリングするデバイス・アプリケーションです。

参照：「サーバ起動同期」 953 ページ。

リダイレクタ

クライアントと Mobile Link サーバ間で要求と応答をルート指定する Web サーバ・プラグインです。このプラグインによって、負荷分散メカニズムとフェールオーバ・メカニズムも実装されます。

リファレンス・データベース

Mobile Link では、Ultra Light クライアントの開発に使用される SQL Anywhere データベースです。開発中は、1 つの SQL Anywhere データベースをリファレンス・データベースとしても統合データベースとしても使用できます。他の製品によって作成されたデータベースは、リファレンス・データベースとして使用できません。

リモート ID

SQL Anywhere と Ultra Light データベース内のユニークな識別子で、Mobile Link によって使用されます。リモート ID は NULL に初期設定されていますが、データベースの最初の同期時に GUID に設定されます。

リモート・データベース

Mobile Link または SQL Remote では、統合データベースとデータを交換するデータベースを指します。リモート・データベースは、統合データベース内のすべてまたは一部のデータを共有できます。

参照：

- 「同期」 972 ページ
- 「統合データベース」 972 ページ

リレーショナル・データベース管理システム (RDBMS)

関連するテーブルの形式でデータを格納するデータベース管理システムです。

参照：「データベース管理システム (DBMS)」 957 ページ。

レプリケーション

物理的に異なるデータベース間でデータを共有することです。Sybase では、Mobile Link、SQL Remote、Replication Server の 3 種類のレプリケーション・テクノロジを提供しています。

レプリケーション・メッセージ

SQL Remote または Replication Server では、パブリッシュするデータベースとサブスクリプションを作成するデータベース間で送信される通信内容を指します。メッセージにはデータを含み、レプリケーション・システムで必要なパススルー文、情報があります。

参照：

- 「レプリケーション」 965 ページ
- 「パブリケーションの更新」 960 ページ

レプリケーションの頻度

SQL Remote レプリケーションでは、リモート・ユーザに対する設定の 1 つで、パブリッシャの Message Agent がレプリケーション・メッセージを他のリモート・ユーザに送信する頻度を定義します。

参照：「レプリケーション」 965 ページ。

ロー・レベルのトリガ

変更されているローごとに一回実行するトリガです。

参照：

- 「トリガ」 959 ページ
- 「文レベルのトリガ」 973 ページ

ローカル・テンポラリ・テーブル

複合文を実行する間だけ存在したり、接続が終了するまで存在したりするテンポラリ・テーブルです。データのセットを 1 回だけロードする必要がある場合にローカル・テンポラリ・テーブルが便利です。デフォルトでは、COMMIT を実行するとローが削除されます。

参照：

- 「テンポラリ・テーブル」 958 ページ
- 「グローバル・テンポラリ・テーブル」 952 ページ

ロール

概念データベース・モデルで、ある視点からの関係を説明する動詞またはフレーズを指します。各関係は 2 つのロールを使用して表すことができます。"contains (A は B を含む)" や "is a member of (B は A のメンバ)" などのロールがあります。

ロールバック・ログ

コミットされていない各トランザクションの最中に行われた変更のレコードです。ROLLBACK 要求やシステム障害が発生した場合、コミットされていないトランザクションはデータベースから破棄され、データベースは前の状態に戻ります。各トランザクションにはそれぞれロールバック・ログが作成されます。このログは、トランザクションが完了すると削除されます。

参照：「トランザクション」 959 ページ。

ロール名

外部キーの名前です。この外部キーがロール名と呼ばれるのは、外部テーブルとプライマリ・テーブル間の関係に名前を指定するためです。デフォルトでは、テーブル名がロール名になります。ただし、別の外部キーがそのテーブル名を使用している場合、デフォルトのロール名はテーブル名に 3 桁のユニークな数字を付けたものになります。ロール名は独自に作成することもできます。

参照：「外部キー」 967 ページ。

ログ・ファイル

SQL Anywhere によって管理されているトランザクションのログです。ログ・ファイルを使用すると、システム障害やメディア障害が発生してもデータベースを回復させることができます。また、データベースのパフォーマンスを向上させたり、SQL Remote を使用してデータをレプリケートしたりする場合にも使用できます。

参照：

- 「トランザクション・ログ」 959 ページ
- 「トランザクション・ログ・ミラー」 959 ページ
- 「フル・バックアップ」 962 ページ

ロック

複数のトランザクションを同時に実行しているときにデータの整合性を保護する同時制御メカニズムです。SQL Anywhere では、2 つの接続によって同じデータが同時に変更されないようにするために、また変更処理の最中に他の接続によってデータが読み込まれないようにするために、自動的にロックが適用されます。

ロックの制御は、独立性レベルを設定して行います。

参照：

- 「独立性レベル」 973 ページ
- 「同時性 (同時実行性)」 973 ページ
- 「整合性」 970 ページ

ワーク・テーブル

クエリの最適化の最中に中間結果を保管する内部保管領域です。

一意性制約

NULL 以外のすべての値が重複しないことを要求するカラムまたはカラムのセットに対する制限です。テーブルには複数の一意性制約を指定できます。

参照：

- 「外部キー制約」 968 ページ
- 「プライマリ・キー制約」 962 ページ
- 「制約」 970 ページ

解析ツリー

クエリを代数で表現したものです。

外部キー

別のテーブルにあるプライマリ・キーの値を複製する、テーブルの 1 つ以上のカラムです。テーブル間の関係は、外部キーによって確立されます。

参照：

- 「プライマリ・キー」 962 ページ
- 「外部テーブル」 968 ページ

外部キー制約

カラムまたはカラムのセットに対する制約で、テーブルのデータが別のテーブルのデータとどのように関係しているかを指定するものです。カラムのセットに外部キー制約を加えると、それらのカラムが外部キーになります。

参照：

- 「制約」 970 ページ
- 「検査制約」 969 ページ
- 「プライマリ・キー制約」 962 ページ
- 「一意性制約」 967 ページ

外部ジョイン

テーブル内のすべてのローを保護するジョインです。SQL Anywhere では、左外部ジョイン、右外部ジョイン、全外部ジョインがサポートされています。左外部ジョインは JOIN 演算子の左側にあるテーブルのローを保護し、右側にあるテーブルのローがジョイン条件を満たさない場合には NULL を返します。全外部ジョインは両方のテーブルに含まれるすべてのローを保護します。

参照：

- 「ジョイン」 954 ページ
- 「内部ジョイン」 973 ページ

外部テーブル

外部キーを持つテーブルです。

参照：「外部キー」 967 ページ。

外部ログイン

リモート・サーバとの通信に使用される代替のログイン名とパスワードです。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。外部ログインは、リモート・サーバと通信するときに使用する代替のログイン名とパスワードです。

競合

リソースについて対立する動作のことです。たとえば、データベース用語では、複数のユーザがデータベースの同じローを編集しようとした場合、そのローの編集権についての競合が発生します。

競合解決

Mobile Link では、競合解決は 2 人のユーザが別々のリモート・データベースの同じローを変更した場合にどう処理するかを指定するロジックのことです。

検査制約

指定された条件をカラムやカラムのセットに課す制約です。

参照：

- 「制約」 970 ページ
- 「外部キー制約」 968 ページ
- 「プライマリ・キー制約」 962 ページ
- 「一意性制約」 967 ページ

検証

データベース、テーブル、またはインデックスについて、特定のタイプのファイル破損をテストすることです。

作成者 ID

Ultra Light の Palm OS アプリケーションでは、アプリケーションが作成されたときに割り当てられる ID のことです。

参照元オブジェクト

テーブルなどのデータベースの別のオブジェクトをオブジェクト定義が直接参照する、ビューなどのオブジェクトです。

参照：「外部キー」 967 ページ。

参照整合性

データの整合性、特に異なるテーブルのプライマリ・キー値と外部キー値との関係を管理する規則を厳守することです。参照整合性を備えるには、それぞれの外部キーの値が、参照テーブルにあるローのプライマリ・キー値に対応するようにします。

参照：

- 「プライマリ・キー」 962 ページ
- 「外部キー」 967 ページ

参照先オブジェクト

ビューなどの別のオブジェクトの定義で直接参照される、テーブルなどのオブジェクトです。

参照：「プライマリ・キー」 962 ページ。

識別子

テーブルやカラムなどのデータベース・オブジェクトを参照するときに使う文字列です。A ～ Z、a ～ z、0 ～ 9、アンダースコア (_)、アットマーク (@)、シャープ記号 (#)、ドル記号 ($) のうち、任意の文字を識別子として使用できます。

述部

条件式です。オプションで論理演算子 AND や OR と組み合わせて、WHERE 句または HAVING 句に条件のセットを作成します。SQL では、unknown と評価される述部が false と解釈されます。

照合

データベース内のテキストのプロパティを定義する文字セットとソート順の組み合わせのことです。SQL Anywhere データベースでは、サーバを実行しているオペレーティング・システムと言語によって、デフォルトの照合が決まります。たとえば、英語版 Windows システムのデフォルトの照合は 1252LATIN1 です。照合は、照合順とも呼ばれ、文字列の比較とソートに使用します。

参照：

- 「文字セット」 973 ページ
- 「コード・ページ」 953 ページ
- 「エンコード」 951 ページ

世代番号

Mobile Link では、リモート・データベースがデータをアップロードしてからダウンロード・ファイルを適用するようにするためのメカニズムのことです。

参照：「ファイルベースのダウンロード」 961 ページ。

制約

テーブルやカラムなど、特定のデータベース・オブジェクトに含まれた値に関する制約です。たとえば、一意性制約があるカラム内の値は、すべて異なっている必要があります。テーブルに、そのテーブルの情報と他のテーブルのデータがどのように関係しているのかを指定する外部キー制約が設定されていることもあります。

参照：

- 「検査制約」 969 ページ
- 「外部キー制約」 968 ページ
- 「プライマリ・キー制約」 962 ページ
- 「一意性制約」 967 ページ

整合性

データが適切かつ正確であり、データベースの関係構造が保たれていることを保証する規則を厳守することです。

参照：「参照整合性」 969 ページ。

正規化

データベース・スキーマを改善することです。リレーショナル・データベース理論に基づく規則に従って、冗長性を排除したり、編成を改良します。

正規表現

正規表現は、文字列内で検索するパターンを定義する、一連の文字、ワイルドカード、演算子です。

生成されたジョイン条件

自動的に生成される、ジョインの結果に対する制限です。キーとナチュラルの 2 種類があります。キー・ジョインは、KEY JOIN を指定したとき、またはキーワード JOIN を指定したが、CROSS、NATURAL、または ON を使用しなかった場合に生成されます。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・ジョインは NATURAL JOIN を指定したときに生成され、生成されたジョイン条件は、2 つのテーブルの共通のカラム名に基づきます。

参照：

- 「ジョイン」 954 ページ
- 「ジョイン条件」 955 ページ

接続 ID

クライアント・アプリケーションとデータベース間の特定の接続に付けられるユニークな識別番号です。現在の接続 ID を確認するには、次の SQL 文を使用します。

```
SELECT CONNECTION_PROPERTY( 'Number' );
```

接続プロファイル

ユーザ名、パスワード、サーバ名などの、データベースに接続するために必要なパラメータのセットです。便宜的に保管され使用されます。

接続起動同期

Mobile Link のサーバ起動同期の 1 つの形式で、接続が変更されたときに同期が開始されます。

参照：「サーバ起動同期」 953 ページ。

相関名

クエリの FROM 句内で使用されるテーブルやビューの名前です。テーブルやビューの元の名前か、FROM 句で定義した代替名のいずれかになります。

抽出

SQL Remote レプリケーションでは、統合データベースから適切な構造とデータをアンロードする動作を指します。この情報は、リモート・データベースを初期化するときに使用されます。

参照：「レプリケーション」 965 ページ。

通信ストリーム

Mobile Link では、Mobile Link クライアントと Mobile Link サーバ間での通信にネットワーク・プロトコルが使用されます。

転送ルール

QAnywhere では、メッセージの転送を発生させる時期、転送するメッセージ、メッセージを削除する時期を決定する論理のことです。

統合データベース

分散データベース環境で、データのマスタ・コピーを格納するデータベースです。競合や不一致が発生した場合、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。

参照：

- 「同期」 972 ページ
- 「レプリケーション」 965 ページ

統合化ログイン

オペレーティング・システムへのログイン、ネットワークへのログイン、データベースへの接続に、同一のユーザ ID とパスワードを使用するログイン機能の 1 つです。

動的 SQL

実行される前に作成したプログラムによって生成される SQL です。Ultra Light の動的 SQL は、占有容量の小さいデバイス用に設計された変形型です。

同期

Mobile Link テクノロジを使用してデータベース間でデータをレプリケートする処理です。

SQL Remote では、同期はデータの初期セットを使ってリモート・データベースを初期化する処理を表すために特に使用されます。

参照：

- 「Mobile Link」 945 ページ
- 「SQL Remote」 948 ページ

同時性 (同時実行性)

互いに独立し、場合によっては競合する可能性のある 2 つ以上の処理を同時に実行することです。SQL Anywhere では、自動的にロックを使用して各トランザクションを独立させ、同時に稼働するそれぞれのアプリケーションが一貫したデータのセットを参照できるようにします。

参照：

- 「トランザクション」 959 ページ
- 「独立性レベル」 973 ページ

独立性レベル

あるトランザクションの操作が、同時に処理されている別のトランザクションの操作からどの程度参照できるかを示します。独立性レベルには 0 から 3 までの 4 つのレベルがあります。最も高い独立性レベルには 3 が設定されます。デフォルトでは、レベルは 0 に設定されています。SQL Anywhere では、スナップショット、文のスナップショット、読み込み専用文のスナップショットの 3 つのスナップショットの独立性レベルがあります。

参照：「スナップショット・アイソレーション」 955 ページ。

内部ジョイン

2 つのテーブルがジョイン条件を満たす場合だけ、結果セットにローが表示されるジョインです。内部ジョインがデフォルトです。

参照：

- 「ジョイン」 954 ページ
- 「外部ジョイン」 968 ページ

物理インデックス

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。

文レベルのトリガ

トリガ付きの文の処理が完了した後に実行されるトリガです。

参照：

- 「トリガ」 959 ページ
- 「ロー・レベルのトリガ」 966 ページ

文字セット

文字セットは記号、文字、数字、スペースなどから成ります。"ISO-8859-1" は文字セットの例です。Latin1 とも呼ばれます。

参照：

- 「コード・ページ」 953 ページ
- 「エンコード」 951 ページ
- 「照合」 970 ページ

文字列リテラル

文字列リテラルとは、一重引用符 (') で囲まれ、シーケンスで並べられた文字のことです。

論理インデックス

物理インデックスへの参照 (ポインタ) です。ディスクに保存される論理インデックス用のインデックス構造はありません。

索引

記号

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

あ

い

う

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

つ

て

と

な

に

ね

は

ひ

ふ

へ

ほ

ま

み

む

め

も

ゆ

よ

ら

り

る

れ

ろ

わ