iAnywhere SOLUTIONS™
A SYBASE COMPANY

Adaptive Server® Anywhere SQL ユーザーズ・ガイド

パート番号：DC03939-01-0902-01
改訂：2005 年 3 月

版権

Copyright © 2005 iAnywhere Solutions, Inc., Sybase, Inc. All rights reserved.

ここに記載されている内容を iAnywhere Solutions, Inc.、Sybase, Inc. またはその関連会社の書面による事前許可を得ずに電子的、機械的、手作業、光学的、またはその他のいかなる手段によっても複製、転載、翻訳することを禁じます。

Sybase、SYBASE のロゴ、Adaptive Server、AnswerBase、Anywhere、EIP、Embedded SQL、Enterprise Connect、Enterprise Portal、GainMomentum、iAnywhere、jConnect MASS DEPLOYMENT、Netimpact、ObjectConnect、ObjectCycle、OmniConnect、Open ClientConnect、Open ServerConnect、PowerBuilder、PowerDynamo、Powersoft、Quickstart Datamart、Replication Agent、Replication Driver、SQL Anywhere、SQL Central、SQL Remote、Support Plus、SWAT、Sybase IQ、Sybase System 11、Sybase WAREHOUSE、SyBooks、XA-Library は米国法人 Sybase, Inc. の登録商標です。Backup Server、Client-Library、jConnect for JDBC、MainframeConnect、Net-Gateway、Net-Library、Open Client、Open Client/Server、S-Designor、SQL Advantage、SQL Debug、SQL Server、SQL Server Manager、Sybase Central、Watcom、Web.SQL、XP Server は米国法人 Sybase, Inc. の商標です。

ここに記載されている上記以外の社名および製品名は、各社の商標または登録商標の場合があります。

目次

はじめに ix
SQL Anywhere Studio のマニュアル x
表記の規則 xiv
Adaptive Server Anywhere サンプル・データベース xvii
詳細情報の検索／フィードバックの提供 xviii

1 データベースの設計 3
概要 4
データベース設計の概念 5
設計のプロセス 13
テーブル・プロパティの設計 31

2 データベース・オブジェクトの使用 35
概要 36
データベースの編集 37
テーブルの編集 51
ビューの編集 69
インデックスの編集 79
テンポラリ・テーブルの編集 97
Sybase Central におけるデータベース・オブジェクトのコピー 98

3 データ整合性の確保 101
データ整合性の概要 102
カラム・デフォルトの使い方 107
テーブル制約とカラム制約の使い方 114
ドメインの使い方 119
エンティティ整合性と参照整合性の確保 124
システム・テーブルの整合性ルール 130

4 トランザクションと独立性レベル 133
トランザクションの概要 134
独立性レベルと一貫性 139
トランザクションのブロックとデッドロック 148
独立性レベルの選択 152
独立性レベルのチュートリアル 157
ロックの仕組み 175
特殊な同時性の問題 192
レプリケーションと同時性 195
まとめ 198

5 パフォーマンスのモニタリングと改善 201
パフォーマンス分析ツール 202
最高のパフォーマンスを得るヒント 211
パフォーマンス向上へのキャッシュの使用 231
インデックスの使用によるパフォーマンス改善 239
キーを使ったクエリのパフォーマンス改善 240
クエリ結果のソート 242
クエリ処理中のワーク・テーブルの使用 243
データベースのパフォーマンスのモニタリング 246
断片化 254
データベース・プロシージャのプロファイリング 259

6 クエリ：テーブルからのデータの選択 275
クエリの概要 276
SELECT リスト：カラムの指定 279
FROM 句：テーブルの指定 288
WHERE 句：ローの指定 290

7 クエリ結果の要約、グループ化、ソート 305
集合関数を使用したクエリ結果の要約 306
GROUP BY 句：クエリ結果のグループへの編成 312
GROUP BY 句の概要 314
HAVING 句：データ・グループの選択 318
ORDER BY 句：クエリ結果のソート 320
UNION、INTERSECT、EXCEPT を使用した、クエリ結果に対する集合操作の実行324
標準と互換性 332

8 ジョイン : 複数テーブルからのデータ検索 335
サンプル・データベース・スキーマ 336
ジョイン操作 337
ジョインの概要 338
明示的なジョイン条件 (ON 句) 345
クロス・ジョイン 349
内部ジョインと外部ジョイン 351
特殊なジョイン 360
ナチュラル・ジョイン 369
キー・ジョイン 374

9 共通テーブル式 391
共通テーブル式について 392
共通テーブル式の一般的な使用例 397
再帰共通テーブル式 401
構成部品の問題 405
再帰共通テーブル式でのデータ型宣言 409
最短距離の問題 411
複数の再帰共通テーブル式の使用 415

10 OLAP の使用 419
OLAP の概要 420
小計の概要 421
NULL 値と小計ロー 424
ROLLUP の使用 426
CUBE の使用 432
GROUPING SETS の使用 435
OLAP 関数の活用 438

11 サブクエリの使用 447
サブクエリの概要 448
WHERE 句でのサブクエリの使用 449
HAVING 句でのサブクエリ 450
サブクエリ比較テスト 453
ANY と ALL を使用した限定比較テスト 455
IN 条件によるセット・メンバシップのテスト 459
存在テスト 461
外部参照 463

サブクエリとジョイン 464
ネストされたサブクエリ 467
サブクエリ操作 469

12 データの追加、変更、削除 481
データ修正文 482
INSERT によるデータの追加 484
UPDATE によるデータの変更 489
INSERT によるデータの変更 491
DELETE によるデータの削除 492

13 クエリの最適化と実行 495
オプティマイザの役割 496
オプティマイザの仕組み 498
クエリ実行アルゴリズム 510
物理データの編成とアクセス 531
インデックス 535
セマンティック・クエリ変形 546
サブクエリと関数のキャッシュ 562
アクセス・プランの解釈 564

14 Transact-SQL との互換性 589
Transact-SQL のサポートの概要 590
Adaptive Server のアーキテクチャ 594
Transact-SQL との互換性を意識したデータベースの設定 602
互換性のある SQL 文の記述方法 613
Transact-SQL のプロシージャ言語の概要 619
ストアド・プロシージャの自動変換 622
Transact-SQL プロシージャから返される結果セット 624
Transact-SQL プロシージャの中の変数 625
Transact-SQL プロシージャでのエラー処理 626

15 他の SQL ダイアレクトとの違い 629
Adaptive Server Anywhere SQL の機能 630

16 データベースにおける XML の使用 635
XML とは 636

リレーショナル・データベースにおける XML 文書の格納 637
リレーショナル・データを XML としてエクスポートする 639
XML 文書をリレーショナル・データとしてインポートする 641
クエリ結果を XML として取得する 649
SQL/XML を使用してクエリ結果を XML として取得する 675

17 データのインポートとエクスポート 687
データベースとの間でのデータの転送 688
バルク・オペレーションのパフォーマンスの側面 689
バルク・オペレーションのデータ・リカバリの問題 690
データベースへのデータのインポート 691
データベースからのデータのエクスポート 704
データベースの再構築 719
データベースの抽出 732
Adaptive Server Anywhere へのデータベースの移行 733
SQL コマンド・ファイルの使用 741
Adaptive Server Enterprise の互換性 744

18 リモート・データへのアクセス 745
概要 746
リモート・データ・アクセスの基本概念 748
リモート・サーバの使用 751
外部ログインの使用 759
プロキシ・テーブルの編集 762
リモート・テーブルのジョイン 768
複数のローカル・データベースのテーブルのジョイン 770
ネイティブ文のリモート・サーバへの送信 772
リモート・プロシージャ・コール (RPC) の使用 773
トランザクションの管理とリモート・データ 776
内部オペレーション 778
リモート・データ・アクセスのトラブルシューティング 783

19 リモート・データ・アクセスのサーバ・クラス 787
概要 788
JDBC ベースのサーバ・クラス 789
ODBC ベースのサーバ・クラス 793

20 プロシージャ、トリガ、バッチの使用 811
プロシージャとトリガの概要 812
プロシージャとトリガの利点 813
プロシージャの概要 815
ユーザ定義関数の概要 825
トリガの概要 830
バッチの概要 839
制御文 841
プロシージャとトリガの構造 845
プロシージャから返される結果 849
プロシージャとトリガでのカーソルの使用 856
プロシージャとトリガでのエラーと警告 860
プロシージャでの EXECUTE IMMEDIATE 文の使用 871
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 873
プロシージャを作成するときのヒント 874
バッチに使用できる文 877
プロシージャからの外部ライブラリの呼び出し 879

21 データベースでのデバッグ論理 889
データベースにおけるデバッグの概要 890
チュートリアル：デバッガの使用開始 893
ブレークポイントの活用 906
変数の編集 910
接続の活用 912

索引 913

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、Sybase Adaptive Server 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 Adaptive Server 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 Studio の紹介*』を読んでから、このマニュアルを読んでください。

SQL Anywhere Studio のマニュアル

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

SQL Anywhere Studio のマニュアル

SQL Anywhere Studio のマニュアルは、各マニュアルを 1 つの大きなヘルプ・ファイルにまとめたオンライン形式、マニュアル別の PDF ファイル、および有料の製本版マニュアルで提供されます。SQL Anywhere Studio のマニュアルは、次の分冊マニュアルで構成されています。

- **『SQL Anywhere Studio の紹介』** このマニュアルでは、SQL Anywhere Studio のデータベース管理と同期テクノロジの概要について説明します。また、SQL Anywhere Studio を構成する各部分について説明するチュートリアルも含まれています。
- **『SQL Anywhere Studio 新機能ガイド』** このマニュアルは、SQL Anywhere Studio のこれまでのリリースのユーザを対象としています。ここでは、製品の今回のリリースと以前のリリースで導入された新機能をリストし、アップグレード手順を説明しています。
- **『Adaptive Server Anywhere データベース管理ガイド』** このマニュアルでは、データベースおよびデータベース・サーバの実行、管理、設定について説明しています。
- **『Adaptive Server Anywhere SQL ユーザーズ・ガイド』** 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。
- **『Adaptive Server Anywhere SQL リファレンス・マニュアル』** このマニュアルは、Adaptive Server Anywhere で使用する SQL 言語の完全なリファレンスです。また、Adaptive Server Anywhere のシステム・テーブルとシステム・プロシージャについても説明しています。
- **『Adaptive Server Anywhere プログラミング・ガイド』** このマニュアルでは、C、C++、Java 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。Visual Basic や PowerBuilder などのツールのユーザは、それらのツールのプログラミング・インタフェースを使用できます。また、Adaptive Server Anywhere ADO.NET データ・プロバイダについても説明します。

- **『Adaptive Server Anywhere SNMP Extension Agent ユーザーズ・ガイド』** このマニュアルでは、Adaptive Server Anywhere SNMP Extension Agent を SNMP 管理アプリケーションとともに使用できるように設定して、Adaptive Server Anywhere データベースを管理できるようにする方法を説明します。

- **『Adaptive Server Anywhere エラー・メッセージ』** このマニュアルでは、Adaptive Server Anywhere エラー・メッセージの完全なリストを、その診断情報とともに説明します。

- **『SQL Anywhere Studio セキュリティ・ガイド』** このマニュアルでは、Adaptive Server Anywhere データベースのセキュリティ機能について説明します。Adaptive Server Anywhere 7.0 は、米国政府から TCSEC (Trusted Computer System Evaluation Criteria) の C2 セキュリティ評価を授与されています。このマニュアルには、Adaptive Server Anywhere の現在のバージョンを、C2 基準を満たした環境と同等の方法で実行することを望んでいるユーザにとって役に立つ情報が含まれています。

- **『Mobile Link 管理ガイド』** このマニュアルでは、モバイル・コンピューティング用の Mobile Link データ同期システムについてあらゆる角度から説明します。このシステムによって、Oracle、Sybase、Microsoft、IBM の単一データベースと、Adaptive Server Anywhere や Ultra Light の複数データベースの間でのデータ共有が可能になります。

- **『Mobile Link クライアント』** このマニュアルでは、Adaptive Server Anywhere リモート・データベースと Ultra Light リモート・データベースの設定を行い、これらを同期させる方法について説明します。

- **『Mobile Link サーバ起動同期ユーザーズ・ガイド』** このマニュアルでは、Mobile Link のサーバによって開始される同期について説明します。サーバによって開始される同期とは、統合データベースから同期の開始を可能にする Mobile Link の機能です。

- **『Mobile Link チュートリアル』** このマニュアルには、Mobile Link アプリケーションの設定と実行を行う方法を説明するチュートリアルがいくつか用意されています。

- **『QAnywhere ユーザーズ・ガイド』** このマニュアルでは、Mobile Link QAnywhere について説明します。Mobile Link QAnywhere は、従来のデスクトップ・クライアントやラップトップ・クライアントだけでなく、モバイル・クライアントや無線クライアント用のメッセージング・アプリケーションの開発と展開を可能にするメッセージング・プラットフォームです。

- **『Mobile Link およびリモート・データ・アクセスの ODBC ドライバ』** このマニュアルでは、Mobile Link 同期サーバから、または Adaptive Server Anywhere リモート・データ・アクセスによって、Adaptive Server Anywhere 以外の統合データベースにアクセスするための ODBC ドライバの設定方法について説明します。

- **『SQL Remote ユーザーズ・ガイド』** 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、あらゆる角度から説明します。このシステムによって、Adaptive Server Anywhere または Adaptive Server Enterprise の単一データベースと Adaptive Server Anywhere の複数データベースの間で、電子メールやファイル転送などの間接的リンクを使用したデータ共有が可能になります。

- **『SQL Anywhere Studio ヘルプ』** このマニュアルには、Sybase Central や Interactive SQL、その他のグラフィカル・ツールに関するコンテキスト別のヘルプが含まれています。これは、製本版マニュアル・セットには含まれていません。

- **『Ultra Light データベース・ユーザーズ・ガイド』** このマニュアルは、Ultra Light 開発者を対象としています。ここでは、Ultra Light データベース・システムの概要について説明します。また、すべての Ultra Light プログラミング・インタフェースに共通する情報を提供します。

- **Ultra Light のインタフェースに関するマニュアル** 各 Ultra Light プログラミング・インタフェースには、それぞれに対応するマニュアルを用意しています。これらのインタフェースは、RAD (

ラピッド・アプリケーション開発) 用の Ultra Light コンポーネントとして提供されているものと、C、C++、Java 開発用の静的インタフェースとして提供されているものがあります。

このマニュアル・セットの他に、PowerDesigner と InfoMaker には、独自のオンライン・マニュアル (英語版) がそれぞれ用意されています。

マニュアルの形式

SQL Anywhere Studio のマニュアルは、次の形式で提供されています。

- **オンライン・マニュアル** オンライン・マニュアルには、SQL Anywhere Studio の完全なマニュアルがあり、SQL Anywhere ツールに関する印刷マニュアルとコンテキスト別のヘルプの両方が含まれています。オンライン・マニュアルは、製品のメンテナンス・リリースごとに更新されます。これは、最新の情報を含む最も完全なマニュアルです。

  Windows オペレーティング・システムでオンライン・マニュアルにアクセスするには、[スタート] － [プログラム] － [SQL Anywhere 9] － [オンライン・マニュアル] を選択します。オンライン・マニュアルをナビゲートするには、左ウィンドウ枠で HTML ヘルプの目次、索引、検索機能を使用し、右ウィンドウ枠でリンク情報とメニューを使用します。

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

- **PDF 版マニュアル** SQL Anywhere の各マニュアルは、Adobe Acrobat Reader で表示できる PDF ファイルで提供されています。

  PDF 版マニュアルは、オンライン・マニュアルまたは Windows の [スタート] メニューから利用できます。

- **製本版マニュアル** 製本版マニュアルをご希望の方は、ご購入いただいた販売代理店または弊社営業担当までご連絡ください。

表記の規則

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

SQL 構文の表記規則

SQL 構文の表記には、次の規則が適用されます。

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

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

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

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

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

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

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

- **オプション部分**　文のオプション部分は角カッコで囲みます。

  RELEASE SAVEPOINT [*savepoint-name*]

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

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

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

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

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

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

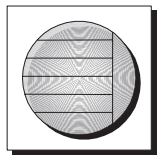
  QUOTES オプションを使用する場合は、ON または OFF のどちらかを選択する必要があります。角カッコと中カッコは入力しないでください。

グラフィック・アイコン

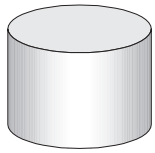
このマニュアルでは、次のアイコンを使用します。

- クライアント・アプリケーション

- Sybase Adaptive Server Anywhere などのデータベース・サーバ



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



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

- プログラミング・インタフェース

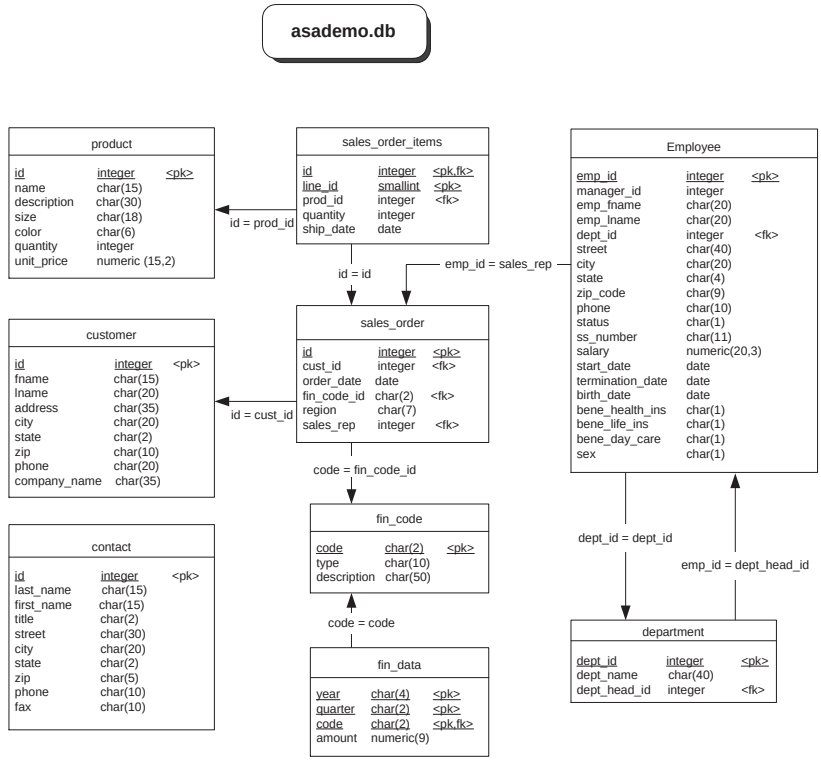
Adaptive Server Anywhere サンプル・データベース

このマニュアルでは、多くの例で Adaptive Server Anywhere サンプル・データベースが使用されています。

サンプル・データベースは、*asademo.db* という名前のファイルに保存され、SQL Anywhere ディレクトリに置かれています。

サンプル・データベースは小規模の企業の例です。データベースには、この企業の内部情報 (従業員、部署、経理) とともに、製品情報や販売情報 (受注、顧客、連絡先) が入っています。データベースに含まれる情報はすべて架空のものです。

次の図は、サンプル・データベース内のテーブルと各テーブル間の関係を示しています。



詳細情報の検索／フィードバックの提供

詳細情報の検索

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

ご質問がある場合や支援が必要な場合は、次に示す iAnywhere Solutions 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 Studio バージョンのビルド番号を明記し、現在発生している問題について詳しくお知らせくださいますようお願いいたします。バージョン情報は、コマンド・プロンプトで **dbeng9 -v** と入力して確認できます。

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

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

ニュースグループに関するお断り

iAnywhere Solutions は、ニュースグループ上に解決策、情報、または意見を提供する義務を負うものではありません。また、システム・オペレータ以外のスタッフにこのサービスを監視させて、操作状況や可用性を保証する義務もありません。

iAnywhere Solutions のテクニカル・アドバイザとその他のスタッフは、時間のある場合にかぎりニュースグループでの支援を行います。こうした支援は基本的にボランティアで行われるため、解決策や情報を定期的に提供できるとはかぎりません。支援できるかどうかは、スタッフの仕事量に左右されます。

フィードバック

このマニュアルに関するご意見、ご提案、フィードバックをお寄せください。

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

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

第 1 部　データベースの設計と作成

第 1 部では、データベースの設計と構築に関わる主要な概念と方式について説明します。データベースの設計、およびテーブル、ビュー、インデックスを扱う技術を取り上げます。また、参照整合性やトランザクションについても説明します。

第 1 章

データベースの設計

この章の内容

この章では、リレーショナル・データベース設計の基本的概念について説明し、独自のデータベースを設計する方法について段階的に説明していきます。これには概念データ・モデルとして知られる技術を使い、エンティティとその関係に焦点を当てています。

概要

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

詳細については、『*A Database Primer*』(C. J. Date 著) などの入門書も参考にしてください。データベース理論に興味がある場合は、『*An Introduction to Database Systems*』(C. J. Date 著) が役立ちます。

データベース設計の概念

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

エンティティと関係

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

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

ER (実体関連) 図

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

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

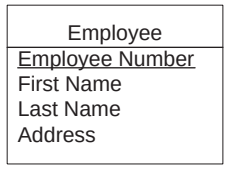
エンティティ

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

属性

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

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



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

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

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

関係

エンティティ間の関係は、データベースにおける動詞の役割を果たします。従業員は部署のメンバであり、事業所は都市にあります。データベース内の関係はテーブル間の外部キー関係として表示されることも、それ自体が独立したテーブルとして表示されることもあります。この章ではそれぞれの例を紹介します。

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

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

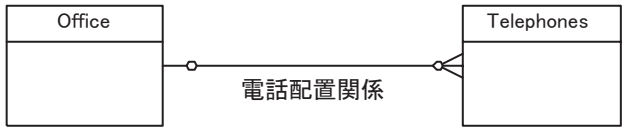
ヒント
厳密なデータベース構造は、部署に 2 人マネージャがいるなどの矛盾を排除できるので便利です。一方で、設計者としては、予測していない使用があった場合のために、多少の拡張ができる程度に柔軟に設計してください。設計が良くできているデータベースを拡張するのは、通常それほど難しくありませんが、既存のテーブル構造を変更すると、データベース全体およびそのクライアント・アプリケーションが使えなくなるおそれがあります。

関係のカーディナリティ

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

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

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

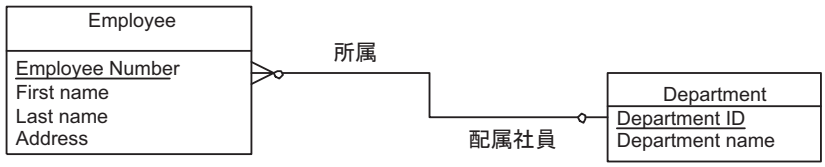


- **多対多の関係**　この関係では、両方のエンティティへの接続を示す複数の線を引きます。つまり、1 つの倉庫に各種部品を保管することができ、複数の倉庫に 1 種類の部品を保管することができます。

  役割

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

1. 従業員は部署の*メンバ*です。
2. 部署は従業員を*含みます*。



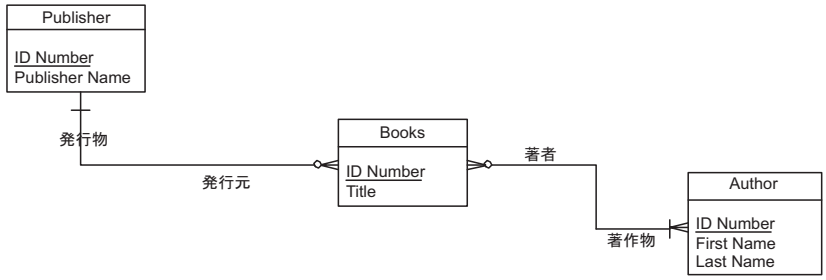
役割は非常に重要です。これは作業を検証する便利で効率的な手段を提供するためです。

ヒント
左から右に読む場合も、右から左に読む場合も、次のルールに従うことでこれらの図を間単に読むことができます。1. 最初のエンティティの名前を読みます。2. *最初のエンティティ*の横にある役割を読みます。3. *2 つめのエンティティ*への接続のカーディナリティを読みます。4. 2 つめのエンティティの名前を読みます。

必須要素

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

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



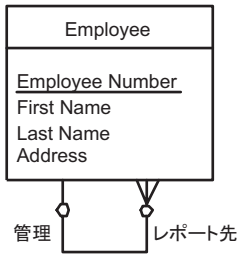
この図は次の 4 つの文に対応しています。

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

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

再帰関係

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



この図は次の 2 つの文に対応します。

1. 1 人の従業員は最大 1 人の他の従業員にレポートします。

2. 1 人の従業員は 0 人以上の従業員を管理します。

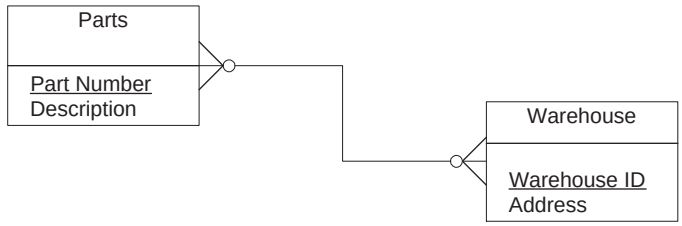
この関係の場合、両方向において関係が必然的に任意になることに注意してください。マネージャでない従業員もいます。同様に、少なくとも 1 人の従業員は組織の長で、だれの監督下にもありません。

当然、従業員は自分自身のマネージャにはなれません。この制限は一種の *ビジネス・ルール*です。ビジネス・ルールについては、後述の「設計のプロセス」13 ページを参照してください。

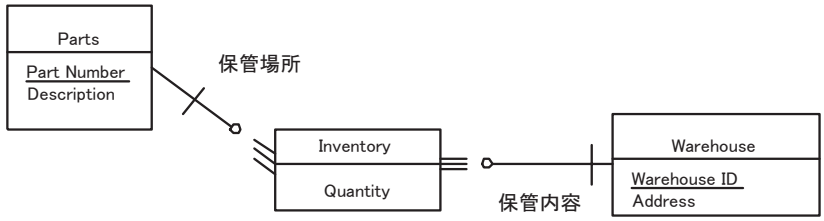
多対多の関係をエンティティに変更

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

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



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



変換に関して次の点に注意してください。

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

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

3. 新しいエンティティは Parts エンティティと Warehouse エンティティの両方に依存します。これは、新しいエンティティが両方のエンティティの識別子によって識別されるということです。この新しい図では、Parts エンティティからのある識別子と Warehouse エンティティからのある識別子が Inventory エンティティのエント

リをユニークに識別します。2 つの新しい関係を新しい Inventory エンティティにジョインする、円と複数の線の間に表示される三角は依存性を表します。

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

設計のプロセス

設計には主に次に示す 5 つの手順があります。

- 「手順 1：エンティティと関係を決定する」13 ページ
- 「手順 2：必要なデータを決定する」16 ページ
- 「手順 3：データを正規化する」19 ページ
- 「手順 4：関係を解析する」24 ページ
- 「手順 5：設計を検証する」28 ページ

データベース設計の実装の詳細については、「データベース・オブジェクトの使用」35 ページを参照してください。

手順 1：エンティティと関係を決定する

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

1. **上位レベルのアクティビティの定義**　このデータベースを使用する一般的なアクティビティを決定します。たとえば、従業員に関する情報を追跡して、変更があればそれをすぐ利用できるように記録しておくことです。

2. **エンティティを決定する**　作成したアクティビティ・リストから、情報の管理に必要なサブジェクト領域を決定します。これらのサブジェクトがエンティティになります。たとえば*従業員*の採用、*部署*への配属、*スキル・レベル*の決定などです。

3. **関係を決定する**　アクティビティを検討し、エンティティ間の関係がどのようなものか決定します。たとえば、部品と倉庫の間には関係があります。各関係を表す 2 つの役割を定義します。

4 **アクティビティを分解する**　ここでは上位アクティビティを定義しました。ここでは上位レベルのアクティビティが、下位レベルのアクティビティに分解できるかどうかを検討します。たとえば、*従業員データを管理する*という上位レベルのアクティビティは、次に示す下位レベルのアクティビティに分解できます。

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

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

エンティティと関係の例

例

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

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

上位レベルのアクティビティの定義

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

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

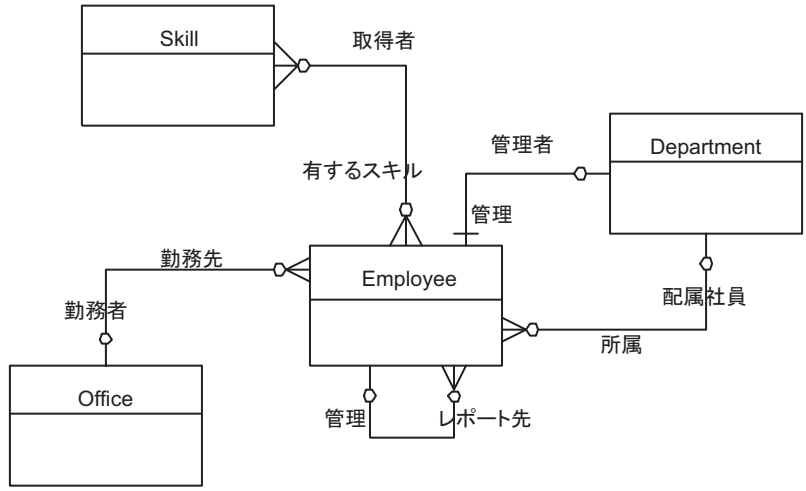
- 部署のデータの管理
- 事業所のデータの管理

エンティティと関係の決定

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

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

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



上位レベルのアクティビティの分解

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

- 従業員の追加または削除
- 事業所の追加または削除
- 部署の従業員のリスト
- 新しいスキルのスキル・リストへの追加

- 従業員のスキルの決定
- 各スキルに対する従業員のスキル・レベルの決定
- 特定のスキルに関して同一スキル・レベルにある従業員すべての決定
- 従業員のスキル・レベルの変更

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

ビジネス・ルールの決定

ビジネス・ルールは、1 対多、1 対 1、多対多の関係を決定します。

データベースに関係する可能性があるビジネス・ルールを次に示します。

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

手順 2：必要なデータを決定する

❖ **必要なデータを決定するには、次の手順に従います。**

1. 基本となるデータを決定します。
2. 追跡する必要があるデータをすべてリストします。

3　各エンティティのデータを準備します。

4　各エンティティで使用可能なデータをリストします。エンティティ (サブジェクト) を記述するデータは、誰が、何を、どこで、いつ、なぜ、の基準を満たすようにします。

5　各関係 (動詞) に必要なデータをリストします。

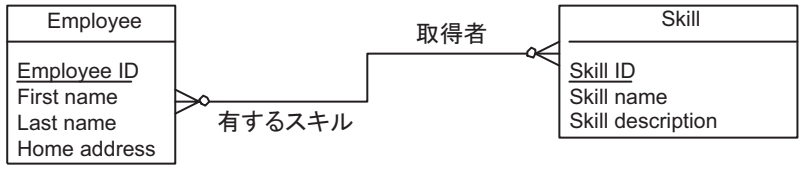
6　各関係のデータがある場合はリストします。

基礎となるデータの決定

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

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

この図は、次のようになります。



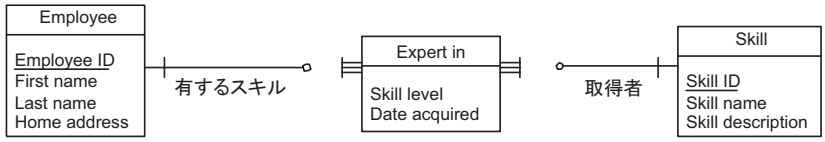
この図には、リストした属性すべてが表示されているわけではない点に注意してください。足りない項目は 2 種類のカテゴリに分かれます。

1. 他の関係に暗黙的に含まれている場合。たとえば、Employee department と Employee office は Department と Office エンティティの関係によってそれぞれ示されます。

2. どのエンティティとも関連しておらず、エンティティ間の関係と関連しているため存在しない場合。この図は不適当です。

項目の 1 つ目のカテゴリは ER 図全体を作成すれば自然に収まります。

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



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

注意

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

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

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

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

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

手順 3：データを正規化する

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

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

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

正規形

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

❖ データベースのデータを正規化するには、次の手順に従います。

1. データをリストします。
   - 各エンティティが少なくとも 1 つのキーを持つことを確認します。各エンティティは識別子を持つ必要があります。
   - 関係のキーを確認します。関係のキーとはジョインしている 2 つのエンティティからのキーです。
   - 基本となるデータのリストの中に計算されたデータがないか確認します。通常、計算されたデータはリレーショナル・データベースには格納されません。

2 データを第 1 正規形にします。

- 属性が同じエントリに対して複数の値を持つ可能性がある場合は、繰り返されるデータを削除します。
- 削除したデータで 1 つまたは複数のエンティティか関係を作成します。

3 データを第 2 正規形にします。

- 複数のキーを持つエンティティと関係を確認します。
- 複数のキーのうち、一部にのみ依存するデータを削除します。
- 削除したデータで 1 つまたは複数のエンティティと関係を作成します。

4 データを第 3 正規形にします。

- エンティティと関係のキー以外のデータに依存するデータを削除します。
- 削除したデータで 1 つまたは複数のエンティティと関係を作成します。

データと識別子

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

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

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

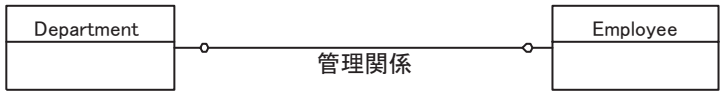
エンティティまたは*関係*	属性
Office	*Office code Office address Phone number
Works in	*Office code *Employee ID
Department	*Department ID Department name
Heads	*Department ID *Employee ID
Member of	*Department ID *Employee ID
Skill	*Skill ID Skill name Skill description
Expert in	*Skill ID *Employee ID Skill level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

エンティティまたは*関係*	属性
Employee	*Employee ID last name first name Social security number Address phone number date of birth

データを第 1 正規形にする

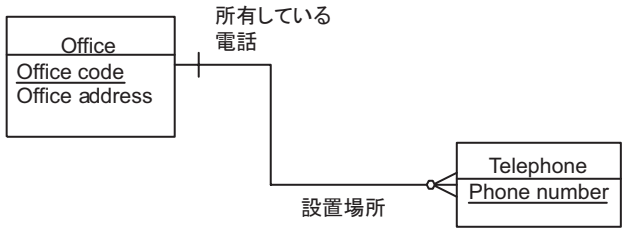
- 第 1 正規形をテストするには、繰り返される値を持つ属性を探します。
- 複数の値が単一の項目に適用できる場合は属性を削除します。これらの繰り返される属性を新しいエンティティに入れます。

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



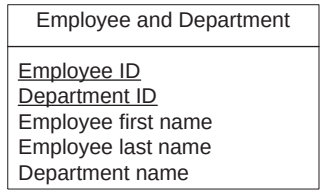
繰り返される属性を削除し、Telephone というエンティティを新しく作成します。Telephone と Office の間に関係を作成します。

データを第 2 正規形にする



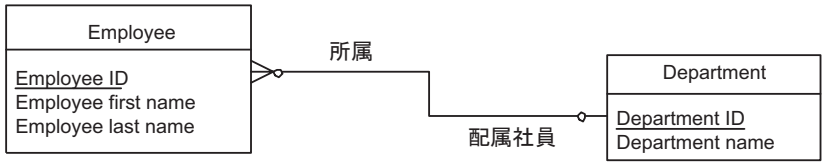
- キー全体に依存しないデータを削除します。
- 複数の属性からなる識別子を持つエンティティと関係にのみ注目します。第 2 正規形をテストするには、識別子全体に依存しないデータを削除します。各属性は、識別子を構成するすべての属性に依存するようにします。

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



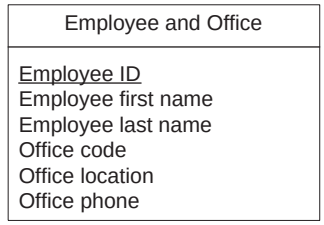
他の従業員データが依存しない識別子 Department ID を Department というエンティティに移動します。また、その識別子に依存するすべての属性も移動します。Employee と Department の間に関係を作成します。

データを第 3 正規形にする

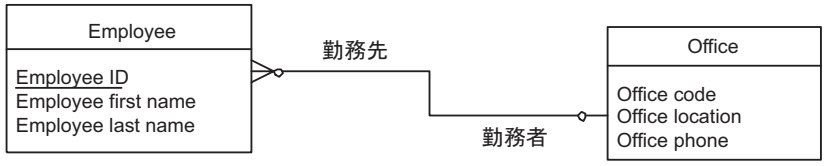


- キーに直接依存しないデータを削除します。
- 第 3 正規形をテストするには、識別子に直接ではなく他の属性に依存する属性を削除します。

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



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



手順 4：関係を解析する

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

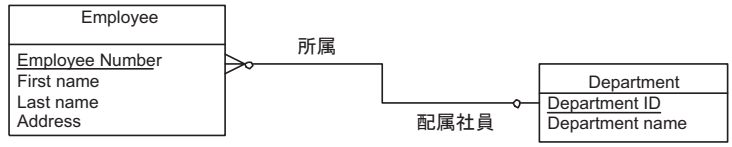
概念データ・モデルは実装作業とはほとんど関係ありませんが、物理データ・モデルはテーブル構造や特定のデータベース・アプリケーションで利用できるオプションと密接に関係しています。ここでのアプリケーションとは Adaptive Server Anywhere を指します。

データを含まない関係の解析

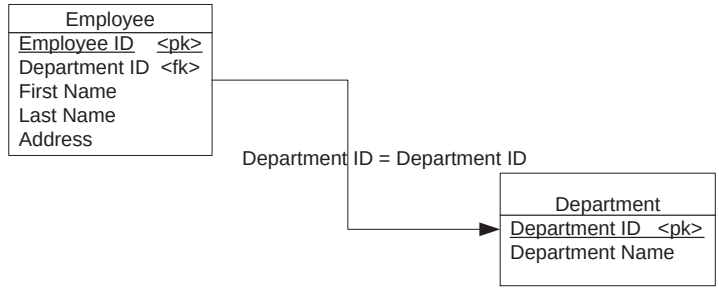
データを含まない関係を実装するには、外部キーを設定します。「**外部キー**」とは、他のテーブルからのプライマリ・キー値を含むカラムまたはカラム・セットです。外部キーを使用すると、複数のテーブルのデータに同時にアクセスできます。

Sybase PowerDesigner の DataArchitect コンポーネントなどのデータベース設計ツールは、物理データ・モデルを生成してくれます。ただし、自分で行う場合は、何をキーとするかを決定するのに役立つ基本ルールがあります。

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

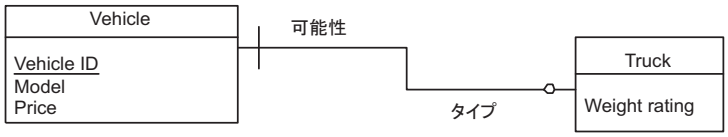


エンティティはテーブルになる点に注意してください。エンティティの識別子はテーブルのプライマリ・キー (または少なくともその一部) になります。属性はカラムになります。1 対多の関係では、*1* つのエンティティの識別子が*複数*のテーブルの新しい外部キー・カラムとして表示されます。

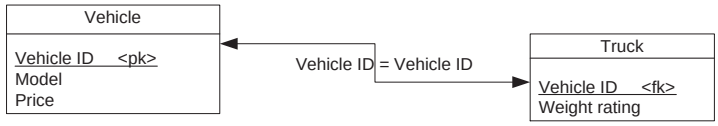


この例では、Employee *エンティティ*が Employee *テーブル*になります。同様に、Department エンティティが Department テーブルになります。Department ID と呼ばれる外部キーが Employee テーブルに表示されます。

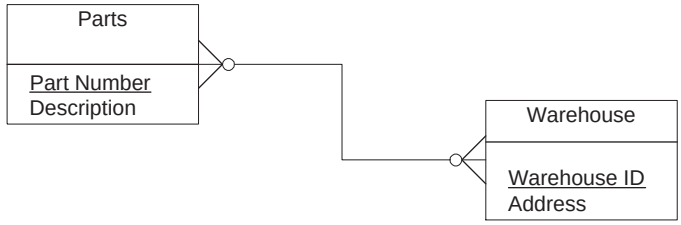
- **1 対 1**　1 対 1 の関係では、外部キーはどちらのテーブルにも方向付けできます。関係が一方では必須で、他方で任意であれば、任意の側に方向付けされます。この例では、乗り物が必ずしもトラックである必要はないので、Truck テーブルに外部キー (Vehicle ID) を置きます。



この ER モデルは次のデータベース構造に解析されます。

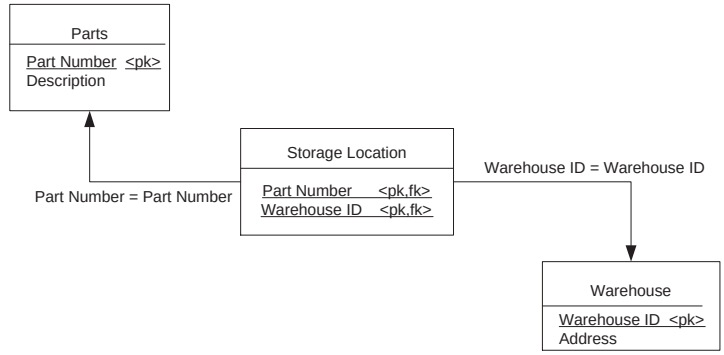


- **多対多**　多対多の関係では、新しいテーブルと 2 つの外部キーが作成されます。この方法は効率的なデータベースを作成するためには欠かせません。

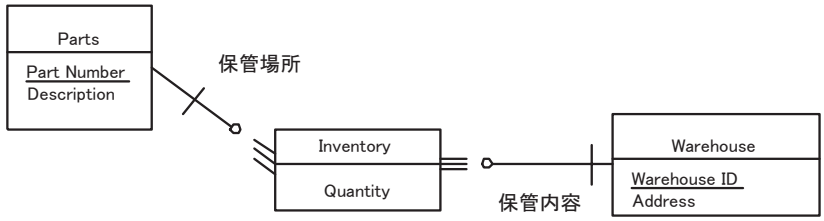


新しい Storage Location テーブルは Parts テーブルと Warehouse テーブルを関連付けます。

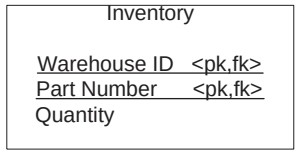
データを含む関係の解析



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



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



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

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

手順 5 : 設計を検証する

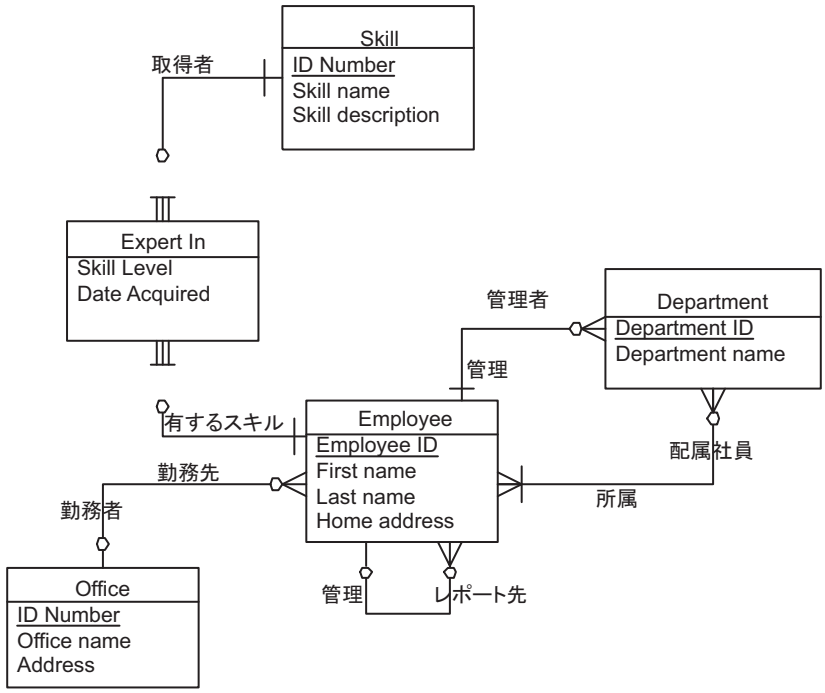
要求がサポートされていることを確認してから、作成した設計を実装します。設計の最初に確認したアクティビティを検査して、それに必要なすべてのデータにアクセスできることを確認します。

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

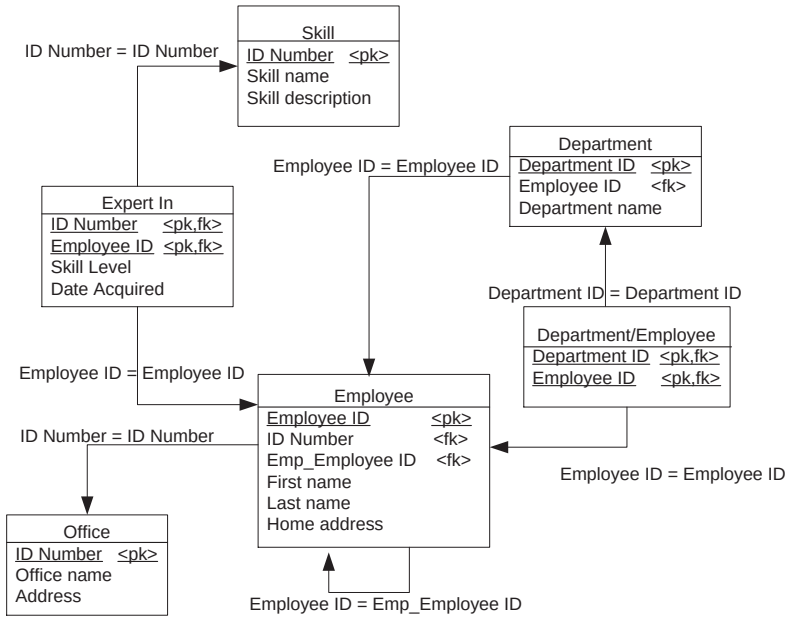
これらすべての条件を満たしていれば、作成した設計を実装できます。

最終設計

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



対応する物理データ・モデルは次のようになります。


Skill
ID Number <pk>
Skill name
Skill description
ID Number = ID Number
Department
Department ID <pk>
Employee ID <fk>
Department name
Employee ID = Employee ID
Expert In
ID Number <pk,fk>
Employee ID <pk,fk>
Skill Level
Date Acquired
Department ID = Department ID
Department/Employee
Department ID <pk,fk>
Employee ID <pk,fk>
Employee ID = Employee ID
Employee
Employee ID <pk>
ID Number <fk>
Emp_Employee ID <fk>
First name
Last name
Home address
ID Number = ID Number
Employee ID = Employee ID
Office
ID Number <pk>
Office name
Address
Employee ID = Emp_Employee ID

テーブル・プロパティの設計

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

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

カラム名の選択

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

カラムのデータ型の選択

次に、Adaptive Server Anywhere で使用可能なデータ型を示します。

- 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、『ASA SQL リファレンス・マニュアル』>「SQL データ型」を参照してください。

long binary データ型は、イメージ (たとえばビット・マップとして格納されるもの) やワープロ文書をデータベースに格納するのに使用できます。このデータ型の情報は、通常バイナリ・ラージ・オブジェクトまたは BLOBS と呼ばれます。

各データ型の詳細については、『ASA SQL リファレンス・マニュアル』> 「SQL データ型」を参照してください。

NULL と非 NULL

カラム値がレコードに対して必須である場合、そのカラムを非 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 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、カラムには非 NULL を明示的に宣言してください。

NULL 値の詳細については、『ASA SQL リファレンス・マニュアル』> 「NULL 値」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、『ASA SQL リファレンス・マニュアル』> 「探索条件」を参照してください。

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、『ASA SQL リファレンス・マニュアル』> 「探索条件」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」101 ページを参照してください。

例

サンプル・データベースの Department テーブルには、カラム dept_id、dept_name、dept_head_id があります。その定義は次のとおりです。

カラム	データ型	サイズ	Null ／非 Null	制約
dept_id	integer	-	非 null	なし

カラム	データ型	サイズ	**Null ／非 Null**	制約
dept_name	char	40	非 null	なし
dept_head_id	integer	-	null	なし

非 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。

第 2 章

データベース・オブジェクトの使用

この章の内容

この章では、テーブル、ビュー、インデックスなどのデータベース・オブジェクトの作成、変更、削除のメカニズムについて説明します。

概要

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

この章では、データベースとそのオブジェクトの作成方法について説明します。この中には、Sybase Central、Interactive SQL、コマンド・ライン・ユーティリティの手順も含まれています。この章を開始するにあたり、概念的な情報が必要な場合は次の章を参照してください。

- 「データベースの設計」3 ページ
- 「データ整合性の確保」101 ページ
- 『SQL Anywhere Studio の紹介』> 「Sybase Central について」
- 『SQL Anywhere Studio の紹介』> 「Interactive SQL の使用」

この章のタスクを実行する SQL 文は、「**データ定義言語**」(DDL) と呼ばれます。データベース・オブジェクトの定義は、データベース・スキーマを形成します。スキーマを空のデータベースとみなすことができます。

プロシージャとトリガもデータベース・オブジェクトですが、これらについては、「プロシージャ、トリガ、バッチの使用」811 ページを参照してください。

章の内容

この章では、次の情報について説明します。

- データベース・オブジェクトを使用した作業の概要 (この項)
- データベースの作成方法と使用方法
- テーブル、ビュー、インデックスの作成方法と変更方法

データベースの編集

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

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

- データベース・ファイルには、データベースを構築する場合のスキーマ定義を保持する、システム・テーブルも含まれています。

データベースの作成

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

トランザクション・ログ

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

トランザクション・ログの配置の詳細については、『ASA データベース管理ガイド』> 「データ保護を目的としたデータベースの構成」を参照してください。

データベース・ファイルの互換性

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

大きなファイルに対するサイズが制限される場合や、Adaptive Server Anywhere がサポートする場合以外は、すべてのオペレーティング・システム間でデータベース・ファイルには互換性があります。オペレーティング・システム間の互換は、データベース・ファイルをコピーすれば可能です。同様に、パーソナル・サーバで作成されたデータベースは、ネットワーク・サーバで使用できます。Adaptive Server Anywhere サーバは、ソフトウェアの旧バージョンで作成されたデータベースを管理できますが、旧バージョンのサーバは新バージョンのデータベースを管理できません。

制限事項の詳細については、『ASA データベース管理ガイド』> 「サイズと数の制限」を参照してください。

異なるアプリケーションを使用してデータベースを作成する

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

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

さらに高度な使用を実現するために、Sybase PowerDesigner などのデータベース設計ツールでは、すぐれた設計のデータベースを開発するための徹底的で信頼性の高い方法を提供します。

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

データベースの作成 (Sybase Central の場合)

Sybase Central では、[データベースの作成] ユーティリティを使用してデータベースを作成できます。データベースを作成すると、Sybase Central の左ウィンドウ枠内のサーバの下に、そのデータベースが表示されます。

詳細については、「データベースの作成 (SQL の場合)」40 ページ と 「データベースの作成 (コマンド・ラインの場合)」41 ページ を参照してください。

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

1. [ツール] － [Adaptive Server Anywhere 9] － [データベースの作成] を選択します。
2. ウィザードの指示に従います。

❖ 現在の接続に基づいた新しいデータベースを作成するには、次の手順に従います。

1. データベースに接続します。
2. 左ウィンドウ枠で、Adaptive Server Anywhere プラグインを選択します。
3. 右ウィンドウ枠で、[ユーティリティ] タブをクリックします。
4. 右ウィンドウ枠で、[データベースの作成] をダブルクリックします。
5. ウィザードの指示に従います。

Windows CE 用データベースの作成

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

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

データベースの作成 (SQL の場合)

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

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

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

詳細については、『ASA SQL リファレンス・マニュアル』> 「CREATE DATABASE 文」を参照してください。

例

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

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

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

円記号 (¥) は SQL のエスケープ文字であるため、2 つ付けます。詳細については、『ASA SQL リファレンス・マニュアル』> 「文字列」を参照してください。

データベースの作成 (コマンド・ラインの場合)

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

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

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

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

```
dbinit -p 4096 company.db
```

詳細については、『ASA データベース管理ガイド』> 「初期化ユーティリティ」を参照してください。

データベースの消去

データベースを消去すると、データベースへの変更を記録したトランザクション・ログを含むすべてのテーブルとデータがディスクから削除されます。データベース・ファイルはすべて、誤ってファイルを変更したり削除したりするのを防ぐために読み込み専用となっています。

Sybase Central では、[データベースの消去] ユーティリティを使用してデータベースを消去できます。このユーティリティにアクセスするには、データベースのどれかに接続する必要がありますが、[データベース消去] ウィザードからはほかのデータベースも消去できます。実行していないデータベースを消去するには、そのデータベース・サーバを起動しておきます。

Interactive SQL では、DROP DATABASE 文を使用してデータベースを消去できます。データベース・サーバの -gu コマンド・ライン・オプションを使用して、必要なパーミッションを設定できます。デフォルトの設定では、DBA 権限を必要とします。

dberase ユーティリティを使用して、コマンド・ラインからデータベースを消去する方法もあります。このユーティリティの使用中に、消去するデータベースを起動しないでください。

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

1. 左ウィンドウ枠で、Adaptive Server Anywhere プラグインを選択します。
2. 右ウィンドウ枠で、[ユーティリティ] タブをクリックします。
3. 右ウィンドウ枠で、[データベースの消去] をダブルクリックします。
4. ウィザードの指示に従います。

❖ データベースを消去するには、次の手順に従います (SQL の場合)。

- DROP DATABASE 文を実行します。

  次に例を示します。

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

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

- コマンド・ラインから、*dberase* ユーティリティを実行します。

  次に例を示します。

```
dberase company.db
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「DROP DATABASE 文」と『ASA データベース管理ガイド』> 「消去ユーティリティ」を参照してください。

データベースとの接続の切断

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

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

```
SELECT connection_property( 'number' )
```

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

1. データベースを選択します。
2. [ツール] ー [切断] を選択します。

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

- DISCONNECT 文を実行します。

例 1

次の文は、Interactive SQL で DISCONNECT を使用して、すべての接続を切断する方法を示します。

```
DISCONNECT ALL
```

例 2

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

```
EXEC SQL DISCONNECT :conn_name
```

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

1. DBA 権限を使用して、既存のデータベースに接続します。
2. DROP CONNECTION 文を実行します。

例

次の文は、ID 番号 4 の接続を削除します。

```
DROP CONNECTION 4
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「DISCONNECT 文 [ESQL] [Interactive SQL]」と 『ASA SQL リファレンス・マニュアル』> 「DROP CONNECTION 文」を参照してください。

データベース・オブジェクトのプロパティの設定

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

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

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

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

1. Sybase Central の左ウィンドウ枠で、対象のオブジェクトがあるフォルダを開きます。

2　オブジェクトを選択します。オブジェクトのプロパティが Sybase Central の右ウィンドウ枠に表示されます。

3　右ウィンドウ枠で、対象プロパティを編集できる該当タブをクリックします。

オブジェクトのプロパティ・シートでも、プロパティを表示、編集できます。プロパティ・シートを表示するには、オブジェクトを右クリックし、ポップアップ・メニューで [プロパティ] を選択します。

データベース・オプションの設定

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

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

1　対象のサーバを開きます。

2　目的のデータベースを右クリックし、ポップアップ・メニューで [オプション] を選択します。

3　値を編集します。

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

- SET OPTION 文で対象のプロパティを指定します。

ヒント
[データベースのオプション] ダイアログからも、指定のユーザとグループに対してデータベース・オプションを設定できます (あるユーザまたはグループに対してこのダイアログを開いた場合、それぞれを [ユーザのオプション] ダイアログまたは [グループのオプション] ダイアログと呼びます)。

データベース自体のオプションを設定する場合は、実際にデータベースの PUBLIC グループのオプションを設定します。これは、すべてのユーザとグループが、PUBLIC からオプションの設定を継承するためです。

統合データベースの指定

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

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

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

1. 対象のデータベースを右クリックし、ポップアップ・メニューで [プロパティ] を選択します。
2. [SQL Remote] タブをクリックします。
3. [このリモート・データベースに対応する統合データベースが存在する] オプションを選択します。
4. 必要な設定を実行します。

データベースのシステム・オブジェクトの表示

データベース内のテーブル、ビュー、ストアド・プロシージャ、またはドメインは、システム・オブジェクトです。システム・ビュー、プロシージャ、ドメインが Sybase Transact-SQL の互換性を広くサポートする場合、システム・テーブルにはデータベース自身の情報が格納されます。

Interactive SQL では、システム・オブジェクトすべてのリストは表示できませんが、システム・テーブルの内容をブラウズできます。詳細については、「システム・テーブルの表示」67 ページを参照してください。

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

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

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

詳細については、『ASA SQL リファレンス・マニュアル』> 「システム・テーブル」を参照してください。

データベース編集時の SQL 文のロギング

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

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

Interactive SQL における文のロギングについては、『SQL Anywhere Studio の紹介』> 「コマンドのロギング」を参照してください。

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

1 [ビュー] ー [設計の詳細] を選択します。

[Sybase Central] ウィンドウの最下部に [設計の詳細] ウィンドウ枠が表示されます。

2 [設計の詳細] ウィンドウ枠で右クリックし、ポップアップ・メニューで [オプション] を選択します。

[オプション] ダイアログが表示されます。

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

接続しないでデータベースを起動する

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

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

1 対象のサーバを右クリックし、ポップアップ・メニューで [データベースの開始] を選択します。

2 [データベースの開始] ダイアログに必要な値を入力します。

データベースが、サーバの下に未接続のデータベースとして表示されます。

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

1. Interactive SQL を起動します。
2. START DATABASE 文を実行します。

例

sample サーバ上で、データベース・ファイル *c:¥asa9¥sample_2.db* を sam2 として起動します。

```
START DATABASE 'c:¥asa9¥sample_2.db'
 AS sam2
 ON sample
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「START DATABASE 文 [Interactive SQL]」を参照してください。

jConnect メタデータ・サポートを既存のデータベースにインストールする

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

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

1. アップグレードするデータベースに接続します。
2. 左ウィンドウ枠で、Adaptive Server Anywhere プラグインを選択します。
3. 右ウィンドウ枠で、[ユーティリティ] タブをクリックします。
4. 右ウィンドウ枠で、[データベースのアップグレード] をダブルクリックします。
5. ウィザードの概要ページで [次へ] をクリックします。

6 アップグレードするデータベースをリストから選択します。[次へ] をクリックします。

7 必要であれば、データベースのバックアップ作成も選択できます。[次へ] をクリックします。

8 [jConnect メタ情報サポートをインストール] オプションを選択します。

9 ウィザードの指示に従います。

テーブルの編集

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

この項では、テーブルを作成、変更、削除する方法について説明します。例は Interactive SQL 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。

詳細については、『SQL Anywhere Studio の紹介』> 「Interactive SQL でのテーブル値の編集」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、CREATE TABLE 文と ALTER TABLE 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、『ASA データベース管理ガイド』> 「グループが所有するテーブルの参照」と『ASA データベース管理ガイド』> 「データベース・オブジェクトの名前とプレフィクス」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。次に、Interactive SQL の SQL 文または Sybase Central のいずれかを使用して、実際のデータを保持する新しいテーブルを作成できます。

作成できるテーブルのタイプは次の 2 つです。

- **ベース・テーブル**　永続的なデータを保管するテーブルです。テーブルとそのデータは、それらを明示的に削除しないかぎり存在し続けます。テンポラリ・テーブルやビューと区別するため、これをベース・テーブルと呼びます。
- **テンポラリ・テーブル**　テンポラリ・テーブルのデータは、単一の接続に対してだけ保持されます。グローバル・テンポラリ・テーブルの定義は、それが削除されないかぎりデータベースに

保持されます (データはこのかぎりではありません)。ローカル・テンポラリ・テーブルの定義とデータは、単一の接続の間だけ存在します。

テンポラリ・テーブルの詳細については、「テンポラリ・テーブルの編集」97 ページを参照してください。

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

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

1 データベースに接続します。

2 [テーブル] フォルダを開きます。

3 [ファイル] メニューから [新規] － [テーブル] を選択します。

[テーブル作成] ウィザードが表示されます。

4 [テーブル作成] ウィザードで次の作業を行います。

- 新しく作成するテーブルの名前を入力する。
- テーブルの所有者を選択する。
- テーブルのプライマリ・キーを作成する。
- その他の必要なオプションを設定する。

5 [完了] をクリックします。

6 右ウィンドウ枠の [カラム] タブで、テーブルにカラムを追加できます。

7 終了したら、[ファイル] － [保存] を選択します。

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

1 DBA 権限を使用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 skill (
     skill_id INTEGER NOT NULL,
     skill_name CHAR( 20 ) NOT NULL,
     skill_type CHAR( 20 ) NOT NULL
 )
```

詳細については、『ASA SQL リファレンス・マニュアル』>「CREATE TABLE 文」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 では、右ウィンドウ枠の [SQL] タブでこれらの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。

Sybase Central で作業する間、[カラム] フォルダで選択したカラムがあれば、メニュー・コマンドを使用してカラムを管理する (プライマリ・キーからの追加または削除、プロパティの変更、または削除) こともできます。

データベース・オブジェクト・プロパティの変更については、「データベース・オブジェクトのプロパティの設定」44 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、『ASA データベース管理ガイド』> 「テーブルに対するパーミッションの付与」と『ASA データベース管理ガイド』> 「ユーザ・パーミッションの取り消し」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。

❖ 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. データベースに接続します。
2. [テーブル] フォルダを開きます。
3. 左ウィンドウ枠で、変更するテーブルを選択します。
4. 右ウィンドウ枠で [カラム] タブをクリックし、必要な変更を行います。

ヒント

テーブルの [カラム] タブを選択し、[ファイル] － [カラムを追加] を選択して、カラムを追加する方法もあります。

カラムを削除するには、[カラム] タブでカラムを選択し、[編集] － [削除] を選択します。

カラムをテーブルにコピーするには、右ウィンドウ枠の [カラム] タブでカラムを選択し、[コピー] ボタンをクリックします。対象のテーブルを選択し、右ウィンドウ枠の [カラム] タブをクリックして、次に [貼り付け] ボタンをクリックします。

また、[テーブルの保存] ボタンをクリックするか、[ファイル] － [テーブルの保存] 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。

詳細については、『ASA SQL リファレンス・マニュアル』> 「ALTER TABLE 文」を参照してください。

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。

❖ 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 権限を使用してデータベースに接続します。
2. ALTER TABLE 文を実行します。

例

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 skill テーブルに追加します。

```
ALTER TABLE skill
 ADD skill_description CHAR( 254 )
```

skill_description カラムが追加され、スキルの説明を数行分記入できます。

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、サンプル・データベースの skill_description カラムを最大 254 文字から最大 80 文字に変更します。

```
ALTER TABLE skill
 MODIFY skill_description CHAR( 80 )
```

80 文字を超える既存の記述は自動的にカットされ、警告が表示されます。

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

```
ALTER TABLE skill
 RENAME skill_type TO classification
```

次の文は、classification カラムを削除します。

```
ALTER TABLE skill
 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 skill
 RENAME qualification
```

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ります。

詳細については、『ASA SQL リファレンス・マニュアル』> 「ALTER TABLE 文」と「データ整合性の確保」101 ページを参照してください。

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。

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

1. データベースに接続します。
2. データベースの [テーブル] フォルダを開きます。
3. テーブルを右クリックし、ポップアップ・メニューで [削除] を選択します。

❖ **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 権限を使用してデータベースに接続します。

2 DROP TABLE 文を実行します。

例

次に示す DROP TABLE コマンドは、skill 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 skill テーブルの定義を削除します。

```
DROP TABLE skill
```

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 文も、テーブル上のインデックスをすべて削除します。

詳細については、『ASA SQL リファレンス・マニュアル』> 「DROP 文」を参照してください。

テーブル内の情報のブラウズ

Sybase Central または Interactive SQL 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 から、テーブル内のデータを編集できます。

プライマリ・キーの管理

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

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、テーブル作成時のカラムの順序によって決定されます。カラムの順序は、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)

Sybase Central では、テーブルのプライマリ・キーは次の場所に表示されます。

- テーブルのプロパティ・シートの [カラム] タブ
- 左ウィンドウ枠でテーブルが選択されている場合、右ウィンドウ枠の [カラム] タブ

テーブルのプロパティ・シートと [カラム] タブのリストには、いずれもプライマリ・キーのカラムが (非キー・カラムとともに) 表示されます。プライマリ・キー・カラムの順序は、テーブル作成時のカラムの順序によって決定されます。ただし、プライマリ・キー内では、カラムの順序は実際と異なる場合があります。

❖ プライマリ・キーを作成／編集するには、次の手順に従います (Sybase Central の場合)。

1. [テーブル] フォルダを開きます。

2　左ウィンドウ枠で対象のテーブルを選択します。

3　右ウィンドウ枠で、[カラム] タブをクリックします。

4　希望のプライマリ・キーを変更します。

カラムをプライマリ・キー・カラムにするには、カラム名の横のチェック・ボックスにチェック・マークを付けてください。プライマリ・キーからカラムを削除するには、チェック・ボックスのチェック・マークを外してください。

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、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。

❖ 既存のテーブルのプライマリ・キーを修正するには、次の手順に従います (SQL の場合)。

1　DBA 権限を使用してデータベースに接続します。

2　ALTER TABLE 文を実行します。

例 1

次の文は、前の例と同様に skill テーブルを作成します。ただし、今回はプライマリ・キーが追加されます。

```
CREATE TABLE skill (
   skill_id INTEGER NOT NULL,
   skill_name CHAR( 20 ) NOT NULL,
   skill_type CHAR( 20 ) NOT NULL,
   primary key( skill_id )
 )
```

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 skill_id を持つローは複数設定できません。テーブル内の各ローは、プライマリ・キーでユニークに識別されます。

例 2

次の文は、skill テーブルのプライマリ・キーにカラム skill_id と skill_type を追加します。

```
ALTER TABLE skill
   ADD PRIMARY KEY ( "skill_id", "skill_type" )
```

ALTER TABLE 文で PRIMARY KEY 句を指定する場合は、CREATE TABLE 文または別の ALTER TABLE 文が作成したプライマリ・キーを、テーブルが持たないようにしてください。

例 3

次の文は、skill テーブルのプライマリ・キーからカラムをすべて削除します。プライマリ・キーを削除する前に、使用するデータベースへの影響を確認してください。

```
ALTER TABLE skill
   DELETE PRIMARY KEY
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「ALTER TABLE 文」と「プライマリ・キーの管理 (Sybase Central の場合)」58 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。

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 [外部キー] タブに表示されます。

プライマリ・テーブルのプライマリ・キーの値とテーブルに含まれる外部カラムの値が一致しない場合は、テーブルに外部キーを作成できません。

外部キーを作成すると、右ウィンドウ枠の各テーブルの [参照元テーブル] タブで追跡できます。このフォルダには、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。

❖ 指定されたテーブルで新しい外部キーを作成するには、次の手順に従います (Sybase Central の場合)。

1 外部キーを作成するテーブルを選択します。

2 右ウィンドウ枠で [外部キー] タブをクリックします。

3 [ファイル] － [新規] － [外部キー] を選択します。

[外部キー作成] ウィザードが表示されます。

4 ウィザードの指示に従います。

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

1 外部キーを削除するテーブルを選択します。

2 右ウィンドウ枠で [外部キー] タブをクリックします。

3 削除する外部キーを右クリックし、ポップアップ・メニューで [削除] を選択します。

❖ 指定されたテーブルを参照する外部キーを持つテーブルを表示するには、次の手順に従います (Sybase Central の場合)。

1 左ウィンドウ枠で対象のテーブルを選択します。

2 右ウィンドウ枠で、[参照元テーブル] 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、[外部キー] タブで外部キーを右クリックし、ポップアップ・メニューで [プロパティ] を選択します。

[参照元テーブル] タブでテーブルを右クリックして、ポップアップ・メニューで [プロパティ] を選択すると、参照元テーブルのプロパティを表示できます。

外部キーの管理 (SQL の場合)

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

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

❖ 既存のテーブルの外部キーを修正するには、次の手順に従います (SQL の場合)。

1. DBA 権限を使用してデータベースに接続します。
2. ALTER TABLE 文を実行します。

例 1

従業員が習得した各スキルのスキル・レベルについて説明したデータを保管する emp_skill テーブルを、次のように作成します。

```
CREATE TABLE emp_skill(
     emp_id INTEGER NOT NULL,
     skill_id INTEGER NOT NULL,
     "skill level" INTEGER NOT NULL,
     PRIMARY KEY( emp_id, skill_id ),
     FOREIGN KEY REFERENCES employee,
     FOREIGN KEY REFERENCES skill
 )
```

emp_skill テーブルには、2 つのカラム emp_id と skill_id で構成されるプライマリ・キーがあります。1 人の従業員が複数のスキルを持っている場合は、複数のローに表示されます。また、複数の従業員があるスキルを持っている場合は、skill_id カラムが何度か表示されることがあります。ただし、ある従業員とスキルの組み合わせのエントリは 1 つだけです。

emp_skil テーブルには、2 つの外部キーがあります。外部キーのエントリは、emp_id カラムには employee テーブルにある有効な従業員番号が必要であり、skill_id カラムには skill テーブルにある有効なエントリが必要であることを示します。

例 2

次のように、foreignkey という名前の外部キーを既存のテーブル skill に追加して、テーブル contact のプライマリ・キーに対するこの外部キーを参照できます。

```
ALTER TABLE skill
 ADD FOREIGN KEY "foreignkey" ("skill_id")
 REFERENCES "DBA"."contact" ("id")
```

この例では、skill テーブル (外部テーブル) の skill_id カラムと contact テーブル (プライマリ・テーブル) の id カラムの間に関係を確立します。"DBA" はテーブル contact の所有者を示します。

例 3

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

```
ALTER TABLE skill
 ADD NOT NULL FOREIGN KEY "foreignkey" ("skill_id")
 REFERENCES "DBA"."contact" ("id")
 ON UPDATE RESTRICT
 ON DELETE RESTRICT
```

Sybase Central では、[外部キー作成] ウィザード、または外部キーのプロパティ・シートで、プロパティを指定することもできます。

詳細については、『ASA SQL リファレンス・マニュアル』> 「ALTER TABLE 文」と「外部キーの管理 (Sybase Central の場合)」60 ページ を参照してください。

計算カラムの使用

計算カラムは、同一ロー内にある従属カラムと呼ばれる他のカラムの値を参照する式を値としているカラムのことです。計算カラムは、1つまたは複数の従属カラムの値を含む複雑な式をインデックス化するような場合に特に便利です。クエリの最適化中、Adaptive Server 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(
    SHIPMENT_ID   INTEGER NOT NULL PRIMARY KEY,
    SHIPMENT_DATE TIMESTAMP,
    PRODUCT_CODE  CHAR(20) NOT NULL,
    QUANTITY      INTEGER NOT NULL,
    TOTAL_PRICE   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 ( TOTAL_PRICE / QUANTITY ) BETWEEN 2.00 AND 4.00
```

しかし、WHERE 句の述部は、単一ベースのカラムを参照しないため、検索引数が使用できません。SHIPMENTS テーブルのサイズが比較的大きい場合、インデックス検索の方が逐次スキャンより適している場合があります。これは、次のように SHIPMENT テーブルの計算カラム AVERAGE_COST を作成することで可能です。

```
ALTER TABLE shipments
 ADD average_cost DECIMAL(30,22)
 COMPUTE (total_price / quantity)

 CREATE INDEX idx_avg_cost ON shipments( average_cost
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、Adaptive Server 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 (TOTAL_PRICE / QUANTITY ) AS X
FROM
 SHIPMENTS', 1 )
 FROM DUMMY
```

SHIPMENTS テーブルに対して、上記のクエリは **NUMERIC(30,22)** を返します。最適化中に、Adaptive Server 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 AVERAGE_COST BETWEEN 2.00 AND 4.00
```

ここで 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_avg_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。計算カラムの値は、ローが挿入され更新されると自動的にデータベース・サーバによって維持されます。ほとんどのアプリケーションでは、計算カラム値を直接更新したり挿入したりする必要はありませんが、計算カラムは他と同様にベース・カラムなので、機能する場合に述部や式で直接参照される場合があります。

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

計算カラム式の修正

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

計算カラムにより、INSERT 文と UPDATE 文は制約を受けます。

- **直接挿入または直接更新は許可されない**　計算カラムには、値を直接挿入できません。カラム Computed が計算カラムの場合は、次の文は 挿入したカラムが重複しています。エラーで失敗します。

  ```
  -- Incorrect statement
   INSERT INTO T1 (id, computed)
   VALUES( 3006, 'bad insert statement' )
  ```

  同様に、計算カラムの値を直接変更する UPDATE 文は使用できません。

- **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。

計算カラムが再計算されない場合

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

データベース内／間でテーブルまたはカラムをコピーする

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

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

- 指定されたロケーションへ SELECT 文の結果を挿入するには、『ASA SQL リファレンス・マニュアル』> 「SELECT 文」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、『ASA SQL リファレンス・マニュアル』> 「INSERT 文」を参照してください。

システム・テーブルの表示

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

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

詳細については、『ASA SQL リファレンス・マニュアル』> 「システム・テーブル」を参照してください。

❖ システム・テーブルを表示するには、次の手順に従います (Sybase Central の場合)。

1 接続済みの対象データベースを右クリックし、ポップアップ・メニューで [所有者別にオブジェクトをフィルタ] を選択します。

2 SYS の横にあるチェック・ボックスを選択して、[OK] をクリックします。

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

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

1 データベースに接続します。

2 SELECT 文を実行し、ブラウズするシステム・テーブルを指定します。システム・テーブルの所有者はユーザ ID SYS です。

例

[結果] ウィンドウ枠の [結果] タブにある sys.systable テーブルの内容を表示します。

```
SELECT *
 FROM SYS.SYSTABLE
```

ビューの編集

ビューは計算テーブルです。ビューを使用して、データベースのユーザに正確な情報を、制御できるフォーマットで表示できます。

ビューとベース・テーブルの類似点

ビューは、次のような点で、データベースの永久テーブル (ベース・テーブルとも呼ばれます) に似ています。

- ベース・テーブルと同様に、ビューにもアクセス・パーミッションを設定できます。
- ビューにも SELECT クエリを実行できます。
- ビューによっては、UPDATE、INSERT、DELETE を実行できます。
- ほかのビューに基づいたビューを作成できます。

ビューと永久テーブル間の相違点

ビューと永久テーブルの違いを次に示します。

- ビューにはインデックスを設定できません。
- すべてのビューに UPDATE、INSERT、DELETE を実行できるわけではありません。
- ビューには整合性の制約とキーを設定できません。
- ビューはベース・テーブル中の情報を参照しますが、その情報を自分では持っていません。ビューは呼び出されるたびに計算されます。

アクセスを調整する利点

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。

- **セキュリティの向上**　関連データへのアクセスだけが許可されます。
- **利便性の向上**　ベース・テーブルよりも見やすい形でユーザとアプリケーション開発者にデータを提供します。
- **一貫性の向上**　よく使われるクエリの定義をデータベース内で集中管理します。

ビューの作成

データをブラウズするとき、SELECT 文は 1 つ以上のテーブルで動作して、結果セット、つまり同様にテーブルを作成します。ベース・テーブルと同様に、SELECT 文の結果セットにもカラムとローがあります。ビューを作成するクエリには名前が与えられ、システム・テーブルに定義が格納されます。

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

```
SELECT dept_ID, count(*)
 FROM employee
 GROUP BY dept_ID
```

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。

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

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

   [ビュー作成] ウィザードが表示されます。
4. ウィザードの指示に従います。ウィザードの終了後、右ウィンドウ枠の [SQL] タブでコードを編集できます。
5. 使用するテーブルとカラムを入力して、そのコードの編集を終了します。上記の例では、**employee** と **dept_ID** を入力します。
6. [ファイル] － [ビューの保存] を選択します。

   新しいビューが [ビュー] フォルダに表示されます。

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

1 データベースに接続します。

2 CREATE VIEW 文を実行します。

例

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

```
CREATE VIEW DepartmentSize AS
 SELECT dept_ID, count(*)
 FROM employee
 GROUP BY dept_ID
```

ビューの情報はデータベース内に分けて格納されないため、ビューを参照すると関連した SELECT 文が実行され、正しいデータが取得されます。

この方法の長所は、employee テーブルが修正されると、DepartmentSize ビューの情報も自動的に更新されることです。一方、複雑な SELECT 文を使用すると、正しい情報を検索する SQL の要求回数が、ビューを使用するたびに増加する場合があります。

詳細については、『ASA SQL リファレンス・マニュアル』>「CREATE VIEW 文」を参照してください。

ビューの使い方

SELECT 文に対する制限

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

ビューを作成するには、必要とする正確な結果が必要なフォーマットで得られるまで SELECT クエリを編集します。SELECT クエリが作成できたら、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
```

ビューの更新

UPDATE、INSERT、DELETE は、ビューを作成する SELECT 文の内容によっては、そのビューに対して使用できます。

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

ビューのコピー

Sybase Central では、データベース間でビューをコピーできます。この作業を実行するには、Sybase Central の右ウィンドウ枠でビューを選択し、別の接続済みデータベースの [ビュー] フォルダへそれをドラッグします。新しいビューが作成されて、元のビューのコード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、ビューのコード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 句の使い方

ビューに対して INSERT または UPDATE 文が使用可能であっても、基本となるテーブルに追加または更新されたローが、ビュー自体の目的に合わないこともあります。たとえば、INSERT または UPDATE 文で基本となるテーブルを修正しても、ビューには新しいローはありません。

WITH CHECK OPTION 句の使用例

次の例では、WITH CHECK OPTION 句の利便性について説明しています。このオプション句は、CREATE VIEW 文の最後の句です。

❖ **営業部の従業員を表示するビューを作成するには、次の手順に従います (SQL の場合)。**

1 次の文を入力します。

```
CREATE VIEW sales_employee
 AS SELECT emp_id,
         emp_fname,
         emp_lname,
         dept_id
 FROM employee
 WHERE dept_id = 200
```

このビューは、次のコマンドと同様のデータを持ちます。

```
SELECT *
 FROM sales_employee
```

Interactive SQL では、次のように表示されます。

emp_id	emp_fname	emp_lname	dept_id
129	Philip	Chin	200
195	Marc	Dill	200
299	Rollin	Overbey	200
467	James	Klobucher	200
…	…	…	…

2 **Philip Chin をマーケティング部門に移動する**　次のようにビューを更新すると、ビューの選択基準と一致しないエントリがビューから消えます。

```
UPDATE sales_employee
 SET dept_id = 400
 WHERE emp_id = 129
```

3 **営業部の全従業員を表示する**　ビューを調べます。

```
SELECT *
 FROM sales_employee
```

Emp_id	Emp_fname	Emp_lname	Dept_id
129	Philip	Chin	200
195	Marc	Dill	200

Emp_id	Emp_fname	Emp_lname	Dept_id
299	Rollin	Overbey	200
467	James	Klobucher	200
…	…	…	…

4　営業部の全従業員を表示する　ビューを調べます。

```
SELECT *
 FROM sales_employee
```

Emp_id	emp_fname	emp_lname	dept_id
195	Marc	Dill	200
299	Rollin	Overbey	200
467	James	Klobucher	200
641	Thomas	Powell	200
…	…	…	…

ビュー作成時に WITH CHECK OPTION を使用すると、完成したローがビュー作成条件を満たすように、ビューに対する UPDATE 文または INSERT 文が検査されます。条件が合わない場合は、エラーが生じて操作は拒否されます。

次に示す修正された sales_employee ビューは、更新文を拒否し、次のエラー・メッセージを返します。

```
Invalid value for column 'dept_id' in table 'employee'
```

- **営業部の従業員を表示するビューを作成する (2 度目)**　ここでは、WITH CHECK OPTION を使用します。

```
CREATE VIEW sales_employee
 AS SELECT emp_id, emp_fname, emp_lname, dept_id
     FROM employee
     WHERE dept_id = 200
 WITH CHECK OPTION
```

検査オプションの継承

sales_employee ビューを使用してさらにビュー (仮に V2 と呼びます) が作成された場合は、V2 自体に検査オプションがなくても、sales_employee ビューの WITH CHECK OPTION の基準に合わなければ、V2 に対する更新や挿入は拒否されます。

ビューの変更

Sybase Central と Interactive SQL のどちらを使用してもビューを修正できます。この場合、既存のビューの名前を直接変更することはできません。代わりに、新しい名前を付けて新しくビューを作成し、以前のコードをそこにコピーしてから、元のビューを削除します。

Sybase Central では、ビュー、プロシージャ、関数のコードは、右ウィンドウ枠の各オブジェクトの [SQL] タブで編集できます。ビューを右クリックし、ポップアップ・メニューで [新しいウィンドウで編集] を選択すると、ビューを別のウィンドウで編集できます。Interactive SQL では、ALTER VIEW 文を使用して、ビューを修正できます。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。

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」44 ページを参照してください。

パーミッションの設定の詳細については、『ASA データベース管理ガイド』> 「テーブルに対するパーミッションの付与」と『ASA データベース管理ガイド』> 「ビューに対するパーミッションの付与」を参照してください。パーミッションの取り消しについては、『ASA データベース管理ガイド』> 「ユーザ・パーミッションの取り消し」を参照してください。

❖ **ビュー定義を編集するには、次の手順に従います (Sybase Central の場合)。**

1. [ビュー] フォルダを開きます。
2. 対象のビューを選択します。

3 右ウィンドウ枠で、[SQL] タブをクリックし、ビューのコードを編集します。

ヒント
複数のビューを編集する場合は、各ビューを右ウィンドウ枠の [SQL] タブで編集するより、各ビューに対して別のウィンドウを開く方が便利な場合があります。ビューを右クリックし、ポップアップ・メニューで [新しいウィンドウで編集] を選択すると、別のウィンドウを開くことができます。

❖ ビュー定義を編集するには、次の手順に従います (SQL の場合)。

1 DBA 権限を使用して、またはビューの所有者として、データベースに接続します。

2 ALTER VIEW 文を実行します。

例

DepartmentSize ビュー (「ビューの作成」70 ページの項を参照) のカラム名を、よりわかりやすい名前に変更します。

```
ALTER VIEW DepartmentSize
   (Dept_ID, NumEmployees)
 AS
   SELECT dept_ID, count(*)
   FROM Employee
   GROUP BY dept_ID
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「ALTER VIEW 文」を参照してください。

ビューの削除

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

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

1. [ビュー] フォルダを開きます。
2. 対象のビューを右クリックし、ポップアップ・メニューで [削除] を選択します。

❖ 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 権限を使用して、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 VIEW 文を実行します。

例

DepartmentSize ビューを削除します。

```
DROP VIEW DepartmentSize
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「DROP 文」を参照してください。

ビュー内の情報のブラウズ

ビュー内のデータをブラウズするため、Interactive SQL ユーティリティを使用できます。このユーティリティでは、クエリを実行して、表示するデータを識別します。これらのクエリの使用の詳細については、「クエリ：テーブルからのデータの選択」275 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右クリックし、ポップアップ・メニューで [Interactive SQL によるデータ表示] を選択します。Interactive SQL が起動して、[結果] ウィンドウ枠の [結果] タブにビューの内容が表示されます。そのビューをブラウズできるようにするために、Interactive SQL は `select * from owner.view` 文を実行します。

システム・テーブル内のビュー

データベースのビューに関するすべての情報は、システム・テーブル SYS.SYSTABLE に入っています。システム・ビュー SYS.SYSVIEWS は、この情報を見やすい形で表示します。詳細については、『ASA SQL リファレンス・マニュアル』> 「SYSTABLE システム・テーブル」と 『ASA SQL リファレンス・マニュアル』> 「SYSVIEWS システム・ビュー」を参照してください。

Interactive SQL を使って、これらのテーブルの情報をブラウズできます。[SQL 文] ウィンドウ枠に次の文を入力すれば、SYS.SYSVIEWS ビューのカラムがすべて表示されます。

```
SELECT *
 FROM SYS.SYSVIEWS
```

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 = 'DepartmentSize';
 OUTPUT TO viewtext.sql
 FORMAT ASCII
```

インデックスの編集

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

インデックス・セットの選択

データベースに適切なインデックス・セットを選択することは、パフォーマンスを最適化する上で重要です。適切なセットを識別することは、労力を要する作業でもあります。いくつかのインデックスによって得られるパフォーマンス上の利益は重要ですが、インデックスに伴うコストも記憶領域とデータ修正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・コンサルタントは、単一のクエリまたは一連の操作を分析して、データベースに追加するインデックスを推奨します。また、使用されていないインデックスを通知します。

インデックス・コンサルタントの詳細については、「インデックス・コンサルタントの起動」89 ページを参照してください。

インデックスはいつ使うか

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

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

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

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

インデックス・コンサルタントの詳細については、「インデックス・コンサルタントの起動」89 ページを参照してください。

頻繁に検索するカラムにインデックスを使う

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

パフォーマンスの詳細については、「インデックスの使用」213 ページを参照してください。

Adaptive Server 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。

インデックスを使用することでデータベース文のパフォーマンスが向上する場合は、Adaptive Server Anywhere は自動的にインデックスを使用します。インデックスは、一度作成すれば明示的に参照する必要はありません。ローが削除、更新、挿入された場合は、自動的にインデックスの更新が行われます。

インデックス・ヒント

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

インデックス・ヒントを指定するには、**FORCE INDEX** または **WITH (INDEX ())** 句をクエリに追加します。詳細については、『ASA SQL リファレンス・マニュアル』> 「FROM 句」を参照してください。

データベース・オブジェクト・プロパティの変更については、「データベース・オブジェクトのプロパティの設定」44 ページを参照してください。

クラスタード・インデックスの使用

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

クラスタード・インデックスを作成すると、インデックスのパフォーマンスをさらに向上させることができます。Adaptive Server Anywhere のクラスタード・インデックスには、テーブルのローが、対応するインデックス内での順序とほぼ同じ順序で格納されます。

クラスタード・インデックス機能を使用すると、連続した 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、各ページをメモリ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。

たとえば、クラスタード・インデックスに連続して現れる 2 つのローを選択する場合、取り出す 2 つのローは同じページに連続して現れ、メモリに読み込むページの数を半分に減らすことができます。

Adaptive Server Anywhere では、インデックスが概算でクラスタ化されます。サーバはキーの順序を保持しようとしますが、完全なクラスタは保証されません。また、データベースに挿入されるローが増加するにつれて、クラスタ化の機能は低下します。

次の文を使用して、テーブルごとに 1 つのクラスタード・インデックスを実装できます。

- CREATE TABLE 文
- ALTER TABLE 文
- CREATE INDEX 文
- DECLARE LOCAL TEMPORARY TABLE 文

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、インデックス・キーの順序でテーブルをアンロードできます。
- LOAD TABLE 文は、インデックス・キーの順序でローをテーブルに挿入します。
- INSERT 文は、新しいローを挿入するときに、プライマリ・キーの順序が隣接するローと同じテーブル・ページに挿入しようとします。
- REORGANIZE TABLE 文はクラスタード・インデックスに従ってローを再編成することによって、クラスタ化をリストアできます。クラスタ化が指定されていないテーブルでは、プライマリ・キーを使用してテーブルが順序付けられます。

オプティマイザは、テーブルのローがキーの順序で格納されていると想定して、インデックス・スキャンのコストを計算します。

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。

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

1. [インデックス] フォルダを開きます。
2. [ファイル] － [新規] － [インデックス] を選択します。

   [インデックス作成] ウィザードが表示されます。
3. インデックスに名前を付け、リストからテーブルを選択します。[次へ] をクリックします。
4. ウィザードの指示に従います。

   新しいインデックスが [インデックス] フォルダに表示されます。

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

1. DBA 権限を使用して、またはインデックスを設定するテーブルの所有者として、データベースに接続します。
2. CREATE INDEX 文を実行します。

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

例

サンプル・データベースで従業員の姓の検索速度を上げるために、次の文を使用してインデックス EmpNames を作成します。

```
CREATE INDEX EmpNames
 ON employee (emp_lname, emp_fname)
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「CREATE INDEX 文」と「パフォーマンスのモニタリングと改善」201 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。外部キーのインデックスでは、プライマリ・テーブルに対応するローがあるかどうかと、そのハッシュ値が一致するかどうかについても検証できます。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。

テーブルやデータベース全体の検証は、どの接続において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。そうしないと、実際にデータベースの破損がない場合でも、何らかのデータベースの破損を示す重大なエラーがレポートされる可能性があります。

❖ 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 権限を使用して、またはインデックスを設定するテーブルの所有者として、データベースに接続します。
2. 左ウィンドウ枠で [インデックス] フォルダを開きます。
3. 対象のインデックスを右クリックし、ポップアップ・メニューで [検証] を選択します。

❖ インデックスを検証するには、次の手順に従います (SQL の場合)。

1. DBA 権限を使用して、またはインデックスを設定するテーブルの所有者として、データベースに接続します。
2. VALIDATE INDEX 文を実行します。

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

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

例

インデックス EmployeeIndex 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 EmployeeIndex
```

インデックス EmployeeIndex を検証します。**-i** スイッチは、そのオブジェクト名をインデックスとして指定します。

```
dbvalid -i EmployeeIndex
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「VALIDATE INDEX 文」と『ASA データベース管理ガイド』> 「検証ユーティリティ」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 権限を使用して、またはインデックスを設定するテーブルの所有者として、データベースに接続します。
2. 左ウィンドウ枠で [インデックス] フォルダを開きます。
3. 対象のインデックスを右クリックし、ポップアップ・メニューで [削除] を選択します。

❖ 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 権限を使用して、またはインデックスと関連付けられた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、データベースからインデックスを削除します。

```
DROP INDEX EmpNames
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「DROP 文」を参照してください。

インデックス・コンサルタントの概要

適切に設定されたインデックスを選択すると、データベースのパフォーマンスに大きな違いが出ます。適切に設定されたインデックスを選択するタスクを手助けするために、Adaptive Server Anywhere にはインデックス・コンサルタントが含まれています。インデックス・コンサルタントは、単一のクエリまたは各種のデータベース要求 (「**負荷**」と呼ばれる) に対するインデックスの選択プロセスを支援します。インデックス・コンサルタントは、仮想インデックスの多くの異なるセットを作成します。これらの仮想インデックスが実際に存在するかのように、インデックス・コンサルタントは、セットごとにクエリやその他の要求を最適化します。次に、インデックス・コンサルタントは、これらの調査の結果を収集して各種の推奨内容にまとめます。

インデックス・コンサルタントが行う処理には、いくつかの段階があります。これらの段階を理解することで、このツールを最大限に生かすことができます。

1. インデックス・コンサルタントの起動

   インデックス・コンサルタントは、Interactive SQL または Sybase Central から実行できます。Interactive SQL からインデックス・コンサルタントにアクセスして、単一のクエリ用のインデックスを分析できます。Sybase Central からアクセスすると、負荷用のインデックスを分析できます。

   インデックス・コンサルタントの起動方法については、「インデックス・コンサルタントの起動」89 ページを参照してください。

2. 負荷の取得

   負荷とは、クエリまたはその他のデータ操作文のセットで、インデックス・コンサルタントは、負荷に対しパフォーマンスを最適化しようとします。目的に応じて、負荷を代表的な操作で構成さ

れるセットにしたり、一連の主なボトルネック操作のセットを識別したりできます。インデックス・コンサルタントは、1 つまたは複数の負荷を取得し、後で分析するために格納します。

詳細については、「負荷の知識」91 ページを参照してください。

3. 負荷の分析

インデックス・コンサルタントは、候補インデックスを生成し、パフォーマンスに対するそれらの効果を調べて、負荷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・セットごとに負荷内の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。

またインデックス・コンサルタントは、任意の負荷に対して、異なる設定を使用した複数の分析結果を格納できます。

インデックス・コンサルタントを使用した負荷の分析には、いくつかのオプションがあります。

- **クラスタード・インデックスの推奨** このオプションを選択すると、インデックス・コンサルタントは、クラスタード・インデックスと非クラスタード・インデックスの効果を分析します。

  一部の負荷に対しては、クラスタード・インデックスを適切に選択すると、非クラスタード・インデックスよりも大幅にパフォーマンスが改善されます。ただし、このためには REORGANIZE TABLE 文を使用してテーブルを再編成する必要があります。さらに、クラスタード・インデックスの効果を考慮すると分析に時間がかかります。

  クラスタード・インデックスの詳細については、「クラスタード・インデックスの使用」81 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** 各負荷に対して、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。しかしクエリ・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、Adaptive Server Anywhere のクエリ・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、負荷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行することも、保存して独自に確認、分析することもできます。

詳細については、「分析の知識」92 ページを参照してください。

4. 推奨内容の実装

   インデックス・コンサルタントにより提供された SQL スクリプトを実行して、その推奨内容を実装できます。その前に、推奨内容をデータベースに関して持っている知識に照らして評価することもできます。

Interactive SQL でのインデックス・コンサルタントの使用

Interactive SQL では、インデックス・コンサルタントを使用して一度に 1 つの文のみを分析できます。

詳細については、「推奨内容の評価」95 ページと「推奨内容の実装」96 ページを参照してください。

インデックス・コンサルタントの起動

インデックス・コンサルタントは、適切なインデックスを選択する手助けをします。次の方法でインデックス・コンサルタントを使用できます。

- **Interactive SQL** Interactive SQL からインデックス・コンサルタントを使用して、個別のクエリに対するインデックスの利益を分析できます。
- **Sybase Central** Sybase Central からインデックス・コンサルタントを使用して、負荷、すなわちデータベース要求のセットに対するインデックスの利益を分析できます。

❖ インデックス・コンサルタントを起動するには、次の手順に従います (Sybase Central の場合)。

1. Sybase Central の左ウィンドウ枠で [インデックス] フォルダを右クリックします。
2. ポップアップ・メニューで [インデックス・コンサルタント] を選択します。

❖ インデックス・コンサルタントを起動するには、次の手順に従います (Interactive SQL の場合)。

1. 分析対象のクエリを Interactive SQL の [SQL 文] ウィンドウ枠に入力します。
2. [ツール] － [インデックス・コンサルタント] を選択します。

インデックス・コンサルタントの終了

インデックス・コンサルタントは、分析のために負荷を取得している間、データベースのパフォーマンスに影響を与えます。このため、データベース・サーバのウィンドウには、インデックス・コンサルタントが負荷を取得している間、情報メッセージが表示されます。

通常、インデックス・コンサルタントは、Sybase Central のユーザ・インタフェースから終了します。場合によっては、インデックス・コンサルタントを別のマシンから終了する必要もあります。たとえば、インデックス・コンサルタントがうっかり実行したままになっていた場合などです。

負荷の取得を一時停止または終了するには、それぞれ sa_pause_workload_capture プロシージャと sa_stop_workload_capture プロシージャを呼び出します。

```
call sa_pause_workload_capture;
 call sa_stop_workload_capture;
```

インデックス・コンサルタントを終了するには、sa_stop_index_consultant システム・ストアド・プロシージャを呼び出します。

```
call sa_stop_index_consultant
```

インデックス・コンサルタントの知識

データベースに最適な設定をするインデックスを選択することは、1 つの明確に定義された答えがある場合は、問題となりません。インデックス・コンサルタントは、いくつかのインデックス・スキームの結果を包括的に調査し、インデックスに付随するコストと利益を慎重に分析して、推奨を行います。しかし、問題の性質上、インデックス・コンサルタントの推奨内容が常に最適の設定をするインデックスであるとは保証されません。

この項では、インデックス・コンサルタントの仕組みの一部を説明します。この項は、インデックス・コンサルタントの推奨内容をより適切に判断するための情報を提供することを目的としています。

負荷の知識

インデックス・コンサルタントのオペレーションにおける最初の手順は、データ操作とクエリ操作の負荷を取得することです。これは、1 つのアプリケーションまたは一連のアプリケーションがデータベースに対して操作を実行するときの情報を取得することで行われます。

負荷は、取得手順で実行されたすべての SELECT、INSERT、UPDATE、DELETE 文で構成されます。これらの操作の内容は、負荷の一部として取得されません。たとえば、ストアド・プロシージャ、トリガ、またはイベント・ハンドラの一部として実行されたクエリは負荷の一部として取得されますが、周囲のコードは取得されません。このため、次に示す結果がもたらされます。

- パラメータは取得されない。

  パラメータで表された SQL 文は、固有のリテラル値を文定義に置き換えて格納されます。たとえば、WHERE 句にパラメータを含むクエリが、パラメータに異なる値を代入して 15 回実行される場合、インデックス・コンサルタントは、15 個のクエリを別個に取得します。それぞれのクエリには、パラメータの代わりに別個の定数が含まれます。

- テンポラリ・テーブルに対するクエリは分析されない。

  テンポラリ・テーブルに対するクエリは負荷の一部として収集されますが、テンポラリ・テーブル内のデータは取得されません。したがって、そのクエリは、分析段階で分析されません。インデックス・コンサルタントは、テンポラリ・テーブルについて適切なインデックスを推奨できません。

- 接続状態はすべて取得されるわけではない。

  接続状態とは、クエリが実行されるときに有効なデータベース・オプションのセットなどを指します。多くの場合、OPTIMIZATION_GOAL オプションは、オプティマイザが選択する実行プランに大きな影響を与えます。このオプションは、クエリとともに格納されます。しかし、その他のオプションは格納されません。

- サーバの状態は取得されない。

サーバの状態とは、データがキャッシュにあるか、またはディスクからフェッチされなければならないか、といったことを指します。また、同時に実行される操作の影響も指します。

警告
負荷の取得手順中にデータベース・スキーマを変更しないでください。また、取得手順と分析手順の間にデータベース・スキーマを変更しないでください。変更を行うと、インデックス・コンサルタントの推奨内容が正しくなくなります。

分析の知識

分析手順は、次の一連のオペレーションを繰り返し実行することで実行されます。

1. 仮想インデックスの候補セットを作成する。

   仮想インデックスには、実際のデータは含まれず、実際のクエリの実行には使用できません。仮想インデックスは、オプティマイザがクエリに適切な実行プランを選択するときに使用します。インデックス・コンサルタントは、仮想インデックスのセット案を多数生成します。

2. この仮想インデックスの候補セットに対して、負荷の操作を最適化する。

   インデックス・コンサルタントは、Adaptive Server Anywhere のクエリ・オプティマイザが選択した、負荷内の各文に対するプランを取り出します。オプティマイザは、各文の候補セットから適用可能な仮想インデックスを検討します。しかし、文は実行されません。インデックス・コンサルタントは、ユーザのデータを修正しません。

   各クエリについて、クエリ・オプティマイザは、数多くある実行プラン案の実行コストを比較します。実行コストは、内部的なコスト・モデルに基づいて推定されます。どのテーブルにインデックスを使用してアクセスし、どのテーブルにはインデックスを使用せずにアクセスするかは、各実行プランのコストに影響する重要な選択の 1 つです。各仮想インデックスのセットにより、新しい実行プラン案が可能となり、別のプランがクローズされます。

コスト・モデルは、データベース・サーバの状態に依存します。特に、インデックス・コンサルタントそのものはディスクからデータを読み込んだり、データベースに対して操作を実行しませんが、コスト・モデルは、どのデータがキャッシュにあり、どのデータをディスクにアクセスして得なければならないかに依存します。したがって、インデックス・コンサルタントを実行して、特定の負荷について分析する場合、実行するごとに異なる推奨内容が生成されることがあります。たとえば、起動されたばかりのデータベース・サーバでインデックス・コンサルタントを実行する場合 (したがってキャッシュにデータがない場合) と、しばらく稼働しているデータベース・サーバで実行する場合では、推奨内容が異なることがあります。

コスト・モデルの詳細については、「オプティマイザの推定とヒストグラム」498 ページを参照してください。

推奨内容の知識

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、負荷と分析の概要が示されます。負荷内のクエリ数、推奨インデックス数、推奨インデックスに必要なページ数、推奨インデックスがもたらすと考えられる利益などの情報が含まれます。利益値は、内部的なコスト単位に基づいて測定されます。

[推奨インデックス] タブ

[推奨インデックス] タブには、各推奨インデックスに関するデータが含まれます。次に示すような情報が提供されます。

- **[クラスタード]** 各テーブルは、最大 1 つのクラスタード・インデックスを持つことができます。場合によっては、クラスタード・インデックスは、非クラスタード・インデックスと比べて大きな利益をもたらします。

  クラスタード・インデックスの詳細については、「クラスタード・インデックスの使用」81 ページを参照してください。

- **[ページ]** インデックスを作成することを選択した場合に、インデックスを保持するために必要な推定データベース・ページ数。

データベース・ページ・サイズの詳細については、『ASA データベース管理ガイド』> 「初期化ユーティリティ」を参照してください。

- **[相対利益]** 指定されたインデックスを作成した場合の、総合的な推定利益を示す、1 から 10 までの数字。数字が大きいほど、推定される利益が大きいことを示します。

  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、「推奨内容の評価」95 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、負荷内のすべての操作について合計され、内部的なコスト単位に基づいて測定されます。

  コスト・モデルの詳細については、「オプティマイザの推定とヒストグラム」498 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、負荷内の個別の要求に対する推奨内容の効果の内訳が示されます。ここには、推奨インデックスを適用する前と後の推定コストや、クエリが使用した仮想インデックスなどの情報が含まれます。ボタンを使用すると、要求に対して最適と判断された実行プランを表示できます。

[更新] タブ

[更新] タブには、推奨内容の効果の内訳が示されます。

[未使用インデックス] タブ

[未使用インデックス] タブには、データベースにすでに存在し、負荷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。

推奨内容の評価

インデックス・コンサルタントの推奨内容は、可能なオプションの包括的な調査に基づいていますが、運用データベースに実装する前に推奨内容を評価することをおすすめします。たとえば、次のような確認を行います。

- **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、負荷に含まれていなかったほか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、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、分析中のサーバの状態は、基本的に静的です (ただし、ほか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。

推奨内容の実装

インデックス・コンサルタントは、推奨内容を SQL スクリプトとして提供します。インデックス・コンサルタントから推奨内容を実装することも、スクリプトを保存して後で使用することもできます。

提案されたインデックスの名前は、分析の名前から生成されます。データベースにインデックスを作成する前に、名前を変更できます。

Interactive SQL からコマンド・ファイルを実行する方法については、「SQL コマンド・ファイルの使用」741 ページを参照してください。

システム・テーブルのインデックス

データベースのインデックスについての情報は、すべてシステム・テーブルの SYS.SYSINDEX と SYS.SYSIXCOL に入っています。システム・ビュー SYS.SYSINDEXES は、この情報を見やすい形で表示します。これらのテーブルは、Sybase Central または Interactive SQL を使用してブラウズできます。

テンポラリ・テーブルの編集

テンポラリ・テーブルは、ローカル・テンポラリ・テーブルもグローバル・テンポラリ・テーブルも、データの一時的な保管という同じ目的を果たします。ただし、両者の相違とそれぞれの利点は、各テーブルの存在期間にあります。

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

詳細については、『ASA SQL リファレンス・マニュアル』>「DECLARE LOCAL TEMPORARY TABLE 文」を参照してください。

「**グローバル・テンポラリ**」テーブルの定義は永久的にデータベース内に残りますが、ローは指定された接続内にだけ存在します。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、テーブル定義はデータベース内に残るので、次にデータベースを開くときにアクセスできます。

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。ほかの DB 領域と同様に、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。

詳細については、『ASA SQL リファレンス・マニュアル』>「CREATE TABLE 文」を参照してください。

Sybase Central におけるデータベース・オブジェクトのコピー

Sybase Central では、既存のデータベース・オブジェクトをコピーし、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。

オブジェクトをコピーするには、Sybase Central の左ウィンドウ枠でオブジェクトを選択し、該当するフォルダまたはコンテナにドラッグします。または、オブジェクトをコピーして、該当するフォルダまたはコンテナにペーストします。新しいオブジェクトが作成され、元のオブジェクトのコードが新しいオブジェクトにコピーされます。同じデータベース内でオブジェクトをコピーする場合は、新しいオブジェクトの名前を変更してください。

データベース内の別のオブジェクトにオブジェクトをペーストすることもできます。たとえば、テーブルをユーザにペーストすると、このテーブルに対するパーミッションがこのユーザに与えられます。

別のアプリケーションで使用するためにデータベース・オブジェクトのコードをコピーする

Sybase Central では、次にリストされているオブジェクトのいずれかをコピーすると、そのオブジェクトの SQL がクリップボードにコピーされるため、Interactive SQL やテキスト・エディタなどのほかのアプリケーションにペーストできます。たとえば、Sybase Central でインデックスをコピーし、テキスト・エディタにペーストすると、そのインデックスの CREATE INDEX 文が表示されます。

- アーティクル
- 検査制約
- カラム ([コピー] ボタンのみ)
- ドメイン
- イベント
- 外部ログイン
- 外部キー
- インデックス

- メッセージ型
- プロシージャと関数
- パブリケーション
- リモート・サーバ
- システム・トリガ
- テーブル
- トリガ
- Ultra Light プロジェクト
- Ultra Light 文
- 一意性制約
- ユーザ
- ビュー
- Web サービス

第 3 章

データ整合性の確保

この章の内容

整合性確保のための制約をデータベースに組み入れると、データベースの信頼性は確実に上がります。この章では、データの有効性と信頼性を確保するための Adaptive Server Anywhere の機能について説明します。

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することも、整合性の確保に役立ちます。

この章では、SQL 文の CREATE TABLE 文と ALTER TABLE 文を使って説明します。この 2 つの文の基本的な使い方については、「データベース・オブジェクトの使用」35 ページを参照してください。

データ整合性の概要

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

Adaptive Server 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」811 ページを参照してください。

データが有効でない場合

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

正しくない情報

- オペレータが売り上げトランザクションの日付を間違って入力した。
- オペレータが一桁間違えて入力し、社員の給料が 10 分の 1 になった。

データの重複

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

失われた外部キー関係

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

データベース内の整合性制約

データベースのデータの有効性を確保するには、データが有効かどうかを判断するための検査と、データが従う規則 (ビジネス・ルール) が必要です。この検査と規則を合わせたものが「**制約**」です。

データベースの整合性制約の構築

データベース自体に組み込まれた制約は、クライアント・アプリケーション内に組み込まれた制約や、データベースのユーザに説明された制約よりも高い信頼性を備えています。データベースに組み込まれた制約はデータベースの定義の一部であり、すべてのアプリケーションに対して常に適用されるからです。データベース内に制約が設定されると、それ以降に発生するすべてのデータベースとの対話に対してその制約が適用されます。

これに対して、クライアント・アプリケーションに組み込まれた制約は、アプリケーションが変更されるたびに無効となる可能性があります。また、複数のアプリケーションに組み込んだり、1 つのアプリケーションの複数箇所に組み込んだりする必要があります。

データベースのデータが変わる場合

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

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

データ整合性の支援策

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

デフォルト値

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

- すべてのユーザ、またはクライアント・アプリケーションによるトランザクション日を記録するデフォルト値を、その日の日付にする。
- 新しいローが追加された場合を除いて、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」107 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。

- 非 NULL 値制約を適用すると、カラムに NULL 値が入力されるのを防ぐことができる。
- 検査制約をカラムに適用すると、カラム内のデータが常に一定の条件を満たすようにできる。たとえば、給与カラムに上限と下限を設定して入力エラーを防げます。
- 検査制約を使って、それぞれのカラム値の差を検査できる。たとえば、図書館データベースで、date_returned (返却日) エントリが date_borrowed (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。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」811 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」114 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」124 ページを参照してください。プライマリ・キーと外部キーの関係の正しい設計に関する詳細については、「データベースの設計」3 ページを参照してください。

高度な整合性規則のためのトリガ

データ整合性の確保にはトリガも使用できます。「**トリガ**」はデータベースに格納されたストアド・プロシージャの一種で、特定のテーブル情報が修正されると自動的に実行されます。トリガはデータベース管理者や開発者にとって、データの信頼性を維持するための強力な手段です。

トリガの詳細については、「プロシージャ、トリガ、バッチの使用」811 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**　データベースの作成時の整合性制約を実装します。
- **ALTER TABLE 文**　既存のデータベースに対して、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、『ASA SQL リファレンス・マニュアル』> 「 SQL 文」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。
- 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。
- 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、**sales_order** テーブルの **id** カラムに条件を追加して、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。

```
ALTER TABLE sales_order
 MODIFY id DEFAULT AUTOINCREMENT
```

他のデフォルト値も同様に指定できます。詳細については、『ASA SQL リファレンス・マニュアル』> 「ALTER TABLE 文」と 『ASA SQL リファレンス・マニュアル』> 「CREATE TABLE 文」を参照してください。

カラム・デフォルトの修正と削除

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、**order_date** というカラムのデフォルト値を、CURRENT DATE に変更します。

```
ALTER TABLE sales_order
 MODIFY order_date 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、**order_date** カラムからデフォルトを削除します。

```
ALTER TABLE sales_order
 MODIFY order_date DEFAULT NULL
```

Sybase Central でカラム・デフォルトを設定する

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

❖ カラムのプロパティ・シートを表示するには、次の手順に従います。

1. データベースに接続します。
2. データベースの [テーブル] フォルダを開きます。

3　変更するカラムのあるテーブルをダブルクリックします。

4　右ウィンドウ枠で、[カラム] タブをクリックします。

5　対象のカラムを選択します。

6　[ファイル] ー [プロパティ] を選択します。

カラムのプロパティ・シートが表示されます。

現在の日付／時刻デフォルト

データ型が DATE、TIME、TIMESTAMP のカラムには、現在の日付、現在の時刻、現在のタイムスタンプをデフォルトとして使用できます。指定するデフォルト値は、カラムのデータ型に一致させてください。

現在の日付をデフォルトにすると便利な例

次のような場合、現在の日付をデフォルト値として設定すると便利です。

- 顧客データベースで、電話があった日付を記録する。
- 売り上げ入力データベースで、注文の日付を記録する。
- 図書館データベースで、会員が本を借りた日を記録する。

現在のタイムスタンプ

現在のタイムスタンプは、現在の日付と同じように使えるデフォルト値で、さらに細かい情報を記録できます。たとえば、コンタクト管理アプリケーションのユーザは、一日に何度も同一の顧客と連絡を取ることがあります。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。

タイムスタンプ、時刻、日付の詳細については、『ASA SQL リファレンス・マニュアル』> 「SQL データ型」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。

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、それまでのカラムの最大値に 1 を加えた値を割り当てます。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」124 ページを参照)。たとえば、インデックスの最初のカラムにオートインクリメントを適用すると効果的です。これは、サーバが最大値を検索する場合に、インデックスまたはキー定義を使用できるためです。

これ以外のケースにも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整数 (4 バイト) で保持されている場合に、2^{31} － 1 より大きい値または大きい 2 倍以上などの数値が入力されると、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、『ASA SQL リファレンス・マニュアル』> 「@@identity グローバル変数」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。

Ultra Light アプリケーションでは、テーブルが作成された時点でのオートインクリメントの初期値は 0 ではありません。カラムに符号付データ型が指定されている場合は、オートインクリメントによって負の値が生成されます。

オートインクリメントを適用するカラムを符号なしデータ型に設定し、負の値が生成されないようにしてください。

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」611 ページを参照してください。

NEWID デフォルト

UUID (ユニバーサル・ユニーク識別子) を使用して、テーブルのローをユニークに識別できます。UUID は、GUID (グローバル・ユニーク識別子) 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。

詳細については、『ASA SQL リファレンス・マニュアル』> 「NEWID 関数 [その他]」または 『ASA SQL リファレンス・マニュアル』> 「UNIQUEIDENTIFIER データ型 [バイナリ]」を参照してください。

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のうち、データベースの値が不正確でも問題がないときに使います。

NULL 値の詳細については、『ASA SQL リファレンス・マニュアル』> 「NULL 値」を参照してください。

文字列と数値デフォルト

カラムが文字列または数値のデータ型であれば、特定の文字列または数値をデフォルトに指定できます。指定するデフォルト値は、カラムのデータ型に変換可能なものにしてください。

文字列と数値デフォルトは、カラムに同じデータを入力することが多いときに便利です。たとえば、ある会社に本社 **city_1** と小規模の事業所 **city_2** の 2 つのオフィスがある場合、所在地カラムのデフォルトを **city_1** にしておけば入力が簡単になります。

定数式デフォルト

定数式もデフォルト値として使用できます。ただし、データベース・オブジェクトを参照していない場合にかぎります。定数式では、「*本日より 15 日後*」といったエントリを、デフォルト値としてカラムに設定できます。この場合は次のように入力します。

```
... DEFAULT ( dateadd( day, 15, getdate() ) )
```

テーブル制約とカラム制約の使い方

基本的なテーブル構造 (カラムの番号、名前、データ型、テーブルの名前とロケーション) 以外にも、CREATE TABLE 文と ALTER TABLE 文を使って、データの整合性を保つためのさまざまなテーブル属性値を指定できます。制約を使って、カラムに含まれる値や、異なるカラムに含まれるそれぞれの値の関係に制限を課すことができます。制約は、テーブル全体、または個別のカラムに対して適用できます。

警告
テーブルを修正する際には、他のデータベース・ユーザの処理が妨げられるおそれがあります。他のユーザがデータベースに接続していても ALTER TABLE 文を使用できますが、目的のテーブルが使用されていると ALTER TABLE 文は実行できません。また、大規模なテーブルの修正には時間がかかり、その間そのテーブルを使用できません。

この項では、制約を使用してテーブルに正しい値が確実に入力されるようにする方法について説明します。

カラムに対する検査制約の使い方

カラムの値がある一定の基準を満たすようにするため、検査条件を使用します。これらの条件は、非常識なデータの入力を防ぐ簡単なものであったり、企業のポリシーや内部規定などが反映される厳密なものであったりします。

個々のカラムに対する検査条件は、カラム内の値を一定の範囲に収める必要がある場合に便利です。

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 customer
 MODIFY phone
 CHECK ( phone LIKE '(___) ___-____' )
```

例 2

- エントリに入力されるデータが、あらかじめ決められたいくつかの値のいずれかになるように設定できます。次の例では、**city** カラムにある決まった値 (たとえばその会社の事業所の所在地) 以外を入力できないように制約しています。

```
ALTER TABLE office
 MODIFY 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 employee テーブルの従業員の入社日 **start_date** を必ず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 employee
 MODIFY start_date
 CHECK ( start_date 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。

カラムの検査は、FALSE が返された場合にのみエラーになります。UNKNOWN が返されても、その変更は実行されます。

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。この方法のほうが柔軟性が高いため、検査制約に名前を付けるか、できる限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、図書館データベースで、**date_borrowed** (貸出日) が必ず **date_returned** (返却日) 以前の日付になるように設定します。

```
ALTER TABLE loan
 ADD CONSTRAINT valid_date CHECK(date_returned >=
date_borrowed)
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「ALTER TABLE 文」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。そのデータ型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。

posint データ型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。たとえば、次に示すユーザ定義データ型は正の整数しか受け付けません。@ 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。

```
CREATE DATATYPE posint INT
 CHECK ( @col > 0 )
```

ALTER TABLE 文と DELETE CHECK 句を組み合せると、ドメインから継承されたものを含め、テーブル定義からすべての検査制約を削除できます。

ドメインでカラムが定義されたあとに、ドメイン定義の制約が変更された場合は、そのカラムに変更は適用*されません*。カラムは作成時にドメインの制約を継承しますが、それ以降は、両者の間に関係はありません。

ドメインの詳細については、『ASA SQL リファレンス・マニュアル』> 「ドメイン」を参照してください。

Sybase Central でテーブル制約およびカラム制約を編集する

Sybase Central では、カラム制約の追加、変更、削除をテーブルやカラムのプロパティ・シートの [制約] タブで行います。

❖ 制約を管理するには、次の手順に従います。

1. [テーブル] フォルダを開きます。
2. 右ウィンドウ枠で、変更するテーブルをダブルクリックします。
3. 右ウィンドウ枠には、一意性制約のタブと検査制約のタブが別々にあります。
4. 制約に適切な変更を加えます。たとえば、テーブル制約またはカラム制約を追加する場合は、[検査制約] タブをクリックして、[ファイル] － [新規] － [テーブル検査制約]、または [ファイル] － [新規] － [カラム検査制約] を選びます。

検査制約の修正と削除

テーブル上の既存の検査制約を変更するには、いくつかの方法があります。

- テーブルまたはカラムに新しい検査制約を追加できます。
- カラムの検査制約を NULL に設定すると削除できます。次に、**customer** 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 customer
 MODIFY 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、**customer** 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 customer
 MODIFY phone
 CHECK ( phone LIKE '___-___-____' )
```

- テーブルに定義された検査制約を修正できます。
  - 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、制約を個別に修正できます。

- 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、修正、削除できます。詳細については、「Sybase Central でテーブル制約およびカラム制約を編集する」116 ページを参照してください。

テーブルからカラムを削除しても、そのカラムと関連付けられていた検査制約はテーブル制約から削除されません。制約を削除しないと、テーブルにデータを挿入する場合や、単に問い合わせるだけでも「カラムが見つかりません」というエラー・メッセージが生成されます。

テーブルの検査制約は FALSE が返された場合にのみエラーとなります。戻り値が UNKNOWN の場合は変更できます。

ドメインの使い方

「ドメイン」とはユーザ定義データ型のことです。これを他の属性と一緒に使用して、値の許容範囲を制限したりデフォルト値を設定したりできます。ドメインを使うと既存のデータ型を拡張できます。値の許容範囲の制限には、通常検査制約が使われます。さらに、ドメインではデフォルト値を設定できます。値は NULL であってもそうでなくてもかまいません。

また、さまざまな目的のために独自のドメインを定義できます。

- エラーの多くは、不適当な値の入力を制限することで防げます。ドメインに設定した制約で、値を一定の範囲またはフォーマットに保持させたいすべてのカラムと変数に、その範囲内またはフォーマットの値だけを保持させることができます。たとえば、あるデータ型によって、データベースに入力されるクレジット・カード番号に確実に正しい桁数が入るようにできます。
- ドメインを使用すると、アプリケーションやデータベース構造が非常にわかりやすくなります。
- ドメインは非常に便利です。たとえば、テーブルの識別子をすべて正の整数にし、デフォルト値としてオートインクリメントにするとします。テーブルを新しく作成するたびにこうした制約を設定するよりも、新しいドメインを定義して識別子がそのドメイン型の値以外をとらないように設定するほうが、作業が少なくて済みます。

ドメインの詳細については、『ASA SQL リファレンス・マニュアル』>「SQL データ型」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 左ウィンドウ枠で [ドメイン] フォルダを選びます。

2 [ファイル] － [新規] － [ドメイン] を選びます。

[ドメイン作成] ウィザードが表示されます。

3 ウィザードの指示に従います。

Sybase Central では、すべてのドメインは [ドメイン] フォルダに表示されます。

❖ **ドメインをカラムに割り当てるには、次の手順に従います (Sybase Central の場合)。**

1 右ウィンドウ枠で対象のテーブルの [カラム] タブをクリックします。

2 対象のカラムのデータ型カラムで、次のいずれかを実行します。

- ドロップダウン・リストからドメインを選ぶ。または
- ドロップダウン・リストの横のボタンをクリックして、プロパティ・シートにあるドメインを選ぶ。

ドメインの作成 (SQL の場合)

CREATE DOMAIN 文は、ドメインの作成と定義に使用できます。

❖ **新しいドメインを作成するには、次の手順に従います (SQL の場合)。**

1 データベースに接続します。

2 CREATE DOMAIN 文を実行します。

例 1：簡単なドメイン

氏名と住所をそれぞれカラムに入力するデータベースがあるとします。たとえば、次のようにドメインを定義してください。

```
CREATE DOMAIN persons_name CHAR(30)
 CREATE DOMAIN street_address CHAR(35)
```

こうして定義されたドメインは、既存のデータ型と同様にいくらでも使用できます。たとえば、次のようにしてテーブルを定義できます。

```
CREATE TABLE customer (
    id INT  DEFAULT AUTOINCREMENT  PRIMARY KEY
    name persons_name
    address 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 INT
 DEFAULT AUTOINCREMENT
 CHECK ( @col > 0 )
```

この検査制約には、変数 @col が使用されます。この定義を使用して、上記の customer テーブルの定義を書き換えることができます。

```
CREATE TABLE customer (
    id identifier PRIMARY KEY
    name persons_name
    address street_address
 )
```

例 3：組み込みドメイン

Adaptive Server Anywhere には、定義済みドメインがいくつかあります。このような定義済みドメインは、自作のドメインと同様に自由に使用できます。たとえば、次のような通貨データ・ドメインがすでに定義されています。

```
CREATE DOMAIN MONEY NUMERIC(19,4)
 NULL
```

詳細については、『ASA SQL リファレンス・マニュアル』>「CREATE DOMAIN 文」を参照してください。

ドメインの削除

Sybase Central または DROP DOMAIN 文を使用すると、ドメインを削除できます。

削除できるのは、ユーザ DBA 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 1 つでも使用されているドメインは削除できないため、それを使用しているカラムや変数をすべて削除してから、ドメインの削除を行ってください。

❖ ドメインを削除するには、次の手順に従います (Sybase Central の場合)。

1. [ドメイン] フォルダを開きます。
2. 目的のドメインを右クリックし、ポップアップ・メニューから [削除] を選択します。

❖ ドメインを削除するには、次の手順に従います (SQL の場合)。

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

例

次の文は、customer_name ドメインを削除します。

```
DROP DOMAIN customer_name
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「DROP 文」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、パーソナル・サーバはデータベース内の情報を識別し、各テーブル内のすべてのローは (データベース構造に定義されている) テーブル間の関係を適切に維持できます。

エンティティ整合性の確保

ローが挿入または更新されるたびに、データベース・サーバはそのテーブルのプライマリ・キーの有効性、つまりテーブルの各ローがプライマリ・キーを使ってユニークに識別できることを保証します。

例 1

サンプル・データベースの **employee** 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

サンプル・データベースの **sales_order_items** 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、**line_id** カラムは製品に対応する行を識別します。**id** カラムと **line_id** 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。

こうしたエラー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。

アプリケーション開発システムやデータベース設計ツールには、データベース・テーブルを作成したり修正したりできるものがあります。このようなシステムを使用している場合、CREATE TABLE 文や ALTER TABLE 文を明示的に入力する必要はありません。与えられた情報からアプリケーションが自動的に作成します。

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」58 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、『ASA SQL リファレンス・マニュアル』> 「CREATE TABLE 文」を参照してください。テーブル構造の変更については、『ASA SQL リファレンス・マニュアル』> 「ALTER TABLE 文」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

サンプル・データベースには、employee テーブルと department テーブルが含まれています。employee テーブルのプライマリ・キーは employee ID、また department テーブルのプライマリ・キーは department ID です。employee 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。employee テーブルのそれぞれの department ID が、department 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。employee 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 department 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 department テーブルのカラムを参照できると、department 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。employee 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」58 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。

Adaptive Server 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。

Adaptive Server 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**　参照先のプライマリ・キーの値が変更されそうになると、エラーを発生させて変更を許可しません。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。このような場合に次の文を実行すると、「テーブル 'department' のプライマリ・キーは他のテーブルから参照されています。」というエラーが発生します。

```
DELETE FROM department
 WHERE dept_id = 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。
- 参照整合性に違反するローを探索条件にして SELECT 文を再実行する。
- **dept_id** ローを **department** 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。

システム・テーブルの整合性ルール

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。

システム・テーブル	説明
SYS.SYSTABLE	SYS.SYSTABLE の **view_def** カラムには検査制約が格納される。ビューの場合、**view_def** にはそのビューを作成した CREATE VIEW コマンドが格納される。テーブルがベース・テーブルであるかビューであるかは、**table_type** カラムに BASE または VIEW のどちらの値が入っているかで判別する。
SYS.SYSTRIGGER	SYS.SYSTRIGGER には参照整合性アクションが格納される。 **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 文字が格納される。 **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 文字が格納される。格納される文字は、挿入、削除 (A)、挿入、更新 (B)、更新 (C)、削除 (D)、削除、更新 (E)、挿入 (I)、更新 (U)、挿入、削除、更新 (M)。 **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る。
SYS.SYSFOREIGNKEYS	このビューは、SYS.SYSFOREIGNKEY テーブルと SYS.SYSFKCOL テーブルからの外部キー情報を、より見やすい形で表示する。
SYS.SYSCOLUMNS	このビューはテーブル SYS.SYSCOLUMN からの情報を、より見やすい形で表示する。これにはカラムのデフォルト設定やプライマリ・キーの情報などが含まれる。

各システム・テーブルの内容の詳細については、『ASA SQL リファレンス・マニュアル』> 「システム・テーブル」を参照してください。これらのテーブルとビューは、Sybase Central または Interactive SQL を使って見ることができます。

第 4 章

トランザクションと独立性レベル

この章の内容

SQL 文はトランザクションにグループ分けできます。トランザクションは、すべての文が実行されるか、どの文も実行されないというプロパティを持ちます。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。

この章では、トランザクションの定義と、アプリケーションでそれらを使用する方法について説明します。また、Adaptive Server Anywhere で独立性レベルを設定し、同時に実行されるトランザクション間の干渉を制限する方法も示します。

トランザクションの概要

データの整合性を確保するには、データベースの情報が「**一貫している**」状態を識別できることが重要です。一貫性の概念は例によってよく理解することができます。

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仮定します。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。

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、Adaptive Server 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、『ASA データベース管理ガイド』> 「バックアップとデータ・リカバリ」を参照してください。

データベースの同時使用の詳細については、「同時性の概要」136 ページを参照してください。

トランザクションの使用

Adaptive Server 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開始または終了するコマンドや動作を理解すると、これらの機能を最大限に利用できるようになります。

トランザクションの開始

トランザクションは次のいずれかのイベントで開始します。

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

トランザクションの完了

トランザクションは次のいずれかのイベントで完了します。

- データベースの変更を確定する COMMIT 文
- トランザクションで行われたすべての変更を取り消す ROLLBACK 文
- オートコミットが実行される文。データ定義コマンドの ALTER、CREATE、COMMENT、DROP はすべて自動的にコミットを実行します。
- データベースへの接続を解除すると、暗黙的なロールバックが実行されます。
- ODBC と JDBC には各文の後で COMMIT 文を実行するオートコミットの設定があります。デフォルトでは、ODBC と JDBC のオートコミットの設定は ON にする必要があり、各文は単一のトランザクションとして処理されます。トランザクション設計機能を活用したい場合は、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、『ASA プログラミング・ガイド』> 「オートコミットまたは手動コミット・モードの設定」を参照してください。

- データベース・オプション CHAINED 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、『ASA プログラミング・ガイド』> 「オートコミットまたは手動コミット・モードの設定」と 『ASA データベース管理ガイド』> 「CHAINED オプション [互換性]」を参照してください。

Interactive SQL のオプション

Interactive SQL により、アプリケーションからいつどのようにトランザクションを終了するかを制御できます。

- AUTO_COMMIT オプションを ON に設定すると、Interactive SQL は、文が正常に実行された場合はその後に結果を自動的にコミットし、失敗した場合は ROLLBACK を自動的に実行する。

- 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する。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。

Adaptive Server 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」589 ページを参照してください。

同時性の概要

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

例

デパートのデータベース・システムでは、多くの店員が同時に顧客アカウントを更新する必要があります。各店員は、顧客に対応するときにアカウントの状態を更新できなければなりません。データベースを使用している人が誰もいなくなるまで待つことはできません。

同時性に関する知識の必要な方

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

同時性に影響するトランザクション・サイズ

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

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

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

トランザクション内のセーブポイント

「**セーブポイント**」を使用して関連する文をグループ分けすることによって、トランザクション内の重要な状態を識別し、そこに戻ることができます。

SAVEPOINT 文は、トランザクションの中に中間ポイントを定義します。そのポイント以降の変更はすべて ROLLBACK TO SAVEPOINT 文を使ってキャンセルできます。RELEASE SAVEPOINT 文の実行後、またはトランザクションの終了後は、セーブポイントは使えなくなります。セーブポイントは、COMMIT には影響しないことに注意してください。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

4 つの独立性レベル

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

Adaptive Server Anywhere が提供する 4 種類の独立性レベルを次に示します。

独立性レベル	特性
0- コミットしない読み出し	◆ 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
1- コミットする読み出し	◆ 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
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
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

これらの独立性レベルは、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 (コミットしない読み出し) に設定されています。

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

独立性レベルはトランザクション間にのみあります。同じトランザクション内の複数のカーソルは相互に干渉する可能性があります。

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。(ほか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/92 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、「ダーティリードのチュートリアル」157 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、「繰り返し不可能読み出しのチュートリアル」160 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 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、最初とは異なるローのセットを取得します。

  - 幻ローの詳細については、「幻ローのチュートリアル」166 ページを参照してください。

この他にも一貫性が失われるケースがあります。上記の 3 つのケースは典型的で、トランザクション間のロック量を説明するのに便利なため、ISO SQL/92 標準でも選択されています。

独立性レベルとダーティ・リード、繰り返し不可能読み出し、幻ロー

それぞれの独立性レベルでは、Adaptive Server Anywhere で許可される矛盾の動作タイプが異なります。表中の x は動作を防止できることを表し、√ は動作が起こる可能性があることを示します。

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- コミットしない読み出し	√	√	√
1- コミットする読み出し	x	√	√
2- 繰り返し可能読み出し	x	x	√
3- 直列化可能	x	x	x

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。

ODBC では、4 つの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。これについては、「ValuePtr パラメータ」145 ページで説明しています。

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、ほか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

Adaptive Server 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、ほか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」811 ページを参照してください。

カーソルは、ほかのアプリケーションを介して Adaptive Server Anywhere を使っているときのみに使用します。詳細については、『ASA プログラミング・ガイド』> 「アプリケーションでの SQL の使用」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、『ASA プログラミング・ガイド』> 「Adaptive Server Anywhere のカーソル」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。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、『ASA SQL リファレンス・マニュアル』> 「FROM 句」の WITH テーブル・ヒントに関する項を参照してください。

独立性レベルはデータベース・オプションです。データベース・オプションは、SET OPTION 文を使用して変更します。たとえば次のコマンドは、現在のユーザの独立性レベルを最高レベルの 3 に設定します。

```
SET OPTION ISOLATION_LEVEL = 3
```

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、ほかのユーザやグループの独立性レベルも変更できます。

❖ 現在のユーザ ID 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、特殊グループ 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
```

❖ 現在のセッションのみに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
```

接続を解除すると、独立性レベルは以前の値に戻ります。

デフォルトの独立性レベル

データベースに接続すると、データベース・サーバは次のように最初の独立性レベルを決定します。

1. デフォルトの独立性レベルは、ユーザや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。最初にほかの設定が見つからない場合、Adaptive Server 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、『ASA データベース管理ガイド』> 「ユーザ ID とパーミッションの管理」を参照してください。

SET OPTION 文の構文の詳細については、『ASA SQL リファレンス・マニュアル』> 「SET OPTION 文」を参照してください。

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」146 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_TXT_SERIALIZABLE	3

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_TXT_SERIALIZABLE	8

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で、部分によって異なる独立性レベルを設定したい場合があります。Adaptive Server 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。

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、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、『ASA SQL リファレンス・マニュアル』>「FROM 句」の WITH テーブル・ヒントに関する項を参照してください。

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、ほか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、「幻ローのチュートリアル」166 ページを参照してください。

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、処理中のローがほか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。

Adaptive Server 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ほか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、ほか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。

「2 フェーズ・ロック」187 ページでは、デッドロックについて説明します。これは、2 つ以上のトランザクションがお互いをブロックし合った結果、どの処理も進むことができない状態をいいます。

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」149 ページを参照してください。

BLOCKING 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 BLOCKING の設定によって異なります。ローを修正したい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。

- BLOCKING 設定が 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。
- BLOCKING 設定が 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。

BLOCKING 設定が OFF の場合、文は待機せず終了し、行った部分的な変更はロールバックされます。この場合は、あとでもう一度トランザクションの実行を試みます。

ブロックは、独立性レベルが高くなると起こりやすくなります。ロックもチェックの回数も独立性とともに増えるからです。独立性レベルが高いと、通常は同時性が低下します。低下の度合いは、同時に実行するトランザクションによって異なります。

BLOCKING オプションの詳細については、『ASA データベース管理ガイド』> 「BLOCKING オプション [データベース]」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。データベースが 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。

Adaptive Server Anywhere は、最後にブロックされた文を自動的にロールバックして、デッドロックを解放します。ロールバックされたトランザクションにはデッドロックの種類を示すエラーを返します。

サーバが使用するデータベース・スレッドの数は、個々のデータベースの設定によって異なります。

データベース・スレッド数の設定の詳細については、『ASA データベース管理ガイド』> 「スレッド動作の制御」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、『ASA SQL リファレンス・マニュアル』> 「sa_conn_info システム・プロシージャ」を参照してください。

データベース・サーバは、LOG_DEADLOCKS オプションと sa_report_deadlocks システム・プロシージャを使用して詳細なデッドロック・レポートを提供します。LOG_DEADLOCKS オプションを ON にした場合、データベース・サーバは、ブロックされた接続に関する情報を内部バッファに記録します。

❖ デッドロック・レポートを使用するには、次の手順に従います。

1. LOG_DEADLOCKS オプションをオンにします。

```
SET OPTION PUBLIC.LOG_DEADLOCKS='ON'
```

データベース・サーバは、デッドロックに関する情報を内部バッファに記録します。

詳細については、『ASA データベース管理ガイド』>「LOG_DEADLOCKS オプション [データベース]」を参照してください。

2 sa_report_deadlocks を使用してデッドロック情報を検索します。

```
CALL sa_report_deadlocks()
```

詳細については、『ASA SQL リファレンス・マニュアル』>「sa_report_deadlocks システム・プロシージャ」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、妨げる可能性はずっと少なくなります。

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、または口座残高を確認するだけの場合でも、確実に正確な情報が返されるよう最大の注意を払う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、トランザクションがほかのトランザクションを待機するどうかは配慮しませんが、データベースのほかのユーザを妨げないためには正当性を犠牲にします。

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ほか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。

Adaptive Server Anywhere では、0、1、2、3 の 4 つの独立性レベルがあります。レベル 3 は完全な独立性を提供し、トランザクションはスケジュールが直列可能となる方法でインタリーブされます。

直列可能なスケジュール

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

この方法でトランザクションを同時に適用すると、前の項で説明した一貫性が失われる 3 つのケースを含む、さまざまな結果が生じる可能性があります。トランザクションが順次実行された場合、つまり 1 つ

のトランザクションが完全に終了した後、次のトランザクションが開始された場合、データベースの最終状態が達成されることがあります。トランザクションをある順序で順次実行した結果、データベースが実際のスケジュールと同じ状態になった場合、そのスケジュールは「**直列可能**」であるといいます。

Adaptive Server Anywhere が直列化を処理する方法の詳細については、「2 フェーズ・ロック」187 ページを参照してください。

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

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

直列可能とは、同時性が影響を与えないということ

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

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

直列不可能なスケジュールは矛盾を生じさせる

「典型的な矛盾のケース」140 ページで説明した矛盾のケースは、スケジュールが直列可能でないときに生じる典型的な問題です。それぞれのケースでは、すべてのトランザクションが順次実行された場合には起こり得ないはずの結果を生じるような方法で文がインタリーブされたため、一貫性が失われました。たとえば、あるトランザクションがあるローにデータを挿入または更新しているときに、別のトランザクションがそのローを選択できる場合、ダーティ・リードが発生します。

各種独立性レベルでの典型的なトランザクション

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

典型的なレベル 0 トランザクション

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

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

典型的なレベル 1 トランザクション

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

たとえば、カーソルを使用して在庫レベルを更新するトランザクションには、特に独立性レベル 1 が適しています。なぜなら、品目の入荷や販売時に在庫レベルに対して行う各調整が失われることがなく、このように頻繁に調整しても、ほかのトランザクションへの影響が最小限ですむからです。

典型的なレベル 2 トランザクション

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

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

典型的なレベル 3 トランザクション

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

独立性レベル 3 がいかに高度の整合性を提供するとしても、同時に多数のトランザクションの実行をサポートする必要がある大きなシステムでは使用を控える必要があります。Adaptive Server Anywhere はこのレベルではほか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ほかの多くのトランザクションの処理を妨げる可能性があります。

独立性レベル 2 と 3 での同時性の改善

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

直列可能なトランザクションは、ほか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、Adaptive Server Anywhere では、ロック数が増加してもパフォーマンスは向上するという最適化を実行できます。たとえば、検索基準を満たしているかどうかに関係なく、すべてのローの読み込みをロックするので、データベース・サーバはローの読み込みとロックの設定のオペレーションを自由に組み合わせることができます。

ロックの影響の削減

独立性レベル 3 でトランザクションを実行することは、できるかぎり避けてください。多数のロックを設定すると、他の同時に実行されるトランザクションの効率を妨げることになります。

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

特に、インデックスを追加すると、トランザクションの処理効率が格段に向上することがわかります。これは、トランザクションのうちの少なくとも 1 つに独立性レベル 3 を設定して実行する場合に特に顕著です。インデックスには、次の 2 つの利点があります。

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

Adaptive Server 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」175 ページを参照してください。

パフォーマンス、そしてコマンドを実行するために Adaptive Server Anywhere が情報にアクセスする方法の詳細については、「パフォーマンスのモニタリングと改善」201 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。

ダーティリードのチュートリアル

このチュートリアルでは、複数のトランザクションを同時に実行するときに発生する矛盾の 1 つを再現してみます。小規模の商品販売会社で、2 人の従業員が、会社のデータベースに同時にアクセスするとします。1 人は会社の Sales Manager で、もう 1 人は Accountant です。

Sales Manager は、会社で販売している T シャツの価格を 0.95 ドル上げようとしていますが、SQL 言語の構文に少し問題があります。それと同時に、Sales Manager が知らないうちに、Accountant が現在の在庫の小売り価格を計算し、次の管理ミーティングに提出するレポートに記載しようとしています。

ヒント：
データベースを次の方法で変更するときには、UPDATE の代わりに SELECT を使用して変更内容をテストしてから変更した方が賢明です。

この例では、実際にこの 2 人が同時にデモンストレーション・データベースを使用するケースを示します。

1. Interactive SQL を起動します。
2. Sales Manager としてサンプル・データベースに接続します。
   - [接続] ダイアログで、ODBC データ・ソース ASA 9.0 Sample を選択します。
   - [詳細] タブで、次の文字列を入力してウィンドウを識別しやすくします。

```
ConnectionName=Sales Manager
```

[OK] をクリックして接続します。

3. Interactive SQL をもう 1 つ起動します。

4. Accountant としてサンプル・データベースに接続します。

   - [接続] ダイアログで、ODBC データ・ソース ASA 9.0 Sample を選択します。
   - [詳細] タブで、次の文字列を入力してウィンドウを識別しやすくします。

     ```
     ConnectionName=Accountant
     ```

   - [OK] をクリックして接続します。

5. Sales Manager として、すべての T シャツの価格を 0.95 ドル上げます。

   - [Sales Manager] ウィンドウで次のコマンドを実行します。

     ```
     SELECT id, name, unit_price
      FROM product;
      UPDATE PRODUCT
      SET unit_price = unit_price + 95
      WHERE NAME = 'Tee Shirt'
     ```

結果は次の表のようになります。

id	name	unit_price
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 * unit_price )
   AS inventory
 FROM product
```

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」139 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 product
 SET unit_price = unit_price + 0.95
 WHERE NAME = 'Tee Shirt';
```

id	name	unit_price
300	Tee Shirt	9.95
301	Tee Shirt	14.95
302	Tee Shirt	14.95
400	Baseball Cap	9.00

id	name	unit_price
…	…	…

8. Accountant は、計算した値に誤りがあったことに気づきません。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 * unit_price )
   AS inventory
 FROM product;
```

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。COMMIT 文を入力して変更を確定できますが、ここでは、ROLLBACK を入力して、マシンのデモンストレーション・データベースの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。

繰り返し不可能読み出しのチュートリアル

「ダーティリードのチュートリアル」157 ページの例では、矛盾のケースとして、まずダーティ・リードを取り上げました。その中では、Sales Manager が価格を更新している間に、Accountant が計算を実行してしまいました。Accountant は、Sales Manager が入力後に訂正を加えている過程の間違った情報を使用して計算しました。

次の例では、別のタイプの矛盾を説明します。これは、「繰り返し不可能読み出し」というものです。この例でも、同時にデモンストレーション・データベースを使用する同じ 2 人を使います。Sales Manager

はプラスチック・バイザーに新しい価格を設定するとします。Accountant は最近注文があったいくつかの品目の価格を調べるとします。

この例では、独立性レベル 0 ではなく 1 を使って、両方の接続を開始します。独立性レベル 0 は Adaptive Server Anywhere で提供されるデモンストレーション・データベースのデフォルト値です。独立性レベルを 1 に設定して、前のチュートリアルで示した一貫性が失われるケース (ダーティ・リード) を防止します。

1. Interactive SQL を起動します。
2. Sales Manager としてサンプル・データベースに接続します。
   - [接続] ダイアログで、ODBC データ・ソース ASA 9.0 Sample を選択します。
   - [詳細] タブで、次の文字列を入力してウィンドウが容易に識別できるようにします。

     ```
     ConnectionName=Sales Manager
     ```

   - [OK] をクリックして接続します。
3. Interactive SQL をもう 1 つ起動します。
4. Accountant としてサンプル・データベースに接続します。
   - [接続] ダイアログで、ODBC データ・ソース ASA 9.0 Sample を選択します。
   - [詳細] タブで、次の文字列を入力してウィンドウが容易に識別できるようにします。

     ```
     ConnectionName=Accountant
     ```

   - [OK] をクリックして接続します。
5. 次のコマンドを実行して、Accountant の接続の独立性レベルを 1 に設定します。

   ```
   SET TEMPORARY OPTION ISOLATION_LEVEL = 1
   ```

6. 次のコマンドを実行して、Sales Manager のウィンドウに独立性レベル 1 を設定します。

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 1
```

7. Accountant は、バイザーの価格をリストすることにします。Accountant として、次のコマンドを実行します。

```
SELECT id, name, unit_price FROM product
```

id	name	unit_price
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
…	…	…

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, unit_price FROM product
 WHERE name = 'Visor';
 UPDATE product
 SET unit_price = 5.95 WHERE id = 501;
 COMMIT;
```

id	name	unit_price
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。Sales Manager が新しい価格を入力し変更をコミットしたにもかかわらず、Accountant のウィンドウには旧価格が表示されています。

   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。ご自身で試してみ

てください。Accountant として、select コマンドを再実行してみましょう。今度は、Sales Manager の設定した販売価格が表示されることを確認します。

```
SELECT id, name, price
 FROM product
```

id	name	unit_price
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
…	…	…

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ほかの人が値を変更できます。ほかの人が行った変更によって矛盾が生じるのは、Accountant のトランザクションの途中で変更が行われるためです。そうしたイベントにより、スケジュールは直列不可能となります。

10. Accountant はこれに気づき、以降は価格参照中にほかからの変更を防ぐことにします。繰り返し不可能読み出しはレベル 2 で防止します。Accountant として次の処理を行ってください。

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 2;
 SELECT id, name, unit_price
 FROM product
```

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 product
 SET unit_price = 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。そのために、サーバは Accountant が読み込む product 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、product テーブルのプラスチック・バイザーのローに書き込みロックをかける必要があります。書き込みロックは排他ロックであるため、Accountant のトランザクションが読み込みロックを解放するまで待機しなければなりません。

12. Accountant が価格の閲覧を終了しました。データベースを誤って変更するのを防ぐために、ROLLBACK 文でトランザクションを完了します。

```
ROLLBACK
```

データベース・サーバがこの文を実行するとすぐに、Sales Manager のトランザクションも完了します。

id	name	unit_price
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、ほか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、ほかのトランザクションがそのローにどのようなロックをかけているかをチェックしません。チェックが不要のため、レベル 0 のトランザクション処理は著しく速くなります。この速度は一貫性を犠牲にして得られたものです。別のトランザクションが書き込みロックをかけているローを読むと、ダーティ・データを返す危険性があります。

レベル 1 では、トランザクションはローを読む前に書き込みロックがかかっていないかをチェックします。操作は 1 つ増えますが、このトランザクションでは読み込むデータはすべてコミット済みであることが保証されます。独立性レベルを 0 ではなく 1 に設定し、最初のチュートリアルを繰り返してみます。Sales Manager が T シャツの価格を更新するトランザクションの最中は、Accountant の計算は処理を進めることができず、不完全なままになることがわかります。

Accountant が独立性レベルを 2 に上げたとき、データベース・サーバは読み込みロックの使用を開始しました。それ以降、選択内容に適合するローごとに読み込みロックをかけました。

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。ルートの制御は、BLOCKING オプションを設定して行います。BLOCKING が ON になっていると、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。

BLOCKING オプションの詳細については、「BLOCKING オプション」148 ページを参照してください。

幻ローのチュートリアル

次のチュートリアルでも同じ例を使います。ここでは、Sales Manager が新しい部署を作成している最中に、Accountant が部署テーブルを見ています。その場合、幻ローが表示されます。

まだ実行していない場合は、前のチュートリアル「繰り返し不可能読み出しのチュートリアル」160 ページの手順 1 から 4 までを行ってください。Interactive SQL が 2 つ起動されます。

1. 次のコマンドを実行して、Sales Manager のウィンドウに独立性レベル 2 を設定します。

   ```
   SET TEMPORARY OPTION ISOLATION_LEVEL = 2;
   ```

2. 次のコマンドを実行して、Accountant のウィンドウに独立性レベル 2 を設定します。

   ```
   SET TEMPORARY OPTION ISOLATION_LEVEL = 2;
   ```

3. Accountant のウィンドウに次のコマンドを入力し、すべての部署をリストします。

   ```
   SELECT * FROM department
    ORDER BY dept_id;
   ```

dept_id	dept_name	dept_head_id
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
…	…	…

4. Sales Manager は外国市場に焦点を当てた新しい部署を設定することを決めます。emp_id 129 の Philip Chin を新しい部署の責任者にします。

```
INSERT INTO department
    (dept_id, dept_name, dept_head_id)
    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。

5. しかし Accountant は、新しい部署が作成されたことに気づきません。独立性レベル 2 で、データベース・サーバはローを変更しないようにロックをかけますが、ほか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 department
 ORDER BY dept_id;
```

dept_id	dept_name	dept_head_id
100	R & D	501
200	Sales	902
300	Finance	1293
500	Shipping	703
…	…	…

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。ほかのローはロックがかからないため、Sales Manager が新しいローを挿入するのを防止することはできません。

6. Accountant は今後そうしたことが起こらないように、現在のトランザクションの独立性レベルを 3 に上げることにします。次のコマンドを Accountant として入力します。

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 3
 SELECT *
 FROM department
 ORDER BY dept_id
```

7. Sales Manager は、大企業を対象にした営業活動を行う新たな部署を追加したいと思っています。Sales Manager のウィンドウで次のコマンドを実行します。

```
INSERT INTO department
    (dept_id, dept_name, dept_head_id)
 VALUES(700, 'Major Account Sales', 902)
```

Accountant のロックがコマンドをブロックするため、Sales Manager のウィンドウは実行中に一時停止します。ツールバーの [SQL 文の中断] ボタン (または [SQL] メニューの [中断]) をクリックしてこのエントリを一時停止します。

8. Adaptive Server Anywhere に添付されているデモンストレーション・データベースの変更を防止するには、新しい部署の挿入をロールバックする必要があります。Sales Manager のウィンドウで次のコマンドを実行します。

```
ROLLBACK
```

Accountant が独立性レベルを 3 に上げ、department テーブルのすべてのローを再度選択した場合、データベース・サーバはテーブルの各ローに対挿入ロックを設定し、新しいローの挿入を防止するためにテーブルの最後にもう 1 つ幻ロックを設定します。Sales Manager がテーブルの最後に新しいローを挿入しようとすると、そのコマンドはこの最後のロックによってブロックされます。

Sales Manager は独立性レベル 2 で接続されているにもかかわらず、Sales Manager のコマンドはブロックされました。データベース・サーバは独立性レベルと各トランザクション文の要求に応じ、読み込みロックと同様に幻ロックを設定します。一度対挿入ロックが設定されると、このロックは同時に実行されるほか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」175 ページを参照してください。

実際のロックの重要性についてのチュートリアル

前のチュートリアルと同じ例を使います。このチュートリアルでは、Accountant と Sales Manager の両方は、受注テーブルと受注品目テーブルに関わるタスクを持っているとします。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、ほかのトランザクションが基準を満たすローを挿入することを禁止します。設置するロック数は検索基準やデータベースの設計によって異なります。

まだ実行していない場合は、前のチュートリアルの手順 1 から 3 を実行してください。Interactive SQL が 2 つ起動されます。

1. 次のコマンドを実行して、Sales Manager のウィンドウと Accountant のウィンドウに独立性レベル 2 を設定します。

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 2
```

2. 毎月、販売担当者には、その月の各人の売り上げ高に対し、一定の割合のコミッションが支払われます。Accountant は、2001 年 4 月分のコミッションの小切手を準備しています。彼の最初のタスクは、その月の各担当者の売り上げ合計額を計算することです。

   Accountant のウィンドウに次のコマンドを入力します。価格、受注情報、従業員データがそれぞれ別のテーブルに保存されます。外部キー関係を使用してこれらのテーブルをジョインすることにより、必要な情報を結合します。

```
SELECT emp_id, emp_fname, emp_lname,
    SUM(sales_order_items.quantity * unit_price)
       AS "April sales"
 FROM employee
    KEY JOIN sales_order
    KEY JOIN sales_order_items
    KEY JOIN product
 WHERE '2001-04-01' <= order_date
    AND order_date < '2001-05-01'
 GROUP BY  emp_id, emp_fname, emp_lname
```

emp_id	emp_fname	emp_lname	April sales
129	Philip	Chin	2160.00
195	Marc	Dill	2568.00
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
…	…	…	…

3. Sales Manager は、Philip Chin の多額の売り上げがデータベースに入力されていないことに気づきました。Philip はコミッションの迅速な支払いを希望しているため、Sales Manager は 4 月 25 日の彼の漏れていた注文を入力します。

   Sales Manager のウィンドウに次のコマンドを入力します。1 つの注文に多くの品目を含めることができるため、受注と品目は別のテーブルに入力されます。品目を追加する前に、売り上げ注文にエントリを作成します。参照整合性を維持するために、データベース・サーバは注文がすでに存在する場合にかぎり、トランザクションが品目を注文に追加することを許可します。

```
INSERT into sales_order
 VALUES ( 2653, 174, '2001-04-22', 'r1',
       'Central', 129);
 INSERT into sales_order_items
 VALUES ( 2653, 1, 601, 100, '2001-04-25' );
 COMMIT;
```

4. Accountant は、Sales Manager が新しい注文を追加したことを知りません。新しい注文がもっと前に入力された場合は、Philip Chin の 4 月の売り上げ計算に含まれます。

   Accountant のウィンドウで、4 月の売り上げ合計を再計算します。同じコマンドを使用しますが、Philip Chin の 4 月の売り上げ額が $4560.00 ドルに変更されていることに注意してください。

emp_id	emp_fname	emp_lname	April sales
129	Philip	Chin	4560.00
195	Marc	Dill	2568.00
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
…	…	…	…

ここで、Accountant は 4 月の注文すべてにマークを付けて、コミッションが支払い済みであることを示します。Sales Manager が入力したばかりの注文が 2 度目の検索で見つかりましたが、たとえ Philip の 4 月の売り上げ合計に含まれていなかったとしても、支払い済みのマークが付いています。

5. 独立性レベル 3 では、データベース・サーバは対挿入ロックを設定し、検索または選択条件に合うローをほかのトランザクションが追加できないようにします。

   最初に、Philips の漏れていた注文の挿入をロールバックします。Sales Manager のウィンドウで次のコマンドを実行します。

```
ROLLBACK
```

6. Accountant のウィンドウで、次の 2 つの文を実行します。

```
ROLLBACK;
 SET TEMPORARY OPTION ISOLATION_LEVEL = 3;
```

7. 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 sales_order_items
 WHERE id = 2653;
 DELETE
 FROM sales_order
 WHERE id = 2653;
 COMMIT;
```

8. Accountant のウィンドウで、前と同じクエリを実行します。

```
SELECT emp_id, emp_fname, emp_lname,
    SUM(sales_order_items.quantity * unit_price)
       AS "April sales"
 FROM employee
    KEY JOIN sales_order
    KEY JOIN sales_order_items
    KEY JOIN product
 WHERE '2001-04-01' <= order_date
    AND order_date < '2001-05-01'
 GROUP BY  emp_id, emp_fname, emp_lname
```

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 sales_order
 VALUES ( 2653, 174, '2001-04-22',
          'r1','Central', 129)
```

Sales Manager のウィンドウはハングし、操作は完了しません。ツールバーの [SQL 文の中断] ボタン (または [SQL] メニューの [中断]) をクリックしてこのエントリを一時停止します。

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 sales_order
 VALUES ( 2653, 174, '2001-05-05', 'r1',
       'Central', 129)
```

Sales Manager のウィンドウは再度ハングします。ツールバーの [SQL 文の中断] ボタン (または [SQL] メニューの [中断]) をクリックしてこのエントリを一時停止します。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、これらのロックはほか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することができます。

11. Sales Manager は、注文 2651 に 2 番目の品目を追加したいとします。次のコマンドを使用します。

```
INSERT INTO sales_order_items
 VALUES ( 2651, 2, 302, 4, '2001-05-22' )
```

すべてがうまく処理され、Sales Manager は、注文 2652 にも次の品目を追加することにします。

```
INSERT INTO sales_order_items
 VALUES ( 2652, 2, 600, 12, '2001-05-25' )
```

Sales Manager のウィンドウはハングします。ツールバーの [SQL 文の中断] ボタン (または [SQL] メニューの [中断]) をクリックしてこのエントリを一時停止します。

12. 変更を取り消しデモンストレーション・データベースの変更を防止し、このチュートリアルを終了します。Sales Manager のウィンドウに次のコマンドを入力します。

```
ROLLBACK
```

Accountant のウィンドウに同じコマンドを入力します。

```
ROLLBACK
```

ここで、両方のウィンドウを閉じます。

ロックの仕組み

データベース・サーバがトランザクションを処理するとき、テーブルのローを 1 つまたは複数ロックできます。ロックはほか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」189 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、ほか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。

テーブルのロックについての情報の取得

sa_locks システム・プロシージャを使用して、データベースが持つロックの情報をリストできます。詳細については、『ASA SQL リファレンス・マニュアル』> 「sa_locks システム・プロシージャ」を参照してください。

ロックは Sybase Central でも表示できます。左ウィンドウ枠でデータベースを選択すると、右ウィンドウ枠に [テーブル・ロック] タブが表示されます。このタブでは、それぞれのロックに対する接続 ID、ユーザ ID、テーブル名、ロック・タイプ、ロック名が表示されます。

ロックできるオブジェクト

Adaptive Server Anywhere は次のオブジェクトをロックできます。

- **テーブルのロー**　トランザクションは、特定のローをロックして、別のトランザクションによる変更を防止できます。ローを変更したいトランザクションは、そのローに書き込みロックをかける必要があります。

- **ロー間の挿入ポイント**　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、ローを順次スキャンします。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することができます。

- **テーブル・スキーマ**　トランザクションはテーブルのスキーマをロックして、ほか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することができます。

これらのオブジェクトのうち、ローがおそらく最もわかりやすいです。ローの読み込み、更新、削除、または挿入を行うトランザクションが、ほかのトランザクションの同時アクセスを制限する必要があるというのは理解できます。同様に、新しいカラムを挿入するなどしてテーブルの構造を変更するトランザクションは、ほかのトランザクションに重大な影響を与える可能性があります。このような場合は、ほかのトランザクションのアクセスを制限してエラーを回避することが不可欠です。

ローの順序

ローの順序を決める際には、インデックスを使用できます。この場合、インデックスを構築するときに確立した特定の基準に基づいて決定します。

インデックスがない場合、Adaptive Server Anywhere ではディスク上の物理的配置によってローの順序が決められます。逐次スキャンの場合は、データベース・サーバの内部動作によって特定の順序が定義されます。逐次スキャンのローの順序には依存しないでください。ただし、ローのスキャンという観点からは、Adaptive Server Anywhere は、独自の選択による順序を使用しているにもかかわらず、インデックス・スキャンと同様に要求を処理します。インデックスを使用する場合と同じように、スキャンの位置にロックをかけることができます。

トランザクションはスキャン位置をロックすることによって、ローを順序づける値を特定の範囲に関する、ほかのトランザクションによるいくつかのアクションを回避できます。挿入ロックおよび対挿入ロックは、常にスキャン位置に設定されます。

たとえば、あるトランザクションがローを削除し、それに伴い特定のプライマリ・キー値を削除するとします。このトランザクションは、変更をコミットするかロールバックするまで、この動作を実行する権利を保持する必要があります。ローの削除の場合は、ほかのトランザクションが同じプライマリ・キー値を使用してローを挿入することに

よって、ロールバックが不可能になることのないようにしなければなりません。このローが占有するスキャン位置にロックを設定すると、ほかのトランザクションへの影響を最小限に抑えて、この権利を確保します。

ロックの種類

Adaptive Server Anywhere は、4 種類のロックを使用してロック・スキームを実装し、トランザクション間に適切な独立性レベルを確保します。

- **読み込みロック**　(共有)
- **幻ロックまたは対挿入ロック**　(共有)
- **書き込みロック**　(排他)
- **対幻ロックまたは挿入ロック**　(共有)

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。

これらのロックはそれぞれ独自の目的を持っており、すべて一緒に機能します。それぞれのロックは、ある特定の矛盾を防止し、それが欠けるとその矛盾が発生する可能性があります。選択した独立性レベルに応じ、データベース・サーバはこれらのロックのいくつかまたはすべてを使用して必要な一貫性レベルを維持します。

前述の 4 つのロックは、次のように使われます。

- 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。書き込みロックが設定されると、ほか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も書き込みロックも取得できません。書き込みロックは排他ロックです。
- トランザクションがローを読むときには、「**読み込みロック**」が設定されます。1 つのローに対して、複数のトランザクションが読み込みロックを持つことができます。読み込みロックは共有ロック、すなわち非排他ロックです。ただし、一度読み込みロックのかかったローに対しては、どのトランザクションも書

き込みロックを取得できません。したがって、トランザクションは読み込みロックを設定することで、ほかのトランザクションによるローの修正や削除を防ぐことができます。

- 「**対挿入ロック**」または「**幻ロック**」は、幻ローを防ぐためにスキャン位置に設定されます。このロックは、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、ほか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。インデックスを使用する検索での対挿入ロックは、読み込まれるローごとに読み込みロックを、さらに結果セットの最後のインデックスに対する挿入を防止するためにもう 1 つの読み込みロックを必要とします。インデックスを使わない検索での幻ローは、テーブル内のすべてのローに読み込みロックが必要です。これは、挿入によって結果セットが変更され、それによって同時性が損なわれるのを防ぐためです。

- 「**挿入ロック**」または「**対幻ロック**」は、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ます。一度 1 つの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、ほかのトランザクションはそのローへ対挿入ロックを設定できません。対応するローの読み込みロックが、挿入ロックと同時に必ず設定され、ほかのプロセスが挿入ロックを無視してローを更新または破壊できないようにします。

Adaptive Server Anywhere は、これら 4 種類の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、代わりに次の項で説明するように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。

排他ロックと共有ロック

上記の 4 種類のロックは次の 2 つに分類されます。

- **排他ロック**　1 つのトランザクションだけが、一度にテーブルの 1 つのローに対して排他ロックをかけられます。ほか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、排他ロックを取得することはできません。トランザクションが排他ロックを取得すると、ほか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。

  書き込みロックは排他ロックです。

- **共有ロック**　同時に複数のトランザクションが 1 つのローに共有ロックを設定できます。共有ロックは非排他ロックとも呼ばれます。

  読み込みロック、挿入ロック、対挿入ロックは共有ロックです。

一度に 1 つのトランザクションだけが 1 つのローを変更できます。2 つのトランザクションが同時に 1 つの値を変更できてしまうと、1 つの値が 2 つの異なる値に変更されることになります。したがって、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、トランザクションの意図が競合することはありません。したがって、読み込みロックは共有ロックでも構いません。

対挿入ロックと挿入ロックにも同様の理論を適用できます。多くのトランザクションは、それぞれが対挿入ロックを取得することによって、特定のスキャン位置へのローの挿入を回避できます。同様の論理が挿入ロックにも適用できます。特定のトランザクションが排他アクセスを必要とする場合、同じローに対して対挿入ロックと挿入ロックをかけることによって簡単に排他アクセスを実現できます。これらのロックは、同じトランザクションが保持している場合は競合しません。

どのローが競合するか

次の表に、競合するロックの組み合わせを示します。

	読み込み	書き込み	対挿入	挿入
読み込み		競合		
書き込み	競合	競合		
対挿入				競合
挿入			競合	

これらの競合は、異なるトランザクションがロックを保持している場合にだけ発生します。たとえば、1 つのトランザクションが単一のスキャン位置に対挿入ロックと挿入ロックの両方をかけると、その位置に対する排他アクセスが可能となります。

クエリ時のロック

ユーザが SELECT 文を入力したときに Adaptive Server 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。

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ほか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

意外かもしれませんが、独立性レベル 1 でトランザクションを実行する場合、Adaptive Server Anywhere は独立性レベル 0 で実行するときとほとんど同じ数のロックしか使用しません。実際、それぞれのレベルでデータベース・サーバのオペレーションが異なる点は、2 つしかありません。

オペレーションの最初の違いはロックの設定とは無関係で、むしろロックへの配慮に関するものです。独立性レベル 0 では、トランザクションは、ほかのトランザクションが書き込みロックを設定しているかどうかに関係なく、自由にどのローでも読み込みます。一方独立性レベル 1 のトランザクションは、各ローを読み込む前に書き込みロックがかかっているかをチェックします。書き込みロックがかかっているローは読み込むことができません。この場合、ダーティ・データを読み込むことになるからです。

オペレーションの 2 番目の違いは、カーソル安定性の作成です。カーソル安定性は、カーソルの現在のローに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読み込みロックをかけ、カーソルの別のローが現在のローになるとすぐにこれらのロックをすべて削除します。

独立性レベル 2 の SELECT 文

独立性レベル 2 では、Adaptive Server Anywhere は手順を修正し、読み込みを繰り返し可能にします。SELECT コマンド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

ローの選択を制限するほか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、ローの値が基準を満たしているかをテストし、基準を満たすローに読み込みロックをかけます。

すべての独立性レベルの場合と同様に、レベル 2 で設定したロックには、レベル 1 と 0 のロックがすべて含まれます。このため、ここでも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、Adaptive Server Anywhere はすべてのスケジュールが直列可能であることを確認する必要があります。特に、低レベルでの要件に加えて、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対挿入ロックを使用します。ユーザが選択を行うと、データベース・サーバは、結果セットに情報を提供する各ローに読み込みロックをかけます。そうすることにより、そのローの使用を終了する前に、ほかのトランザクションがそれ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 で使用する手順と似ていますが、*付加された基準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。さもないと、結果を使用中に、だれかがデータベースにアクセスし、別の部署の従業員を販売部に移動する可能性があるからです。

基準を満たすかどうかに関係なくすべてのローに読み込みロックをかけることは、次の 2 つの重要な事柄を含んでいます。

- データベース・サーバは、独立性レベル 2 よりも多くのロックをかける必要がある。
- データベース・サーバは若干効率的に処理を行うことができる。データベース・サーバは、ローの情報が受け入れられるかどうかに関係なくロックをかけるため、読み込みながら各ローに読み込みロックを迅速にかけることができます。

サーバが設定する対挿入ロックの数は、基準やテーブル内で使用可能なインデックスによって大きく左右されます。Employee ID が 123 の従業員に関する情報を選択するとします。その Employee ID が従業員テーブルのプライマリ・キーである場合、データベース・サーバはオペレーションを簡略化できます。データベース・サーバは、プライマリ・キーに自動的に作成されたインデックスを使用して、ローを効率的に検索できます。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ほかの Employee ID を 123 に変更する危険性もありません。サーバは、その ID 番号の従業員に関する情報を含むローだけ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。この部署に従業員が追加される可能性があるため、サーバは従業員テーブルのすべてのローを読み込み、販売部の従業員かどうかを調べる必要があります。この場合は、各ローに読み込みロックと対挿入ロックの両方を設定する必要があります。

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。Adaptive Server Anywhere は、次の手順でデータ整合性を確保します。

挿入時のロックがどのように使用されるかの詳細については、「対挿入ロック」186 ページを参照してください。

1. 新しいローを格納する場所をメモリに確保します。この場所は、他のデータベースからは最初見えないため、別のトランザクションがアクセスする心配はありません。

2. 新しいローに値を入れます。

3. 新しいローに書き込みロックをかけます。

4. 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、ほかのトランザクションは、対挿入ロックをかけて挿入をブロックすることができません。

5. テーブルにローを挿入します。ここで、ほか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、そのローの修正や削除はできません。

6. 該当するインデックスをすべて更新し、必要に応じて、参照整合性とユニーク性の両方を調べます。参照整合性を調べるということは、外部キーが指しているプライマリ・キーが存在しているか確認することです。プライマリ・キーの値はユニークである必要があります。ほか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。

7. 参照整合性が守られている場合は、トランザクションをコミットできます。すなわち、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。

8. カスケード・オプションを選択している場合は、必要に応じてほかのローも挿入し、トリガを起動します。

ユニーク性

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

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

オーファンと参照整合性

外部キーは、通常別のテーブルにあるプライマリ・キーを参照します。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。Adaptive Server 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要求できます。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・キーを持たない別の

ローを挿入できます。この両方のオペレーションを同じトランザクションで実行します。さもなければ、データサーバはこのオペレーションを許可しません。

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;
```

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。

ローを挿入するときに必要なプライマリ・キーが存在しても、データベース・サーバは結果をコミットするときにそのプライマリ・キーが存在しているかを確認する必要があります。この確認は、対象となるローに読み込みロックを設定して行います。読み込みロックをかけてもほか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、削除や変更はできません。

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。

1. 該当するローに書き込みロックをかけます。
2. インデックスに変更されたエントリが含まれている場合、古い値に対応する各インデックス・エントリを削除します。それによって発生したオーファンの記録を作成します。
3. 該当する値を更新します。
4. インデックスが付いた値を変更した場合は、新しいインデックス・エントリを追加します。必要に応じて、ユニークであることを確認し、プライマリ・キーまたは外部キーが変更されていれば、参照整合性を確認します。

5. 参照整合性が守られている場合は、トランザクションをコミットできます。すなわち、ローのエントリすべての以前の値を含め、トランザクション・ログ・ファイルにオペレーションを記録し、すべてのロックを解放します。

6. このオプションを選択していて、プライマリまたはセカンダリ・キーが影響を受ける場合は、挿入または削除オペレーションをカスケードします。

テーブルの 1 つの値を変更するという一見単純に見えるオペレーションでも、多数のオペレーションが必要となることに驚かれるでしょう。データベース・サーバが実行する作業量は、変更する値がプライマリ・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。属性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。

1. 該当するローに書き込みロックをかけます。

2. ローの値に対して存在する各インデックス・エントリを削除します。各インデックス・エントリを削除する直前に、必要に応じて 1 つまたは複数の対挿入ロックを設定し、削除がコミットされる前に別のトランザクションが同様のエントリを挿入することを防止します。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。

3. ほか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。

4. 参照整合性が守られている場合は、トランザクションをコミットできます。すなわち、そのローのすべてのエントリの値を含め、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、ローを無効にします。

5. このオプションを選択していて、プライマリ・キーまたは外部キーを修正した場合は、削除オペレーションをカスケードします。

対挿入ロック

データベース・サーバは、DELETE オペレーションがロールバックできることを確認する必要があります。この確認は、部分的に対挿入ロックを設定して行います。このロックは排他的ではありませんが、ほかのトランザクションは、DELETE オペレーションのロールバックが不可能になるようなローの挿入はできません。たとえば、削除したローにプライマリ・キー値や、別のユニークな値が含まれていた可能性があります。同じ値を持つローの挿入を別のトランザクションに許可した場合、DELETE を取り消すとユニークなプロパティが侵害されてしまいます。

Adaptive Server Anywhere は、インデックスを介して一意性制約を実行します。1 つの属性のプライマリ・キーを 1 つだけ持つ単純なテーブルの場合は、幻ロックは 1 つで十分です。しかしそれ以外の場合は、ロック数は急激に増加してしまう可能性があります。たとえば、テーブルに、属性に関するプライマリ・キーやほかのインデックスが含まれていないことがあります。テーブルのローは基本的な順序を持たないため、挿入を防止するにはテーブル全体に対挿入ロックをかけるしか方法がありません。

ローを削除するということは、多くのロックをかけることを意味します。データベースの同時性への影響を最小限にするには、いくつかの方法があります。すでに説明したように、インデックスとプライマリ・キーはテーブルのローに順番を設定するため、必要なロック数が削減されます。データベース・サーバはこの順番を自動的に利用します。テーブルのすべてのローにロックをかける代りに、*次の*ローにロックをかけることができます。インデックスがないと、ローは順番を持たず、次のローという概念は無意味なものとなります。

データベース・サーバは、削除するローの次のローに対挿入ロックをかけます。テーブルの最後のローを削除する場合は、データベース・サーバは実際には見えない最後のローに対挿入ロックをかけます。実際、テーブルにインデックスが含まれていない場合、設定する対挿入ロック数は、テーブルのロー数よりも 1 つ多くなります。

対挿入ロックと読み込みロック

1 つ以上の対挿入ロックは挿入ロックを排除し、1 つ以上の読み込みロックは書き込みロックを排除しますが、対挿入ロック／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、対挿入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、対挿入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、ほか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。

2 フェーズ・ロック

通常は、前の項で説明したロックに関する一般的な説明を理解していれば十分です。しかし、データベース・サーバの機能を理解しておくと、基本的なオペレーションを実行するときに役立つ場合もあります。これによって、データベースのユーザが遭遇する問題を理解したり、予測することができます。

2 フェーズ・ロックは、スケジュールが直列可能であることを確認するコンテキストでは重要です。「**2 フェーズ・ロック・プロトコル**」は、各トランザクションの手順を指定します。

このプロトコルは、すべてのトランザクションが従う場合、直列可能な正しいスケジュールを保証するため重要です。また、ロック方法の中には一貫性が失われるいくつかのケースを許可するものがあるのはなぜかを理解することもできます。

2 フェーズ・ロック・プロトコル

1. トランザクションは、ローを操作する前に、まずそのローにロックをかけます。
2. ロックの解放後は、トランザクションはロックを追加できません。

実際、トランザクションは COMMIT または ROLLBACK 文で終了するまで、通常はロックを保持します。トランザクションが終了する前にロックを解放すると、変更のロールバック操作ができなくなるため、この場合はローを変更前の状態に戻すためにローを操作しなければなりません。

2 フェーズ・ロック・プロトコルは、次の重要な原理を導きます。

2 フェーズ・ロック原理
すべてのトランザクションが 2 フェーズ・ロック・プロトコルに従うと、インタリーブされたスケジュールはすべて直列可能となります。

つまり、すべてのトランザクションが 2 フェーズ・ロック・プロトコルに従うと、前述のように一貫性が失われることがなくなります。

このプロトコルは、データの完全な一貫性を確保するために必要な操作を定義しますが、データベースのいくつかの操作によって、ある種の矛盾を許容することもできます。矛盾をすべて防止すると、多くの場合データベースの効率低下につながります。

書き込みロックは独立性レベルに関わらず、修正、挿入、削除されたローにかけられます。このロックはコミットおよびロールバックされるまで常に保持されます。

各独立性レベルで適用される読み込みロック

独立性レベル	読み込みロック
0	なし。
1	結果セット内のローのうち、カーソルが置かれているローのみがロックされる。
2	結果セット内のローが、ユーザが COMMIT または ROLLBACK を実行するまでロックされる。
3	結果セットの処理で読み込まれたすべてのローおよび通過したすべての挿入ポイントにロックがかけられる。

詳細については、「直列可能なスケジュール」152 ページを参照してください。

ロックの詳細については、2 つの項で説明します。まず、INSERT、UPDATE、DELETE、または SELECT 時に何が発生するかを説明し、次に、各独立性レベルが読み込みロック、対挿入ロック、挿入ロックの設定にどのように影響するかについて説明します。

独立性レベルを設定して、データベース・サーバ内のロック数を制御できますが、すべてのレベル、たとえばレベル 0 でも発生するロックも多数あります。これらのロック操作は基本的なものです。たとえば、いったんあるトランザクションがローを更新すると、ほかのトランザクションはそのトランザクションが完了するまで同じローを修正できなくなります。これを考慮しないと、最初のトランザクションをロールバックできなくなります。

データベース・サーバが独立性レベル 0 で実行するロック・オペレーションは、基本的な内容を示しているので、最初に正確に理解する必要があります。それ以外のレベルは、下のレベルの機能を失うことなく、さらに機能が追加されたものです。したがって、独立性レベルが高くなると、低いレベルにはない操作が追加されます。

読み込みロックの早期解放

独立性レベル 3 では、読み込むすべてのローに読み込みロックをかけます。通常、処理が終了するまでロックは解放されません。実際、スケジュールを直列可能とするためには、トランザクションがロックを早期に解放しないことが重要です。

Adaptive Server Anywhere は、トランザクションが完了するまで常に書き込みロックを保持します。早期にロックを解放すると、別のトランザクションがそのローを修正でき、最初のトランザクションをロールバックできなくなる可能性があります。

読み込みロックは、ある特別な状況でのみ解放されます。独立性レベル 1 では、カーソルの現在のローにだけ読み込みロックをかけます。ただし独立性レベル 1 では、そのローが現在のローでなくなるとロックは解放されます。データベース・サーバは独立性レベル 1 で繰り返し読み出しを保証する必要がないため、この動作は問題ありません。

独立性レベルの詳細については、「独立性レベルの選択」152 ページを参照してください。

特別な最適化

前の項では、すべてのトランザクションをある 1 つの独立性レベルで実行するときのロックについて説明しました。たとえば、すべてのトランザクションを独立性レベル 2 で実行する場合、ロックは前の項で示したように実行されます。

実際には、データベースで、複数のトランザクションを異なるレベルで実行しなければならない場合があります。口座間の金銭の移動のようなトランザクションは直列可能でなければならないため、独立性レベル 3 で実行する必要があります。住所変更や毎日の売り上げ平均の計算などのトランザクションでは、低レベルの独立性で十分な場合がよくあります

データベース・サーバは、レベル 3 で処理するトランザクションがない場合、いくつかのオペレーションを最適化してパフォーマンスを向上させます。特に、レベル 3 のトランザクションをサポートするには、多くの特別な対挿入ロックや挿入ロックが必要となります。レベル 3 のトランザクションが存在しないとき、場合によっては、データベース・サーバはある種のロックの設定やチェックを省くことができます。

たとえば、データベース・サーバは対挿入ロックを使用して、次の 2 つの状態を確保します。

1. ユニークな属性を持つテーブル内の削除項目をロールバックできる。

2. レベル 3 のトランザクションで幻ローを防ぐ。

レベル 3 のトランザクションが特定のテーブルを使用していない場合、データベース・サーバはユニークな属性を含まないテーブルのインデックスに対挿入ロックをかける必要はありません。ただし、レベル 3 のトランザクションが 1 つでも存在する場合は、レベル 0 のトランザクションであっても、対挿入ロックをかけることによって、レベル 3 のトランザクションがそれらの操作を識別できるようにする必要があります。

データベース・サーバは、前述の最適化を試みるときは必ずテーブルに記録を付加します。レベル 3 のトランザクションが急に開始されても、必要なロックが設定されることを確信できます。

同時に使用される独立性レベルの組み合わせは、データベースのさまざまなユーザが実行するオペレーションに大きく左右されるため、ほとんど制御はできません。しかし、可能なかぎり、レベル 3 のオペレーションを実行する時間を選択してください。なぜなら、レベル 3 のオペレーションはデータベースのオペレーションを大幅に遅らせる可能性があるからです。データベース・サーバは低レベルのオペレーションで余分なオペレーションを実行しなければならないため、この影響は増大します。

特殊な同時性の問題

この項では次の特殊な同時性の問題について説明します。

- 「プライマリ・キーの生成」192 ページ
- 「データ定義文と同時性」194 ページ

プライマリ・キーの生成

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

番号を生成するにはいくつかの方法があります。

例

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

この問題を解決するには、次に示すように方法がいくつかあります。

- 送り状番号を入力するユーザごとに数字の範囲を設定する。

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

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

  このテーブルには、最後に使った送り状番号を入れるローが 1 つあります。新しく送り状を作成するには、データベースに接続し、テーブルの数字を 1 増やし、コミットします。1 増加した

数字は新しい送り状に使います。ほかのユーザも送り状番号を取得できます。瞬時に終わる別のトランザクションで、送り状番号のローが更新されたからです。

- NEWID のデフォルト値のあるカラムを UNIQUEIDENTIFIER バイナリ・データ型と組み合わせて使用して、完全にユニークな識別子を生成する。

  UUID 値と GUID 値を使用してテーブル内のローをユニークに識別できます。この値は、1 台のコンピュータで生成された値が、ほか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」111 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 (
    order_id INTEGER NOT NULL DEFAULT
AUTOINCREMENT,
    order_date DATE,
    primary key( order_id )
 )
```

  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。

レプリケートされたデータベースのユニークな値

データベースをレプリケートし、後でマージの必要なエントリを複数の人間が追加する場合は、別の方法が必要です。詳細については、「レプリケーションと同時性」195 ページを参照してください。

データ定義文と同時性

CREATE INDEX、ALTER TABLE、TRUNCATE TABLE のように、テーブル全体を変更するようなデータ定義文は、その文が動作しているテーブルがその時点でほかの接続に使われている場合は処理されません。これらのデータ定義文は時間がかかり、データベース・サーバはコマンドの実行中は対象となるテーブルの参照要求を処理しません。

CREATE TABLE 文は同時性の問題は起こしません。

GRANT 文、REVOKE 文、SET OPTION 文も同時性の問題を起こしません。これらのコマンドは、データベース・サーバに送られる新しい SQL 文には影響しますが、未処理の文には影響しません。

データベースに接続しているユーザに対しては、GRANT と REVOKE は許可されません。

データ定義文と同期されたデータベース
同期を使用するデータベースでデータ定義文を使用する場合は特に注意が必要です。詳細については、別のマニュアル 『Mobile Link 管理ガイド』> 「Mobile Link 管理ガイド」と 『SQL Remote ユーザーズ・ガイド』> 「SQL Remote ユーザーズ・ガイド」を参照してください。

レプリケーションと同時性

ネットワークのコンピュータの中には、会社の外に持ち出したり、ときどきネットワークに接続されるポータブル・コンピュータが含まれることがあります。ネットワークに接続されていない間にデータベース・アプリケーションを使いたいこともあります。

この問題を解決するには、データベース・レプリケーションが最適です。SQL Remote または Mobile Link 同期を使用すると、統合または master データベースの情報を任意の数のほかのコンピュータにパブリッシュできます。特定のコンピュータでレプリケートされた情報を精密に制御できます。だれでも特定のテーブル、またはテーブルのローやカラムの一部を受け取ることができます。各ユーザが受け取る情報をカスタマイズすることで、データベースのコピーを必要以上に大きくせず、必要な情報だけを含めるようにできます。

SQL Remote のレプリケーションと Mobile Link 同期の詳細については、別のマニュアル『*SQL Remote ユーザーズ・ガイド*』と『*Mobile Link 同期ユーザーズ・ガイド*』を参照してください。そのため、この項での説明は完全なものではありません。この項ではむしろ、ロックおよび同時性での考慮すべき点に直接関係のある概念を説明します。

SQL Remote および Mobile Link は、レプリケートされるデータベースを中央の統合データベースから更新したり、この中央のデータをリモート・マシンで処理されるトランザクションの結果として更新できます。更新はどちらの方向でも行えるため、この機能は「**双方向レプリケーション**」と呼ばれます。

トランザクションの結果は統合データベースに反映できるため、トランザクションを中央マシンまたはリモート・マシンのどちらで処理するかに関係なく、トランザクションの同時実行を可能にします。

トランザクションが複数のマシンで同時に発生することがあります。しかも、同じデータを処理する場合があります。この場合、マシンは物理的には接続されていません。リモート・マシンがロックを設定したり、変更されたローを識別するために統合データベースに接続する方法はありません。そのため、すべてのトランザクションが 1 つのサーバで処理される場合とは違って、ロックで矛盾を防止することはできません。

リモート・マシンがデータベースの完全なコピーを保持できないという問題もあります。主要な統合データベースで直接実行されているトランザクションがあるとします。このトランザクションは、複数のテーブルのローを処理できます。同じトランザクションがリモート・データベースで実行されないことがあります。そのマシンで対象となるテーブルの 1 つまたは両方がレプリケートされる保証がないからです。たとえ、同じテーブルが存在しても、2 つのデータベースの情報が同期化された時期により、まったく同じ情報を含まないことがあります。

上記の制約に適合するには、レプリケーションはトランザクションではなくオペレーションを基に作成します。「**オペレーション**」は、テーブル内の 1 つのローへの変更を示します。この変更は、UPDATE、INSERT、または DELETE 文の実行結果です。UPDATE または DELETE のオペレーションでは各カラムの初期値が識別され、INSERT または UPDATE のトランザクションでは最終値が記録されます。

1 つのトランザクションから、オペレーションがまったく生じないことも、1 つまたは複数生じることもあります。複数のトランザクションから 1 つのオペレーションが生じることはありません。2 つのトランザクションがテーブルを変更する場合、それに対応して複数のオペレーションが発生します。

リモート・コンピュータで処理されるトランザクションからオペレーションが発生した場合は、情報をマージできるように統合データベースに渡す必要があります。これに対して、統合コンピュータのトランザクションでオペレーションが発生した場合、オペレーションをいくつかのリモート・サイトに送信する必要がありますが、ほかには送る必要がありません。各リモート・サイトには、データベースの一部のレプリカが含まれているため、SQL Remote は、データベースのレプリカ部分に関係する場合にだけリモート・サイトにオペレーションを渡します。

トランザクション・ログ・ベースのレプリケーション

SQL Remote は「**トランザクション・ログ・ベース**」のレプリケーション・メカニズムを使用します。SQL Remote をマシンで起動すると、トランザクション・ログをスキャンして、転送する必要があるオペレーションを識別し、1 つまたは複数のメッセージを準備します。

SQL Remote は、いくつかの方法を使用してこれらのメッセージをコンピュータに渡すことができます。SQL Remote は、メッセージを含むファイルを作成し、指定されたディレクトリにそれを保管します。また、SQL Remote は、一般的なメッセージ・プロトコルを使用してメッセージを渡すこともできます。現在使用している電子メール・システムもおそらく使用できます。

リモート・サイトのオペレーションを統合データベースにマージするときに問題が起こる可能性があります。異なるリモート・サイトの 2 人のユーザが同じテーブルの同じ値を変更したとします。Adaptive Server Anywhere に組み込まれているロック機能は、同じサーバで同時に処理される複数のトランザクション間の矛盾は防止できますが、同じ値を変更するパーミッションを持つ 2 人のリモート・ユーザ間の矛盾を自動的に防止することはできません。

データベース管理者は、適切にデータベースを設計したり、競合解決アルゴリズムを作成したりすることで、このような問題を防止できます。たとえば、1 人のユーザだけが特定のテーブル内の特定の値の範囲を更新するように決めることができます。そのような制限が無理な場合は、SQL Remote の競合解決機能を使用して、問題のデータにふさわしい方法で矛盾を解決するトリガと手順を実行できます。

SQL Remote には、データベース・レプリケーションを最大限に利用するのに必要なツールとプログラミング機能が備わっています。詳細については、『*SQL Remote ユーザーズ・ガイド*』と『*Mobile Link 同期ユーザーズ・ガイド*』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。それぞれのトランザクションは、加えた変更をロールバックしたり、これらの変更をコミットして確定すると終了できます。

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

パフォーマンスを向上させるには、複数のトランザクションを同時に実行する必要があります。各トランザクションは、コンポーネント SQL 文で構成されています。複数のトランザクションを同時に実行する場合、データベース・サーバは個々の文の実行をスケジュールします。トランザクションを同時に実行すると、同じトランザクションを順次実行した場合にはない、新しい矛盾した結果を生じる可能性があります。

多くの矛盾が考えられますが、次に示す 4 つのケースは ISO SQL/92 標準にも記載され、独立性レベルもこれを基に定義されているため、特に重要です。

- **ダーティ・リード**　あるトランザクションがデータを修正した後、コミットする前に別のトランザクションがそのデータを読み込んでしまうこと。
- **繰り返し不可能読み出し**　あるトランザクションが同じローを 2 度読み込んだ場合に、得られる値が異なること。
- **幻ロー**　トランザクションが特定の基準に従ってローを 2 回選択したときに、2 回目の結果セットに新しいローが含まれること。

- **更新内容の消失**　トランザクションがローに対して行った変更が、別のトランザクションが前のデータを基にして更新内容を保存することを許可されたため、完全に消失してしまうこと。

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

ロックは、許可する干渉の量とタイプを制御します。Adaptive Server 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、Adaptive Server Anywhere はスケジュールが直列可能であること、つまり、すべてのトランザクションを実行した結果とそれらを順次実行した結果が同じになることを保証します。

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

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

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

ポータブル・コンピュータの普及に伴い、データベースのレプリケーションの必要性が高まります。レプリケーションは Adaptive Server Anywhere の機能の中でも特に便利なものですが、同時性に関して新しい矛盾を引き起こします。これについては、別のマニュアルで触れることにします。

第 5 章

パフォーマンスのモニタリングと改善

この章の内容

この章では、データベースのパフォーマンスをモニタリングし改善する方法について説明します。

パフォーマンス分析ツール

データベースのパフォーマンスを改善する作業は手間がかかるというわけではありませんが、常に計画を立ててから開始することをおすすめします。データベースの現在のパフォーマンスを評価し、変更を加える前にまずすべての可能なオプションを検討します。Adaptive Server Anywhere のパフォーマンス機能とパフォーマンス分析ツールを使用してデータベースのスキーマを再評価することで、パフォーマンスの問題を診断し、訂正できます。また、データベースのパフォーマンスを最適に保てます。

根本的に、データベースのパフォーマンスは、その骨格の設計に大きく左右されます。したがって、パフォーマンスを改善する最も基本的な方法の 1 つは、優れたスキーマを設計することです。データベース・スキーマは、データベースのスケルトンであり、テーブル、ビュー、トリガ、そしてそれらの関係などの定義が含まれます。データベース・スキーマを再評価し、これから説明する、少しの変更で大きな改善がもたらされる領域に留意してください。

Adaptive Server 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ツールが用意されています。ツールには、要求ロギング、プロシージャ・プロファイリング、グラフィカルなプラン、パフォーマンス・モニタ、タイミング・ユーティリティなどがあります。

要求ロギング

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、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をおすすめします。要求レベル・ログは、問題の原因となっている可能性の高い、サーバに対する特定の要求を特定するためにも役立ちます。

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。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。

たとえば、タイムスタンプ、接続 ID、要求の種類などの情報がログに含まれます。-zr サーバ・オプションを使用できます。-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。

詳細については、『ASA データベース管理ガイド』> 「-zr サーバ・オプション」と 『ASA データベース管理ガイド』> 「-zo サーバ・オプション」を参照してください。

sa_get_request_times ([request_log_filename [, connection_id]]) ストアド・プロシージャは、要求ログを読み込み、グローバル・テンポラリ・テーブル **satmp_request_time** にログ内の文とその実行時間を記録します。INSERT 文、UPDATE 文、DELETE 文の場合、実行時間そのものが記録されます。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。

原因を突き止めるために **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([request_log_filename [, connection_id]]) を使用して **sa_get_request_times()** を呼び出し、得られた **satmp_request_time** を別のグローバル・テンポラリ・テーブル **satmp_request_profile** に要約できます。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。

要求ログのフィルタ

要求ログへの出力は、特定の接続やデータベースからの要求のみを含めるために除外することができます。これにより、アクティブな接続の多いサーバや複数のデータベースのあるサーバをモニタリングするときのログ・サイズを縮小できます。

❖ 接続に基づ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('requests_for_connection',connecti
on-id)
```

connection-id は、**sa_conn_info()** から入手できます。

❖ 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('requests_for_database',database-
id)
```

database-id は、データベースに接続している場合に **connection_property('DBNumber')** から入手できます。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リセットします。

```
call sa_server_option('requests_for_connection',-1)

call sa_server_option('requests_for_database',-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。
  - 値 **sql+hostvars** を指定して、-zr コマンド・ライン・オプションを使用します。
  - 次の文を実行します。

```
call
sa_server_option('RequestLogging','sql+host')
```

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。このテンポラリ・テーブルがないバージョン 9.0.0 より前のデータベースでは、ホスト変数の値は無視されます。

インデックス・コンサルタント

インデックス・コンサルタントは、負荷が高いときや複雑なクエリが実行されるときのパフォーマンスの改善に役立つインデックスを推奨します。単一または複数のクエリを入力として受け取り、データベースにインデックスを追加することや、使用されていないインデックスを削除することを推奨します。

インデックス・コンサルタントは、データベース内に一連の仮想インデックスを作成します。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、個々のカラム・インデックスだけでなく複数のカラム・インデックスも評価します。またクラスタード・インデックスまたは非クラスタード・インデックスの影響を調べます。

インデックス・コンサルタントの詳細については、「インデックス・コンサルタントの概要」86 ページを参照してください。

プロシージャのプロファイル表示

プロシージャ・プロファイリングは、ストアド・プロシージャ、関数、イベント、トリガの実行所要時間を示します。また、プロシージャの各行の実行時間も表示できます。データベース・プロファイリング情報を使用すると、どのプロシージャ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・プロシージャを分析できます。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。

ストアド・プロシージャを使用して、サーバが収集したプロシージャ・プロファイリング情報を表示できます。データベース内のすべてのプロシージャに関する情報を取得するには、**sa_procedure_profile_summary** ストアド・プロシージャを使用します。このプロシージャを使用すると、ストアド・プロシージャ、関数、イベント、トリガのプロファイリング・データを同じ結果セット内で表示できます。ただし、この情報の検査には、Sybase Central の使用をおすすめします。

プロファイリングは、動的に有効または無効にできます。またプロファイリングで生成されるデータは永続化されず、サーバによってメモリに格納されます。Sybase Central の [プロファイル] タブを使用してこのデータを表示できます。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、サーバが停止されるまで、データベースはプロファイリング情報を収集します。プロファイリング情報は累積されます。その精度は、1 ミリセカンドです。

グラフィカルなプラン

Interactive SQL のグラフィカルなプラン機能を使用すると、クエリの実行プランを表示できます。この機能は、特定のクエリのパフォーマンス上の問題を診断するのに役立ちます。たとえば、プラン内の情報に基づいてデータベースのどこにインデックスを追加するかを判断できます。

グラフィカルなプランは、短いプランや長いプランよりも多くの情報を提供します。グラフィカルなプランの表示は、統計情報付きか統計情報なしかを選択できます。どちらを選択した場合も、プランの中で最も高コストとして推定された部分をすぐに表示できます。統計情報付きのグラフィカルなプランの表示は高コストですが、クエリの実行時にサーバがモニタしている実際のクエリ実行統計が表示されます。このため、クエリ・オプティマイザがアクセス・プランの作成時に使用する推定を、実行中にモニタされた実際の統計と直接比較できます。ただし、オプティマイザはクエリのコストを正確に推定できないことが多いので、実際の統計と推定とに違いがあると予想してください。グラフィカルなプランは、アクセス・プランのデフォルト・フォーマットです。

グラフィカルな図でノードをクリックすると、プラン内のノードに関する詳細情報を取得できます。統計情報付きのグラフィカルなプランには、グラフィカルなプランに表示されるすべての推定に加えて、文

を実行したときの実際の実行時間コストが表示されます。これを行うには、文を実際に実行します。これは、コストの高いクエリの場合、プランへのアクセスに遅延が生じる可能性があることを意味します。また、削除や更新など、クエリのどのような部分も実際に実行しなければなりませんが、ロールバックを利用すればこれらの変更は取り消し可能です。

パフォーマンスに問題があり、推定されるロー・カウントや実行時間が予測とは異なる場合は、統計情報付きのグラフィカルなプランを使用してください。統計情報付きのグラフィカルなプランを使用すると、推定と実際の統計を比較できます。実際値と推定値が大きく異なる場合は、情報不足のためにオプティマイザが適切な推定を準備できない可能性があることを示す警告とみなすことができます。

次に、統計情報付きのグラフィカルなプランでチェックできる主な統計と、考えられる対応策を示します。

- ロー・カウントは、結果セット内のローを測定します。推定ロー・カウントが実際のロー・カウントと大きく異なる場合は、基本となっている述部の選択性が正しくない可能性があります。

- クエリ・オプティマイザの正常な処理には、正確な選択性推定が不可欠です。たとえば、オプティマイザが述部の選択性が高い (5% の選択性など) と誤って推定しても、実際には述部の選択性がかなり低い (50% など) 場合は、パフォーマンスが低下することがあります。一般的に、推定は正確ではないことがあります。ただし、極端に大きな誤差は問題がある可能性を示しています。ヒストグラムが存在しないベース・カラムに対する述部がある場合は、ヒストグラムを作成するために CREATE STATISTICS 文を実行すると問題を修正できる場合があります。選択性の誤差に問題が残る場合は、最後の手段として、クエリ・テキストに述部とともにユーザ推定選択性を指定することも可能です。

- 実行時間は、クエリの実行所要時間を測定します。テーブル・スキャンまたはインデックス・スキャンの実行時間が正しくない場合は、REORGANIZE TABLE 文を実行するとパフォーマンスを向上できることがあります。sa_table_fragmentation 関数と sa_index_density 関数を使用して、テーブルとインデックスのどちらが断片化されているかを判断できます。

- 推定ソースが Guess の場合、オプティマイザが使用する情報はなく、問題を示している可能性があります。推定ソースが Index で、選択性推定が正しくない場合は、インデックスに無駄が多い可能性があります。REORGANIZE INDEX 文を使用してインデックスの断片化をデフラグすると改善できる場合があります。

- キャッシュの読み込み数とヒット数がまったく同じ場合は、データベース全体がキャッシュにあり、良好な状態です。読み込み数がヒット数よりも多い場合は、サーバがキャッシュにアクセスしようとしたが失敗し、ディスクから読み込むことを意味します。これは、ハッシュ・ジョインなどの場合に予想されます。ネスト・ループ・ジョインなどの場合、キャッシュ・ヒット率が低いことは、パフォーマンスの問題を示す場合があり、キャッシュ・サイズを大きくすると改善できる可能性があります。

パフォーマンス・モニタ

パフォーマンス・モニタは、ディスク・アクセスやメモリ・アクセスなど、データベース・サーバの動作に関する情報を詳細に追跡するのに役立ちます。

Sybase Central パフォーマンス・モニタを使用すると、接続できる Adaptive Server Anywhere データベース・サーバのさまざまな統計値を、Sybase Central でグラフ形式で表示できます。Sybase Central の統計値はすべて、[統計情報] フォルダに表示されます。

次に、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・モニタを使用しているときは、パフォーマンス・モニタがサーバに対して実際のクエリを使用して統計値を収集していることに注意してください。つまり、モニタ処理自体が統計値の一部 (秒あたりのキャッシュ読み込みなど) に影響を与えているのです。サーバの統計値をグラフにするより正確な方法として、Windows パフォーマンス モニタを使用することもできます。

Windows 版モニタには 2 つの利点があります。

- 主にネットワーク通信に関わるパフォーマンス統計値をより多く提供します。
- Sybase Central 版モニタとは異なり、Windows 版モニタは非介入です。Windows NT 版モニタは、サーバに対してクエリを実行する代わりに、共有メモリ・スキームを使用するので、統計値自体には影響を与えません。

複数のバージョンの Adaptive Server 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。

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、*< インストール・ディレクトリ >¥samples¥asa¥* ディレクトリにあります。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。これらのツールを使用して、統計情報付きのグラフィカルなプランよりもさらに正確に測定できます。これらのユーティリティにより、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。

Fetchtst は、任意のクエリについてフェッチ率を測定します。Instest 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 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。

同時実行性

データベース・サーバがトランザクションを処理するとき、テーブルのローを 1 つまたは複数ロックできます。ロックは他のトランザクションが同時にアクセスすることを防止し、データベースに格納されている情報の信頼性を維持します。また、更新中の情報を識別して、結果クエリの精度を高めます。

データベース・サーバは自動的にこれらのロックを設定するので、明示的な指示は必要ありません。データベース・サーバは、トランザクションが獲得したすべてのロックを、そのトランザクションが完了するまで保持します。ローにアクセスしているトランザクションは、ロックを保持しているといえます。ロックの種類により、他のトランザクションのそのローへのアクセスは限定されるか、まったくできなくなります。

頻繁に 1 つまたは複数のローに多数のユーザが同時にアクセスする場合、パフォーマンスは低下することがあります。ロックが問題になっていると考えられる場合は、**sa_locks** プロシージャを使用して、データベースのロックに関する情報を入手できます。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。

最高のパフォーマンスを得るヒント

Adaptive Server Anywhere は、非常に優れたパフォーマンスを自動的に提供します。ここでは、この製品から最高のパフォーマンスを引き出すためのヒントを提供します。

常にトランザクション・ログを使用する

Adaptive Server Anywhere はトランザクション・ログがないほうがディスクで管理する情報が少なくなるので実行が速くなる、と思っているユーザもいるかもしれません。しかし、実際はその逆です。トランザクション・ログは優れた保護を提供するだけでなく、パフォーマンスを飛躍的に改善します。

トランザクション・ログなしで操作すると、Adaptive Server Anywhere は各トランザクションの終わりにチェックポイントを実行します。こうした変更の書き込みには、大量のリソースが消費されます。

しかし、トランザクション・ログを使用すると、Adaptive Server Anywhere では変更が発生したときにその詳細を示すメモを書き込めばいいだけです。新しいデータベース・ページすべてを、最も効率のいい時に一度に書き込むように選択できます。「チェックポイント」は、情報がデータベース・ファイルに入力され、矛盾せず、最新のものであるということを確実にします。

ヒント
必ずトランザクション・ログを使用するようにしてください。トランザクション・ログはデータの保護を助け、*しかも*パフォーマンスを大いに向上させます。

トランザクション・ログを、メイン・データベース・ファイルとは別の物理デバイスに入れることで、パフォーマンスをさらに改善できます。追加されたドライブ・ヘッドは、通常、トランザクション・ログの終わりを探す必要はありません。

キャッシュ・サイズの拡大

Adaptive Server 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、Adaptive Server 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、Windows NT/2000/XP、Windows 95/98/Me では、データベース・サーバは必要に応じてキャッシュ・サイズを動的に変更します。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。

Windows CE と Novell NetWare では、キャッシュ・サイズはデータベース・サーバ起動時に設定されます。同時に実行される他のアプリケーションやプロセスの要件を考慮に入れて、できるだけ多くのメモリをデータベースのキャッシュに割り付けるようにしてください。特に、Java オブジェクトを利用するデータベースにとっては、キャッシュ・サイズが大きいと非常にメリットがあります。データベースで Java を使用する場合は、最低 8 MB のキャッシュが必要です (ただし、Windows CE 版の Adaptive Server Anywhere にはデータベース内の Java 機能はありません)。

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ほう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 をさらに購入するのも無駄ではない場合もあります。

詳細については、「パフォーマンス向上へのキャッシュの使用」231 ページを参照してください。

テーブル構造の正規化

通常、テーブルの各カラムの情報はプライマリ・キーの値だけに依存させてください。そうでない場合は、1 つのテーブルに同じ情報の複数のコピーが含まれてしまい、テーブルを正規化しなければならなくなります。

正規化によって、リレーショナル・データベース内の重複データは減少します。たとえば、会社の従業員が多数のオフィスで働いているとします。データベースを正規化するには、住所や代表電話番号などオフィスに関する情報を、各従業員用にコピーせず、個別のテーブルに入れておくことを考えます。

ただし、一般的に正規化の利点は大きく見られすぎる傾向があります。重複する情報量が少ない場合は、情報をコピーし、トリガなどの制約を使用して整合性を維持するほうが良い場合があります。

インデックスの有効な使用

クエリを実行するとき、Adaptive Server Anywhere は各テーブルへのアクセス方法を選択します。インデックスは、アクセスを高速化します。データベース・サーバが適切なインデックスを見つけられないときは、テーブルを順にスキャンしなければならず、時間を要します。

たとえば、大規模なデータベースから複数の人を検索する必要があるのに、姓か名前のどちらかしかわからないとします。インデックスがない場合、Adaptive Server 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、Adaptive Server 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。

インデックスの使用

Adaptive Server 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。Adaptive Server 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、Adaptive Server 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。詳細については、「インデックス・コンサルタントの概要」86 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して、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します。

詳細については、「インデックス」535 ページと「クラスタード・インデックスの使用」81 ページを参照してください。

異なるファイルの異なるデバイスへの配置

ディスク・ドライブは、最新のプロセッサや RAM よりオペレーション速度が非常に遅くなります。しばしば、ディスクによるページの読み込みまたは書き込みをただ待っていることが、データベース・サーバの処理を低速にする原因になっています。

異なる物理データベース・ファイルを異なる物理デバイスに入れると、ほとんどの場合データベースのパフォーマンスは向上します。たとえば、1 つのディスク・ドライブがキャッシュとデータベース・ページとの相互スワップを実行中の場合も、別のデバイスはログ・ファイルに書き込みができます。

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。

Adaptive Server 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、同じデータベースを入れる追加ファイルです。用途に適した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**」で重複コピーを管理できます。Adaptive Server 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 (1 つ使用する場合に) 物理的に別個のドライブに置くと、ディスク障害に対する保護を強化できます。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、Adaptive Server 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、dblog コマンド・ライン・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。

Adaptive Server Anywhere では、ソートや UNION 処理など、オペレーション用キャッシュで使用できる領域よりも多くの領域が必要な場合があります。こうした領域を必要とする場合、Adaptive Server Anywhere 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、4 種類目のファイルである「**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することになります。

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、Adaptive Server 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。Adaptive Server Anywhere は次の環境変数を次に示す順に調べ、テンポラリ・ファイルを入れるディレクトリを決定します。

1. ASTMP
2. TMP
3. TMPDIR
4. TEMP

上記のいずれも定義されていない場合、Adaptive Server Anywhere はテンポラリ・ファイルを現在のディレクトリに入れますが、このロケーションは最高のパフォーマンスを実現するためには適当ではありません。

環境変数が定義されていない場合、Adaptive Server Anywhere は、UNIX ではテンポラリ・ファイルを */tmp* ディレクトリに入れます。Windows では、現在のディレクトリにいれます。

マシンが十分な数の高速デバイスを持っている場合、これらのファイルをそれぞれ別のデバイスに入れてパフォーマンスを改善できます。データベースを複数の DB 領域に分割し、別のデバイスへの配置もできます。その場合は、テーブルを個別の DB 領域にグループ分けして、一般的なジョイン操作によって異なるファイルから情報を読み込めるようにします。

同様の方式では、テンポラリ・ファイルとデータベース・ファイルの RAID デバイスまたは Windows NT 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。

詳細については、『ASA データベース管理ガイド』> 「-fc サーバ・オプション」を参照してください。

ワーク・テーブルの詳細については、「クエリ処理中のワーク・テーブルの使用」243 ページを参照してください。

データ・リカバリについては、『ASA データベース管理ガイド』>「バックアップとデータ・リカバリ」を参照してください。

トランザクション・ログと *dbcc* ユーティリティについては、『ASA データベース管理ガイド』>「トランザクション・ログ・ユーティリティのオプション」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、Adaptive Server 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・ファイルを使用しないでオートコミット・モードを使用すると、特にコストが高くなります。各文の終わりで、チェックポイントが強制的に実行されます。チェックポイントとは、多数の情報ページをディスクに書き込む操作です。

各アプリケーション・インタフェースには、オートコミット動作を設定する独自の方法があります。Open Client、ODBC、JDBC インタフェースでは、オートコミットはデフォルトの動作です。

オートコミットの詳細については、『ASA プログラミング・ガイド』>「オートコミットまたは手動コミット・モードの設定」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」689 ページを参照してください。

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。

データベースを検証する場合のパフォーマンスの改善については、『ASA データベース管理ガイド』> 「データベース検証時のパフォーマンスの改善」を参照してください。

Adaptive Server Anywhere の圧縮機能の使用

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、Adaptive Server Anywhere のパフォーマンスを大幅に向上できる場合があります。

特定の状況で圧縮を有効にすることが役立つかどうかを判別するには、該当するネットワークで該当のアプリケーションを使用してパフォーマンス分析を実行してから、運用環境で圧縮を使用することをおすすめします。

圧縮機能を有効にすると、データ・パケットに格納される情報量が増大し、特定のデータ・セットの送信に必要なパケット数が減少します。パケット数を減らすと、データを高速で送信できます。

圧縮閾値を指定すると、圧縮対象になるデータ・パケットの最小サイズを選択できます。圧縮閾値の最適値は、使用するネットワークのタイプや速度など、さまざまな要素の影響を受けることがあります。

圧縮機能を使用したパフォーマンスの改善については、『ASA データベース管理ガイド』> 「パフォーマンス改善のための通信圧縮設定の調整」を参照してください。

圧縮設定の詳細については、『ASA データベース管理ガイド』> 「Compress 接続パラメータ [COMP]」と 『ASA データベース管理ガイド』> 「CompressionThreshold 接続パラメータ [COMPTH]」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

次の状況に該当するかどうかを調べてください。

- ネットワークの遅延時間が長い
- アプリケーションがカーソルを開く要求と閉じる要求を多数送信する

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の数を減らし、パフォーマンスを向上させることができます。

これらのパラメータについては、『ASA データベース管理ガイド』> 「LazyClose 接続パラメータ [LCLOSE]」と 『ASA データベース管理ガイド』> 「PreFetchOnOpen 接続パラメータ」を参照してください。

テーブル幅の縮小

カラム数の多いテーブルは、ワイド・テーブルと呼ばれます。テーブル内のカラム数が多いため個々のローのサイズがデータベース・ページ・サイズを超える場合、各ローは、複数のデータベース・ページに分割されます。ローが多数のページに分割されるほど、各ローの読み込みには時間がかかります。パフォーマンスが悪い場合に、カラム数の多いテーブルがある場合は、テーブルを正規化してカラム数を減ら

すことを検討します。テーブルを正規化できない場合は、データベース・ページ・サイズを拡大するとパフォーマンスが改善されることがあります。特にほとんどのテーブルの幅が広い場合は有効です。

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、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。

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。

適切なデータ型の使用

データ型は、値の範囲、値に対して可能な演算、メモリでの値の格納方法など、特定のデータ・セットに関する情報を格納します。データに適切なデータ型を使用することで、パフォーマンスを改善できます。たとえば、数値データのみを含む値にデータ型 char または string を割り当てないでください。また、コストの高い数値型や文字列型でなく、経済的なデータ型をできるだけ選択してください。

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を設定 (AUTOINCREMENT ON) 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。

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、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、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。

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、*employee* テーブルから *department* 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 employee 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。

カラムの順番に注意する

ローのカラムは、作成された順に逐次方式でアクセスされます。たとえば、ローの終わりのカラムにアクセスするために、Adaptive Server Anywhere は、ロー内で手前にあるカラムをスキップする必要があります。プライマリ・キー・カラムは、常にローの始めに格納されます。このため、テーブル内で小さいカラムか頻繁にアクセスされるカラム、またはその両方に当てはまるカラムが、あまりアクセスされないカラムの前にくるようにテーブルを作成することが重要です。

新しい機能を利用するためのアップグレード

Adaptive Server Anywhere のパフォーマンス機能と操作性は、常に評価され、拡張されています。どの後続リリースにおいても、パフォーマンスの最適化に役立つ新機能や動作の変更を利用できます。

新リリースで古いバージョンの Adaptive Server Anywhere で作成されたデータベースを実行することもできますが、新機能の大半は、データベースをアップグレードした場合のみ利用できます。アップグレード・ユーティリティを実行すると、システム・テーブル、システム・プロシージャ、データベース・オプションが追加または修正され、データベースを古いバージョンの Adaptive Server Anywhere から新しいバージョンにアップグレードできます。

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呼ばれることもあります。

データベースを再構築すると、データやスキーマを含むすべての情報がデータベースから削除され、同一の形式ですべての情報が戻されるので、ディスク・ドライブの断片化デフラグメンテーションと同じように、スペースが埋められてパフォーマンスが改善されます。また、再構築は、特定の設定を変更する機会にもなります。一方、アップグレード・ユーティリティの使用は、再構築より短時間で済みますが、ファイルの格納状態には影響しません。

データベースをアップグレードするとき、再構築を行うことによっても最新バージョンのすべての新機能とパフォーマンスの改善点を利用できるようになります。

データベース設定とサーバ設定の検査

データベースとサーバの設定を検査し、ハードウェアとファイル領域が使用状況に適切であるかどうかを確認してください。

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。Adaptive Server Anywhere がサポートするページ・サイズ (単位はバイト) は、1024、2048、4096、8192、16384、または 32768 です。デフォルトは、2048 バイトです。どのページ・サイズを選択しても利点と欠点があります。

ページが小さいと、保持する情報が少なくなり、特にページの半分をわずかに超えるローを挿入する場合などは領域を効率良く使用できません。しかし、ページ・サイズが小さいと、同じサイズのキャッシュにより多くのページを格納できるため、Adaptive Server Anywhere を少ないリソースで実行できます。特に、メモリが限られている小型のマシンでデータベースを実行しなければならない場合に便利です。また、不特定のロケーションから少量の情報を取得するためにデータベースを使う場合にも便利です。

一方、ページ・サイズが大きいと、Adaptive Server Anywhere はデータベースをより効率的に読み込めます。また、データベースの規模が大きい場合や、逐次テーブル・スキャンを実行するクエリの場合にパフォーマンスが高くなる傾向があります。通常は、ディスクの物理的な設計のために、大きなブロックを少数取り出す方が、小さなブロックを多数取り出す場合よりも効率的です。大きいページ・サイズには、他の利点もあります。インデックスのファンアウトを改善することによってインデックス・レベル数を減らし、カラム数の多いテーブルが作成できます。

ただし、ページ・サイズが大きいと、メモリも余分に必要になります。1 ページに格納できるロー数は最大 255 であるため、ローの少ないテーブルは各ページの領域を満たさず、領域の利用は非効率的になります。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。

サーバ・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。そして、満足できる結果を得られた最小のページ・サイズ (4 K 以上) を選択します。同じサーバで多数のデータベースが起動される場合は、正しい (そして適切な) ページ・サイズの選択が特に重要です。

既存のデータベースのページ・サイズは変更できません。その代わり、新しいデータベースを作成し、*dbinit* の `-p` オプションを使用してページ・サイズを指定します。たとえば、次のコマンドは 4 K 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
```

大きいページ・サイズの詳細については、『ASA データベース管理ガイド』> 「最大ページ・サイズの設定」を参照してください。

分散読み込み

Windows NT サービス・パック 2 以降のシステムまたは Windows 2000/XP システムで作業している場合は、4 K 以上のページ・サイズを使用すると、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 K 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。

ファイル、テーブル、インデックスの断片化の検査

データベースの変更には断片化がともないます。ファイル、テーブル、またはインデックスが必要以上に断片化されていると、パフォーマンスが低下することがあります。これはデータベースのサイズに比例して重要になります。Adaptive Server 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。

パフォーマンスの低下が著しい場合は、次の事項を検討してください。

- データベースを再構築して、テーブルかインデックスまたはその両方の断片化を低減する。
- 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。
- 多数のテーブルで削除／更新／挿入アクティビティを大々的に行った場合は、データベースを再構築する。

- 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、テーブル、インデックスの断片化を検出し、修正する方法の詳細については、「断片化」254 ページを参照してください。

オペレーティング・システムのファイルをデフラグする

オペレーティング・システムのファイルをデフラグするには、利用できるディスクのデフラグメンテーション・ユーティリティを定期的に実行します。ファイルの断片化は、パフォーマンスに悪影響を与える場合があります。

Windows NT/2000/XP でデータベースを起動すると、データベース・サーバがデータベース・ファイル内のファイルの断片化数を調べ、断片化の数が 1 より大きい場合、サーバのメッセージ・ウィンドウに次の情報を表示します。

nnn フラグメントで構成されるデータベース・ファイル "mydatabase.db"

また、DBFileFragments データベース・プロパティを使用しても、データベース・ファイルの断片化の数を取得できます。

インデックスの断片化を最小限に抑える

インデックスは特定のカラムの検索が高速になるように設計されていますが、インデックス付きテーブルに対して多数の DELETE が実行されると、断片化することがあります。このため、インデックスが頻繁にアクセスされ、キャッシュが小さいためにインデックスがすべて収まらないと、パフォーマンス低下を招く場合があります。

sa_index_density ストアド・プロシージャは、データベースのインデックスの断片化レベルに関する情報を提供します。このプロシージャを実行するには、DBA 権限が必要です。次の文は、**sa_index_density** ストアド・プロシージャを呼び出します。

```
CALL sa_index_density (['table_name'[,'owner_name']])
```

インデックスが極端に断片化されている場合は、REORGANIZE TABLE を実行します。また、インデックスを削除して再作成する方法もあります。ただし、インデックスがプライマリ・キーの場合は、外部キー・インデックスの削除と再作成も必要になります。

また、テーブルのクラスタード・インデックスを作成してパフォーマンスを改善できます。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。

テーブルの断片化を最小限に抑える

テーブルの断片化は、ローが連続して格納されていない場合、またはローが複数のページに分割されている場合に発生します。ローがこのような状態の場合、ページ・アクセスが増えるため、パフォーマンスが低下します。

Adaptive Server 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、現在確保されたページ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続きのロー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。

sa_table_fragmentation ストアド・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。次の文は、**sa_table_fragmentation** ストアド・プロシージャを呼び出します。

```
CALL sa_table_fragmentation (['table_name' [,'owner_name']])
```

テーブルの断片化を最小限に抑えるには、次の 3 つの方法があります。

- テーブル・ページ内で将来の更新用に確保される領域の割合を指定できます。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。
- 特定のテーブルを再編成できます。特定のテーブルやテーブルの一部をデフラグするには、REORGANIZE TABLE を実行します。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。

- データベース全体を再構築できます。再構築は、システム・テーブルを含め、すべてのテーブルをデフラグする点でより包括的です。さらに、再構築によって、テーブルのローが並び替えられ、クラスタード・インデックスやプライマリ・キーで指定された順番になります。

適切なハードウェアの入手

Adaptive Server Anywhere を PC で実行する場合は、CPU、メモリ、ディスクについて最低要件が満たされていることを確認してください。

- Adaptive Server Anywhere は、最低 4 MB のメモリで実行できます。データベース内で Java を使用する場合、Adaptive Server Anywhere には 8 MB のメモリが必要になります。管理ツールを使用する場合、Adaptive Server Anywhere には RAM が 32 MB 以上必要です。使用するコンピュータには、このメモリ容量に加えて、オペレーティング・システム用のメモリ容量が必要になります。

- データベースとログ・ファイルを保持するのに十分なディスク領域。

- これらは、最低必要容量であることに留意してください。ハードウェア要件を最低限満たしている場合で、パフォーマンスが思わしくない場合は、ハードウェアの一部または全体のアップグレードを検討してください。一般論として、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、『ASA データベース管理ガイド』> 「-fc サーバ・オプション」を参照してください。

オプティマイザの優先度の選択

OPTIMIZATION_GOAL オプションは、Adaptive Server Anywhere において SQL 文を応答時間 (最初のロー) に対して最適化するか、リソースの総消費量 (すべてのロー) に対して最適化するかを制御できます。すなわち、クエリ処理の最適化の目的を、最初のローを迅速に返すことに設定するか、または完全な結果セットを返すコストを最小限に抑えることに設定できます。

このオプションを first-row に設定すると、Adaptive Server 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 Adaptive Server Anywhere オプティマイザでは、可能であれば結果の実体化を必要とするアクセス・プランは使用しないで、最初のローを返すまでの時間を短縮します。この設定では、たとえばオプティマイザは、明示的なソートの操作を必要とするアクセス・プランではなく、クエリの ORDER BY 句を満たすインデックスを使用するアクセス・プランを採用します。

クエリの FROM 句の FASTFIRSTROW テーブル・ヒントを使用すると、特定のクエリの最適化ゴールを first-row に設定できます。この場合、OPTIMIZATION_GOAL 設定を変更する必要はありません。

このオプションを all-rows (デフォルト) に設定すると、Adaptive Server Anywhere はクエリを最適化して、予測される最短の合計検索時間でアクセス・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 に all-rows を設定するのが適切です。

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、検査制約を使用しなくてよいなど、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、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、実

行エンジン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。

小さいテーブルに関する統計収集の検討

Adaptive Server 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。Adaptive Server 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、Adaptive Server 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、Adaptive Server Anywhere によって自動的に管理されます。

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。このような状況がもっとも発生しやすいのは、大量のデータが追加、更新、または削除されて以来実行されたクエリが少ない場合です。統計が不正確な場合、または利用できない場合、パフォーマンスに悪影響を与えます。統計の更新に Adaptive Server 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。また、新しいバージョン (バージョン 9.0 以降) のサーバの場合、Adaptive Server Anywhere は、LOAD TABLE やクエリ実行中だけでなく、UPDATE DML 文でも統計を更新しようとします。

ただし、特異な状況では、この方法では効果的でないことがあります。条件の成功する確率がオプティマイザの推定とは異なることがあらかじめ判明している場合、ユーザ推定を明示的に探索条件として指定できます。

ユーザ定義の推定はパフォーマンスを改善することがありますが、継続的に使用される文にユーザ定義の推定を明示的に指定しないでください。データが変更されると、明示的な推定が不正確になり、オプティマイザが誤って不適切なプランを選択することがあります。

ソフトウェアによって選択されたアクセス・プランが不適切であり、パフォーマンス問題を回避するために選択性推定を使用したものの、それが不正確であった場合は、USER_ESTIMATES を OFF に設定して、値を無視できます。

詳細については、『ASA SQL リファレンス・マニュアル』> 「明示的な選択性推定」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。

パフォーマンス向上へのキャッシュの使用

データベース・キャッシュとは、メモリの領域で、データベース・サーバがデータベース・ページを格納して繰り返し高速にアクセスするために使用します。キャッシュでアクセスできるページが増えると、データベース・サーバがディスクからデータを読み込まなければならない回数が減ります。ディスクからデータを読み込むオペレーションは速度が遅いので、使用できるキャッシュの容量がパフォーマンスを決める重要な要因になることがよくあります。

データベース・キャッシュのサイズは、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・ラインで制御します。

動的キャッシュ・サイズ決定

Adaptive Server Anywhere は、データベース・キャッシュの自動サイズ変更機能を提供します。この機能は、オペレーティング・システムによって異なります。Windows NT/2000/XP、Windows 95/98/Me、UNIX の各オペレーティング・システムでは、キャッシュは増大したり縮小したりします。詳細は、次の項で説明します。

完全な「**動的キャッシュ・サイズ決定**」によって、不適切なメモリ割り付けによるデータベース・サーバのパフォーマンスへの影響を防止できます。データベース・サーバの効率がキャッシュ・サイズを拡大することによって向上する場合は、使用可能なメモリがあるかぎりキャッシュ・サイズが拡大し、他のアプリケーションがキャッシュ・メモリを必要とするときは小さくなります。この処理によって、データベース・サーバはシステム上の他のアプリケーションに過度の影響を与えないで済みます。動的キャッシュ・サイズ決定の効果は、当然、システムで使用できる物理メモリによって限られます。

通常、動的キャッシュ・サイズ決定は、約 1 分間隔でキャッシュの必要量を推定します。ただし、新しいデータベースが起動されたときやファイル量が大幅に増加した場合、5 秒間隔で 30 秒間統計がサンプリングされ、キャッシュのサイズが変更されます。最初の 30 秒間が経過すると、サンプリング率は 1 分間隔に戻ります。データベースの起動後、またはサンプリング率の上昇をもたらした最後のファイル増加以降に、ファイルが 8 分の 1 増加した場合、大幅な増加が生じたと見なされます。サンプリング率の変化により、データベースが動的に起動された場合、または大量のデータが挿入された場合に、ただちにキャッシュ・サイズが変更され、パフォーマンスがさらに向上します。

動的キャッシュ・サイズ決定は、さまざまな状況でデータベース・キャッシュを明示的に設定する必要をなくし、Adaptive Server Anywhere をより使いやすくします。

Windows CE、Novell NetWare には、動的キャッシュ・サイズ変更機能はありません。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。

AWE キャッシュの詳細については、『ASA データベース管理ガイド』 > 「-cw サーバ・オプション」を参照してください。

キャッシュが使用するメモリの制限

キャッシュの初期サイズ、最小サイズ、最大サイズはすべて、データベース・サーバのコマンド・ラインから制御できます。

- **初期キャッシュ・サイズ** 初期キャッシュ・サイズを変更するには、データベース・サーバの -c コマンド・ライン・オプションを指定します。デフォルトの値は次のとおりです。

  - **Windows CE** 式は次のとおりです。

    ```
    max( 600K, 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、マシンの物理メモリの 25% です。

  - **Windows NT/2000/XP、Windows 95/98/Me、NetWare** 式は次のとおりです。

    ```
    max( 2M, 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、マシンの物理メモリの 25% です。

    Windows 2000、Windows XP、または Windows Server 2003 で AWE キャッシュが使用されている場合、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、『ASA データベース管理ガイド』> 「-cw サーバ・オプション」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)」234 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 コマンド・ライン・オプションを指定します。デフォルトは、使用しているマシンの物理メモリによって異なるヒューリスティックに基いています。UNIX 以外のマシンでは、これは通常、物理メモリ量の合計の約 90% で、256 MB 以下です。UNIX マシンでは、次に示す容量より大きくなります。

```
(25% total physical-memory,
    25%(available-physical-memory + available-swap-
space),
    sum-of-file-sizes-on-the-command-line)
```

  ただし、256 MB 以下で、24 MB 以上です。

- **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 コマンド・ライン・オプションを指定します。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。

また、動的キャッシュ・サイズ決定を無効にするには、-ca コマンド・ライン・オプションを使用します。

コマンド・ライン・オプションの詳細については、『ASA データベース管理ガイド』> 「データベース・サーバ」を参照してください。

動的キャッシュ・サイズ決定 (Windows NT/2000/XP、Windows 95/98/Me)

Windows NT/2000/XP と Windows 95/98/Me 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。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 2000、Windows XP、Windows Server 2003 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。

詳細については、『ASA データベース管理ガイド』> 「-cw サーバ・オプション」を参照してください。

動的キャッシュ・サイズ決定 (UNIX)

UNIX では、データベース・サーバはスワップ領域とメモリを使用してキャッシュ・サイズを管理します。スワップ領域はシステムワイドなリソースで、たいていの UNIX オペレーティング・システムにありますが、ないものもあります。この項では、メモリとスワップ領域の合計を「**システム・リソース**」と呼びます。詳細については、使用しているオペレーティング・システムのマニュアルを参照してください。

起動時に、データベースは指定した最大キャッシュ・サイズをシステム・リソースから割り付けます。この一部をメモリ (初期キャッシュ・サイズ) にロードし、残りをスワップ領域として残します。

データベース・サーバが使用するシステム・リソースの総量は、データベース・サーバが停止するまで一定です。ただし、メモリにロードされる比率は変わります。データベースサーバは、1 分ごとにキャッシュとオペレーションの統計を評価します。データベース・サーバがビジーでメモリが必要になると、キャッシュ・ページをスワップ領域からメモリに移します。システム内の他の処理がメモリを必要とした場合、データベース・サーバがキャッシュ・ページをメモリからスワップ領域に移すことがあります。

初期キャッシュ・サイズ

デフォルトでは、初期キャッシュ・サイズは使用可能なシステム・リソースに基づいたヒューリスティックを使用して割り当てられます。初期キャッシュ・サイズは、データベース・サイズの総量の 1.1 倍より常に小さくなります。

初期キャッシュ・サイズが使用可能なシステム・リソースの 4 分の 3 より大きい場合は、データベース・サーバが「メモリ不足」エラーで終了します。

初期キャッシュ・サイズは、-c オプションを使用して変更できます。

最大キャッシュ・サイズ

最大キャッシュ・サイズは、マシン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・サイズはマシン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、サーバが「メモリ不足」エラーで終了します。使用可能なメモリより大きい最大キャッシュ・サイズを指定すると、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 Adaptive Server 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。

8 MB より小さい最大キャッシュ・サイズを指定すると、Java アプリケーションを実行できなくなります。最大キャッシュ・サイズを小さくすると、パフォーマンスに影響を与えます。

`-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、『ASA データベース管理ガイド』> 「-ch サーバ・オプション」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。

詳細については、『ASA データベース管理ガイド』> 「-cl サーバ・オプション」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加算されました。

- **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 NT、Windows 2000、Windows XP で使用できます。

これらのプロパティの詳細については、『ASA データベース管理ガイド』> 「サーバ・レベルのプロパティ」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」246 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・サーバはデータベース起動時に要求されたすべてのデータベース・ページを、収集ページ数が最大数に到達するまで (値はキャッシュ・サイズとデータベース・サイズに基づく)、収集速度が最小閾値を下回るまで、またはデータベースが停止するまで、追跡し続けます。サーバが収集最大ページ数と収集閾値を制御することに注意してください。いったん収集が完了すると、参照されたページはデータベースに記録されるので、次回のデータベース起動時にキャッシュの準備に使用できます。

キャッシュ・ウォーミング (再ロード) はデフォルトでオンになっていて、-cr データベース・サーバ・オプションで制御されます。
キャッシュを準備するために、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、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・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、『ASA データベース管理ガイド』> 「-cc サーバ・オプション」、『ASA データベース管理ガイド』> 「-cr サーバ・オプション」、『ASA データベース管理ガイド』> 「-cv サーバ・オプション」を参照してください。

インデックスの使用によるパフォーマンス改善

インデックスを適切に選択すると、パフォーマンスに大きな違いが生じます。インデックスの作成と管理の詳細については、「インデックスの編集」79 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 employee
WHERE emp_id = 390
```

サーバがこのクエリを実行する一番簡単な方法は、**employee** テーブルの 75 のローすべてを調べ、各ローの employee ID 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、Adaptive Server Anywhere によって確保されます。emp_id カラムは employee 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、employee テーブルのローの数が 100 の場合も、1,000,000 の場合もほとんど同じ時間で行うことができます。

プライマリ・キーを使ったクエリのパフォーマンス改善

次は、プライマリ・キーを使用してパフォーマンスを改善する文です。

```
SELECT *
FROM employee
WHERE emp_id = 390
```

[プラン] タブの情報

[結果] ウィンドウ枠の [プラン] タブには、次の情報が表示されます。

```
employee<employee>
```

[プラン] タブの PLAN 記述のカッコ内の名前がテーブル名と同じである場合は、パフォーマンス改善のためにそのテーブルのプライマリ・キーが使われています。

外部キーを使ったクエリのパフォーマンス改善

次に示すクエリは、顧客 ID 113 を持つ顧客からの注文をリストします。

```
SELECT *
FROM sales_order
WHERE cust_id = 113
```

[プラン] タブの情報

[結果] ウィンドウ枠の [プラン] タブには、次の情報が表示されます。

```
sales_order <ky_so_customer>
```

ky_so_customer は、sales_order テーブルが customer テーブルに対して持つ外部キーです。

別個のプライマリ・キー・インデックスと外部キー・インデックス

プライマリ・キーと外部キーについて、インデックスが個別に自動的に作成されます。これによって、Adaptive Server Anywhere で多数のオペレーションを効率的に実行できます。この機能は、バージョン 7.0 で導入されたものです。

クエリ結果のソート

ローが予測可能な順序で表示されるように、多くのクエリには ORDER BY 句があります。インデックスは、情報をすばやく順序付けます。たとえば、次のクエリでは、customer テーブルの lname カラムにあるインデックスを使って、customer テーブルのローに姓のアルファベット順にアクセスします。

```
SELECT *
 FROM customer
 ORDER BY customer.lname
```

WHERE 句と ORDER BY 句を使ったクエリ

WHERE 句と ORDER BY 句が両方存在するときは、問題が起こる可能性があります。

```
SELECT *
FROM customer
WHERE id > 300
ORDER BY company_name
```

Adaptive Server Anywhere は 2 つの方式のうち 1 つを選択します。

1. customer テーブル全体を、company_name の順序で検索し、各ローの customer_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。

値が 300 を超える id がほとんどない場合は、2 番目の方式の方が適切です。スキャンする必要のあるローの数が少なく、ソートに時間がかからないためです。id 値の大半が 300 以上である場合は、ソートを必要としない最初の方式の方が適切です。

ソートの詳細については、「ORDER BY 句：クエリ結果のソート」320 ページまたは 「GROUP BY 句：クエリ結果のグループへの編成」312 ページを参照してください。

クエリ処理中のワーク・テーブルの使用

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 Adaptive Server 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、Adaptive Server Anywhere は OPTIMIZATION_GOAL の設定に基づいてさまざまな方式を選択します。デフォルトはすべてのローです。OPTIMIZATION_GOAL が最初のローに設定されている場合、Adaptive Server Anywhere はワーク・テーブルを使用しないようにします。すべてのローに設定されている場合、クエリの合計実行コストが減少するなら、ワーク・テーブルが使用されます。

ワーク・テーブルが使用されるのは、次の場合です。

ワーク・テーブルが必要なとき

- クエリに ORDER BY 句、GROUP BY 句、または DISTINCT 句があり、Adaptive Server 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 が最初のローに設定されている場合、Adaptive Server Anywhere はワーク・テーブルを使用しません。ただし、OPTIMIZATION_GOAL がすべてのローに設定されている場合は、ワーク・テーブルを作成してローをソートするよりも、インデックスを使用してクエリのローをすべてフェッチする方が高コストになることがあります。最適化ゴールがすべてのローに設定されている場合、Adaptive Server 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、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るとき。

- sensitive 値を使用してカーソルが開かれ、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るとき。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、クエリが開かれるときに、ワーク・テーブルにクエリの結果が移植されるとき。
- 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。

データベース・サーバのクエリ・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のパフォーマンスが得られるかどうかを決定します。Adaptive Server Anywhere の新しいリリースでは、オプティマイザが強化されており、クエリへのアクセス・プランが改善されます。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。

データベースのパフォーマンスのモニタリング

Adaptive Server 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。それらには Sybase Central からアクセスでき、クライアント・アプリケーションからは関数として呼び出せます。また、サーバはこれらの統計を Windows パフォーマンス モニタで使用できるようにします。

この項では、クライアント・アプリケーションからパフォーマンスと関連の統計値にアクセスする方法、Sybase Central を使ってデータベースのパフォーマンスをモニタする方法、Windows パフォーマンス モニタを使ってデータベースのパフォーマンスをモニタする方法、ファイル、テーブル、インデックスの断片化を検出する方法について説明します。

クライアント・アプリケーションからのデータベース統計値の取得

Adaptive Server 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、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**　エンジンごとに指定されたプロパティの値を返します。
- **connection_property 関数**　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。
- **db_property 関数**　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。

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。

詳細については、『ASA SQL リファレンス・マニュアル』> 「PROPERTY 関数 [システム]」、『ASA SQL リファレンス・マニュアル』> 「CONNECTION_PROPERTY 関数 [システム]」、『ASA SQL リファレンス・マニュアル』> 「DB_PROPERTY 関数 [システム]」を参照してください。

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、『ASA SQL リファレンス・マニュアル』> 「システム関数」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。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 = property( propnum )
```

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。

これらの統計値の多くは、Sybase Central のデータベース管理ツールを使って、グラフ形式で閲覧できます。

Sybase Central からのデータベース統計値のモニタリング

Sybase Central パフォーマンス・モニタを使用すると、接続できる Adaptive Server Anywhere データベース・サーバの統計値を、Sybase Central でグラフ形式で表示できます。Sybase Central の統計値はすべて、[統計情報] フォルダに表示されます。

次に、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・モニタを使用しているときは、パフォーマンス・モニタがサーバに対して実際のクエリを使用して統計値を収集していることに注意してください。つまり、モニタ処理自体が統計値の一部 (秒あたりのキャッシュ読み込みなど) に影響を与えているのです。サーバのより精密な統計値をグラフにする方法として、Windows パフォーマンス モニタを使用することもできます。

プロパティの設定については、「データベース・オブジェクトのプロパティの設定」44 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

パフォーマンス・モニタは、[統計情報] フォルダを開いている Sybase Central の右ウィンドウ枠に表示できます。

❖ パフォーマンス・モニタを開くには、次の手順に従います。

1. 左ウィンドウ枠で対象のサーバを選択します。
2. 右ウィンドウ枠で、[パフォーマンス・モニタ] タブをクリックします。

注意
パフォーマンス・モニタがグラフにするのは、前もって追加された統計値のみです。

参照

- 「統計値の追加と削除」249 ページ
- 「Sybase Central パフォーマンス・モニタの設定」250 ページ
- 「Windows パフォーマンス モニタからのデータベース統計値のモニタリング」251 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. 左ウィンドウ枠で対象のサーバを選択します。
2. 右ウィンドウ枠で、[統計情報] タブをクリックします。
3. グラフ化されていない統計値を右クリックし、ポップアップ・メニューで [パフォーマンス・モニタに追加] を選択します。

❖ 統計値を Sybase Central パフォーマンス・モニタから削除するには、次の手順に従います。

1. 次のいずれかを行います。
   - 右ウィンドウ枠の [統計情報] タブで作業中の場合は、現在グラフ化されている統計値を右クリックします。
   - 右ウィンドウ枠の [パフォーマンス・モニタ] タブで作業中の場合は、削除する統計値を凡例内で右クリックします。

2 ポップアップ・メニューで、[パフォーマンス・モニタから削除] を選択します。

ヒント
統計値の追加と削除は、統計値のプロパティ・シートにある [パフォーマンス・モニタ] からもできます。

参照

- 「Sybase Central パフォーマンス・モニタを開く」248 ページ
- 「Sybase Central パフォーマンス・モニタの設定」250 ページ
- 「Windows パフォーマンス モニタからのデータベース統計値のモニタリング」251 ページ

Sybase Central パフォーマンス・モニタの設定

Sybase Central パフォーマンス・モニタは設定可能です。使用するグラフの種類とグラフの更新の間隔を選択できます。

❖ グラフの種類を選択するには、次の手順に従います。

1 [ツール] － [オプション] を選択します。

2 [オプション] ダイアログで [チャート] タブをクリックします。

3 グラフの種類を選択します。

❖ 更新間隔を設定するには、次の手順に従います。

1 [ツール] － [オプション] を選択します。

2 [オプション] ダイアログで [チャート] タブをクリックします。

3 スライダを動かして新しい時間の値を反映させます。または、用意されたテキスト・ボックスに値を直接入力します。

参照

- 「Sybase Central パフォーマンス・モニタを開く」248 ページ
- 「統計値の追加と削除」249 ページ
- 「Windows パフォーマンス モニタからのデータベース統計値のモニタリング」251 ページ

Windows パフォーマンス モニタからのデータベース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 版モニタには 2 つの利点があります。

- 主にネットワーク通信に関わるパフォーマンス統計値をより多く提供します。
- Sybase Central 版モニタとは異なり、Windows 版モニタは非介入です。Windows 版モニタは、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。

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、『ASA データベース管理ガイド』> 「パフォーマンス・モニタの統計値」を参照してください。

注意：
Windows パフォーマンス モニタは、Windows NT、Windows 2000、Windows XP で使用できます。複数のバージョンの Adaptive Server Anywhere を同時に実行する場合は、複数のバージョンのパフォーマンス・モニタも同時に実行できます。

❖ Windows NT 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 Adaptive Server Anywhere のエンジンまたはデータベースを実行し、パフォーマンス モニタを起動します。

- [スタート] － [プログラム] － [管理ツール (共通)] － [パフォーマンス モニタ] を選択します。

2 [編集] － [グラフに追加] を選択するか、ツールバーの [+] ボタンをクリックします。[グラフに追加] ダイアログが表示されます。

3 [オブジェクト] リストから、次のいずれかを選択します。

- **[Adaptive Server Anywhere 接続]** 単一の接続のパフォーマンスをモニタする場合に選択します。モニタする接続を表示されたリストから選択します。
- **[Adaptive Server Anywhere データベース]** 単一のデータベースのパフォーマンスをモニタする場合に選択します。モニタするデータベースを表示されたリストから選択します。
- **[Adaptive Server Anywhere エンジン]** サーバワイドでパフォーマンスをモニタする場合に選択します。

[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。

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 [Adaptive Server Anywhere 接続] または [Adaptive Server Anywhere データベース] を選択した場合は、[インスタンス] ボックスからインスタンスを選択します。

6 選択したカウンタの説明を表示するには、[説明] をクリックします。

7 カウンタを表示するには、[追加] をクリックします。

8　表示したいカウンタをすべて選択したら [完了] をクリックします。

Windows パフォーマンス モニタの詳細については、プログラムのオンライン・ヘルプを参照してください。

断片化

データベースに変更を加えるにつれて、データベースのファイル、テーブル、インデックスが断片化することがあります。断片化によってパフォーマンスが低下する可能性があります。Adaptive Server Anywhere では、ファイル、テーブル、インデックスの断片化レベルを査定できるように情報が提供されます。

この項では、ファイル、テーブル、インデックスの断片化を検出する方法と、それらをデフラグする方法について説明します。

ファイルの断片化

データベース・ファイルが断片化しすぎていると、パフォーマンスに影響が出ます。これはディスク断片化であり、データベースが大きくなるにつれ、重要になります。

Windows NT/2000/XP 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、サーバのメッセージ・ウィンドウに次の情報が表示されます。

`nnn` フラグメントで構成される `mydatabase.db` データベース・ファイル

また、DBFileFragments データベース・プロパティを使用しても、データベース・ファイルの断片化の数を取得できます。

詳細については、『ASA データベース管理ガイド』> 「データベース・レベルのプロパティ」を参照してください。

❖ ファイルをデフラグするには、次の手順に従います。

1. データベースをディスク・パーティションに単独で入れます。
2. 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

ローが連続して格納されていない場合や、複数ページに分割されている場合は、パフォーマンスが低下します。これは、この種のローがあるためにページへのアクセスが増えるためです。テーブルの断片化は、ファイルの断片化とは異なります。

Adaptive Server 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。

詳細については、『ASA SQL リファレンス・マニュアル』> 「CREATE TABLE 文」、『ASA SQL リファレンス・マニュアル』> 「ALTER TABLE 文」、『ASA SQL リファレンス・マニュアル』> 「DECLARE LOCAL TEMPORARY TABLE 文」、『ASA SQL リファレンス・マニュアル』> 「LOAD TABLE 文」を参照してください。

sa_table_fragmentation ストアド・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。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 ストアド・プロシージャを呼び出します。

```
CALL sa_table_fragmentation (['table_name'
[,'owner_name']])
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「sa_table_fragmentation システム・プロシージャ」を参照してください。

テーブルのデフラグメンテーション

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、次のプロシージャが役立ちます。データベースをアンロードして再ロードする方が、システム・テーブルな

ど、すべてのテーブルの断片化ををデフラグする点で包括的です。特定のテーブルやテーブルの一部の断片化ををデフラグするには、REORGANIZE TABLE を実行します。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。

❖ データベース内のすべてのテーブルをデフラグするには、次の手順に従います。

1. データベースをアンロードします。
2. データベースを再ロードし、ディスク領域を再利用してパフォーマンスを向上させます。

データベースのアンロードの詳細については、『ASA データベース管理ガイド』> 「dbunload コマンド・ライン・ユーティリティを使用したデータベースのアンロード」を参照してください。

データベースの再構築の詳細については、『ASA データベース管理ガイド』> 「再構築ユーティリティ」を参照してください。

❖ 個々のテーブルをデフラグするには、次の手順に従います。

- REORGANIZE TABLE 文を実行します。

詳細については、『ASA SQL リファレンス・マニュアル』> 「REORGANIZE TABLE 文」を参照してください。

インデックスの断片化

インデックスは特定のカラムの検索が高速になるように設計されていますが、インデックス付きテーブルに対して多数の DELETE が実行されると、断片化することがあります。このため、インデックスが頻繁にアクセスされ、キャッシュが小さいためにインデックスがすべて収まらないと、パフォーマンス低下を招く場合があります。

sa_index_density ストアド・プロシージャは、データベースのインデックスの断片化レベルに関する情報を提供します。このプロシージャを実行するには、DBA 権限が必要です。次の文は、sa_index_density ストアド・プロシージャを呼び出します。

```
CALL sa_index_density (['table_name'[,'owner_name']])
```

インデックスが極端に断片化されている場合は、REORGANIZE TABLE を実行します。また、インデックスを削除して再作成する方法もあります。ただし、インデックスがプライマリ・キーの場合は、外部キー・インデックスの削除と再作成も必要になります。

詳細については、『ASA SQL リファレンス・マニュアル』>「REORGANIZE TABLE 文」を参照してください。

インデックスの削除の詳細については、「インデックスの削除」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リング

Adaptive Server Anywhere には、クエリのパフォーマンスをテストできるように多数のツールが用意されています。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。

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： *SQL Anywhere 9¥Samples¥Asa¥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。この機能は fetchtst に似ていますがより限定的です。

ロケーション： *SQL Anywhere 9¥Samples¥Asa¥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： *SQL Anywhere 9¥Samples¥Asa¥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： *SQL Anywhere 9¥Samples¥Asa¥PerformanceTransaction*

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、『ASA SQL リファレンス・マニュアル』> 「sa_get_request_profile システム・プロシージャ」と 『ASA SQL リファレンス・マニュアル』> 「sa_get_request_times システム・プロシージャ」を参照してください。

データベース・プロシージャのプロファイリング

プロシージャ・プロファイリングは、ストアド・プロシージャ、関数、イベント、システム・トリガ、トリガの実行所要時間を示します。また、プロシージャの各行の実行時間も表示できます。データベース・プロファイリング情報を使用すると、どのプロシージャ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。

プロファイリングが有効になっている場合、Adaptive Server Anywhere は、どのストアド・プロシージャ、関数、イベント、システム・トリガ、トリガが使用されるかをモニタし、その実行所要時間とそれぞれの呼び出し回数を追跡します。

プロファイリング情報はサーバによってメモリに格納され、Sybase Central の [プロファイル] タブまたは Interactive SQL で表示できます。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、サーバが停止されるまで、データベースはプロファイリング情報を収集します。

Interactive SQL でプロファイリング情報を取得する方法の詳細については、「Interactive SQL でのプロシージャ・プロファイリング情報の表示」268 ページを参照してください。

プロシージャ・プロファイリングの有効化

プロシージャ・プロファイリングは、すべての接続ごとにプロシージャとトリガの使用状況を追跡します。プロファイリングを有効にするには、Sybase Central または Interactive SQL を使用します。プロシージャ・プロファイリングを有効にして使用するには、DBA 権限が必要です。

❖ **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. 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. 左ウィンドウ枠からデータベースを選択します。

3 [ファイル] － [プロパティ] を選択します。

[データベース] プロパティ・シートが表示されます。

4 [プロファイリング] タブで、[このデータベースでプロファイリングを可能にする] をオンにします。

5 [OK] をクリックしてプロパティ・シートを閉じます。

注意

プロファイリングを有効にするには、Sybase Central でデータベースを右クリックする方法もあります。その場合は、表示されるポップアップ・メニューで [プロファイリング] － [プロファイリングの開始] を選択します。

❖ 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 (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 ON 設定を指定して sa_server_option ストアド・プロシージャを呼び出します。

たとえば、次のように入力します。

```
CALL sa_server_option ( 'procedure_profiling', '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。

❖ ユーザごとにプロシージャ・プロファイリングをフィルタするには、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 次の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。

プロシージャ・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。

次の項の説明は、DBA 権限を持つ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。

❖ 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 (Sybase Central の場合)。

1. 左ウィンドウ枠からデータベースを選択します。
2. [ファイル] － [プロパティ] を選択します。

   [データベース] プロパティ・シートが表示されます。
3. [プロファイリング] タブで、[すぐにリセット] をクリックします。
4. [OK] をクリックしてプロパティ・シートを閉じます。

注意

プロファイリングをリセットするには、Sybase Central でデータベースを右クリックする方法もあります。その場合は、表示されるポップアップ・メニューで [プロファイリング] － [プロファイリング情報のリセット] をクリックします。

❖ 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 (SQL の場合)。

- RESET 設定を指定して sa_server_option ストアド・プロシージャを呼び出します。

  たとえば、次のように入力します。

```
CALL sa_server_option ('procedure_profiling',
'RESET')
```

プロシージャ・プロファイリングの無効化

プロファイリング情報の確認後に、プロファイリングを無効にするか、クリアできます。プロファイリングを無効にすると、データベースはプロファイリング情報の収集を停止します。ただし、その時点までに収集された情報は、Sybase Central の [プロファイル] タブに残ります。プロファイリングをクリアすると、データベースはプロファイリングをオフにして、Sybase Central の [プロファイル] タブからプロファイリング・データをすべて削除します。

❖ 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 (Sybase Central の場合)。

1. 左ウィンドウ枠からデータベースを選択します。
2. [ファイル] － [プロパティ] を選択します。

   [データベース] プロパティ・シートが表示されます。
3. [プロファイリング] タブで、[このデータベースでプロファイリングを可能にする] オプションをオフにします。
4. [OK] をクリックしてプロパティ・シートを閉じます。

注意

プロファイリングを無効にするには、Sybase Central でデータベースを右クリックする方法もあります。その場合は、表示されるポップアップ・メニューで [プロファイリング] － [プロファイリングの停止] を選択します。

❖ 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 (SQL の場合)。

- OFF 設定を指定して sa_server_option ストアド・プロシージャを呼び出します。

  たとえば、次のように入力します。

```
CALL sa_server_option ('procedure_profiling', 'OFF')
```

❖ **プロファイリングをクリアするには、次の手順に従います (Sybase Central の場合)。**

1. 左ウィンドウ枠からデータベースを選択します。
2. [ファイル] － [プロパティ] を選択します。

   [データベース] プロパティ・シートが表示されます。
3. [プロファイリング] タブで、[すぐにクリア] をクリックします。

   有効になっているプロファイリングは、クリアしかできません。
4. [OK] をクリックしてプロパティ・シートを閉じます。

注意

プロファイリングをクリアにするには、Sybase Central でデータベースを右クリックする方法もあります。その場合は、表示されるポップアップ・メニューで [プロファイリング] － [プロファイリング情報のクリア] を選択します。

❖ **プロファイリングをクリアするには、次の手順に従います (SQL の場合)。**

- CLEAR 設定を指定して sa_server_option ストアド・プロシージャを呼び出します。

  たとえば、次のように入力します。

```
CALL sa_server_option ('procedure_profiling',
'CLEAR')
```

Sybase Central でのプロシージャ・プロファイリング情報の表示

プロシージャ・プロファイリングでは、データベース全体、特定タイプのオブジェクト、特定のプロシージャのうち、どれを対象として選択するかに応じて異なる情報が得られます。情報は次の方法で表示できます。

- データベース内でプロファイル対象となる全オブジェクトの詳細
- すべてのストアド・プロシージャと関数の詳細
- すべてのイベントの詳細
- すべてのトリガの詳細
- すべてのシステム・トリガの詳細
- プロファイル対象となるオブジェクトの個別の詳細

プロファイリング情報を表示するには、データベースに接続してプロファイリングを有効にします。

データベース全体のプロファイリング情報を表示すると、次のカラムが表示されます。

- **Name**　オブジェクト名。
- **Owner**　オブジェクトの所有者。
- **Table**　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ます)。
- **Event**　システム・トリガのタイプを表示します。タイプは **Update** または **Delete** です。
- **Type**　オブジェクト・タイプ。たとえば、プロシージャ。
- **# Exes.**　各オブジェクトが呼び出された回数。
- **#msecs.**　各オブジェクトの合計実行時間。

これらのカラムには、データベース内で実行されたプロシージャすべてのプロファイリング情報の概要が表示されます。1 つのプロシージャが他のプロシージャを呼び出すことができるため、ユーザが呼び出したより多くの項目がリストされる場合があります。

❖ ストアド・プロシージャと関数の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。

1 左ウィンドウ枠で [プロシージャとファンクション] フォルダを選択します。

2 右ウィンドウ枠の [プロファイル] タブをクリックします。

データベースにあるストアド・プロシージャと関数すべてのプロファイリング情報が、[プロファイル] タブに表示されます。

❖ イベントの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。

1 左ウィンドウ枠で [イベント] フォルダを開きます。

データベースにあるイベントすべてのリストが、右ウィンドウ枠の [イベント] タブに表示されます。

2 右ウィンドウ枠の [プロファイル] タブをクリックします。

データベースにあるイベントすべてのプロファイリング情報が、[プロファイル] タブに表示されます。

❖ トリガの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。

1 左ウィンドウ枠で [トリガ] フォルダを開きます。

データベースのトリガすべてのリストが、[トリガ] タブに表示されます。

2 右ウィンドウ枠の [プロファイル] タブをクリックします。

データベースにあるトリガすべてのプロファイリング情報が、[プロファイル] タブに表示されます。

❖ システム・トリガの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。

1 左ウィンドウ枠で [システム・トリガ] フォルダを開きます。

データベースのトリガすべてのリストが、[システム・トリガ] タブに表示されます。

2 右ウィンドウ枠の [プロファイル] タブをクリックします。

データベースにあるシステム・トリガすべてのプロファイリング情報が、[プロファイル] タブに表示されます。

特定のプロシージャに関するプロファイリング情報の表示

Adaptive Server Anywhere では、個々のストアド・プロシージャ、関数、イベント、トリガに関するプロシージャ・プロファイリング情報が提供されます。Sybase Central では、個々のプロシージャについてさまざまな情報が表示されます。データベース内のストアド・プロシージャ、関数、イベント、トリガに関する情報は表示されません。

特定のプロシージャに関するプロファイリング情報を調べる場合は、次のカラムが表示されます。

- **Calls** オブジェクトが呼び出された回数。
- **Milliseconds** 各オブジェクトの合計実行時間。
- **Line** プロシージャの各行の隣の行番号。
- **Source** 1 行ずつの SQL プロシージャ。

プロシージャは 1 行ずつに分割されます。どの行の実行時間が最も長く、また変更することでプロシージャのパフォーマンスを向上できるかを調べることができます。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。

❖ ストアド・プロシージャまたは関数のプロファイリング情報を表示するには、次の手順に従います。

1 左ウィンドウ枠でデータベースを拡張します。

2 左ウィンドウ枠で [プロシージャとファンクション] フォルダを選択します。

データベースにあるストアド・プロシージャと関数すべてのリストが、右ウィンドウ枠の [プロシージャとファンクション] タブに表示されます。

3 左ウィンドウ枠で、プロファイル情報を取得するストアド・プロシージャまたは関数をクリックします。

4 右ウィンドウ枠の [プロファイル] タブをクリックします。

特定のストアド・プロシージャまたは関数に関するプロファイリング情報が、右ウィンドウ枠の [プロファイル] タブに表示されます。

❖ イベントのプロファイリング情報を表示するには、次の手順に従います。

1 左ウィンドウ枠でデータベースを拡張します。

2 左ウィンドウ枠で [イベント] フォルダを選択します。

データベースにあるイベントすべてのリストが、右ウィンドウ枠の [イベント] タブに表示されます。

3 左ウィンドウ枠で、プロファイル情報を取得するイベントをクリックします。

4 右ウィンドウ枠の [プロファイル] タブをクリックします。

特定のイベントに関するプロファイリング情報が、右ウィンドウ枠の [プロファイル] タブに表示されます。

❖ トリガのプロファイリング情報を表示するには、次の手順に従います。

1 左ウィンドウ枠でデータベースを拡張します。

2 左ウィンドウ枠で [トリガ] フォルダを開きます。

テーブルにあるトリガすべてのリストが、右ウィンドウ枠の [トリガ] タブに表示されます。

3 右ウィンドウ枠で、プロファイル情報を取得するトリガを選択します。

4 右ウィンドウ枠の [プロファイル] タブをクリックします。

特定のトリガに関するプロファイリング情報が、右ウィンドウ枠の [プロファイル] タブに表示されます。

❖ システム・トリガのプロファイリング情報を表示するには、次の手順に従います。

1 左ウィンドウ枠でデータベースを拡張します。

2 左ウィンドウ枠で [システム・トリガ] フォルダを開きます。

テーブルにあるシステム・トリガすべてのリストが、右ウィンドウ枠の [トリガ] タブに表示されます。

3 右ウィンドウ枠で、プロファイル情報を取得するシステム・トリガを選択します。

4 右ウィンドウ枠の [プロファイル] タブをクリックします。

特定のシステム・トリガに関するプロファイリング情報が、右ウィンドウ枠の [プロファイル] タブに表示されます。

Interactive SQL でのプロシージャ・プロファイリング情報の表示

ストアド・プロシージャを使用して、プロシージャ・プロファイリング情報を表示できます。Sybase Central と Interactive SQL のどちらを使用した場合も、同じプロファイリング情報が表示されます

データベース内のすべてのプロシージャに関する情報を取得するには、sa_procedure_profile_summary ストアド・プロシージャを使用します。このプロシージャを使用すると、ストアド・プロシージャ、関数、イベント、システム・トリガ、トリガのプロファイリング・データを同じ結果セット内で表示できます。次のパラメータを使用して、プロシージャから返されるローを制限します。

- **p_object_name**　プロファイル情報を取得するオブジェクトの名前を指定します。
- **p_owner_name**　プロファイル情報を取得するオブジェクトの所有者を指定します。
- **p_table_name**　トリガのプロファイル情報を取得するテーブルを指定します。
- **p_object_type**　プロファイル情報を取得するオブジェクトのタイプを指定します。次の 5 つのオプションから選択できます。次の値の 1 つを選択すると、結果セットは指定したタイプのオブジェクトだけに限定されます。
  - **P**　ストアド・プロシージャ
  - **F**　関数
  - **T**　トリガ
  - **E**　イベント
  - **S**　システム・トリガ
- **p_ordering**　結果セットのソート順を指定します。

1 つのプロシージャが別のプロシージャを呼び出すことができるため、指定したよりも多数の項目がリストされることがあるので注意してください。

次の項の説明は、DBA 権限を持つ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。

❖ すべてのプロシージャの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。

1. sa_procedure_profile_summary ストアド・プロシージャを実行します。

   たとえば、次のように入力します。

   ```
   CALL sa_procedure_profile_summary
   ```

2. [SQL] － [実行] を選択します。

   データベースにあるプロシージャすべての情報を含む結果セットが、[結果] ウィンドウ枠に表示されます。

sa_procedure_profile_summary ストアド・プロシージャの詳細については、『ASA SQL リファレンス・マニュアル』>「sa_procedure_profile_summary システム・プロシージャ」を参照してください。

Interactive SQL を使用した特定のプロシージャに関するプロファイリング情報の表示

特定のプロシージャの各行に関する情報を取得するには、sa_procedure_profile ストアド・プロシージャを使用します。結果セットには、プロシージャ内の行の行番号、実行時間、合計実行時間に対する割合が含まれます。次のパラメータを使用すると、プロシージャから返されるローを制限できます。

- **p_object_name** プロファイル情報を取得するオブジェクトの名前を指定します。
- **p_owner_name** プロファイル情報を取得するオブジェクトの所有者を指定します。
- **p_table_name** トリガのプロファイル情報を取得するテーブルを指定します。

クエリにパラメータを指定しない場合、プロシージャは呼び出されたことのあるプロシージャすべてのプロファイリング情報を返します。

❖ プロシージャの特定行のプロファイリング情報を表示するには、次の手順に従います。

1. sa_procedure_profile ストアド・プロシージャを実行します。

   たとえば、次のように入力します。

   ```
   CALL sa_procedure_profile
   ```

2. [SQL] － [実行] を選択します。

   個々のプロシージャ行に関する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。

sa_procedure_profile ストアド・プロシージャの詳細については、『ASA SQL リファレンス・マニュアル』> 「sa_procedure_profile システム・プロシージャ」を参照してください。

第 2 部　データのクエリと変更

第 2 部では、データのクエリ方法と変更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。

第 6 章

クエリ：テーブルからのデータの選択

この章の内容

SELECT 文は、データベースからデータを取り出します。1 つ以上のテーブルでローのサブセットを検索するときと、1 つ以上のテーブルでカラムのサブセットを検索するときに使用できます。

この章では、単一テーブルの SELECT 文の基本的な使い方を中心に説明します。SELECT の高度な使用方法は、後の章で説明します。

クエリの概要

クエリはデータベースにデータを要求し、結果を受け取ります。この処理はデータ検索とも呼ばれます。SQL クエリはすべて、SELECT 文を使用して表現されます。

句で構成されているクエリ

SELECT 文は複数の句から成り立っています。次の SELECT 構文では、各行が別々の句です。ここではごく一般的な句を示します。

SELECT *select-list*
[**FROM** *table-expression*]
[**WHERE** *search-condition*]
[**GROUP BY** *column-name*]
[**HAVING** *search-condition*]
[**ORDER BY** { *expression* | *integer* }]

次に、SELECT 文の各句について説明します。

- この句は、SELECT 文にのみ必須です。SELECT 句は、SELECT 文で必須の句です。
- FROM 句は、どのテーブルからカラムを検索するかを指定します。この句は、テーブルからデータを取り出すすべてのクエリに必要です。FROM 句を持たない SELECT 文には別の意味がありますが、それについてはこの章では説明しません。

  ほとんどのクエリはテーブルを操作しますが、クエリを使用して、ビュー、他のクエリ (抽出テーブル)、ストアド・プロシージャの結果セットなど、カラムとローを持つ他のオブジェクトからデータを取り出せます。詳細については、『ASA SQL リファレンス・マニュアル』> 「FROM 句」を参照してください。
- ON 句は、FROM 句の中のテーブルを結合する方法を指定します。ON 句は、複数テーブルのクエリにしか使用しないので、この章では説明しません。

  複数テーブルのクエリについては、「ジョイン：複数テーブルからのデータ検索」335 ページ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」305 ページを参照してください。

これらの句の大半はオプションですが、SELECT 文に記述するときは、正しい順序で記述してください。

SELECT 文の構文の詳細については、『ASA SQL リファレンス・マニュアル』>「SELECT 文」を参照してください。

SQL クエリ

このマニュアルの SELECT 文とその他の SQL 文の表記では、それぞれの句を個別の行で示し、SQL キーワードを大文字で示します。これは必須条件ではありません。SQL キーワードは大文字／小文字のいずれでも入力でき、どこでも改行できます。

キーワードと改行

たとえば、次の SELECT 文は、Contact 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 first_name, last_name
FROM Contact
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 first_name,
last_name from contact
wHere state
 = 'CA'
```

文字列と識別子の大文字と小文字の区別

Adaptive Server Anywhere データベースでは、識別子 (テーブル名、カラム名など) は、大文字と小文字を区別しません。

文字列はデフォルトでは大文字と小文字が区別されないので、'CA'、'ca'、'cA'、'Ca' は同じです。ただし、大文字と小文字を区別するデータベースを作成する場合は、文字列中の大文字／小文字が重要になります。このサンプル・データベースでは大文字と小文字を区別しません。

識別子の修飾

どのオブジェクトのことを指しているのかはっきりしない場合には、データベース識別子の名前を修飾します。たとえば、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 DBA.contact.city
FROM contact
WHERE state = 'CA'
```

この章の例で示すのは単一テーブルのクエリですから、構文のモデルや例に出てくるカラム名を、カラムが属しているテーブルや所有者の名前で修飾することは通常はありません。

これらの要素は読みやすさを考慮して省略してあるものなので、修飾しても決して間違いではありません。

この章の後続の項では、SELECT 文の構文をもっと詳しく分析します。

SELECT リスト：カラムの指定

select リスト

select リストは、通常はカンマで間を区切った一連のカラム名か、またはすべてのカラムを表す省略形として 1 つのアスタリスクで構成されています。

より一般的には、select リストには、1 つ以上の式がカンマで区切られた形で記述されます。select リストの一般的な構文は次のようになります。

SELECT *expression* [, *expression*]...

リスト中に、有効な識別子としての規則を満たしていないテーブルやカラムを記述する場合は、それらの識別子を二重引用符で囲んでください。

select リストの式に記述することができるのは、* (すべてのカラム)、カラム名のリスト、文字列、カラムの見出し、算術演算子のある式です。また、集合関数も記述できます。集合関数については、「クエリ結果の要約、グループ化、ソート」305 ページで説明します。

式を構成する各要素の詳細については、『ASA SQL リファレンス・マニュアル』>「式」を参照してください。

以下の項では、select リストで使用できる、さまざまな種類の式を示します。

テーブルからのすべてのカラムの選択

SELECT 文の中のアスタリスク (*) には特殊な意味があります。アスタリスクは、FROM 句で指定されたすべてのテーブルにあるすべてのカラム名を表します。1 つのテーブルのカラムをすべて参照したいときにこれを使用すると、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 department 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 department
```

結果は次のようになります。

dept_id	dept_name	dept_head_id
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 dept_id, dept_name, dept_head_id
 FROM department
```

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 department.*
 FROM department
```

テーブルからの特定カラムの選択

次の構文を使用すると、1 つのテーブルから特定のカラムだけを選択できます。

SELECT *column_name* [, *column_name*]…
FROM *table-name*

カラム名とカラム名の間はカンマで区切ってください。次に例を示します。

```
SELECT emp_lname, emp_fname
 FROM employee
```

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、department 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 dept_id, dept_name
 FROM department
```

dept_id	dept_name
100	R & D
200	Sales
300	Finance
400	Marketing
…	…

```
SELECT dept_name, dept_id
 FROM department
```

dept_name	dept_id
R & D	100
Sales	200
Finance	300

dept_name	dept_id
Marketing	400
…	…

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*

SELECT *column-name alias*

SELECT *alias* = *column-name*

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 dept_name を Department に変更できます。

```
SELECT dept_name AS Department,
     dept_id AS "Identifying Number"
 FROM department
```

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

dept_id の代わりに使用されているエイリアス 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 dept_name AS Department,
     dept_id AS "integer"
 FROM department
```

Adaptive Server Enterprise との互換性を確保する場合は、30 バイト以下の、引用符で囲んだエイリアスを使用します。

クエリ結果の文字列

これまで見てきた SELECT 文は、FROM 句のテーブルのデータだけで構成された結果を生成しました。文字列を一重引用符で囲み、それを select リストの他の要素とカンマで区切ると、その文字列もクエリ結果に表示されます。

文字列に引用符を記述するには、その前に引用符をもう 1 つ記述します。

次に例を示します。

```
SELECT 'The department''s name is' AS "Prefix",
     Department = dept_name
 FROM department
```

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、より複雑にすることができます。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, unit_price
 FROM product
```

name	quantity	unit_price
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 product
```

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 * unit_price AS "Inventory value"
 FROM product
```

name	Inventory value
Tee Shirt	252
Tee Shirt	756
Tee Shirt	1050
Baseball Cap	1008
…	…

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、他のすべての計算に優先します。

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 * unit_price - 5
 FROM product
```

誤解を避けるために、カッコを使用することをおすすめします。次のクエリは前述のクエリと同じ意味で、同じ結果を生成しますが、こちらの方がわかりやすいと言えます。

```
SELECT name, ( quantity * unit_price ) - 5
 FROM product
```

演算子の優先度の詳細については、『ASA SQL リファレンス・マニュアル』> 「演算子の優先度」を参照してください。

文字列の演算

文字列連結演算子を使用して文字列を連結できます。「||」(SQL/92 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。

次の例は、select リストでの文字列連結演算子の使用法を示しています。

```
SELECT emp_id, emp_fname || ' ' || emp_lname AS Name
 FROM employee
```

emp_id	Name
102	Fran Whitney
105	Matthew Cobb
129	Philip Chin
148	Julie Jordan
…	…

日付と時刻の演算

日付カラムと時刻カラムでも演算子を使用できますが、そのためには一般的に関数を使用することになります。SQL 関数については、『ASA SQL リファレンス・マニュアル』>「SQL 関数」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、Adaptive Server 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 contact 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 contact
```

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 contact
```

NULL 値は互いに区別されない

DISTINCT キーワードは、複数の NULL 値を互いに重複するものとして処理します。つまり、SELECT 文に DISTINCT が記述されていると、どんなに多くの NULL 値が検出された場合でも、結果には 1 つの NULL しか返されません。

FROM 句：テーブルの指定

テーブル、ビュー、またはストアド・プロシージャのデータに関係のある SELECT 文には、FROM 句が必須です。

FROM 句には、2 つ以上のテーブルをリンクする JOIN 条件を記述することができるほか、他のクエリ (抽出テーブル) へのジョインを記述することができます。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」335 ページを参照してください。

テーブル名の修飾

FROM 句では、テーブルとビューについては常に、次のようなフルネームでの構文を作成できます。

```
SELECT select-list
 FROM owner.table_name
```

テーブル名、ビュー名、またはプロシージャ名を修飾する必要があるのは、そのオブジェクトが現在の接続のユーザ ID と異なるユーザ ID によって所有されている場合、または現在の接続のユーザ ID が所属するグループ名と異なるユーザ ID によって所有されている場合だけです。

相関名の使用

テーブル名に相関名を指定すると入力の手間が省けます。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.dept_id, d.dept_name
 FROM Department d
```

たとえば WHERE 句の中など、Department テーブルに対する他のすべての参照には、必ず相関名を使用してください。相関名は有効な識別子としての規則に従っている必要があります。

FROM 句 の詳細については、『ASA SQL リファレンス・マニュアル』> 「FROM 句」を参照してください。

テーブル以外のオブジェクトのクエリ

FROM 句内の最も一般的な要素は、テーブル名です。テーブルに似た構造を持つ、すなわち、適切に定義されたローとカラムを持つ他のデータベース・オブジェクトからローを問い合わせることも可能です。テーブルやビューからのクエリに加えて、抽出テーブル (実体は SELECT 文) や結果セットを返すストアド・プロシージャも使用できます。

たとえば、次のクエリはストアド・プロシージャの結果セットを操作します。

```
SELECT *
 FROM sp_customer_products( 149 )
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「FROM 句」を参照してください。

WHERE 句：ローの指定

SELECT 文の WHERE 句は、どのローを検索するのかを正確に決定する探索条件を指定します。一般的なフォーマットは次のとおりです。

SELECT *select_list*
FROM *table_list*
WHERE *search-condition*

WHERE 句の探索条件 (修飾または述部とも呼ばれる) には、次のものがあります。

- **比較演算子** (=、<、> など) たとえば、給料が $50,000 を上回る従業員全員をリストする場合は、次のようになります。

```
SELECT emp_lname
 FROM employee
 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 emp_lname
 FROM employee
 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 company_name , state
 FROM customer
 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 company_name , phone
 FROM customer
 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 dept_name
 FROM Department
 WHERE dept_head_id 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 emp_fname, emp_lname
 FROM employee
 WHERE salary > 50000
 AND emp_fname like 'A%'
```

さらに、WHERE キーワードは次のことを導入できます。

- **Transact-SQL ジョインの条件**　ジョインについては、「ジョイン：複数テーブルからのデータ検索」335 ページを参照してください。

探索条件の詳細については、『ASA SQL リファレンス・マニュアル』>「探索条件」を参照してください。

後続の項では、WHERE 句の使用法を説明します。

WHERE 句での比較演算子の使用

WHERE 句で比較演算子が使用できます。演算子は次の構文に従います。

WHERE *expression comparison-operator expression*

比較演算子の詳細については、『ASA SQL リファレンス・マニュアル』>「比較演算子」を参照してください。式を構成する各要素の説明については、『ASA SQL リファレンス・マニュアル』>「式」を参照してください。

比較についての注意事項

- **ソート順**　文字データの比較の場合、< はソート順の前の方、> はソート順の後の方であるという意味です。ソート順は、データベースを作成するときに選択した照合順によって決まります。データベースに対して *dbinfo* コマンド・ライン・ユーティリティを実行すると、照合順を確認できます。

  ```
  dbinfo -c "uid=DBA;pwd=SQL"
  ```

  照合順は Sybase Central からも確認できます。データベース・プロパティ・シートの [詳細情報] タブにあります。

- **後続ブランク**　データベースの作成時に、比較を目的とした場合に後続ブランクを無視するかどうかを指定します。

  デフォルトでは、後続ブランクを無視しないでデータベースを作成します。たとえば、'Dirk' は 'Dirk　' と同じではありません。後続ブランクが無視されるように、ブランクを埋め込んだデータベースを作成できます。Adaptive Server Enterprise データベースの場合、後続ブランクはデフォルトでは無視されます。

- **日付の比較**　日付を比較するときには、< はより古いことを意味し、> はより新しいことを意味します。

- **大文字と小文字の区別**　データベースの作成時に、文字列比較が大文字と小文字を区別するかどうかを指定します。

  デフォルトでは、データベースは大文字と小文字を区別しないで作成されます。たとえば、'Dirk' は 'DIRK' と同じです。大文字と小文字を区別するように、データベースを作成できます。Adaptive Server Enterprise データベースの場合は、これがデフォルトの動作です。

次は比較演算子を使用する SELECT 文です。

```
SELECT *
 FROM product
 WHERE quantity < 20
 SELECT E.emp_lname, E.emp_fname
 FROM employee E
 WHERE emp_lname > 'McBadden'
 SELECT id, phone
 FROM contact
 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 product
 WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
 AND NOT unit_price > 10
 SELECT id, name, quantity
 FROM product
 WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
 AND unit_price !> 10
```

WHERE 句での範囲 (between と not between) 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, unit_price
 FROM product
 WHERE unit_price BETWEEN 10 AND 15
```

name	unit_price
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, unit_price
 FROM product
 WHERE unit_price NOT BETWEEN 10 AND 15
```

name	unit_price
Tee Shirt	9
Baseball Cap	9
Visor	7
Visor	7
…	…

WHERE 句でのリストの使用

IN キーワードは、値のリストのいずれかに一致する値を選択するためのキーワードです。式は定数またはカラム名であり、リストは、定数のセット、またはより一般的には、サブクエリです。

たとえば、IN を使用せずに、オンタリオ州、マニトバ州、またはケベック州内の連絡先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 company_name , state
 FROM customer
 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 company_name , state
 FROM customer
 WHERE state IN( 'ON', 'MB', 'PQ')
```

おそらく、IN キーワードのもっとも重要な用途は、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

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、バイナリ、日時データとともに使用します。

LIKE の構文は、次のとおりです。

{ **WHERE** | **HAVING** } *expression* [**NOT**] **LIKE** *match-expression*

一致する式は、次の特殊記号を含むマッチ式と比較されます。

記号	意味
%	文字数が 0 以上の文字列に一致します。
_	文字 1 個に一致します。
[specifier]	カッコ内の指定子のフォームは、次のとおりです。 • **範囲**　範囲のフォームは *rangespec1-rangespec2* です。*rangespec1* は文字の範囲の始点を指し、ハイフンは範囲、*rangespec2* は文字範囲の終点を指します。 • **セット**　セットは、任意の順序での値の個別の集合で構成されます。たとえば、[a2bR] などです。 範囲 [a-f] と、セット [abcdef] と [fcbdae] は、同じ値セットを返します。
[^specifier]	指定子の前にある脱字記号 (^) は非包含を表します。[^a-f] は a ～ 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、Contact テーブルの last_name カラムで LIKE を使用しています。クエリのフォームは次のとおりです。

```
SELECT last_name
 FROM contact
 WHERE last_name LIKE match-expression
```

最初の例は次のように入力します。

```
SELECT last_name
 FROM contact
 WHERE last_name 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	McEvoy
'%er'	**er** で終わるすべての名前を検索	Brier, Miller, Weaver, Rayner
'%en%'	**en** を含むすべての名前を検索	Pettengill, Lencki, Cohen
'_ish'	**ish** で終わるすべての 4 文字の名前を検索	Fish
'Br[iy][ae]r'	Brier、Bryer、Briar、または Bryar の検索	Brier
'[M-Z]owell'	M ～ Z の範囲の 1 文字で始まり、**owell** で終わるすべての名前を検索	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	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、「**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 Contact
 WHERE phone = '415%'
```

日付値と時刻値での LIKE の使用

文字データだけではなく、日時フィールドでも LIKE を使用できます。日付値や時刻値で LIKE を使用すると、日付は標準的な DATETIME フォーマットに変換され、次に VARCHAR に変換されます。

DATETIME 値を検索するときに LIKE を使用する場合の特徴は、日付エントリと時刻エントリにはさまざまな日付部分があるため、検索に成功するには等号テストを慎重に書き込まなければならないということです。

たとえは、カラム arrival_time に値 9:20 と現在の日付を入力するときに、次のように句を指定するとします。

```
WHERE arrival_time = '9:20'
```

このエントリには、時刻だけではなく日付も格納されているため、値の検索に失敗します。しかし、次の句なら 9:20 という値を検出します。

```
WHERE arrival_time LIKE '%09:20%'
```

NOT LIKE の使用

LIKE とともに使用できるのと同じワイルドカード文字を、NOT LIKE にも使用できます。次のクエリのいずれかを使用すると、Contact 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 Contact
 WHERE phone NOT LIKE '415%'
 SELECT phone
 FROM Contact
 WHERE NOT phone LIKE '415%'
```

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 first_name, last_name
 FROM contact
 WHERE first_name = '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。

Interactive SQL は、Interactive SQL セッションの間は quoted_identifier を自動的に ON に設定します。

❖ 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、Adaptive Server Anywhere オプションは quoted_identifier が ON です。

文字列での引用符

文字エントリ内でリテラル引用符を指定する方法は 2 つあります。1 つめの方法は、引用符を 2 回連続して使用する方法です。たとえば、一重引用符で文字列を開始し、そのエントリの一部として一重引用符を 1 つ入力する場合は、一重引用符を 2 つ使用します。

```
'I don''t understand.'
```

二重引用符の場合は次のようになります (quoted_identifier が OFF の場合)。

```
"He said, ""It is not really confusing."""
```

2 つめの方法は、quoted_identifier が OFF の場合にしか使用できませんが、引用符を別の種類の引用符で囲む方法です。つまり、二重引用符が入っているエントリは一重引用符で囲み、逆に一重引用符が入っているエントリは二重引用符で囲みます。次にその例を示します。

```
'George said, "There must be a better way."'
 "Isn't there a better way?"
 'George asked, "Isn''t there a better way?"'
```

不定の値：NULL

カラムに NULL がある場合は、ユーザまたはアプリケーションがそのカラムになにも入力していないことを意味します。このカラムのデータ値は、不定か、使用不可です。

NULL は 0 (数値) やブランク (文字値) と同じではありません。むしろ、NULL 値によって、数値カラムの場合の 0 や文字カラムの場合のブランクなどの意図的な入力と、入力がない場合とを区別できます。入力が行われていない場合は、数値カラムと文字カラムは両方とも NULL です。

NULL の入力

NULL は、create table 文に指定したように NULL 値が許可されたカラムに、次の 2 つの方法で入力できます。

- **デフォルト**　データが入力されず、カラムに他のデフォルト設定がない場合、NULL が入力されます。
- **明示的な入力**　NULL (引用符なし) と入力することで、NULL 値を明示的に入力できます。

  NULL という単語を引用符をつけて文字カラムに入力すると、NULL は null 値としてではなく、1 つのデータとして処理されます。

たとえば、department テーブルの dept_head_id カラムは null を許可します。次のように、マネージャのいない部署のローを 2 種類入力できます。

```
INSERT INTO department (dept_id, dept_name)
 VALUES (201, 'Eastern Sales')
 INSERT INTO department
 VALUES (202, 'Western Sales', null)
```

NULL が検索された場合

NULL が検索されると、Interactive SQL によるクエリ結果表示の適切な位置に (NULL) が表示されます。

```
SELECT *
 FROM department
```

dept_id	dept_name	dept_head_id
100	R & D	501
200	Sales	904
300	Finance	1293
400	Marketing	1576
500	Shipping	703
201	Eastern Sales	(NULL)
202	Western Sales	(NULL)

NULL のためのカラムのテスト

探索条件で IS NULL を使用すると、カラム値を NULL と比較したり、その比較の結果に基づいてカラム値を選択したりするなど、特定の動作を実行することができます。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、unit_price が $15 未満または NULL のローだけを選択します。

```
SELECT quantity , unit_price
 FROM product
 WHERE unit_price < 15
 OR unit_price IS NULL
```

NULL が指定の値や別の NULL に等しいか (または等しくないか) わからないので、NULL 比較の結果はすべて UNKNOWN になります。

true を決して返さない条件があり、そうした条件を使用するクエリは結果セットを返しません。たとえば、NULL は不定の値があるという意味であるため、次の比較は決して true とは見なされません。

```
WHERE column1 > NULL
```

WHERE 句でカラム名を 2 つ使用する場合、つまり 2 つのテーブルをジョインする場合にも、この論理は適用されます。次の条件を含む句は、

```
WHERE column1 = column2
```

カラムが NULL を含んでいるときにはローを返しません。

次のパターンの NULL や非 NULL もあります。

```
WHERE column_name IS [NOT] NULL
```

次に例を示します。

```
WHERE advance < $5000
 OR advance IS NULL
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「NULL 値」を参照してください。

NULL のプロパティ

次の項目で、NULL のプロパティを詳しく説明します。

- **FALSE と UNKNOWN の相違**　FALSE と UNKNOWN はどちらも値を返しませんが、false の反対 ("not false") は true なので、FALSE と UNKNOWN の間には大きな論理的相違があります。次に例を示します。

  ```
  1 = 2
  ```

  上の例は false と評価されます。

  ```
  1 != 2
  ```

  上の例は、false の反対なので true と評価されます。しかし、"not unknown" は依然として不定です。比較の中に null 値がある場合は、式を否定しても、反対のロー・セットや反対の真の値は得られません。

- **複数の NULL を 1 つの値で置き換える**　ISNULL 組み込み関数を使用して、複数の NULL を 1 つの特定の値と置き換えます。置換は表示目的のためだけに実行されます。実際のカラム値には影響しません。構文は次のとおりです。

  ISNULL(*expression*, *value*)

  たとえば、次の文を使用して test のすべてのローを選択し、すべての null 値を 'unknown' という値でカラム t1 に表示します。

  ```
  SELECT ISNULL(t1, 'unknown')
   FROM test
  ```

- **NULL と評価される式**　オペランドのいずれかが null の場合、算術演算子またはビット処理演算子のある式は NULL と評価されます。次に例を示します。

  ```
  1 + column1
  ```

  colimn1 が NULL であれば NULL と評価されます。

- **文字列と NULL の連結**　文字列と NULL を連結すると、式は文字列と評価されます。次に例を示します。

  ```
  SELECT 'abc' || NULL || 'def'
  ```

  文字列 **abcdef** を返します。

論理演算子による複数の条件の接続

論理演算子 AND、OR、NOT を使用して、WHERE 句で複数の探索条件を接続します。

AND の使用

AND 演算子は 2 つ以上の条件を結合し、それらの条件のすべてが true である場合にだけ、結果を返します。たとえば次のクエリは、連絡先の姓が Purcell で、名前が Beth のローだけを検出します。Beth Glassmann のローは検出しません。

```
SELECT *
 FROM contact
 WHERE first_name = 'Beth'
     AND last_name = 'Purcell'
```

OR の使用

OR 演算子も 2 つ以上の条件を結合しますが、それらの条件のうち、*いずれかが* true の場合に、結果を返します。次のクエリは、first_name カラムの中で Elizabeth の別名を含むローを検索します。

```
SELECT *
 FROM contact
 WHERE first_name = 'Beth'
     OR first_name = '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 contact
 WHERE NOT state = 'CA'
```

1 つの文で複数の論理演算子が使用されている場合、通常は、AND 演算子が OR 演算子の前に評価されます。実行順序はカッコを使用して変更できます。次に例を示します。

```
SELECT *
 FROM contact
 WHERE ( city = 'Lexington'
           OR city = 'Burlington' )
     AND state = 'MA'
```

第 7 章

クエリ結果の要約、グループ化、ソート

この章の内容

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。

この章では、クエリ結果のグループ化とソートの方法について説明します。

集合関数を使用したクエリ結果の要約

テーブル中のすべてのロー、WHERE 句によって指定されるテーブルのサブセット、またはテーブル中のローが 1 つ以上のグループに、集合関数を適用できます。集合関数を適用するそれぞれのロー・セットから、Adaptive Server Anywhere が単一の値を生成します。

使用できる集合関数の一部を次に示します。

- **avg(expression)** 返されたローについて提供された式の平均。
- **count(expression)** 提供されたグループで、式が非 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、『ASA SQL リファレンス・マニュアル』> 「集合関数」を参照してください。

AVG、SUM、LIST、COUNT と共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 (unit_price + 1)
 FROM product
```

例

次のクエリは、employee 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 employee
```

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 の詳細については、「HAVING 句 : データ・グループの選択」318 ページを参照してください。

WHERE 句や JOIN 条件の中では、集合関数は使用できません。しかし、select リストの集合関数による SELECT 文には、多くの場合、その集合関数が適用されるローを制限する WHERE 句があります。

SELECT 文に WHERE 句が入っているが GROUP BY 句は入っていない場合、集合関数は、WHERE 句が指定するローのサブセットに対して単一の値を生成します。

GROUP BY 句がない SELECT 文で集合関数を使用すると、必ず単一の値が生成されます。これは、その関数がテーブル中のすべてのローに作用するか、WHERE 句が定義したローのサブセットに作用するかに関係なく、生成されます。

同一の select リストで 2 つ以上の集合関数を使用でき、単一の SELECT 文で 2 つ以上のスカラ集合関数を作成できます。

集合関数と外部参照

Adaptive Server Anywhere バージョン 8 以降は、SQL/99 標準に従い、サブクエリでの集合関数の使用を明確にします。この変更は、Adaptive Server Anywhere の以前のバージョン用に記述された文の動作に影響を与えます。以前は正しかったクエリでエラー・メッセージが生成され、結果セットが変わる可能性があります。

集合関数がサブクエリに使用され、集合関数の参照先カラムが外部参照の場合は、集合関数全体が外部参照として扱われるようになりました。これは、集合関数がサブクエリ内ではなく外部ブロック内で計算されるようになり、サブクエリ内の定数となることを意味しています。

サブクエリに含まれる外部参照の集合関数の使用には、次の制限が適用されるようになりました。

- 外部参照の集合関数を指定できるのは、SELECT リストまたは HAVING 句にあるサブクエリの中だけです。また、これらの句は、外部ブロックの隣になくてはなりません。
- 外部参照の集合関数に指定できるのは、1 つの外部カラム参照だけです。
- ローカル・カラム参照と外部カラム参照を、同じ集合関数に混在させることはできません。

新しい標準に関連する問題は、ローカル参照しか含まないように集合関数を書き換えると回避することができます。たとえば、サブクエリ (`SELECT MAX(S.y + R.y) FROM S`) がローカル・カラム参照 (S.y) と外部カラム参照 (R.y) の両方で構成されていると、不正になります。このサブクエリは (`SELECT MAX(S.y) + R.y FROM S`) に書き換えることができます。書き換えると、集合関数はローカル・カラム参照だけを持つことになります。外部参照の集合関数が SELECT や HAVING 以外の句に指定されている場合にも、同様の書き換えを使用できます。

例

次のクエリの場合、Adaptive Server Anywhere バージョン 7 では次に示す結果が生成されました。

```
SELECT name, (SELECT SUM(p.quantity)
                FROM sales_order_items)
 FROM product p
```

name	sum(p.quantity)
Tee Shirt	30,716
Tee Shirt	59,238

最新のバージョンで同じクエリを実行すると、エラー・メッセージ「ASA エラー -149: 'name' に対する関数またはカラムの参照も GROUP BY 句に記述する必要があります。」が生成されます。この文が有効ではなくなったのは、外部参照の集合関数 sum(p.quantity) が外部ブロック内で計算されるようになったためです。最新のバージョンでは、このクエリはセマンティック上は次のクエリと同じです (Z が結果セットに表示されないことを除く)。

```
SELECT name,
        SUM(p.quantity) as Z,
        (SELECT Z FROM sales_order_items)
 FROM product 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(last_lname)
FROM  contact
```

COUNT(*) の使用

COUNT(*) 関数は、引数として式を必要としません。定義上、この関数は、特定のカラムに関する情報を使用しないからです。COUNT(*) 関数は、1 つのテーブルのローの総数を検出します。次の文は、従業員の総数を検出します。

```
SELECT COUNT(*)
FROM employee
```

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。

ほかの集合関数と同じように、count(*) も、select リストにあるほかの集合関数や WHERE 句などと結合できます。

```
SELECT count(*), AVG(unit_price)
FROM product
WHERE  unit_price > 10
```

count(*)	AVG(product.unit_price)
5	18.2

DISTINCT を伴う集合関数の使用

DISTINCT キーワードは、SUM、AVG、COUNT のオプションです。DISTINCT を使用すると、重複した値が除外されてから、合計、平均、またはカウントが計算されます。

たとえば、連絡先のある都市の数を検出するには、次のように入力します。

```
SELECT count(DISTINCT city)
FROM contact
```

count(distinct contact.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 first_name ) "first names",
        count( DISTINCT last_name ) "last names"
 FROM contact
```

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 product
 WHERE unit_price > 50
```

count(DISTINCT name)
0

```
SELECT AVG(unit_price)
 FROM product
 WHERE unit_price > 50
```

AVG(product.unit_price)
(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 つ以上のカラム名か数値データ型を使用して計算されたカラムの結果によってグループ化 (GROUP BY) することができます。

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(unit_price) AS Price
 FROM product
 GROUP BY name
```

name	Price
Tee Shirt	12.333333333
Baseball Cap	9.5
Visor	7
Sweatshirt	24
…	…

集合関数と 1 つの GROUP BY 句を持つ SELECT 文が生成する計算値 (ベクトル集約値) は、結果の各ローにカラムとして表示されます。それとは対照的に、集合関数があって GROUP BY 句がないクエリが生成する計算値 (スカラ集約値) は、カラムとして表示されますが、ローは 1 つだけです。次に例を示します。

```
SELECT AVG(unit_price)
 FROM product
```

AVG(product.unit_price)
13.3

GROUP BY 句の概要

クエリに GROUP BY 句が含まれていると、どのクエリが有効でどのクエリが無効かを判断するのは困難です。この項では、クエリの結果と有効性をよりよく理解できるように、GROUP BY のあるクエリについて説明します。

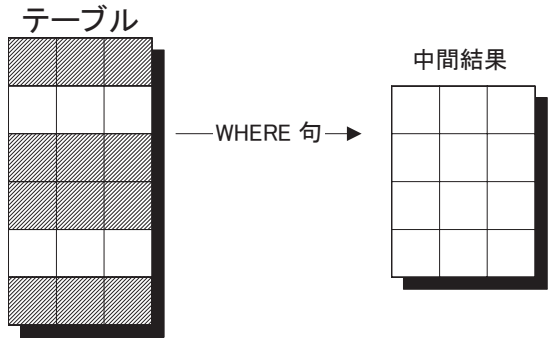
GROUP BY のあるクエリを実行する方法

次のようなフォームの単一テーブルのクエリを考えてみます。

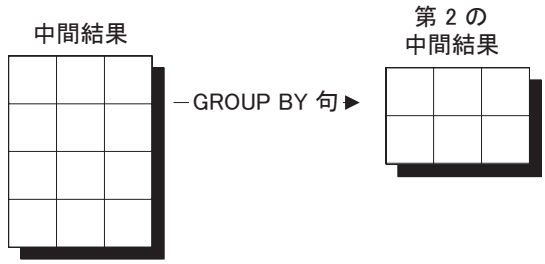
SELECT *select-list*
  FROM *table*
  WHERE *where-search-condition*
  GROUP BY [*group-by-expression* | **ROLLUP** *(group-by-expression)*]
  HAVING *having-search-condition*

このクエリは、次のように実行されると考えられます。

1. **WHERE 句を適用する**　テーブルのローの一部だけで構成される中間結果が生成されます。



2. **結果をグループに分割する**　GROUP BY 句の指示どおりに、各グループに対してローが 1 つある中間結果が生成されます。生成された各ローには、グループごとの *group-by-expression* と、*select-list* および *having-search-condition* の集合関数の計算結果が含まれます。

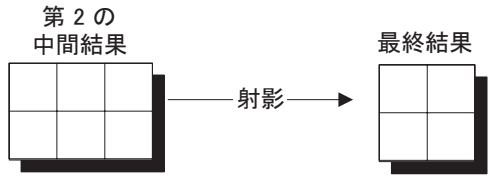


3. **ROLLUP 演算があれば適用する**　ROLLUP 演算の一部として計算された小計ローが、結果セットに追加されます。

   詳細については、「ROLLUP の使用」426 ページを参照してください。

4. **HAVING 句を適用する**　第 2 の中間結果で HAVING 句の基準に満たないローは、この時点で削除されます。

5. **プロジェクトの結果を表示する**　クエリの結果セットに表示する必要のあるカラムだけが手順 3 から取り出されます。つまり、*select-list* にある式に対応するカラムだけが取り出されます。



このプロセスでは、GROUP BY 句のあるクエリについて、いくつかの要件が作成されます。

- WHERE 句が最初に評価される。したがって、どの集合関数も、WHERE 句を満たすローについてのみ評価されます。
- 最終結果セットは、分割されたローを保持している第 2 の中間結果から構築される。第 2 の中間結果は、*group-by-expression* に一致するローを保持しています。したがって、集合関数では

ない式が *select-list* にある場合、同様に *group-by-expression* にもその式がなければなりません。プロジェクションの段階では関数の評価は行われません。

- *group-by-expression* にある式を、*select-list* に含まないことも可能です。その式は、結果にプロジェクションされます。

複数のカラムを持つ GROUP BY

GROUP BY 句に 1 つ以上の式をリストして、グループをネストできます。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(unit_price)
 FROM product
 GROUP BY name, size
```

name	size	AVG(product.unit_price)
Tee Shirt	Small	9
Tee Shirt	Medium	14
Tee Shirt	One size fits all	14
Baseball Cap	One size fits all	9.5
…	…	…

select リストにない GROUP BY 句内のカラム

Adaptive Server Enterprise と Adaptive Server 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 contact
 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(unit_price)
 FROM product
 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 product
 WHERE name LIKE '%shirt%'
 GROUP BY name
 HAVING SUM(quantity) > 100
```

name	SUM(product.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。

HAVING 句：データ・グループの選択

HAVING 句は、クエリが返すローを制限します。HAVING 句は、WHERE 句が SELECT 句の条件を設定するのと同じような方法で、GROUP BY 句の条件を設定します。

HAVING 句の探索条件は、WHERE 探索条件と同じです。ただし、WHERE 探索条件では集合関数を指定できませんが、HAVING 探索条件ではそれができるという点が異なります。したがって次の例は有効です。

```
HAVING AVG(unit_price) > 20
```

しかし、次の例は無効です。

```
WHERE AVG(unit_price) > 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、product 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 product
 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」307 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 product
 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、product 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 product
 GROUP BY name
 HAVING COUNT(*) > 1
 AND MAX(unit_price) > 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 product
 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、product 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 product
 WHERE name like '%shirt%'
 ORDER BY name, quantity DESC
```

id	name	quantity
600	Sweatshirt	39
601	Sweatshirt	32
302	Tee Shirt	75

id	name	quantity
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 product
 WHERE name like '%shirt%'
 ORDER BY 2, 3 DESC
 SELECT id, name, quantity
 FROM product
 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、Adaptive Server 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 product
 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 はほか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 employee
 ORDER BY emp_lname
```

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 employee
 ORDER BY emp_lname
```

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 employee
 ORDER BY emp_lname 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
START AT 値は 1 以上の値にしてください。TOP 値は、定数の場合に 1 以上の値にし、変数の場合に 0 以上にしてください。

ORDER BY と GROUP BY

ORDER BY 句を使用して、特定の方法で GROUP BY の結果を順序付けできます。

例

次のクエリは、各製品の平均価格を検出し、その平均価格順に結果をソートします。

```
SELECT name, AVG(unit_price)
 FROM product
 GROUP BY name
 ORDER BY AVG(unit_price)
```

name	**AVG(product.unit_price)**
Visor	7
Baseball Cap	9.5
Tee Shirt	12.333333333
Shorts	15
…	…

UNION、INTERSECT、EXCEPT を使用した、クエリ結果に対する集合操作の実行

この項で説明する演算子は、2 つ以上のクエリの結果に対して集合操作を実行します。こうした操作の多くは WHERE 句や HAVING 句を使用しても実行できますが、中にはこれらの集合ベースの演算子を使用せずに実行するのは非常に困難な操作もあります。次に例を示します。

- データが非正規化形式で格納されている場合、たとえテーブルが関連付けられていなくても、異なるように思われる情報を 1 つの結果セットにまとめたいと思うことがあります。
- WHERE 句や HAVING 句内では、集合演算子によって、NULL を扱う方法が異なります。WHERE 句や HAVING 句内では、非 NULL のエントリが同じである NULL を含む 2 つのローは、同じと見なされません。2 つの NULL 値は、同じと定義されていないためです。集合演算子は、そうした 2 つのローを同じと見なします。

  NULL と集合操作の詳細については、「集合演算子と NULL」328 ページ を参照してください。

詳細については、『ASA SQL リファレンス・マニュアル』> 「EXCEPT 演算」、『ASA SQL リファレンス・マニュアル』> 「INTERSECT 演算」、『ASA SQL リファレンス・マニュアル』> 「UNION 演算」を参照してください。

UNION 演算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。結果セットにあるカラムは、最初のテーブルのカラムを参照したのと同じ名前です。UNION 演算子はいくつでも使用できます。次に例を示します。

```
x UNION y UNION z
```

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 演算は、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 演算は、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 演算と同様に、EXCEPT と INTERSECT では ALL 修飾子を必要とします。ALL 修飾子を使用すると、結果セットから重複ローが削除されることを防ぎます。

詳細については、『ASA SQL リファレンス・マニュアル』> 「EXCEPT 演算」と 『ASA SQL リファレンス・マニュアル』> 「INTERSECT 演算」を参照してください。

集合操作のルール

UNION 演算、EXCEPT 演算、INTERSECT 演算には、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。

```
-- This is an example of an invalid statement
 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 contact
 UNION
 SELECT city
 FROM customer
 UNION
 SELECT city
 FROM employee
```

UNION 演算では、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 contact
 UNION
 SELECT city
 FROM customer
```

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 contact
 UNION
 SELECT city AS Cities
 FROM customer
```

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 contact
      UNION
      SELECT city
      FROM customer )
 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 Cities = city
      FROM contact
      UNION
      SELECT city
      FROM customer )
 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 Cities = city
 FROM contact
 UNION
 SELECT city
 FROM customer
 ORDER BY 1
```

集合演算子と NULL

集合演算子 UNION、EXCEPT、INTERSECT と探索条件内では、NULL を扱う方法が異なります。この違いが、集合演算子を使用する主な理由の 1 つです。

ローを比較するとき、集合演算子は、NULL 値を互いに等しいものとして扱います。対照的に、探索条件で NULL が NULL と比較された場合、結果は不定 (真ではない) となります。

この違いがもたらす特に有効な結果の 1 つとして、`query-1 EXCEPT ALL query-2` の結果セット内のロー数が、常に各クエリの結果セット内のロー数の差異であるということです。

テーブル T1 と T2 を例に説明します。各テーブルには、次のカラムがあります。

```
col1 INT,
 col2 CHAR(1)
```

テーブル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.col2
 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)
4	(NULL)
3	(NULL)
4	(NULL)

T1 の NULL を含むローは、比較によって除外されていません。対照的に、EXCEPT ALL を使用してこの問題にアプローチすると、両方のテーブルに含まれる NULL を持つローが結果から除外されます。この場合、T2 の (3, NULL) ローは、T1 の (3, NULL) ローと同じと認識されています。

```
SELECT col1, col2
 FROM T1
 EXCEPT ALL
 SELECT col1, col2
 FROM T2
```

col1	col2
2	b
3	(NULL)
4	(NULL)
4	(NULL)

EXCEPT 演算子を使用すると、結果がさらに制限されます。EXCEPT 演算子は、T1 から (3, NULL) のローを両方とも削除し、また (4, NULL) ローの 1 つを重複として除外しています。

```
SELECT col1, col2
 FROM T1
 EXCEPT
 SELECT col1, col2
 FROM T2
```

col1	col2
2	b
4	(NULL)

標準と互換性

この項では、Adaptive Server Anywhere の GROUP BY 句の、標準と互換性について説明します。

GROUP BY と SQL/92 標準

GROUP BY について、SQL/92 標準では次のように定めています。

- SELECT 句の式で使用されるカラムは、GROUP BY 句になければならない。そうでない場合、このカラムを使用する式は集合関数でなければならない。
- GROUP BY 式は、select リストからのカラム名だけを含むことができるが、ベクトル集合関数の引数としてのみ使用されるカラム名を含むことはできない。

ベクトル集合関数による標準的な GROUP BY の結果は、1 グループあたり、1 つの値を持つ 1 つのローを生成します。

Adaptive Server Anywhere と Adaptive Server Enterprise は、HAVING に select リストにも GROUP BY 句にもない集合関数を含むことができる拡張機能をサポートしています。

Adaptive Server Enterprise との互換性

Adaptive Server Enterprise は、Adaptive Server Anywhere ではサポートされていない、GROUP BY 句に対する拡張をサポートしています。拡張機能には次のようなものがあります。

- **select リスト中のグループ化されないカラム**　Adaptive Server Enterprise では、GROUP BY 句にないカラム名を select リストに入れることができます。たとえば、Adaptive Server Enterprise では次の文が有効です。

```
SELECT name, unit_price
 FROM product
 GROUP BY name
```

この構文は Adaptive Server Anywhere ではサポートされていません。

- **ネストされた集合関数**　スカラ集合関数の中にベクトル集合関数をネストする次のクエリは、Adaptive Server Enterprise では有効ですが、Adaptive Server Anywhere では無効です。

```
SELECT MAX(AVG(unit_price))
 FROM product
 GROUP BY name
```

- **GROUP BY と ALL**　Adaptive Server Anywhere では、GROUP BY 句に ALL を使用できません。

- **GROUP BY のない HAVING**　Adaptive Server Anywhere では、select 句と having 句内のすべての式が集合関数でないかぎり、GROUP BY 句のない HAVING は使用できません。たとえば、次のクエリは Adaptive Server Enterprise では有効ですが、Adaptive Server Anywhere ではサポートされていません。

```
--unsupported syntax
 SELECT unit_price
    FROM product
    HAVING COUNT(*) > 8
```

ただし、MAX 関数と COUNT 関数は集合関数なので、次の文は Adaptive Server Anywhere で有効です。

```
SELECT MAX(unit_price)
 FROM product
 HAVING COUNT(*) > 8
```

- **HAVING の条件**　Adaptive Server Enterprise は、select リストにも GROUP BY 句にもない集合関数以外の関数を HAVING に入れることができる拡張機能をサポートしています。Adaptive Server Anywhere では、このタイプは集合関数だけが許可されます。

- **ORDER BY または GROUP BY を伴う DISTINCT**　Adaptive Server Enterprise では、select リストだけでなく、SELECT DISTINCT クエリにもないカラムを ORDER BY 句または

GROUP BY 句の中で使用できます。そのため、SELECT DISTINCT 結果セットでは値が反復されることになります。Adaptive Server Anywhere は、この動作をサポートしていません。

- **複数の UNION の中のカラム名** Adaptive Server Enterprise では、クエリの UNION 中の ORDER BY 句でカラムを使用できます。Adaptive Server Anywhere では、ORDER BY 句で整数を使用して、結果を順序付けするカラムをマーク付けする必要があります。

第 8 章

ジョイン：複数テーブルからのデータ検索

この章の内容

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。

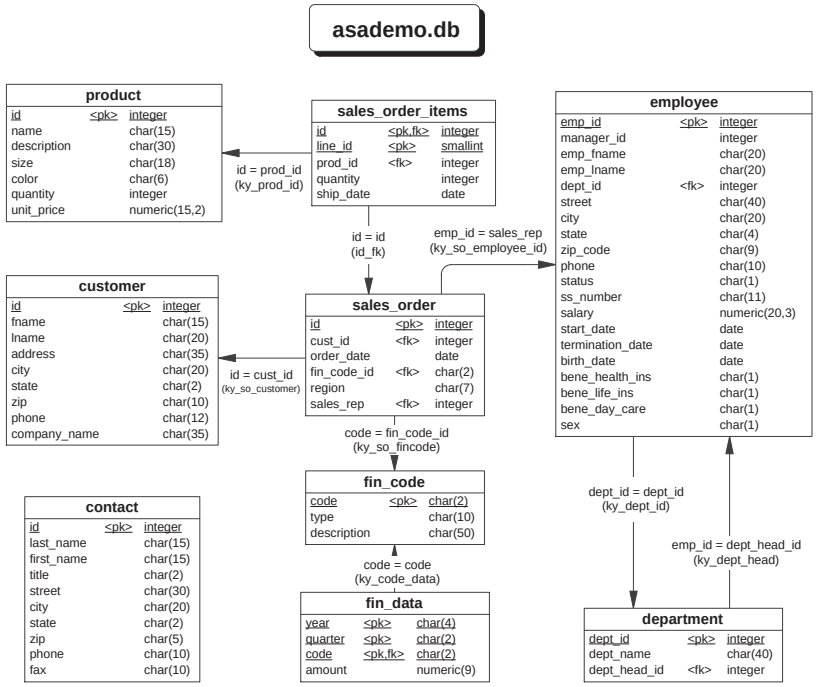
ジョイン操作では、複数のテーブル (またはビュー) からの情報を使用して 1 つのより大きいテーブルを再作成します。各種のジョインを使用すると、特定のタスクに適したさまざまな仮想テーブルを作成できます。

始める前に

この章は、クエリと SELECT 文の構文についての知識があることを前提にしています。クエリについては、「クエリ：テーブルからのデータの選択」275 ページを参照してください。

サンプル・データベース・スキーマ

この章では、サンプル・データベースを頻繁に参照します。次の図は、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。サンプル・データベースは、*asademo.db* ファイルに保管され、インストール・ディレクトリにあります。



ジョイン操作

リレーショナル・データベースは別々のタイプのオブジェクトに関する情報を別々のテーブルに保存します。たとえば、あるテーブルには従業員だけの情報があり、別のテーブルには部署関連の情報があります。employee 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。department テーブルには、部署名や部長名などの情報が入ります。

さまざまなテーブルの情報を組み合わせて使用すれば、ほとんどの問い合わせに対する答えを取得できます。たとえば、「営業部を管理しているのは誰か」という問い合わせがあります。この人物の名前を見つけるには、department テーブルの情報から正しい人物を特定し、employee テーブルでその名前を検索します。

ジョインは複数のテーブルからの情報を取り入れた新規の仮想テーブルを作ることによって、そのような質問に答える手段です。たとえば、employee テーブルと department 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。

ジョインの概要

「ジョイン」とは、テーブル内のローを、指定したカラムの値と比較することによって結合する操作のことです。この項では、Adaptive Server Anywhere のジョイン構文の概要を説明します。ここで説明した概念を以降の各項ではさらに詳しく説明します。

FROM 句

FROM 句を使用して、ジョインの対象となるベース・テーブル、テンポラリ・テーブル、ビュー、抽出テーブルを指定します。FROM 句は、SELECT 文または UPDATE 文で使用できます。次は、FROM 句の構文を簡略化したものです。

FROM *table_expression*, …

文中の各項目を次に説明します。

table_expression:
table | *view* | *derived table* | *joined table* | **(** *table_expression*, … **)**

table or view:
[*userid*.] *table-or-view-name* [[**AS**] *correlation-name*]

derived table:
(*select-statement* **)** [**AS**] *correlation-name* [**(** *column-name*, … **)**]

joined table:
table_expression join_operator table_expression [**ON** *join_condition*]

join_operator:
[**KEY** | **NATURAL**] [*join_type*] **JOIN** | **CROSS JOIN**

join_type:
INNER | **FULL** [**OUTER**] | **LEFT** [**OUTER**] | **RIGHT** [**OUTER**]

注意

CROSS JOIN には ON 句を使用できません。

完全な構文については、『ASA SQL リファレンス・マニュアル』> 「FROM 句」を参照してください。

ON 句を使った構文については、『ASA SQL リファレンス・マニュアル』> 「探索条件」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、次のクエリではテーブル product とテーブル sales_order_items からデータが取り出されます。

```
SELECT *
FROM product JOIN sales_order_items
  ON product.id = sales_order_items.prod_id
```

このクエリのジョイン条件は、次の部分です。

```
product.id = sales_order_items.prod_id
```

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 customer JOIN sales_order
  ON sales_order.cust_id = customer.id
```

これに対し、「**生成されたジョイン条件**」とは、KEY JOIN または NATURAL JOIN を指定すると自動的に作成されるジョイン条件のことです。キー・ジョインの場合、生成されたジョイン条件はテーブル間の外部キー関係に基づいています。ナチュラル・ジョインの場合には、名前が同じカラムに基づいています。

ヒント：
キー・ジョイン構文とナチュラル・ジョイン構文は、どちらもショートカットです。つまり、KEY も NATURAL も*付けずに* JOIN キーワードを使用して ON 句内で同じジョイン条件を明示的に記述しても、同一の結果が得られます。

キー・ジョインまたはナチュラル・ジョインで ON 句を使用すると、使用されるジョイン条件は、明示的に指定したジョイン条件と生成されたジョイン条件の「**論理積**」になります。つまり、ジョイン条件はキーワード AND と組み合わされます。

ジョインしたテーブル

Adaptive Server Anywhere では、次のようなジョイン条件の指定をサポートしています。

- **CROSS JOIN (クロス・ジョイン)** 2 つのテーブルのクロス・ジョインによって、両方のテーブルにあるローの組み合わせで可能なものすべてが生成されます。結果セットのサイズは、1 番目のテーブルにあるローの数と 2 番目のテーブルにあるローの数を乗算したものです。クロス・ジョインは、直積とも呼ばれます。クロス・ジョインでは ON 句を使用できません。

- **KEY JOIN (キー・ジョイン ー デフォルト)** データベースにすでに構築されている外部キー関係に基づいて、ジョイン条件が自動的に生成されます。ジョイン・タイプを指定しないでキーワード JOIN を使用する場合や、ON 句がない場合は、キー・ジョインがデフォルトになります。

- **NATURAL JOIN (ナチュラル・ジョイン)** 名前が同じカラムに基づいて、ジョイン条件が自動的に生成されます。

- **ON 句を使用したジョイン** 明示的ジョイン条件を指定します。これをキー・ジョインまたはナチュラル・ジョインと併用すると、ジョイン条件には生成されたジョイン条件と明示的ジョイン条件の両方が含まれます。KEY や NATURAL の付かない JOIN キーワードと併用すると、生成されるジョイン条件はありません。クロス・ジョインでは ON 句を使用できません。

内部ジョインと外部ジョイン

キー・ジョイン、ナチュラル・ジョイン、ON 句付きジョインは、INNER、LEFT OUTER、RIGHT OUTER、FULL OUTER を指定して修飾することもできます。デフォルトは INNER です。LEFT、RIGHT、FULL を使う場合、キーワード OUTER はオプションです。

内部ジョインでは、結果の各ローがジョイン条件を満たします。

左外部ジョインまたは右外部ジョインでは、テーブルのどちらか一方のすべてのローの値が保護されます。もう一方のテーブルでは、ジョイン条件を満たさないローに NULL が返されます。たとえば右外部ジョインでは、右側が保護され、左側に NULL が入力されます。

全外部ジョインでは、両方のテーブルのすべてのローが保護され、ジョイン条件を満たさないローに NULL が入ります。

2 つのテーブルのジョイン

簡単な内部ジョインの計算方法を理解するために、次のクエリを例にして考えてみましょう。これは、「在庫数と同数の受注数があったのはどの製品のサイズか」という質問への回答です。

```
SELECT DISTINCT name, size,
     sales_order_items.quantity
 FROM product JOIN sales_order_items
 ON product.id = sales_order_items.prod_id
   AND product.quantity = sales_order_items.quantity
```

name	size	quantity
Baseball Cap	One size fits all	12
Visor	One size fits all	36

このクエリは次のように解釈できます。次に示すのはこのクエリの処理概念の説明であり、ジョインを含むクエリのセマンティックを例証するためのものです。ここで述べる内容は、Adaptive Server 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。

- テーブル product とテーブル sales_order_items の直積が作成されます。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 `product.id = sales_order_items.prod_id` 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 `product.quantity = sales_order_items.quantity` であるため)。
- product.name、product.size、sales_order_items.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」352 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

Adaptive Server 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、Adaptive Server Anywhere では文が左から右に評価されます。したがって、`A JOIN B JOIN C` は `(A JOIN B) JOIN C` と同義です。次に、別の例を示します。

```
SELECT *
FROM A JOIN B JOIN C JOIN D
```

この文は次の文と同義です。

```
SELECT *
FROM ((A JOIN B) JOIN C) JOIN D
```

3 つ以上のテーブルをジョインした場合、そのジョインにはテーブル式が含まれます。`A JOIN B JOIN C` の例では、テーブル式 `A JOIN B` が C にジョインされます。つまり、概念上は A と B がジョインされ、その結果が C にジョインされます。

テーブル式に外部ジョインが含まれるときは、ジョインの順序が重要になります。たとえば、`A JOIN B LEFT OUTER JOIN C` は `(A JOIN B) LEFT OUTER JOIN C` と解釈されます。テーブル式 `A JOIN B` は C にジョインされるという意味です。このとき、テーブル式 `A JOIN B` は保護され、テーブル C には NULL が入力されます。

外部ジョインの詳細については、「外部ジョイン」352 ページを参照してください。

Adaptive Server 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」378 ページを参照してください。

Adaptive Server 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」370 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、『ASA SQL リファレンス・マニュアル』> 「比較演算子を使用した変換」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、SELECT 文だけではなく DELETE 文、UPDATE 文、INSERT 文でも使用できます。ANSI_UPDATE_CONSTRAINTS オプションが OFF に設定されていれば、ジョインを含むカーソルをいくつか更新できます。Adaptive Server 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 ON がデフォルトです。

詳細については、『ASA データベース管理ガイド』> 「ANSI_UPDATE_CONSTRAINTS オプション [互換性]」を参照してください。

ANSI 以外のジョイン

Adaptive Server Anywhere は、ISO/ANSI 規格のジョインをサポートしています。また、次に示す標準以外のジョインもサポートしています。

- 「Transact-SQL の外部ジョイン (*= or =*)」357 ページ
- 「ジョインで重複する相関名 (スター・ジョイン)」362 ページ
- 「キー・ジョイン」374 ページ
- 「ナチュラル・ジョイン」369 ページ

REWRITE 関数を使うと、ANSI の機能に相当する ANSI 以外のジョインを確認できます。

詳細については、『ASA SQL リファレンス・マニュアル』>「REWRITE 関数 [その他]」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。

例

次のクエリでは、最初の ON 句が sales_order を customer にジョインするために使用されています。2 番目の ON 句は、テーブル式 (sales_order JOIN customer) をベース・テーブル sales_order_item にジョインするために使用されています。

```
SELECT *
 FROM sales_order JOIN customer
     ON sales_order.cust_id = customer.id
   JOIN sales_order_items
     ON sales_order_items.id = sales_order.id
```

参照先になるテーブル

ON 句で参照されるテーブルは、その ON 句が修飾するジョインの一部である必要があります。たとえば、次の構文は無効です。

```
FROM (A KEY JOIN B) JOIN (C JOIN D ON A.x = C.x)
```

ここでの問題は、ジョイン条件 `A.x = C.x` がテーブル A を参照していることです。テーブル A は、このジョイン条件が修飾するジョイン (ここでは `C JOIN D`) ではありません。

ただし、ANSI/ISO 規格 SQL99 と Adaptive Server Anywhere 7.0 については、この規則は適用されません。つまり、テーブル式の間にカンマを使用すれば、ジョインの ON 条件は、構文中にその ON 条件より前にある FROM 句内のテーブルを参照できます。このため、次の構文は有効になります。

```
FROM (A KEY JOIN B) , (C JOIN D ON A.x = C.x)
```

カンマの詳細については、「カンマ」349 ページを参照してください。

生成されたジョインと ON 句

キーワード JOIN を使用し、ジョイン・タイプを指定していない場合、ON 句を使用しなければ、キー・ジョインがデフォルトです。指定のない JOIN とともに ON 句を使用すると、キー・ジョインはデフォルトにはならず、生成されたジョイン条件は何も適用されません。

たとえば、次の例はキー・ジョインです。キーワード JOIN が使用されており、ON 句もない場合はキー・ジョインがデフォルトだからです。

```
SELECT *
 FROM A JOIN B
```

次は、テーブル A とテーブル B のジョインであり、ジョイン条件 `A.x = B.y` も使用されています。したがって、このジョインはキー・ジョインでは*ありません*。

```
SELECT *
 FROM A JOIN B ON A.x = B.y
```

KEY JOIN または NATURAL JOIN を指定し、*さらに* ON 句も使用すると、最終的なジョイン条件は、生成されたジョイン条件と明示的ジョイン条件の論理積になります。たとえば、次の文にはジョイン条件が 2 つ入っています。1 つはキー・ジョインから生成されたジョイン条件で、もう 1 つは ON 句で明示的に記述されたジョイン条件です。

```
SELECT *
 FROM A KEY JOIN B ON A.x = B.y
```

キー・ジョインによって生成されたジョイン条件が `A.w = B.z` の場合、次の文は上の文と同義です。

```
SELECT *
 FROM A JOIN B
   ON A.x = B.y
   AND A.w = B.z
```

キー・ジョインの詳細については、「キー・ジョイン」374 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 department JOIN employee
     ON department.dept_id = employee.dept_id
```

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) などを含む条件といった、任意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 product.name
 FROM product JOIN sales_order_items
 ON product.id = sales_order_items.prod_id
     AND product.quantity > sales_order_items.quantity
```

探索条件の詳細については、『ASA SQL リファレンス・マニュアル』> 「探索条件」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」353 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」374 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共に 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* x *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」378 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)」362 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。キーワード OUTER はオプションです。これらの変更子はクロス・ジョインには適用されません。

内部ジョイン

デフォルトのジョインは「**内部ジョイン**」です。つまり、ジョイン条件を満たすローだけが結果セットに含まれます。

例

たとえば、次のクエリの結果セットで、それぞれのローには、キー・ジョイン条件を満たす 1 つの customer ローと 1 つの sales_order 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 fname, lname, order_date
 FROM customer KEY INNER JOIN sales_order
 ORDER BY order_date
```

fname	lname	order_date
Hardy	Mums	1/2/00
Aram	Najarian	1/3/00
Tommie	Wooten	1/3/00
Alfredo	Margolis	1/6/00
…	…	…

内部ジョインとキー・ジョインはデフォルトなので、次の FROM 句を使用しても前述の例と同じ結果が得られます。

```
FROM customer JOIN sales_order
```

外部ジョイン

2 つのテーブルの左または右の「**外部ジョイン**」は、一方のテーブルではすべてのローを保護し、他方のテーブルにはジョイン条件が満たされないときに NULL を入力します。「**左外部ジョイン**」は左側のテーブルのローをすべて保護し、「**右外部ジョイン**」は右側テーブルのローをすべて保護します。「**全外部ジョイン**」では、両方のテーブルのローがすべて保護されます。

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 =*)」357 ページ を参照してください。

例

たとえば次の文には、注文するかどうかにかかわらず、すべての顧客が含まれます。顧客が注文していない場合には、受注情報に対応する結果のそれぞれのカラムに NULL 値が入ります。

```
SELECT lname, order_date, city
 FROM customer LEFT OUTER JOIN sales_order
      ON customer.id = sales_order.cust_id
 WHERE customer.state = 'NY'
 ORDER BY order_date
```

lname	order_date	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、Adaptive Server 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、customer 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、sales_order のカラムに NULL が入れられます。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 customer KEY LEFT OUTER JOIN sales_order
     ON sales_order.order_date < '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 lname, order_date
 FROM customer KEY LEFT OUTER JOIN sales_order
     WHERE sales_order.order_date < '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、customer テーブルを sales_order テーブルに左外部ジョインします。結果セットには customer 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 customer と sales_order を左外部ジョインします。結果セットには customer 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、『ASA SQL リファレンス・マニュアル』>「探索条件」を参照してください。

複雑な外部ジョインについて

クエリに外部ジョインを使用したテーブル式が含まれるときは、ジョインの順序が重要になります。たとえば、`A JOIN B LEFT OUTER JOIN C` は `(A JOIN B) LEFT OUTER JOIN C` と解釈されます。つまり、テーブル式 `(A JOIN B)` が C に結合されるという意味です。このとき、テーブル式 `(A JOIN B)` は保護され、テーブル C には NULL が入力されます。

次に、以下の文を考えてみます。A、B、C はテーブルです。

```
SELECT *
 FROM A LEFT OUTER JOIN B RIGHT OUTER JOIN C
```

この文を理解するには、まず「Adaptive Server 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。Adaptive Server 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」379 ページで説明しています。

また、外部ジョインのジョイン条件には、必ず、FROM 句内で先に参照されているテーブルだけを入れます。この制限事項は ANSI/ISO 規格に基づくものであり、あいまいさを排除するためのものです。たとえば、次の 2 つの文は構文的に正しくありません。テーブル自体が参照される前に C がジョイン条件内で参照されるからです。

```
SELECT *
 FROM (A LEFT OUTER JOIN B ON B.x = C.x) JOIN C
```

および

```
SELECT *
 FROM A LEFT OUTER JOIN B ON A.x = C.x, C
```

ビューと抽出テーブルの外部ジョイン

外部ジョインは、ビューと抽出テーブルにも指定できます。

次に例を示します。

```
SELECT *
 FROM V LEFT OUTER JOIN A ON (V.x = A.x)
```

この例は、次のように解釈できます。

- ビュー V が計算されます。
- ジョイン条件 `V.x = A.x` を使用して V のローをすべて保護することによって、計算したビュー V のすべてのローが A にジョインされます。

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 employee.emp_id, dept_name
   FROM employee JOIN department
     ON employee.dept_id = department.dept_id
   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、sales_order は NULL 入力です。

```
SELECT DISTINCT V.emp_id, region, V.dept_name
   FROM V LEFT OUTER JOIN sales_order
     ON V.emp_id = sales_order.sales_rep
```

emp_id	region	dept_name
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 =*)

Adaptive Server Anywhere では ANSI/ISO SQL 規格に基づき、キーワード LEFT OUTER、RIGHT OUTER、FULL OUTER をサポートします。また、バージョン 12 よりも前の Adaptive Server Enterprise との互換性のために LEFT OUTER と RIGHT OUTER のキーワードに対応する Transact-SQL の *= and =* もサポートしています。ただし、Transact-SQL のセマンティックには、いくつかの制限事項と潜在的な問題があります。

Transact-SQL 外部ジョインの詳細については、ホワイトペーパー『*Transact-SQL 外部ジョインのセマンティックと互換性*』(http://www.ianywhere.jp/tech/index.html) を参照してください。

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。

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)。

例

たとえば、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 fname, lname, order_date
 FROM customer, sales_order
 WHERE customer.id *= sales_order.cust_id
 ORDER BY order_date
```

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 fname, lname, order_date
 FROM customer LEFT OUTER JOIN sales_order
 ON customer.id = sales_order.cust_id
 ORDER BY order_date
```

Transact-SQL 外部ジョインの制限事項

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、抽出テーブルを使用したジョインなど、特殊なジョインについて説明します。

セルフジョイン

「**セルフジョイン**」では、異なる相関名を使用して同一テーブルを参照することによって、テーブルがそれ自体にジョインされます。

例 1

次のセルフジョインは従業員のペアのリストを作成します。各従業員の名前が全従業員の名前との組み合わせで表示されます。

```
SELECT a.emp_fname, a.emp_lname,
         b.emp_fname, b.emp_lname
 FROM employee AS a CROSS JOIN employee AS b
```

emp_fname	emp_lname	emp_fname	emp_lname
Fran	Whitney	Fran	Whitney
Fran	Whitney	Matthew	Cobb
Fran	Whitney	Philip	Chin
Fran	Whitney	Julie	Jordan
…	…	…	…

employee 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 = 5,625 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。

emp_fname	emp_lname	emp_fname	emp_lname
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.emp_fname, a.emp_lname,
         b.emp_fname, b.emp_lname
 FROM employee AS a CROSS JOIN employee AS b
 WHERE a.emp_id != b.emp_id
```

この重複するローを除くと、ジョインは 75 × 74 = 5,550 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。

emp_fname	emp_lname	emp_fname	emp_lname
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 = 2,775 ローのリストを作成できます。

```
SELECT a.emp_fname, a.emp_lname,
         b.emp_fname, b.emp_lname
 FROM employee AS a CROSS JOIN employee AS b
 WHERE a.emp_id < b.emp_id
```

この文は、従業員 a の emp_id が従業員 b の emp_id 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、employee 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.emp_fname, report.emp_lname,
     manager.emp_fname, manager.emp_lname
 FROM employee AS report JOIN employee AS manager
     ON (report.manager_id = manager.emp_id)
 ORDER BY report.emp_lname, report.emp_fname
```

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。

emp_fname	emp_lname	emp_fname	emp_lname
Alex	Ahmed	Scott	Evans
Joseph	Barker	Jose	Martinez
Irene	Barletta	Scott	Evans
Jeannette	Bertrand	Jose	Martinez
…	…	…	…

ジョインで重複する相関名 (スター・ジョイン)

重複するテーブル名は、「**スター・ジョイン**」を作成するために使用します。スター・ジョインでは、1 つのテーブルまたはビューが複数のテーブルやビューにジョインされます。

スター・ジョインを作成するには、同じテーブル名、ビュー名、または相関名を FROM 句内で 2 回以上使用します。これは、ANSI/ISO SQL 規格の拡張機能です。重複名を使用しても機能は追加されませんが、使用すると特定のクエリを簡単に作成できます。

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 Adaptive Server 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
```

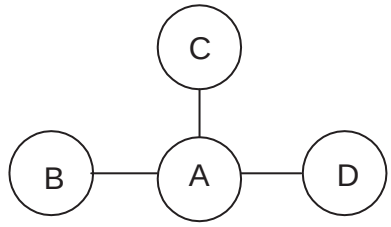
次の例では、テーブル A が 3 つのテーブル B、C、D にジョインされます。

```
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、『ASA データベース管理ガイド』> 「EXTENDED_JOIN_SYNTAX オプション [データベース]」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 employee 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、customer.id や employee.id 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 employee テーブルを使用するだけで、このジョインは可能です。

```
SELECT customer.fname, customer.lname,
      sales_order.order_date
 FROM     sales_order KEY JOIN customer,
         sales_order KEY JOIN employee
 WHERE employee.emp_fname = 'Rollin'
   AND employee.emp_lname = 'Overbey'
 ORDER BY sales_order.order_date
```

fname	lname	order_date
Tommie	Wooten	1/3/00
Michael	Agliori	1/8/00
Salton	Pepper	1/17/00
Tommie	Wooten	1/23/00
…	…	…

次の例は、ANSI/ISO 標準構文の文と同義です。

```
SELECT customer.fname, customer.lname,
   sales_order.order_date
 FROM sales_order JOIN customer
   ON sales_order.cust_id = customer.id
 JOIN employee
   ON sales_order.sales_rep = employee.emp_id
 WHERE employee.emp_fname = 'Rollin'
   AND employee.emp_lname = 'Overbey'
 ORDER BY sales_order.order_date
```

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、product、sales_order_items、customer、employee になります。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」336 ページを参照) を見ると、これらのテーブルがすべて sales_order テーブルを介して関連していることがわかります。sales_order 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。employee 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」360 ページを参照してください。

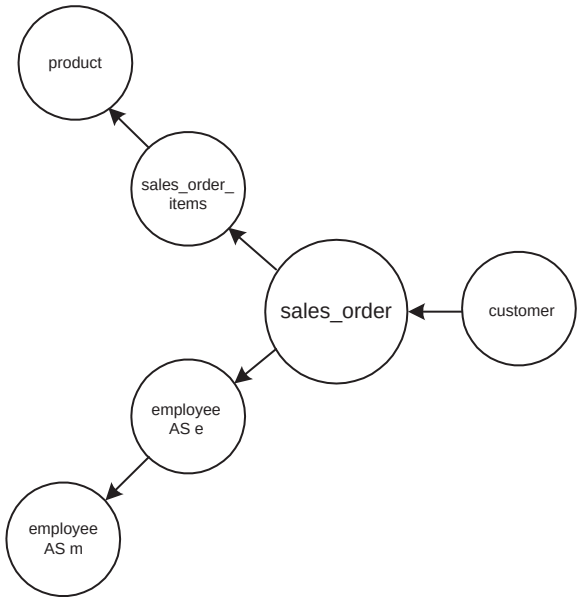
次の文は sales_order テーブルを中心にしてスター・ジョインを作成します。このジョインは、結果セットにすべての顧客が含まれるように、すべて外部ジョインになっています。注文しなかった顧客もいますが、そういう顧客の他の値は NULL になっています。結果セットのカラムは、顧客、製品、注文数、営業部員の管理者名です。

```
SELECT customer.fname, product.name,
   SUM(sales_order_items.quantity), m.emp_fname
 FROM    sales_order
             KEY RIGHT OUTER JOIN customer,
          sales_order
             KEY LEFT OUTER JOIN sales_order_items
             KEY LEFT OUTER JOIN product,
         sales_order
             KEY LEFT OUTER JOIN employee AS e
```

```
                LEFT OUTER JOIN employee AS m
                ON (e.manager_id = m.emp_id)
  WHERE customer.state = 'CA'
  GROUP BY customer.fname, product.name, m.emp_fname
  ORDER BY SUM(sales_order_items.quantity) DESC,
customer.fname
```

fname	name	SUM(sales_order_items.quantity)	emp_fname
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 customer.fname, product.name,
   SUM(sales_order_items.quantity), m.emp_fname
 FROM sales_order LEFT OUTER JOIN sales_order_items
     ON sales_order.id = sales_order_items.id
   LEFT OUTER JOIN product
     ON sales_order_items.prod_id = product.id
   LEFT OUTER JOIN employee as e
     ON sales_order.sales_rep = e.emp_id
   LEFT OUTER JOIN employee as m
     ON e.manager_id = m.emp_id
   RIGHT OUTER JOIN customer
     ON sales_order.cust_id = customer.id
 WHERE customer.state = 'CA'
 GROUP BY customer.fname, product.name, m.emp_fname
 ORDER BY SUM(sales_order_items.quantity) DESC,
customer.fname
```

抽出テーブルに関連したジョイン

抽出テーブルでは FROM 句内にクエリをネストできます。抽出テーブルを使用すると、ビューを作成せずに、グループのグループ化を実行したり、グループとのジョインを組み立てたりできます。

次の例では、内側の SELECT 文 (カッコに囲まれている) は顧客 ID 値でグループ分けされた抽出テーブルを作成します。外側の SELECT 文はこのテーブルに相関名 sales_order_counts を割り当て、ジョイン条件を使用してそれを customer テーブルとジョインします。

```
SELECT lname, fname, number_of_orders
 FROM customer JOIN
     (    SELECT cust_id, count(*)
         FROM sales_order
         GROUP BY cust_id    )
     AS sales_order_counts (cust_id, number_of_orders)
     ON (customer.id = sales_order_counts.cust_id)
 WHERE number_of_orders > 3
```

結果は各顧客の注文数を含む、4 件以上の注文をした顧客名のテーブルです。

抽出テーブルのキー・ジョインについては、「ビューと抽出テーブルのキー・ジョイン」385 ページを参照してください。

抽出テーブルのナチュラル・ジョインについては、「ビューと抽出テーブルのナチュラル・ジョイン」372 ページを参照してください。

抽出テーブルの外部ジョインについては、「ビューと抽出テーブルの外部ジョイン」355 ページを参照してください。

ナチュラル・ジョイン

ナチュラル・ジョインを指定すると、同じ名前を持つカラムに基づいてジョイン条件が生成されます。生成されたジョイン条件がベース・テーブルのナチュラル・ジョインに有効になるためには、同じ名前のカラムがどちらのテーブルにも少なくとも 1 つは存在する必要があります。共通するカラム名がなければ、エラーが発生します。

テーブル A と B が共通のカラム名を 1 つ持っており、そのカラムが x であるとします。その場合は次のようになります。

```
SELECT *
 FROM A NATURAL JOIN B
```

これは、次のクエリと同義です。

```
SELECT *
 FROM A JOIN B
  ON A.x = B.x
```

テーブル A と B が共通のカラム名を 2 つ持っており、そのカラムが a と b である場合、`A NATURAL JOIN B` は次のクエリと同義です。

```
A JOIN B
  ON A.a = B.a
  AND A.b = B.b
```

例

たとえば、テーブル employee と department には共通のカラム名 dept_id 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 emp_fname, emp_lname, dept_name
 FROM employee NATURAL JOIN department
 ORDER BY dept_name, emp_lname, emp_fname
```

emp_fname	emp_lname	dept_name
Janet	Bigelow	Finance
Kristen	Coe	Finance
James	Coleman	Finance

emp_fname	emp_lname	dept_name
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 emp_fname, emp_lname, dept_name
 FROM employee JOIN department
   ON (employee.dept_id = department.dept_id)
 ORDER BY dept_name, emp_lname, emp_fname
```

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 `employee.dept_id = department.dept_id` 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 emp_fname, emp_lname, dept_name
 FROM employee NATURAL JOIN department
   ON employee.manager_id = department.dept_head_id
```

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 emp_fname, emp_lname, dept_name
 FROM employee JOIN department
   ON employee.manager_id = department.dept_head_id
    AND employee.dept_id = department.dept_id
```

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、Adaptive Server Anywhere は、ジョイン演算子の左右にあるカラムを比較して、同じ名前のカラムを検索することでジョイン条件を生成します。

次の文を例にとります。

```
SELECT *
 FROM (A JOIN B) NATURAL JOIN (C JOIN D)
```

この文には 2 つのテーブル式があります。テーブル式 `A JOIN B` のカラム名がテーブル式 `C JOIN D` のカラム名と比較されると、一致するカラム名のうちあいまいさのないペアに対してジョイン条件が生成されます。*一致するカラム名のうちあいまいさのないペア*とは、カラム名が両方のテーブル式に出現することがあっても同一テーブル式内では 2 回出現しないという意味です。

あいまいなカラム名のペアが 1 つでもあると、エラーになります。ただし、カラム名がもう一方のテーブル式内のカラム名とも一致しなければ、カラム名が同じテーブル式で 2 回出現しても構いません。

ナチュラル・ジョイン・リスト

ナチュラル・ジョインの少なくとも片方の側にテーブル式のリストがある場合、そのリスト内の各テーブル式に対して別のジョイン条件が生成されます。

次のテーブルを考えてみます。

- テーブル A はカラム a、b、c で構成されている。
- テーブル B は、カラム a と d で構成されている。
- テーブル C はカラム d と c で構成されている。

このような場合、ジョイン `(A,B) NATURAL JOIN C` によって次の 2 つのジョイン条件が生成されます。

```
ON A.c = C.c
  AND B.d = C.d
```

A-C または B-C に共通のカラム名がなければ、エラーが発行されます。

テーブル C がカラム a、d、c で構成されている場合、ジョイン `(A,B) NATURAL JOIN C` は無効です。その理由は、カラム a が 3 つのテーブルすべてに出現するため、ジョインがあいまいになってしまうためです。

例

次の例は、「販売した製品と販売担当者の情報を売り上げ別に提供してほしい」という質問に対する回答です。

```
SELECT *
 FROM (employee KEY JOIN sales_order)
   NATURAL JOIN (sales_order_items KEY JOIN product)
```

これは次と同義です。

```
SELECT *
 FROM (employee KEY JOIN sales_order)
   JOIN (sales_order_items KEY JOIN product)
     ON sales_order.id = sales_order_items.id
```

ビューと抽出テーブルのナチュラル・ジョイン

ANSI/ISO SQL 規格の拡張機能では、ナチュラル・ジョインのどちらの側にもビューまたは抽出テーブルを指定できます。次の文を考えてみます。

```
SELECT *
 FROM View1 NATURAL JOIN View2
```

View1 内のカラムは View2 内のカラムと比較されます。たとえば、両方のビューにカラム emp_id が出現し、それと同じ名前を持つカラムが他になければ、生成されるジョイン条件は `(View1.emp_id = View2.emp_id)` になります。

例

次の例で、ナチュラル・ジョインに使用するビューにはカラムだけでなく式を指定することができ、これらの式やカラムはナチュラル・ジョインでは同じように扱われることを説明します。まず、カラム x のあるビュー V を次のように作成します。

```
CREATE VIEW V(x) AS
 SELECT R.y + 1
 FROM R
```

次に、このビューから抽出テーブルへのナチュラル・ジョインを作成します。抽出テーブルには、カラム x があり、相関名 T が付けられています。

```
SELECT *
 FROM V NATURAL JOIN (SELECT P.y FROM P) as T(x)
```

このジョインは、次のクエリと同義です。

```
SELECT *
 FROM V JOIN (SELECT P.y FROM P) as T(x) ON (V.x = T.x)
```

キー・ジョイン

キー・ジョインを指定すると、データベース内の外部キー関係に基づいてジョイン条件が生成されます。キー・ジョインを使用するには、テーブル間に外部キー関係が必要になります。この関係がない場合は、エラーになります。

キー・ジョインは、ANSI/ISO SQL 規格の Sybase 版拡張機能です。キー・ジョインによって、より高度な機能は提供されませんが、特定のクエリを簡単に作成できます。

キー・ジョインがデフォルトの場合

次のすべてが当てはまる場合には、Adaptive Server 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、次のクエリは簡単なキー・ジョインであり、データベース内の外部キー関係に基づいてテーブル product と sales_order_items をジョインしています。

```
SELECT *
 FROM product KEY JOIN sales_order_items
```

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 product JOIN sales_order_items
```

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、Adaptive Server Anywhere がサンプル・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 product JOIN sales_order_items
 ON sales_order_items.prod_id = product.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 の場合は、上の例は次のクエリと同義です。

```
SELECT *
 FROM A JOIN B
 ON A.x = B.y AND A.w = B.z
```

複数の外部キー関係がある場合のキー・ジョイン

外部キー関係に基づいてジョイン条件が生成されるときに、外部キー関係が 2 つ以上ある場合があります。このような場合、Adaptive Server 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、Adaptive Server Anywhere でキー・ジョインのジョイン条件が生成される過程について説明します。このまとめについては、「キー・ジョイン操作規則」388 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 [外部キー] 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。

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」336 ページを参照してください。

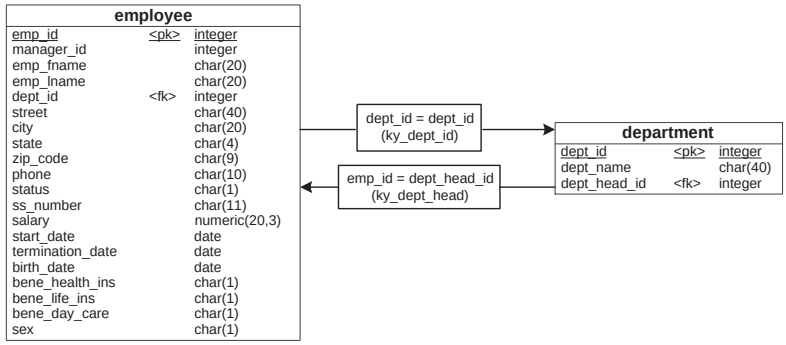
ジョイン条件の生成

Adaptive Server 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

サンプル・データベースでは、テーブル employee と department の間で 2 つの外部キー関係が定義されています。employee テーブルの外部キー ky_dept_id は department テーブルを参照し、department テーブルの外部キー ky_dept_head は employee 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 employee.emp_lname, department.dept_name
 FROM employee KEY JOIN department
```

例 2

このクエリは、department テーブルに相関名 ky_dept_id 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 ky_dept_id 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 employee.emp_lname, ky_dept_id.dept_name
 FROM employee KEY JOIN department AS ky_dept_id
```

次のクエリは上の例と同義です。この例では、department 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 employee.emp_lname, department.dept_name
 FROM employee JOIN department
     ON department.dept_id = employee.dept_id
```

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 ky_dept_head 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 employee に相関名 ky_dept_head を指定することで外部キー ky_dept_head を使用します。

```
SELECT ky_dept_head.emp_lname, department.dept_name
 FROM employee AS ky_dept_head KEY JOIN department
```

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 employee.emp_lname, department.dept_name
 FROM employee JOIN department
     ON department.dept_head_id = employee.emp_id
```

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、employee テーブルに外部キー department を定義するとします。

```
ALTER TABLE employee ADD FOREIGN KEY department
(dept_id) REFERENCES department (dept_id)
```

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 department 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 employee.emp_lname, department.dept_name
 FROM employee KEY JOIN department
```

注意

この例を Interactive SQL で実行する場合は、次の文を使ってサンプル・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 employee DROP FOREIGN KEY department
```

テーブル式のキー・ジョイン

Adaptive Server 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。Adaptive Server 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」379 ページ
- 「テーブル式リストのキー・ジョイン」381 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、Adaptive Server 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、Adaptive Server Anywhere はテーブル・ペア A-C と B-C の外部キー関係を調べて、ジョイン条件を 1 つだけ生成します。複数の外部キー関係が存在する場合は、次のキー・ジョイン決定規則に基づき、この 2 つのペアに対して 1 つのジョイン条件を生成します。

- まず、参照先となるプライマリ・キー・テーブルのうち、その 1 つの相関名と同じ役割名を持つ単一の外部キーを指定するために A-C および B-C の両方が調べられる。この基準に合う外部キーが 1 つだけ存在する場合には、それが使用される。テーブルの相関名と同じ役割名の外部キーが 2 つ以上ある場合、そのジョインはあいまいとみなされてエラーが発行される。
- テーブルの相関名と同じ名前の外部キーがない場合は、そのテーブルの外部キー関係が検索される。見つかった外部キー関係が 1 つであれば、それが使用される。2 つ以上見つかると、そのジョインはあいまいとみなされてエラーが発行される。
- 外部キー関係がまったく存在しない場合は、エラーが発行される。

詳細については、「複数の外部キー関係がある場合のキー・ジョイン」375 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 employee.emp_lname, ky_dept_id.dept_name
 FROM (employee KEY JOIN department as ky_dept_id)
   KEY JOIN sales_order
```

このクエリは次のように解釈できます。

- テーブル式 `(employee KEY JOIN department as ky_dept_id)` が調べられ、外部キー ky_dept_id に基づいてジョイン条件 `employee.dept_id = ky_dept_id.dept_id` が生成される。
- 次に、テーブル・ペア employee-sales_order と ky_dept_id-sales_order が調べられる。テーブル sales_order と employee 間、*および*、テーブル sales_order と ky_dept_id 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 sales_order と employee (ky_so_employee_id) 間には外部キー関係が 1 つだけ存在し、テーブル sales_order と ky_dept_id 間に外部キーは存在しない。したがって、ジョイン条件 `sales_order.emp_id = employee.sales_rep` 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 employee.emp_lname, department.dept_name
 FROM (employee JOIN department
     ON ( employee.dept_id = department.dept_id ) )
 JOIN sales_order
     ON (employee.emp_id = sales_order.sales_rep)
```

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、Adaptive Server 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
```

Adaptive Server Anywhere では、2 つのテーブル・ペア A-C と B-C のそれぞれに対してジョイン条件を生成することで、C と (A,B) をジョインするためのジョイン条件が生成されます。このように複数の外部キー関係が存在する場合は、次のキー・ジョイン規則に従って処理されます。

- 各ペアに対して、プライマリ・キー・テーブルの相関名と同じ名前の役割名を持つ外部キーが検索される。この基準に合う外部キーが 1 つだけ存在する場合には、それが使用される。2 つ以上見つかると、そのジョインはあいまいとみなされてエラーが発行される。
- 各ペアに、テーブルの相関名と同じ名前の外部キーがない場合は、そのテーブル間の外部キー関係が検索される。見つかった外部キー関係が 1 つであれば、それが使用される。2 つ以上見つかると、そのジョインはあいまいとみなされてエラーが発行される。
- 各ペアに、外部キー関係がまったく存在しない場合は、エラーが発行される。
- 各ペアにジョイン条件を 1 つだけ決定できる場合は、AND によってジョイン条件が組み合わされる。

詳細については、「複数の外部キー関係がある場合のキー・ジョイン」375 ページを参照してください。

例

次は、ある地域で 1 つ以上の注文を受けたすべての営業部員の名前を返すクエリです。

```
SELECT DISTINCT employee.emp_lname,
   ky_dept_id.dept_name, sales_order.region
 FROM (sales_order, department AS ky_dept_id)
    KEY JOIN employee
```

emp_lname	dept_name	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、sales_order と employee、および department AS ky_dept_id と employee を扱います。

sales_order と employee 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 (ky_so_employee_id) 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 `(employee.emp_id = sales_order.sales_rep)` になります。

department AS ky_dept_id と employee 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 ky_dept_id で、クエリ内の department 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 ky_dept_id を使用して作成されます。生成されるジョイン条件は `(employee.dept_id = ky_dept_id.dept_id)` 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 employee.emp_lname,
department.dept_name, sales_order.region
 FROM ( sales_order, department )
    JOIN employee
    ON employee.emp_id = sales_order.sales_rep
    AND employee.dept_id = department.dept_id
```

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、Adaptive Server Anywhere はテーブル式リストの各テーブルに対してジョイン条件を生成します。

たとえば、次の文は、テーブル式リストと、カンマを含まないテーブル式のキー・ジョインです。この例では、テーブル A とテーブル式 `C NATURAL JOIN D` のジョイン条件と、テーブル B とテーブル式 `C NATURAL JOIN D` のジョイン条件が生成されます。

```
SELECT *
 FROM (A,B) KEY JOIN (C NATURAL JOIN D)
```

ここでは、`(A,B)` がテーブル式リストで、`C NATURAL JOIN D` はテーブル式です。したがって、Adaptive Server 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 間の条件の*どちらか 1 つ*が生成されることで E に対するキー・ジョインのジョイン条件が生成されます。これについては、「カンマを含まないテーブル式のキー・ジョイン」379 ページを参照してください。

(A,B) と E の間にジョイン条件が生成される場合は、A-E と B-E に 1 つずつ、合計 2 つのジョイン条件が作成される必要があります。それぞれのテーブル・ペア内には有効な外部キー関係が見つけられる必要があります。これについては、「テーブル式リストのキー・ジョイン」381 ページを参照してください。

C NATURAL JOIN D と E の間にジョイン条件が作成される場合、ジョイン条件は 1 つだけ作成されます。そのために、C-E と D-E のペア内で外部キー関係が 1 つだけ見つけられます。これについては、「カンマを含まないテーブル式のキー・ジョイン」379 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 employee.emp_lname,
ky_dept_id.dept_name
 FROM (sales_order, department AS ky_dept_id)
   KEY JOIN (employee JOIN department AS d
     ON employee.emp_id = d.dept_head_id)
```

次の 2 つのジョイン条件が生成されます。

- テーブル・ペア sales_order-employee と sales_order-d の間には外部キー関係が 1 つだけ必要である。その外部キー関係とは、`sales_order.sales_rep = employee.emp_id` である。
- テーブル・ペア ky_dept_id-employee と ky_dept_id-d の間には外部キー関係が 1 つだけ必要である。その外部キー関係とは、`ky_dept_id.dept_id = employee.dept_id` である。

この例は次の文と同義です。次の例では、必ずしも相関名 `department AS ky_dept_id` を作成する必要はありません。相関名が必要になるのは、2 つの外部キーのうち employee と department のジョインで使用するキーを明確にするときだけです。

```
SELECT DISTINCT employee.emp_lname,
department.dept_name
 FROM (sales_order, department)
   JOIN (employee JOIN department AS d
       ON employee.emp_id = d.dept_head_id)
     ON sales_order.sales_rep = employee.emp_id
       AND department.dept_id = employee.dept_id
```

ビューと抽出テーブルのキー・ジョイン

キー・ジョインにビューまたは抽出テーブルが含まれている場合、Adaptive Server Anywhere ではテーブルと同じ基本手順に従います。ただし、次の点が異なります。

- 各キー・ジョインについて、クエリとビューの FROM 句内のテーブルのペアが調べられ、すべてのペアのセットに対してジョイン条件が *1* つ生成される。この場合、ビュー内の FROM 句にカンマやジョインのキーワードが含まれているかどうかは考慮されない。
- ビューまたは抽出テーブルの相関名と同じ役割名を持つ外部キーに基づいてテーブルがジョインされる。

- キー・ジョイン内にビューまたは抽出テーブルが含まれる場合、ビューには UNION、ORDER BY、DISTINCT、GROUP BY または、集合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、エラーが発行される。

抽出テーブルとビューの機能は同じです。抽出テーブルの場合、定義済みのビューを参照しないで、テーブルの定義を文中に記述する点だけが異なります。

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」378 ページで説明した複数の外部キー関係に関する規則に基づいて生成されます。ただし、ビューの参照テーブルではなく、ビューの相関名と同じ名前の外部キーを検索することが異なります。

上記のビュー定義のいずれか使用すると、`View1 KEY JOIN B` の処理を次のように解釈できます。

テーブル・ペア C-B と D-B が調べられ、ジョイン条件が 1 つだけ生成されます。複数の外部キー関係が存在する場合には、次のキー・ジョイン決定規則に基づいてジョイン条件が生成されます。

- まず、ビューの相関名と同じ役割名を持つ 1 つの外部キーに対して C-B と D-B が両方とも調べられる。この基準に合う外部キーが 1 つだけ存在する場合には、それが使用される。ビューの相関名と同じ役割名の外部キーが 2 つ以上ある場合、そのジョインはあいまいとみなされてエラーが発行される。
- ビューの相関名と同じ名前の外部キーがない場合は、そのテーブル間の外部キー関係が検索される。見つかった外部キー関係が 1 つであれば、それが使用される。2 つ以上見つかると、そのジョインはあいまいとみなされてエラーが発行される。
- 外部キー関係がまったく存在しない場合は、エラーが発行される。

ここで生成されたジョイン条件が `B.y = D.z` であるとすると、次に示す元のジョインを展開できます。

```
SELECT *
 FROM View1 KEY JOIN B
```

この文は次の文と同義です。

```
SELECT *
 FROM View1 JOIN B ON B.y = View1.z
```

詳細については、「複数の外部キー関係がある場合のキー・ジョイン」375 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 department.dept_name, employee.*
 FROM employee JOIN department
   ON employee.emp_id = department.dept_head_id
```

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 (sales_order, department ky_dept_id)
```

これは次と同義です。

```
SELECT *
 FROM V JOIN (sales_order, department ky_dept_id)
 ON (V.emp_id = sales_order.sales_rep
 AND V.dept_id = ky_dept_id.dept_id)
```

キー・ジョイン操作規則

以下の規則は、これまでに説明した情報をまとめたものです。

規則 1 : 2 つのテーブルのキー・ジョイン

この規則は A KEY JOIN B に適用されます。A と B はベース・テーブルまたはテンポラリ・テーブルです。

1. B を参照する A のすべての外部キーを見つける。

   テーブル B の相関名が役割名になる外部キーが存在する場合には、それを優先外部キーとする。

2. A を参照する B のすべての外部キーを見つける。

   テーブル A の相関名が役割名になる外部キーが存在する場合には、それを優先外部キーとする。

3. 優先キーが 2 つ以上存在する場合、そのジョインはあいまいである。構文エラー SQLE_AMBIGUOUS_JOIN (-147) が発行される。

4. 優先キーが 1 つだけ存在する場合には、この KEY JOIN 式に生成されたジョイン条件を定義するために、この外部キーが選択される。

5. 優先キーがまったくない場合は、A と B 間にある他の外部キーが使用される。

   - A と B の間に外部キーが 2 つ以上ある場合、そのジョインはあいまいである。構文エラー SQLE_AMBIGUOUS_JOIN (-147) が発行される。
   - 外部キーが 1 つだけ存在する場合には、この KEY JOIN 式に生成されたジョイン条件を定義するために、この外部キーが選択される。

- 外部キーがまったくない場合、ジョインは無効であり、エラーが生成される。

規則 2：カンマを含まないテーブル式のキー・ジョイン

この規則は `A KEY JOIN B` に適用されます。ここで、A と B はカンマを含まないテーブル式です。

1. テーブルの各ペア (テーブル式 A から 1 つとテーブル式 B から 1 つ) について、すべての外部キーをリストし、テーブル間のすべての優先キーにマークする。優先キー決定規則は上記の規則 1 で指定されている。
2. 優先キーが 2 つ以上存在する場合、そのジョインはあいまいである。構文エラー `SQLE_AMBIGUOUS_JOIN (-147)` が発行される。
3. 優先キーが 1 つだけ存在する場合には、この KEY JOIN 式に生成されたジョイン条件を定義するために、この外部キーが選択される。
4. 優先キーがまったくない場合は、テーブルのペア間にある他の外部キーが使用される。
   - 外部キーが 2 つ以上存在する場合、そのジョインはあいまいである。構文エラー `SQLE_AMBIGUOUS_JOIN (-147)` が発行される。
   - 外部キーが 1 つだけ存在する場合には、この KEY JOIN 式に生成されたジョイン条件を定義するために、この外部キーが選択される。
   - 外部キーがまったくない場合、ジョインは無効であり、エラーが生成される。

規則 3：テーブル式リストのキー・ジョイン

この規則は `(A1, A2, ...) KEY JOIN ( B1, B2, ...)` に適用されます。A1、B1、... はカンマを含まないテーブル式を表します。

1. テーブル式 Ai と Bj の各ペアについて、規則 1 または 2 を適用し、テーブル式 `(Ai KEY JOIN Bj)` にユニークな生成されたジョイン条件を見つける。テーブル式のペアの KEY JOIN のいずれかが規則 1 または 2 によってあいまいであると判断されると、構文エラーになる。

2. この KEY JOIN 式の生成されたジョイン条件は、手順 1 のジョイン条件の論理積である。

規則 4：カンマを含むテーブル式とリストのキー・ジョイン

この規則は (A1, A2, ...) KEY JOIN (B1, B2, ...) に適用されます。A1、B1、... はカンマを含むテーブル式を表します。

1. テーブル式 Ai と Bj の各ペアについて、規則 1、2、または 3 を適用し、テーブル式 (Ai KEY JOIN Bj) にユニークな生成されたジョイン条件を見つける。テーブル式のペアの KEY JOIN のいずれかが規則 1、2、または 3 によってあいまいであると判断されると、構文エラーになる。

2. この KEY JOIN 式の生成されたジョイン条件は、手順 1 のジョイン条件の論理積である。

第 9 章

共通テーブル式

この章の内容

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、クエリ内でテンポラリ・ビューのように使用できます。この章では、共通テーブル式を使用する方法について説明します。

共通テーブル式について

共通テーブル式は、1 つの SELECT 文のスコープ内のみで認識されるテンポラリ・ビューです。共通テーブル式によって、クエリをより簡単に記述できます。また、共通テーブル式なしでは記述できないクエリもあります。

クエリに複数の集合関数が含まれる場合やストアド・プロシージャ内でプログラム変数を参照するビューを定義する場合に、共通テーブル式を使用すると便利です。また、共通テーブル式を使用する必要がある場合もあります。さらに、共通テーブル式は値のセットを一時的に格納する際に便利です。

再帰共通テーブル式によって、たとえば社内の報告関係など、階層情報を表すテーブルに問い合わせできます。また、再帰共通テーブル式は、構成部品の問題や最短距離の問題の解決にも使用できます。

再帰クエリの詳細については、「再帰共通テーブル式」401 ページを参照してください。

どの部署の従業員数が最も多いかを特定する問題を例にとります。サンプル・データベースの employee 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 dept_id, count(*) AS n
 FROM employee
 GROUP BY dept_id
```

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 dept_id, n
 FROM ( SELECT dept_id, count(*) AS n
        FROM employee GROUP BY dept_id ) AS a
 WHERE a.n =
   ( SELECT max(n)
     FROM ( SELECT dept_id, count(*) AS n
            FROM employee GROUP BY dept_id ) AS b )
```

この文は正確な結果をもたらしますが、いくつか欠点もあります。最初の欠点は、サブクエリの繰り返しによってこの文がぎこちなくなっていることです。2 つめの欠点は、この文ではサブクエリ間の関係が明確でないことです。

このような問題を回避する 1 つの方法として、ビューを作成し、そのビューを使用してクエリを再作成します。この方法によって、前述の問題を回避できます。

```
CREATE VIEW CountEmployees(dept_id, n) AS
     SELECT dept_id, count(*) AS n
     FROM employee GROUP BY dept_id;

SELECT dept_id, n
 FROM CountEmployees
 WHERE n = ( SELECT max(n)
             FROM CountEmployees );
```

この方法の欠点は、ビューの作成時にエンジンがシステム・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 度しか使用しない場合は、代わりに共通テーブル式を使用することをおすすめします。

共通テーブル式の使用

共通テーブル式は、WITH 句を使用して定義します。WITH 句は、SELECT 文内の SELECT キーワードに先行します。句の内容は、1 つ以上のテンポラリ・ビューを定義します。これらのテンポラリ・ビューは、文内の他の場所から参照できます。この句の構文は、CREATE VIEW 文の構文とよく似ています。共通テーブル式を使用して、前述のクエリを次のように記述できます。

```
WITH CountEmployees(dept_id, n) AS
   ( SELECT dept_id, count(*) AS n
     FROM employee GROUP BY dept_id )
 SELECT dept_id, 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(dept_id, n) AS
   ( SELECT dept_id, count(*) AS n
     FROM employee GROUP BY dept_id )
 SELECT dept_id, n
 FROM CountEmployees
 WHERE n = ( SELECT min(n)
             FROM CountEmployees )
```

サンプル・データベースでは、従業員数が最も少ない 9 人編成の部署は 2 つあります。

複数の相関名

テーブルを使用するときに、共通テーブル式の複数のインスタンスに異なる相関名をつけることができます。これによって、共通テーブル式をそれ自体にジョインできます。たとえば、次のクエリは、従業員数が同じ部署を組み合わせます。ただし、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(dept_id, n) AS
     ( SELECT dept_id, count(*) AS n
       FROM employee GROUP BY dept_id )
 SELECT a.dept_id, a.n, b.dept_id, b.n
 FROM CountEmployees AS a JOIN CountEmployees AS b
 ON a.n = b.n AND a.dept_id < b.dept_id
```

複数のテーブル式

1 つの WITH 句で、2 つ以上の共通テーブル式を定義できます。これらの定義はカンマで区切る必要があります。次の例は、支払い給与総額の最も少ない部署と、従業員数が最も多い部署をリストします。

```
WITH
   CountEmployees(dept_id, n) AS
     ( SELECT dept_id, count(*) AS n
       FROM employee GROUP BY dept_id ),
   DeptPayroll( dept_id, amt ) AS
      ( SELECT dept_id, sum(salary) AS amt
        FROM employee GROUP BY dept_id )
 SELECT count.dept_id, count.n, pay.amt
 FROM CountEmployees AS count JOIN DeptPayroll AS pay
 ON count.dept_id = pay.dept_id
 WHERE count.n = ( SELECT max(n) FROM CountEmployees )
    OR pay.amt = ( SELECT min(amt) FROM DeptPayroll )
```

共通テーブル式を使用できる条件

共通テーブル式はクエリ本体または任意のサブクエリ内から参照可能ですが、共通テーブル式を定義できるのは 3 か所のみです。

- **最上位レベルの SELECT 文**　共通テーブル式は、最上位レベルの SELECT 文内で使用できますが、サブクエリ内では使用できません。

```
WITH DeptPayroll( dept_id, amt ) AS
   ( SELECT dept_id, sum(salary) AS amt
     FROM employee GROUP BY dept_id )
 SELECT dept_id, 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 (dept_id, size, pay) AS
    WITH
      CountEmployees(dept_id, n) AS
        ( SELECT dept_id, count(*) AS n
          FROM employee GROUP BY dept_id ),
      DeptPayroll( dept_id, amt ) AS
        ( SELECT dept_id, sum(salary) AS amt
          FROM employee GROUP BY dept_id )
   SELECT count.dept_id, count.n, pay.amt
   FROM CountEmployees count JOIN DeptPayroll pay
   ON count.dept_id = pay.dept_id
   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。

```
INSERT INTO LargestPayrolls (dept_id, payroll,
date)
   WITH DeptPayroll( dept_id, amt ) AS
     ( SELECT dept_id, sum(salary) AS amt
       FROM employee
       GROUP BY dept_id )
   SELECT dept_id, amt, CURRENT TIMESTAMP
   FROM DeptPayroll
   WHERE amt = ( SELECT max(amt)
                 FROM DeptPayroll )
```

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、他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( dept_id, amt ) AS
     ( SELECT dept_id, sum(salary) AS amt
       FROM employee GROUP BY dept_id )
 SELECT dept_id, amt
 FROM DeptPayroll
 WHERE amt = ( SELECT max(amt)
               FROM DeptPayroll )
```

プログラム変数を参照するビュー

プログラム変数への参照を含むビューを作成すると便利な場合があります。たとえば、特定の顧客を識別する変数をプロシージャ内に定義するとします。そして、その顧客の購入履歴を問い合わせたいとしま

す。何度も同様の情報にアクセスしたり、複数の集合関数を使用したりする必要がある場合は、その特定の顧客に関する情報を含むビューを作成すると便利です。

プログラム変数を参照するビューは作成できません。なぜならば、ビューの範囲をプロシージャの範囲に制限する方法がないからです。一度作成すると、ビューは他の場合にも使用できます。しかし、プロシージャのクエリ内で共通テーブル式を使用できます。共通テーブル式の範囲はその文に制限されるため、変数の参照にあいまい性はありませんので、許可されます。

次の文は、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 emp_fname || ' ' || emp_lname AS sales_rep_name,
        sales_rep AS sales_rep_id,
        sum( p.unit_price * i.quantity ) AS total_sales
 FROM employee LEFT OUTER JOIN sales_order AS o
               INNER JOIN sales_order_items AS i
               INNER JOIN product AS p
 WHERE '2000-01-01' <= order_date
                   AND order_date < '2001-01-01'
 GROUP BY sales_rep, emp_fname, emp_lname
```

前述のクエリは、次のプロシージャ内で使用されている共通テーブル式の基礎となります。調査する販売担当者の ID 番号と年度が入力パラメータです。このプロシージャからわかるように、WITH 句内では、プロシージャ・パラメータと宣言されたどのローカル変数も参照できます。

```
CREATE PROCEDURE sales_rep_total (
   IN rep  INTEGER,
   IN yyyy INTEGER )
 BEGIN
   DECLARE start_date DATE;
   DECLARE end_date   DATE;
   SET start_date = YMD( yyyy,  1,  1 );
   SET   end_date = YMD( yyyy, 12, 31 );
   WITH total_sales_by_rep ( sales_rep_name,
                             sales_rep_id,
                             month,
                             order_year,
                             total_sales ) AS
   ( SELECT emp_fname || ' ' || emp_lname AS
sales_rep_name,
```

```
            sales_rep AS sales_rep_id, month(
order_date),
            year(order_date),
            sum( p.unit_price * i.quantity ) AS
total_sales
     FROM employee LEFT OUTER JOIN sales_order o
                        INNER JOIN sales_order_items i
                        INNER JOIN product p
     WHERE start_date <= order_date AND
                         order_date <= end_date AND
           sales_rep = rep
     GROUP BY year(order_date), month(order_date),
              emp_fname, emp_lname, sales_rep )
   SELECT sales_rep_name,
             monthname( YMD(yyyy, month, 1) ) AS
month_name,
             order_year,
             total_sales
   FROM total_sales_by_rep
   WHERE total_sales =
     ( SELECT max( total_sales) FROM total_sales_by_rep
)
   ORDER BY order_year ASC, month ASC;
 END;
```

次の文は、前述のプロシージャの呼び出し方法を示しています。

```
CALL sales_rep_total(129, 2000);
```

値を格納するビュー

SELECT 文内またはプロシージャ内に特定の値のセットを格納すると便利な場合があります。たとえば、ある会社が販売担当者の成績を四半期ごとでなく、1 年を 3 期に分けて分析する方法を採用しているとします。1 年を 3 期に分けた日付セットが四半期のようには組み込まれていないため、プロシージャ内で日付を格納する必要があります。

```
WITH thirds (q_name, q_start, q_end) AS
 ( SELECT 'T1', '2000-01-01', '2000-04-30' UNION
   SELECT 'T2', '2000-05-01', '2000-08-31' UNION
   SELECT 'T3', '2000-09-01', '2000-12-31' )
 SELECT q_name,
        sales_rep,
        count(*) AS num_orders,
        sum( p.unit_price * i.quantity ) AS total_sales
 FROM thirds LEFT OUTER JOIN sales_order AS o
```

```
      ON q_start <= order_date AND order_date <= q_end
                     INNER JOIN sales_order_items AS i
                     INNER JOIN product AS p
  GROUP BY q_name, sales_rep
  ORDER BY q_name, sales_rep
```

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,
          sales_rep,
          count(*) AS num_orders,
          sum(p.unit_price * i.quantity) AS total_sales
   FROM thirds JOIN sales_order AS o
     ON q_start <= order_date AND order_date <= q_end
             KEY JOIN sales_order_items AS i
             KEY JOIN product AS p
   GROUP BY q_name, sales_rep
   ORDER BY q_name, sales_rep;
 END;

CALL sales_by_third (2000);
```

再帰共通テーブル式

共通テーブル式は、再帰できます。WITH の直後に RECURSIVE キーワードを使用すると、共通テーブル式は再帰的になります。1 つの WITH 句には、複数の再帰式を含めることもできます。また再帰共通テーブル式と非再帰共通テーブル式の両方を含めることができます。

再帰共通テーブル式は、階層の任意の深さに対する関係を返すクエリを記述するのに便利な方法です。たとえば、社内の報告関係を表すテーブルがある場合、特定の 1 人に報告を行うすべての従業員を返すクエリを間単に記述できます。

クエリの記述方法によって、再帰レベル数を制限することも、まったく制限しないことも可能です。たとえば、レベル数を制限して、トップレベルの管理者のみを返すことができますが、指揮系統が予期したよりも長い場合、除外される従業員も出てきます。レベル数を制限しない場合は、従業員が除外されることはありません。ただし、たとえばある従業員が直接または間接的に自分自身に報告するなど、図にあらゆる循環が含まれる場合、無限の再帰となる可能性があります。この状況は、たとえば会社の従業員が取締役会の一員である場合などに、社内の管理階層内で発生することがあります。

再帰を使用すると、ツリー、またはツリーに似たデータ構造のテーブルをトラバースすることがはるかに簡単になります。再帰式を使用せずに単一の文内でそうした構造をトラバースする唯一の方法は、考えられる各レベルごとに 1 回ずつテーブルをそれ自体にジョインすることです。たとえば、報告階層に含まれるレベルが最大で 7 つある場合、employee テーブルをそれ自体に 7 回ジョインする必要があります。会社の組織が変更され、新しい管理レベルが導入された場合は、クエリを再度記述する必要があります。

再帰共通テーブル式には、*初期サブクエリ* (シード) と、各繰り返しの間に結果セットにローを追加する*再帰サブクエリ*が含まれます。この 2 つの部分は、UNION ALL 演算子を使用してのみ接続できます。初期サブクエリは、通常の非再帰クエリで、最初に処理されます。再帰部分には、直前の繰り返しで追加されたローへの参照が含まれます。再帰は、繰り返しによって新しいローが生成されなくなったときに自動的に停止します。直前の繰り返しより前に選択されたローを参照する方法はありません。

再帰サブクエリの select リストは、初期サブクエリの select リストと数およびデータ型が一致する必要があります。データ型の自動変換が行えない場合は、一方のサブクエリの結果を明示的にキャストして、もう一方のサブクエリのデータ型に一致させます。

共通テーブル式の詳細については、「共通テーブル式について」392 ページを参照してください。

階層データの選択

次のクエリは、管理レベルで従業員をリストする方法を示しています。level 0 は、管理者を持たない従業員を表します。level 1 は、level 0 の管理職の 1 人に直接報告を行う従業員を表し、level 2 は、level 1 の管理職に直接報告を行う従業員を表します。以下、同様に続きます。

```
WITH RECURSIVE
   manager ( emp_id, manager_id,
             emp_fname, emp_lname, mgmt_level ) AS
 ( ( SELECT emp_id, manager_id,       -- initial
subquery
            emp_fname, emp_lname, 0
     FROM employee AS e
     WHERE manager_id = emp_id )
   UNION ALL
   ( SELECT e.emp_id, e.manager_id,   -- recursive
subquery
            e.emp_fname, e.emp_lname, m.mgmt_level + 1
     FROM employee AS e JOIN manager AS m
      ON   e.manager_id =  m.emp_id
       AND e.manager_id <> e.emp_id
       AND m.mgmt_level < 20 ) )
 SELECT * FROM manager
 ORDER BY mgmt_level, emp_lname, emp_fname
```

再帰クエリ内の条件として管理レベルを 20 未満に制限するのは、重要な予防措置です。これによって、テーブル・データに循環が含まれる場合も無限再帰を防げます。

MAX_RECURSIVE_ITERATIONS オプション

MAX_RECURSIVE_ITERATIONS オプションは、暴走する再帰クエリを捕らえることを目的として設計されています。このオプションのデフォルト値は 100 です。この再帰レベル数を超えた再帰クエリは終了し、エラーを発生させます。

このオプションがあれば停止条件は重要でないように見えるかもしれませんが、通常はそうではありません。各繰り返しの間に選択されるローの数は、急激に増える可能性があり、最大に達する前にデータベースのパフォーマンスに深刻な影響を与えることがあります。再帰クエリ内に停止条件を設けるのは、各状況に適した制限を設定する手段です。

再帰共通テーブル式に関する制限

再帰共通テーブル式に適用される制限を次に示します。

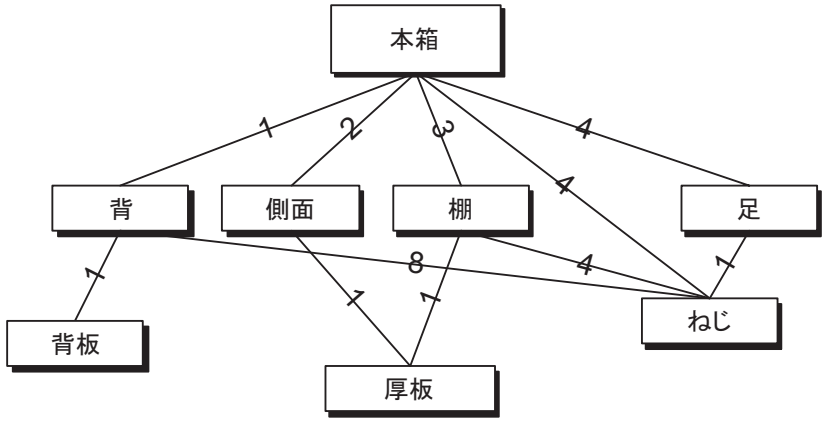
- 他の再帰共通テーブル式への参照は、再帰共通テーブル式の定義内に含められません。したがって、再帰共通テーブル式は相互に再帰できません。ただし、非再帰共通テーブル式には、再帰共通テーブル式への参照を含められます。また再帰共通テーブル式には、非再帰共通テーブル式への参照を含められます。
- 初期サブクエリと再帰サブクエリ間で唯一許可されている set 演算子は、UNION ALL です。他の set 演算子は許可されていません。
- 再帰サブクエリ定義内では、再帰テーブル式への自己参照は再帰サブクエリの FROM 句内のみに含めることができます。
- 再帰サブクエリの FROM 句内に自己参照が含まれる場合、再帰テーブルへの参照は外部ジョインの NULL 入力側に含められません。
- 再帰サブクエリには、DISTINCT 句、GROUP BY 句、ORDER BY 句は含められません。
- 再帰サブクエリでは、集合関数はすべて使用できません。

- 再帰サブクエリの暴走を防ぐために、再帰レベル数が MAX_RECURSIVE_ITERATIONS オプションの現在の設定を超えるとエラーが生成されます。このオプションのデフォルト値は 100 です。

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
本箱	背	1
本棚	側面	2
本棚	棚	3
本棚	脚	4

コンポーネント	サブコンポーネント	数量
本棚	ねじ	4
背	背板	1
背	ねじ	8
側面	厚板	1
棚	厚板	1
棚	ねじ	4

次の文は、bookcase テーブルを作成し、前述のテーブルに示されたデータを挿入します。

```
CREATE TABLE bookcase (
        component      VARCHAR(9),
        subcomponent   VARCHAR(9),
     quantity       integer,
      PRIMARY KEY (component, subcomponent)
 );

INSERT INTO bookcase
   SELECT 'bookcase', 'back',      1 UNION
   SELECT 'bookcase', 'side',      2 UNION
   SELECT 'bookcase', 'shelf',     3 UNION
   SELECT 'bookcase', 'foot',      4 UNION
   SELECT 'bookcase', 'screw',     4 UNION
   SELECT 'back',     'backboard', 1 UNION
   SELECT 'back',     'screw',     8 UNION
   SELECT 'side',     'plank',     1 UNION
   SELECT 'shelf',    'plank',     1 UNION
   SELECT 'shelf',    'screw',     4;
```

bookcase テーブルを作成したら、前述の部品テーブルを次のクエリを使用して再作成できます。

```
SELECT * FROM bookcase
 ORDER BY component, subcomponent;
```

このテーブルが作成できたら、本棚の組み立てに必要な基本部品と各必要数量のリストを生成できます。

```
WITH RECURSIVE parts (component, subcomponent,
quantity) AS
 ( SELECT component, subcomponent, quantity
   FROM bookcase WHERE component = 'bookcase'
     UNION ALL
   SELECT b.component, b.subcomponent, p.quantity *
b.quantity
   FROM parts p JOIN bookcase b ON p.subcomponent =
b.component )
 SELECT subcomponent, sum(quantity) AS quantity
 FROM parts
 WHERE subcomponent NOT IN ( SELECT component FROM
bookcase )
 GROUP BY subcomponent
 ORDER BY subcomponent;
```

このクエリの結果を次に示します。

サブコンポーネント	数量
背板	1
脚	4
厚板	5
ねじ	24

もう 1 つの選択肢として、このクエリを別の再帰レベルを実行するように書き直すと、メインの SELECT 文内でサブクエリを記述する必要がなくなります。

```
WITH RECURSIVE parts (component, subcomponent,
quantity) AS
 ( SELECT component, subcomponent, quantity
   FROM bookcase WHERE component = 'bookcase'
     UNION ALL
   SELECT p.subcomponent, b.subcomponent,
     IF b.quantity IS NULL
     THEN p.quantity
     ELSE p.quantity * b.quantity
     ENDIF
   FROM parts p LEFT OUTER JOIN bookcase b
   WHERE p.subcomponent = b.component AND
         p.subcomponent IS NOT NULL
```

```
 )
SELECT component, sum(quantity) FROM parts
WHERE subcomponent IS NULL
GROUP BY component
ORDER BY component;
```

このクエリの結果は、前述のクエリの結果とまったく同じです。

再帰共通テーブル式でのデータ型宣言

テンポラリ・ビュー内のカラムのデータ型は、初期サブクエリのデータ型によって定義されます。再帰サブクエリのカラムのデータ型は、一致する必要があります。データベース・サーバは、再帰サブクエリによって返された値がその初期クエリの値に一致するよう、自動的に変換しようとします。これが不可能な場合、または変換で情報が失われた場合、エラーが生成されます。

通常、初期サブクエリがリテラル値または NULL を返す場合、明示的なキャストが必要な場合がほとんどです。初期サブクエリが再帰サブクエリとは異なるカラムから値を選択する場合も、明示的なキャストが必要な場合があります。

キャストは、初期サブクエリのカラムが再帰サブクエリのカラムと同じドメインを持っていない場合に必要な場合があります。初期サブクエリでは、NULL 値に常にキャストを適用する必要があります。

たとえば、本棚の構成部品のサンプルは、初期サブクエリが bookcase テーブルからのローを返すことによって、選択されたカラムのデータ型を継承するため、正しく動作します。

詳細については、『ASA SQL ユーザーズ・ガイド』> 「構成部品の問題」を参照してください。

このクエリが次のように書き直された場合は、明示的なキャストが必要です。

```
WITH RECURSIVE parts (component, subcomponent,
quantity) AS
 ( SELECT NULL, 'bookcase', 1         -- ERROR! Wrong
domains!
     UNION ALL
   SELECT b.component, b.subcomponent,
          p.quantity * b.quantity
   FROM parts p JOIN bookcase b
     ON p.subcomponent = b.component )
 SELECT * FROM parts
 ORDER BY component, subcomponent
```

キャストを行わない場合、次の理由でエラーが発生します。

- コンポーネント名の正しいデータ型は VARCHAR だが、最初のカラムは NULL である。
- 数字 1 は short integer と見なされるが、quantity カラムのデータ型は INT である。

2 番目のカラムには、キャストは不要です。初期クエリのこのカラムがすでに文字列であるためです。

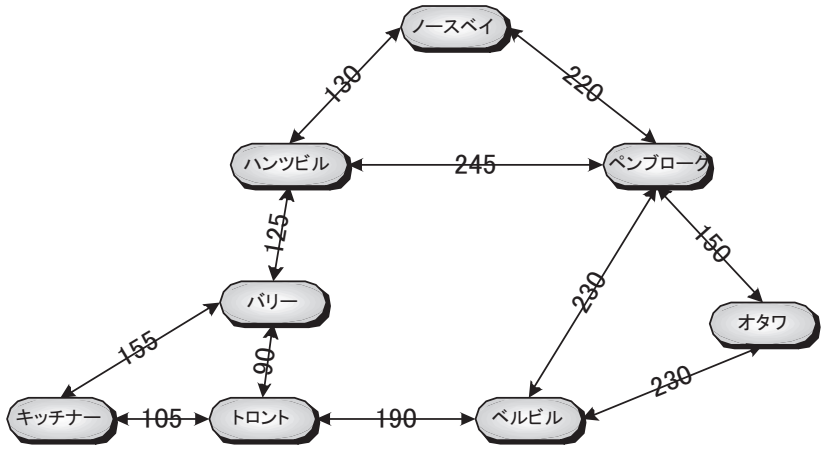
初期サブクエリでデータ型をキャストすれば、クエリを意図したとおりに動作させることができます。

```
WITH RECURSIVE parts (component, subcomponent,
quantity) AS
 ( SELECT CAST(NULL AS VARCHAR),  -- CASTs must be used
          'bookcase',             -- to declare the
          CAST( 1   AS INT)       -- correct datatypes
       UNION ALL
   SELECT b.component, b.subcomponent,
          p.quantity * b.quantity
   FROM parts p JOIN bookcase b
     ON p.subcomponent = b.component )
 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      AS VARCHAR(10),
     destination AS VARCHAR(10),
     distance    AS INT,
   PRIMARY KEY (origin, destination)
 );

INSERT INTO travel
   SELECT 'Kitchener',  'Toronto',    105 UNION
   SELECT 'Kitchener',  'Barrie',     155 UNION
   SELECT 'North Bay',  'Pembroke',   220 UNION
   SELECT 'Pembroke',   'Ottawa',     150 UNION
   SELECT 'Barrie',     'Toronto',     90 UNION
   SELECT 'Toronto',    'Belleville', 190 UNION
   SELECT 'Belleville', 'Ottawa',     230 UNION
   SELECT 'Belleville', 'Pembroke',   230 UNION
   SELECT 'Barrie',     'Huntsville', 125 UNION
   SELECT 'Huntsville', 'North Bay',  130 UNION
   SELECT 'Huntsville', 'Pembroke',   245;

INSERT INTO travel   -- Insert the return trips
   SELECT destination, origin, distance
   FROM travel;
```

次に、再帰共通テーブル式を記述します。移動はキッチナー市から始まるため、初期サブクエリは、キッチナー市を起点とする可能なすべてのパスを、その距離とともに選択することから始まります。

再帰サブクエリは、パスを拡張します。各パスに対し、再帰サブクエリは、直前のセグメントの目的地から続くセグメントを追加し、新しいセグメントの距離を加えて、各ルートの現在のトータル・コストを管理します。効率をあげるために、次の条件のいずれかに該当した場合、ルートは終端されます。

- パスが出発地に戻る場合。
- パスが直前の地点に戻る場合。
- パスが目的地に達した場合。

この例では、どのパスもキッチナー市に戻ってはならず、全てのパスはペンブローク市に達した場合、終端されなければなりません。

環状のグラフを探索するために再帰クエリを使用する場合は、再帰クエリが適切に終了するように保証することが特に重要です。この例の場合、前述の条件では不十分です。ルートに 2 つの中継地の間を行ったり来たりする移動が無制限に含まれる可能性があるからです。次の再帰クエリは、どのルートのセグメント数も最大 7 つまでに制限することで、ルートの終わりを保証しています。

このクエリ例のポイントは現実的なルートを選択することであるため、メイン・クエリでは、距離が最短ルート比 50% 増未満のルートのみを選択しています。

```
WITH RECURSIVE
     trip (route, destination, previous, distance,
segments) AS
 ( SELECT CAST(origin || ', ' || destination AS
VARCHAR(256)),
          destination, origin, distance, 1
   FROM travel
   WHERE origin = 'Kitchener'
     UNION ALL
   SELECT route || ', ' || v.destination,
          v.destination,            -- current endpoint
          v.origin,                -- previous endpoint
          t.distance + v.distance,  -- total distance
          segments + 1              -- total number of
segments
   FROM trip t JOIN travel v ON t.destination = v.origin
   WHERE v.destination <> 'Kitchener'  -- Don't return
to start
     AND v.destination <> t.previous   -- Prevent
backtracking
     AND v.origin     <> 'Pembroke'   -- Stop at the end
     AND segments                      -- TERMINATE
RECURSION!
           < ( SELECT count(*)/2 FROM travel ) )
 SELECT route, distance, segments FROM trip
 WHERE destination = 'Pembroke' AND
       distance < 1.5 * ( SELECT min(distance)
                          FROM trip
                        WHERE destination = 'Pembroke' )
 ORDER BY distance, segments, route;
```

この文を前述のデータ・セットに対して実行すると、次のような結果となります。

ルート	距離	セグメント数
キッチナー、バリー、ハンツビル、ペンブローク	525	3
キッチナー、トロント、ベルビル、ペンブローク	525	3
キッチナー、トロント、バリー、ハンツビル、ペンブローク	565	4
キッチナー、バリー、ハンツビル、ノースベイ、ペンブローク	630	4
キッチナー、バリー、トロント、ベルビル、ペンブローク	665	4
キッチナー、トロント、バリー、ハンツビル、ノースベイ、ペンブローク	670	5
キッチナー、トロント、ベルビル、オタワ、ペンブローク	675	4

複数の再帰共通テーブル式の使用

再帰クエリには、複数の再帰クエリを含めることができますが、それらの再帰クエリが共通の要素を持たず互いに素であることが条件となります。また、再帰クエリには再帰共通テーブル式と非再帰共通テーブル式を混在させることができます。共通テーブル式が 1 つでも再帰的である場合は、RECURSIVE キーワードを含める必要があります。

たとえば、前述のクエリと同じ結果を返す次のクエリは、別の非再帰共通テーブル式を使用して最短ルートの距離を選択しています。2 つめの共通テーブル式の定義は、1 つめの定義からカンマで区切られています。

```
WITH RECURSIVE
   trip (route, destination, previous, distance,
segments) AS
     ( SELECT CAST(origin || ', ' || destination AS
VARCHAR(256)),
              destination, origin, distance, 1
       FROM travel
       WHERE origin = 'Kitchener'
       UNION ALL
       SELECT route || ', ' || v.destination,
              v.destination,
              v.origin,
              t.distance + v.distance,
              segments + 1
       FROM trip t JOIN travel v ON t.destination =
v.origin
       WHERE v.destination <> 'Kitchener'
         AND v.destination <> t.previous
         AND v.origin      <> 'Pembroke'
         AND segments
               < ( SELECT count(*)/2 FROM travel ) ),
   shortest ( distance ) AS                 --
Additional,
     ( SELECT min(distance)                 -- non-
recursive
       FROM trip                            -- common
table
       WHERE destination = 'Pembroke' )     --
expression
```

```
 SELECT route, distance, segments FROM trip
 WHERE destination = 'Pembroke' AND
       distance < 1.5 * ( SELECT distance FROM shortest
)
 ORDER BY distance, segments, route;
```

非再帰共通テーブル式と同様、再帰式をストアド・プロシージャ内で使用した場合、ローカル変数やプロシージャ・パラメータへの参照を含められます。たとえば、次に定義する best_routes プロシージャは、指定した 2 つの都市の間の最短ルートを特定します。

```
CREATE PROCEDURE best_routes (
    IN initial VARCHAR(10),
    IN final   VARCHAR(10)
 )
 BEGIN
   WITH RECURSIVE
       trip (route, destination, previous, distance,
segments) AS
   ( SELECT CAST(origin || ', ' || destination AS
VARCHAR(256)),
            destination, origin, distance, 1
     FROM travel
     WHERE origin = initial
       UNION ALL
     SELECT route || ', ' || v.destination,
           v.destination,           -- current endpoint
           v.origin,               -- previous endpoint
            t.distance + v.distance,  -- total distance
            segments + 1             -- total number of
segments
     FROM trip t JOIN travel v ON t.destination =
v.origin
     WHERE v.destination <> initial     -- Don't return
to start
       AND v.destination <> t.previous  -- Prevent
backtracking
       AND v.origin      <> final       -- Stop at the
end
       AND segments                     -- TERMINATE
RECURSION!
             < ( SELECT count(*)/2 FROM travel ) )
   SELECT route, distance, segments FROM trip
   WHERE destination = final AND
```

```
        distance < 1.4 * ( SELECT min(distance)
                           FROM trip
                           WHERE destination = final )
  ORDER BY distance, segments, route;
END;

CALL best_routes ( 'Pembroke', 'Kitchener' );
```

第 10 章

OLAP の使用

この章の内容

この章では、オンライン分析処理 (OLAP) 機能について説明します。OLAP は、リレーショナル・データベースに格納されている情報に関するデータ分析手法の 1 つです。OLAP を使用すると、小計ローを含む結果セットを取得し、データを多次元のキューブに編成できます。OLAP では、フィルタを使用してデータを掘り下げることができ、結果セットを迅速に返すことができるように設計されています。

OLAP の概要

オンライン分析処理 (OLAP) により、リレーショナル・データベース内に格納されているデータを分析できます。OLAP を使用して取得した結果セットには小計ローが含まれ、それを多次元のキューブに編成することができます。

OLAP により、時系列分析、コスト配分、ゴールシーク、特定の多次元構造変更、非手続き型モデリング、例外の警告、データ・マイニングなどを実行できます。Adaptive Server Anywhere にはデータに対する多次元概念ビューが用意されており、階層および複数の階層を完全にサポートしています。

OLAP 機能により、次のタスクを実行できます。

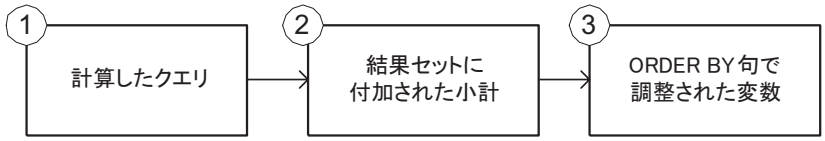
- 多次元にまたがって適用される計算
- 複数の階層にまたがって適用される計算
- 時系列の傾向分析
- 基本データへのリーチスルー

データベース製品の中には、独立した OLAP モジュールを提供しているものもありますが、この場合 OLAP モジュールへデータを移動してからデータを分析する必要があります。Adaptive Server Anywhere では、OLAP 機能をデータベース自身に組み込んでいます。これにより、(ストアド・プロシージャなどの) 他のデータベース機能とのシームレスな配備や統合が容易になりました。

小計の概要

小計ローはデータの分析を支援するもので、特に大量のデータ、多次元のデータ、複数のテーブルに含まれているデータ、または複数のデータベースを一緒にしたような場合に有効です。たとえば、販売マネージャは、販売担当別、地域別、四半期別の売上高のレポートを検索でき、売り上げのパターンを把握するのに便利です。データの小計により、販売マネージャはさまざまな側面から売り上げの全体像を把握できます。販売マネージャが比較したい基準に基づいて集計情報を提供すると、データの分析がさらに容易になります。

OLAP を使用することで、ローとカラムの小計の分析と計算のプロセスがエンド・ユーザに見えないようになります。次の図は、Adaptive Server Anywhere による小計の作成方法の概念を示しています。



1. このステップで中間結果セットを取得できます。クエリは、設定されている条件に応じて集計されます。
2. OLAP が小計を作成してそれを結果セットに付加します。
3. ローは、クエリ内の ORDER BY 句に応じて整列されます。

注意
このシーケンスは、エンド・ユーザにはシームレスとなっています。

OLAP 演算について

OLAP 演算 (ROLLUP、CUBE、および GROUPING SETS) について、次のように考えます。

- クエリ用の「**プレフィクス**」リストが構成されます。プレフィクスとは、*group-by-list* 内の項目のサブセットです。

プレフィクスは、クエリの *group-by-list* 内の項目から最も右側の 1 つまたは複数の項目を除外することで構成されます。残ったカラムは、「**プレフィクス・カラム**」と呼ばれます。

次に、ROLLUP クエリの例を示します。

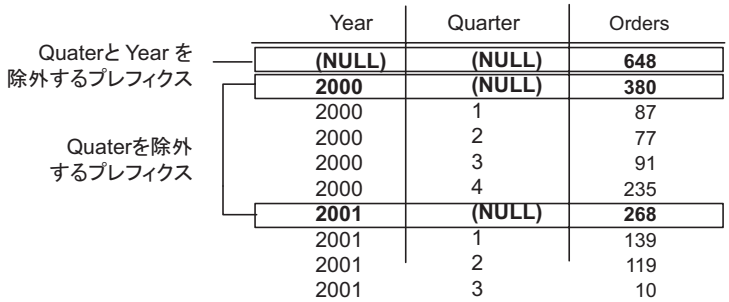
```
SELECT year (order_date) Year, quarter (order_date)
Quarter, count (*) Orders
 FROM sales_order
 GROUP BY ROLLUP (Year, Quarter)
 ORDER BY Year, Quarter
```

たとえば、上記の例では、「*group-by-list*」には 2 つの変数 (**Year** と **Quarter**) が含まれています。

GROUP BY ROLLUP（**Year**，**Quarter**）

このクエリには、プレフィクスが 2 つあります。

- **Quarter を除外するプレフィクス**　プレフィクス・カラム・セットには、カラム Year のみが含まれます。
- **Quarter と Year の両方を除外するプレフィクス**　プレフィクス・カラムはありません。



注意

プレフィクスの数は、*group-by-list* 内の項目数と同じです。

- プレフィクスごとに、プレフィクス・カラムが同じ値のすべてのローに対応する小計ローが構成されます。

  たとえば、もう一度次のクエリについて考えてみます。

```
SELECT year (order_date) AS Year, quarter
(order_date) AS Quarter, COUNT (*) Orders
 FROM sales_order
 GROUP BY ROLLUP (Year, Quarter)
 ORDER BY Year, Quarter
```

  このクエリでは、Year カラムを含むプレフィクスは、Year=2000 の計算ローと Year=2001 の計算ローを生成します。カラムのないプレフィクスに対する計算ローは 1 つです。このローは、中間結果セット内のすべてのローの小計です。

  小計ロー内の各カラムの値は、次のとおりです。

  - **プレフィクスに含まれるカラム** カラムの値。たとえば、上記のクエリでは、Year=2000 のローの小計の場合、Year カラムの値は 2000 です。

  - **プレフィクスから除外されるカラム** NULL。たとえば、Year カラムで構成されるプレフィクスによって生成される小計ローの場合、Quarter カラムの値は NULL です。

  - **集合関数** 除外されたカラムの値の総計。

    小計値は、集約されたローではなく、基本となるデータのローを使用して計算されます。たとえば SUM や COUNT など、多くの場合、どちらを使用しても結果は同じです。しかし、AVG、STDDEV、VARIANCE などの統計関数の場合、この区別は重要で、異なる結果となります。

NULL 値と小計ロー

GROUP BY 演算の入力ローに NULL が含まれる場合、ROLLUP、CUBE、または GROUPING SETS 演算の加算結果である小計ローと元の入力データの一部である NULL 値を含むローの区別がつきにくい可能性があります。

GROUPING 関数は、*group by list* 内のカラムを引数として受け取ることで、他のものから小計ローを区別します。この関数は、ローが小計ローであるためにカラムが NULL の場合は 1 を返し、それ以外の場合は 0 を返します。

例

次の例では、結果セットに GROUPING カラムが含まれています。入力データの結果として NULL を含むローが強調表示されていますが、これは小計ローだからではありません。GROUPING カラムが強調表示されています。このクエリは、**employee** テーブルと **sales_order** テーブルとの外部ジョインです。クエリでは、Texas、New York、または California に住む女性従業員を選択します。販売担当者ではない女性従業員 (したがって売り上げがない) のカラムには、NULL が表示されています。

```
SELECT employee.emp_id AS Employee,
        year(order_date) AS Year,
        COUNT(*) AS Orders,
        GROUPING ( Employee ) AS GE,
        GROUPING ( Year ) AS GY
 FROM employee LEFT OUTER JOIN sales_order
 ON employee.emp_id = sales_order.sales_rep
 WHERE employee.sex IN ('F')
 AND employee.state IN ('TX', 'CA', 'NY')
 GROUP BY ROLLUP (Year, Employee)
 ORDER BY Year, Employee
```

次のテーブルは、クエリからの結果セットを示します。

これらの従業員は販売担当ではないので、記録すべき売り上げ成績がない。

Employee	Year	Orders	GY	GE
(NULL)	(NULL)	5	1	0
(NULL)	(NULL)	169	1	1
102	**(NULL)**	**1**	**0**	**0**
390	**(NULL)**	**1**	**0**	**0**
1062	**(NULL)**	**1**	**0**	**0**
1090	**(NULL)**	**1**	**0**	**0**
1507	**(NULL)**	**1**	**0**	**0**
(NULL)	2000	98	1	0
667	2000	34	0	0
949	2000	31	0	0
1142	2000	33	0	0
(NULL)	2001	66	1	0
667	2001	20	0	0
949	2001	22	0	0
1142	2001	24	0	0

ROLLUP の使用

ROLLUP 演算は、集約ローの小計を計算します。ROLLUP は、GROUP BY 句を持つクエリの結果セットに小計ローを追加します。ROLLUP は、選択したカラム内にある値の階層の集約を示す結果セットを生成します。合計と小計を示す結果セットが必要な場合に ROLLUP 演算を使用します。

SQL 構文	次のセットを定義
`GROUP BY ROLLUP (A, B, C);`	(A, B, C) (A, B) (A) ( )

ROLLUP クエリから返されるローの詳細については、「OLAP 演算について」421 ページを参照してください。クエリでの GROUP BY 句の使用の詳細については、『ASA SQL リファレンス・マニュアル』>「GROUP BY 句」を参照してください。

例 1

次の例では、クエリが年および四半期の受注数を合計したものを返します。

```
SELECT year (order_date) AS Year, quarter (order_date)
AS Quarter, COUNT (*) AS Orders
 FROM sales_order
 GROUP BY ROLLUP (Year, Quarter)
 ORDER BY Year, Quarter
```

次のテーブルは、クエリからの結果セットを示します。小計ローは結果セット内で強調表示されています。各小計ローでは、小計が計算される 1 つまたは複数のカラムに NULL が入っています。

	Year	Quarter	Orders
1	**(NULL)**	**(NULL)**	**648**
2	**2000**	**(NULL)**	**380**
3	2000	1	87
	2000	2	77
	2000	3	91
	2000	4	235
2	**2001**	**(NULL)**	**268**
3	2001	1	139
	2001	2	119
	2001	3	10

[1] のローは、両年 (2000 年と 2001 年) の全四半期を通した受注合計数を示します。このローには、Year および Quarter カラムの両方に NULL があり、すべてのカラムがプレフィクスから除外されています。

注意
すべての ROLLUP 演算は、集約カラムを除く各カラムに NULL があるローを持つ結果セットを返します。このローは、集合関数に対する各カラムの合計を表します。たとえば、SUM が該当する集合関数であった場合、このローはすべての値の総合計を示します。

---

[2] の強調表示されたローは、それぞれ 2000 年と 2001 年の受注合計数を示します。いずれのローの Quarter カラムの値は Year の小計の計算用にロールアップされるため、このカラムには NULL が入っています。

注意
このような結果セットのロー数は、ROLLUP クエリに表示される変数の数によります。

---

[3] の残りのローでは、両年の各四半期の受注合計数を求めた集計情報が得られます。

例 2

これはもう 1 つの ROLLUP 演算の例で、最初の例よりもう少し複雑な結果セットを返します。この結果セットは、年、四半期、および地域別に販売受注数を合計したものです。この例では、第 1 および第 2 四半期と 2 つの選択された地域 (カナダおよび東部地区) のみを調べています。

```
SELECT year (order_date) AS Year, quarter (order_date)
AS Quarter, region, COUNT (*) AS Orders
 FROM sales_order
 WHERE region IN ('Canada', 'Eastern') AND quarter IN
('1', '2')
 GROUP BY ROLLUP (Year, Quarter, Region)
 ORDER BY Year, Quarter, Region
```

次のテーブルは、クエリからの結果セットを示します。各小計ローでは、小計が計算される 1 つまたは複数のカラムに NULL が入っています。

	Year	Quarter	Region	Orders
1	**(NULL)**	**(NULL)**	**(NULL)**	**183**
	2000	**(NULL)**	**(NULL)**	**68**
	2000	1	(NULL)	36
	2000	1	Canada	3
	2000	1	Eastern	33
2	2000	2	(NULL)	32
	2000	2	Canada	3
	2000	2	Eastern	29
	2001	**(NULL)**	**(NULL)**	**115**
	2001	1	(NULL)	57
	2001	1	Canada	11
	2001	1	Eastern	46
	2001	2	(NULL)	58
	2001	2	Canada	4
	2001	2	Eastern	54

最初のロー [1] は、すべてのローを集約したもので、Year、Quarter、Region カラムの両方に NULL が入っています。このローの Orders カラムの値は、2000 年と 2001 年の第 1 および第 2 四半期におけるカナダと東部地区の受注合計数を示しています。

[2] のローは、各年 (2000 年と 2001 年) の第 1 および第 2 四半期におけるカナダと東部地区の受注合計数を示します。これらのローの値 [2] は、ロー [1] で示している総合計と等しくなります。

[3] のローでは、その地域における指定した年の四半期の受注合計数に関するデータが得られます。

Year	Quarter	Region	Orders
(NULL)	(NULL)	(NULL)	183
2000	(NULL)	(NULL)	68
2000	**1**	**(NULL)**	**36**
2000	1	Canada	3
2000	1	Eastern	33
2000	**2**	**(NULL)**	**32**
2000	2	Canada	3
2000	2	Eastern	29
2001	(NULL)	(NULL)	115
2001	**1**	**(NULL)**	**57**
2001	1	Canada	11
2001	1	Eastern	46
2001	**2**	**(NULL)**	**58**
2001	2	Canada	4
2001	2	Eastern	54

3

[4] のローでは、結果セット内の各年、各四半期における各地域の受注合計数に関するデータが得られます。

Year	Quarter	Region	Orders
(NULL)	(NULL)	(NULL)	183
2000	(NULL)	(NULL)	68
2000	1	(NULL)	36
2000	1	Canada	3
2000	1	Eastern	33
2000	2	(NULL)	32
2000	2	Canada	3
2000	2	Eastern	29
2001	(NULL)	(NULL)	115
2001	1	(NULL)	57
2001	1	Canada	11
2001	1	Eastern	46
2001	2	(NULL)	58
2001	2	Canada	4
2001	2	Eastern	54

4

ROLLUP ローおよび小計ローの概要

ROLLUP は、GROUP BY クエリ・セットの UNION に相当します。次の 2 つのクエリの結果セットは同じです。GROUP BY A, B の結果セットは、A と B が定数として含まれるすべてのローの小計で構成されます。union を可能にするため、カラム C には NULL が割り当てられています。

ROLLUP クエリ	**ROLLUP** がない場合の同等のクエリ

`SELECT A, B, C,`
`  SUM( D )`
`FROM T1`
`GROUP BY ROLLUP`
`  (A, B, C);`
	`SELECT * FROM (`
	`  ( SELECT A, B, C, SUM( D )`
	`  GROUP BY A, B, C )`
	`  UNION ALL`
	`  ( SELECT A, B, NULL, SUM( D )`
	`  GROUP BY A, B )`
	`  UNION ALL`
	`  ( SELECT A, NULL, NULL, SUM( D )`
	`  GROUP BY A )`
	`  UNION ALL`
	`  ( SELECT NULL, NULL, NULL, SUM(`
	`D ) )`
	`  )`

CUBE の使用

CUBE 演算は OLAP の機能です。OLAP は、データの分析および管理方法に合うようにデータを編成するやり方で、必要な結果セットを生成する時間と手間を省くことができます。レポート用の合計値を事前に計算して、レポート計算を迅速に行えるようにします。CUBE は、レポートで大量のデータを処理したい場合に便利です。

ROLLUP と同様に、CUBE 演算子は結果セット内の集約値の小計を計算します。CUBE 演算が変数で実行されると、結果セットにはその変数値の組み合わせに基づいて多くの小計ローが含まれます。ROLLUP と異なり、CUBE はクエリで指定するあらゆる変数の組み合わせを返します。

CUBE 演算子は、データの次元に関する情報を追加した結果セットを返します。これにより、データのカラムをさらに分析することができます。これを次元と言います。CUBE は、可能なあらゆる次元の組み合わせについてのクロス作表レポートを提供し、選択したカラムのあらゆる値の組み合わせの集約を示した結果セットを生成します。

注意
CUBE は、次元が同一階層の一部でない場合にとても便利です。

SQL 構文	次のセットを定義
`GROUP BY CUBE (A, B, C);`	(A, B, C) (A, B) (A, C) (A) (B, C) (B) (C) ( )

CUBE クエリから返されるローの詳細については、「OLAP 演算について」421 ページを参照してください。クエリでの GROUP BY 句の使用の詳細については、『ASA SQL リファレンス・マニュアル』>「GROUP BY 句」を参照してください。

例

次の例では、クエリが受注合計数を計算した結果セットを返し、次に年および四半期ごとの受注数の小計を計算します。

注意
比較する変数の数が増えるにつれて、クエリが返すキューブがより複雑になります。

```
SELECT year (order_date) AS Year, quarter (order_date)
AS Quarter, COUNT (*) AS Orders
 FROM sales_order
 GROUP BY CUBE (Year, Quarter)
 ORDER BY Year, Quarter
```

次のテーブルは、クエリからの結果セットを示します。小計ローは結果セット内で強調表示されています。各小計ローでは、小計が計算される 1 つまたは複数のカラムに NULL が入っています。

	Year	Quarter	Orders
1	**(NULL)**	**(NULL)**	**648**
2	**(NULL)**	**1**	**226**
	(NULL)	**2**	**196**
	(NULL)	**3**	**101**
	(NULL)	**4**	**125**
3	**2000**	**(NULL)**	**380**
	2000	1	87
	2000	2	77
	2000	3	91
	2000	4	125
3	**2001**	**(NULL)**	**268**
	2001	1	139
	2001	2	119
	2001	3	10

[1] の強調表示されているローは、両年の全四半期を通した受注合計数を示します。Orders カラムの値は、[3] の各ローの値を合計したものです。また、[2] のローにある 4 つの値の合計でもあります。

注意
すべての CUBE 演算は、集約カラムを除く各カラムに NULL が含まれているローが少なくとも 1 つある結果セットを返します。このローは、集合関数に対する各カラムの合計を表します。

次の強調表示されているロー [2] は、両年の四半期ごとの受注合計数を示します。その下にある [3] のローは、最初が 2000 年の全四半期の受注合計数を示し、次に 2001 年の全四半期の受注合計数を示します。

CUBE の概要

CUBE 演算は、可能性のあるすべての変数の組み合わせを含む GROUPING SETS クエリと同等です。

CUBE クエリ	CUBE がない場合の同等のクエリ

`SELECT A, B, C, SUM( D )`
`FROM t`
`GROUP BY CUBE( A, B, C)`
	`SELECT A, B, C,`
	`SUM( D )`
	`FROM t`
	`GROUP BY GROUPING SETS(`
	`( A, B, C ),`
	`( A, B ), ( A ), ( B, C ),`
	`( B ), ( A, C ), ( C ),`
	`() )`

上記の SQL クエリで使用されている GROUPING SETS 構文の詳細については、「GROUPING SETS の使用」435 ページを参照してください。

GROUPING SETS の使用

GROUPING SETS により、複数の異なるグループ化カラムにあるグループを同一クエリで計算できます。CUBE と ROLLUP は事前に定義された小計を結果セットに追加しますが、GROUPING SETS は追加する小計を明示的に指定できます。ROLLUP または CUBE を使用するより複雑ですが、結果セットをよりうまく制御できます。この演算は、キューブの多次元結果セットの一部のみを返したい場合に有効です。GROUPING SETS を使用すると、比較する変数を選択でき、CUBE のようにすべての変数の比較を返したり、ROLLUP のように階層サブセットを返したりしません。

SQL 構文	次のセットを定義
`GROUP BY GROUPING SETS ((A, B), (A, C), (C));`	(A, B) (A, C) (C)

GROUPING SETS クエリから返されるローの詳細については、「OLAP 演算について」421 ページを参照してください。クエリでの GROUP BY 句の使用の詳細については、『ASA SQL リファレンス・マニュアル』>「GROUP BY 句」を参照してください。

例

次の例は、GROUPING SETS を使用したクエリから返された結果を調整する方法を示します。このクエリは結果セットが各年の小計ローを返すように指定していますが、各四半期を返すようには指定していません。

```
SELECT year (order_date) AS Year, quarter (order_date)
AS Quarter, COUNT (*) AS Orders
 FROM sales_order
 GROUP BY GROUPING SETS ((Year, Quarter), (Year))
 ORDER BY Year, Quarter
```

次のテーブルは、クエリからの結果セットを示します。小計ローが強調表示されています。

Year	Quarter	Orders
2000	**(NULL)**	**380**
2000	1	87
2000	2	77
2000	3	91
2000	4	125
2001	**(NULL)**	**268**
2001	1	139
2001	2	119
2001	3	10

結果セットには、各年 (2000 年と 2001 年) の小計ローのみが含まれています。両年の全四半期の受注合計数はないことに注意してください。

CUBE 演算子を使用してクエリを実行する場合は、各年 (2000 年、2001 年) の各四半期 (1、2、3、4) の受注全体に対する小計ローが計算されます。

GROUPING SETS の概要

ROLLUP と同様に、GROUPING SETS 演算は GROUP BY クエリの UNION ALL に相当します。

GROUPING SETS クエリ	GROUPING SETS がない場合の同等のクエリ

`SELECT A, B, C,`
`  SUM( D )`
`FROM t`
`GROUP BY GROUPING SETS`
`  ( A, B, C )`
	`SELECT * FROM (`
	`( SELECT A, NULL, NULL, SUM( D )`
	`FROM t GROUP BY A )`
	`UNION ALL`
	`( SELECT NULL, B, NULL, SUM( D )`
	`FROM t GROUP BY B )`
	`UNION ALL`
	`( SELECT NULL, NULL, C, SUM( D )`
	`FROM t GROUP BY C )`
	`)`

連結された GROUPING SETS の使用

もう 1 つの GROUPING SETS の使用方法は、連結された GROUPING SETS を使用することです。これにより、簡単に多数のグループ化の組み合わせを生成できます。連結された GROUPING SETS を使用すると、各 GROUPING SETS リストからクロス乗算が可能になります。CUBE と ROLLUP で指定されたグループ化の連結を実行できます。少数の連結グループ化であっても、クロス乗算演算により、多数の最終グループを生成できます。

SQL 構文	次のセットを定義
`GROUP BY GROUPING SETS (A, B)` `(Y, Z)`	(A, Y) (A, Z) (B, Y) (B, Z)

OLAP 関数の活用

OLAP 関数は、移動平均、ランク、累積値の計算など、データ上で分析タスクを実行する機能を提供します。SELECT 文の *select-list* または ORDER BY 句に OLAP 関数を含めることができます。OLAP 関数は、集合関数の引数としては使用できません。したがって、OLAP 関数を SUM(RANK()) というように指定することはできません。

OLAP 関数を使用する場合は、関数が適用されるローとその順序を定義する「**ウィンドウ**」が指定されます。現在のローに対して相対的なローのセットが、先行するローと後続のローの範囲または数として定義されます。たとえば、過去 3 か月間の平均を計算できます。

次のような OLAP 関数を使用できます。

- **RANK 関数** データ・セットからの値をランクの順でリストに集計します。たとえば、販売担当者を各四半期の総売り上げの少ない順にリストします。
- **レポート関数** 非集約値と集約値を比較します。たとえば、経費が平均以下の四半期をすべてリストします。
- **Window 関数** 移動ウィンドウを定義することでデータを分析します。たとえば、NASDAQ 値の 30 日間の移動平均を計算します。

RANK 関数

RANK 関数により、クエリで指定した順番に基づいて、結果セットにある各ローのランク値を計算できます。たとえば、販売マネージャが社内の最上位と最下位の販売担当や販売地域、最高と最低の販売製品を識別する必要があるかもしれません。RANK 関数はこのような情報を提供できます。

次の統計関数が RANK 関数として分類されます。

- **CUME_DIST 関数**　ORDER BY 句での定義にしたがって、クエリから返された 1 つのローについて、そのクエリから返されたもう一方のローに対する位置を計算します。詳細については、『ASA SQL リファレンス・マニュアル』> 「CUME_DIST 関数 [集合]」を参照してください。

- **RANK 関数**　ORDER BY 句での定義にしたがって、クエリから返された 1 つのローについて、そのクエリから返されたもう一方のローに対するランクを計算します。詳細については、『ASA SQL リファレンス・マニュアル』> 「RANK 関数 [集合]」を参照してください。

- **DENSE_RANK 関数**　グループ内の値のランクを計算します。RANK と DENSE_RANK との違いは、RANK では値の間に結合がある場合に、ランク順の差をそのままにします。詳細については、『ASA SQL リファレンス・マニュアル』> 「DENSE_RANK 関数 [集合]」を参照してください。

- **PERCENT_RANK 関数**　ランクの分数ではなくパーセントを計算します。この関数は、0 から 1 の 10 進数値を返します。詳細については、『ASA SQL リファレンス・マニュアル』> 「PERCENT_RANK 関数 [集合]」を参照してください。

- **ROW_NUMBER 関数**　ORDER BY 句で特定された各ローに固有の番号を割り当てます。詳細については、『ASA SQL リファレンス・マニュアル』> 「ROW_NUMBER 関数 [集合]」を参照してください。

例 1

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 emp_lname, salary, sex,
       RANK () OVER (ORDER BY salary DESC) "Rank"
       FROM employee
 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。

emp_lname	salary	sex	Rank
Shishov	72995.00	F	1
Wang	68400.00	M	2
Cobb	62000.00	M	3
Morris	61300.00	M	4
Diaz	54900.00	M	5
Driscoll	48023.00	M	6
Hildebrand	45829.00	F	7
Goggin	37900.00	M	8
Rebeiro	34576.00	M	9
Bigelow	31200.00	F	10
Lynch	24903.00	M	11

例 2

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 1 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 emp_lname, salary, sex,
      RANK () OVER (PARTITION BY sex
      ORDER BY salary DESC) "Rank"
      FROM employee
 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。

従業員は、男女別に給与順に降順でランキングされている。

emp_lname	salary	sex	Rank
Wang	68400.00	M	1
Cobb	62000.00	M	2
Morris	61300.00	M	3
Diaz	54900.00	M	4
Driscoll	48023.00	M	5
Goggin	37900.00	M	6
Rebeiro	34576.00	M	7
Lynch	24903.00	M	8
Shishov	72995.00	F	1
Hildebrand	45829.00	F	2
Bigelow	31200.00	F	3

例 3

次の例は、ユタ州とアリゾナ州の女性従業員のリストを検索して、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 emp_lname, salary,
      PERCENT_RANK () OVER (ORDER BY salary DESC) "Rank"
       FROM employee
 WHERE state IN ('UT','AZ') AND sex IN ('F')
```

次のテーブルは、クエリからの結果セットを示します。

emp_lname	salary	Rank
Shishov	72995.00	0
Jordan	51432.00	0.25
Hildebrand	45829.00	0.5
Bigelow	31200.00	0.75
Bertrand	29800.00	1

例 4

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 emp_lname, salary,
        PERCENT_RANK () OVER (ORDER BY salary DESC)
"Rank"
        FROM employee
        WHERE sex IN ('M')  )
        AS DT ( last_name, salary, percent )
 WHERE percent < 0.05
```

次のテーブルは、クエリからの結果セットを示します。

last_name	salary	percent
Scott	96300.00	0
Sheffield	87900.00	0.025
Lull	87900.00	0.025

レポート関数

レポート関数により、非集約値と集約値を比較できます。たとえば、販売担当が指定した年についてある製品を平均数以上注文した全顧客のリストを作成する必要がある場合や、マネージャが部門の平均給与とある従業員の給与とを比較したい場合などです。

レポート関数と組み合わせて、集合関数を使用できます。使用可能な集合関数のリストについては、『ASA SQL リファレンス・マニュアル』> 「集合関数」を参照してください。

注意
LIST 関数を除いて、Window 関数と同じ統計関数を使用できます。

例 1

次のクエリは、平均販売数より多く販売された製品のリストを示す結果セットを返します。結果セットは、年ごとに分割されています。

```
SELECT *
 FROM (SELECT year(order_date) AS Year, prod_id,
         SUM( quantity ) AS Q,
         AVG (SUM(quantity))
         OVER (PARTITION BY Year) AS Average
         FROM sales_order JOIN sales_order_items
         GROUP BY year(order_date), prod_id
         ORDER BY Year)
     AS derived_table
 WHERE Q > Average
```

次のテーブルは、クエリからの結果セットを示します。

Year	prod_id	Q	Average
2000	700	2700	1787.000
2000	601	1932	1787.000
2000	600	2124	1787.000
2000	400	2030	1787.000
2001	700	1836	1048.000
2001	401	1057	1048.000
2001	400	1248	1048.000

2000 年は、平均注文数が 1,787 で、4 製品 (700、601、600、400) が平均以上の販売量でした。2001 年は、平均注文数が 1,048 で、3 製品が平均を超えました。

例 2

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
         emp_lname   AS Employee,
         dept_id     AS Dept,
         CAST( salary as DECIMAL( 10, 2) ) AS Salary,
         CAST( AVG( salary )
             OVER (PARTITION BY dept_id) AS DECIMAL
(10,2) ) AS Average,
         CAST (STDDEV_POP (salary)
            OVER (PARTITION BY dept_id) AS DECIMAL (10,
2 ) ) AS Std_Dev
         FROM employee
         GROUP BY Dept, Employee, Salary)
     AS derived_table
 WHERE Salary > Average + Std_Dev
 ORDER BY Dept, 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	Dept	Salary	Average	Std_Dev
Lull	100	87900.00	58736.28	16829.60
Sheffield	100	87900.00	58736.28	16829.60
Scott	100	96300.00	58736.28	16829.60
Sterling	200	64900.00	48390.95	13869.60
Savarino	200	72300.00	48390.95	13869.60
Kelly	200	87500.00	48390.95	13869.60
Shea	300	138948.00	59500.00	30752.40
Blaikie	400	54900.00	43640.67	11194.02
Morris	400	61300.00	43640.67	11194.02
Evans	400	68940.00	43640.67	11194.02
Martinez	500	55500.80	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 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。

Window 関数

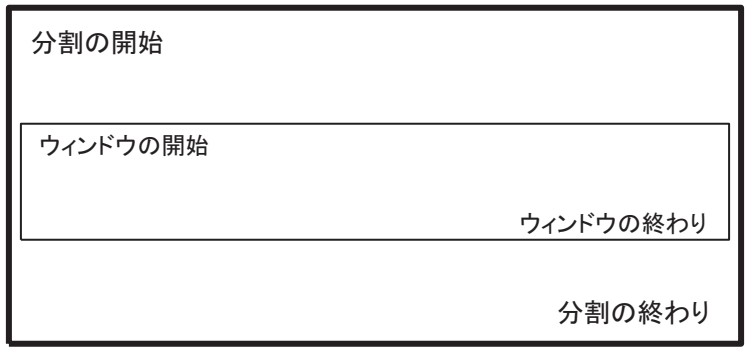
Window 関数を使用すると、各ローを囲むウィンドウの集約値を計算してデータを分析できます。結果セットは、ローのセットを表す合計値を返します。Window 関数を使用して、指定した期間における会社の販売数の移動平均を計算できます。

Window 関数と組み合わせて、(LIST 以外の) 集合関数を使用できます。

使用可能な集合関数のリストについては、『ASA SQL リファレンス・マニュアル』> 「集合関数」を参照してください。

注意
レポート関数と同じ統計関数を使用できます。

次の図は、含まれる分割とウィンドウ表示を示しています。ウィンドウは、任意のロー数で定義され、分割内を降順で並べます。



例

次の例は、Window 関数を示したものです。クエリは、部門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 dept_id, emp_lname, start_date, salary,
 SUM(salary) OVER (PARTITION BY dept_id
 ORDER BY start_date
 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
"Sum_Salary"
 FROM employee
 WHERE state IN ('CA', 'UT', 'NY', 'AZ') AND dept_id IN
('100', '200')
 ORDER BY dept_id, start_date;
```

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、部門ごとに分割されています。

dept_id	emp_lname	start_date	salary	Sum_Salary
100	Whitney	1984-08-28	45700.00	45700.00
100	Cobb	1985-01-01	62000.00	107700.00
100	Shishov	1986-06-07	72995.00	180695.00
100	Driscoll	1986-07-01	48023.69	228718.69
100	Guevara	1986-10-14	42998.00	271716.69
100	Wang	1988-09-29	68400.00	340116.69
100	Soo	1990-07-31	39075.00	379191.69
100	Diaz	1990-08-19	54900.00	434091.69
200	Overbey	1987-02-19	39300.00	39300.00
200	Martel	1989-10-16	55700.00	95000.00
200	Savarino	1989-11-07	72300.00	167300.00
200	Clark	1990-07-21	45000.00	212300.00
200	Goggin	1990-08-05	37900.00	250200.00

部門 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。

第 11 章

サブクエリの使用

この章の内容

クエリを作成するときには、WHERE 句と HAVING 句を使用してクエリが返すローを制限します。

場合によって、選択するローが複数のテーブルに格納されている情報に依存していることがあります。WHERE 句または HAVING 句にサブクエリを使用することによって、別のテーブルから取得された指定に従って、1 つのテーブルからローを選択できます。これを行うための別の方法については、「ジョイン：複数テーブルからのデータ検索」335 ページを参照してください。

始める前に

このサブクエリについての章は、クエリと SELECT 文の構文についての知識があることを前提にしています。クエリについては、「クエリ：テーブルからのデータの選択」275 ページを参照してください。

サブクエリの概要

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、あるテーブルに製品だけの情報を、別のテーブルには注文関連の情報を保存します。製品テーブルにはさまざまな製品の情報が入っています。受注品目テーブルには、顧客の注文についての情報が入っています。

一般的に、テーブルを 1 つだけ使用して回答できるのは、最も簡単な質問だけです。たとえば、在庫数が 50 未満になったときに製品を再注文する場合、次のクエリで「在庫数が少ない製品は何か」という問い合わせに対する回答を得ることができます。

```
SELECT id, name, description, quantity
 FROM product
 WHERE quantity < 50
```

ただし、「在庫切れ寸前の状態」が、一般顧客が注文するタイプごとに品目量が異なる場合は、数字 "50" を sales_order_items テーブルから取得する値に置き換える必要があります。

サブクエリの構造

サブクエリは通常のクエリのように構成され、メイン・クエリの SELECT 句、FROM 句、WHERE 句、または HAVING 句の中に置かれます。たとえば、前述の例では、サブクエリを使用してある顧客が注文する品目の平均数を選択し、その数をメイン・クエリに使用して在庫数が少ない製品を検索します。次に示すクエリは、顧客が注文した各タイプの平均品目数の 2 倍未満の製品名とその説明を検索します。

```
SELECT name, description
 FROM product WHERE quantity <  2 * (
     SELECT avg(quantity)
     FROM sales_order_items
     )
```

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。

WHERE 句でのサブクエリの使用

WHERE 句内のサブクエリは、ロー選択のプロセスの一部として機能します。ローの選択に使用する基準が別のテーブルの結果に依存するときに、WHERE 句内にサブクエリを使用します。

例

在庫数が平均注文数の 2 倍よりも少ない製品を検索します。

```
SELECT name, description
 FROM product WHERE quantity <  2 * (
     SELECT avg(quantity)
     FROM sales_order_items)
```

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、sales_order_items 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 sales_order_items
```

これによって sales_order_items 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 product
 WHERE quantity < 2*25.851413
```

サブクエリを使用すると、この 2 つの手順を 1 つのオペレーションにまとめることができます。

WHERE 句でのサブクエリの目的

WHERE 句内でのサブクエリは、探索条件の一部です。WHERE 句内で使用できる簡単な探索条件については、「クエリ：テーブルからのデータの選択」275 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 句内のほかの式と同様に、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 product
 GROUP BY name
 HAVING avg(quantity) > 2* (
     SELECT avg(quantity)
     FROM sales_order_items
     )
```

name	avg(product.quantity)
Tee Shirt	52.333333
Baseball Cap	62
Shorts	80

クエリは次のように実行します。

- サブクエリは sales_order_items 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 product テーブルを調べて、平均数の製品を計算し、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

「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 番号を検索する」

```
SELECT prod_id, line_id
 FROM sales_order_items
 GROUP BY prod_id, line_id
 HAVING 2* avg(quantity) > (
     SELECT quantity
     FROM product
     WHERE product.id = sales_order_items.prod_id)
```

prod_id	line_id
401	2
401	1
401	4
501	3
…	…

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 sales_order_items.prod_id 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。product 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 product テーブルのレコードは 1 つだけになります。

サブクエリのテスト

HAVING 句内で使用できる簡単な探索条件については、「クエリ：テーブルからのデータの選択」275 ページの章で説明します。サブクエリは WHERE 句または HAVING 句に置かれる式なので、サブクエリの探索条件も見なれたものになります。

次の探索条件があります。

- **サブクエリ比較テスト**　メイン・クエリのテーブルにある各レコードについて、サブクエリが生成した 1 つの値と式の値を比較します。
- **限定比較テスト**　サブクエリが生成した値のそれぞれのセットと式の値を比較します。
- **サブクエリ・セット・メンバシップ・テスト**　サブクエリが生成した値のセットのいずれかと、式の値が一致するかどうかを調べます。
- **存在テスト**　サブクエリがローを生成するかどうかを調べます。

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 product
 WHERE quantity <  2 * (
     SELECT avg(quantity)
     FROM sales_order_items)
```

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、sales_order_items テーブルから取り出します。

```
SELECT avg(quantity)
 FROM sales_order_items
```

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、sales_order_items 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 product
 WHERE quantity <  2 * (
     SELECT avg(quantity), max (quantity)
     FROM sales_order_items)
```

このクエリは、「select リストの中にカラムが 2 つ以上指定されています。」というエラーを返します。

ANY と ALL を使用した限定比較テスト

限定比較テストは、ALL テストと ANY テストの 2 つのカテゴリに分類できます。

ANY テスト

ANY テストは、SQL 比較演算子 (=、<>、<、<=、>、>=) のいずれかと組み合せて使用して、1 つの値をサブクエリが生成するデータ値のカラムと比較します。テストを実行するには、SQL は指定された比較演算子を使用して、テスト値をカラムのデータ値のそれぞれと比較します。*いずれか*の比較の結果が TRUE になる場合、ANY テストは TRUE を返します。

ANY を使用するサブクエリは 1 つのカラムを返します。

例

注文番号 2005 の最初の製品が出荷された後に受けた注文の注文 ID と顧客 ID を検索します。

```
SELECT id, cust_id
 FROM sales_order
 WHERE order_date > ANY (
     SELECT ship_date
     FROM sales_order_items
     WHERE id=2005)
```

id	cust_id
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、sales_order 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, cust_id
 FROM sales_order
 WHERE order_date > SOME (
     SELECT ship_date
     FROM sales_order_items
     WHERE id=2005)
```

キーワード SOME はキーワード ANY と同等です。

ANY 演算子についての注意

ANY テストには、このほかに 2 つの重要な特徴があります。

- **空のサブクエリの結果セット**　サブクエリが空の結果セットを生成する場合、ANY テストは FALSE を返します。結果がない場合、少なくとも 1 つの結果が比較テストを満たしているというのは真ではないので、これは理にかなっています。
- **サブクエリの結果セットの NULL 値**　サブクエリの結果セットには少なくとも 1 つの NULL 値があることが前提です。結果セットの NULL 以外のすべてのデータ値に対して比較テストが false の場合、ANY は NULL を返します。これは、比較テストが

保持するサブクエリの値があるかどうか、この状況では確定できないためです。値があるかどうかは、結果セットの NULL データの「正確な」値によって異なります。

ALL テスト

ANY テストと同様、ALL テストは、6 つの SQL 比較演算子 (=、<>、<、<=、>、>=) のいずれかと組み合せて使用して、1 つの値をサブクエリが生成するデータ値と比較します。テストを実行するには、SQL は指定された比較演算子を使用して、テスト値を結果セットのデータ値のそれぞれと比較します。すべての比較の結果が TRUE になる場合、ALL テストは TRUE を返します。

例

「注文番号 2001 のすべての製品が出荷された後に受けた注文の注文 ID と顧客 ID を検索する」という要求は、当然 ALL テストで処理されます。

```
SELECT id, cust_id
 FROM sales_order
 WHERE order_date > ALL (
     SELECT ship_date
     FROM sales_order_items
     WHERE id=2001)
```

id	cust_id
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、sales_order テーブルの注文 ID と顧客 ID が結果セットに示されます。このように ALL テストは AND 演算子に似ています。前述のクエリは、「この注

文は注文番号 2001 の最初の製品が出荷される前に受けたものか、注文番号 2001 の 2 番目の製品が出荷される前に受けたものか、さらに…」というように解釈できます。

ALL 演算子についての注意

ALL テストには、このほかに 3 つの重要な特徴があります。

- **空のサブクエリの結果セット**　サブクエリが空の結果セットを生成した場合、ALL テストは TRUE を返します。結果がない場合、比較テストが結果セットのどの値に対しても適用しているというのは真なので、これは理にかなっています。

- **サブクエリの結果セットの NULL 値**　結果セットのいずれかの値に対する比較テストが FALSE の場合、ALL は FALSE を返します。すべての値が TRUE の場合は TRUE を返します。それ以外の場合は、UNKNOWN を返します。たとえば、サブクエリの結果セットに NULL 値があっても、NULL 以外のすべての値の探索条件が TRUE の場合などです。

- **ALL テストの否定**　次の 2 つの式は同じでは*ありません*。

```
NOT a = ALL (subquery)
 a <> ALL (subquery)
```

このテストの詳細については、「限定比較テスト」472 ページを参照してください。

IN 条件によるセット・メンバシップのテスト

サブクエリ・セット・メンバシップ・テストを使用して、メイン・クエリからの値をサブクエリの複数の値と比較できます。

サブクエリ・セット・メンバシップ・テストは、メイン・クエリの各ローの 1 つのデータ値を、サブクエリが生成したデータ値の 1 つのカラムと比較します。メイン・クエリのデータ値がカラムのデータ値の*いずれか*と一致する場合、サブクエリは TRUE を返します。

例

Shipping 部または Finance 部の部長である従業員の名前を選択します。

```
SELECT emp_fname, emp_lname
 FROM employee
 WHERE emp_id IN (
     SELECT dept_head_id
     FROM department
     WHERE (dept_name='Finance' or dept_name =
'Shipping'))
```

emp_fname	emp_lname
Jose	Martinez
Mary Anne	Shea

この例のサブクエリは次のようになっています。

```
SELECT dept_head_id
 FROM department
 WHERE (dept_name='Finance' OR dept_name = 'Shipping')
```

これは、Shipping 部と Finance 部の部長に対応する ID 番号を、department 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。

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 emp_fname, emp_lname
 FROM employee
 WHERE emp_id =ANY (
     SELECT dept_head_id
     FROM department
     WHERE (dept_name='Finance' or dept_name =
'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 emp_fname, emp_lname
 FROM employee
 WHERE emp_id NOT IN (
     SELECT dept_head_id
     FROM department
     WHERE (dept_name='Finance' OR dept_name =
'Shipping'))
```

存在テスト

サブクエリ比較テストとセット・メンバシップ・テストに使用されるサブクエリは、いずれもサブクエリ・テーブルからデータ値を返します。しかし、場合によっては、*どの*結果をサブクエリが返すのかではなく、サブクエリが*何らかの*結果を返すのかどうかが重要であることがあります。存在テスト (EXISTS) は、サブクエリがクエリ結果のローを生成するかどうかを調べます。サブクエリが 1 つ以上の結果のローを返す場合、EXISTS テストは TRUE を返します。結果のローを返さない場合は、FALSE を返します。

例

ここでは、「2001 年 7 月 13 日以降に発注したのはどの顧客か」という要求を、サブクエリを使って表現してみます。

```
SELECT fname, lname
 FROM customer
 WHERE EXISTS (
     SELECT *
     FROM sales_order
     WHERE (order_date > '2001-07-13') AND
      (customer.id = sales_order.cust_id))
```

fname	lname
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、customer 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 customer 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」469 ページを参照してください。

```
SELECT *
 FROM sales_order
 WHERE (order_date > '2001-07-13') AND (customer.id =
sales_order.cust_id)
 SELECT ship_date
 FROM sales_order
 WHERE (order_date > '2001-07-13') AND (customer.id =
sales_order.cust_id)
```

便宜上、"SELECT *" という表記を使用していますが、sales_order テーブルのどのカラムが SELECT 文に指定されるかどうかは問題ではありません。

存在テストの否定

EXISTS テストの論理は、NOT EXISTS というフォームで否定できます。この場合、テストはサブクエリがローを返さない場合に TRUE を、ローを返す場合に FALSE を返します。

相関サブクエリ

サブクエリには customer 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 customer 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。sales_order テーブルの注文日が 2001 年 7 月 13 日より後で、customer テーブルと sales_order テーブルの顧客 ID が一致していれば、customer 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。

外部参照

通常、サブクエリの本体では、メイン・クエリのアクティブなローにあるカラムの値を参照する必要があります。次のクエリを考えてみます。

```
SELECT name, description
 FROM product
 WHERE quantity < 2 * (
     SELECT avg(quantity)
     FROM sales_order_items
     WHERE product.id = sales_order_items.prod_id)
```

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 sales_order_items 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 product.id カラムは、サブクエリではなく、メイン・クエリの FROM 句に指定されているテーブルのカラムを参照します。SQL は product テーブルの各ローの間を移動して、サブクエリの WHERE 句を評価するときに、現在のローの id 値を使用します。

外部参照の説明

このサブクエリの product.id カラムは、外部参照の例です。外部参照を使用するサブクエリを相関サブクエリと言います。外部参照は、サブクエリの FROM 句内のどのテーブルのどのカラムも参照しないカラム名です。代わりに、このカラム名は、メイン・クエリの FROM 句に指定されるテーブルのカラムを参照します。前述の例が示すように、外部参照のカラムの値は、メイン・クエリによって現在テストされているローから抽出されます。

サブクエリとジョイン

サブクエリのオプティマイザは、サブクエリを利用するクエリの多くをジョインとして自動的に書き換えます。

例

「Mrs. Clarke と Suresh がいつ注文し、どの担当者が注文を受けたか」という要求を考えてみます。これは次のクエリを使用して回答できます。

```
SELECT order_date, sales_rep
 FROM sales_order
 WHERE cust_id IN (
     SELECT id
     FROM customer
     WHERE lname = 'Clarke' OR fname = 'Suresh')
```

order_date	sales_rep
2001-01-05	1596
2000-01-27	667
2000-11-11	467
2001-02-04	195
…	…

サブクエリは WHERE 句に名前がリストされている 2 人の顧客に対応する顧客 ID のリストを生成します。メイン・クエリはこの 2 人の注文に対応する注文日と担当者を検索します。

ジョインによるサブクエリの置き換え

同じ問い合わせをジョインを使用して応答できます。このクエリの、2 つのテーブルのジョインを使用した代替フォームを次に示します。

```
SELECT order_date, sales_rep
 FROM sales_order, customer
 WHERE cust_id=customer.id AND
   (lname = 'Clarke' OR fname = 'Suresh')
```

このフォームのクエリは sales_order テーブルを customer テーブルにジョインして各顧客の注文を検索し、Suresh と Clarke のレコードだけを返します。

サブクエリとして作成できないジョイン

これらのクエリはどちらも正確な注文日と担当者を検索します。どちらかの方がより正確ということはありません。通常は、サブクエリのフォームの方が自然に感じられます。顧客 ID についての情報は要求されておらず、この問い合わせに応答するために sales_order テーブルと customer テーブルをジョインするのは不自然に感じられる場合があるためです。

しかし、customer テーブルからの情報を含むように要求が変更されると、サブクエリのフォームは機能しなくなります。たとえば、「Mrs Clarke と Suresh がいつ注文し、どの担当者が注文を受け、この 2 人の顧客の氏名は何か」という要求の場合は、メインの WHERE 句に customer テーブルを含む必要があります。

```
SELECT fname, lname, order_date, sales_rep
 FROM sales_order, customer
 WHERE cust_id=customer.id AND (lname = 'Clarke' OR
fname = 'Suresh')
```

fname	lname	order_date	sales_rep
Belinda	Clarke	1/5/01	1596
Belinda	Clarke	1/27/00	667
Belinda	Clarke	11/11/00	467
Belinda	Clarke	2/4/01	195
…	…	…	…

ジョインとして作成できないサブクエリ

同様に、サブクエリは機能するが、ジョインは機能しないという場合があります。次に例を示します。

```
SELECT name, description, quantity
 FROM product
 WHERE quantity <  2 * (
     SELECT avg(quantity)
     FROM sales_order_items)
```

name	description	quantity
Tee Shirt	Tank Top	28

name	description	quantity
Baseball Cap	Wool cap	12
Visor	Cloth Visor	36
…	…	…

この場合、内部クエリは集計クエリで外部クエリは集計クエリではないので、2 つのクエリを簡単なジョインで組み合わせることはできません。

ジョインの詳細については、「ジョイン：複数テーブルからのデータ検索」335 ページを参照してください。

ネストされたサブクエリ

これまで見てきたように、サブクエリは、常にクエリの WHERE 句か HAVING 句内にあります。サブクエリ自体に WHERE 句か HAVING 句、またはその両方が含まれることがあり、その結果サブクエリが別のサブクエリ内にある場合があります。別のサブクエリ内のサブクエリは、「**ネストされたサブクエリ**」と呼ばれます。

例

Fees 部の任意の品目が注文された日に出荷された注文の注文 ID とライン ID をリストします。

```
SELECT id, line_id
 FROM sales_order_items
 WHERE ship_date = ANY (
     SELECT order_date
     FROM sales_order
     WHERE fin_code_id IN (
         SELECT code
         FROM fin_code
         WHERE (description = 'Fees')))
```

id	line_id
2001	1
2001	2
2001	3
2002	1
…	…

ネストされたサブクエリの説明

- この例では、最も内側のサブクエリが、"Fees" という説明がある財務コードのカラムを生成します。

```
SELECT code
 FROM fin_code
 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 order_date
 FROM sales_order
 WHERE fin_code_id IN (subquery)
```

- 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, line_id
 FROM sales_order_items
 WHERE ship_date = ANY (subquery)
```

ネストされたサブクエリのレベルは、3 つを超えることもできます。レベルの最大数はありませんが、3 つ以上のレベルのクエリは、それ以下のレベルのクエリに比べて、実行にかなりの時間がかかります。

サブクエリ操作

クエリにサブクエリが含まれていると、どのクエリが有効でどのクエリが有効でないかを理解するのは難しい場合があります。同様に、マルチレベルのクエリが何を実行するかを理解するのも非常に複雑ですが、Adaptive Server Anywhere でサブクエリがどのように処理されるかを理解するうえで役に立ちます。クエリ処理の概要については、「クエリ結果の要約、グループ化、ソート」305 ページを参照してください。

相関サブクエリ

単純なクエリでは、データベース・サーバはクエリのローごとに一度、WHERE 句を評価して処理します。しかし場合によっては、サブクエリが 1 つの結果しか返さず、データベース・サーバが結果セット全体に対して 2 回以上評価する必要がないことがあります。

非相関サブクエリ

次のクエリを考えてみます。

```
SELECT name, description
 FROM product
 WHERE quantity <  2 * (
     SELECT avg(quantity)
     FROM sales_order_items)
```

この例では、サブクエリは 1 つの値、sales_order_items テーブルからの平均数を計算します。クエリを評価するときに、データベース・サーバはこの値を一度計算し、その値を product テーブルの quantity フィールドにあるそれぞれの値と比較して、対応するローを選択するかどうかを決定します。

相関サブクエリ

サブクエリに外部参照が含まれていると、このような方法は使用できません。たとえば、次のクエリのサブクエリを考えてみます。

```
SELECT name, description
 FROM product
 WHERE quantity < 2 * (
     SELECT avg(quantity)
     FROM sales_order_items
     WHERE product.id=sales_order_items.prod_id)
```

これは product テーブル内のアクティブなローに依存する値を返します。このようなサブクエリを相関サブクエリと呼びます。この場合、サブクエリは外部クエリのローごとに異なる値を返すことがあり、それによってデータベース・サーバが複数の評価を実行することが必要な場合があります。

WHERE 句のサブクエリのジョインへの変換

Adaptive Server Anywhere のクエリ・オプティマイザは、一部のマルチレベルのクエリをジョインを使用するように変換します。変換はユーザによるアクションを必要とすることなく実行されます。この項では、データベースでのクエリのパフォーマンスを理解できるように、どのサブクエリがジョインに変換できるのかを説明します。

例

「Mrs. Clarke と Suresh がいつ注文し、どの担当者が注文を受けたか」という問い合わせは、2 つのレベルのクエリとして作成できます。

```
SELECT order_date, sales_rep
 FROM sales_order
 WHERE cust_id IN (
     SELECT id
     FROM customer
     WHERE lname = 'Clarke' OR fname = 'Suresh')
```

代替の、同等の方法では、ジョインを使用してクエリを作成します。

```
SELECT fname, lname, order_date, sales_rep
 FROM sales_order, customer
 WHERE cust_id=customer.id AND
   (lname = 'Clarke' OR fname = 'Suresh')
```

マルチレベルのクエリをジョインで作成するために満たす必要がある基準は、演算子のタイプによって異なります。サブクエリが WHERE 句内にある場合は、次のフォームになることに注意してください。

SELECT *select-list*
FROM *table*
WHERE
[**NOT**] *expression comparison-operator* **(** *subquery* **)**
| [**NOT**] *expression comparison-operator* { **ANY | SOME** } **(** *subquery* **)**
| [**NOT**] *expression comparison-operator* **ALL (** *subquery* **)**
| [**NOT**] *expression* **IN (** *subquery* **)**

| **[NOT] EXISTS (** *subquery* **)**
GROUP BY *group-by-expression*
HAVING *search-condition*

サブクエリをジョインに変換できるかどうかは、演算子のタイプ、クエリの構造、サブクエリの構造など、さまざまな要素によって異なります。

比較演算子

比較演算子 (=、<>、<、<=、>、>=) に続くサブクエリは、ジョインに変換する場合、一定の条件を満たさなければなりません。比較演算子に続くサブクエリは、メイン・クエリのローごとに値を 1 つずつ返す場合は有効です。この基準に加えて、サブクエリが次の場合にはジョインに変換できます。

- GROUP BY 句を含んでいない
- キーワード DISTINCT を含んでいない
- UNION クエリではない
- 集計クエリではない

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 order_date, sales_rep
 FROM sales_order
 WHERE cust_id = (
     SELECT id
     FROM customer
     WHERE fname = '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 order_date, sales_rep
 FROM sales_order, customer
 WHERE cust_id=customer.id AND (lname = 'Clarke' OR
fname = '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 product
 WHERE quantity <  2 * (
     SELECT avg(quantity)
     FROM sales_order_items)
```

限定比較テスト

キーワード ALL、ANY、SOME のいずれかに続くサブクエリは、一定の条件を満たす場合にだけ、ジョインに変換できます。

- メイン・クエリが GROUP BY 句を含んでおらず、集計クエリでない。または、サブクエリが 1 つの値を返す。
- サブクエリが GROUP BY 句を含んでいない。
- サブクエリがキーワード DISTINCT を含んでいない。
- サブクエリが UNION クエリではない。
- サブクエリが集計クエリではない。
- 'expression comparison-operator ANY/SOME (subquery)' の部分が否定される。
- 'expression comparison-operator ALL (subquery)' の部分が否定される。

最初の 4 つの条件は、比較的簡単です。

例

「Mrs. Clarke と Suresh がいつ注文し、どの担当者が注文を受けたか」という要求は、サブクエリのフォームで処理できます。

```
SELECT order_date, sales_rep
 FROM sales_order
 WHERE cust_id = ANY (
     SELECT id
     FROM customer
     WHERE lname = 'Clarke' OR fname = 'Suresh')
```

または、ジョインのフォームで表現できます。

```
SELECT fname, lname, order_date, sales_rep
 FROM sales_order, customer
 WHERE cust_id=customer.id AND (lname = 'Clarke' OR
fname = 'Suresh')
```

しかし、「Mrs. Clarke、Suresh、および顧客でもある従業員が、いつ注文したか」という要求は union クエリとして表現されるので、ジョインには変換できません。

```
SELECT order_date, sales_rep
 FROM sales_order
 WHERE cust_id = ANY (
     SELECT id
     FROM customer
     WHERE lname = 'Clarke' OR fname = 'Suresh'
     UNION
     SELECT emp_id
     FROM employee)
```

同様に、「注文番号 2001 のすべての製品が出荷された後に受け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、当然サブクエリで表現されます。

```
SELECT id, cust_id
 FROM sales_order
 WHERE NOT order_date > ALL (
     SELECT ship_date
     FROM sales_order_items
     WHERE id=2001)
```

これは次のジョインに変換されます。

```
SELECT sales_order.id, cust_id
 FROM sales_order, sales_order_items
 WHERE (sales_order_items.id=2001) and (order_date <=
ship_date)
```

しかし、「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ます。

```
SELECT id, cust_id
 FROM sales_order
 WHERE NOT order_date > ALL (
     SELECT first (ship_date)
     FROM sales_order_items )
```

このため、これはジョインに変換されません。

ANY と ALL 演算子を使用するサブクエリの否定

5 つ目の条件はやや複雑です。次のフォームのクエリを考えてみます。

SELECT *select-list*
FROM *table*
WHERE NOT *expression comparison-operator* **ALL(** *subquery* **)**
このクエリは、ジョインに変換されます。次のフォームのクエリも同様です。

SELECT *select-list*
FROM *table*
WHERE *expression comparison-operator* **ANY(** *subquery* **)**

次の 2 つを見てみます。

SELECT *select-list*
FROM *table*
WHERE *expression comparison-operator* **ALL(** *subquery* **)**

および

SELECT *select-list*
FROM *table*
WHERE NOT *expression comparison-operator* **ANY(** *subquery* **)**

この 2 つは変換できません。

ANY と ALL に論理的に同等な式

これは、最初の 2 つのクエリは、後の 2 つが同等であるのと同様に、同等であるためです。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。次の式を見てみます。

```
NOT ((X > A) AND (X > B))
```

これは次の式と同等です。

```
(X <= A) OR (X <= B)
```

同様に、次の式を見てみます。

```
NOT order_date > ALL (
     SELECT first (ship_date)
     FROM sales_order_items )
```

これは次の式と同等です。

```
order_date <= ANY (
     SELECT first (ship_date)
     FROM sales_order_items )
```

ANY と ALL の否定

次の式を見てみます。

NOT *column-name operator* **ANY(** *subquery* **)**

これは次の式と同等です。

column-name inverse-operator **ALL(** *subquery* **)**

同様に、次の式を見てみます。

NOT *column-name operator* **ALL(** *subquery* **)**

これは次の式と同等です。

column-name inverse-operator **ANY(** *subquery* **)**

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。

演算子とその否定の表

次の表に、それぞれの演算子とその逆をリストします。

Operator	inverse-operator
=	<>
<	=>

Operator	inverse-operator
>	=<
=<	>
=>	<
<>	=

セット・メンバシップ・テスト

キーワード IN に続くサブクエリを含むクエリは、次の条件を満たす場合にだけジョインに変換されます。

- 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)' 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。

```
SELECT emp_fname, emp_lname
 FROM employee
 WHERE emp_id IN (
     SELECT dept_head_id
     FROM department
     WHERE (dept_name='Finance' or dept_name =
'Shipping'))
```

これは条件を満たすので、ジョインを使用するクエリに変換されます。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 emp_fname, emp_lname
 FROM employee
 WHERE emp_id IN (
     SELECT dept_head_id
     FROM department
     WHERE (dept_name='Finance' or dept_name =
'Shipping')
     UNION
     SELECT cust_id
     FROM sales_order)
```

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、否定のサブクエリで表現されます。

```
SELECT emp_fname, emp_lname
 FROM employee
   WHERE NOT emp_id IN (
     SELECT dept_head_id
     FROM department
     WHERE (dept_name='Finance' OR dept_name =
'Shipping'))
```

これは変換されません。

IN キーワードと ANY キーワードに続くサブクエリがジョインに変換されるために満たす必要がある条件は同じです。これは偶然ではありません。理由を次の式で説明します。

IN 演算子を使用するクエリは、ANY 演算子を使用するクエリに変換できる

WHERE *column-name* **IN(** *subquery* **)**

これは次の式と論理的に同等です。

WHERE *column-name* **= ANY(** *subquery* **)**

では次のクエリを見てみます。

```
SELECT emp_fname, emp_lname
 FROM employee
 WHERE emp_id IN (
     SELECT dept_head_id
     FROM department
     WHERE (dept_name='Finance' or dept_name =
'Shipping'))
```

前述の 2 つの式が同等であれば、このクエリは次のクエリと同等です。

```
SELECT emp_fname, emp_lname
 FROM employee
 WHERE emp_id = ANY (
     SELECT dept_head_id
     FROM department
     WHERE (dept_name='Finance' or dept_name =
'Shipping'))
```

概念としては、Adaptive Server Anywhere は IN 演算子を使用するクエリを、ANY 演算子を使用するクエリに変換し、それに応じてサブクエリをジョインに変換するかどうかを決定します。

存在テスト

キーワード EXISTS に続くサブクエリは、次の 2 つの条件を満たす場合にだけジョインに変換されます。

- メイン・クエリが GROUP BY 句を含んでおらず、集計クエリでない。または、サブクエリが 1 つの値を返す。
- 'EXISTS (subquery)' の部分が否定されていない。
- サブクエリが相関である。つまり、外部参照を含んでいる。

例

「どの顧客が 2001 年 7 月 13 日以降に発注したか」という要求は、外部参照 **customer.id** = **sales_order.cust_id** 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、ジョインに変換できます。

```
SELECT fname, lname
 FROM customer
 WHERE EXISTS (
     SELECT *
     FROM sales_order
     WHERE (order_date > '2001-07-13')
  AND (customer.id = sales_order.cust_id))
```

EXISTS キーワードは基本的に、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 fname, lname
 FROM customer, sales_order
 WHERE (sales_order.order_date > '2001-07-13')
 AND (customer.id = sales_order.cust_id)
```

第 12 章

データの追加、変更、削除

この章の内容

この章では、データベース中のデータを修正する方法について説明します。

INSERT、UPDATE、DELETE 文を中心に説明を進めます。バルク・ロードとアンロードの文についても説明します。

データ修正文

データの追加、変更、削除に使う文を「**データ修正文**」といいます。最も一般的な文は、次のようなものです。

- **INSERT** テーブルに新規のローを追加。
- **UPDATE** テーブルにある既存のローを変更。
- **DELETE** テーブルから特定のローを削除。

単一の INSERT、UPDATE、または DELETE 文は、いずれも 1 つのテーブルまたはビューのデータだけを変更します。

一般的な文以外に、LOAD TABLE、TRUNCATE TABLE 文は、特にデータのバルク・ロードと削除に便利です。

データ修正文はまとめて、SQL の「**データ修正言語** (DML)」部分として知られています。

データ修正のパーミッション

修正するデータベース・テーブルに適切なパーミッションがある場合だけ、データの修正文を実行できます。データベースの管理者とデータベース・オブジェクトの所有者は GRANT 文と REVOKE 文を使用して、だれがどのデータ修正機能にアクセスするかを決定します。

パーミッションを個人ユーザ、グループ、または public グループに付与できます。パーミッションの詳細については、『ASA データベース管理ガイド』> 「ユーザ ID とパーミッションの管理」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、トランザクション・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。

トランザクションの詳細については、「トランザクションと独立性レベル」133 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 つのフォームがあります。VALUES キーワードまたは SELECT 文を使用できます。

値を使う INSERT

VALUES キーワードで新しいロー内の一部、またはすべてのカラムの値を指定します。VALUES キーワードを使った INSERT 文の簡略バージョンの構文は、次のとおりです。

INSERT [**INTO**] *table-name* [(*column-name*, …)]
    VALUES (*expression* , …)

SELECT * によるクエリを実行した結果に表示される順で、テーブルの各カラムに値を入力すると、カラム名のリストを省略できます。

SELECT からの INSERT

INSERT 文に SELECT 文を使用して、1 つ以上のテーブルから値を引き出せます。データを挿入するテーブルに多数のカラムがある場合は、WITH AUTO NAME を使用して構文を簡単にすることもできます。WITH AUTO NAME を使用する場合、カラム名を指定する必要があるのは、INSERT 文と SELECT 文の両方ではなく、SELECT 文のみです。SELECT 文の名前には、カラム参照かエイリアスの式を指定してください。

SELECT 文を使用した INSERT 文の簡略バージョンの構文は、次のとおりです。

INSERT [**INTO**] *table-name*
    WITH AUTO NAME *select-statement*

INSERT 文の詳細については、『ASA SQL リファレンス・マニュアル』> 「INSERT 文」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、department 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 department
 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表示されます。

たとえば dept_id と dept_name 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 department (dept_id, dept_name)
 VALUES ( 703, 'Western Sales' )
```

dept_head_id カラムにはデフォルトがありませんが、NULL を使用できます。このカラムには NULL が割り当てられます。

カラム名をリストする順番は、値をリストする順番と一致させます。次の例は、前の例と同じ結果になります。

```
INSERT INTO department (dept_name, dept_id )
 VALUES ('Western Sales', 703)
```

指定されないカラムの値

ローにある一部のカラムだけに値を指定すると、値が指定されないカラムには次のいずれかが発生します。

- **NULL が入力される**　カラムに NULL を使用でき、そのカラムにデフォルト値がない場合は、NULL が表示される。
- **デフォルト値が入力される**　カラムにデフォルト値がある場合は、そのデフォルト値が表示される。

- **ユニークな連続値が入力される** カラムにオートインクリメント・デフォルトか IDENTITY プロパティがある場合、ユニークな連続値が表示される。
- **INSERT が拒否され、エラー・メッセージが表示される** カラムに NULL を使用できず、デフォルトもない場合、エラー・メッセージが表示される。

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。

制約を使用したカラム・データの制限

カラムまたはドメインに対する制約を作成できます。制約はそのデータの種類によって、追加の可否を決定できます。

制約の詳細については、「テーブル制約とカラム制約の使い方」114 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 dept_head_id カラムに NULL を明示的に挿入します。

```
INSERT INTO department
 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表示されるように、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入力します。

デフォルトの詳細については、「カラム・デフォルトの使い方」107 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 Adaptive Server が自動的に変換できるデータ型にします。

例

カラムが CREATE TABLE 文で同じ順序になっている場合、どちらのテーブルのカラム名も指定する必要はありません。product テーブルにあるのと同じフォーマットの製品情報を持つローが、newproduct というテーブル内にいくつか含まれているとします。product に newproduct のすべてのローを追加するには、次のようにします。

```
INSERT product
 SELECT *
 FROM newproduct
```

INSERT 文中の SELECT 文に式を使用できます。

一部のカラムへのデータ挿入

SELECT 文を使用して、VALUES 句を使用する場合と同様、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、product 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 product 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 10 を加えます。

```
INSERT INTO product
 SELECT id+ 10, name, description,
     'Extra large', color, 50, unit_price
 FROM product
 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の LONG BINARY カラムに格納する場合は、ファイルの内容を変数に読み込んで、その変数を INSERT 文の値として指定するアプリケーションを記述できます。

アプリケーションへの INSERT 文の追加の詳細については、『ASA プログラミング・ガイド』> 「準備文の使用方法」と 『ASA SQL リファレンス・マニュアル』> 「SET 文」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。

この外部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' ) )
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「xp_read_file システム・プロシージャ」を参照してください。

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*

サンプル・データベースの customer テーブル内の会社 Newton Ent. 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 customer
 SET company_name = 'Einstein, Inc.'
 WHERE company_name = '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 customer
 SET company_name = 'Einstein, Inc.'
 WHERE company_name 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 customer
 SET company_name = 'Einstein, Inc.'
 WHERE id = 109
```

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 product
 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、『ASA SQL リファレンス・マニュアル』> 「INSERT 文」を参照してください。

DELETE によるデータの削除

DELETE 文は、次のような単純なフォームになっています。

DELETE [**FROM**] *table-name*
  WHERE *column-name* = *expression*

次のような、より複雑なフォームも使用できます。

DELETE [**FROM**] *table-name*
  FROM *table-list*
  WHERE *search-condition*

WHERE 句

WHERE 句を使用して削除するローを指定します。WHERE 句がないと、DELETE 文によってテーブルのすべてのローが削除されます。

FROM 句

DELETE 文の 2 番目に位置する FROM 句には、テーブルからデータを選択し、最初に指定されたテーブルから一致するデータを削除する、特別な働きがあります。FROM 句で選択したローに、削除の条件を指定します。

例

この例ではサンプル・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 OFF 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 = 'OFF'
```

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  product.id,
     SUM(product.unit_price *
sales_order_items.quantity) as "Value Sold"
 FROM  product JOIN sales_order_items
 ON product.id = sales_order_items.prod_id
 GROUP BY product.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 product テーブルから削除できます。

```
DELETE
 FROM product
 FROM product NATURAL JOIN ProductPopularity
 WHERE "Value Sold" < 20000
```

例を完了したら、変更をロールバックしてください。

```
ROLLBACK
```

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、トランザクション・ログにはテーブルのトランケート操作が個別に記録されないためです。

DROP TABLE 文を実行しないかぎり、TRUNCATE TABLE 文によって空になったテーブルの定義は、インデックスや他の関連オブジェクトとともにデータベースに残されます。

他のテーブルに参照整合性制約で参照するローがあると、TRUNCATE TABLE 文を使用できません。外部テーブルからローを削除するか、外部テーブルをトランケートしてからプライマリ・テーブルをトランケートします。

TRUNCATE TABLE 構文

TRUNCATE TABLE 文の構文は次のとおりです。

TRUNCATE TABLE *table-name*

たとえば、sales_order テーブルのデータをすべて削除するには、次の文を入力します。

```
TRUNCATE TABLE sales_order
```

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。

第 13 章

クエリの最適化と実行

この章の内容

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。この章では、オプティマイザがクエリを最適化する手順について説明します。まず、オプティマイザの設計の基本となる仮定条件について説明し、次に選択性推定、コスト推定、その他の最適化手順について説明します。

更新文、挿入文、削除文も最適化する必要がありますが、この章では選択クエリを中心に説明します。その他のコマンドの最適化も同様の原則に従います。

オプティマイザの役割

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。オプティマイザは、選択した方式を「**アクセス・プラン**」というフォームで表します。アクセス・プランには、スキャンするテーブル、インデックスがある場合は各テーブルに使用するインデックス、使用するジョイン方式、テーブルを読み込む順序が記述されています。同じ目的を実行するプランが数多く存在することがよくあります。また、変数を変えることによって、可能なアクセス・プランの数がさらに増える場合があります。

コストベース

オプティマイザは、場合によっては独自の、効率的なアルゴリズムを使用して、可能な選択肢から選択を開始します。オプティマイザの判断の基になるのは、各クエリが必要とするリソースの予測です。オプティマイザは、ディスク・アクセス操作のコストと各操作で推定される CPU コストの両方を考慮します。

構文に依存しない

ほとんどのコマンドは、SQL 言語を使用してさまざまな方法で表すことができます。これらの表現は、同じタスクを実行するという点でセマンティック上は等しくなりますが、構文はまったく異なる場合があります。ごくわずかな例外はありますが、Adaptive Server Anywhere 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。

良いプランが必ずしも最高のプランではない

オプティマイザの目的は、良いアクセス・プランを見つけることです。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。

コストがかかる複雑なクエリの場合、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。

オプティマイザの仕組み

Adaptive Server Anywhere オプティマイザは、クエリでテーブルにアクセスする順序と、テーブルごとにインデックスを使用するかどうかを決定します。オプティマイザは、最適の方式を選択しようとします。

各クエリを実行する最適な方式は、最短の時間と最低のコストで結果を得るものです。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各方式のコストを決定し、コストが最低の方式を選択します。

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。特定のコスト・モデルの詳細は、sa_get_dtt() ストアド・プロシージャで確認できます。

デフォルトでは、クエリ処理はすべての結果セットを返すように最適化されています。OPTIMIZATION_GOAL オプションを使用してこのデフォルトを変更すると、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。

[結果] ウィンドウ枠の [プラン] タブを開いて、任意のクエリのアクセス・プランを Interactive SQL で表示できます。表示される詳細の程度を変更するには、([ツール] メニューから) [オプション] ダイアログを開いて、[プラン] タブの設定を変更します。

最適化ゴールの詳細については、『ASA データベース管理ガイド』>「OPTIMIZATION_GOAL オプション [データベース]」を参照してください。

アクセス・プランの解釈の詳細については、「アクセス・プランの解釈」564 ページを参照してください。

オプティマイザの推定とヒストグラム

オプティマイザは、データベースに格納されている**ヒストグラム**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。

ヒストグラム

ヒストグラムはカラム統計とも呼ばれ、単一カラムの値の分散に関する情報を格納します。Adaptive Server 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。

Adaptive Server 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。Adaptive Server 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。

Adaptive Server Anywhere は、クエリ実行の副次的結果としてヒストグラムを更新します。クエリが実行されると、Adaptive Server Anywhere は、特定の述部に対してヒストグラムが推定したローの数と、その述部を満たすために実際に検索されたローの数を比較してから、ヒストグラムの値を調整して、その後に行われる最適化の誤差限界を減らします。

Adaptive Server Anywhere は、LOAD TABLE 文と UPDATE 文、また CREATE INDEX 文中でもヒストグラムを作成または更新します。

ヒューリスティック

採用できそうな実行プラン内の各テーブルについて、オプティマイザは結果の一部となるローの数を推定します。ローの数は、テーブルのサイズとクエリの WHERE 句または ON 句に指定された制限によって異なります。

Adaptive Server Anywhere は、カラムに対する特定のクエリ述部を満たすローの数を、そのカラムに対するヒストグラムによって推定します。そのためには、指定の述部を満たす値を含むすべての範囲にあるローの数を合算します。クエリの結果セットに部分的に含まれるヒストグラムの値の範囲には、内挿法が使用されます。

多くの場合、オプティマイザはより高度なヒューリスティックを使用します。たとえば、オプティマイザは適切な統計がない場合にかぎってデフォルトの推定値を使用します。また、オプティマイザは、インデックスとキーを使用してロー数の推定精度を上げます。単一のカラムで推定する例を次に示します。

- カラム内である値を持つロー数：そのカラムがユニーク・インデックスを持つかプライマリ・キーである場合、ロー数は 1 つだと推定します。
- インデックス付きカラムで定数と比較したときのロー数：インデックスを調査し、比較条件を満たすローのパーセンテージを推定します。
- 外部キーからプライマリ・キーへのロー数 (キー・ジョイン)：テーブルの相対的サイズを使って推定します。たとえば、5000 ローのテーブルが 1000 ローのテーブルに対して外部キーを持つとき、オプティマイザは 1 つのプライマリ・キー・ローに対して 5 個の外部キー・ローがあると推定します。

参照

ヒストグラムの詳細については、『ASA SQL リファレンス・マニュアル』> 「SYSCOLSTAT システム・テーブル」を参照してください。

述部の選択性とカラム値の分散を取得する方法の詳細については、次を参照してください。

- 『ASA SQL リファレンス・マニュアル』> 「sa_get_histogram システム・プロシージャ」
- 『ASA データベース管理ガイド』> 「ヒストグラム・ユーティリティ」
- 『ASA SQL リファレンス・マニュアル』> 「ESTIMATE 関数 [その他]」
- 『ASA SQL リファレンス・マニュアル』> 「ESTIMATE_SOURCE 関数 [その他]」

カラム統計の更新

カラム統計は、データベースのシステム・テーブル SYSCOLSTAT に永久的に格納されます。大量のデータが INSERT 文、UPDATE 文、または DELETE 文を使用して変更されると、統計は自動的に更新されます。

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、DROP STATISTICS 文や CREATE 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 は単に既存の統計を削除します。

利用可能な統計の精度が高くなると、それに従ってオプティマイザがより適切に推定を計算できるため、後続のクエリのパフォーマンスが向上します。ただし、不正確な推定値が問題になるのは、それが原因でクエリの最適化が不十分になる場合だけです。

LOAD TABLE を実行すると、テーブルの統計が作成されます。Adaptive Server Anywhere は 5 つ以上のローを持つテーブルのヒストグラムを自動的に作成します。ただし、CREATE STATISTICS 文を実行したときには、ローの数に関係なく指定されたテーブルのヒストグラムが作成されます。

カラム統計の詳細については、次を参照してください。

- 『ASA SQL リファレンス・マニュアル』> 「SYSCOLSTAT システム・テーブル」
- 『ASA SQL リファレンス・マニュアル』> 「DROP STATISTICS 文」
- 『ASA SQL リファレンス・マニュアル』> 「CREATE STATISTICS 文」

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 employee
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、Adaptive Server 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、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。

基本となる仮定条件

Adaptive Server 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。

Adaptive Server 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。Adaptive Server 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、Adaptive Server Anywhere が別のユーザのクエリを実行して得た知識の恩恵を受けることができます。

このように統計収集のメカニズムは、データベース・サーバにとって不可欠な部分であり、外部のメカニズムを必要としません。これが役立つ状況であると判断した場合、最適化に使用するデータ分散の推定

をデータベース・サーバに提供できます。たとえば、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は、これらを管理し、必要な場合はいつでも更新してください。

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を指定できます。デフォルトは最初のローです。

詳細については、『ASA データベース管理ガイド』>「OPTIMIZATION_GOAL オプション [データベース]」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、『ASA データベース管理ガイド』>「OPTIMIZATION_WORKLOAD オプション [データベース]」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。システム・テーブル SYSCOLSTAT 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、Adaptive Server Anywhere は、クエリ実行中に収集された統計を使用して SYSCOLSTAT 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。

通常、述部を満たすインデックスがある

通常、Adaptive Server Anywhere はインデックスを使用して述部を評価できます。インデックスを使用することで、オプティマイザはデータへのアクセスをスピードアップし、読み込む情報量を減らします。たとえば、OPTIMIZATION_GOAL が first-row に設定されている場合、Adaptive Server Anywhere オプティマイザはインデックスを使用して ORDER BY 句と GROUP BY 句の指定を満たそうとします。

適切なインデックスが見つからないと、オプティマイザは逐次テーブル・スキャンを行いますが、これにはコストがかかります。テーブルをジョインする場合にインデックスがあれば、パフォーマンスを大幅に向上させることができます。一般的な要求を効率的に処理できるようになる場合には、インデックスをテーブルに追加するか、クエリを書き換えてください。

仮想メモリは貴重なリソースである

オペレーティング・システムと多くのアプリケーションは、一般的なコンピュータのメモリを頻繁に共有します。Adaptive Server Anywhere は、メモリを貴重なリソースとして扱います。Adaptive Server 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マシン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。

Adaptive Server 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。

サブクエリを EXISTS 述部として書き換える

Adaptive Server 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。Adaptive Server Anywhere は、これらの目的を遂行するため、IN、ANY、SOME 述部などの集合演算サブクエリすべてを EXISTS 述部として書き換えます。これにより、Adaptive Server 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**」は、クエリの残りのさらに高いレベルに格納されている 1 つまたは複数のテーブルへの明示的な参照を持たないサブクエリです。

次は、非相関サブクエリを持つ通常のクエリを示しています。このクエリは、2001 年 1 月 1 日に注文していないすべての顧客に関する情報を選択します。

非相関サブクエリ

```
SELECT *
 FROM customer c
 WHERE c.id NOT IN
     (    SELECT o.cust_id
         FROM sales_order o
         WHERE o.order_date = '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 sales_order テーブルを読み込み、2001 年 1 月 1 日に注文したすべての顧客のワーク・テーブルを作成します。次に、customer テーブルを読み込んで、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、Adaptive Server 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、EXISTS 述部を使用して、このようなクエリを書き換えます。このフォームでは、サブクエリが「**相関**」サブクエリになり、customer テーブルの id カラムへの明示的な参照を持つようになります。

相関サブクエリ

```
SELECT *
 FROM customer c
 WHERE NOT EXISTS
     (    SELECT *
         FROM sales_order o
         WHERE o.order_date = '2000-01-01'
             AND o.cust_id = c.id )

c<seq> :o<key_so_customer>
```

このクエリは、上記のクエリとセマンティック上は等しくなりますが、この新しい構文で表すと、次の 2 つの利点が明らかになります。

1. オプティマイザは、sales_order テーブルの cust_id 属性または order_date 属性のインデックスを選択して使用できます (ただし、サンプル・データベースでは、id カラムと cust_id 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。

Adaptive Server Anywhere は、処理中にこの相関サブクエリの結果をキャッシュできます。この方式では、Adaptive Server Anywhere は以前に計算された結果を再利用します。上記のクエリの場合、顧客の ID 番号が customer 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。

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」562 ページを参照してください。

アクセス・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、そのアクセス・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。クエリと、ストアド・プロシージャ、ストアド関数、トリガ内で実行される INSERT、UPDATE、DELETE の各文の場合、オプティマイザは、クエリが実行されてから次に実行されるまでの間、実行プランをキャッシュします。

ストアド・プロシージャ、ストアド関数、またはトリガ内の 1 つの文が 1 つの接続によって複数回実行されると、オプティマイザはその文について再利用可能なプランを作成します。再利用可能なプランでは、選択性推定やリライト最適化にホスト変数の値は使用されません。このため、再利用可能なプランの方が高コストになることがあります。再利用可能なプランのコストが文に最適と思われるコストに近い場合、オプティマイザは再利用可能なプランをプラン・キャッシュに追加します。それ以外の場合は、最適化を避けることによる節約よりも実行のたびに最適化する利点の方が重要なので、実行プランはキャッシュされません。

プラン・キャッシュは、アクセス・プランを実行したときに使われるデータ構造の接続ごとのキャッシュです。キャッシュされたプランの再利用には、キャッシュ内のプランを調べて初期状態にリセットする処理も含まれます。通常、これは文を最適化するよりかなり高速です。使用頻度が低く、キャッシュの使用量が増えない場合、キャッシュされたプランはディスクに格納されます。オプティマイザはクエリを定期的に最適化し直して、キャッシュされたプランの方がまだ効率的であるかどうかを確認します。

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。

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 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 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。

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。

詳細については、『ASA データベース管理ガイド』>「MAX_PLANS_CACHED オプション [データベース]」を参照してください。

最適化の手順

Adaptive Server Anywhere オプティマイザは次の手順に従って、適切なアクセス・プランを生成します。

1. パーサが、SQL クエリを内部表現に変換します。クエリが書き換えられて、構文的に異なるフォームに変換されることがありますが、セマンティック上は変わりません。たとえば、サブクエリがジョインに書き換えられることがあります。これらの変換により、文の分析が容易になります。

2. 最適化が実行直前に正しく開始されます。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。ほかの多くの商用データベース・システムとは異なり、Adaptive Server Anywhere は各文を実行する直前に最適化を行います。

3. オプティマイザは、文に対してセマンティックの最適化を行います。より良い、効率的なアクセス・プランになる場合は、必ずオプティマイザが各 SQL 文を書き換えます。

4. オプティマイザが、各サブクエリに対してジョイン列挙を行います。

5. オプティマイザがアクセス順を最適化します。

Adaptive Server 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。したがって、より良い選択性分析を実行できるため、より良い選択ができることになります。

Adaptive Server Anywhere は、実行される各クエリを、以前の実行回数に関係なく最適化します。ただし、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は例外です。ストアド・プロシージャまたはユーザ定義関数内のクエリの場合、オプティマイザは再利用できるようにアクセス・プランをキャッシュします。

詳細については、「アクセス・プランのキャッシュ」507 ページを参照してください。

Adaptive Server Anywhere では、クエリを実行するたびに統計が保存されるため、オプティマイザは以前のプランを実行した経験から学習し、適切な場合には、その選択を調整できます。

単純なクエリ

クエリが単純なもの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。オプティマイザは、インデックス・スキャンと逐次テーブル・スキャンのどちらを使用するかを決定し、即座にアクセス・プランを作成して実行します。このとき、手順 4 と 5 は省かれます。

単純なクエリとは、単一のカラムに対して、サブクエリ、複数のテーブル、プロキシ・テーブル、ユーザ定義関数、NUMBER(*)、UNION、集合関数、DISTINCT、GROUP BY、または複数の述部を含まない DYNAMIC SCROLL カーソルまたは NO SCROLL カーソルです。単純なクエリに ORDER BY を使用できるのは、WHERE 句に各プライマリ・キー・カラムの条件がある場合だけです。

クエリ実行アルゴリズム

ここでは、Adaptive Server Anywhere がクエリを計算するために使用するアルゴリズムについて説明します。

テーブル・アクセス・アルゴリズム

単一テーブルへの基本的なアクセス方法は、インデックス・スキャンと逐次テーブル・スキャンです。

インデックス・スキャン

インデックス・スキャンでは、探索条件を満たすローを判断するときにインデックスを使用します。読み込まれるのは、条件を満たすページだけです。インデックスによって、ローをソートして返すことができます。

インデックス・スキャンは、短いプランと長いプランに *correlation_name*<*index_name*> と表示されます。ここで、*correlation_name* は FROM 句に指定された相関名、または指定されていない場合はテーブル名です。*index_name* はインデックス名です。

インデックスは、大規模なテーブルから少数のローを効率的に読み込むメカニズムを提供します。ただし、インデックス・スキャンでは、ページがデータベースからランダムに読み込まれるため、逐次読み込みよりコストが高くなります。また、あるテーブル・ページに探索条件を満たす複数のローがあると、インデックス・スキャンはそのページを複数回参照します。インデックス・スキャンによって一致するページが少数しかないと、そのページがキャッシュに残り、複数のアクセスによる余分な I/O は生じません。ただし、多数のページが探索条件と一致すると、すべてがキャッシュに収まらないことがあります。このため、インデックス・スキャンではディスクから同じページが複数回読み込まれることになります。

OPTIMIZATION_GOAL が first-row に設定されていると、オプティマイザは逐次テーブル・スキャンよりインデックス・スキャンを優先する傾向があります。これは、テーブル・スキャンに比べると、インデックスの方がクエリの最初のローを短時間で返す傾向があるためです。

また、インデックスは、順序付けの要件を満たすためにも使用できます。これは、ORDER BY 句で明示的に定義することも、GROUP BY または DISTINCT 句で暗黙的に必要とすることもあります。順序付き GROUP BY 方式と順序付き DISTINCT 方式は、ハッシュベースのグループ化や DISTINCT より短い時間で最初のローを返すことができます。ただし、結果セット全体を返すときには低速になる場合があります。

探索条件が検索引数可能であり、オプティマイザによる探索条件の選択性推定が低いためにインデックス・スキャンの方が逐次テーブル・スキャンより低コストの場合、オプティマイザはインデックス・スキャンを使用して探索条件を満たします。

インデックス・スキャンは、ローがインデックスからフェッチされた後で検索引数不可能な探索条件を評価することもできます。インデックス・スキャンで条件を評価することは、インデックス・スキャン後にフィルタ内の条件を評価するより多少効率的です。

Adaptive Server Anywhere がインデックスを使用できる場合の詳細については、「述部の分析」547 ページを参照してください。

最適化のゴールの詳細については、『ASA データベース管理ガイド』>「OPTIMIZATION_GOAL オプション [データベース]」を参照してください。

逐次テーブル・スキャン

逐次テーブル・スキャンでは、テーブルの全ページのローすべてが、データベースに格納された順に読み込まれます。

逐次テーブル・スキャンは、短いプランと長いプランに *correlation_name*<*seq*> と表示されます。ここで、*correlation_name* は FROM 句に指定された相関名、または指定されていない場合はテーブル名です。

この種のスキャンが使用されるのは、テーブル・ページの大多数にクエリの探索条件と一致するローがある場合、または適切なインデックスが定義されていない場合です。

逐次テーブル・スキャンではインデックス・スキャンより多数のページが読み込まれる場合があります。ただし、ディスクの連続するブロックからページが読み込まれるため、ディスク I/O はかなり低コス

トです (このパフォーマンスの改善は、データベース・ファイルがディスク上で断片化されていない場合には最適です)。逐次 I/O では、ディスク・ヘッドの移動と回転待ち時間のためのオーバヘッドは最小限になります。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。

独立性レベル 3 の場合は、アクセスするローが探索条件を満たしていなくても、Adaptive Server Anywhere は各ローをロックします。このレベルでは、逐次テーブル・スキャンはテーブル内のローすべてをロックしますが、インデックス・スキャンは探索条件と一致するローにだけをロックします。つまり、マルチユーザ環境で逐次テーブル・スキャンを使用すると、スループットが大幅に悪くなる可能性があります。このため、独立性レベル 3 ではオプティマイザは逐次アクセスよりインデックス・アクセスを優先します。逐次スキャンでは、スキャン中にテーブル・カラムと定数の間の単純な比較述部を効率的に評価できます。スキャン中のテーブルだけを参照するその他の探索条件は、これらの単純な比較の後で評価されます。このアプローチは、逐次スキャンの後でフィルタ内の条件を評価する方法より多少効率的です。

ハッシュ・テーブル・スキャン

ハッシュ・テーブル・スキャンは、Adaptive Server Anywhere がハッシュ・ジョインの構築側をメモリ内テーブルのようにスキャンする演算子です。これを使用して、次のような構造を持つプランを変換することができます。

```
table1<seq>*JH ( arbitrary dfos... ( table2<seq> ) )
```

上記の構造を次のような構造に変換できます。

```
table1<seq>*JF ( arbitrary dfos... ( HTS JNB table2<idx> )
)
```

ここで 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。

ハッシュ・ジョインとスキャンの間で演算子が使用されると、他の演算子で処理される必要のあるローの数が減ります。

この方式は、たとえば、構築側のローの数がインデックスのカーディナリティと比べて小さい場合など、インデックス調査で選択性が高い場合に最も便利です。

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。

オプティマイザは閾値構築側を計算しますが、これはハッシュ・ジョイン代替実行に対する閾値の計算方法と似ていて、さらに実行中に (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。

並列インデックス・スキャン

複数のディスク・スピンドルを持つボリュームでは、Adaptive Server Anywhere はインデックスを並行して使用しながらテーブルをスキャンできます。並列インデックス・スキャンにより、サーバは複数のディスクにまたがるテーブルおよびインデックス・ページを活用して複数の入出力操作を同時にスケジュールできます。並列して実行される各スキャンでは、異なるキーを照合するローを検索して、必要な場合にこれらのページがメモリにあるかを確認します。

注意
並列インデックス・スキャンは、圧縮インデックスではサポートされておらず、独立性レベル 3 ではありません。

IN リスト

IN リスト・アルゴリズムは、インデックスを使用して IN 述部を満たすことができる場合に使用されます。たとえば、次のクエリの場合、オプティマイザはプライマリ・キー・インデックスを使用して employee テーブルにアクセスできることを認識します。

```
SELECT *
 FROM employee
 WHERE emp_id in (102, 105, 129)
```

このアクセスを行うために、左側に特別な IN リスト・テーブルを指定したジョインが構築されます。ローは IN リストからフェッチされ、employee テーブルの調査に使用されます。同じインデックスを使用して、複数の IN リストの条件を満たすことができます。オプティマイザがあるインデックスを使用しないで IN 述部を満たすことを選択した場合 (別のインデックスの方が高パフォーマンスを提供する可能性があるなどの理由で)、IN リストはフィルタ内の述部になります。

抽出テーブル

抽出テーブルは、クエリの FROM 句に含まれている SELECT 文です。SELECT 文の結果セットは、論理的にはテーブルのように扱われます。クエリ・オプティマイザは、クエリの書き換え時、たとえば UNION、INTERSECT、または EXCEPT の操作に基づくセットを含むクエリでも抽出テーブルを生成することがあります。グラフィカル・プランには、抽出テーブルの名前と、計算されたカラムのリストが表示されます。

抽出テーブルの詳細については、「FROM 句 : テーブルの指定」288 ページと 『ASA SQL リファレンス・マニュアル』> 「FROM 句」を参照してください。

例

サンプル・データベースの次のクエリは、グラフィカル・プランに抽出テーブルを持ちます。

```
SELECT emp_id FROM employee
 UNION ALL
 SELECT dept_id FROM (
     SELECT TOP 5 dept_id
     FROM department
     ORDER BY dept_name DESC ) MyDerivedTable
```

再帰テーブル

再帰テーブルは、クエリ内の WITH 句の結果構築される共通テーブル式です。WITH 句は、再帰的な UNION クエリに使用されます。共通テーブル式は、1 つの SELECT 文のスコープ内のみで認識されるテンポラリ・ビューです。

詳細については、「共通テーブル式」391 ページを参照してください。

ジョイン・アルゴリズム

ジョイン・アルゴリズムが必要になるのは、FROM 句に複数のテーブルが指定されている場合です。ユーザは、使用するジョイン・アルゴリズムを指定できません。その指定はオプティマイザが行います。

ネスト・ループ・ジョイン・アルゴリズム

ネスト・ループ・ジョインでは、左側のローごとに右側全体が読み込まれ、左側と右側のジョインが計算されます (オプティマイザは要求内のブロックごとに適切なジョイン順を選択するため、クエリに含まれるテーブルの構文上の順序は重要ではありません)。

オプティマイザがネスト・ループ・ジョインを選択するのは、ジョイン条件に等号条件がない場合、または first-row の時間を最適化する場合などです。

ネスト・ループ・ジョインは右側を何回も読み込むため、右側のコストに大きく影響されます。右側がインデックス・スキャンまたは小さなテーブルの場合は、以前の反復でキャッシュされたページを利用して右側を計算できます。これに対して、右側が逐次テーブル・スキャ

ンまたは多数のローと一致するインデックス・スキャンの場合は、右側をディスクから何回も読み込みます。通常、ネスト・ループ・ジョインの効率は、ほかのジョイン方式より低くなります。ただし、ネスト・ループ・ジョインが最初の一致するローを返す速度は、結果全体を計算してから返すジョイン方式より高速です。

ジョインで構成されるクエリに sensitive セマンティックを提供できるジョイン・アルゴリズムは、ネスト・ループ・ジョインだけです。つまり、ジョイン上の sensitive カーソルは、ネスト・ループ・ジョインを使用した場合のみ実行できます。

セミジョインは、右側から最初の一致するローだけをフェッチします。これは、より効率的なネスト・ループ・ジョインですが、EXISTS キーワード、ときには DISTINCT キーワードの使用時しか使用できません。

ネスト・ループ・セミジョイン・アルゴリズム

前述のネスト・ループ・ジョインと同様に、ネスト・ループ・セミジョイン・アルゴリズムは、ネスト・ループ・アルゴリズムを使用して左側の各ローを右側のローとジョインします。ネスト・ループ・ジョインと同様に、右側が何度も読み込まれるため、大きい入力にはインデックス・スキャンの方が適しています。ただし、ネスト・ループ・セミジョインは 2 つの点でネスト・ループ・ジョインと異なります。第 1 に、セミジョインは左側の値のみを出力します。右側は、結果に表示される左側のローを制限するためだけに使用されます。第 2 に、ネスト・ループ・セミジョイン・アルゴリズムでは、最初に一致するローが見つかった時点ですぐに右側の検索が停止します。ジョインとして書き換えられた存在限定 (IN、SOME、ANY、EXISTS) のネストされたクエリのテーブル式がジョインの入力に含まれる場合、ネスト・ループ・セミジョインをジョイン・アルゴリズムとして使用できます。

ネスト・ブロック・ジョインとソート・ブロック・アルゴリズム

ネスト・ブロック・ジョインは、ブロック・ネスト・ループ・ジョインと呼ばれることもあります。このジョインは、左側からローのブロックを読み込み、ジョイン属性 (ジョイン条件に使用されているカラム) 順にローをソートします。ネスト・ブロック・ジョインの左側の子は、ソート・ブロック・ノードと呼ばれます。同じジョイン属性

を持つローのブロックごとに、右側が 1 回スキャンされます。このアルゴリズムでは、右側の各ローとジョインされるローが左側に複数ある場合には、ネスト・ループ・ジョインよりも良いものです。

同じジョイン属性値を持つローが左側に多数あり、探索条件を満たすインデックスが右側にあると、オプティマイザはネスト・ブロック・ジョインを選択します。

各ネスト・ブロック・ジョインの左側の子は、ソート・ブロック・ノードです。このノードのコストは、左側に入力したローを読み込んでソートするためのコストです。

左側の入力は、ブロック単位でメモリに読み込まれます。左側の入力でテーブルに変更があっても、結果には現れません。このため、ネスト・ブロック・ジョインは sensitive セマンティックを提供できません。

ネスト・ブロック・ジョインは、左側の入力にあるローをロックしてからメモリにコピーされます。

ネスト・ブロック・ジョイン・アルゴリズムはメモリ不足状態では無効になります。

ハッシュ・ジョイン・アルゴリズム

ハッシュ・ジョイン・アルゴリズムは、2 つの入力のうち小さい方のメモリ内ハッシュ・テーブルを作成してから、大きい方の入力を読み込みます。次に、メモリ内ハッシュ・テーブルを調査して一致するローを検索します。見つかったローはワーク・テーブルに書き込まれます。小さい方の入力がメモリに収まらない場合は、ハッシュ・ジョイン演算子によって両方の入力が小さなワーク・テーブルに分割されます。これらの小さくなったワーク・テーブルは、小さい方の入力がメモリに収まるようになるまで再帰的に処理されます。

小さい方の入力がメモリに収まる場合は、大きい方の入力のサイズに関係なく、ハッシュ・ジョイン・アルゴリズムのパフォーマンスが最高になります。通常、入力の一方が他方よりかなり小さいと予測される場合に、オプティマイザはハッシュ・ジョインを選択します。

特定のジョイン属性値を持つすべてのローを保持できるほど十分なキャッシュ・メモリがない環境でハッシュ・ジョイン・アルゴリズムを実行した場合、ハッシュ・ジョイン・アルゴリズムは完了できません。この場合、ハッシュ・ジョイン方式では中間結果が廃棄され、代

わりにインデックスベースのネスト・ループ・ジョインが使用されます。小さい方のテーブルのローがすべて読み込まれ、それを使用してワーク・テーブルが調査され、一致するローが検索されます。このインデックスベースの方式は他のジョイン方式に比べてかなり低速です。また、オプティマイザはクエリの実行中にメモリ不足を検出すると、ハッシュ・ジョインを使用したアクセス・プランの生成を避けます。メモリ不足のためにネスト・ループ方式が必要になると、パフォーマンス・カウンタの値が増分されます。このモニタ値を読むには、QueryLowMemoryStrategy データベースまたは接続プロパティ、または Windows パフォーマンス モニタの [クエリ：メモリ不足時方式] カウンタを使用します。

ハッシュ・ジョイン・アルゴリズムはメモリ不足状態の Windows CE では無効になります。

注意： Windows パフォーマンス モニタは、Windows CE、95、98、ME では使用できません。

詳細については、『ASA データベース管理ガイド』> 「接続レベルのプロパティ」の「QueryLowMemoryStrategy」を参照してください。

ハッシュ・ジョイン・アルゴリズムは、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、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ます。

ハッシュ・ジョインは、入力のローをロックしてからメモリにコピーされます。

ハッシュ・セミジョイン・アルゴリズム

ハッシュ・ジョイン・アルゴリズムのハッシュ・セミジョイン変形は、左側と右側のセミジョインを実行します。前述のネスト・ループ・セミジョインと同様に、右側のローは、結果に表示される左側のローを決定するためだけに使用されます。ハッシュ・セミジョインでは、右側のローを読み込んでメモリ内ハッシュ・テーブルが作成されます。その後、左側の各ローによってこのテーブルが調査されます。一致するローが見つかると、左側のローが直ちに結果に出力され、左側の次のローについて、再び調査プロセスが開始されます。少なくとも 1 つの等号ジョイン条件がないと、クエリ・オプティマイザはハッシュ・セミジョインを考慮しません。ネスト・ループ・セミジョイン

と同様に、ジョインとして書き換えられた存在限定 (IN、SOME、ANY、EXISTS) のネストされたクエリのテーブル式がジョインの入力に含まれる場合、ハッシュ・セミジョインを使用できます。ジョイン条件に不等号が含まれる場合、または右側のインデックス検索を低コストにする適切なインデックスが存在しない場合は、ハッシュ・セミジョインの方がネスト・ループ・セミジョインよりパフォーマンスに優れています。

ハッシュ・ジョインと同様に、操作を完了するのに十分なキャッシュ・メモリがない場合は、ハッシュ・セミジョイン・アルゴリズムがネスト・ループ・セミジョイン方式に戻ることがあります。この場合は、パフォーマンス・カウンタの値が増分されます。このモニタ値を読むには、QueryLowMemoryStrategy データベースまたは接続プロパティ、または Windows パフォーマンス モニタの [クエリ：メモリ不足時方式] カウンタを使用します。

*注意：*Windows パフォーマンス モニタは、Windows CE、95、98、ME では使用できません。

詳細については、『ASA データベース管理ガイド』> 「接続レベルのプロパティ」の「QueryLowMemoryStrategy」を参照してください。

ハッシュ非セミジョイン・アルゴリズム

ハッシュ・ジョイン・アルゴリズムのハッシュ非セミジョイン変形は、左側と右側の非セミジョインを実行します。前述のネスト・ループ非セミジョインと同様に、右側のローは、結果に表示される左側のローを決定するためだけに使用されます。ハッシュ非セミジョインでは、右側のローを読み込んでメモリ内ハッシュ・テーブルが作成されます。その後、左側の各ローによってこのテーブルが調査されます。右側のどのローとも一致しない左側のローだけが出力されます。ネスト・ループ非セミジョインと同様に、非ジョインとして書き換えられたユニバーサルに修飾された (NOT IN、ALL、NOT EXISTS) ネスト・クエリのテーブル式がジョインの入力に含まれる場合、ハッシュ非セミジョインを使用できます。ジョイン条件に不等号が含まれる場合、または右側のインデックス検索を低コストにする適切なインデックスが存在しない場合は、ハッシュ非セミジョインの方がネスト・ループ非セミジョインよりパフォーマンスに優れています。

ハッシュ・ジョインと同様に、操作を完了するのに十分なキャッシュ・メモリがない場合は、ハッシュ非セミジョイン・アルゴリズムがネスト・ループ非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。

注意： Windows パフォーマンス モニタは、Windows CE、95、98、ME では使用できません。

詳細については、『ASA データベース管理ガイド』> 「接続レベルのプロパティ」の「QueryLowMemoryStrategy」を参照してください。

マージ・ジョイン・アルゴリズム

マージ・ジョイン・アルゴリズムは 2 つの入力を読み込みます。このとき、2 つの入力はどちらもジョイン属性で順序付けされています。左側の入力のローごとに、右側の入力のローにソート順にアクセスすることで、一致する右側のローをすべて読み込みます。

入力がまだジョイン属性によって順序付けされていない場合 (以前のマージ・ジョインがあるため、または探索条件を満たすためにインデックスが使用されたためなど)、オプティマイザはソートを追加して正しいロー順を生成します。このソートによってマージ・ジョインにコストが追加されます。

マージ・ジョインがハッシュ・ジョインより優れているのは、マージ・ジョインが同じ属性を対象としている場合に、ソートによるコストを複数のジョインに分散できることです。入力のサイズが同じと思われる場合、またはソートによるコストを複数の操作に分散できる場合、オプティマイザはハッシュ・ジョインよりマージ・ジョインを選択します。

再帰ハッシュ・ジョイン・アルゴリズム

再帰ハッシュ・ジョインは、ハッシュ・ジョイン・アルゴリズムの変形の 1 つで、再帰的な UNION クエリで使用されます。

詳細については、「ハッシュ・ジョイン・アルゴリズム」517 ページと「再帰共通テーブル式」401 ページを参照してください。

再帰左外部ハッシュ・ジョイン・アルゴリズム

再帰左外部ハッシュ・ジョインは、ハッシュ・ジョイン・アルゴリズムの変形の 1 つで、特定の再帰的な UNION クエリで使用されます。

詳細については、「ハッシュ・ジョイン・アルゴリズム」517 ページと「再帰共通テーブル式」401 ページを参照してください。

重複排除アルゴリズム

重複排除演算子は、重複するローを含まない出力を生成します。重複排除ノードは、ネストされたクエリをジョインに変換する場合などに、オプティマイザが採用します。

詳細については、「ハッシュ DISTINCT アルゴリズム」521 ページ、「順序付き DISTINCT アルゴリズム」522 ページ、「インデックス DISTINCT アルゴリズム」522 ページを参照してください。

ハッシュ DISTINCT アルゴリズム

ハッシュ DISTINCT アルゴリズムでは、入力が読み込まれ、メモリ内ハッシュ・テーブルが構築されます。入力のローは、ハッシュ・テーブル内で見つかると無視されます。それ以外の場合は、ワーク・テーブルに書き込まれます。入力全体がメモリ内ハッシュ・テーブルに収まらない場合は、小さなワーク・テーブルに分割され、再帰的に処理されます。

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ハッシュ DISTINCT アルゴリズムが適切に機能します。

ハッシュ DISTINCT はワーク・テーブルを使用するため、insensitive セマンティックまたは value sensitive セマンティックを提供できます。

利用可能なキャッシュ・メモリがほとんどない環境でハッシュ DISTINCT アルゴリズムを実行した場合、ハッシュ DISTINCT アルゴリズムは完了できません。この場合、ハッシュ DISTINCT 方式では中間結果が廃棄され、代わりにインデックス DISTINCT アルゴリズムが使用されます。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。

ハッシュ DISTINCT は、前に返されなかったローを見つけるとすぐにそのローを返します。ただし、ハッシュ DISTINCT の結果が完全に実体化されてから、クエリがローを返さなければなりません。このため、オプティマイザは必要に応じてワーク・テーブルを実行プランに追加します。

また、ハッシュ DISTINCT は入力のローをロックします。

順序付き DISTINCT アルゴリズム

入力がすべてのカラムによって順序付けされている場合は、順序付き DISTINCT を使用できます。このアルゴリズムでは、各ローが読み込まれ、前のローと比較されます。両者が同じであれば後の入力ローは無視され、それ以外の場合は出力されます。順序付き DISTINCT が効果的なのは、ローが (インデックスまたはマージ・ジョインの可能性があるため) すでに順序付けされている場合です。入力が順序付けされていない場合、オプティマイザはソートを挿入します。ワーク・テーブルは、順序付き DISTINCT 自体では使用されませんが、挿入されたソートによって使用されます。

インデックス DISTINCT アルゴリズム

インデックス DISTINCT アルゴリズムは、入力からのユニークなローのワーク・テーブルを管理します。ローが入力から読み込まれると、ワーク・テーブル上のインデックスが検索され、入力ローにこれまであった重複の有無がチェックされます。重複が見つかると、入力のローは無視されます。それ以外の場合は、ワーク・テーブルに挿入されます。SELECT リストのすべてのカラムについて、ワーク・テーブルのインデックスが作成されます。インデックスのパフォーマンスを向上させるために、最初の式としてハッシュ式が追加されます。このハッシュ式は計算値で、SELECT リストに含まれるすべてのカラムの値が埋め込まれています。

インデックス DISTINCT 方式では、DISTINCT ローが検出されるたびに返されます。このため、ほかの重複排除方式に比べると、最初のいくつかのローを迅速に返すことができます。インデックス DISTINCT アルゴリズムは、一度に 2 つのローだけをメモリに格納するので、メモリ容量が極端に小さい状況でも十分に機能します。ただし、DISTINCT ローが多数ある場合は、通常、インデックス DISTINCT ア

ルゴリズムの実行コストの方がハッシュ DISTINCT より悪くなります。DISTINCT ローの格納に使用されるワーク・テーブルがキャッシュに収まらないと、ワーク・テーブルのページがランダム・アクセス・パターンで何度も再読み込みされてしまいます。

インデックス DISTINCT 方式ではワーク・テーブルが使用されるため、sensitive セマンティックを完全には提供できません。ただし、insensitive セマンティックも完全には提供できないので、insensitive カーソルには別のワーク・テーブルが必要です。

また、インデックス DISTINCT 方式は入力のローをロックします。

グループ化アルゴリズム

グループ化アルゴリズムでは、入力の合計が計算されます。グループ化アルゴリズムを適用できるのは、クエリに GROUP BY 句がある場合、またはクエリに集合関数 (`SELECT COUNT(*) FROM T` など) がある場合だけです。

詳細については、「ハッシュ GROUP BY アルゴリズム」523 ページ、「順序付き GROUP BY アルゴリズム」525 ページ、「インデックス GROUP BY アルゴリズム」525 ページ、「単一の GROUP BY アルゴリズム」526 ページを参照してください。

ハッシュ GROUP BY アルゴリズム

ハッシュ GROUP BY アルゴリズムでは、グループ・ローのメモリ内ハッシュ・テーブルが作成されます。ローが入力から読み込まれると、グループ・ローが更新されます。ハッシュ・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。分割用のメモリが足りないと、オプティマイザはハッシュ GROUP BY からの中間結果を廃棄し、インデックス GROUP BY アルゴリズムを使用します。オプティマイザはクエリの実行中にメモリ不足を検出すると、ハッシュ GROUP BY アルゴリズムを使用したアクセス・プランの生成を避けます。

グループがメモリに収まる場合は、入力のサイズに関係なくハッシュ GROUP BY アルゴリズムが適切に機能します。

ハッシュ GROUP BY アルゴリズムは、結果のローをすべて計算してから最初のローを返します。また、完全な sensitive または values sensitive カーソルを満たすために使用できます。ハッシュ GROUP BY の結果が完全に実体化されてから、クエリがローを返されなければなりません。このため、オプティマイザは必要に応じてワーク・テーブルを実行プランに追加します。

ハッシュ GROUP BY SETS アルゴリズム

GROUPING SETS クエリを実行する場合、ハッシュ GROUP BY アルゴリズムの変形が使用されます。

詳細については、「ハッシュ GROUP BY アルゴリズム」523 ページを参照してください。

クラスタード・ハッシュ GROUP BY アルゴリズム

場合によっては、入力テーブルのグループ化カラム内の値はクラスタ化されているので、似たような値が互いに接近して現れます。たとえば、常に現在の日付に設定されているカラムがテーブルに含まれている場合、単一の日付を持つすべてのローがテーブル内で比較的近くなります。クラスタード・ハッシュ GROUP BY アルゴリズムは、このクラスタ化を利用します。

使用可能なメモリよりも大幅に大きいテーブルをグループ化する場合、オプティマイザはクラスタード・ハッシュ GROUP BY を使用する可能性があります。特に、HAVING 述部がローの小さい部分のみを返す場合に効果的です。

クラスタード・ハッシュ GROUP BY アルゴリズムでは、データがクエリ実行と同時に更新されるような環境で使用される場合、オプティマイザの作業の一部が大幅に無駄になる可能性があります。したがって、クラスタード・ハッシュ GROUP BY は、一時バッチ形式の更新や読み込みベースのクエリを特徴とする OLAP 負荷に最適です。OPTIMIZATION_WORKLOAD オプションを **OLAP** に設定して、調査する候補にクラスタード・ハッシュ GROUP BY アルゴリズムを含めるようにオプティマイザに指示します。

詳細については、『ASA データベース管理ガイド』>「OPTIMIZATION_WORKLOAD オプション [データベース]」を参照してください。

順序付き GROUP BY アルゴリズム

順序付き GROUP BY 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。

順序付き GROUP BY SETS アルゴリズム

GROUPING SETS クエリを実行するときに使用される順序付き GROUP BY アルゴリズムの変形です。このアルゴリズムは、使用可能なインデックスが必要です。

詳細については、「順序付き GROUP BY アルゴリズム」525 ページを参照してください。

インデックス GROUP BY アルゴリズム

インデックス GROUP BY アルゴリズムは、インデックス DISTINCT アルゴリズムに似ています。このアルゴリズムでは、グループごとに 1 つのローで構成されるワーク・テーブルが構築されます。入力のローが読み込まれると、インデックスを使用してワーク・テーブル内で関連グループが検索されます。集合関数が更新され、グループ・ローがワーク・テーブルに再び書き込まれます。グループ・レコードが見つからなければ、新しいグループ・レコードが初期化され、ワーク・テーブルに挿入されます。

インデックス GROUP BY が選択されるのは、オプティマイザが入力のサイズを極端に小さいと判断した場合です。

インデックス GROUP BY は結果のローをすべて計算してから、最初のローを返して、入力を完全に実体化します。このアルゴリズムは、FULLY INSENSITIVE 要件を満たすために使用できます。

単一の GROUP BY アルゴリズム

GROUP BY を指定しなければ、単一ローの集合が生成されます。

単一の XML GROUP BY アルゴリズム

XML ベースのクエリを処理している場合に使用するアルゴリズムです。

ソート GROUP BY SETS アルゴリズム

GROUPING SETS を含む OLAP クエリを処理する場合に使用されるアルゴリズムです。

このアルゴリズムは、インデックスがなく、順序付き GROUP BY SETS アルゴリズムを使用できない場合に使用できます。

ソートと UNION アルゴリズム

ソート・アルゴリズムを適用できるのは、クエリに ORDER BY 句が含まれている場合です。UNION アルゴリズムは、UNION クエリに適用できます。

詳細については、「マージ・ソート」526 ページと 「UNION ALL」527 ページ を参照してください。

マージ・ソート

ソート演算子は入力をメモリに読み込み、メモリ内でソートしてからソート結果を出力します。入力全体がメモリに収まらない場合は、複数のソート済み処理が作成されてからマージされます。ソートはすべての入力ローを読み込むまで、ローを返しません。ソートは入力ローをロックします。

利用可能なキャッシュ・メモリがほとんどない環境でマージ・ソート・アルゴリズムを実行すると、マージ・ソート・アルゴリズムが完了できない場合があります。この場合、マージ・ソート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、ほかのジョイン方式よりかなり低速です。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、マージ・ソート・アルゴリズムを使用したアクセス・プランの生成を避けます。メモリ不足のためにインデックスベースの方式が必要な場合は、パフォーマンス・カウンタの値が増分されます。このモニタを読むには、QueryLowMemoryStrategy プロパティまたは Windows パフォーマンス モニタの [クエリ：メモリ不足時方式] カウンタを使用します。

ソート・パフォーマンスは、ソート・キーのサイズ、ローのサイズ、入力の合計サイズの影響を受けます。多数のローの場合は、VALUES SENSITIVE カーソルを使用する方が低コストになることがあります。その場合、SELECT リストのカラムは、ソートに使用されるワーク・テーブルにはコピーされません。ソートでは、出力ローがワーク・テーブルに書き込まれませんが、ソートの結果が実体化されてから、ローがアプリケーションに返されなければなりません。このため、オプティマイザは必要に応じてワーク・テーブルを追加します。

UNION ALL

UNION ALL アルゴリズムでは、重複に関係なく各入力からローが読み込まれ、出力されます。このアルゴリズムは、UNION 句と UNION ALL 句を実装するために使用されます。UNION の場合は、UNION ALL によって生成される重複を削除するための重複排除アルゴリズムが必要です。

再帰 UNION

再帰 UNION アルゴリズムは、再帰的な UNION クエリの実行中に使用されます。

詳細については、「再帰共通テーブル式」401 ページを参照してください。

上位 N のソート

上位 N のソート・アルゴリズムは、TOP N 句と ORDER BY 句を含むクエリで使用されます。このアルゴリズムは、結果セットに必要なローのみをソートする場合に効率的です。

詳細については、『ASA SQL リファレンス・マニュアル』> 「SELECT 文」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・プランには次の方式を使用できます。

フィルタ、事前フィルタ、ハッシュ・フィルタ・アルゴリズム

フィルタは探索条件に該当します。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。

事前フィルタはフィルタと同じですが、入力に依存しません。たとえば、`WHERE 1 = 2` の場合は事前フィルタが該当します。

あるテーブルのローをすべてフェッチしてから、別のテーブルのローをフェッチしてそのテーブルにジョインする実行プランでは、ハッシュ・フィルタを使用できます。ハッシュ・フィルタはビット配列を構築し、最初のテーブルで探索条件と一致する各ローに対して 1 ビットを有効にします。2 番目のテーブルにアクセスするとき、最初のテーブルとジョインできなかったローが拒否されます。

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 とジョインします。R のローをすべて読み込んでから、T のローを読み込み、R とジョインできない T のローを直ちに拒否します。これによって、2 番目のハッシュ・ジョインで格納するローの数を減らすことができます。

詳細については、「WHERE 句：ローの指定」290 ページを参照してください。

ロック・アルゴリズム

ロックは、特定の独立性レベルにロックがあることを示しています。たとえば、独立性レベル 1 では、一度に 1 つのローにだけロックが保持されます。独立性レベル 0 の場合、ロックは取得されませんが、そのノードはロックと呼ばれます。この場合、ロック・ノードではローが引き続き存在しているかどうかが検証されます。

詳細については、「ロックの仕組み」175 ページを参照してください。

ローの制限アルゴリズム

ローの制限は、SELECT 文の TOP n または FIRST 句によって設定されます。

詳細については、『ASA SQL リファレンス・マニュアル』>「SELECT 文」を参照してください。

ブルーム・フィルタ・アルゴリズム

ブルーム・フィルタは、ハッシュ・マップ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。ブルーム・フィルタは、次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、単一のカラムに基づいた 2 つのテーブルのジョインでは、ジョインの基準を満たすかどうか判断するために関連するすべてのローが読み込まれる必要があります。
- クエリ内の後の操作が、操作の結果内のローを参照する場合。たとえば、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。

エクスプロード・アルゴリズム

エクスプロード・アルゴリズムは、EXCEPT や INTERSECT 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。エクスプロード・アルゴリズムは、結果セット内のロー数が正しいことを保証します。

詳細については、「UNION、INTERSECT、EXCEPT を使用した、クエリ結果に対する集合操作の実行」324 ページを参照してください。

Window 関数アルゴリズム

このアルゴリズムは、Window 関数を使用する OLAP クエリを評価するのに使用します。

詳細については、「OLAP 関数の活用」438 ページと 「Window 関数」444 ページを参照してください。

物理データの編成とアクセス

各テーブルやエントリに対する記憶領域の割り付けは、クエリの効率に大きく影響します。次に示す各項目は、クエリの実行速度に影響するため、特に重要です。

挿入されたローに対するディスク割り付け

Adaptive Server 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、Adaptive Server 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、Adaptive Server 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、Adaptive Server 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 K 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。

Adaptive Server Anywhere ではローをどのような順序でも保存できる

Adaptive Server 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、エンジンは、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、Adaptive Server 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。

Adaptive Server 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、『ASA SQL リファレンス・マニュアル』> 「CREATE TABLE 文」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 Adaptive Server 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、エンジンは複数のディスク・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、Adaptive Server 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、Adaptive Server Anywhere は別のローが以前に使用していた領域に新しいローを挿入します。

また、各ページの空き領域のレコードを保持しています。新しいローを挿入するよう要求すると、Adaptive Server Anywhere は、まず既存のページの領域のレコードを検索します。既存のページで十分な領域を

見つけると、新しいローをそのページに配置し、必要であればそのページの内容を再編成します。見つけられない場合には、新しいページを開始します。

ただし、多くのローを削除し、空き領域を使用できる程度の小さなローを新しく挿入しなかった場合、時間の経過とともにデータベース内の情報がまばらになることがあります。その場合は、テーブルを再ロードするか、REORGANIZE TABLE 文を使用してテーブルの断片化を解除します。

詳細については、『ASA SQL リファレンス・マニュアル』>「REORGANIZE TABLE 文」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。

Adaptive Server Anywhere は、少なくとも 2 K のページ・サイズを持つデータベース内に、十分な大きさのテーブルに対応するビットマップを作成します。各テーブルのビットマップには、DB 領域ファイル全体での各テーブル・ページの位置が反映されます。2 K、4 K、または 8 K のページからなるデータベースの場合、サーバは一度に 1 ページを読み込むのではなく、ビットマップを利用して大きなブロック (64 K) のテーブル・ページを読み込みます。これによって、ディスクに対する I/O 操作の合計数が減少し、その結果、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サーバの条件を制御できません。

ビットマップはページ・マップとも呼ばれ、バージョン 8.0 以降で作成されたデータベースにしか使用できないので注意してください。データベースが以前のバージョンからアップグレードされている場合、サーバは条件を満たしていてもそれらのデータベース・テーブルのビットマップは作成しません。ワーク・テーブルやシステム・テーブルに対してはビットマップは作成されません。

4 KB のような大きなページ・サイズを選択する場合、キャッシュのサイズを増やすとよいでしょう。ページが大きいと、同じ領域に格納できるページ数は少なくなります。たとえば、1 MB のメモリには、1 ページを 1 KB とすると 1000 ページ格納できますが、1 ページ 4 KB であれば 250 ページしか格納できません。格納できるページ数は、使用しているデータベースと、アプリケーションが実行するクエリの性質によってまったく異なります。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、Adaptive Server Anywhere は頻繁に使用されるページをディスクにスワップし始めるため、パフォーマンスが低下します。

ページ・サイズは、インデックスにも影響を与えます。デフォルトでは、インデックス・ページのハッシュ・サイズは 10 バイトです。ここには、各インデックス・エントリのデータから最初の約 10 バイトを格納します。これにより、4 K のページを使用する約 200 のファンアウトができます。つまり、各インデックス・ページに 200 のロー、2 レベルのインデックスでは 40,000 のローが格納できます。インデックスのレベルを 1 つ上げると処理できるテーブルは 200 倍大きくなります。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。大規模なデータベースには、4 K のページが必要です。

Adaptive Server 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。

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、Adaptive Server 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 Adaptive Server 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・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。詳細については、「インデックス・コンサルタントの概要」86 ページを参照してください。

どのようなときにインデックスを作成するか

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

- **キーとユニークなカラム**　Adaptive Server 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」538 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。Adaptive Server 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、サンプル・データベース内の employee.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。詳細については、「インデックス・コンサルタントの概要」86 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ほうが上回ってしまいます。

詳細については、「インデックスの編集」79 ページを参照してください。

インデックスのパフォーマンス向上

インデックスから予期したパフォーマンスが得られない場合は、次の措置を検討してください。

- 複合インデックスの再編成
- ページ・サイズの拡大

この 2 つの措置は、次に説明するように、インデックスの選択性とインデックス・ファンアウトを高めることを目的としています。

インデックスの選択性

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

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

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

*注意：*Windows パフォーマンス モニタは、Windows CE、95、98、ME 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」567 ページを参照してください。

FullCompare 関数の詳細については、『ASA データベース管理ガイド』> 「データベース・レベルのプロパティ」を参照してください。

インデックス構造とインデックス・ファンアウト

インデックスは、ツリーのように多数のレベルで編成されています。インデックスの最初のページはルート・ページと呼ばれ、その次の下位レベルの 1 つ以上のページへと分岐して、さらにそれが分岐して、インデックスの最下位レベルに達します。最下位レベルのインデックス・ページは、リーフ・ページと呼ばれます。*n* レベルを持つイン

デックスの場合、特定のローを探すにはインデックス・ページを *n* 回読み込む必要があり、実際のローを含むデータ・ページを 1 回読み込む必要があります。通常、使用頻度の高いインデックス・ページはキャッシュに格納される傾向があるため、ディスクからの読み込みは *n* 回よりも少なくて済みます。

「**インデックス・ファンアウト**」は、1 ページに格納されるインデックス・エントリの数です。ファンアウトが大きなインデックスは、ファンアウトが小さなインデックスよりレベル数が少なくなります。したがって、通常はインデックス・ファンアウトが大きいほど、インデックスのパフォーマンスが向上します。

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

詳細については、『ASA SQL リファレンス・マニュアル』> 「sa_index_levels システム・プロシージャ」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 employee (emp_lname, emp_fname)
```

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、emp_lname と emp_fname の複合インデックスは、多数の従業員が同じ姓を持っている場合に役立ちます。各従業員はユニークな ID を持っており、カラム emp_lname は追加の選択性を提供しないため、emp_id と emp_lname 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。

圧縮 B ツリー・インデックス方式は、複合インデックスのパフォーマンスを大幅に向上させます。

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。

複数のカラムを持つプライマリ・キーには、プライマリ・キー定義ではなくテーブル定義に指定されている順番にカラムを含む複合インデックスが常に自動的に作成されます。プライマリ・キーを含めて実行する検索を考慮して、どのカラムを最初に置くのかを決定してください。また、プライマリ・キーが頻繁に検索される場合は、その後のカラムにインデックスを追加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 fname_lname
     ON employee emp_fname, emp_lname;
 CREATE INDEX lname_fname
     ON employee emp_lname, emp_fname;
```

次に、名前 John を検索するとします。使用できる唯一のインデックスは、インデックスの最初のカラムに名前を格納しているインデックスです。姓、名前の順に編成されているインデックスは使用できません。これは、John という名前の人がインデックスのどこに現れるかわからないためです。

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

または、それぞれが 1 つのカラムだけを含むインデックスを 2 つ作成する方法もあります。ただし、Adaptive Server 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、Adaptive Server 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。

プライマリ・キー・インデックスとカラムの順序

プライマリ・キー・インデックスのカラムの順序は、PRIMARY KEY 制約に指定されたカラムの順序に関係なく、テーブル定義にカラムが指定されている順序と同じになります。さらに、Adaptive Server Anywhere は、テーブルのプライマリ・キー・カラムが各ローの先頭になるように追加の制約を確保します。したがって、既存のテーブルにプライマリ・キーが追加されると、サーバはキー・カラムが各ローの先頭になるようにテーブル全体を書き換える場合があります。

プライマリ・キーに複数のカラムが表示される場合、必要な検索のタイプを検討してください。適切であれば、最も頻繁に検索されるカラムが最初に表示されるよう、テーブルの定義でカラムの順序を切り替えるか、必要に応じてその他のカラムに対して別のインデックスを作成してください。

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、Adaptive Server 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
```

および

```
SELECT col1, col2, col3 from example
 ORDER BY col1 DESC, col2 ASC, col3 DESC
```

ORDER BY 句に ASC と DESC のほか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。次に例を示します。

```
SELECT col1, col2, col3 from table1
 ORDER BY col1 ASC, col2 ASC, col3 ASC
```

このクエリは最適化されません。

インデックスのその他の使用法

Adaptive Server 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、Adaptive Server 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、Adaptive Server 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、Adaptive Server 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」175 ページを参照してください。

- **選択性の推定**　インデックスは順序付けされているため、オプティマイザはインデックスの上位レベルをスキャンすることで、特定のクエリを満たす値のパーセンテージを推定できます。このアクションは、部分インデックス・スキャンと呼ばれます。

インデックスのタイプ

Adaptive Server Anywhere は 2 種類のインデックスをサポートしており、インデックス・カラムの宣言された幅に応じて、一方が自動的に選択されます。カラム幅の合計が 10 バイト未満の場合、Adaptive Server Anywhere は、順序保管コード、つまりハッシュ値を含む B ツリー・インデックスを使用して、インデックス・データを表します。インデックス・キー長がデータベースのページ・サイズの 1/8 または 256 バイト (のどちらか小さい方) より長い場合にも、ハッシュ B ツリー・インデックスを使用します。宣言された長さの合計がこの 2 つの範囲の間にあるデータ値の場合、Adaptive Server Anywhere は各キーを圧縮形式で格納する圧縮 B ツリー・インデックスを使用します。

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格納されます。クラスタード・インデックスはパフォーマンスを向上させますが、1 つのテーブルにつき 1 つのインデックスしかクラスタできません。

ハッシュ B ツリー・インデックス

ハッシュ B ツリー・インデックスには、テーブルから実際のローの値が格納されることはありません。代わりに、オリジナル・データの順序保管コードが格納されます。このハッシュ値の格納に使用される各インデックス・エントリのバイト数はハッシュ・サイズと呼ばれ、ハッシュ・サイズはインデックス・カラムすべての宣言された幅に基づきサーバによって自動的に選択されます。サーバは、インデックスを検索して特定のローを探すときに、これらのハッシュされた値を比較します。

ハッシュ値

Adaptive Server Anywhere は、値の順序付けを決定するため、値をインデックスで表します。たとえば、名前のカラムにインデックスを付ける場合、Amos が Smith の前になることを知っていなければなりません。

Adaptive Server Anywhere は、インデックス内の各値に対して、対応するハッシュ値を作成します。インデックスには実際の値ではなく、ハッシュ値が保管されます。Adaptive Server Anywhere はハッシュ値を使用して操作を実行できます。たとえば、2 つの値が等しいかどうか、または 2 つの値のどちらが大きいかなどが、わかります。

整数のように小さな記憶タイプにインデックスを付ける場合、Adaptive Server Anywhere で作成されたハッシュ値には、元の値と同じ量の領域が割り当てられます。たとえば、整数のハッシュ値のサイズは 4 バイトであり、これは整数の保管に必要な領域と同じ大きさです。ハッシュ値のサイズが同じなので、Adaptive Server Anywhere は実際の値と 1 対 1 対応でハッシュ値を使用できます。Adaptive Server Anywhere はハッシュ値を比較して、2 つの値が等しいかどうか、またはどちらの値が大きいかを常に知ることができます。ただし、実際の値を取り出すには、対応するテーブルからエントリを読み込む以外に方法はありません。

大きなデータ型を持つカラムにインデックスを付ける場合、ハッシュ値がデータ型のサイズよりも短くなることがよくあります。たとえば、文字列値のカラムにインデックスを付ける場合、使用されるハッ

シュ値の長さは最大 9 バイトです。このため、Adaptive Server Anywhere は、常にハッシュ値だけを使用して 2 つの文字列を比較できるわけではありません。ハッシュ値が等しい場合、Adaptive Server Anywhere は、テーブルから実際の 2 つの値を取り出して比較します。

たとえば、たくさんの本の似たようなタイトルにインデックスを付けるとします。特定のタイトルを検索する場合、インデックスでは可能性のある複数のローしか識別できません。この場合、Adaptive Server Anywhere は候補となるローをそれぞれ取り出して、タイトル全体を調べます。

複合インデックス

順序付けられた一連のカラムは、複合インデックスとも呼ばれます。しかし、これらのインデックスの各インデックス・キーは最大 9 バイトのハッシュ値になります。したがって、ハッシュ値は必ずしも正しいローをユニークに識別できるわけではありません。2 つのハッシュ値が等しい場合、Adaptive Server Anywhere は実際の値を取り出して比較します。

圧縮 B ツリー・インデックス

圧縮 B ツリー・インデックスは、インデックスの内部ノード内の各インデックス値を圧縮形式で格納します。圧縮形式で格納するため、圧縮 B ツリー・インデックスには Patricia Trie アルゴリズムを使って値が格納されます。これは、データ表現を圧縮するために skip-count で増大された Trie データ構造の最適化された形式です。その結果、圧縮 B ツリー・インデックスは、データ長全体が適度に大きな場合、ハッシュ・インデックスに比べて大幅に向上します。さらに重要なのは、この圧縮アルゴリズムでは同じ (またはほぼ同じ) インデックス値が効率的に処理されることです。そのため、インデックス値に含まれる共通の部分文字列は、必要記憶域とパフォーマンスにほとんど影響しません。インデックス・カラムの宣言された幅の合計が、10 バイト以上で、データベース・ページ・サイズの 1/8 または最大 256 バイトまでの間である場合は、圧縮 B ツリー・インデックスが自動的に選択されます。

ページ・サイズの推奨値

データベースのページ・サイズは、インデックス・ファンアウトに大きく影響します。ページ・サイズが 2 倍になると、インデックス・ファンアウトも約 2 倍になります。

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。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。このため、4 K など、大きなページ・サイズを使用してインデックス・パフォーマンスを向上させることを検討してください。また、長い文字列カラムに圧縮 B ツリー・インデックスを使用してインデックスを付けたいが、ページ・サイズが小さいという制限のためにインデックスを作成できない場合は、ページ・サイズを大きくすることを検討してください。

セマンティック・クエリ変形

効率的に操作するために、通常 Adaptive Server Anywhere はクエリを書き換え、場合によってはいくつかの手順を実行して、新しいフォームにします。Adaptive Server Anywhere は、クエリの表現を変えても、その新しいバージョンで同じ結果が計算されることを保証します。つまり、Adaptive Server Anywhere は、クエリをセマンティック上は等しく、構文上は異なるフォームに書き換えます。

Adaptive Server Anywhere は、さまざまな書き換え操作を実行できます。アクセス・プランを読めば、しばし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に気付きます。たとえば、オプティマイザは可能なかぎりジョインを使ったサブクエリを書き換えようとします。オプティマイザは SQL 文を自由に書き換えることができ、その方法をいくつか備えているという事実は、ユーザにとって重要です。

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。ネストされた条件を置き換えることができます。

```
if ( X != NULL ) {
     if ( X->var != 0 ) {
          ... statements ...
     }
 }
```

これは、次のように短い式に置き換えることができます。

```
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。Adaptive Server 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 言語の仕様では区別されないため、

順序が変わったフォームもセマンティック上は等しくなります。特に、クエリ・オプティマイザは、WHERE 句、HAVING 句、ON 句の述部の順序を完全に自由に変更できます。

述部の分析

「**述部**」は条件式であり、論理演算子 AND や OR と組み合わせて、WHERE 句、HAVING 句、または ON 句に条件のセットを構成します。SQL では、UNKNOWN と評価される述部が FALSE として解釈されます。

インデックスを使用してテーブルからローを取り出すことができる述部を、「検索引数可能 (**sargable**)」であるといいます。この名前は、*search argument-able* というフレーズからとったものです。定数、ほかのカラム、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。Adaptive Server Anywhere はこの文を効率的に評価するため、employee テーブルのプライマリ・インデックスを使用します。

```
SELECT *
 FROM employee
 WHERE employee.emp_id = 123
```

```
employee<employee>
```

反対に、次に示す述部は、検索引数可能では*ありません*。emp_id 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 employee
 where employee.emp_id <> 123
```

```
employee<seq>
```

同様に、名前が "k" という文字で*終わる*すべての従業員を検索する場合、インデックスは役に立ちません。この結果を計算するには、それぞれのローを個別に調べるしかありません。

関数

通常、カラム名に対する関数を持つ述部は、検索引数可能ではありません。たとえば、次に示すクエリにはインデックスは使用されません。

```
SELECT * from sales_order
 WHERE year(order_date)='2000'
```

関数を使用しないようにクエリを書き換えて、検索引数可能にできる場合があります。たとえば、上記のクエリを次のように書き換えることができます。

```
SELECT * from sales_order
 WHERE order_date > '1999-12-31'
 AND order_date < '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、テーブル内のほか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 order_date がある場合は、order_date カラムから抽出した年の値を保持する計算カラム order_year を作成できます。

```
ALTER TABLE sales_order
 ADD order_year INTEGER
 COMPUTE year(order_date)
```

次に、カラム order_year のインデックスを通常どおり追加できます。

```
CREATE INDEX idx_year
 ON sales_order (order_year)
```

次の文を実行します。

```
SELECT * from sales_order
 WHERE year(order_date) = '2000'
```

それに対してサーバは、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。

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(order_date)` が整数の代わりに文字列を返した場合には、オプティマイザは式の計算カラムを置換しません。その結果、必要なローの取り出しにインデックス idx_year を使用できなくなってしまいます。

計算カラムの詳細については、「計算カラムの使用」64 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
x > 25	*x* = 4 OR *y* = 5
x = *z*	*x* = *y*
x IN (4, 5, 6)	*x* NOT IN (4, 5, 6)
x LIKE 'pat%'	*x* LIKE '%tern'
x = 20 - 2	*x* + 2 = 20

述部が検索引数可能かどうか明らかでない場合があります。この場合、述部を、検索引数可能になるように書き換えることができます。各例では、"u" はアルファベット順で "t" の後にくるという事実を利用し、述部 *x* LIKE 'pat%' を *x* >= 'pat' and *x* < 'pau' に書き換えることが可能です。このフォームでは、属性 *x* のインデックスを使用して、一定範囲内の値を検索できます。幸い、Adaptive Server Anywhere では、この特殊な変形を自動で行います。

テーブルでのインデックス検索に使用される検索引数可能な述部は、「**マッチング**」述部です。WHERE 句には、たくさんのマッチング述部が含まれることがあります。最も適切な述部は、ジョイン方式によって決まります。オプティマイザは、ジョイン方式の変更を検討する場合、マッチング述部の選択を再評価します。

セマンティック変形の種類

オプティマイザは、クエリのより効率的で適切な表現方法を探すため、さまざまな変形を実行します。オプティマイザがこれらの変形を実行するため、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 の削除
- サブクエリのネスト解除
- UNION または GROUP 化されたビューでの述部のプッシュダウン
- ジョインの削除
- minimum 関数または maximum 関数の最適化
- OR、リスト内最適化
- LIKE 最適化
- 外部ジョインから内部ジョインへの変換
- 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

以降のサブセクションでは、それぞれの操作について説明します。

不要な DISTINCT の削除

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実際はプライマリ・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、product 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 product p
```

```
p<seq>
```

データベース・サーバは、実際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 product 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。

```
SELECT DISTINCT *
 FROM sales_order o JOIN customer c
     ON o.cust_id = c.id
 WHERE c.state = 'NY'
```

```
c<seq> JNL o<ix_sales_cust>
```

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、Adaptive Server 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。

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、Adaptive Server Anywhere は、これを内部ジョインに変換できるものとみなします。

```
SELECT s.*
 FROM sales_order_items s
 WHERE EXISTS
       (    SELECT *
          FROM product p
          WHERE s.prod_id = p.id
              AND p.id = 300 AND p.quantity > 300)
```

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 product p JOIN sales_order_items s
     ON p.id = s.prod_id
 WHERE p.id = 300 AND p.quantity > 20
```

```
p<seq> JNL s<ky_prod_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 product p
 WHERE EXISTS
      (    SELECT *
         FROM sales_order_items s
         WHERE s.prod_id = p.id
             AND s.id = 2001)
```

Adaptive Server 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 product p JOIN sales_order_items s
     ON p.id = s.prod_id
 WHERE s.id = 2001
```

```
DistI[ s<id_fk> JNL p<product> ]
```

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 つのローしか一致させられない場合、Adaptive Server 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 product p
 WHERE p.id =
      (    SELECT s.prod_id
         FROM sales_order_items s
         WHERE s.id = 2001
             AND s.line_id = 1 )
```

Adaptive Server Anywhere は、このクエリを次のように書き換えます。

```
SELECT p.*
 FROM product p, sales_order_items s
 WHERE p.id = s.prod_id
     AND s.id = 2001
     AND s.line_id = 1
```

```
p<seq> JNL s<sales_order_items>
```

サブクエリのネストを解除するリライト最適化の実行中は、DUMMY テーブルは特殊テーブルとして扱われます。サブクエリをフラットにする処理は、サブクエリ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。

GROUP 化されたビューまたは UNION ビューへ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、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。

例

たとえば、ビュー product_summary が次のように定義されているとします。

```
CREATE VIEW product_summary( product_id, num_orders,
total_qty) as
SELECT prod_id, count(*), sum( quantity )
FROM sales_order_items
GROUP BY prod_id
```

このビューは、注文があった製品ごとに、含まれる注文の件数と、すべての注文の受注数量の合計を返します。ここで、このビューに対する次のクエリを考えてみます。

```
SELECT *
FROM product_summary
WHERE product_id = 300
```

このクエリは、出力を製品 ID 300 に関するものに制限します。このクエリとビューからのクエリを結合して、次のようにセマンティック上は等しい 1 つの SELECT 文にまとめることができます。

```
SELECT prod_id, count(*), sum( quantity )
 FROM sales_order_items
 GROUP BY prod_id
 HAVING prod_id = 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、product_id 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 prod_id, count(*), sum( quantity )
 FROM sales_order_items
 WHERE prod_id = 300
 GROUP BY prod_id
```

これによって、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 prod_id のインデックスを使用して製品 300 のローだけを迅速に取り出すことができます。sales_order_items 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。

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、この最適化が使用されるのは、クエリにプライマリ・キーと外部キーのジョインがあり、プライマリ・テーブルのプライマリ・キー・カラムだけがクエリで参照される場合です。

例

次のクエリを例にとります。

```
SELECT s.id, s.line_id, p.id
 FROM sales_order_items s KEY JOIN product p
```

これは、次のように書き換えられます。

```
SELECT s.id, s.line_id, s.prod_id
 FROM sales_order_items s
 WHERE s.prod_id IS NOT NULL.
```

2 番目のクエリは、sales_order_items テーブルの中で、製品への 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。

ジョインの削除による最適化は、外部ジョインに含ま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。ほかの特定の条件下では、プライマリ・キーとプライマリ・キーのジョインに含まれるテーブルも削除の候補になる場合があります。

この最適化が使用されると、カラムの置換のために DESCRIBE の結果が予想とは異なるものになる可能性があることに注意してください。また、UPDATE 文が削除されたベース・テーブルを 1 つ以上参照していると、UPDATE または DETELE WHERE CURRENT 要求が失敗する場合があります。この問題を回避するには、削除されたテーブルの追加カラムがクエリの SELECT リストにあることを確認するか (最適化を始めから回避する)、または別個の文を使用して必要なローを更新してください。

minimum 関数または maximum 関数の最適化

MIN/MAX リライト最適化は、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、インデックスを使用して単一ロー・ルックアップの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
- WHERE 句には結合等号条件だけが指定されている
- 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、sales_order_items テーブルの (prod_id ASC, quantity ASC) にインデックス prod_qty があるとします。次のクエリを考えてみます。

```
SELECT MIN( quantity )
 FROM sales_order_items
 Where prod_id = 300
```

このクエリは内部的に次のように書き換えられます。

```
SELECT MIN( quantity )
 FROM ( SELECT FIRST quantity
        FROM sales_order_items
        WHERE prod_id = 300 and quantity IS NOT NULL
        ORDER BY prod_id 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アクセス・プラン (省略形) は、次のとおりです。

```
GrByS[ RL[ sales_order_items<prod_qty> ] ]
```

IN リストの最適化

Adaptive Server Anywhere オプティマイザは、インデックス・カラム上で IN 述部を利用する特殊な最適化をサポートしています。この最適化は、同じインデックス・カラムに対して OR で結合された複数の述部にも等しく適用されます。これは、どちらもセマンティック上は等しいためです。最適化を有効にするには、IN リストに定数だけを指定します。

オプティマイザは、修飾する IN リスト述部を検出した場合に、IN リスト述部にインデックス検索を十分に検討できる選択性がある場合には、IN リスト述部をネスト・ループ・ジョインに変換します。この最適化の機能を、次の例で説明します。

次のクエリがあるとします。

```
SELECT *
 FROM sales_order
 WHERE sales_rep = 142 or sales_rep = 1596
```

これは、上記の 2 人の営業担当者が扱うすべての注文をリストします。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 sales_order
 WHERE sales_rep IN (142, 1596)
```

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高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 sales_rep 属性で sales_order 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。

```
IN JNL sales_order<ky_so_employee_id>
```

LIKE 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。

例

次のそれぞれの例で、LIKE 述部のパターンがリテラル定数またはホスト変数であり、X がベース・テーブルのカラムであると仮定します。

- `X LIKE '%'` は `X IS NOT NULL` と書き換えられる
- `X LIKE 'abc'` は `X = 'abc'` と書き換えられる
- `X LIKE 'abc%'` は、述部 `X < 'abcZ'` と `X > = 'abc_'` に拡大される

Z と _ は、このデータベースの照合順に対応する高い値と低い値を表します。ブランクが埋め込まれた文字列を格納するようにデータベースが設定されている場合には、正しいセマンティックになるように、2 つめの比較演算子は >= ではなく > とします。

外部ジョインから内部ジョインへの変換

ほとんどの部分で、オプティマイザはアクセス・プラン用に左側が深い処理ツリーを生成します。このルールの唯一の例外は、右側が深いネスト外部ジョイン式の存在です。LEFT OUTER JOIN または RIGHT OUTER JOIN の計算に使用するクエリ実行エンジンのアルゴリズムでは、どのようなジョイン方式の場合も、保護されたテーブルを NULL 入力テーブルよりも前に置きます。そこで、オプティマイザはできるだけ LEFT または RIGHT 外部ジョインを内部ジョインに変換しようとします。これは、内部ジョインは交換可能で、オプティマイザがジョイン列挙を実行するときの自由度が大きくなるためです。

NULL 入力テーブルに対する NULL を許容しない述部がクエリの WHERE 句にある場合は、LEFT OUTER JOIN または RIGHT OUTER JOIN を内部ジョインに変換できます。この述部は NULL を許容しないため、外部ジョインによって生成されるすべて NULL のローは結果から省かれます。その結果、このクエリはセマンティック上は内部ジョインと等しくなります。

例

次のクエリを例にとります。

```
SELECT *
 FROM product p KEY LEFT OUTER JOIN sales_order_items s
 WHERE s.quantity > 15
```

これは、数量の多い製品すべてとその注文をリストするクエリです。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。このクエリの問題は、`s.quantity` が NULL の場合に WHERE 句の述部 `s.quantity > 15` が FALSE として解釈されるため、注文のない製品がこの述部によって結果から削除されることです。そのため、このクエリは、セマンティック上は次に示すものと同じです。

```
SELECT *
 FROM product p KEY JOIN sales_order_items s
 WHERE s.quantity > 15
```

また、サーバが最適化するのも、クエリを書き換えたこのフォームになります。

この例で、クエリは正しく書き換えられないのはほぼ確実で、次のようになります。

```
SELECT *
 FROM product p
     KEY LEFT OUTER JOIN sales_order_items s
           ON s.quantity > 15
```

そのため、数量テストは外部ジョイン条件の一部になっています。

この最適化が単純な外部ジョイン・クエリに適用されるのはまれで、通常はクエリが外部ジョインを使用して書き換えられる 1 つ以上のビューを参照する場合に適用できます。クエリの WHERE 句には、1 つ以上のテーブル式からの NULL 入力ローをすべて削除するようにビューの出力を制限する条件を含めることができるため、この最適化が適用可能になります。

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/ON/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、Adaptive Server Anywhere は、オプティマイザが利用できる簡略または暗黙的な条件を発見するために、オリジナルのクエリ・テキスト内で探索条件を詳細に分析します。

前処理として、ビューが拡張され、マージされてから、オリジナルの文の述部に対していくつかの簡略化が行われます。次に例を示します。

- `X = X` は、X が NULL 入力可能な場合は `X IS NOT NULL` に書き換えられ、それ以外の場合は述部は削除される
- `ISNULL(X,X)` は単に `X` に書き換えられる
- `X+0` は、X が数値カラムの場合は `X` に書き換えられる

- `AND 1=1` は削除される
- `OR 1=0` は削除される
- 単一要素で構成される IN リスト述部は、単純な等号条件に変換される

この前処理の後に、Adaptive Server 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。Adaptive Server 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

オリジナルのクエリが次のように定義されているとします。

```
SELECT e.emp_lname, s.id, s.order_date
 FROM sales_order s, employee e
 WHERE (e.emp_id = s.sales_rep and
           (s.sales_rep = 142 or s.sales_rep = 1596)
       )
          OR
       ( e.emp_id = s.sales_rep and s.cust_id = 667)
```

このクエリには結合等価ジョイン条件がありません。したがって、オプティマイザは述部の詳細分析を実行しなければ、効率的なアクセス・プランを発見できません。幸い、Adaptive Server 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.emp_lname, s.id, s.order_date
 FROM sales_order as s, employee as e
 WHERE e.emp_id = s.sales_rep AND
     (s.sales_rep = 142 or s.sales_rep = 1596 or
s.cust_id = 667)'
```

これによって、ジョインは内部ジョイン・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、Adaptive Server 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() または LOWER() 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。Adaptive Server 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。大文字と小文字の変換を排除すると、対応するテーブルのインデックス検索に比較述部を使用できるようになります。

例

大文字と小文字を区別しないデータベースに対して、次のクエリを実行するとします。

```
SELECT *
 FROM customer
 WHERE UPPER(lname) = 'SMITH'
```

このクエリは内部的に次のように書き換えられます。

```
SELECT *
 FROM customer
 WHERE lname = 'SMITH'
```

これで、オプティマイザは customer.lname にインデックスを使用することを検討できます。

サブクエリと関数のキャッシュ

Adaptive Server 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、Adaptive Server 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、Adaptive Server 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)、Adaptive Server 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、Adaptive Server 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、Adaptive Server 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、Adaptive Server 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、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、Adaptive Server 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。

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。

ユーザ定義関数は、作成時に NOT DETERMINISTIC と指定されないかぎり、決定性関数として扱われます。

ユーザ定義関数の詳細については、『ASA SQL リファレンス・マニュアル』> 「CREATE FUNCTION 文」を参照してください。

アクセス・プランの解釈

オプティマイザは、すべての文に応答して選択したクエリ最適化方式 (プラン) を通知します。

オプティマイザの仕事は、クエリのセマンティックを把握し、その結果を計算するプランを構築することです。このプランは使用されている構文と正確に一致しないことがあります。オプティマイザは、セマンティック上等しいフォームであれば、クエリを自由に書き換えることができます。

Adaptive Server Anywhere がクエリを書き換える場合に従う規則の詳細については、「サブクエリを EXISTS 述部として書き換える」505 ページと「セマンティック・クエリ変形」546 ページを参照してください。

オプティマイザがクエリの実装に使用する方式については、「クエリ実行アルゴリズム」510 ページを参照してください。

Interactive SQL または SQL 関数を使用すると、プランを表示できます。アクセス・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
- 長いテキスト
- グラフィカル
- 統計情報付きグラフィカル
- Ultra Light (短い、長い、またはグラフィカル)

また、特定のカーソル・タイプの SQL クエリのプランを取得できます。

プランへのアクセス方法の詳細については、「プランへのアクセス」584 ページを参照してください。プランを読み込む方法については、「テキスト・プラン」572 ページと「グラフィカルなプラン」575 ページを参照してください。

アクセス・プランに表示される統計とその他の項目について、次に説明します。

プランに使用される省略形

次の表に、短いプランと、グラフィカルなプランの省略名のフォームに使用される省略形を示します。

名前	短いプラン／省略名
ハッシュ EXCEPT ALL	EAH
ハッシュ EXCEPT	EH
ハッシュ GROUP BY	GrByH
ハッシュ・テーブル・スキャン	HTS
クラスタード・ハッシュ GROUP BY	GrByHClust
ハッシュ ROLLUP GROUP BY	GrByHR
順序付き GROUP BY	GrByO
順序付き ROLLUP GROUP BY	GrByOR
単一ロー GROUP BY	GrByS
インデックス GROUP BY	GrByI
ハッシュ DISTINCT	DistH
インデックス DISTINCT	DistI
順序付き DISTINCT	DistO
上位 N のソート	StrN
ハッシュ・フィルタ	HF
ハッシュ INTERSECT ALL	IAH
ハッシュ INTERSECT	IH
EXISTS ジョイン	JE
ネスト・ループ・セミジョイン	JNLS
ハッシュ EXISTS	JHE

名前	短いプラン／省略名
ハッシュ NOT EXISTS	JHNE
ハッシュ・ジョイン	JH
ソート・ブロック	SrtBl
左外部ハッシュ・ジョイン	JHO
全外部ハッシュ・ジョイン	JHFO
再帰ハッシュ・ジョイン	JHR
左外部再帰ハッシュ・ジョイン	JHRO
ネスト・ブロック・ジョイン	JNB
左外部ネスト・ブロック・ジョイン	JNBO
NOT EXISTS ジョイン	JNE
ネスト・ループ・ジョイン	JNL
左外部ネスト・ループ・ジョイン	JNLO
全外部ネスト・ループ・ジョイン	JNLFO
マージ・ジョイン	JM
左外部マージ・ジョイン	JMO
全外部マージ・ジョイン	JMFO
マージ EXCEPT	EM
マージ EXCEPT ALL	EAM
マージ INTERSECT	IM
マージ INTERSECT ALL	IAM
ロー制限	RL
ロー・レプリケート	RR
抽出テーブル	DT

名前	短いプラン／省略名
再帰テーブル	RT。短いプランの場合は rt<seq>
再帰 UNION	RU
UNION ALL	UA
テーブル・スキャン	短いプランの場合は *tablename*<seq>、グラフィカルなプランの場合はテーブル名
インデックス・スキャン	短いプランの場合は *tablename*<*indexname*>、グラフィカルなプランの場合はテーブル名
IN リスト	IN

各アルゴリズムの説明については、「クエリ実行アルゴリズム」510 ページを参照してください。

プランに使用される一般的な統計

次の統計は実際の測定値です。

統計情報	説明
Invocations	ローがサブツリーから要求された回数
RowsReturned	現在のノードについて返されたローの数
RunTime	子の時間を含めたサブツリーの実行所要時間
CacheHits	成功したキャッシュ読み込み数
CacheRead	キャッシュの中で検索されたデータベース・ページの数
CacheReadTable	キャッシュから読み込まれたテーブル・ページの数
CacheReadIndLeaf	キャッシュから読み込まれたインデックス・リーフ・ページの数
CacheReadIndInt	キャッシュから読み込まれたインデックス内部ノード・ページの数
DiskRead	ディスクから読み込まれたページ数
DiskReadTable	ディスクから読み込まれたテーブル・ページの数

統計情報	説明
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
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	クエリ処理に対し、最初のローを迅速に返すようにするか、完全な結果セットを返す負荷を最小限に抑えるようにするかを決定し、クエリを最適化する方法を指定する 『ASA データベース管理ガイド』>「OPTIMIZATION_GOAL オプション [データベース]」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する 『ASA データベース管理ガイド』>「OPTIMIZATION_WORKLOAD オプション [データベース]」を参照してください。
ANSI update constraints	更新が許される範囲を制御する (オプションは、OFF、CURSORS、STRICT) 『ASA データベース管理ガイド』>「ANSI_UPDATE_CONSTRAINTS オプション [互換性]」を参照してください。
Optimization level	今後の使用のために予約されている
Select list	クエリによって選択される式のリスト

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

項目	説明
Estimated rows	テーブルの推定ロー数
Estimated pages	テーブルの推定ページ数
Estimated row size	テーブルの推定ロー・サイズ
Page maps	複数ページの読み込みにページ・マップが使用される場合は YES

インデックス・スキャンに関連するプランの項目

項目	説明
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 詳細については、「テーブルとページのサイズ」533 ページを参照してください。
Estimated leaf pages	リーフ・ページの推定数
Cardinality	推定ロー数と異なる場合の、インデックスのカーディナリティ。バージョン 6 以前の Adaptive Server Anywhere データベースにのみ適用されます。
Selectivity	範囲バウンドと一致する推定ロー数
Direction	FORWARD または BACKWARD

項目	説明
Range bounds	範囲バウンドは、リスト (col_name=value) または col_name IN [low, high] として表示されます。

ジョイン、フィルタ、事前フィルタに関連するプランの項目

項目	説明
Predicate	このノードで評価される探索条件、選択性推定、測定値。 詳細については、「プラン内の選択性」582 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 操作に関連するカラム

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

項目	説明
Group-by list	GROUP BY 句に指定されているすべてのカラム

DISTINCT に関連するプランの項目

項目	説明
Distinct list	DISTINCT 句に指定されているすべてのカラム

IN リストに関連するプランの項目

項目	説明
In List	指定したセットのすべての式
Expression SQL	リストと比較される式

SORT に関連するプランの項目

項目	説明
Order-by	ソート基準となるすべての式のリスト

ロー制限に関連するプランの項目

項目	説明
Row limit count	FIRST または TOP n で指定された、返されるローの最大数

テキスト・プラン

テキスト・プランには、短いプランと長いプランの 2 種類があります。Interactive SQL でプラン・タイプを選択するには、[ツール] メニューから [オプション] ダイアログを開き、[プラン] タブをクリックします。SQL 関数を使用してプランにアクセスする方法については、「SQL 関数を使用したプランへのアクセス」585 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 sales_order テーブルと sales_order_items テーブルからの外部 SELECT 文、もう 1 つは product テーブルから選択するサブクエリです。

```
SELECT *
 FROM sales_order AS o
     KEY JOIN sales_order_items AS i
 WHERE EXISTS
     (    SELECT *
         FROM product p
         WHERE p.id = 300 )
```

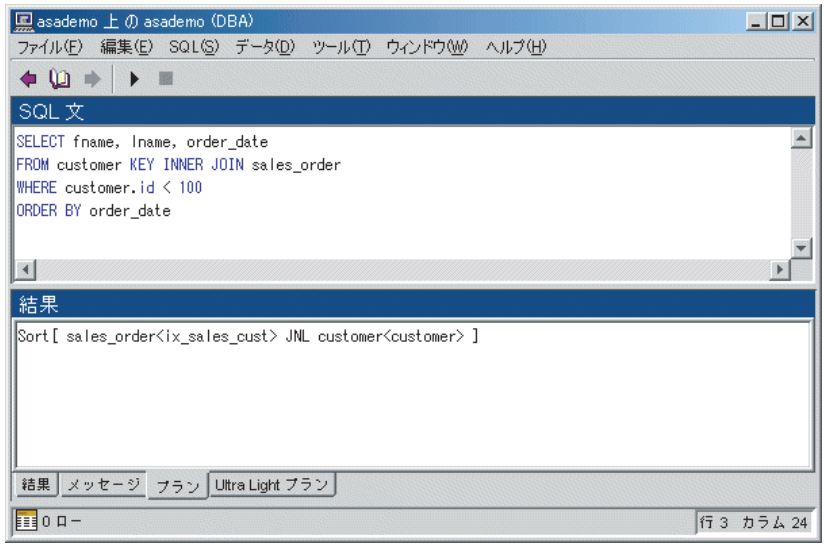
```
i<seq> JNL o<sales_order> :p<seq>
```

ジョイン方式の区切りには、コロンを使用します。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、ほかのクエリ・ブロックのジョイン方式がリストされます。このようなほかのクエリ・ブロックのジョイン方式の順序は、文における順序や文の実行順序とは一致しない場合があります。

短いテキスト・プラン

短いプランは、プランを短時間で比較する場合に便利です。短いプランでは、すべてのアクセス・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 SORT で始まります。customer テーブルは、同じ名前 (customer) のプライマリ・キー・インデックスによってアクセスされます。カラム customer.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、オプティマイザがクエリの処理にネスト・ループ・ジョインを使用していることを示します。最後に、外部キー・インデックス ky_so_customer を使用して sales_order テーブルがアクセスされ、customer テーブル内で一致するローが検索されます。

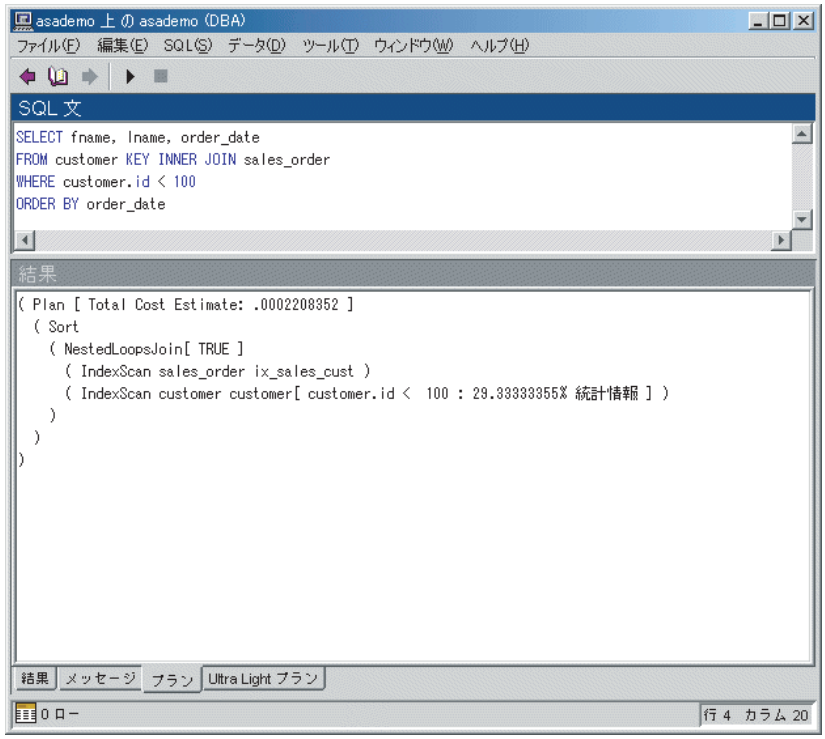


プランに使用されるコード・ワードの詳細については、「プランに使用される省略形」565 ページを参照してください。

長いテキスト・プラン

長いプランでは、短いプランより若干多くの情報が提供されます。また、情報はスクロールしなくても簡単に印刷したり表示したりできる状態で提供されます。

長いプランの場合、上記と同じクエリの出力の 1 行目は `Plan [ I/O Estimate: 1 ]` です。Plan または Sub-plan は、クエリ・ブロックの開始を示します (この場合は 1 つしかありません)。I/O estimates は、クエリに必要な I/O 数 (この場合は 1) を示します。また、このプランは、結果がソートされることと、ジョイン・アルゴリズムとしてネスト・ループ・ジョインが使用されることを示します。アルゴリズムと同じ行に、TRUE、またはアルゴリズムの探索条件と選択性推定があります (この場合はありません)。WHERE 条件は FILTER で始まる行に示され、その後に探索条件とその選択性推定、最後に選択性推定のソースが示されます。



プランに使用されるコード・ワードの詳細については、「プランに使用される省略形」565 ページを参照してください。

グラフィカルなプラン

グラフィカルなプランには、グラフィカルなプランと統計情報付きのグラフィカルなプランの 2 種類があります。Interactive SQL でプラン・タイプを選択するには、[ツール] メニューから [オプション] ダイアログを開き、[プラン] タブをクリックします。SQL 関数を使用してプランにアクセスする方法については、「SQL 関数を使用したプランへのアクセス」585 ページを参照してください。

グラフィカルなプランを表示したら、左ウィンドウ枠を右クリックして [カスタマイズ] を選択し、表示方法を設定します。

後で参照するために、グラフィカルなプランを印刷できます。プランを印刷するには、ノードを右クリックし、[印刷] を選択します。

グラフィカルなプランの各ノードのコンテキスト別ヘルプを表示するには、ノードを選択し、右クリックして [ヘルプ] を選択します。たとえば、[Nested Loops Join] を右クリックして [ヘルプ] を選択すると、実行のその部分で使用されるリソースに関する情報が表示されます。グラフィカルなプランの各要素にカーソルを置くと、ポップアップ情報が表示されます。

グラフィカルなプラン

グラフィカルなプランは、短いプランや長いプランよりも多くの情報を提供します。グラフィカルなプラン、または統計情報付きのグラフィカルなプランを表示するように選択できます。どちらを選択した場合も、プランの中で最も高コストとして推定された部分をすぐに表示できます。統計情報付きのグラフィカルなプランの表示は高コストですが、クエリの実行時にサーバがモニタしている実際のクエリ実行統計が表示されます。このため、クエリ・オプティマイザがアクセス・プランの作成時に使用する推定を、実行中にモニタされた実際の統計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、2 つの推定に違いがあると予想してください。グラフィカルなプランは、アクセス・プランのデフォルト・フォーマットです。

グラフィカルなプランは、主要な情報の一部を視覚的に提供することを目的として設計されています。

- グラフィカルなプランに表示される各操作は、コンテナに入っています。コンテナは、操作の種類がデータの実体化、インデックス・スキャン、テーブル・スキャン、またはその他の操作のいずれであるかを示します。
- プランでは、操作が次の操作に渡すローの数が、操作を結んでいる線の太さによって示されています。これは、クエリ内の大半のデータに対して実行される操作を視覚的に表します。
- 特に遅い操作のコンテナは、赤い枠線で表示されます。

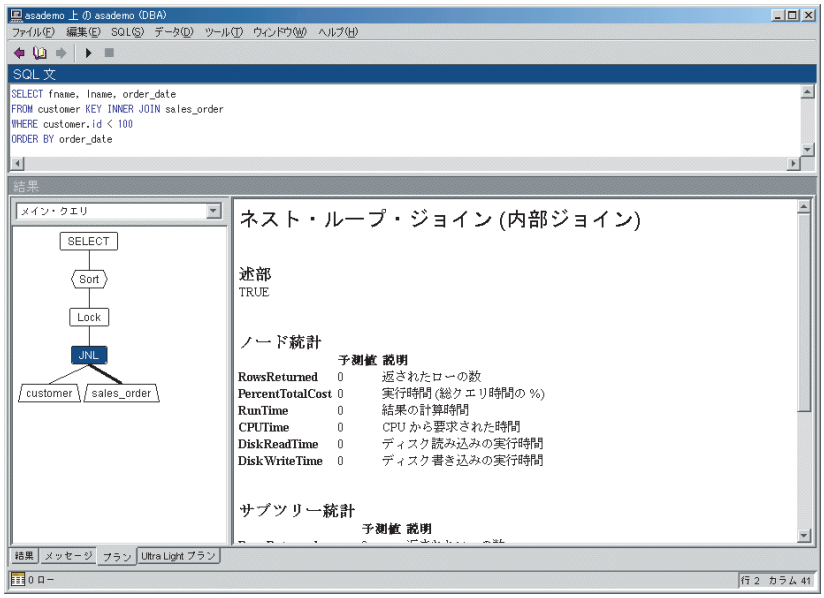
一般に、太い線上のノードと赤いボックスのあるサブツリー内のノードの実行が、クエリ・パフォーマンスを決定します。

これらの表示機能は、それぞれカスタマイズできます。

クエリが単純なクエリとして認識されている場合、一部の最適化手順が省略され、クエリ・オプティマイザ・セクションと述部セクションの両方がグラフィカル・プランに表示されません。単純なクエリの詳細については、「オプティマイザの仕組み」498 ページを参照してください。

次の例では、短いテキスト・プランと長いテキスト・プランの説明に使用したクエリをグラフィカル・プランで表したものです。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。Lock ノードは、結果セットが実体化されること、またはローがアプリケーションに返されることを示します。この場合、ソートするには結果が実体化される必要があります。レベル 0 では、ローは実際にはロックされません。Adaptive Server Anywhere は、ローがベース・テーブルから読み込まれた後に削除されていないことを確認するだけです。レベル 1 では、ローは次のローがアクセスされるまでロックされます。レベル 2 と 3 では読み込みロックが適用され、COMMIT 時まで保持されます。

グラフィカルな図でノードをクリックすると、プラン内のノードに関する詳細情報を取得できます。この例では、nested loops join ノードが選択されています。右ウィンドウ枠には、そのノードに関連する情報だけが表示されています。たとえば、述部記述は単に TRUE となっています。これは、このクエリ実行段階では適用される述部がないことを示します (クエリ内の述部 (customer.id < 100) のみが customer テーブルのクエリ・ノードに適用されます)。



プランに使用される省略形の詳細については、「プランに使用される省略形」565 ページを参照してください。

統計情報付きのグラフィカルなプラン

統計情報付きのグラフィカルなプランには、グラフィカルなプランに表示されるすべての推定に加えて、文を実行したときの実際の実行時間コストが表示されます。これを行うには、文を実際に実行します。これは、コストの高いクエリの場合、プランへのアクセスに遅延が生じる可能性があることを意味します。また、削除や更新など、クエリのどのような部分も実際に実行されますが、ロールバックしてこれらの変更の取り消しができることも意味します。

パフォーマンスに問題があり、推定されるロー・カウントや実行時間が予測とは異なる場合は、統計情報付きのグラフィカルなプランを使用してください。統計情報付きのグラフィカルなプランを使用すると、比較するために推定と実際の統計が提供されます。実際値と推定値が大きく異なる場合は、情報不足のためにオプティマイザが適切なアクセス・プランを選択できない可能性があることを示す警告とみなすことができます。

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、ルート演算子についてのみ表示されます。

次に、統計情報付きのグラフィカルなプランでチェックできる主な統計と、考えられる対応策を示します。

- **選択性統計値** 述部の選択性 (条件式) は、条件を満たすローの割合のことです。推定される述部の選択性が提供する情報に基づいて、オプティマイザはコストの推定を行います。選択性推定が不正確な場合、クエリ・オプティマイザは不適切なアクセス・プランしか生成できません。たとえば、オプティマイザが述部の選択性が高い (5% の選択性など) と誤って推定しても、実際の選択性がかなり低い (50% など) 場合は、不適切なプランを選択してしまう可能性があります。選択性推定は正確なものではありませんが、極端に大きなエラーは問題がある可能性を示しています。

  クエリの重要な部分の選択性情報が不正確であると判断した場合、CREATE STATISTICS を使用して対象となるカラムの新しい統計値セットを生成できます。まれに、潜在的な管理上の問題があっても、明示的な選択性推定を指定できる場合もあります。

  選択性の詳細については、「プラン内の選択性」582 ページを参照してください。

  統計作成の詳細については、『ASA SQL リファレンス・マニュアル』>「CREATE STATISTICS 文」を参照してください。

  ユーザ推定の詳細については、『ASA SQL リファレンス・マニュアル』>「明示的な選択性推定」を参照してください。

  選択性統計は、クエリが単純なクエリと判断された場合に表示されないため、オプティマイザは省略されます。単純なクエリの詳細については、「オプティマイザの仕組み」498 ページを参照してください。

  次のような場合に不適切な選択性の兆候が見られます。

  - **RowsReturned の実際値と推定値**　RowsReturned は結果セット内のロー数です。RowsReturned 統計値は、ツリーの先頭にあるルート・ノードのテーブル内に表示されます。

返された推定ローと返された実際の数とが大幅に食い違っている場合、オプティマイザが不正確な選択性情報に基づいて動作していることを示す警告とみなすことができます。

- **述部選択性の実際値と推定値** 「述部」というサブヘッダを検索して、述部の選択性を確認します。述部情報の解釈については、「プラン内の選択性」582 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。選択性エラーに問題が残る場合は、最後の手段として、クエリ・テキストに述部とともにユーザ推定選択性を指定することも可能です。

- **推定ソース** 選択性推定のソースも、統計ウィンドウ枠の述部サブヘッダの下にリストされています。述部情報の解釈については、「プラン内の選択性」582 ページを参照してください。

  推定ソースが「予測」の場合、オプティマイザが使用する情報がないことを示します。推定ソースが Index で、選択性推定が正しくない場合は、インデックスが偏っていることが問題である可能性があります。REORGANIZE TABLE 文を使用してインデックスの断片化を解除すると改善できる場合があります。

  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、『ASA SQL リファレンス・マニュアル』> 「ESTIMATE_SOURCE 関数 [その他]」を参照してください。

- **キャッシュの読み込み数とヒット数** キャッシュの読み込み数とヒット数がまったく同じ場合は、クエリを実行するのに必要なすべての情報がデータベース全体がキャッシュにあり、良好な状態です。読み込み数がヒット数よりも多い場合は、サーバがキャッシュにアクセスしようとしたが失敗し、ディスクから読み込むことを意味します。これは、ハッシュ・ジョインなどの場合に予想されます。ネスト・ループ・ジョインなどの場合、キャッシュ・ヒット率が低いことは、パフォーマンスの問題を示す場合があり、キャッシュ・サイズを大きくすると改善できる可能性があります。

キャッシュ管理の詳細については、「キャッシュ・サイズの拡大」212 ページを参照してください。

- **適切なインデックスの不足** クエリ実行プランからは、インデックスが良好なパフォーマンスの提供に役立っているかどうかが不明な場合があります。Adaptive Server 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。

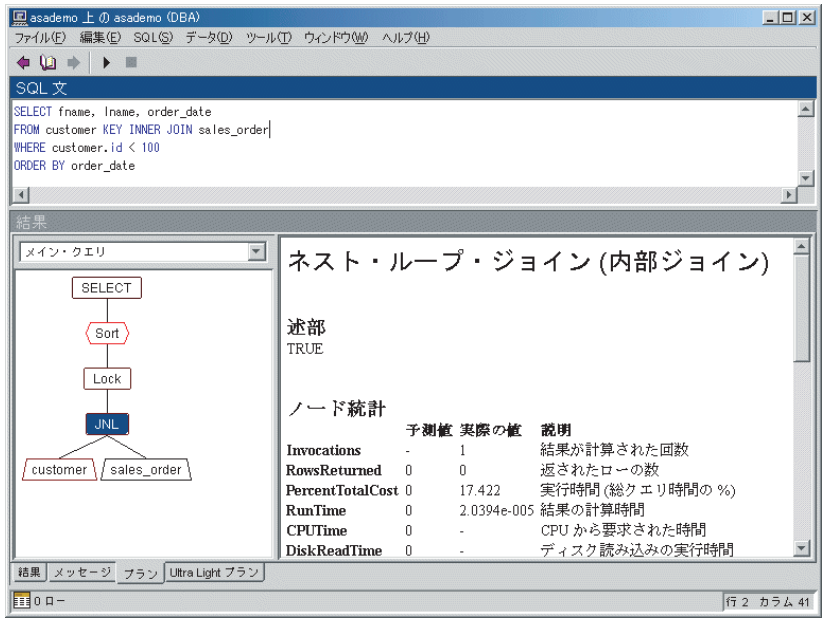
  追加のインデックスがクエリに対して良好なパフォーマンスを提供するのに役立っているか疑わしい場合、インデックス・コンサルタントを使用します。詳細については、「インデックス・コンサルタントの概要」86 ページを参照してください。

  インデックスとパフォーマンスの詳細については、「最高のパフォーマンスを得るヒント」211 ページと「インデックスの使用」213 ページを参照してください。

- **データの断片化の問題** 実行時間の実際値と推定値は、ルート・ノード統計値で提供されます。実行時間は、クエリの実行所要時間を測定します。テーブル・スキャンまたはインデックス・スキャンの実行時間が正しくない場合は、REORGANIZE TABLE 文を実行するとパフォーマンスを向上できることがあります。

  詳細については、『ASA SQL リファレンス・マニュアル』>「REORGANIZE TABLE 文」と「ファイル、テーブル、インデックスの断片化の検査」224 ページを参照してください。

次の例に、統計情報付きのグラフィカルなプランを示します。この例でも、nested loops join ノードが選択されています。右ウィンドウ枠の統計は、クエリのその部分で使用されているリソースを示します。



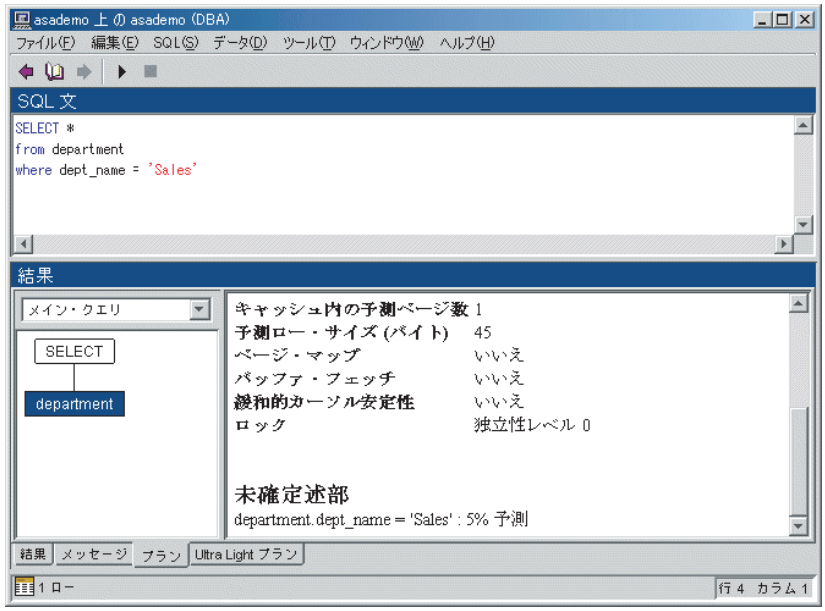
プランに使用されるコード・ワードの詳細については、「プランに使用される省略形」565 ページを参照してください。

プラン内の選択性

次の例では、探索条件の選択性を示す Predicate (述部) を示します。この例では、選択されたノードが department 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と選択性の統計として [述部] が表示されています。

選択性情報は、一部の最適化手順が省略された単純なクエリに対しては表示されない場合があります。単純なクエリの詳細については、「オプティマイザの仕組み」498 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。つまり、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。

```
department.dept_name = 'Sales' : 5% Guess; true 1/5 20%
```

この述部は次のように解釈できます。

- `department.dept_name = 'Sales'` は探索条件です。
- `5%` 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 5% 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、『ASA SQL リファレンス・マニュアル』> 「ESTIMATE 関数 [その他]」を参照してください。
- 推定ソースは「予測」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、『ASA SQL リファレンス・マニュアル』> 「ESTIMATE_SOURCE 関数 [その他]」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 で、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。

プランへのアクセス

Interactive SQL または SQL 関数を使用してアクセス・プランを表示できます。プランは、インデックス・コンサルタントでも表示できます。

Interactive SQL を使用したプランへのアクセス

Interactive SQL では、次の種類のプランを使用できます。

- **短いプラン**
- **長いプラン**
- **グラフィカルなプラン**
- **統計情報付きのグラフィカルなプラン**

表示するプランのタイプを選択するには、[ツール] － [オプション] をクリックし、[プラン] タブをクリックします。

プランを表示するには、クエリを実行して [プラン] タブを開きます。[プラン] タブは、Interactive SQL ウィンドウの最下部にあります。

Ultra Light プラン

クエリの Ultra Light プランも表示できます。Ultra Light プランには、統計は含まれていません。

Interactive SQL で Ultra Light プランを表示するには、[ツール] メニューから [オプション] ダイアログを開き、[Ultra Light プランを表示] を選択します。このオプションはデフォルトで選択されています。上記のタイプ (グラフィカル、短いプラン、または長いプラン) を 1 つ選択して、Ultra Light プランのタイプを制御します。[統計情報付きのグラフィカルなプラン] を選択すると、グラフィカルなプランが表示されます。

[Ultra Light プラン] タブが Interactive SQL ウィンドウの最下部に表示されます。クエリによっては、Adaptive Server Anywhere に選択したプランと Ultra Light の実行プランが異なる場合があります。

詳細については、『ASA SQL リファレンス・マニュアル』> 「GRAPHICAL_ULPLAN 関数 [その他]」を参照してください。

SQL 関数を使用したプランへのアクセス

SQL 関数を使用してプランにアクセスし、出力を XML フォーマットで取り出すことができます。

- 短いプランにアクセスする方法については、『ASA SQL リファレンス・マニュアル』> 「EXPLANATION 関数 [その他]」を参照してください。
- 長いプランにアクセスする方法については、『ASA SQL リファレンス・マニュアル』> 「PLAN 関数 [その他]」を参照してください。
- グラフィカルなプランにアクセスする方法については、『ASA SQL リファレンス・マニュアル』> 「GRAPHICAL_PLAN 関数 [その他]」を参照してください。
- Ultra Light プランにアクセスする方法については、『ASA SQL リファレンス・マニュアル』> 「GRAPHICAL_ULPLAN 関数 [その他]」、『ASA SQL リファレンス・マニュアル』>

「SHORT_ULPLAN 関数 [その他]」、または 『ASA SQL リファレンス・マニュアル』> 「LONG_ULPLAN 関数 [その他]」を参照してください。

第 3 部　SQL のダイアレクトと互換性

第 3 部では、Transact-SQL の互換性と、他の SQL ソフトウェアにはない Adaptive Server Anywhere の機能について説明します。

第 14 章

Transact-SQL との互換性

この章の内容

Transact-SQL は SQL のダイアレクト、つまり、SQL から派生した言語で、Adaptive Server Enterprise がサポートしています。

この章では、Adaptive Server Anywhere と Adaptive Server Enterprise の両方と互換性のあるアプリケーションを作成する方法について説明します。Transact-SQL 言語のさまざまな要素と文、Adaptive Server Enterprise のシステム・テーブル、ビュー、プロシージャに対する、Adaptive Server Anywhere のサポートについて説明します。

Transact-SQL のサポートの概要

Adaptive Server Anywhere は、Sybase Adaptive Server Enterprise がサポートする SQL のダイアレクトである「**Transact-SQL**」の大規模なサブセットをサポートします。この章では、Adaptive Server Anywhere と Adaptive Server Enterprise の間の SQL の互換性について説明します。

目的

次に、Adaptive Server 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 Adaptive Server 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**　Adaptive Server 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 Adaptive Server 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、Adaptive Server Anywhere データベースで実行するには多少の変更が必要です。

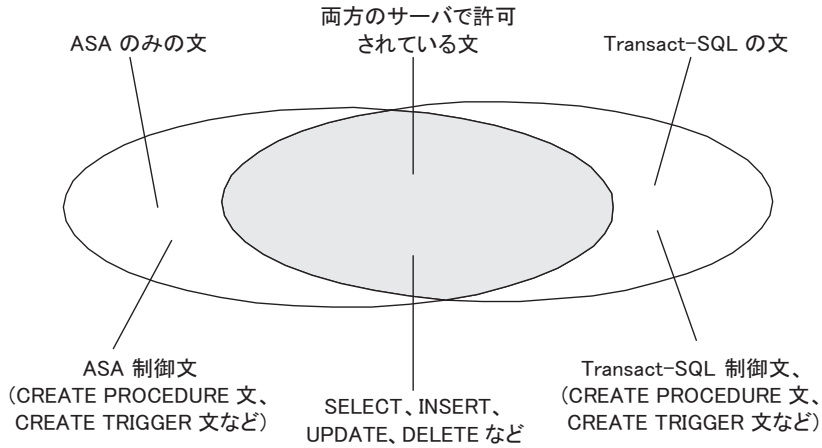
Transact-SQL のサポート方法

次に、Adaptive Server 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、Adaptive Server 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 Adaptive Server Anywhere のバージョンでサポートされる構文とともに、別の Transact-SQL 文がサポートされます。それらの文の場合には、Adaptive Server Anywhere は SQL の 2 種類の「**ダイアレクト**」をサポートします。この章では、それらのダイアレクトを Transact-SQL と Watcom-SQL と呼びます。

- プロシージャ、トリガ、バッチは Transact-SQL または Watcom-SQL ダイアレクトのどちらでも実行されます。バッチまたはプロシージャ中では、1 つのダイアレクトの制御文だけを通して使用します。たとえば、ダイアレクトごとに異なったフロー制御文があります。

2 つのダイアレクトが重なり合う状況を、次の図に示します。



類似点と相違点

Adaptive Server 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、Adaptive Server Anywhere は、すべての数値関数、1 つを除くすべての文字列関数、すべての集合関数、すべての日付と時間関数をサポートします。もう 1 つの例としては、'=*' と '*=' 演算子を使った Transact-SQL 外部ジョインと、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、Adaptive Server 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、Adaptive Server 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、Adaptive Server 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

Adaptive Server 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 Adaptive Server 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。

Adaptive Server 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 Adaptive Server Anywhere バージョン
- CALL 文
- CREATE PROCEDURE 文、CREATE FUNCTION 文、CREATE TRIGGER 文の Adaptive Server 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、Adaptive Server 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 のアーキテクチャ

Adaptive Server Enterprise と Adaptive Server 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。Adaptive Server Anywhere は、管理の必要性が少ないワークグループまたは部署単位のサーバやパーソナル・データベースに最適です。Adaptive Server Enterprise は大規模データベースを持つ企業レベルのサーバに最適です。

この項では、Adaptive Server Enterprise と Adaptive Server 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 Adaptive Server 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 Adaptive Server 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。

master データベースのない Adaptive Server Anywhere

Adaptive Server 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。Adaptive Server 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。

Adaptive Server 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、Adaptive Server Anywhere は、Adaptive Server Enterprise の **sp_addlogin** システム・プロシージャの実行を提供し、同等の処理、つまりデータベースへのユーザの追加を実行します。

Open Server サポートの詳細については、『ASA データベース管理ガイド』> 「Open Server としての Adaptive Server Anywhere」を参照してください。

ファイル操作文

Adaptive Server Anywhere では、Transact-SQL 文の DUMP DATABASE と LOAD DATABASE をサポートしません。Adaptive Server Anywhere には、構文の異なる独自の CREATE DATABASE 文と DROP DATABASE 文があります。

デバイス管理

Adaptive Server 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、Adaptive Server 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。

Adaptive Server 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、『ASA データベース管理ガイド』> 「データベース・ファイルの処理」を参照してください。

デフォルトとルール

Adaptive Server Anywhere は、Transact-SQL の CREATE DEFAULT 文や CREATE RULE 文はサポートしません。CREATE DOMAIN 文を使用すると、デフォルトとルール (いわゆる、CHECK 条件) がドメインの定義に組み込まれるので、Transact-SQL の CREATE DEFAULT 文と CREATE RULE 文に対して、類似した機能性が与えられます。

Adaptive Server Enterprise では、CREATE DEFAULT 文は名前付き「**デフォルト**」を作成します。このデフォルトは、特定のカラムにバインドすることによって、カラムのデフォルト値として使用できます。また、**sp_bindefault** システム・プロシージャを使用してドメインのカラムにバインドすると、ドメインのすべてのカラムのデフォルト値として使用できます。

CREATE RULE 文は、名前のついた「**ルール**」を作成します。このルールは、特定のカラムにバインドすることによって、さまざまなカラムのドメイン定義に使用でき、データ型にバインドすると、そのドメインのすべてのカラムのルールとして使用できます。ルールをデータ型やカラムにバインドするには、**sp_bindrule** システム・プロシージャを使用します。

Adaptive Server Anywhere では、ドメインはデフォルト値とそれに対応した CHECK 条件を持つことができ、その CHECK 条件はそのデータ型に基づいて定義されたすべてのカラムに適用されます。ドメインを作成するには、CREATE DOMAIN 文を使用します。

デフォルト値とルール、または CHECK 条件は、個々のカラムについて、CREATE TABLE 文または ALTER TABLE 文を使用して定義できます。

これらの文の Adaptive Server Anywhere 構文の詳細については、『ASA SQL リファレンス・マニュアル』> 「 SQL 文」を参照してください。

システム・テーブル

Adaptive Server Anywhere には、独自のシステム・テーブルの他に、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。これらのビューについては、2 つ

の製品のシステム・カタログを説明する『ASA SQL リファレンス・マニュアル』> 「Transact-SQL 互換のビュー」で個別に説明されています。この項では相違点の概要について説明します。

Adaptive Server 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。Adaptive Server 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 ID **dbo**) がシステム・テーブルを所有します。Adaptive Server Anywhere では、システム所有者 (ユーザ ID **SYS**) がシステム・テーブルを所有します。ユーザ ID **dbo** は、Adaptive Server 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 Adaptive Server 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ほか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ほか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、Adaptive Server 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、ほかのユーザにデータベース内でのオブジェクト作成とコマンド実行のパーミッションを付与できます。デフォルトのデータベース管理者は、ユーザ ID **DBA** です。

- RESOURCE パーミッションは、ユーザがデータベース内でオブジェクトを作成するのを許可します。これは Adaptive Server Enterprise で個々の CREATE 文にパーミッションを付与する代わりのものです。

- Adaptive Server 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 Adaptive Server 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 (Adaptive Server Anywhere では RESOURCE、Adaptive Server Enterprise では CREATE 文ごとのパーミッション) を持つユーザ ID をデータベースに作成し、このユーザ ID からオブジェクトを作成してください。同じユーザ ID を両方の環境で使用すると、オブジェクト名と修飾子が 2 つのデータベースで同一になり、互換性のあるアクセスが可能になります。

ユーザとグループ

Adaptive Server Enterprise と Adaptive Server Anywhere では、ユーザとグループのモデルに違いがあります。

Adaptive Server Enterprise では、サーバに接続する各ユーザは、サーバに対するログイン ID とパスワードと、そのサーバ上でアクセスする各データベースに対するユーザ ID が必要です。データベースの各ユーザは、1 つのグループにしか属せません。

Adaptive Server 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。

いずれのサーバもユーザ・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、Adaptive Server 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 Adaptive Server 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、Adaptive Server Anywhere がサポートする、Adaptive Server Enterprise のユーザとグループ管理のシステム・プロシージャを示します。

各プロシージャの引数については、『ASA SQL リファレンス・マニュアル』> 「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」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。Adaptive Server Anywhere では、ユーザをデータベースに追加する。
sp_adduser	Adaptive Server Enterprise と Adaptive Server 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** とは異なるタスクだが、Adaptive Server Anywhere では同じである。
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。Adaptive Server Anywhere では、ユーザをデータベースから削除する。
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。Adaptive Server 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 Adaptive Server Anywhere で非常に似ています。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 Adaptive Server Anywhere の両方で有効です。

```
GRANT INSERT, DELETE
 ON TITLES
 TO MARY, SALES
```

この文は、**TITLES** 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。

Adaptive Server 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ほかのユーザに付与することを許可します。ただし、Adaptive Server Anywhere は WITH GRANT OPTION を GRANT EXECUTE 文で使用することを許可しません。

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 Adaptive Server Anywhere は、データベース全体に適用されるユーザ・パーミッションに異なったモデルを使用します。これらのモデルについては、「ユーザとグループ」599 ページで説明します。Adaptive Server 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、Adaptive Server 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、ほか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。

Adaptive Server 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、Adaptive Server Anywhere と Adaptive Server Enterprise の相違点のいくつかを解除できます。ほかの相違点は、Adaptive Server 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] － [Adaptive Server Anywhere 9] － [データベースの作成] を選択します。
3. ウィザードの指示に従います。
4. [Adaptive Server Enterprise のエミュレート] ボタンが表示されたら、このボタンをクリックし、[次へ] をクリックします。
5. ウィザードの指示に従います。

❖ Transact-SQL と互換性のあるデータベースを作成するには、次の手順に従います (コマンド・ラインの場合)。

- コマンド・プロンプトで次のコマンドを入力します。

```
dbinit -b -c -k db-name.db
```

これらのオプションの詳細については、『ASA データベース管理ガイド』> 「初期化ユーティリティのオプション」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. Adaptive Server 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 'db-name.db'
 ASE COMPATIBLE
```

この文で、ASE COMPATIBLE は Adaptive Server Enterprise と互換性があるという意味です。

大文字と小文字を区別するデータベースの作成

デフォルトでは、Adaptive Server Enterprise データベースでの文字列比較は大文字と小文字を区別しますが、Adaptive Server Anywhere では大文字と小文字を区別しません。

Adaptive Server 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* コマンド・ライン・ユーティリティを使用している場合は、`-c` コマンド・ライン・スイッチを指定します。

比較の後続ブランクを無視

Adaptive Server 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* コマンド・ライン・ユーティリティを使用している場合は、`-b` コマンド・ライン・スイッチを指定します。

このオプションを選択すると、Adaptive Server Enterprise と Adaptive Server Anywhere では次の 2 つの文字列を等しいとみなします。

```
'ignore the trailing blanks   '
 'ignore the trailing blanks'
```

このオプションを選択しないと、Adaptive Server Anywhere ではこの 2 つの文字列は異なっているとみなします。

このオプションを選択すると、クライアント・アプリケーションでフェッチした文字列にブランクが埋め込まれます。

古いシステム・ビューの削除

Adaptive Server 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。これには、*dbinit* `-k` コマンド・ライン・スイッチを使用します。

データベース作成時にこのオプションを使用しない場合、次の 2 つの文は異なった結果を返します。

```
SELECT * FROM SYSCOLUMNS ;

SELECT * FROM dbo.syscolumns ;
```

❖ 既存のデータベースから Adaptive Server Anywhere のシステム・ビューを削除するには、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. 次の文を実行します。

```
DROP VIEW SYS.SYSCOLUMNS ;
DROP VIEW SYS.SYSINDEXES
```

警告
dbo.syscolumns や dbo.sysindexes のシステム・ビューは削除しないでください。

Transact-SQL との互換性を維持するためのオプション設定

Adaptive Server 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 を許可すると宣言しないかぎり、新しいカラムに NULL を許可しません。Adaptive Server 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/92 ISO 規格と互換性があります。

Adaptive Server Enterprise を SQL/92 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。

Adaptive Server 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/92 ISO 規格に一致する Adaptive Server Anywhere とは異なります。

quoted_identifier オプションは、Adaptive Server Enterprise と Adaptive Server Anywhere の両方で使用できます。識別子と文字列が互換性を持って処理されるように、両方のデータベースでこのオプションが同じ値に設定されていることを確認します。

SQL/92 に準拠させるには、Adaptive Server Enterprise と Adaptive Server 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 Adaptive Server 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 オプションの詳細については、『ASA データベース管理ガイド』> 「QUOTED_IDENTIFIER オプション [互換性]」を参照してください。

automatic_timestamp オプションの ON への設定

Transact-SQL では、**timestamp** カラムは特殊なプロパティで定義されています。**automatic_timestamp** オプションが ON に設定されていると、Adaptive Server Anywhere の **timestamp** カラムの処理は Adaptive Server Enterprise の処理になります。

Adaptive Server Anywhere で **automatic_timestamp** オプションが ON (デフォルトは OFF) に設定されていて、TIMESTAMP データ型を持つ新しいカラムに特にデフォルト値が設定されていないときは、デフォルト値に **timestamp** が与えられます。

timestamp カラムについては、「Transact-SQL の特殊な timestamp カラムとデータ型」609 ページを参照してください。

string_rtruncation オプションの設定

Adaptive Server Enterprise と Adaptive Server Anywhere では、string_rtruncation オプションがサポートされています。このオプションは、INSERT 文または UPDATE 文字列がトランケートされたときのエラー・メッセージの表示を制御します。各データベースのオプション設定は同じ値にしてください。

STRING_RTRUNCATION オプションの詳細については、『ASA データベース管理ガイド』> 「STRING_RTRUNCATION オプション [互換性]」を参照してください。

Transact-SQL と互換性のあるデータベース・オプションの詳細については、『ASA データベース管理ガイド』> 「互換性オプション」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**　パスワードの大文字と小文字を区別するかしないかは、ほかの識別子とは取り扱い方が違います。

データの大文字と小文字の区別

Adaptive Server 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。Adaptive Server 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

Adaptive Server 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。ユーザ ID はほかの識別子と同様に取り扱われ、常に大文字と小文字が区別されません。データベースで使用するデフォルトのユーザ ID は、大文字の **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。Adaptive Server 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

Adaptive Server Anywhere では、パスワード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とは別に指定されます。デフォルトでは、パスワードで大文字と小文字を区別するかどうかを指定しない場合、パスワードはデータベースの大文字と小文字の区別の設定に従います。ただし、大文字と小文字の区別がないデータベースにおいて、パスワードで大文字と小文字を区別するように指定でき、また大文字と小文字を区別するデータベースにおいて、パスワードで大文字と小文字を区別しないように指定できます。パスワードで使用される

拡張文字は、パスワードの大文字小文字の区別設定にかかわらず、大文字と小文字を区別します。データベースで使用するデフォルトのパスワードは、大文字の **SQL** です。

パスワードの大文字と小文字の区別の設定については、『ASA SQL リファレンス・マニュアル』> 「CREATE DATABASE 文」の PASSWORD CASE 句を参照してください。

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 Adaptive Server Anywhere で異なるネーム・スペースを持っています。

Adaptive Server Anywhere ではインデックスとトリガは、それらが作成されたテーブルの所有者に属します。ある所有者に関して、インデックスとトリガの名前はユニークでなければなりません。たとえば、ユーザ user1 が所有するテーブル t1 とユーザ user2 が所有するテーブル t2 は同じ名前のインデックスを持つことができますが、1 人のユーザが所有する 2 つのテーブルでは、インデックス名を同じものにはできません。

Adaptive Server Enterprise は Adaptive Server Anywhere よりも制限の少ないネーム・スペースを持っています。特定のテーブルではインデックス名はユニークである必要がありますが、どの 2 つのテーブルも同一のインデックス名を持つことができます。互換性のある SQL を作成するには、Adaptive Server Anywhere の、テーブルの所有者に対してユニーク・インデックス名を必要とする制限を採用します。

Adaptive Server Enterprise は、トリガ名について Adaptive Server 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

Adaptive Server 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
Adaptive Server 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。

Adaptive Server 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、(Adaptive Server 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 table_name (
     column_1 INTEGER ,
     column_2 TIMESTAMP DEFAULT TIMESTAMP
 )
```

次の ALTER TABLE 文は、sales_order テーブルに Transact-SQL timestamp カラムを追加します。

```
ALTER TABLE sales_order
 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。Adaptive Server Anywhere ではデータ型は自分で指定します。

AUTOMATIC_TIMESTAMP データベース・オプションが ON に設定されている場合、デフォルト値を設定する必要はありません。TIMESTAMP データ型で作成し、明示的なデフォルト値がない新しいカラムには、デフォルト値 timestamp が与えられます。次の文は AUTOMATIC_TIMESTAMP を ON に設定します。

```
SET OPTION PUBLIC.AUTOMATIC_TIMESTAMP='ON'
```

timestamp カラムのデータ型

Adaptive Server Enterprise では、timestamp カラムはドメインの NULL を許容する VARBINARY(8) として扱われます。Adaptive Server 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。

既存のテーブルへのタイムスタンプの設定

既存のテーブルに特殊な timestamp カラムを追加すると、既存のローはすべて timestamp カラムに NULL 値を持ちます。既存のローのために timestamp 値 (現在の timestamp) を入力するには、データが変わらないようにテーブルのすべてのローを更新します。たとえば、次の文は sales_order テーブルのすべてのローを更新し、どのローの値も変更しません。

```
UPDATE sales_order
 SET region = region
```

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, '1995/10/25 11:08:34.173226')
```

tsequal 関数の最初の引数は、特殊な timestamp カラムの名前です。2 番目の引数は SELECT 文で取得した timestamp です。Embedded SQL では、2 番目の引数は、カラムで最後に FETCH から取得した TIMESTAMP を含むホスト変数であることが多くなります。

特殊な IDENTITY カラム

IDENTITY カラムを作成するには、次に示すように CREATE TABLE 構文を使います。

```
CREATE TABLE table-name (
      ...
      column-name numeric(n,0) IDENTITY NOT NULL,
      ...
 )
```

n はテーブルに挿入されるローの最大数よりも大きい数です。

IDENTITY カラムは、送り状番号や従業員番号のような連続番号を格納します。このカラムは、自動生成されます。IDENTITY カラムの値はテーブルの各ローをユニークに識別します。

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。

Adaptive Server 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。Adaptive Server 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。Adaptive Server Anywhere では、IDENTITY カラムにどの数値データ型でも使用できます。

Adaptive Server Anywhere では、identity カラムとカラムのデフォルト設定の AUTOINCREMENT は同じです。

@@identity を使用する IDENTITY カラム値の検索

初めてローを 1 つテーブルに挿入すると、IDENTITY カラムに 1 という値が割り当てられます。その後は挿入するたびに、カラムの値が 1 ずつ増えます。最後に IDENTITY カラムに挿入した値は、@@identity グローバル変数として使用できます。

@@identity の値は、文がテーブルにローを挿入するたびに変わります。

- 文が IDENTITY カラムのないテーブルに影響を及ぼすと、@@identity は 0 に設定されます。
- 文が複数のローを挿入すると、@@identity は IDENTITY カラムに最後に挿入された値を反映します。

この変更は確定的です。文にエラーが発生したり、文を含むトランザクションがロールバックされたりしても、@@identity は前の値に復元されません。

@@identity の動作の詳細については、『ASA SQL リファレンス・マニュアル』> 「@@identity グローバル変数」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 Adaptive Server Anywhere 間の互換性に関する問題についても説明します。

移植可能な SQL を記述するための一般的なガイドライン

複数のデータベース管理システムで使用するために SQL を記述する場合は、SQL 文をできるかぎり明示的にします。指定した SQL 文を複数のサーバがサポートしている場合でも、デフォルトの動作が各システムで同じであると仮定するのは間違っていることもあります。次に、互換性のある SQL を記述するときの一般的なガイドラインを示します。

- デフォルトの動作を使用しないで、使用可能なオプションをすべて略さずに書きます。
- 演算子の優先順位のデフォルトが同じであるとするのではなく、文中の実行の順序をカッコを使用して明確にします。
- Adaptive Server Enterprise に移植できるように、変数名には @ をプレフィクスとして付ける Transact-SQL の規則に従います。
- プロシージャ、トリガ、バッチでは、BEGIN 文の直後で変数とカーソルを宣言します。Adaptive Server Enterprise では、プロシージャ、トリガ、バッチ内のどこででも宣言できますが、Adaptive Server 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 Adaptive Server 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

Adaptive Server 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、名前付き制約または名前付きデフォルトはサポートしません。

NULL

Adaptive Server 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。Adaptive Server 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、Adaptive Server 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」605 ページを参照してください。

テンポラリ・テーブル

テンポラリ・テーブルを作成するには、CREATE TABLE 文の先頭にシャープ記号 (#) を付けます。このようなテンポラリ・テーブルは、Adaptive Server 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。Adaptive Server Anywhere での宣言テンポラリ・テーブルについては、『ASA SQL リファレンス・マニュアル』>「DECLARE LOCAL TEMPORARY TABLE 文」を参照してください。

テーブルの物理的配置は、Adaptive Server Enterprise と Adaptive Server Anywhere では実行方法が異なります。Adaptive Server Anywhere は **ON** *segment-name* 句をサポートしますが、*segment-name* は Adaptive Server Anywhere の DB 領域を参照します。

CREATE TABLE 文については、『ASA SQL リファレンス・マニュアル』>「CREATE TABLE 文」を参照してください。

互換性のあるクエリの記述方法

Adaptive Server 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、Adaptive Server 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。

Adaptive Server 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 T_string*

table-spec:
[*owner* .]*table-name*
…[[**AS**] *correlation-name*]
…[**(** **INDEX** *index_name* [**PREFETCH** *size*][**LRU** | **MRU**] **)**]

alias-name:*identifier* :
| '*string*' | "*string*"

SELECT 文の詳細については、『ASA SQL リファレンス・マニュアル』> 「SELECT 文」を参照してください。

Adaptive Server 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
- GROUP BY ALL 句

注意

- INTO table_name 句は、SELECT 文の結果セットに基づいて新しいテーブルを作成しますが、テーブル名がシャープ記号 (#) で始まるテンポラリ・テーブルを宣言するためだけに使用できます。宣言テンポラリ・テーブルは 1 つの接続だけのために存在します。
- Adaptive Server 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。
- FOR READ ONLY 句と FOR UPDATE 句は解析されますが、効果はありません。
- 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 Adaptive Server Anywhere によってサポートされます。このキーワードは、指定したテーブルまたはビューの共有ロックを (トランザクションが完了したかどうかに関わらず、要求されたデータ・ページが必要なくなると同時に共有ロックを解放しないで) トランザクションの完了まで保持することによって、さらに制限することができます。HOLDLOCK が指定されているテーブルのために、クエリは独立性レベル 3 で実行されます。

- HOLDLOCK オプションは、それが指定されたテーブルまたはビューにだけ、しかも、そのオプションが使用された文で定義されたトランザクションの間だけ適用されます。独立性レベルを 3 に設定すると、トランザクション内の各 SELECT に適用されます。1 つのクエリの中で HOLDLOCK と NOHOLDLOCK の両方のオプションは指定できません。
- NOHOLDLOCK キーワードは、Adaptive Server 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
  ```

  Adaptive Server Anywhere でこれに対応する文は、SET 文です。

  ```
  SET localvar = 42
  ```

  SET 文と名前の前に @ をプレフィクスとして付ける Transact-SQL の規則を使用して、変数名をオプションで設定できます。

  ```
  SET @localvar = 42
  ```

- Adaptive Server Enterprise は、次に示す SELECT 構文の句をサポートしません。
  - INTO ホスト変数リスト
  - INTO 変数リスト
  - カッコ付きのクエリ
- Adaptive Server Enterprise は、ジョイン用の FROM 句と ON 条件ではなく、WHERE 句でジョイン演算子を使用します。

ジョインの互換性

Transact-SQL では、次に示す構文を使用して、ジョインを WHERE 句に指定します。

start of select, update, insert, delete, or subquery
FROM { *table-list* | *view-list* } **WHERE** [**NOT**]
[*table-name*.| *view name*.]*column-name*
join-operator
[*table-name*.| *view-name*.]*column_name*
[{ **AND** | **OR** } [**NOT**]
[*table-name*.| *view-name*.]*column_name*
join-operator
[*table-name*.| *view-name*.]*column-name*]…
end of select, update, insert, delete, or subquery

WHERE 句の *join-operator* は、比較演算子の場合もあれば、次に示す「**外部ジョイン演算子**」のいずれかの場合もあります。

- *= 左外部ジョイン演算子
- =* 右外部ジョイン演算子

Adaptive Server Anywhere は、Transact-SQL 外部ジョイン演算子をネイティブの SQL/92 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。

Adaptive Server Anywhere は、ジョインが WHERE 句ではなく FROM 句に入っている、外部ジョイン以外のジョイン向けの SQL/92 構文も提供します。

Adaptive Server Anywhere と ANSI ／ ISO SQL 規格におけるジョインの詳細については、「ジョイン：複数テーブルからのデータ検索」335 ページと 『ASA SQL リファレンス・マニュアル』> 「FROM 句」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 =*)」357 ページ を参照してください。

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。

Adaptive Server Anywhere は、SQL/92 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

Adaptive Server Anywhere ストアド・プロシージャ言語は、ISO/ANSI 規格を基準にしており、Transact-SQL ダイアレクトとは多くの点で異なっています。概念と機能は似ていますが、構文が異なります。概念が似ているため、Transact-SQL の Adaptive Server 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。

Adaptive Server 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 Adaptive Server 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 Adaptive Server Anywhere トリガの機能の互換性の概要について説明します。

Adaptive Server Enterprise のトリガは、トリガを起動する文が完了してから実行されます。すなわち、**文レベル**の**完了後**トリガです。Adaptive Server Anywhere は、ローが変更される前または後に実行される**ロー・レベル**・トリガと、文全体の後に実行される文レベル・トリガの両方をサポートします。

ロー・レベルのトリガは、Transact-SQL 互換性機能の一部ではありません。詳細については、「プロシージャ、トリガ、バッチの使用」811 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、Adaptive Server Anywhere ではサポートされない機能、または、Adaptive Server Anywhere では異なる機能を次に示します。

- **ほか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 Adaptive Server 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、ほか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。Adaptive Server Anywhere では、メモリが不足していないかぎり、トリガは無制限にほか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 Adaptive Server Anywhere と Adaptive Server Enterprise の対応は、やや異なります。Adaptive Server Anywhere では、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 の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。

  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。これを行うには、**self_recursion** オプションを ON にします。

- **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。Adaptive Server Anywhere は、トリガで ROLLBACK (または ROLLBACK TRANSACTION) 文を許可しません。トリガする動作とそのトリガがともにアトミック・ステートメントを構成するためです。

  Adaptive Server 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。『ASA SQL リファレンス・マニュアル』> 「ROLLBACK TRIGGER 文」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。Adaptive Server Anywhere の Interactive SQL ユーティリティと Adaptive Server Enterprise の *isql* 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。Adaptive Server 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、Adaptive Server Anywhere はコマンド・ファイルにあるパラメータの使用をサポートします。Adaptive Server Enterprise は、パラメータをサポートしません。

パラメータについては、『ASA SQL リファレンス・マニュアル』> 「PARAMETERS 文 [Interactive SQL]」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、Adaptive Server 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。次に例を示します。

```
select SqlDialect('select * from employee')
```

この文は、Watcom-SQL の値を返します。

Sybase Central を使用するストアド・プロシージャの変換

Sybase Central は、プロシージャとトリガの作成、表示、変更を行うための機能を備えています。

❖ **Sybase Central を使ってストアド・プロシージャを変換するには、次の手順に従います。**

1. 変更するプロシージャの所有者または DBA ユーザとして、Sybase Central からデータベースに接続します。
2. [プロシージャとファンクション] フォルダを開きます。
3. 変換するプロシージャを右クリックし、ポップアップ・メニューで、使用するダイアレクトに合った「～に変換」コマンドを 1 つ選択します。

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。

4 必要に応じて変換されなかった行を書き直します。

5 終了したら、[ファイル] － [プロシージャの保存] 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

Adaptive Server Anywhere は、返される結果セットを指定するのに RESULT 句を使います。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 showdept (@deptname varchar(30))
 AS
     SELECT employee.emp_lname, employee.emp_fname
     FROM department, employee
     WHERE department.dept_name = @deptname
     AND department.dept_id = employee.dept_id
```

Watcom-SQL プロシージャの例

次に、これに対応する Adaptive Server Anywhere プロシージャを示します。

```
CREATE PROCEDURE showdept(in deptname varchar(30))
 RESULT ( lastname char(20), firstname char(20))
 BEGIN
     SELECT employee.emp_lname, employee.emp_fname
     FROM department, employee
     WHERE department.dept_name = deptname
     AND department.dept_id = employee.dept_id
 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」849 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

Adaptive Server Anywhere は SET 文を使用して、プロシージャ中の変数に値を割り当てます。Transact-SQL では、空のテーブル・リストの SELECT 文、または SET 文を使用して値を割り当てます。次のプロシージャは、Transact-SQL 構文の働きを示します。

```
CREATE PROCEDURE multiply
                     @mult1 int,
                     @mult2 int,
                     @result int output
 AS
 SELECT @result = @mult1 * @mult2
```

このプロシージャを呼び出すには、次のようにします。

```
CREATE VARIABLE @product int
 go
 EXECUTE multiply 5, 6, @product OUTPUT
 go
```

変数 **@product** の値は、プロシージャの実行後 30 になります。

SELECT 文の使用による変数割り当ての詳細については、「互換性のあるクエリの記述方法」615 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、『ASA SQL リファレンス・マニュアル』>「SET 文」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	意味
0	プロシージャはエラーを起こすことなく実行された
-1	オブジェクトが見つからない
-2	データ型エラー
-3	プロセスはデッドロックの対象となった

値	意味
-4	パーミッション・エラー
-5	構文エラー
-6	その他のユーザ・エラー
-7	リソース・エラー、たとえば領域が足りない
-8	致命的でない内部の問題
-9	システムの限界に達した
-10	致命的な内部の矛盾
-11	致命的な内部の矛盾
-12	テーブルまたはインデックスが壊れている
-13	データベースが壊れている
-14	ハードウェア・エラー

RETURN 文は、この表の値以外の、ユーザが意味を定義した整数値を返すこともできます。

プロシージャの中での RAISERROR 文の使用

RAISERROR 文は、ユーザ定義エラーを生成するための Transact-SQL 文です。これは、SIGNAL 文と似た機能を持っています。

RAISERROR 文の詳細については、『ASA SQL リファレンス・マニュアル』> 「RAISERROR 文 [T-SQL]」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ほかのプロシージャから呼び出された場合、最も外側のプロシージャが終了してから RAISERROR が返されます。

中間レベルの RAISERROR ステータスとコードは、プロシージャが終了すると失われます。結果が返されるときに RAISERROR とともにエラーが発生した場合は、エラー情報が返され、RAISERROR 情報は失われます。アプリケーションでは、別の実行ポイントでグローバル変数 **@@error** を検査して、中間の RAISERROR ステータスを問い合わせることができます。

Watcom-SQL での Transact-SQL のようなエラー処理

CREATE PROCEDURE 文に ON EXCEPTION RESUME 句を追加して、Watcom-SQL のプロシージャが Transact-SQL に似た方法でエラー処理を行うようにすることができます。

```
CREATE PROCEDURE sample_proc()
 ON EXCEPTION RESUME
 BEGIN
     ...
 END
```

ON EXCEPTION RESUME 句があると、明示的な例外処理コードは実行されません。このため、これらの 2 つの句は一緒に使用しないようにしてください。

第 15 章

他の SQL ダイアレクトとの違い

この章の内容

Adaptive Server 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、ISO/ANSI SQL-92 規格に初級レベルで準拠しており、若干の例外がありますが SQL-99 の中核となる仕様にも準拠しています。

準拠に関する包括的で詳細な情報については、Adaptive Server 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。

この章では、他の SQL ソフトウェアにはない、Adaptive Server Anywhere の機能について説明します。

Adaptive Server Anywhere SQL の機能

次の Adaptive Server 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。

データ型の変換

データ型の変換は、完全実装されています。どのデータ型も、他の任意の型と比較することができ、また他の任意の型と一緒に式の中で使えます。

日付

Adaptive Server 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、『ASA SQL リファレンス・マニュアル』> 「SQL 関数」を参照してください。

整合性

Adaptive Server 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。Adaptive Server 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プライマリ・キーにあるローの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、『ASA SQL リファレンス・マニュアル』> 「CREATE TABLE 文」を参照してください。

ジョイン

Adaptive Server 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、Adaptive Server 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

Adaptive Server 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
 MODIFY column data-type
 DELETE column
 RENAME new-table-name
 RENAME old-column TO new-column
```

MODIFY は、文字カラムの最大長の変更とデータ型の変更に使用できます。詳細については、『ASA SQL リファレンス・マニュアル』> 「ALTER TABLE 文」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

Adaptive Server 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 Adaptive Server 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    emp_lname,
             birth_date,
             (    SELECT dept_name
                 FROM department
                 WHERE emp_id = employee.emp_ID
                 AND dept_id = 200 )
 FROM employee
```

その他の関数

Adaptive Server 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、『ASA SQL リファレンス・マニュアル』> 「SQL 関数」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。

第 4 部　データベースにおける XML

第 4 部では、データベースで XML を使用する方法について説明します。

第 16 章

データベースにおける XML の使用

この章の内容

この章では、XML データのインポート、エクスポート、格納、クエリなど、Adaptive Server Anywhere における XML サポートの概要について説明します。

XML とは

Extensible Markup Language (XML) は、構造化データをテキスト形式で表します。XML は、大規模な電子出版の課題を満たすために設計されました。

XML は、HTML のように単純なマークアップ言語ですが、SGML のように柔軟性があります。XML は、階層型で、その主な目的は、人間とコンピュータの両方が作成し、読むことのできるデータの構造を記述することです。

XML では、さまざまな形式のデータを記述する、一連の静的な要素は提供されておらず、ユーザが要素を定義できます。そのため、XML を使用して多くの種類の構造化データを記述できます。XML 文書では、オプションとして文書型定義 (DTD) または XML スキーマを使用し、XML ファイルで使用される構造、要素、属性を定義できます。

XML の詳細については、http://www.w3.org/XML/ を参照してください。

XML と Adaptive Server Anywhere

Adaptive Server 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

リレーショナル・データベースにおける XML 文書の格納

Adaptive Server 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 データ型を使用する利点

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、次の内容を持つ <product> という要素を生成するとします。

```
<hat>bowler</hat>
```

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。次のように、この要素の内容を XML 型として指定するクエリを記述した場合、

```
SELECT XMLFOREST( CAST( '<hat>bowler</hat>' AS XML ) AS
product )
```

不等号はマークアップされず、次のような結果が得られます。

```
<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、

```
SELECT XMLFOREST( '<hat>bowler</hat>' AS product )
```

次のように、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、「無効なカラム名」652 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、『ASA SQL リファレンス・マニュアル』> 「XML データ型 [文字]」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

Adaptive Server 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、『ASA SQL リファレンス・マニュアル』> 「OUTPUT 文 [Interactive SQL]」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 関数または 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」641 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。Adaptive Server 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、『ASA プログラミング・ガイド』> 「AsaCommand オブジェクトを使用したローの挿入、更新、削除」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

Adaptive Server 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 関数を使用して、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 を使用すると、XML 文書から結果セットを生成できます。OPENXML は、XPath クエリ言語のサブセットを使用して、XML 文書からノードを選択します。

XPath 式の使用

OPENXML を使用すると、XML 文書が解析され、結果はツリーとしてモデル化されます。このツリーはノードで構成されています。XPath 式は、ツリー内のノードを選択するために使用されます。次のリストは、一般的に使用される XPath 式の一部を示しています。

- / XML 文書のルート・ノードを示します。
- **. (単一のピリオド)** XML 文書のカレント・ノードを示します。
- // カレント・ノードのすべての子孫を示します。カレント・ノードも含まれます。
- **..** カレント・ノードの親ノードを示します。
- ***./@attributename*** *attributename* という名前を持つ、カレント・ノードの属性を示します。
- ***./childname*** カレント・ノードの子で、*childname* という名前を持つ要素を示します。

次の XML 文書を考えてみます。

```
<inventory>
  <product id="301" size="Medium">Tee Shirt
   <quantity>54</quantity>
  </product>
  <product id="302" size="One size fits all">Tee Shirt
   <quantity>75</quantity>
  </product>
  <product id="400" size="One size fits all">Baseball
Cap
   <quantity>112</quantity>
  </product>
 </inventory>
```

<inventory> 要素は、ルート・ノードです。この要素は、次の XPath 式を使用して参照できます。

```
/inventory
```

カレント・ノードが <quantity> 要素であると仮定します。このノードは、次の XPath 式を使用して参照できます。

```
.
```

<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、『ASA SQL リファレンス・マニュアル』> 「OPENXML 関数 [文字列]」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、『ASA SQL リファレンス・マニュアル』>「OPENXML 関数 [文字列]」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 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、employee テーブル、sales_order テーブル、customer 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、『ASA SQL リファレンス・マニュアル』> 「XMLELEMENT 関数 [文字列]」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」657 ページを参照してください。

```
CREATE VARIABLE x XML;
 SET x=(SELECT XMLELEMENT( NAME root,
          (SELECT * FROM employee
          KEY JOIN sales_order
          KEY JOIN customer
          FOR XML AUTO, ELEMENTS)));
```

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>
  <employee>
   <emp_id>299</emp_id>
   <manager_id>902</manager_id>
   <emp_fname>Rollin</emp_fname>
   <emp_lname>Overbey</emp_lname>
   <dept_id>200</dept_id>
   <street>191 Companion Ct.</street>

  <city>Kanata</city>
   <state>CA</state>
   <zip_code>94608</zip_code>
   <phone>5105557255</phone>
   <status>A</status>
   <ss_number>025487133</ss_number>
   <salary>39300.000</salary>
   <start_date>1987-02-19</start_date>
```

```
  <birth_date>1964-03-15</birth_date>
   <bene_health_ins>Y</bene_health_ins>
   <bene_life_ins>Y</bene_life_ins>
   <bene_day_care>N</bene_day_care>
   <sex>M</sex>
   <sales_order>
    <id>2001</id>
    <cust_id>101</cust_id>

   <order_date>2000-03-16</order_date>
    <fin_code_id>r1</fin_code_id>
    <region>Eastern</region>
    <sales_rep>299</sales_rep>
    <customer>
     <id>101</id>
     <fname>Michael</fname>
     <lname>Devlin</lname>
     <address>114 Pioneer Avenue</address>

    <city>Kingston</city>
     <state>NJ</state>
     <zip>07070</zip>
     <phone>2015558966</phone>
     <company_name>The Power Group</company_name>
    </customer>
   </sales_order>
 </employee>
 ...
```

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。

```
SELECT * FROM OPENXML( x, '//*' )
  WITH (id INT '@mp:id',
        parent INT '../@mp:id',
        name CHAR(20) '@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)
23	15	emp_id	299
47	15	manager_id	902
74	15	emp_fname	Rollin
...	...	...	...

カラム内の XML の問い合わせ

XML を含むカラムを持つテーブルがある場合、OPENXML を使用して、カラム内のすべての XML 値を一度に問い合わせできます。これには、ラテラル抽出テーブルを使用します。

次の文は、manager_id 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、employee テーブルから生成された XML データが含まれます。

```
CREATE TABLE t (manager_id INT, reports XML);
 INSERT INTO t
 SELECT manager_id, XMLELEMENT( NAME reports,
                     XMLAGG(
                      XMLELEMENT( NAME e, emp_id)))
 FROM employee
 GROUP BY manager_id;
```

次のクエリを実行して、t テーブル内のデータを表示してください。

```
SELECT * FROM t;
```

このクエリは、次の結果を生成します。

manager_id	reports
1293	<reports> <e>148</e> <e>390</e> <e>586</e> <e>757</e> ... </reports>
1576	<reports> <e>184</e> <e>207</e> <e>318</e> <e>409</e> ... </reports>
902	<reports> <e>129</e> <e>195</e> <e>299</e> <e>467</e> ... </reports>
703	<reports> <e>191</e> <e>750</e> <e>868</e> <e>921</e> ... </reports>
...	...

次のクエリは、ラテラル抽出テーブルを使用して、2 つのカラムを持つ結果セットを生成しています。1 つのカラムは、各マネージャの ID をリストします。もう 1 つのカラムは、そのマネージャに報告を行う各従業員の ID をリストします。

```
SELECT manager_id, eid
 FROM t, LATERAL( OPENXML( t.reports, '//e' )
 WITH (eid INT '.') ) dt
```

このクエリは、次の結果を生成します。

manager_id	eid
1293	148
1293	390
1293	586
1293	757
...	...

ラテラル抽出テーブルの詳細については、『ASA SQL リファレンス・マニュアル』> 「FROM 句」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、Adaptive Server 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。

Adaptive Server Anywhere .NET データ・プロバイダを使用して、DataSet を元に XML 文書からデータかスキーマまたはその両方を読み込む方法については、『ASA プログラミング・ガイド』> 「AsaDataAdapter オブジェクトを使用したデータの取得」を参照してください。

クエリ結果を XML として取得する

Adaptive Server 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」650 ページと 『ASA SQL リファレンス・マニュアル』> 「SELECT 文」を参照してください。

- **SQL/XML** Adaptive Server Anywhere は、リレーショナル・データから XML 文書を生成する、ドラフト段階の SQL/XML 規格に基づく関数をサポートしています。

  クエリ内でこれらの関数を 1 つ以上使用する方法については、「SQL/XML を使用してクエリ結果を XML として取得する」675 ページを参照してください。

Adaptive Server Anywhere がサポートする FOR XML 句と SQL/XML 関数により、リレーショナル・データから XML を生成する選択肢が 2 とおり提供されます。多くの場合、どちらを使用しても同じ XML が生成されます。

たとえば、このクエリは、FOR XML AUTO を使用して XML を生成しています。

```
SELECT id, name
 FROM product
 WHERE color='black'
 FOR XML AUTO
```

そして、このクエリは、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 product
 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

Adaptive Server 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、『ASA SQL リファレンス・マニュアル』>「XML データ型 [文字]」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」654 ページを参照してください。

Adaptive Server 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」654 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」657 ページを参照してください。

- **EXPLICIT** 希望するネストに関する情報を含むクエリを記述できます。そのため、生成される XML のフォームを制御できます。

  詳細については、「FOR XML EXPLICIT の使用」661 ページを参照してください。

以降の項では、FOR XML 句の 3 モードに共通する、バイナリ・データ、NULL 値、無効な XML 名に関連する動作について説明します。また、FOR XML 句の使用方法の例を示します。

FOR XML とバイナリ・データ

SELECT 文で FOR XML 句を使用すると、使用するモードに関わらず、すべての BINARY、LONG BINARY、IMAGE、または VARBINARY カラムは、自動的に Base64 エンコード・フォーマットで表される属性または要素として出力されます。

OPENXML を使用して XML から結果セットを生成する場合、OPENXML は、BINARY、LONG BINARY、IMAGE、VARBINARY の各データ型を Base64 でエンコードされていると見なし、自動的に復号化します。

OPENXML の詳細については、『ASA SQL リファレンス・マニュアル』> 「OPENXML 関数 [文字列]」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。

次のデータを含むテーブルを考えてみます。

id	fname	lname	company_name
100	Robert	Michaels	NULL
101	Michael	Devlin	The Power Group

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します。

```
SELECT id, fname, lname, company_name
 FROM customer
 FOR XML RAW
```

company_name 属性は、Robert Michaels の結果から省略されます。

```
<row id="100" fname="Robert" lname="Michaels"/>
 <row id="101" fname="Michael" lname="Devlin"
  company_name="The Power Group"/>'
```

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" fname="Robert" lname="Michaels"
  company_name=""/>
 <row id="101" fname="Michael" lname="Devlin"
  company_name="The Power Group"/>'
```

この場合、空の company_name 属性が Robert Michaels 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、『ASA データベース管理ガイド』>「FOR_XML_NULL_TREATMENT オプション [データベース]」を参照してください。

無効なカラム名

Adaptive Server 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 名文字として有効ではない Unicode 文字は、_x*HHHH*_ としてエンコードされます。ここで、*HHHH* は、その文字の 4 桁の 16 進コードポイントです。Unicode コードポイント値が 16 ビットで指定できない文字は、エンコードされます。

  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 emp_id AS "employee id"
   FROM employee
   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"/>
   ...
  ```

- 非 Unicode 文字は、_x*HHHHHHHH*_ としてエンコードされます。ここで、*HHHHHHHH* は、その文字の 8 桁の 16 進コードポイントです。

- 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。

- 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、『ASA SQL リファレンス・マニュアル』> 「SELECT 文」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、『ASA データベース管理ガイド』> 「TRUNCATION_LENGTH オプション [Interactive SQL]」と『SQL Anywhere Studio ヘルプ』> 「[オプション] ダイアログ : [結果] タブ」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 employee
        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(unit_price) AS Price
 FROM product
 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 emp_dept
 AS SELECT emp_lname, emp_fname, dept_name
 FROM employee JOIN department
 ON employee.dept_id = department.dept_id
 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、<emp_id> 要素と <dept_name> 要素を生成します。

```
SELECT employee.emp_id, department.dept_name
 FROM employee JOIN department
    ON employee.dept_id=department.dept_id
 FOR XML RAW, ELEMENTS
```

次の結果が得られます。

```
<row>
    <emp_id>102</emp_id>
    <dept_name>R &amp; D</dept_name>
 </row>
 <row>
    <emp_id>105</emp_id>
    <dept_name>R &amp; D</dept_name>
 </row>

<row>
    <emp_id>160</emp_id>
    <dept_name>R &amp; D</dept_name>
 </row>
 <row>
    <emp_id>243</emp_id>
    <dept_name>R &amp; D</dept_name>
 </row>
 ...
```

使用方法

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」651 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 emp_id AS id, emp_fname AS
name
                    FROM employee FOR XML RAW))
```

XMLELEMENT 関数の詳細については、『ASA SQL リファレンス・マニュアル』> 「XMLELEMENT 関数 [文字列]」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 product
 WHERE color='black'
 FOR XML RAW
```

次の結果が得られます。

```
<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 employee.emp_id, department.dept_name
 FROM employee JOIN department
    ON employee.dept_id=department.dept_id
 ORDER BY emp_id
 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 employee.emp_id, department.dept_name
 FROM employee JOIN department
    ON employee.dept_id=department.dept_id
 FOR XML RAW
```

次の XML 文書が返されます。

```
<row emp_id="102" dept_name="R &amp; D"/>
 <row emp_id="105" dept_name="R &amp; D"/>
 <row emp_id="160" dept_name="R &amp; D"/>
 <row emp_id="243" dept_name="R &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 employee.emp_id, department.dept_name
 FROM employee JOIN department
    ON employee.dept_id=department.dept_id
 ORDER BY emp_id
 FOR XML AUTO, ELEMENTS
```

この場合、結果セット内の各カラムは、<employee> 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。

```
<employee>
    <emp_id>102</emp_id>
    <department>
       <dept_name>R &amp; D</dept_name>
    </department>
 </employee>

<employee>
    <emp_id>105</emp_id>
    <department>
       <dept_name>R &amp; D</dept_name>
    </department>
 </employee>


<employee>
    <emp_id>160</emp_id>
    <department>
    <dept_name>R &amp; D</dept_name>
    </department>
 </employee>
 ...
```

使用方法

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 オプションの設定については、『ASA データベース管理ガイド』> 「FOR_XML_NULL_TREATMENT オプション [データベース]」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、希望するネストが得られないことがあります。

FOR XML AUTO は、整形式 XML 文書を返しません。これは、文書に単一のルート・ノードが含まれないためです。<root> 要素が必要な場合は、1 つの方法として、XMLELEMENT 関数を使用して挿入できます。次に例を示します。

```
SELECT XMLELEMENT( NAME root,
                    (SELECT emp_id AS id, emp_fname AS
name
                    FROM employee FOR XML AUTO ) )
```

XMLELEMENT 関数の詳細については、『ASA SQL リファレンス・マニュアル』> 「XMLELEMENT 関数 [文字列]」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 product
 WHERE color='black'
 FOR XML AUTO
```

次の XML が生成されます。

```
<product product_id="302"/>
 <product product_id="400"/>
 <product product_id="501"/>
 <product 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 product 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 employee.emp_id, department.dept_name
 FROM employee JOIN department
    ON employee.dept_id=department.dept_id
 ORDER BY emp_id
 FOR XML AUTO
```

前述のクエリによって、次の XML が生成されます。

```
<employee emp_id="102">
    <department dept_name="R &amp; D"/>
 </employee>
 <employee emp_id="105">
    <department dept_name="R &amp; D"/>
 </employee>


 <employee emp_id="160">
    <department dept_name="R &amp; D"/>
 </employee>
 <employee emp_id="243">
    <department dept_name="R &amp; D"/>
 </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 department.dept_name, employee.emp_id
 FROM employee JOIN department
    ON employee.dept_id=department.dept_id
 ORDER BY 1, 2
 FOR XML AUTO
```

結果は次のようにネストされます。

```
<department dept_name="R &amp; D">
    <employee emp_id="102"/>
    <employee emp_id="105"/>
    <employee emp_id="160"/>
    ...
 </department>
```

```
<department dept_name="Sales">
    <employee emp_id="129"/>
    <employee emp_id="195"/>
    <employee emp_id="299"/>
    ...
 </department>
 ...
```

ここでも、クエリによって生成された XML には、<employee> 要素と <department> 要素の両方が含まれています。しかしこの場合は、<department> 要素が <employee> 要素の親となっています。

FOR XML EXPLICIT の使用

FOR XML EXPLICIT を使用して、クエリが返す XML 文書の構造を制御できます。クエリは特定の方法で記述して、希望するネストに関する情報がクエリ結果内で指定されるようにしてください。FOR XML EXPLICIT がサポートするオプションのディレクティブを使用すると、個別のカラムの扱いを設定できます。たとえば、あるカラムが要素内容と属性内容のどちらとして表示されるかを制御できます。また、あるカラムが生成された XML に含まれるのではなく、結果の順序付けのみに使用されるように制御できます。

FOR XML EXPLICIT を使用してクエリを記述する方法の例は、「EXPLICIT モードのクエリの記述」664 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 (**first_name!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」662 ページで説明します)。

Tag	Parent	first_name!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。

```
<first_name>Beth</first_name>
 <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。

```
<first_name>Beth
    <id>102</id>
 </first_name>
```

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」664 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、<first_name> 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 prod_id!1!color という名前の場合、color は <prod_id> 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」670 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」669 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」672 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」673 ページを参照してください。

使用方法

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、『ASA データベース管理ガイド』> 「FOR_XML_NULL_TREATMENT オプション [データベース]」と 「FOR XML と NULL 値」651 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 emp_id='129'>
    <customer cust_id='107' region='Eastern'/>
    <customer cust_id='119' region='Western'/>
    <customer cust_id='131' region='Eastern'/>
 </employee>


 <employee emp_id='195'>
    <customer cust_id='109' region='Eastern'/>
    <customer cust_id='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!emp_id	customer!2!cust_id	customer!2!region
1	NULL	129	NULL	NULL
2	1	129	107	Eastern

Tag	Parent	employee!1!emp_id	customer!2!cust_id	customer!2!region
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 emp_id 属性が作成され、<customer> 要素に対して cust_id 属性と region 属性が作成されます。

次の手順では、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,
       emp_id  AS [employee!1!emp_id],
       NULL    AS [customer!2!cust_id],
       NULL    AS [customer!2!region]
 FROM employee
```

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 emp_id、cust_id、region は属性であると指定しています。

```
SELECT
       2,
       1,
       emp_id,
       cust_id,
       region
 FROM employee KEY JOIN sales_order
```

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,
       emp_id   AS [employee!1!emp_id],
       NULL     AS [customer!2!cust_id],
       NULL     AS [customer!2!region]
 FROM employee
 UNION ALL

SELECT
       2,
       1,
       emp_id,
       cust_id,
       region
 FROM employee KEY JOIN sales_order
```

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,
       emp_id   AS [employee!1!emp_id],
       NULL     AS [customer!2!cust_id],
       NULL     AS [customer!2!region]
 FROM employee
 UNION ALL

SELECT
       2,
       1,
       emp_id,
       cust_id,
       region
 FROM employee KEY JOIN sales_order
 ORDER BY 3, 1
 FOR XML EXPLICIT
```

EXPLICIT モードの構文については、「パラメータ」661 ページを参照してください。

FOR XML EXPLICIT の例

次のクエリ例は、従業員による発注に関する情報を取り出します。この例では、<emp>、<order>、<dept> という 3 種類の要素があります。<emp> 要素は id 属性と name 属性を持ち、<order> 要素は date 属性を、また <dept> 要素は name 属性を持ちます。

```
SELECT
         1            tag,
         NULL         parent,
         emp_id       [emp!1!id],
         emp_fname    [emp!1!name],
         NULL         [order!2!date],
         NULL         [dept!3!name]
 FROM employee
 UNION ALL

SELECT
         2,
         1,
         emp_id,
         NULL,
         order_date,
         NULL
 FROM employee KEY JOIN sales_order
 UNION ALL

SELECT
         3,
         1,
         emp_id,
         NULL,
         NULL,
         dept_name
 FROM employee e JOIN department d
    ON e.dept_id=d.dept_id
 ORDER BY 3, 1
 FOR XML EXPLICIT
```

このクエリから次の結果が得られます。

```
<emp id="102" name="Fran">
    <dept name="R &amp; D"/>
 </emp>
 <emp id="105" name="Matthew">
    <dept name="R &amp; D"/>
 </emp>
```

```
<emp id="129" name="Philip">
    <order date="2000-07-24"/>
    <order date="2000-07-13"/>
    <order date="2000-06-24"/>
    <order date="2000-06-08"/>
    ...
    <dept name="Sales"/>
 </emp>

<emp id="148" name="Julie">
    <dept name="Finance"/>
 </emp>
 ...
```

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,
         emp_id     [emp!1!id!element],
         emp_fname  [emp!1!name!element],
         NULL       [order!2!date!element],
         NULL       [dept!3!name!element]
 FROM employee

UNION ALL
 SELECT
         2,
         1,
         emp_id,
         NULL,
         order_date,
         NULL
 FROM employee KEY JOIN sales_order
 UNION ALL

SELECT
         3,
         1,
         emp_id,
         NULL,
         NULL,
```

```
        dept_name
 FROM employee e JOIN department d
    ON e.dept_id=d.dept_id
 ORDER BY 3, 1
 FOR XML EXPLICIT
```

このクエリから次の結果が得られます。

```
<emp>
    <id>102</id>
    <name>Fran</name>
    <dept>
       <name>R &amp; D</name>
    </dept>
 </emp>

<emp>
    <id>105</id>
    <name>Matthew</name>
    <dept>
       <name>R &amp; D</name>
    </dept>
 </emp>

<emp>
    <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>
    ...
    <dept>
       <name>Sales</name>
    </dept>
 </emp>
 ...
```

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,
         emp_id       [emp!1!id!hide],
         emp_fname    [emp!1!name],
         NULL         [order!2!date],
         NULL         [dept!3!name]
 FROM employee
 UNION ALL

SELECT
         2,
         1,
         emp_id,
         NULL,
         order_date,
         NULL
 FROM employee KEY JOIN sales_order
 UNION ALL

SELECT
         3,
         1,
         emp_id,
         NULL,
         NULL,
         dept_name
 FROM employee e JOIN department d
    ON e.dept_id=d.dept_id
 ORDER BY 3, 1
 FOR XML EXPLICIT
```

このクエリは、次の結果を返します。

```
<emp name="Fran">
    <dept name="R &amp; D"/>
 </emp>
 <emp name="Matthew">
    <dept name="R &amp; D"/>
 </emp>
```

```
<emp name="Philip">
    <order date="2000-07-24"/>
    <order date="2000-07-13"/>
    <order date="2000-06-24"/>
    <order date="2000-06-08"/>
    ...
    <dept name="Sales"/>
 </emp>

<emp name="Julie">
    <dept name="Finance"/>
 </emp>
 ...
```

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、「無効なカラム名」652 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],
         company_name     AS [customer!1!company_name]
 FROM customer
 WHERE id = '115'
 FOR XML EXPLICIT
```

このクエリによって生成される結果では、アンパサンドはエスケープされます。これは、このカラムが XML 型ではないためです。

```
<customer company_name="Sterling &amp; Co.">
    <id>115</id>
 </customer>
```

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],
         company_name     AS
[customer!1!company_name!xml]
 FROM customer
 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>
  <company_name>Sterling & Co.</company_name>
 </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。Adaptive Server 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 product
 FOR XML EXPLICIT
```

このクエリによって生成される結果は、各製品の説明を CDATA セクション内にリストします。CDATA セクションに含まれるデータは、引用されません。

```
<product id="300">
    <![CDATA[Tank Top]]>
 </product>
 <product id="301">
    <![CDATA[V-neck]]>
 </product>

<product id="302">
    <![CDATA[Crew Neck]]>
 </product>
 <product id="400">
    <![CDATA[Cotton Cap]]>
 </product>
 ...
```

SQL/XML を使用してクエリ結果を XML として取得する

SQL/XML は、XML を SQL 言語に機能統合する方法を定める、ドラフト段階の規格です。SQL/XML は、XML とともに SQL を使用する方法を定めています。サポートされる関数を使用して、リレーショナル・データから XML 文書を構成するクエリを記述できます。

無効な名前と SQL/XML

SQL/XML では、スペースを含む式など、有効な XML 名でない式は、FOR XML 句と同様にエスケープされます。XML 型の要素の内容は、引用されません。

無効な式のマークアップの詳細については、「無効なカラム名」652 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」637 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 department,
                   XMLATTRIBUTES (dept_id ),
                   XMLAGG( XMLELEMENT( NAME name,
                             emp_lname )
                             ORDER BY emp_lname )
                 ) AS dept_list
 FROM employee
 GROUP BY dept_id
```

このクエリは、次の結果を生成します。

dept_list
<department dept_id="100"> <name>Breault</name> <name>Cobb</name> <name>Diaz</name> <name>Driscoll</name> ... </department>
<department dept_id="200"> <name>Chao</name> <name>Chin</name> <name>Clark</name> <name>Dill</name> ... </department>
<department dept_id="300"> <name>Bigelow</name> <name>Coe</name> <name>Coleman</name> <name>Davidson</name> ... </department>
...

XMLAGG 関数の詳細については、『ASA SQL リファレンス・マニュアル』> 「XMLAGG 関数 [文字列]」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、employee テーブルの従業員ごとに、<first_name> 要素と <last_name> 要素を連結します。

```
SELECT XMLCONCAT( XMLELEMENT( NAME first_name,
emp_fname ),
                  XMLELEMENT( NAME last_name, emp_lname
)
                   ) AS "Employee_Name"
 FROM employee
```

このクエリは、次の結果を返します。

Employee_Name
<first_name>Fran</first_name> <last_name>Whitney</last_name>
<first_name>Matthew</first_name> <last_name>Cobb</last_name>
<first_name>Philip</first_name> <last_name>Chin</last_name>
<first_name>Julie</first_name> <last_name>Jordan</last_name>
...

詳細については、『ASA SQL リファレンス・マニュアル』>「XMLCONCAT 関数 [文字列]」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, product.name ),
       XMLELEMENT( NAME quantity_left, product.quantity
),
```

```
        XMLELEMENT( NAME description, product.size || '
' ||
                              product.color || ' ' ||
                           product.name )
        ) AS results
 FROM product
 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 product
 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、unit_price に対して属性を生成します。

```
SELECT id, XMLELEMENT( NAME item_description,
                       XMLATTRIBUTES( name,
                                      color,
                                      unit_price )
                     ) AS item_description_element
 FROM product
 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 ( unit_price AS
            price ),
                       product.name
                     ) AS products
 FROM product
 WHERE id > 400
```

詳細については、『ASA SQL リファレンス・マニュアル』>「XMLELEMENT 関数 [文字列]」を参照してください。

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,
                                  color,
                                  unit_price AS price )
                     ) AS product_info
 FROM product
 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、『ASA SQL リファレンス・マニュアル』>「XMLFOREST 関数 [文字列]」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、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。

- **{$id}** sales_order テーブルの id カラムの値を使用して、<id> 要素の内容を生成します。
- **{$order_date}** sales_order テーブルの order_date カラムの値を使用して、<date> 要素の内容を生成します。
- **{$customer}** customer テーブルの company_name カラムから <customer> 要素の内容を生成します。

```
SELECT XMLGEN ( '<order>
                  <id>{$id}</id>
                  <date>{$order_date}</date>
                  <customer>{$customer}</customer>
                  </order>',
                 sales_order.id,
                 sales_order.order_date,
                 customer.company_name AS customer)
  FROM sales_order JOIN customer
  ON customer.id = sales_order.cust_id
```

このクエリは、次の結果を生成します。

order_info

`<order>`
`  <id>2131</id>`
`  <date>2000-01-02</date>`
`  <customer>BoSox Club</customer>`
`  </order>`

`<order>`
`  <id>2126</id>`
`  <date>2000-01-03</date>`
`  <customer>Leisure Time</customer>`
`  </order>`

`<order>`
`  <id>2065</id>`
`  <date>2000-01-03</date>`
`  <customer>Bloomfield's</customer>`
`  </order>`

order_info
<order>   <id>2127</id>   <date>2000-01-06</date>   <customer>Creative Customs Inc.</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="{$id}">
                  <date>{$order_date}</date>
                  <customer>{$customer}</customer>
                  </order>',
                 sales_order.id,
                 sales_order.order_date,
                 customer.company_name AS customer
               ) AS order_info

FROM sales_order JOIN customer
  ON customer.id = sales_order.cust_id
  ORDER BY sales_order.order_date
```

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="2126">\   <date>2000-01-03</date>\   <customer>Leisure Time</customer>\ </order>
<order id="2065">\   <date>2000-01-03</date>\   <customer>Bloomfield's</customer>\ </order>

order_info
<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」675 ページ を参照してください。

XML 文書のヘッダ情報の指定

Adaptive Server 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 fname, lname FROM customer 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 fname="Michael" lname="Devlin"/>
  <row fname="Beth" lname="Reiser"/>
  <row fname="Erin" lname="Niedringhaus"/>
  <row fname="Meghan" lname="Mason"/>
  ...
 </r>
```

XMLGEN 関数の詳細については、『ASA SQL リファレンス・マニュアル』> 「XMLGEN 関数 [文字列]」を参照してください。

第 5 部　リモート・データとバルク・オペレーション

第 5 部では、データベースのロードとアンロードの方法、およびリモートデータへのアクセス方法について説明します。

第 17 章

データのインポートとエクスポート

この章の内容

この章では、Adaptive Server Anywhere データベースとの間でのバルク・データの転送プロセスについて説明します。

データベースとの間でのデータの転送

さまざまな状況下で、データをデータベースから出し入れすることが必要になる場合があります。次に例を示します。

- 新しいデータベースに最初のデータ・セットをインポートする。
- データベースの構造を修正した場合などに、新しいデータベースを構築する。
- スプレッドシートなど、他のアプリケーションで使うためにデータベースからデータをエクスポートする。
- レプリケーションまたは同期に使用するデータベースの抽出を作成する。
- 破損したデータベースを修復する。
- データベースを再構築してパフォーマンスを向上させる。
- 新しいバージョンのデータベース・ソフトウェアを入手し、ソフトウェア・アップグレードを完了する。

多くの場合、これらのタスクは、**バルク・オペレーション**としてグループ化されます。これらは、**DBA 権限**を持つユーザによって実行されます。通常のエンドユーザ・アプリケーションの一部ではありません。バルク・オペレーションは、同時実行性とトランザクション・ログに特定の要求を課す場合があり、通常は他のユーザがデータベースに接続していないときに実行するのが最適です。

バルク・オペレーションのパフォーマンスの側面

バルク・オペレーションのパフォーマンスは、オペレーションがデータベース・サーバの内部と外部のどちらに対するものかによって決まります。

データベース・サーバの外部：クライアント側

INPUT コマンドと OUTPUT コマンドは、Interactive SQL クライアントを通じて実行されます。Interactive SQL がデータベース・サーバとは別のマシン上で動作している場合、読み込みまたは書き込み対象のファイルのパスは、サーバではなくクライアントからの相対パスになります。INPUT は、読み込む各ローへの個別の INSERT 文としてトランザクション・ログに記録されます。このため、INPUT は LOAD TABLE よりかなり遅くなります。また、ON INSERT トリガが INPUT 中に起動することになります。値が指定されていない場合、NULL 入力可能なローには NULL が、NULL 入力不可能な数値カラムには 0 (ゼロ) が、NULL 入力不可能で数値以外のカラムには空の文字列が挿入されます。OUTPUT 文は、SELECT 文の結果セットを多数あるファイル・フォーマットのどれか 1 つで書き出すことができるので、互換性が重要な場合に役立ちます。

データベース・サーバの内部：サーバ側

LOAD TABLE 文、UNLOAD TABLE 文、UNLOAD 文は、データベース・サーバの内部で実行されます。書き込みまたは読み込み対象のファイルのパスは、データベース・サーバからの相対パスになります。LOAD TABLE 文は、単一のコマンドとしてトランザクション・ログに記録されます。データ・ファイルには、ロードされるテーブルと同じ数のカラムが必要です。デフォルト値を持つカラムに値が指定されていない場合で、DEFAULTS オプションが OFF (デフォルト) に設定されているときは、NULL、ゼロ、または空の文字列が挿入され、DEFAULTS オプションが ON に設定されている場合はデフォルト値が挿入されます。内部でインポートとエクスポートを行うと、テキスト形式と BCP フォーマットにだけアクセス可能ですが、この方法の方が速く処理できます。

詳細については、「データのインポートのパフォーマンスのヒント」692 ページを参照してください。

バルク・オペレーションのデータ・リカバリの問題

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。大量のデータをインポートするために**バルク・オペレーション**・モード (**-b** オプション) でデータベース・サーバを実行している場合、データベース・サーバは特定の重要機能を*実行しません*。具体的には、次のとおりです。

データベース・サーバがバルク・オペレーション・モードの場合にデータベース・サーバが実行しない機能	影響
ロールバック・ログの保持	セーブポイントは使用できず、コマンドをアボートするとトランザクションは必ずロールバックされます。
トランザクション・ログの管理	変更の記録がありません。
レコードのロック	重大な影響はありません。
データ定義コマンドの前のオートコミットの実行	オートコミットを無効にしてロールバックを実行すると、最後に実行したコミット以降の処理はすべて取り消されます。

メディア障害がある場合、上記の状態はデータベースに重大な影響を及ぼします。そのため、

COMMIT を頻繁に行う必要のあるデータでは、**-b** オプションはデータベースのオペレーションを遅くする可能性があります。COMMIT ごとにサーバはチェックポイントを実行します。これによって、データベース・サーバの動作が遅くなります。

詳細については、『ASA データベース管理ガイド』> 「-b サーバ・オプション」と『ASA データベース管理ガイド』> 「バックアップとデータ・リカバリ」を参照してください。

データベースへのデータのインポート

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。Adaptive Server Anywhere では、次の操作を行うことができます。

- ASCII ファイルからテーブル全体またはテーブルの一部をインポートする。
- DBASEII などの他のデータベース・フォーマットからテーブル全体またはテーブルの一部をインポートする。
- スクリプトでインポート手順を自動化して、連続した複数のテーブルをインポートする。
- テーブルにデータを挿入または追加する。
- テーブル内のデータを置換する。
- インポートの前またはインポート中にテーブルを作成する。
- BCP フォーマット句を使用して、Adaptive Server Anywhere と Adaptive Server Enterprise 間でファイルをインポートする。

まったく新しいデータベースを作成する場合は、パフォーマンスを考慮して、データをインポートではなく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」719 ページを参照してください。

以降の項では、インポート・ツールと、データベース、データ、テーブルのインポート手順について説明します。

インポートに関する考慮事項

データベースへのデータのインポートを開始する前に、現在あるリソースは何であるか、またデータベースへのデータのインポートによって厳密に何を行うのかを詳細に検討してください。

検討する要素	参照先
使用可能なインポート・ツール	「インポート・ツール」693 ページ
インポートがデータベース・パフォーマンスに影響を与える可能性	「バルク・オペレーションのパフォーマンスの側面」689 ページ 「データのインポートのパフォーマンスのヒント」692 ページ
データとインポート先テーブルとの互換性の問題	「インポート用のテーブル構造」702 ページ

データのインポートのパフォーマンスのヒント

データベースにデータを大量にインポートするには時間がかかることがありますが、時間を節約するための方法がいくつかあります。

- 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。
- 『ASA SQL リファレンス・マニュアル』> 「ALTER DBSPACE 文」に説明されているように、データベースのサイズを拡張する。通常の処理では、領域が必要になった時点で 256 KB ずつ拡張しますが、このコマンドを使うと、領域が必要になる前にデータベースを大幅に拡張できます。大量データをロードするときのパフォーマンスの向上だけでなく、ファイル・システム内で DB 領域の断片化を防ぐことにもなります。
- データのロードにテンポラリ・テーブルを使用する。ローカルまたはグローバルのテンポラリ・テーブルは、データ・セットを繰り返しロードする必要がある場合、または異なる構造を持つテーブルをマージする必要がある場合に役立ちます。
- LOAD TABLE 文を使用する場合、**-b** オプション (バルク・オペレーション・モード) を指定してサーバを起動する必要はないため、このオプションを指定しない。

- INPUT または OUTPUT コマンドを使用している場合は、データベース・サーバと同じマシン上で 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。新しいデータのロードは混雑している時間帯を避けてください。

インポート・ツール

データを Adaptive Server Anywhere データベースにインポートするのに役立つさまざまなツールがあります。これらの各インポート・ツールについて次に説明します。

Interactive SQL の [インポート] ウィザードの使用

Interactive SQL の [インポート] ウィザードでは、インポートするファイル、ファイル・フォーマット、データのインポート先テーブルを選択します。データは、テキスト、DBASEII、DBASEIII、Excel 2.1、FOXPRO、Lotus フォーマットでインポートできます。このデータを既存のテーブルにインポートすることを選択できます。または、ウィザードを使用してまったく新しいテーブルを作成し、設定できます。

次の場合には、Interactive SQL の [インポート] ウィザードの使用を検討してください。

- テキスト以外のフォーマットでのデータのインポートにグラフィカル・インタフェースを使用する場合。
- データのインポートと同時にテーブルを作成する場合。

例 1

この例では、Interactive SQL の [データ] メニューを使用してデータをインポートする方法を示します。

❖ **データをインポートするには、次の手順に従います (Interactive SQL の [データ] メニューの場合)。**

1. Interactive SQL ウィンドウから [データ] － [インポート] を選択します。

[開く] ダイアログが表示されます。

2 インポートするファイルを検索し、[開く] をクリックします。

[インポート] ウィザードが表示されます。

3 インポートするファイルにデータベースの値をどのように格納するかを指定します。

4 [既存のテーブルを使用します] オプションを選択し、既存のテーブルの名前とロケーションを入力します。[次へ] をクリックします。

[参照] ボタンをクリックすると、データをインポートするテーブルも検索できます。

5 ASCII ファイルの場合、ファイルのフィールド・デリミタ、エスケープ文字、後続ブランクを含めるかどうか、およびファイルに対するエンコードを指定できます。エンコード・オプションを使用することで、ファイルの読み込みに使用するコード・ページを指定して文字を確実に正しく処理するようにできます。**(デフォルト)** を選択した場合、Interactive SQL を実行しているマシン用のデフォルト・エンコードが使用されます。

サポートされているコード・ページの詳細については、『ASA データベース管理ガイド』> 「サポートされているコード・ページ」を参照してください。

6 ウィザードの指示に従います。

インポートを行うと、新しいデータが既存のテーブルに追加されます。インポートに成功すると、[メッセージ] ウィンドウ枠に、データのインポートに要した時間が表示されます。インポートに失敗した場合は、インポートに失敗したことを示すメッセージが表示されます。

例 2

次の例は、ASA 9.0 サンプル・データベースの employee テーブルにデータを追加する方法を示しています。この例は、Julia Chan という名前の従業員に関する情報を追加します。

❖ データをサンプル・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* という名前でを保存します。

```
100,500,'Julia','Chan',
 100, '300 Royal
Drive','Springfield','OR','97015','6175553985',
 'A','017239033', 55700,'1984-09-29',,'1968-05-
05','Y','Y','N','F'
```

2 Interactive SQL から、[データ] － [インポート] を選択します。

3 *import.txt* ファイルを検索し、[開く] をクリックします。

[インポート] ウィザードが表示されます。

4 インポートするファイルにデータベースの値をどのように格納するかを指定します。

5 [既存のテーブルを使用します] オプションを選択し、データのインポート先のテーブル名として **employee** と入力します。[次へ] をクリックします。

6 [インポート] ウィザードの残りの手順に従います。

INPUT 文を使用したデータのインポート

INPUT 文を使用すると、1 つまたは複数のテーブルにさまざまなファイル・フォーマットでデータをインポートできます。デフォルトの入力フォーマットを選択したり、INPUT 文ごとにファイル・フォーマットを指定したりすることができます。Interactive SQL では、複数の INPUT 文が含まれたコマンド・ファイルを実行できます。

次の場合は、INPUT 文を使用したデータのインポートを検討してください。

- 1 つ以上のテーブルにデータをインポートする場合。

- コマンド・ファイルを使用してインポート・プロセスを自動化する場合。
- テキスト以外のフォーマットでデータをインポートする場合。

詳細については、『ASA SQL リファレンス・マニュアル』> 「INPUT 文 [Interactive SQL]」を参照してください。

データベースに対する影響

変更は、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、変更の記録がありません。ただし、INPUT 文ではすべてをトランザクション・ログに書き込むので、INPUT 文を使用して大量のデータをインポートすると、パフォーマンスに影響が及びます。

INPUT 文は LOAD TABLE 文よりも低速です。データ・ファイルのフォーマットが DBASE、DBASEII、DBASEIII、FOXPRO、LOTUS のいずれかで、テーブルが存在しない場合は、テーブルが作成されます。

例

この例では、Interactive SQL の INPUT 文を使用してデータをインポートする方法を示します。

❖ **データをインポートするには、次の手順に従います (Interactive SQL の INPUT 文の場合)。**

1. 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 ウィンドウの [SQL 文] ウィンドウ枠に INPUT 文を入力します。次に例を示します。

```
INPUT INTO t1
FROM file1
FORMAT ASCII;
```

   `t1` はデータを入れるテーブルの名前で、`file1` はインポートするデータが入っているファイルの名前です。

3. 文を実行します。

インポートに成功すると、[メッセージ] ウィンドウ枠に、データのインポートに要した時間が表示されます。インポートに失敗した場合は、インポートに失敗したことを示すメッセージが表示されます。

INPUT 文を使用したデータのインポートについては、『ASA SQL リファレンス・マニュアル』> 「INPUT 文 [Interactive SQL]」を参照してください。

次は、ASCII テキスト・ファイルからの INPUT 文の例です。

```
INPUT INTO employee
FROM new_emp.inp
FORMAT ASCII;
```

LOAD TABLE 文を使用したデータのインポート

LOAD TABLE 文を使用すると、データだけを効率的にテキスト、ASCII、または FIXED フォーマットでテーブルにインポートできます。テーブルが存在し、入力ファイルのフィールド数と同じ数のカラムがあり、互換性のあるデータ型を使用して定義されていることが必要です。LOAD TABLE 文は、1 行に 1 ローずつ、値をデリミタで区切ってインポートします。

次の場合は、LOAD TABLE 文の使用を検討してください。

- テキスト・フォーマットでデータをインポートする場合。
- パフォーマンスを改善する必要があり、INPUT 文ではなく LOAD TABLE 文を使用できる場合。

詳細については、『ASA SQL リファレンス・マニュアル』> 「LOAD TABLE 文」を参照してください。

データベースに対する影響

変更は、トランザクション・ログに*記録されません*。データのリカバリが必要な場合に、この情報を入手できません。クラッシュが発生するとデータが失われ、この文を SQL Remote や Mobile Link リモート・データベースで使用できなくなります。

LOAD TABLE 文は、INPUT 文よりもかなり高速です。

LOAD TABLE 文はファイルの内容を、テーブルの既存のローに追加します。既存のローを置き換えるわけではありません。

INSERT 文を使用したデータのインポート

INSERT 文を使用して、データベースにローを追加できます。テーブルに入れるデータを直接 INSERT 文に含めるので、この文は対話型入力とみなされます。リモート・データ・アクセスに INSERT 文を使用して、ファイルではなく別のデータベースからデータをインポートすることもできます。

次の場合は、INSERT 文を使用したデータのインポートを検討してください。

- 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・フォーマットが問題にならない場合。
- ファイルではなく外部データベースからリモート・データをインポートする場合。

詳細については、『ASA SQL リファレンス・マニュアル』> 「INSERT 文」を参照してください。

データベースに対する影響

変更は、トランザクション・ログに記録されます。このため、メディア障害が発生した場合は、行った変更に関する情報をトランザクション・ログからリカバリできます。詳細については、『ASA データベース管理ガイド』> 「トランザクション・ログ」を参照してください。

例

この例では、INSERT 文を使用してデータをインポートする方法を示します。この INSERT 文では、サンプル・データベースの department テーブルにデータを追加します。

❖ データをインポートするには、次の手順に従います (INSERT 文の場合)。

1. 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. INSERT 文を実行します。次に例を示します。

```
INSERT INTO t1
 VALUES ( ... )
```

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。

次の例では、サンプル・データベースの **department** テーブルに新しいローを挿入します。

```
INSERT
 INTO department ( dept_id, dept_name, dept_head_id )
 VALUES ( 700, 'Training', 501)
```

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・テーブルを使用したデータのインポートを検討してください。

- 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。

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・アクセスについては、「リモート・データへのアクセス」745 ページを参照してください。

INSERT 文については、『ASA SQL リファレンス・マニュアル』>「INSERT 文」を参照してください。

ロード・ツール

次のいずれかのツールを使用して、データベースにデータをロードできます。

- 「INSERT 文を使用したデータのインポート」698 ページ
- 「LOAD TABLE 文を使用したデータのインポート」697 ページ

インポート中の変換エラーの処理

外部ソースからロードされたデータは、エラーを含む場合があります。たとえば、カラムのデータ型に合わない日付や数値が含まれていることがあります。CONVERSION_ERROR データベース・オプションを使えば、このような値を NULL 値に変換し、変換エラーを無視できます。

Interactive SQL データベース・オプションの設定の詳細については、『ASA SQL リファレンス・マニュアル』> 「SET OPTION 文」または『ASA データベース管理ガイド』> 「CONVERSION_ERROR オプション [互換性]」を参照してください。

テーブルのインポート

Interactive SQL の [データ] メニューまたは [文] ウィンドウ枠を使用して、既存のデータベースにテーブルをインポートできます。テーブル・データには、テキスト、DBASEII、Excel 2.1、FOXPRO、Lotus フォーマットを使用できます。

❖ テーブルをインポートするには、次の手順に従います (Interactive SQL の [データ] メニューの場合)。

1. データを入れるテーブルが存在することを確認します。
2. Interactive SQL ウィンドウから [データ] － [インポート] を選択します。

   [開く] ダイアログが表示されます。
3. インポートするファイルを検索し、[開く] をクリックします。

   [インポート] ウィザードが表示されます。

4　[次の名前で新しいテーブルを作成します] オプションを選択し、フィールドに新しいテーブルの名前を入力します。

5　ウィザードの指示に従います。

インポートに成功すると、[メッセージ] ウィンドウ枠に、データのインポートに要した時間が表示されます。インポートに失敗した場合は、インポートに失敗したことを示すメッセージが表示されます。[結果] ウィンドウ枠の [結果] タブに、使用された実行プランが表示されます。

❖ テーブルをインポートするには、次の手順に従います (Interactive SQL の場合)。

1　Interactive SQL ウィンドウの [SQL 文] ウィンドウ枠で、データのロード先となるテーブルを作成します。

CREATE TABLE 文を使用すると、テーブルを作成できます。

2　LOAD TABLE 文を実行します。次に例を示します。

```
LOAD TABLE department
 FROM 'dept.txt'
```

3　値の中の後続ブランクをそのままにするには、LOAD TABLE 文中で STRIP OFF 句を使用します。デフォルトの設定 (STRIP ON) では、データの挿入前に、値の後続ブランクを取り除きます。

LOAD TABLE 文はファイルの内容を、テーブルの既存のローに追加します。既存のローを置き換えるわけではありません。テーブルからすべてのローを削除するには、TRUNCATE TABLE 文を使います。

TRUNCATE TABLE 文と LOAD TABLE 文は、カスケード型削除のような参照整合性に関わるアクションを含め、トリガを起動することはありません。

LOAD TABLE 文の構文の詳細については、『ASA SQL リファレンス・マニュアル』> 「LOAD TABLE 文」を参照してください。

インポート用のテーブル構造

テーブルにロードするデータの構造は、必ずしもロード先のテーブルの構造と一致しません。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、いくつか方法があります。

- LOAD TABLE 文を使用して、テーブル内のカラムを並べ替えることができます。
- INSERT 文の一種とグローバル・テンポラリ・テーブルを使用して、インポート・データをテーブルに合うように並べ替えることができます。
- INPUT 文を使用して、カラムの特定のセットまたは順序を指定できます。

カラムに NULL 値を入力できるようにする

インポート中のファイルにテーブルのカラムのサブセットへのデータがある場合、またはカラムの順序が異なる場合は、LOAD TABLE 文の DEFAULTS オプションを使用して、ブランクを埋めて一致しないテーブル構造をマージすることもできます。

DEFAULTS オプションが OFF の場合は、カラム・リストにないカラムすべてに NULL が割り当てられます。DEFAULTS オプションが OFF で、NULL 入力不可能なカラムがカラム・リストから省かれている場合は、データベース・サーバは、空の文字列をカラムの型に変換しようとします。DEFAULTS オプションが ON で、カラムにデフォルト値が入っている場合は、その値が使用されます。

たとえば、employee テーブルに 2 つのカラムをロードし、残っているカラムがある場合はそのカラム値をデフォルト値に設定します。LOAD TABLE 文は次のようになります。

```
LOAD TABLE employee (emp_lname, emp_fname)
 FROM 'new_employees.txt'
 DEFAULTS ON
```

異なるテーブル構造のマージ

INSERT 文の一種とグローバル・テンポラリ・テーブルを使用して、インポート・データをテーブルに合うように並べ替えることができます。

❖ グローバル・テンポラリ・テーブルを使用して構造が異なるデータをロードするには、次の手順に従います。

1. Interactive SQL ウィンドウの [SQL 文] ウィンドウ枠で、入力ファイルと構造が一致するグローバル・テンポラリ・テーブルを作成します。

   CREATE TABLE 文を使用して、グローバル・テンポラリ・テーブルを作成できます。

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 FROM 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、データベース内の 1 つまたは複数の通常のテーブルに挿入します。

データベースからのデータのエクスポート

エクスポートは、データベースからのデータの書き出しに関する管理タスクです。エクスポートは、データベースの大部分を共有する必要がある場合、または特定の基準に従ってデータベースの一部を抽出する必要がある場合に便利なツールです。Adaptive Server Anywhere では、次の操作を行うことができます。

- 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。
- BCP FORMAT 句を使用して、Adaptive Server Anywhere と Adaptive Server Enterprise 間でファイルをエクスポートする。

データベース全体をエクスポートする場合は、パフォーマンスを考慮して、データをエクスポートするのではなくデータベースをアンロードしてください。

データベース全体のアンロードまたは再ロードの詳細については、「データベースの再構築」719 ページを参照してください。

ここでは、エクスポート・ツールの概要と、クエリ結果、データベース、テーブルのエクスポート手順について説明します。

エクスポートの考慮事項

データのエクスポートを開始する前に、現在あるリソースは何であるか、またデータベースから厳密にどのようなタイプの情報をエクスポートするのかを詳細に検討してください。

検討する要素	参照先
使用可能なエクスポート・ツール	「エクスポート・ツール」705 ページ

検討する要素	参照先
エクスポートがデータベースのパフォーマンスに与える影響	「バルク・オペレーションのパフォーマンスの側面」689 ページ
NULL 値の処理方法	「NULL の出力」716 ページ

エクスポート・ツール

次のようなさまざまなツールを使用して、データをエクスポートできます。

Interactive SQL を使用したデータのエクスポート

Interactive SQL の [エクスポート] ウィザードを使用して、データベースから取得したクエリの結果セットからデータをエクスポートできます。

このエクスポート・ツールは、結果セットをファイルに保存する場合に使用します。Interactive SQL で、[データ] － [エクスポート] を選択します。このとき、エクスポートするクエリ結果のファイル・フォーマットを選択できます。

例

次の例は、SQL クエリからファイルに結果セットをエクスポートする方法を示しています。

❖ **Interactive SQL を使用して結果セット・データをエクスポートするには、次の手順に従います。**

1. ASA 9.0 サンプル・データベースを使用して次のクエリを実行します。

```
SELECT * FROM employee
 WHERE state = 'GA'
```

結果セットには、ジョージア州に住むすべての従業員のリストが含まれます。

2 Interactive SQL で、[データ] － [エクスポート] を選択します。

3 [エクスポート] ダイアログで、ファイル名を入力し、ファイル・タイプとエクスポート先を選択します。

4 [OK] をクリックします。

結果セットが、指定した場所にあるファイルにエクスポートされます。

OUTPUT 文を使用したデータのエクスポート

OUTPUT 文を使用して、データベースからクエリ結果、テーブル、またはビューをエクスポートします。

OUTPUT 文は、SELECT 文の結果セットを複数の異なるファイル・フォーマットで書き出すことができるため、互換性が問題で役立ちます。デフォルトの出力フォーマットを指定したり、OUTPUT 文ごとにファイル・フォーマットを指定したりできます。Interactive SQL では、複数の OUTPUT 文が含まれたコマンド・ファイルを実行できます。

次の場合には、Interactive SQL の OUTPUT 文の使用を検討してください。

- 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・プロセスを自動化する場合。

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 文はサーバと同じマシン上で使用して、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、『ASA SQL リファレンス・マニュアル』>「OUTPUT 文 [Interactive SQL]」を参照してください。

例

次の例では、サンプル・データベース内の employee テーブルのデータを、*employee.txt* という名前の .TXT ファイルにエクスポートします。

```
SELECT *
FROM employee;
OUTPUT TO employee.txt
FORMAT ASCII
```

UNLOAD TABLE 文の使用

UNLOAD TABLE 文を使用すると、テキスト、ASCII、または FIXED フォーマットでデータだけを効率的にエクスポートできます。UNLOAD TABLE 文では、1 行に 1 ローずつ、値をカンマで区切ってエクスポートします。データは、プライマリ・キー値の順にエクスポートされるため、再ロードは速くなります。

UNLOAD TABLE 文は、Interactive SQL ウィンドウの [SQL 文] ウィンドウ枠から実行します。

次の場合は、UNLOAD TABLE 文の使用を検討してください。

- テキスト・フォーマットでテーブル全体をエクスポートする場合。
- データベース・パフォーマンスを考慮する場合。

データベースに対する影響

UNLOAD TABLE 文は、アンロード中にテーブル全体に排他ロックを配置します。

OUTPUT 文、UNLOAD 文、UNLOAD TABLE 文のいずれかを選択できる場合は、パフォーマンスを考慮して UNLOAD TABLE 文を選択します。

詳細については、『ASA SQL リファレンス・マニュアル』>「UNLOAD TABLE 文」を参照してください。

例

ASA 9.0 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 employee テーブルをテキスト・ファイル *emp_test.txt* にアンロードできます。

```
UNLOAD TABLE employee TO 'c:¥emp_test.txt'
```

UNLOAD 文の使用

UNLOAD 文は、クエリ結果をファイルにエクスポートするという点で OUTPUT 文に似ています。しかし、UNLOAD 文を使用すると、より効率的にテキスト、ASCII、または FIXED フォーマットだけでデータをエクスポートできます。UNLOAD 文は、1 行に 1 ローずつ、値をカンマで区切ってエクスポートします。

次の場合は、UNLOAD 文を使用したデータのアンロードを検討してください。

- パフォーマンスが重要な場合に、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する場合。
- アプリケーションにエクスポート・コマンドを埋め込む場合。

データベースに対する影響

OUTPUT 文、UNLOAD 文、UNLOAD TABLE 文のいずれかを選択できる場合は、パフォーマンスを考慮して UNLOAD TABLE 文を選択します。

UNLOAD 文を使用するユーザは、テーブルに対して ALTER パーミッションまたは SELECT パーミッションを持っていなければなりません。UNLOAD 文を使用できるユーザの管理の詳細については、『ASA データベース管理ガイド』> 「-gl サーバ・オプション」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。

プロキシ・テーブルを持つデータベースをアンロードして再ロードする場合は、ユーザがローカル・データベースとリモート・データベースの両方に対して同じパスワードを持っていても、外部ログインを作

成して、ローカル・ユーザをリモート・ユーザにマップしてください。外部ログインがない場合は、リモート・サーバに接続できないため再ロードが失敗します。

詳細については、『ASA SQL リファレンス・マニュアル』>「UNLOAD 文」を参照してください。

dbunload ユーティリティを使用したデータのエクスポート

dbunload ユーティリティでは、データベース内の 1 つ、多数、またはすべてのテーブルをエクスポートできます。テーブル・データだけでなくテーブル・スキーマもエクスポートできます。データベースのテーブルを配置し直す場合は、*dbunload* ユーティリティを使用して必要なコマンド・ファイルを作成し、必要に応じて修正できます。テーブルは、構造のみ、データのみ、または構造とデータの両方をアンロードできます。

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 は、グラフィックは異なりますが、機能的には同じものです。どちらを使っても同じ結果が得られます。

次の場合は、*dbunload* ユーティリティの使用を検討してください。

- 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。
- データベースを再構築または抽出する必要がある場合。

データベースに対する影響

データベース内のテーブルを一部だけアンロードする前に、接続を確立します。

dbunload ユーティリティの詳細については、『ASA データベース管理ガイド』> 「dbunload コマンド・ライン・ユーティリティを使用したデータベースのアンロード」を参照してください。

アンロード・ツール

次のツールを使用して、データベースからデータをアンロードできます。

- 「dbunload ユーティリティを使用したデータのエクスポート」709 ページ
- 「[データベース・アンロード] ウィザードの使用」710 ページ
- 「[データのアンロード] ダイアログの使用」711 ページ

[データベース・アンロード] ウィザードの使用

[データベース・アンロード] ウィザードを使用して、既存のデータベースを新しいデータベースにアンロードできます。

Sybase Central の [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 ASCII フォーマットでアンロードし、データベース全体の再作成に必要な Interactive SQL コマンド・ファイルを作成できます。これは、SQL Remote を抽出するときや、同一または少しだけ修正した構造を持つデータベースを新しく作成するときに便利です。[データベース・アンロード] ウィザードは、Adaptive Server Anywhere 内での再使用を目的として Adaptive Server Anywhere ファイルをエクスポートするときに便利です。

注意
Sybase Central の [データベース・アンロード] ウィザードと dbunload ユーティリティは、グラフィックは異なりますが、機能的には同じものです。どちらを使っても同じ結果が得られます。

[データのアンロード] ダイアログの使用

[データのアンロード] ダイアログを使用して、データベース内の 1 つ以上のテーブルをアンロードできます。この機能は、[データベース・アンロード] ウィザードまたはアンロード・ユーティリティ (dbunload) でも使用できますが、このダイアログを使用すると、[データベース・アンロード] ウィザード全体を実行する代わりに 1 ステップでテーブルをアンロードできます。

❖ **[データのアンロード] ダイアログを使用してテーブルをアンロードするには、次の手順に従います。**

1. Sybase Central のデータベースに接続します。
2. Sybase Central の左側のウィンドウ枠で、データベースを展開し、[テーブル] フォルダを展開します。
3. Sybase Central の右側のウィンドウ枠で、データのエクスポート元のテーブルを選択します。

   SHIFT キーを押したままでテーブルをクリックすると複数のテーブルを選択できます。
4. [ファイル] － [データのアンロード] を選択します。

   [データのアンロード] ダイアログが表示されます。
5. 目的のオプションを選択し、[OK] をクリックしてデータをアンロードします。

クエリ結果のエクスポート

Interactive SQL から [データ] メニューまたは OUTPUT 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することができます。

BCP FORMAT 句を使用して、Adaptive Server Anywhere と Adaptive Server Enterprise 間でファイルのインポートとエクスポートを実行できます。

詳細については、「Adaptive Server Enterprise の互換性」744 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] ウィンドウの [SQL 文] ウィンドウ枠にクエリを入力します。
2. [SQL 文の実行] をクリックして、結果セットを表示します。
3. [データ] － [エクスポート] を選択します。

   [名前を付けて保存] ダイアログが表示されます。
4. エクスポートするデータの名前とロケーションを指定します。
5. ファイル・フォーマットを指定して [保存] をクリックします。

   エクスポートに成功すると、[メッセージ] ウィンドウ枠に、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。

❖ 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] ウィンドウの [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** *'c:¥filename'* と入力します。

   たとえば、employee テーブル全体をファイル *employee.dbf* 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 employee;
 OUTPUT TO 'c:¥employee.dbf'
```

3　次のいずれかを行います。

処理	使用する文	例

クエリ結果をエクスポートして別のファイルに結果を追加する	APPEND	`SELECT * FROM employee;`
		`OUTPUT TO 'c:¥employee.dbf'`
		`APPEND`

クエリ結果をエクスポートしてメッセージをインクルードする	VERBOSE	`SELECT * FROM employee;`
		`OUTPUT TO 'c:¥employee.dbf'`
		`VERBOSE`

ASCII (デフォルトのファイル・フォーマット) 以外のファイル・フォーマットを指定する	FORMAT	`SELECT * FROM employee;`
		`OUTPUT TO 'c:¥employee.dbf'`
		`FORMAT dbaseiii;`
		
		ここで、*c:¥employee.dbf* は、新しいファイルのパス、名前、拡張子です。*dbaseiii* は、このファイルのファイル・フォーマットです。一重引用符または二重引用符で文字列を囲むことができますが、引用符が必要となるのは、パスにスペースが含まれる場合のみです。
		
		FORMAT オプションを指定しない場合、デフォルトのファイル・タイプは ASCII になります。

4　文を実行します。

エクスポートに成功すると、[メッセージ] ウィンドウ枠に、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、『ASA SQL リファレンス・マニュアル』> 「OUTPUT 文 [Interactive SQL]」を参照してください。

ヒント

APPEND 文と VERBOSE 文を組み合わせると、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` *'filename.sql'* `APPEND VERBOSE` のように入力します。APPEND と VERBOSE の詳細については、『ASA SQL リファレンス・マニュアル』> 「OUTPUT 文 [Interactive SQL]」を参照してください。

OUTPUT TO 文、APPEND 文、VERBOSE 文は、Interactive SQL の以前のバージョンの演算子 >#、>>#、>&、>>& に相当します。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。

❖ 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. Interactive SQL ウィンドウの [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 employee;
 TO 'c:¥employee.dbf'
```

2. UNLOAD 文を実行します。

   エクスポートに成功すると、[メッセージ] ウィンドウ枠に、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。

データベースのエクスポート

❖ 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 左ウィンドウ枠で、Adaptive Server Anywhere プラグインを選択します。

2 右ウィンドウ枠で、[ユーティリティ] タブをクリックします。

データベースに対して実行可能なすべての操作が右ウィンドウ枠に表示されます。

3 右ウィンドウ枠にある [データベースのアンロード] をダブルクリックします。

[データベース・アンロード] ウィザードが表示されます。

[データベース・アンロード] ウィザードを開くには、左ウィンドウ枠にあるデータベース名を右クリックしてポップアップ・メニューから [データベースのアンロード] を選択する方法、または [ツール] － [Adaptive Server Anywhere 9] － [データベースのアンロード] コマンドを選択する方法もあります。

4 ウィザードの指示に従います。

❖ データベースの全部または一部をアンロードするには、次の手順に従います (コマンド・ラインの場合)。

1 コマンド・プロンプトで *dbunload* コマンドを入力し、-c オプションを使用して接続パラメータを指定します。

たとえば、次のコマンドはデータベース全体を c:¥temp にアンロードします。

```
dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" c:¥temp
```

2 次のいずれかを行います。

- データのみをエクスポートするには、-d を使用します。次に例を示します。

```
dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -d
c:¥temp
```

- 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=asademo;uid=DBA;pwd=SQL" -n
c:¥temp
```

詳細については、『ASA データベース管理ガイド』> 「dbunload コマンド・ライン・ユーティリティを使用したデータベースのアンロード」を参照してください。

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、他のソフトウェアが NULL 値を解釈しない場合があるため、NULL 値をどのように出力するかを指定する方法が 2 つあります。

- Interactive SQL の NULLS オプションを使用すると、デフォルトの動作を設定したり、特定のセッションの出力値を変更したりできます。
- 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・パッケージ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)。

1. Interactive SQL ウィンドウから [ツール] ー [オプション] を選択して、[オプション] ダイアログを表示します。
2. [結果] タブをクリックします。

3　[NULL 値の表示形式] フィールドに、NULL 値と置き換える値を入力します。

4　次のいずれかを行います。

- [設定] をクリックします。これによって変更がデフォルトになります。
- [OK] をクリックします。これによって変更が現在のセッションでのみ有効になります。

Interactive SQL オプションの設定の詳細については、『ASA SQL リファレンス・マニュアル』> 「SET OPTION 文」を参照してください。

テーブルのエクスポート

以下の各項で説明する方法の他に、テーブルのデータをすべて選択しクエリ結果をエクスポートして、テーブルをエクスポートすることもできます。

詳細については、「クエリ結果のエクスポート」711 ページを参照してください。

ヒント
ビューのエクスポートは、テーブルのエクスポートと同じように実行できます。

❖ テーブルをエクスポートするには、次の手順に従います (コマンド・ラインの場合)。

- コマンド・プロンプトで次の *dbunload* コマンドを入力して [ENTER] キーを押します。

```
dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -t
employee c:¥temp
```

-c はデータベース接続パラメータを指定し、-t はエクスポートするテーブル名を指定します。この *dbunload* コマンドは、データをサンプル・データベース (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 *c:¥temp* 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・ファイルからデータベース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 department
 TO 'dept.txt'
```

この文は、department テーブルを、サンプル・データベースからサーバの現在の作業ディレクトリにある *dept.txt* ファイルにアンロードします。ネットワーク・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・マシンにではなく、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。ファイル名のディレクトリが n (¥n は改行文字) またはその他の特殊文字で始まる場合、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 (デリミタ) で区切られます。デリミタに使う文字は、DELIMITED BY 句で変更できます。フィールドは固定幅ではありません。エクスポートされるのは、各エントリの文字だけで、カラム幅全体ではありません。

UNLOAD TABLE 文の構文の詳細については、『ASA SQL リファレンス・マニュアル』> 「UNLOAD TABLE 文」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 (ロード／アンロード) と抽出のプロシージャを使用すると、データベースを再構築して、古いデータベースの一部から新しいデータベースを作成できます。データベースを再構築すると、データベースからすべての情報が取り出され、同一の形式で情報が戻されるので、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、領域が埋められてパフォーマンスが向上します。

データベースのパフォーマンスを向上させる場合、断片化したディスク領域を再利用する場合、または新しいバージョンの Adaptive Server Anywhere にデータベースをアップグレードする場合は、データベースの再構築を検討します。

データベースの再構築は、Sybase Central から実行するか、または *dbunload* コマンド・ライン・ユーティリティを使用して実行できます。

注意
データベースのバックアップを作成してから再構築することをおすすめします。詳細については、『ASA データベース管理ガイド』> 「バックアップとリカバリの概要」を参照してください。

ある Adaptive Server Anywhere データベースから別の Adaptive Server Anywhere データベースへ

再構築では、通常 Adaptive Server Anywhere データベースからデータを取り出し、それを別の Adaptive Server Anywhere データベースに入れ直します。アンロードと再ロードは、通常、どちらか一方のタスクではなく両方が実行されるため、密接な関係があります。

再構築とエクスポート

再構築がエクスポートとは異なる点は、再構築では、データの他にテーブル定義とスキーマのエクスポートとインポートが行われることです。再構築処理のアンロード部分では、ASCII フォーマットのデータ・ファイルと、テーブルとその他の定義が含まれる *reload.sql* ファイルが生成されます。*reload.sql* スクリプトを実行すると、テーブルが再作成され、そこにデータがロードされます。

SQL Remote または Mobile Link を使用している場合には、(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の詳細については、「データベースの抽出」732 ページを参照してください。

レプリケートするデータベースの再構築

データベース再構築のプロシージャは、データベースがレプリケーションに関連するかしないかによって異なります。データベースがレプリケーションに関連する場合は、操作中はトランザクション・ログのオフセットを保存しておいてください。これは、Message Agent と Replication Agent がこの情報を必要とするためです。データベースがレプリケーションに関連しない場合、処理はもっと簡単です。

データベースを再構築する理由

さまざまな理由で、データベースの再構築を検討します。データベースを再構築するのは、次のような場合です。

- **データベースのファイル・フォーマットのアップグレード**
  アップグレード・ユーティリティを適用すると一部の新機能が使用可能になります。ただし、データベースのファイル・フォーマットのアップグレードを必要とする機能もあります。ファイル・フォーマットのアップグレードとは、データベースをアンロードして再ロードすることです。特定の機能を有効にするためにアンロードと再ロードが必要かどうかについては、新機能に関するマニュアルで説明します。

  新しいバージョンの Adaptive Server 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機能を使用する場合は、データベースをアンロードして再ロードしてください。データベースを再構築してからバックアップしてください。データベースのアップ

グレードの詳細については、『SQL Anywhere Studio 新機能ガイド』> 「Adaptive Server Anywhere のアップグレード」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、次の理由からパフォーマンスが向上します。

  - データベース内のページのデータが断片化されている場合、アンロードと再ロードによって断片化を解除できる。

注意

データベースの断片化を解消する必要があるが、データベースへのアクセスは中断させないという稼働条件があるためにデータベース全体の再構築が不可能な場合は、データベースを再構築する代わりにテーブルを再編成することを検討してください。

テーブルの再編成の詳細については、『ASA SQL リファレンス・マニュアル』> 「REORGANIZE TABLE 文」を参照してください。

  - プライマリ・キーの順にデータ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。

再構築の考慮事項

再構築 (ロードとアンロード) と抽出のプロシージャを使用すると、データベースを再構築して、古いデータベースの一部から新しいデータベースを作成できます。データベースを再構築する前に、プロセスの開始方法を検討してください。

検討する要素	参照先
データベースの再構築に関連するもの	「データベースの再構築」719 ページ 「データベースを再構築する理由」720 ページ
再構築に使用可能なツール	「再構築ツールと再ロード・ツール」727 ページ
再構築がデータベース・ユーザに与える影響	「再構築中のダウン時間の最短化」730 ページ
データベースがレプリケーションに関連するかどうか	「レプリケーションに関連するデータベースの再構築」724 ページ 「レプリケーションに関連しないデータベースの再構築」722 ページ
データベースにプロキシ・テーブルが含まれるかどうか	「プロキシ・テーブルのあるデータベースの再構築」727 ページ

レプリケーションに関連しないデータベースの再構築

レプリケーションに関連しないデータベースの場合のみに、次の手順を使用してください。

レプリケーションに関連しないデータベースを Sybase Central から再構築する方法については、『SQL Anywhere Studio 新機能ガイド』> 「データベース・ファイル・フォーマットのアップグレード」を参照してください。

❖ レプリケーションに関連しないデータベースを再構築するには、次の手順に従います (コマンド・ラインの場合)。

1 コマンド・プロンプトで、次のオプションのいずれかを使用して、*dbunload* コマンド・ライン・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=asademo.db;uid=DBA;pwd=SQL" -an asacopy.db`
既存のデータベースへの再ロード	**-ac**	`dbunload -c "dbf=asademo.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=asademo.db;uid=DBA;pwd=SQL" -ar "uid=DBA;pwd=SQL;dbf=newdemo.db"`

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・ラインにアンロード・ディレクトリを指定しません。このため、データのセキュリティが向上します。`-ar` オプションと `-an` オプションは Sybase Central よりも高速ですが、`-ac` は低速です。

2 再ロードしたデータベースを使用する前に、データベースを停止し、トランザクション・ログを圧縮します。

注意

`-an` オプションと `-ar` オプションは、パーソナル・サーバへの接続、または共有メモリ経由によるネットワーク・サーバへの接続にのみ適用されます。

dbunload ユーティリティでは追加オプションを使用できます。追加オプションを使用すると、アンロード作業をチューニングできます。その他、実行中または実行していないデータベースとデータベース・パラメータを指定できる接続パラメータ・オプションもあります。

レプリケーションに関連するデータベースの再構築

データベースが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。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き、古いトランザクション・ログのオフセットは、新しいトランザクション・ログのオフセットとは異なるので、古いログは使用できなくなります。このため、レプリケーションが関係する場合、適切なバックアップが特に重要です。

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。1 つの方法では、*dbunload* ユーティリティの `-ar` オプションを使用して、レプリケーションに干渉しない方法でアンロードと再ロードを実行します。もう 1 つの方法では、手動で同じタスクを実行します。

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、Adaptive Server 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。

これに対し、再構築とは、ロードとアンロードの 2 つの機能を結合したものです。ロードとアンロードでは、Adaptive Server Anywhere のある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、固定フォーマットのデータ・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。

❖ 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)。

1. データベースを停止します。
2. データベース・ファイルとトランザクション・ログ・ファイルを安全なロケーションにコピーしてオフラインでフル・バックアップを実行します。
3. コマンド・プロンプトで、*dbunload* を実行してデータベースを再構築します。

   ```
   dbunload -c connection_string -ar directory
   ```

   connection_string は、DBA 権限を持つ接続です。*directory* は、古いトランザクション・ログのレプリケーション環境で使用されるディレクトリです。データベースへの接続は他にはありません。

   -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、『ASA データベース管理ガイド』> 「アンロード・ユーティリティのオプション」を参照してください。
4. 新しいデータベースを停止します。データベースのリストア後に通常の妥当性検査を実行します。
5. 必要な運用オプションを使用してデータベースを起動します。これで、再ロードされたデータベースにアクセスできます。

注意

dbunload ユーティリティでは追加オプションを使用できます。追加オプションを使用すると、アンロード作業をチューニングできます。その他、実行中または実行していないデータベースとデータベース・パラメータを指定できる接続パラメータ・オプションもあります。

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、そうした操作の実行方法について説明します。

❖ 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。終了オフセットは後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」719 ページを参照してください。
6. 新しいデータベースを停止します。
7. 新しいデータベースで使用されていたトランザクション・ログ・ファイルを消去します。
8. 新しいデータベースで *dblog* を実行します。手順 3 で記録した終了オフセットを `-z` パラメータに指定し、相対オフセットを 0 に設定します。

   ```
   dblog -x 0 -z 137829 database-name.db
   ```
9. Message Agent を実行するときは、コマンド・ラインに元のオフライン・ディレクトリのパスを入力します。

10 データベースを起動します。これで、再ロードされたデータベースにアクセスできます。

プロキシ・テーブルのあるデータベースの再構築

プロキシ・テーブルを持つデータベースをアンロードして再ロードする場合は、ユーザがローカル・データベースとリモート・データベースの両方に対して同じパスワードを持っていても、外部ログインを作成して、ローカル・ユーザをリモート・ユーザにマップしてください。外部ログインがない場合は、リモート・サーバに接続できないため再ロードが失敗します。

外部ログインの詳細については、「外部ログインの使用」759 ページを参照してください。

再構築ツールと再ロード・ツール

データベースを再構築できるさまざまなツールがあります。既存のデータベースにデータを再ロードできるツールもいくつかあります。

dbunload ユーティリティと dbisql ユーティリティを使用したデータベースの再構築

dbunload ユーティリティまたは *dbisql* ユーティリティを使用すると、データベース全体をカンマ区切りの ASCII フォーマットでアンロードし、データベース全体の再作成に必要な Interactive SQL コマンド・ファイルを作成できます。これは、SQL Remote を抽出するときや、同一または少しだけ修正した構造を持つデータベースを新しく作成するときに便利です。これらのユーティリティは、Adaptive Server Anywhere 内での再使用を目的として Adaptive Server Anywhere ファイルをエクスポートするときに便利です。

注意
dbunload/dbisql 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、グラフィックは異なりますが、機能的には同じものです。どちらを使っても同じ結果が得られます。

次の場合は、*dbunload* ユーティリティまたは *dbisql* ユーティリティの使用を検討してください。

- 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、「レプリケーションに関連しないデータベースの再構築」722 ページ、「レプリケーションに関連するデータベースの再構築」724 ページ、『ASA データベース管理ガイド』> 「[データベース・アンロード] ウィザードを使用したデータベースのアンロード」を参照してください。

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、テキスト、ASCII、または FIXED フォーマットでデータだけを効率的にエクスポートできます。UNLOAD TABLE 文では、1 行に 1 ローずつ、値をカンマで区切ってエクスポートします。データはプライマリ・キー値の順にエクスポートされるため、再ロードは速くなります。

次の場合は、UNLOAD TABLE 文を使用したデータベースの再構築を検討してください。

- パフォーマンスが重要な場合。
- 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。

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。

UNLOAD TABLE 文を使用するユーザは、テーブルに対して ALTER パーミッションまたは SELECT パーミッションを持っていなければなりません。

詳細については、『ASA SQL リファレンス・マニュアル』> 「UNLOAD TABLE 文」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

❖ **テーブル・データまたはテーブル・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)。**

1 コマンド・プロンプトで *dbunload* コマンドを入力し、-c オプションを使用して接続パラメータを指定します。

2 -t オプションを使用して、データまたはスキーマのエクスポート対象となるテーブルを指定します。

たとえば、employee テーブルの一部をエクスポートするには、次のコマンドを入力します。

```
dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -t
employee c:¥temp
```

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。

3 次のいずれかを行います。

処理	追加するオプション	例
データのみのエクスポート	**-d**	`dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -d -t employee c:¥temp`
スキーマのみのエクスポート	**-n**	`dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -n -t employee c:¥temp`

4 [*Enter*] キーを押して、コマンドを実行します。

ここまでの例の *dbunload* コマンドは、データまたはスキーマを、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 *c:¥temp* ディレクトリのファイルにアンロードします。データ・ファイルからデータベース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* として現在のディレクトリに作成されます。

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 Adaptive Server Anywhere データベースからアンロードしたすべてのデータを新規に作成したテーブルに挿入します。データベースは、コマンド・ラインから再ロードします。

❖ データベースを再ロードするには、次の手順に従います (コマンド・ラインの場合)。

1. コマンド・プロンプトを開きます。
2. *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ます。

```
dbisql -c "dbn=asademo;uid=DBA;pwd=SQL" reload.sql
```

再構築中のダウン時間の最短化

次の手順を実行すると、ダウン時間を最短に抑えてデータベースを再構築できます。これは、データベースが 1 日 24 時間稼働している場合に特に役立ちます。

手順 1 ～ 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のバックアップ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。

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。
2. バックアップ・データベースを別のマシン上で再構築します。
3. 運用サーバ上で再度 DBBACKUP -r を実行してログの名前を変更してください。
4. 手順 3 のログに対して **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。

   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

抽出では、リモートの Adaptive Server Anywhere データベースが Adaptive Server Enterprise または Adaptive Server Anywhere の統合データベースから削除されます。

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。統合データベースからリモート・データベースを作成して同期させる場合は、抽出ユーティリティを使用することをおすすめします。

データベースを抽出する方法の詳細については、以下を参照してください。

- 『SQL Remote ユーザーズ・ガイド』> 「データベース抽出ユーティリティ」
- 『SQL Remote ユーザーズ・ガイド』> 「抽出ユーティリティの使用」
- 『SQL Remote ユーザーズ・ガイド』> 「抽出ユーティリティのオプション」
- 『SQL Remote ユーザーズ・ガイド』> 「グループの抽出」
- 『Mobile Link クライアント』> 「リモート・データベースの配備」
- 『SQL Remote ユーザーズ・ガイド』> 「Sybase Central でのリモート・データベースの抽出」

Adaptive Server Anywhere へのデータベースの移行

ストアド・プロシージャのセット sa_migrate または [データベース移行] ウィザードを使用して、リモートの Oracle、DB2、Microsoft SQL Server、Sybase Adaptive Server Enterprise、Adaptive Server Anywhere、Microsoft Access データベースにあるテーブルを Adaptive Server Anywhere にインポートできます。

[データベース移行] ウィザードの使用

[データベース移行] ウィザードやストアード・プロシージャの sa_migrate セットを使用してデータを移行する前に、まず**ターゲット・データベース**を作成してください。ターゲット・データベースとは、データの移行先となるデータベースのことです。

データベースの作成については、「データベースの作成」37 ページを参照してください。

[データベース移行] ウィザードを使用すると、リモート・データベースに接続するためのリモート・サーバ、および (必要に応じて) 現在のユーザをリモート・データベースに接続する外部ログインを作成することができます。

❖ リモート・テーブルをインポートするには、次の手順に従います (Sybase Central の場合)。

1. Sybase Central からターゲット・データベースに接続します。
2. 左ウィンドウ枠で、Adaptive Server Anywhere プラグインを選択します。
3. 右ウィンドウ枠で、[ユーティリティ] タブをクリックします。

   データベースに対して実行可能なすべての操作が右ウィンドウ枠に表示されます。
4. [データベースの移行] をダブルクリックします。

[データベース移行] ウィザードが表示されます。

[データベース移行] ウィザードを開くには、左ウィンドウ枠にあるターゲット・データベースを右クリックして、ポップアップ・メニューから [データベースの移行] を選択します。または、[ツール] － [Adaptive Server Anywhere 9] － [データベースの移行] を選択する方法もあります。

5 リストからターゲット・データベースを選択し、[次へ] をクリックします。

6 データの移行元のリモート・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、[新しいリモート・サーバの作成] をクリックして [リモート・サーバ作成] ウィザードを開きます。

リモート・サーバの作成の詳細については、「リモート・サーバの作成」751 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、Adaptive Server 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。

7 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。

8 ターゲット・データベースでテーブルを所有するユーザを選択し、[次へ] をクリックします。

ユーザをまだ作成していない場合は、[今すぐにユーザを作成] をクリックして [Mobile Link ユーザ作成] ウィザードを開きます。

9　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。

ターゲット・データベースがバージョン 8.0.0 以前の場合は、[外部キーの移行] オプションを使用できません。このオプションを使用する場合は、データベースをバージョン 8.0.1 以降にアップグレードしてください。

アップグレードの詳細については、『SQL Anywhere Studio 新機能ガイド』> 「データベースのアップグレード」を参照してください。

10　[完了] をクリックします。

sa_migrate ストアド・プロシージャの使用

sa_migrate ストアド・プロシージャを使用することにより、リモート・データを移行できます。テーブルをまったく修正しないのであれば、シングル・ステップによる方法を使用できます。それに対して、テーブルや外部キー・マッピングを削除するのであれば、拡張方法を使用します。

ストアド・プロシージャのセット sa_migrate を使用している場合は、必ず次の手順を実行してから、リモート・データベースをインポートします。

- 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については、「データベースの作成」37 ページを参照してください。

- 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。

  リモート・サーバの作成については、「リモート・サーバの作成」751 ページを参照してください。

- 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については、「外部ログインの作成」759 ページを参照してください。

- 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。

  ユーザの作成については、『ASA データベース管理ガイド』> 「新しいユーザの作成」を参照してください。

次の例は、sa_migrate ストアド・プロシージャを使用して、リモート・データベース上の 1 人の所有者に属するすべてのテーブルを 1 つの手順でインポートします。

sa_migrate を使用した 1 つのステップでのデータベースの移行

table_name パラメータと *owner_name* パラメータの両方に NULL を指定すると、データベース内のシステム・テーブルを含む、すべてのテーブルが移行します。また、リモート・データベースで、テーブルの所有者が異なっていても名前が同じ場合、それらのテーブルはターゲット・データベースに移行すると 1 人の所有者に属します。このため、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 (1 ステップの場合)。

1. Interactive SQL からターゲット・データベースに接続します。
2. Interactive SQL の [SQL 文] ウィンドウ枠で、sa_migrate ストアド・プロシージャを実行します。次に例を示します。

```
CALL sa_migrate( 'local_a', 'ase', NULL, l_smith,
NULL, 1, 1, 1 )
```

このプロシージャは複数のプロシージャを順番に呼び出し、指定された基準を使用してユーザ l_smith に属するリモート・テーブルをすべて移行します。

ターゲット・データベースに移行後、テーブルをすべて同じユーザによって所有されないようにする場合は、ターゲット・データベース上の所有者ごとに、*local_table_owner* 引数と *owner_name* 引数を指定して sa_migrate プロシージャを実行します。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。

詳細については、『ASA SQL リファレンス・マニュアル』> 「sa_migrate システム・プロシージャ」を参照してください。

バージョン 8.0.0 以前のターゲット・データベースでは、外部キーが自動的に移行されます。外部キーを移行しない場合は、データベース・ファイル・フォーマットをバージョン 8.0.1 以降にアップグレードしてください。

アップグレードの詳細については、『SQL Anywhere Studio 新機能ガイド』> 「データベースのアップグレード」を参照してください。

sa_migrate ストアド・プロシージャを使用した個別のテーブルの移行

❖ **リモート・テーブルをインポートするには、次の手順に従います (修正がある場合)。**

1. Interactive SQL からターゲット・データベースに接続します。
2. sa_migrate_create_remote_table_list ストアド・プロシージャを実行します。次に例を示します。

```
CALL sa_migrate_create_remote_table_list( 'ase',
      NULL, 'remote_a', 'mydb' )
```

リモート・サーバの名前を指定してください。

これにより、移行するリモート・テーブルのリストが dbo.migrate_remote_table_list テーブルに移植されます。このテーブルから、移行しないリモート・テーブルのローを削除できます。

table_name パラメータと *owner_name* パラメータの両方に NULL を指定しないでください。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。

sa_migrate_create_remote_table_list ストアド・プロシージャの詳細については、『ASA SQL リファレンス・マニュアル』> 「sa_migrate_create_remote_table_list システム・プロシージャ」を参照してください。

3 sa_migrate_create_tables ストアド・プロシージャを実行します。次に例を示します。

```
CALL sa_migrate_create_tables( 'local_a', )
```

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。

sa_migrate_create_tables ストアド・プロシージャの詳細については、『ASA SQL リファレンス・マニュアル』> 「sa_migrate_create_tables システム・プロシージャ」を参照してください。

4 リモート・テーブルからターゲット・データベース上のベース・テーブルにデータを移行する場合は、sa_migrate_data ストアド・プロシージャを実行します。次に例を示します。

次のストアド・プロシージャを入力します。

```
CALL sa_migrate_data( 'local_a' )
```

このプロシージャは、各リモート・テーブルから、sa_migrate_create_tables プロシージャで作成されたベース・テーブルにデータを移行します。

sa_migrate_data ストアド・プロシージャの詳細については、『ASA SQL リファレンス・マニュアル』> 「sa_migrate_data システム・プロシージャ」を参照してください。

リモート・データベースから外部キーを移行しない場合は、手順 7 に進みます。

5　sa_migrate_create_remote_fks_list ストアド・プロシージャを実行します。次に例を示します。

```
CALL sa_migrate_create_remote_fks_list( 'ase' )
```

このプロシージャは、dbo.migrate_remote_table_list にリストされている各リモート・テーブルに関連した外部キーのリストを、テーブル dbo.migrate_remote_fks_list に移植します。

再作成しない外部キー・マッピングを、ローカルのベース・テーブルから削除できます。

sa_migrate_create_remote_fks_list ストアド・プロシージャの詳細については、『ASA SQL リファレンス・マニュアル』> 「sa_migrate_create_remote_fks_list システム・プロシージャ」を参照してください。

6　sa_migrate_create_fks ストアド・プロシージャを実行します。次に例を示します。

```
CALL sa_migrate_create_fks( 'local_a' )
```

このプロシージャは、dbo.migrate_remote_fks_list に定義されている外部キー・マッピングをベース・テーブル上に作成します。

sa_migrate_create_fks ストアド・プロシージャの詳細については、『ASA SQL リファレンス・マニュアル』> 「sa_migrate_create_fks システム・プロシージャ」を参照してください。

7　移行用に作成されたプロキシ・テーブルを削除する場合は、sa_drop_proxy_tables ストアド・プロシージャを実行します。次に例を示します。

```
CALL sa_migrate_drop_proxy_tables( 'local_a' )
```

このプロシージャは、移行用に作成したプロキシ・テーブルをすべて削除し、移行プロセスを完了します。

sa_migrate_drop_proxy_tables ストアド・プロシージャの詳細については、『ASA SQL リファレンス・マニュアル』> 「sa_migrate_drop_proxy_tables システム・プロシージャ」を参照してください。

SQL コマンド・ファイルの使用

この項では、一連のコマンドで構成されるファイルの処理方法について説明します。**コマンド・ファイル**は、SQL 文を含むテキスト・ファイルであり、同じ SQL 文を繰り返し実行する場合に便利です。

コマンド・ファイルの作成

コマンド・ファイルは、好みのテキスト・エディタを使用して作成できます。実行される SQL 文にはコメント行を含めることができます。コマンド・ファイルは、一般的には、「**スクリプト**」ともいいます。

Interactive SQL で SQL コマンド・ファイルを開く

Windows プラットフォームでは、Interactive SQL を *.sql* ファイルのデフォルト・エディタに設定できます。設定して、ファイルをダブルクリックすると、そのファイルは Interactive SQL の [SQL 文] ウィンドウ枠に表示されます。

Interactive SQL を *.sql* ファイルのデフォルト・エディタに設定する方法の詳細については、『SQL Anywhere Studio ヘルプ』>「[オプション] ダイアログ : [一般] タブ」を参照してください。

SQL コマンド・ファイルの実行

コマンド・ファイルは次のいずれかの方法で実行できます。

- Interactive SQL の READ コマンドを使用する。たとえば、次の文はファイル *temp.sql* を実行します。

  ```
  READ temp.sql
  ```

  詳細については、『ASA SQL リファレンス・マニュアル』>「READ 文 [Interactive SQL]」を参照してください。

- コマンド・ファイルを [SQL 文] ウィンドウ枠にロードして、そこから直接実行する。

  コマンド・ファイルを再度 [SQL 文] ウィンドウ枠にロードするには、[ファイル] － [開く] を選択します。ファイル名の入力を要求された場合は、*temp.sql* と入力します。

- Interactive SQL のコマンド・ライン引数としてコマンド・ファイルを指定する。

SQL コマンド・ファイルの実行

SQL コマンド・ファイルの実行

コマンド・ファイルは、対話型またはバッチ・モードで実行できます。

❖ すぐにコマンド・ファイルを実行するには、次の手順に従います。

1. Interactive SQL で、[ファイル] ー [スクリプトの実行] を選択します。
2. [開く] ダイアログでファイルを検索し、選択します。選択したら、[開く] をクリックします。

   [スクリプトの実行] メニュー項目の機能は、READ 文と同じです。たとえば、[SQL 文] ウィンドウ枠で次のように入力しても、コマンド・ファイルを実行することが可能です。

   ```
   READ 'c:¥filename.sql'
   ```

   ここで、*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。

❖ [SQL 文] ウィンドウ枠を使用してコマンド・ファイルを実行するには、次の手順に従います。

- 次のコマンドを入力します。

  ```
  READ 'c:¥filename.sql'
  ```

  ここで、*c:¥filename.sql* には、ファイルのパス、名前、拡張子を指定します。パスにスペースが含まれている場合にのみ、一重引用符 (例文参照) が必要です。

❖ コマンド・ファイルをバッチ・モードで実行するには、次の手順に従います。

1. コマンド・プロンプトを開きます。

2　コマンド・ファイルを、Interactive SQL のコマンド・ライン引数として指定します。

たとえば、次のコマンドは、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= ASA 9.0 Sample" myscript.sql
```

SQL コマンド・ファイルのロード

新しい Interactive SQL セッションを開始するとき、コマンド・ファイルの内容を [SQL 文] ウィンドウ枠にロードすることも、そのコマンドをすぐに実行することもできます。

❖ **ファイルのコマンドを [SQL 文] ウィンドウ枠にロードするには、次の手順に従います。**

1　[ファイル] － [開く] を選択します。

2　[開く] ダイアログでファイルを検索し、選択します。

3　選択したら、[開く] をクリックします。

ファイルへのデータベース出力の書き込み

Interactive SQL では、各コマンドの結果セット・データは次のコマンドが実行されると [結果] ウィンドウ枠の [結果] タブから消えます。データの記録を残すには、個々の文を別ファイルに出力して保存します。たとえば、2 つの SELECT 文 *statement1* と *statement2* がある場合、次のようにしてそれぞれを *file1* と *file2* に出力します。

statement1; OUTPUT TO *file1*;
　statement2; OUTPUT TO *file2*;

たとえば、次のコマンドはクエリの結果を *Employees.txt* という名前のファイルに保存します。

```
SELECT * FROM EMPLOYEE;
 OUTPUT TO 'C:¥¥My Documents¥¥Employees.txt'
```

詳細については、『ASA SQL リファレンス・マニュアル』>「OUTPUT 文 [Interactive SQL]」を参照してください。

Adaptive Server Enterprise の互換性

BCP FORMAT 句を使用して、Adaptive Server Anywhere と Adaptive Server Enterprise 間でファイルのインポートとエクスポートを実行できます。BCP 出力はデリミタで区切られた ASCII フォーマットです。Adaptive Server Enterprise で使用するために Adaptive Server 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、『ASA SQL リファレンス・マニュアル』> 「LOAD TABLE 文」または 『ASA SQL リファレンス・マニュアル』> 「UNLOAD TABLE 文」を参照してください。

第 18 章

リモート・データへのアクセス

この章の内容

Adaptive Server Anywhere は、iAnywhere でも iAnywhere 以外でも、ローカル・サーバに格納されているデータと同じように、異なるサーバにあるデータにアクセスできます。

この章では、リモート・データにアクセスするための Adaptive Server Anywhere の設定について説明します。

概要

Adaptive Server 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、データを Adaptive Server 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。ただし、複数のデータベース間のクエリのパフォーマンスは、すべてのデータが単一の Adaptive Server 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。

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 Adaptive Server Anywhere を使用する。
- Sybase、Oracle、DB2 などのリレーショナル・データベースのデータにアクセスする。
- Excel スプレッドシート、MS-Access データベース、FoxPro、テキスト・ファイルなどのデスクトップ・データにアクセスする。
- ODBC インタフェースをサポートするその他のデータ・ソースにアクセスする。
- ローカル・データとリモート・データ間でジョインを実行する。ただしパフォーマンスは、すべてのデータが単一の Adaptive Server 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 Adaptive Server 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 Adaptive Server Anywhere の機能を持たないデータ・ソースに対して、Adaptive Server Anywhere の機能を使用する。たとえば、Oracle に格納されているデータに対して Java の機能を使用したり、スプレッドシートでサブクエリをしたりできます。データを取り出してから操作することによって、リモート・データ・ソースにサポートされていない機能を Adaptive Server 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。

Adaptive Server 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。

- Adaptive Server Anywhere
- Adaptive Server Enterprise
- Oracle
- IBM DB2
- Microsoft SQL Server
- その他の ODBC データ・ソース

利用可能なプラットフォームについては、『SQL Anywhere Studio の紹介』> 「Windows オペレーティング・システムと NetWare オペレーティング・システム」を参照してください。

PowerBuilder DataWindows からのリモート・データへのアクセス

接続時に DBParm Block パラメータを 1 に設定することによって、PowerBuilder DataWindow からリモート・データにアクセスできます。

- 設計環境で、[データベース プロファイル セットアップ] ダイアログの [トランザクション] タブを開いて [Retrieve ブロック化係数] を 1 に設定することによって、Block パラメータを設定できます。
- 接続文字列で、次のフレーズを使用します。

```
DBParm="Block=1"
```

リモート・データ・アクセスの基本概念

この項では、リモート・データのアクセスに必要な基本概念について説明します。

リモート・テーブルのマッピング

クライアント・アプリケーション側から見ると、接続している Adaptive Server 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。

   詳細については、「リモート・サーバの使用」751 ページを参照してください。

2. ローカル・サーバのログイン情報とリモート・サーバのログイン情報とが異なる場合は、ローカル・ユーザのログイン情報をリモート・サーバのユーザのログイン情報にマップします。

   詳細については、「外部ログインの使用」759 ページを参照してください。

3. プロキシ・テーブルの定義を作成します。これによってローカルのプロキシ・テーブルからリモート・テーブルへのマッピングを指定します。リモート・テーブルが置かれているサーバ、リモート・テーブルのデータベース名、所有者名、テーブル名、カラム名を指定します。

詳細については、「プロキシ・テーブルの編集」762 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、または Interactive SQL などのツールを使用して SQL 文を直接実行することもできます。

サーバ・クラス

「**サーバ・クラス**」は各リモート・サーバに割り当てられます。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。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。Adaptive Server 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。Adaptive Server 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。

現在のところ、サーバ・クラスには 2 つのグループがあります。1 つは JDBC ベース、もう 1 つは ODBC ベースです。

JDBC ベースのサーバ・クラスを次に示します。

- **asajdbc**　Adaptive Server Anywhere (バージョン 6 以降)
- **asejdbc**　Adaptive Server Enterprise と SQL Server (バージョン 10 以降)

ODBC ベースのサーバ・クラスを次に示します。

- **asaodbc**　Adaptive Server Anywhere (バージョン 5.5 以降)
- **aseodbc**　Adaptive Server Enterprise と SQL Server (バージョン 10 以降)
- **db2odbc**　IBM DB2
- **mssodbc**　Microsoft SQL Server
- **oraodbc**　Oracle サーバ (バージョン 8.0 以降)
- **odbc**　その他の ODBC データ・ソース

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」787 ページを参照してください。

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、SYSSERVERS テーブルにエントリが追加されます。この項では、リモート・サーバ定義を作成、変更、削除する方法について説明します。

リモート・サーバの作成

CREATE SERVER 文を使用して、リモート・サーバ定義を設定します。この文を直接実行するか、または Sybase Central を使用します。

ODBC 接続では、各リモート・サーバは ODBC データ・ソースに対応します。Adaptive Server 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。

サーバを作成するには、RESOURCE 権限が必要です。

また UNIX プラットフォームでは、ODBC ドライバ・マネージャを参照する必要があります。

CREATE SERVER 文の詳細については、『ASA SQL リファレンス・マニュアル』> 「CREATE SERVER 文」を参照してください。

例 1

次の文は、ASEserver という Adaptive Server Enterprise サーバのエントリを SYSSERVERS テーブルに作成します。

```
CREATE SERVER ASEserver
 CLASS 'ASEJDBC'
 USING 'rimu:6666'
```

文中の各項目を次に説明します。

- **ASEserver**　リモート・サーバの名前
- **ASEJDBC**　サーバが Adaptive Server Enterprise であり、そのサーバへの接続には JDBC が使われることを示すキーワード

- **rimu:6666** リモート・サーバが置かれているマシンの名前と TCP/IP ポート番号

例 2

次の文は、testasa という ODBC ベースの Adaptive Server Anywhere サーバのエントリを SYSSERVERS テーブルに作成します。

```
CREATE SERVER testasa
 CLASS 'ASAODBC'
 USING 'test4'
```

文中の各項目を次に説明します。

- **testasa** このデータベース内で識別するリモート・サーバの名前
- **ASAODBC** サーバが Adaptive Server 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

例 3

次の文は、Unix プラットフォームで remasa という ODBC ベースの Adaptive Server Anywhere サーバのエントリを SYSSERVERS テーブルに作成します。

```
CREATE SERVER remasa
 CLASS 'asaodbc'
 USING 'driver=/opt/sybase/SYBSsa9/lib/
dbodbc9_r.so;dsn=my_asa_dsn'
```

文中の各項目を次に説明します。

- **remasa** このデータベース内で識別するリモート・サーバの名前
- **ASAODBC** サーバが Adaptive Server 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 remase のエントリを sysservers テーブルに作成します。

```
CREATE SERVER remase
 CLASS 'aseodbc'
 USING 'driver=/opt/sybase/SYBSsa9/drivers/lib/
libodbc.so;dsn=my_ase_dsn'
```

文中の各項目を次に説明します。

- **remase**　このデータベース内で識別するリモート・サーバの名前
- **ASEODBC**　サーバが Adaptive Server Enterprise であり、そのサーバへの接続に ODBC が使われることを示すキーワード
- **USING**　ODBC ドライバ・マネージャへの参照

Sybase Central を使用したリモート・サーバの作成

リモート・サーバを、Sybase Central のウィザードを使用して作成できます。詳細については、「リモート・サーバの作成」751 ページを参照してください。

❖ リモート・サーバを作成するには、次の手順に従います (Sybase Central の場合)。

1. Sybase Central からホスト・データベースに接続します。
2. データベースの [リモート・サーバ] フォルダを開きます。
3. [ファイル] － [新規] － [リモート・サーバ] を選択します。

   [リモート・サーバ作成] ウィザードが表示されます。
4. ウィザードの最初のページで、リモート・サーバに使用する名前を入力します。この名前は、ローカル・データベース内でリモート・サーバを識別するためのものです。サーバが提供する名前に一致させる必要はありません。[次へ] をクリックします。
5. 該当するサーバのタイプを選択して、[次へ] をクリックします。

6 データ・アクセス・メソッド (ODBC か JDBC) を選択して接続情報を指定します。[次へ] をクリックします。

- ODBC の場合は、データ・ソース名を指定します。
- JDBC の場合は、*machine-name*:*port-number* のフォームで URL を指定します。

データ・アクセス・メソッド (JDBC か ODBC) は、Adaptive Server Anywhere がリモート・データベースへのアクセスに使用するためのものです。Sybase Central が Adaptive Server Anywhere に接続する方法をこれで決めているわけではありません。

7 リモート・サーバを読み込み専用にするかどうかを指定します。[次へ] をクリックします。

8 リモート・データベース用の外部ログインを作成します。

デフォルトでは、Adaptive Server 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。

9 オプションとして、リモート・サーバ定義が作成される前に [接続のテスト] をクリックして、リモート・サーバに接続できるかどうかをテストできます。

[完了] をクリックして、リモート・サーバ定義を作成します。

リモート・サーバの削除

Sybase Central または DROP SERVER 文を使用すると、Adaptive Server Anywhere システム・テーブルからリモート・サーバを削除できます。このアクションを実行するには、このサーバ上に定義されているすべてのリモート・テーブルが、すでに削除されていなければなりません。

リモート・サーバを削除するには、DBA 権限が必要です。

❖ リモート・サーバを削除するには、次の手順に従います (Sybase Central の場合)。

1. Sybase Central からホスト・データベースに接続します。
2. [リモート・サーバ] フォルダを開きます。
3. 削除するリモート・サーバを右クリックし、ポップアップ・メニューから [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. Interactive SQL からホスト・データベースに接続します。
2. DROP SERVER 文を実行します。

詳細については、『ASA SQL リファレンス・マニュアル』> 「DROP SERVER 文」を参照してください。

例

次の文は testasa という名前のサーバを削除します。

```
DROP SERVER testasa
```

リモート・サーバの変更

Sybase Central または ALTER SERVER 文を使用すると、サーバの属性を変更できます。変更された設定は、次回リモート・サーバに接続する際に有効になります。

サーバを変更するには、RESOURCE 権限が必要です。

❖ リモート・サーバのプロパティを変更するには、次の手順に従います (Sybase Central の場合)。

1. Sybase Central からホスト・データベースに接続します。
2. データベースの [リモート・サーバ] フォルダを開きます。
3. リモート・サーバを右クリックし、ポップアップ・メニューから [プロパティ] を選択します。
4. リモート・サーバのさまざまなプロパティを設定します。

❖ 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. Interactive SQL からホスト・データベースに接続します。
2. ALTER SERVER 文を実行します。

例

次の文は、ASEserver 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 ASEserver です。

```
ALTER SERVER ASEserver
 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。

詳細については、『ASA SQL リファレンス・マニュアル』> 「ALTER SERVER 文」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

Adaptive Server Anywhere を設定するときに、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できると便利です。sp_remote_tables プロシージャは、サーバ上のテーブルのリストを返します。

```
sp_remote_tables servername
                  [,tablename]
                  [, owner ]
                  [, database]
```

tablename、*owner*、または *database* を指定すると、テーブルのリストはその指定に当てはまるものだけに限定されます。

たとえば、リモート・サーバから使用可能なすべての Microsoft Excel シートの中から、excel という名前のもののリストを取得するには、次のようにします。

```
sp_remote_tables  excel
```

'fred' が所有する、asetest という名前の ASE の production データベースにあ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。

```
sp_remote_tables  asetest, null, fred, production
```

詳細については、『ASA SQL リファレンス・マニュアル』>「sp_remote_tables システム・プロシージャ」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 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。Adaptive Server Anywhere はこの機能情報から、どのくらいの SQL 文をリモート・サーバに渡すことができるかを判断します。

サーバの機能を保持するシステム・テーブルは、Adaptive Server Anywhere がリモート・サーバに最初に接続した後でないと、設定されません。この情報は SYSCAPABILITY と SYSCAPABILITYNAME システム・テーブルから取得されます。指定したサーバ名は、CREATE SERVER 文で使用するサーバ名と同じでなければなりません。

次のように、sp_servercaps ストアド・プロシージャを発行します。

```
sp_servercaps  servername
```

詳細については、『ASA SQL リファレンス・マニュアル』>「sp_servercaps システム・プロシージャ」を参照してください。

外部ログインの使用

デフォルトでは、Adaptive Server 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、『ASA データベース管理ガイド』> 「統合化ログインの使用方法」を参照してください。

外部ログインの作成

外部ログインは、Sybase Central または CREATE EXTERNLOGIN 文を使用して作成できます。

ログイン名と DBA アカウントだけが、外部ログインを追加または変更できます。

❖ 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. Sybase Central からホスト・データベースに接続します。
2. データベースの [リモート・サーバ] フォルダを開いて、リモート・サーバを選択します。
3. [ファイル] － [新規] － [外部ログイン] を選択します。

   [外部ログイン作成] ウィザードが表示されます。
4. 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. Interactive SQL からホスト・データベースに接続します。

2 CREATE EXTERNLOGIN 文を実行します。

例

次の文によって、ローカル・ユーザ **fred** は、**banana** というパスワードとリモート・ログイン **frederick** を使用してサーバ **ASEserver** へアクセスします。

```
CREATE EXTERNLOGIN fred
 TO ASEserver
 REMOTE LOGIN frederick
 IDENTIFIED BY banana
```

詳細については、『ASA SQL リファレンス・マニュアル』>「CREATE EXTERNLOGIN 文」を参照してください。

外部ログインの削除

Sybase Central または DROP EXTERNLOGIN 文を使用すると、Adaptive Server Anywhere システム・テーブルから外部ログインを削除できます。

ログイン名と DBA アカウントだけが、外部ログインを削除できます。

❖ 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 Sybase Central からホスト・データベースに接続します。

2 [リモート・サーバ] フォルダを開きます。

3 左ウィンドウ枠でリモート・サーバを選択し、次に右ウィンドウ枠で [外部ログイン] タブをクリックします。

4 外部ログインを右クリックし、ポップアップ・メニューから [削除] を選択します。

❖ 外部ログインを削除するには、次の手順に従います (SQL の場合)。

1 Interactive SQL からホスト・データベースに接続します。

2　DROP EXTERNLOGIN 文を実行します。

例

次の文は、前述の例で作成したローカル・ユーザ fred の外部ログインを削除します。

```
DROP EXTERNLOGIN fred TO ASEserver
```

参照

- 『ASA SQL リファレンス・マニュアル』> 「DROP EXTERNLOGIN 文」

プロキシ・テーブルの編集

リモート・オブジェクトに対応するローカルの「**プロキシ・テーブル**」を作成すると、意識することなくリモート・データにアクセスできるようになります。プロキシ・テーブルを作成するには、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。

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、その場合でもピリオドは必要です。

Adaptive Server Enterprise では、*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。

Adaptive Server Anywhere では、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。

Excel、Lotus Notes、Access では、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**　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。テーブル名を入力しない場合、リモート・テーブル名はローカルのプロキシ・テーブル名と同じであるとみなされます。

例

以下はロケーション文字列の使用例です。

- Adaptive Server Anywhere：

```
'testasa..DBA.employee'
```

- Adaptive Server Enterprise：

```
'ASEServer.pubs2.dbo.publishers'
```

- Excel：

```
'excel;d:¥pcdb¥quarter3.xls;;sheet1$'
```

- Access：

```
'access;¥¥server1¥production¥inventory.mdb;;parts'
```

プロキシ・テーブルの作成 (Sybase Central の場合)

プロキシ・テーブルは、Sybase Central または CREATE EXISTING TABLE 文を使用して作成できます。システム・テーブル用のプロキシ・テーブルは作成できません。

CREATE EXISTING TABLE 文は、リモート・サーバ上にある既存のテーブルにマッピングするプロキシ・テーブルを作成します。Adaptive Server 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、『ASA SQL リファレンス・マニュアル』> 「CREATE EXISTING TABLE 文」を参照してください。

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

1. Sybase Central からホスト・データベースに接続します。
2. 左ウィンドウ枠で [リモート・サーバ] フォルダを開きます。
3. 右ウィンドウ枠で、[プロキシ・テーブル] タブをクリックします。
4. [ファイル] － [新規] － [プロキシ・テーブル] を選択します。
5. ウィザードの指示に従います。

ヒント
左ウィンドウ枠でリモート・サーバを選択すると、対応するプロキシ・テーブルが、右ウィンドウ枠の [プロキシ・テーブル] タブ上に表示されます。また、[テーブル] フォルダにも表示されます。これはアイコンの P という文字によって、他のテーブルと区別されます。

CREATE EXISTING TABLE 文を使用したプロキシ・テーブルの作成

CREATE EXISTING TABLE 文は、リモート・サーバ上にある既存のテーブルにマッピングするプロキシ・テーブルを作成します。Adaptive Server Anywhere は、リモート・ロケーションのオブジェクトからカラム属性とインデックス情報を導出します。

❖ CREATE EXISTING TABLE 文を使用してプロキシ・テーブルを作成するには、次の手順に従います (SQL の場合)。

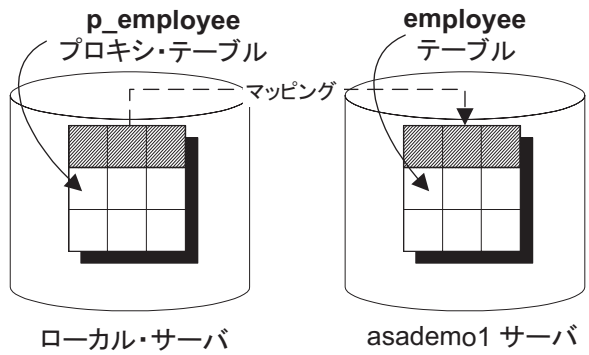
1. ホスト・データベースに接続します。
2. CREATE EXISTING TABLE 文を実行します。

詳細については、『ASA SQL リファレンス・マニュアル』>「CREATE EXISTING TABLE 文」を参照してください。

例 1

サーバ asademo1 のリモート・テーブル employee に対して、ローカル・サーバ上に p_employee 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_employee
 AT 'asademo1..DBA.employee'
```



例 2

次の文は、プロキシ・テーブル a1 を Microsoft Access ファイルの mydbfile.mdb にマッピングします。この例では、AT キーワードにおいてセミコロン (;) 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。Adaptive Server Anywhere データ型を使用して、CREATE TABLE 文を入力します。Adaptive Server 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、CREATE EXISTING TABLE 文で作成されたプロキシ・テーブルを DROP TABLE 文で削除してもリモート・テーブルは削除されませんので、リモート・テーブルを削除したくない場合はそうしてください。

詳細については、『ASA SQL リファレンス・マニュアル』>「CREATE TABLE 文」を参照してください。

❖ CREATE TABLE 文を使用してプロキシ・テーブルを作成するには、次の手順に従います (SQL の場合)。

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

詳細については、『ASA SQL リファレンス・マニュアル』>「CREATE TABLE 文」を参照してください。

例

次の文は、リモート・サーバ asademo1 に employee というテーブルを作成し、リモートのロケーションにマップする members というプロキシ・テーブルを作成します。

```
CREATE TABLE members
 ( membership_id INTEGER NOT NULL,
 member_name CHAR(30) NOT NULL,
 office_held CHAR( 20 ) NULL)
 AT 'asademo1..DBA.employee'
```

リモート・テーブルのカラムのリスト

CREATE EXISTING 文で、カラム・リストを指定しようとしている場合は、リモート・テーブルで使用可能なカラムのリストを取得できると便利です。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 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。

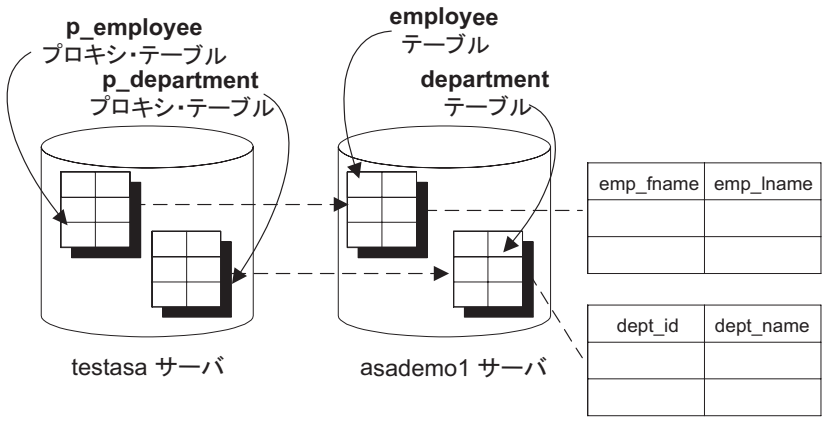
たとえば、asetest という名前の Adaptive Server Enterprise サーバの production データベースにある sysobjects テーブルのカラムのリストを取得するには、次のように指定します。

```
sp_remote_columns  asetest, sysobjects, null,
production
```

詳細については、『ASA SQL リファレンス・マニュアル』>「sp_remote_columns システム・プロシージャ」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・サーバ testasa にマッピングされるサンプル・データベースにあるリモート Adaptive Server Anywhere のテーブル employee と department を示しています。



現実問題として、異なる Adaptive Server Anywhere データベース上のテーブル間でジョインを使用する場合があるかもしれません。ここでは、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、そこからサンプル・データベースにアクセスするためだけに使用します。

2. データベース・サーバを起動して、empty.db とサンプル・データベースの両方を稼働させます。これを実行するには、次のコマンド・ラインをインストール・ディレクトリから実行します。

```
dbeng9 asademo empty
```

3 ユーザ ID **DBA** とパスワード **SQL** を使って、Interactive SQL から *empty.db* に接続します。

4 新しいデータベースで、testasa という名前のリモート・サーバを作成します。サーバ・クラスは asaodbc、接続情報は **'ASA 9.0 Sample'** です。

```
CREATE SERVER testasa
CLASS 'asaodbc'
USING 'ASA 9.0 Sample'
```

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。

6 employee プロキシ・テーブルを定義します。

```
CREATE EXISTING TABLE employee
AT 'testasa..DBA.employee'
```

7 department プロキシ・テーブルを定義します。

```
CREATE EXISTING TABLE department
AT 'testasa..DBA.department'
```

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 emp_fname, emp_lname, dept_name
FROM employee JOIN department
ON employee.dept_id = department.dept_id
ORDER BY emp_lname
```

複数のローカル・データベースのテーブルのジョイン

Adaptive Server 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 Adaptive Server Anywhere データベース内のテーブルをリモート・テーブルとして定義することによって、データベース間のジョインを実行できます。

複数のデータベースの指定の詳細については、「CREATE SERVER 文の USING パラメータ値」790 ページを参照してください。

例

たとえば、データベース db1 を使用していて、データベース db2 にあるテーブルのデータにアクセスする場合は、データベース db2 にあるテーブルを示すプロキシ・テーブルの定義を設定します。testasa という名前の Adaptive Server Anywhere 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。

- 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。
- ジョインの実行元とな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。
- 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 を実行します。これによって、Adaptive Server Anywhere サーバへの「**ループバック**」接続が設定されます。

```
CREATE SERVER local_db2
 CLASS 'asaodbc'
 USING 'testasa_db2'
 CREATE SERVER local_db3
 CLASS 'asaodbc'
 USING     'testasa_db3'
```

JDBC を使用する場合は次のようになります。

```
CREATE SERVER local_db2
 CLASS 'asajdbc'
 USING 'mypc1:2638/db2'
 CREATE SERVER local_db3
 CLASS 'asajdbc'
 USING 'mypc1:2638/db3'
```

- アクセスする他のデータベースにあるテーブルに CREATE EXISTING を使用して、プロキシ・テーブルの定義を作成します。

```
CREATE EXISTING TABLE employee
 AT 'local_db2...employee'
```

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。
- Adaptive Server 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。

指定したサーバに接続できない場合、その理由を説明したメッセージがユーザに送信されます。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。

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、Adaptive Server 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。

詳細については、『ASA SQL リファレンス・マニュアル』>「FORWARD TO 文」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、ASEserver というサーバへの接続を検証します。

```
FORWARD TO ASEserver {SELECT @@version}
```

例 2

次の文は、サーバ ASEserver とのパススルー・セッションを示します。

```
FORWARD TO ASEserver
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

Adaptive Server 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、Sybase Adaptive Server Anywhere、Sybase 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。

リモート・プロシージャの作成

リモート・プロシージャ・コールは、Sybase Central または CREATE PROCEDURE 文を使用して発行できます。

リモート・プロシージャを作成するには、DBA 権限が必要です。

❖ リモート・プロシージャ・コールを発行するには、次の手順に従います (Sybase Central の場合)。

1. Sybase Central からホスト・データベースに接続します。
2. [リモート・サーバ] フォルダを開きます。
3. 左ウィンドウ枠で、リモート・プロシージャを作成するリモート・サーバを選択します。
4. 右ウィンドウ枠で、[リモート・プロシージャ] タブをクリックします。
5. [ファイル] － [新規] － [リモート・プロシージャ] を選択します。

   [リモート・プロシージャ作成] ウィザードが表示されます。
6. ウィザードの指示に従います。

❖ **リモート・プロシージャ・コールを発行するには、次の手順に従います (SQL の場合)。**

1. まず、Adaptive Server 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 文を使用してコール本体を作成するのではなく、プロシージャの実体が存在するロケーションを定義するロケーション文字列を指定する点が異なります。

```
CREATE PROCEDURE remotewho ()
 AT 'bostonase.master.dbo.sp_who'
```

2. 次のようにプロシージャを実行します。

```
call remotewho()
```

詳細については、『ASA SQL リファレンス・マニュアル』>「CREATE PROCEDURE 文」を参照してください。

例

パラメータを指定した例を次に示します。

```
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。その他のデータ型は使用できません。

- [UNSIGNED] SMALLINT
- [UNSIGNED] INT
- [UNSIGNED] BIGINT
- TINYINT
- REAL
- DOUBLE
- CHAR

- BIT

データ型 NUMERIC と DECIMAL は、IN パラメータには指定できますが、OUT パラメータと INOUT パラメータには指定できません。

リモート・プロシージャの削除

Sybase Central または DROP PROCEDURE 文を使用すると、リモート・プロシージャを削除できます。

リモート・プロシージャを削除するには、DBA 権限が必要です。

❖ リモート・プロシージャを削除するには、次の手順に従います (Sybase Central の場合)。

1. [リモート・サーバ] フォルダを開きます。
2. 左ウィンドウ枠でリモート・サーバを選択します。
3. 右ウィンドウ枠で、[リモート・プロシージャ] タブをクリックします。
4. [リモート・プロシージャ] タブで、リモート・プロシージャを右クリックし、ポップアップ・メニューから [削除] を選択します。

❖ リモート・プロシージャを削除するには、次の手順に従います (SQL の場合)。

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

詳細については、『ASA SQL リファレンス・マニュアル』> 「DROP 文」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、Adaptive Server 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」133 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。Adaptive Server 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. Adaptive Server 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、Adaptive Server 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 Adaptive Server 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、解析されます。有効な Adaptive Server Anywhere SQL 文でないと、エラーが発生します。

クエリの正規化

次の手順はクエリの正規化と呼ばれます。この手順の間に、参照され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、Adaptive Server Anywhere がリモート・サーバに引き渡す実際の SQL 文の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。

Adaptive Server Anywhere は、定義されているリモート・サーバそれぞれに対してそのサーバの各種の機能を持ちます。これらの機能情報は syscapabilities システム・テーブルに格納されます。機能情報はリモート・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。

あるリモート・サーバに対する syscapabilities データが初期化されると、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。これによって、ユーザはサーバの既知の機能を変更できます。

リモート・サーバは指定された SQL 文のすべての機能をサポートしているとはかぎらないので、Adaptive Server Anywhere は、クエリがリモート・サーバに対応できるようになるまで、文を単純なコンポーネントに分割しなければなりません。リモート・サーバに渡されない SQL 機能は、Adaptive Server 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。Adaptive Server 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 Adaptive Server 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

通常、1 つの文を処理する最も効果的な方法は、作業に関わるリモート・サーバに元の文のできるだけ多くの部分を渡すことです。Adaptive Server Anywhere は、文のできるだけ多くの部分を渡そうとします。多くの場合、これは元々 Adaptive Server Anywhere に指定されたとおりの、完全な文になります。

Adaptive Server Anywhere は、次のような場合に完全な文を渡します。

- 文中のテーブルがどれも同じリモート・サーバに常駐する。
- リモート・サーバが文中のすべての構文を処理できる。

まれに、リモート・サーバに渡すより、Adaptive Server 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、Adaptive Server 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、『ASA SQL リファレンス・マニュアル』> 「ALTER SERVER 文」を参照してください。

文の部分的なパススルー

ある文に複数のサーバへの参照が含まれている場合、またはリモート・サーバにサポートされていない SQL 機能を文が使用している場合は、クエリは複数の単純な部分に分解されます。

Select

引き渡すことのできない部分が取り除かれながら、SELECT 文は分解されていきます。取り除かれた部分の機能は Adaptive Server Anywhere によって実行されます。たとえば、次の文中にある atan2() 関数をリモート・サーバが処理できないとします。

```
select a,b,c where atan2(b,10) > 3 and c = 10
```

リモート・サーバに送信される文は、次のように変換されます。

```
select a,b,c where c = 10
```

ローカルでは、Adaptive Server Anywhere が "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。

更新と削除

Adaptive Server 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、条件に合うローが検出されたときに、位置付け UPDATE または DELETE "where current of cursor" 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、Adaptive Server 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 または DELETE を処理するには、位置付け UPDATE または DELETE ("where current of cursor") 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

Adaptive Server Anywhere の今回リリースでは、Adaptive Server Anywhere に中間テンポラリ・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・サーバのトラブルシューティングのヒントをいくつか説明します。

リモート・データにサポートされていない機能

次の Adaptive Server Anywhere 機能はリモート・データではサポートされていません。これらの機能を使用しようとすると、問題が発生します。

- リモート・テーブルに対する ALTER TABLE 文
- プロキシ・テーブルで定義されたトリガは起動しない。
- SQL Remote
- リモート・テーブルを参照する外部キーは無視される。
- READTEXT 関数、WRITETEXT 関数、TEXTPTR 関数
- 位置付け UPDATE と DELETE
- 中間テンポラリ・テーブルを必要とする UPDATE と DELETE
- リモート・データに対してオープン・カーソルの後方スクロール。フェッチ文は NEXT または RELATIVE 1 でなければなりません。
- リモート・テーブルのカラムにリモート・サーバに関するキーワードがある場合、そのカラムのデータにはアクセスできない。Adaptive Server Anywhere はリモート・サーバの予約語をすべて認識できるわけではありません。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

Adaptive Server 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、Adaptive Server 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 Adaptive Server 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。

接続の問題点

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、Adaptive Server 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 testasa {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。

  ```
  SET OPTION CIS_OPTION = 7
  ```

  リモート・トレーシングを有効にすると、トレーシング情報がデータベース・サーバ・ウィンドウに表示されます。データベース・サーバの起動時に - o *filename* サーバ・オプションを指定して、この出力をファイルに記録できます。

  CIS_OPTION オプションの詳細については、『ASA データベース管理ガイド』> 「CIS_OPTION オプション [データベース]」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、『ASA データベース管理ガイド』> 「-o サーバ・オプション」を参照してください。

クエリに関する一般的な問題

リモート・テーブルに対して Adaptive Server Anywhere がクエリを処理する方法に特定の種類の問題が発生する場合は、Adaptive Server Anywhere がどのようにそのクエリを実行するかを理解すると役に立ちます。次のようにして、クエリの実行プランだけでなく、リモート・トレーシングも表示できます。

```
SET OPTION CIS_OPTION = 7
```

リモート・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、『ASA データベース管理ガイド』> 「CIS_OPTION オプション [データベース]」を参照してください。

リモート・トレーシングを有効にすると、トレーシング情報がデータベース・サーバ・ウィンドウに表示されます。データベース・サーバの起動時に - o *filename* サーバ・オプションを指定して、この出力をファイルに記録できます。

-o サーバ・オプションの詳細については、『ASA データベース管理ガイド』> 「-o サーバ・オプション」を参照してください。

クエリ上でブロックされるクエリ

1 つの Adaptive Server Anywhere サーバ上で複数のデータベースにアクセスする場合、-gx コマンド・ライン・スイッチを使用して Windows 上でデータベース・サーバが使用するスレッドの数を増やすことが必要な場合があります。

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。

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。リモート要求をキャンセルする 1 つの方法として、この名前を使用して接続を削除します。

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。

第 19 章

リモート・データ・アクセスのサーバ・クラス

この章の内容

この章では、Adaptive Server Anywhere によるサーバの異なるクラスとのインタフェースの方法について説明します。内容は次のとおりです。

- 各サーバ・クラスがサポートするサーバのタイプ
- 各サーバ・クラスの CREATE SERVER 文の USING 句の値
- 特別な設定要件

概要

CREATE SERVER 文に指定するサーバ・クラスは、リモート接続の動作を決定します。サーバ・クラスは、サーバの機能の詳細な情報を Adaptive Server Anywhere に提供します。Adaptive Server Anywhere は、サーバの機能に合わせて SQL 文をフォーマットします。

サーバ・クラスには、2 つのカテゴリがあります。

- JDBC ベースのサーバ・クラス
- ODBC ベースのサーバ・クラス

各サーバ・クラスには各種のユニークな特性があります。リモート・データ・アクセス用にサーバを設定するには、データベース管理者とプログラマがこの特性を知っておく必要があります。

この章を読むときは、サーバ・クラスのカテゴリ (JDBC ベースか ODBC ベース) 全般について述べている項と、個々のサーバ・クラスに固有の項の両方を参照してください。

JDBC ベースのサーバ・クラス

JDBC ベースのサーバ・クラスは、Adaptive Server Anywhere が内部的に Java 仮想マシンと jConnect 4.0 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。

- **asajdbc**　Adaptive Server Anywhere (バージョン 6 以降)
- **asejdbc**　Adaptive Server Enterprise と SQL Server (バージョン 10 以降)

NetWare を使用している場合は、asajdbc クラスのみがサポートされます。

JDBC クラスの設定上の注意

JDBC ベースのクラスで定義されたリモート・サーバにアクセスするときは、次のことに注意してください。

- ローカル・データベースは、Java を使用できるようになっていなければなりません。

  詳細については、『ASA プログラミング・ガイド』> 「データベースを Java 実行可能にする」を参照してください。

- jConnect をロードして実行するには、Java 仮想マシンにはデフォルトのメモリよりも多くのメモリが必要です。これらのメモリ・オプションを、少なくとも次の値に設定してください。

```
SET OPTION PUBLIC.JAVA_NAMESPACE_SIZE = 3000000
 SET OPTION PUBLIC.JAVA_HEAP_SIZE = 1000000
```

- jConnect 4.0 は Adaptive Server Anywhere をインストールすると自動的にインストールされるので、追加のドライバをインストールする必要はありません。
- 最適なパフォーマンスを得るには、iAnywhere では ODBC ベースのクラス (asaodbc か aseodbc) をおすすめします。

- asejdbc サーバ・クラスまたは asajdbc 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 4.x ベースのクライアントを処理できるよう設定してください。jConnect 設定スクリプトは、Adaptive Server Anywhere の場合は *SQL_anywhere.sql*、Adaptive Server Enterprise の場合は *sql_server.sql* です。これらのスクリプトを、使用するリモート・サーバに対して実行します。

サーバ・クラス asajdbc

サーバ・クラス asajdbc のサーバは、Adaptive Server Anywhere (バージョン 6 以降) です。Adaptive Server Anywhere データ・ソースの設定には、特別な要件はありません。

CREATE SERVER 文の USING パラメータ値

アクセスする Adaptive Server Anywhere データベースごとに、個別の CREATE SERVER を実行してください。たとえば、testasa という名前の Adaptive Server Anywhere サーバが 'banana' というマシンで稼働していて、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 Adaptive Server Anywhere を設定します。

```
CREATE SERVER testasadb1
 CLASS 'asajdbc'
 USING 'banana:2638/db1'
 CREATE SERVER testasadb2
 CLASS 'asajdbc'
 USING 'banana:2638/db2'
 CREATE SERVER testasadb2
 CLASS 'asajdbc'
 USING 'banana:2638/db3'
```

/databasename 値を指定しないと、リモート接続はリモートの Adaptive Server Anywhere のデフォルト・データベースを使用します。

CREATE SERVER 文の詳細については、『ASA SQL リファレンス・マニュアル』> 「CREATE SERVER 文」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 Adaptive Server Enterprise と SQL Server (バージョン 10 以降) です。Adaptive Server Enterprise データ・ソースの設定には、特別な要件はありません。

データ型変換：JDBC と Adaptive Server Enterprise

CREATE TABLE 文を発行するときに、Adaptive Server 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、Adaptive Server Anywhere から Adaptive Server Enterprise へのデータ型変換を示します。

Adaptive Server Anywhere のデータ型	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)

Adaptive Server Anywhere のデータ型	**Adaptive Server Enterprise** のデフォルト・データ型
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(20,0)

ODBC ベースのサーバ・クラス

ODBC ベースのサーバ・クラスは以下のとおりです。

- asaodbc
- aseodbc
- db2odbc
- mssodbc
- oraodbc
- odbc

ODBC 外部サーバの定義

ODBC ベースのサーバを定義する最も一般的な方法は、ODBC データ・ソースを基にすることです。これを実行するには、ODBC アドミニストレータにデータ・ソースを作成します。

データ・ソースを定義すると、CREATE SERVER 文の USING 句が ODBC データ・ソース名に一致します。

たとえば、Data Source Name も **mydb2** である **mydb2** という名前の DB2 サーバを定義するには、以下の文を使用します。

```
CREATE SERVER mydb2
 CLASS 'db2odbc'
 USING 'mydb2'
```

データ・ソースの作成の詳細については、『ASA データベース管理ガイド』> 「ODBC データ・ソースの作成」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、以下は ASA への接続の例です。

```
CREATE SERVER testasa
 CLASS 'asaodbc'
 USING 'driver=adaptive server anywhere
9.0;eng=testasa;dbn=sample;links=tcpip{}'
```

この例で接続を定義する Adaptive Server Anywhere データベース・サーバは、名前が **testasa**、データベースが **sample**、使用するプロトコルは TCP-IP です。

参照

各 ODBC サーバ・クラスに固有の情報については、次を参照してください。

- 「サーバ・クラス asaodbc」794 ページ
- 「サーバ・クラス aseodbc」795 ページ
- 「サーバ・クラス db2odbc」797 ページ
- 「サーバ・クラス oraodbc」800 ページ
- 「サーバ・クラス mssodbc」801 ページ
- 「サーバ・クラス odbc」804 ページ

サーバ・クラス asaodbc

サーバ・クラス asaodbc のサーバは、Adaptive Server Anywhere バージョン 5.5 以降です。Adaptive Server 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 Adaptive Server Anywhere サーバにアクセスするには、各データベースへの接続を定義する ODBC データ・ソース名を作成します。これらの ODBC データ・ソース名のそれぞれに、CREATE SERVER 文を発行します。

サーバ・クラス aseodbc

サーバ・クラス aseodbc のサーバは Adaptive Server Enterprise と SQL Server (バージョン 10 以降) です。クラスが aseodbc のリモートの Adaptive Server Enterprise に接続するには、Adaptive Server Anywhere に Adaptive Server Enterprise ODBC ドライバと Open Client 接続ライブラリのインストールが必要です。しかし、パフォーマンスは asejdbc クラスよりも優れています。

注意

- Open Client はバージョン 11.1.1、EBF 7886 以降が必要です。Open Client をインストールして Adaptive Server Enterprise への接続を検証してから、ODBC をインストールして Adaptive Server Anywhere を設定してください。Sybase ODBC ドライバはバージョン 11.1.1、EBF 7911 以降が必要です。
- [Configuration Manager] の [User Data Source] を、次の属性で設定します。
  - [General] タブ

    [Data Source Name] に任意の値を入力します。この値は、CREATE SERVER 文の USING 句に使用されます。

    サーバ名は Sybase interfaces ファイルにあるサーバ名と一致させてください。
  - [Advanced] タブでは、[Application Using Threads] ボックスと [Enable Quoted Identifiers] ボックスをチェックします。
  - [Connection] タブ

    [charset] フィールドを、Adaptive Server 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。Sybase ではこの値を 100 に設定することをおすすめします。

[Select Method] を "0-Cursor" に設定します。

[Packet Size] をできるだけ大きな値に設定します。iAnywhere ではこの値を -1 に設定することをおすすめします。

[Connection Cache] を 1 に設定します。

データ型変換：ODBC と Adaptive Server Enterprise

CREATE TABLE 文を発行するときに、Adaptive Server 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、Adaptive Server Anywhere から Adaptive Server Enterprise へのデータ型変換を示します。

Adaptive Server Anywhere のデータ型	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

Adaptive Server Anywhere のデータ型	Adaptive Server Enterprise のデフォルト・データ型
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
Text	text
binary(n)	binary(n)
long binary	image
Image	image
Bigint	numeric(20,0)

サーバ・クラス db2odbc

サーバ・クラス db2odbc のサーバは、IBM DB2 です。

注意

- iAnywhere は、IBM の DB2 Connect バージョン 5 (修正パック WR09044 付き) の使用を確認しています。この製品の説明に従って、ODBC 構成の設定を行い、テストを実行してください。Adaptive Server 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、Adaptive Server 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、Adaptive Server Anywhere から DB2 へのデータ型変換を示します。

Adaptive Server Anywhere のデータ型	**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

Adaptive Server Anywhere のデータ型	DB2 のデフォルト・データ型
varchar(1-4000)	varchar(n)
varchar(4001-32767)	long varchar
Character varying(1-4000)	varchar(n)
Character varying(4001-32767)	long varchar
long varchar	long varchar
text	long varchar
binary(1-4000)	bit データには varchar
binary(4001-32767)	bit データには long varchar
long binary	bit データには long varchar
image	bit データには long varchar
decimal [デフォルトは p=30 s=6]	decimal(30,6)
numeric [デフォルトは p=30 s=6]	decimal(30,6)
decimal(128, 128)	サポートされていない
numeric(128, 128)	サポートされていない
real	real
float	float
double	float
smallmoney	decimal(10,4)
money	decimal(19,4)
date	date
time	time
smalldatetime	timestamp
datetime	timestamp

Adaptive Server Anywhere のデータ型	DB2 のデフォルト・データ型
timestamp	timestamp

サーバ・クラス oraodbc

サーバ・クラス **oraodbc** のサーバは、Oracle バージョン 8.0 以降です。

注意

- iAnywhere は、バージョン 8.0.03 の Oracle ODBC ドライバの使用を確認しています。この製品の説明に従って、ODBC 構成の設定を行い、テストを実行してください。
- 以下は、myora という Oracle サーバの CREATE EXISTING TABLE 文の例です。

```
CREATE EXISTING TABLE employees
 AT 'myora.database.owner.employees'
```

- Oracle ODBC ドライバの制限により、システム・テーブルの CREATE EXISTING TABLE を発行することはできません。テーブルまたはカラムが見つからないことを示すメッセージが返されます。

データ型変換：Oracle

CREATE TABLE 文を発行するときに、Adaptive Server Anywhere は、データ型を対応する Oracle のデータ型に自動的に変換します。次の表に、Adaptive Server Anywhere から Oracle へのデータ型変換を示します。

Adaptive Server Anywhere のデータ型	Oracle のデータ型
bit	number(1,0)
tinyint	number(3,0)
smallint	number(5,0)

Adaptive Server Anywhere のデータ型	Oracle のデータ型
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
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 SQL Server 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。
- 以下は、mymssql という Microsoft SQL Server サーバの CREATE EXISTING TABLE 文の例です。

```
CREATE EXISTING TABLE accounts,
 AT 'mymssql.database.owner.accounts'
```

データ型変換：Microsoft SQL Server

CREATE TABLE 文を発行するときに、Adaptive Server Anywhere は、データ型を対応する Microsoft SQL Server のデータ型に自動的に変換します。次の表に、Adaptive Server Anywhere から Microsoft SQL Server へのデータ型変換を示します。

Adaptive Server Anywhere のデータ型	Microsoft SQL Server のデフォルト・データ型
bit	bit
tinyint	tinyint
smallint	smallint
int	int
bigint	numeric(20,0)
decimal [デフォルトは p=30 s=6]	decimal(prec, scale)
numeric [デフォルトは p=30 s=6]	numeric(prec, scale)
float	(prec) の場合は float(prec)、それ以外は float
real	real
smallmoney	smallmoney
money	money

Adaptive Server Anywhere のデータ型	Microsoft SQL Server のデフォルト・データ型
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

Microsoft SQL Server の uniqueidentifier カラムは、Adaptive Server Anywhere の uniqueidentifierstr カラムにマップされます。マップされた結果の文字列は、STRTOUUID 関数を使用してバイナリ UUID 値に変換できます。『ASA SQL リファレンス・マニュアル』> 「STRTOUUID 関数 [文字列]」を参照してください。

サーバ・クラス odbc

独自のサーバ・クラスを持たない ODBC データ・ソースでは、サーバ・クラス **odbc** を使用します。ODBC バージョン 2.0 準拠レベル 1 以降に準拠する ODBC ドライバであれば、どれでも使用できます。iAnywhere は次の ODBC データ・ソースの使用を確認しています。

- 「Microsoft Excel (Microsoft 3.51.171300)」804 ページ
- 「Microsoft Access (Microsoft 3.51.171300)」805 ページ
- 「Microsoft FoxPro (Microsoft 3.51.171300)」806 ページ
- 「Lotus Notes SQL 2.0」806 ページ

最新バージョンの Microsoft ODBC ドライバは、Microsoft Data Access Components (MDAC) とともに、http://www.microsoft.com/downloads/search.aspx?displaylang=ja からダウンロードできます。MDAC 2.0 には、以下に示すバージョンの Microsoft ドライバが含まれています。

以降の項では、これらのデータ・ソースへのアクセスについての注意点を説明します。

Microsoft Excel (Microsoft 3.51.171300)

Excel で、それぞれの Excel ブックはいくつかのテーブルを持つデータベースであると、論理的に考えられます。テーブルはブック内でシートにマップされます。ODBC ドライバ・マネージャで ODBC データ・ソース名を設定する場合は、そのデータ・ソースに関連付けられたデフォルトのブック名を指定します。ただし、CREATE TABLE 文を発行する場合には、デフォルトを無効にしてロケーションの文字列にブック名を指定できます。これによって、1 つの ODBC DSN を使用してすべての Excel ブックにアクセスできます。

この例では、**excel** という名前の ODBC データ・ソースが作成されています。mywork というシート (テーブル) を持つ *work1.xls* というブックを作成するには、次のようにします。

```
CREATE TABLE mywork (a int, b char(20))
 AT 'excel;d:¥work1.xls;;mywork'
```

2 番目のシート (テーブル) を作成するには、次のような文を実行します。

```
CREATE TABLE mywork2 (x float, y int)
 AT 'excel;d:¥work1.xls;;mywork2'
```

CREATE EXISTING 文を使用して、既存のワークシートを Adaptive Server 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'
```

Adaptive Server Anywhere によって、テーブルが見つからないと表示された場合は、マップするカラムとローの範囲を明示的に指定する必要があります。次に例を示します。

```
CREATE EXISTING TABLE mywork
 AT 'excel;d:¥work1;;mywork$'
```

シート名に $ を追加すると、ワークシート全体が選択されることを示します。

AT で指定するロケーションの文字列で、フィールド・セパレータとしてピリオドの代わりにセミコロンが使用されていることに注意してください。これは、ファイル名にピリオドが使用されるためです。Excel では所有者名フィールドをサポートしないので、これはブランクのままにしてください。

削除はサポートされていません。また、Excel ドライバは位置付け更新をサポートしないため、更新も可能でない場合があります。

Microsoft Access (Microsoft 3.51.171300)

Access データベースは *.mdb* ファイルに格納されます。ODBC マネージャを使用して、ODBC データ・ソースを作成し、これらのファイルの 1 つにマップします。新しい *.mdb* ファイルは、ODBC マネージャを使って作成できます。Adaptive Server 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します。

```
CREATE TABLE tab1 (a int, b char(10))
 AT 'access...tab1'
```

または

```
CREATE TABLE tab1 (a int, b char(10))
 AT 'access;d:¥pcdb¥data.mdb;;tab1'
```

または

```
CREATE EXISTING TABLE tab1
 AT 'access;d:¥pcdb¥data.mdb;;tab1'
```

Access では所有者名の修飾をサポートしないので、これはブランクのままにしてください。

Microsoft FoxPro (Microsoft 3.51.171300)

FoxPro テーブルは 1 つの FoxPro データベース・ファイル (.dbc) にまとめて格納されるか、それぞれのテーブルが各自の .dbf ファイルに格納されます。.dbf ファイルを使用するときは、ファイル名がロケーションの文字列に入っていることを確認してください。入っていない場合は、Adaptive Server 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 (バージョン 2.04.0203) は、Lotus Web サイトから取得できます。Notes データがリレーショナル・テーブルにどのようにマップされるかについては、ドライバに付属のマニュアルを参照してください。Adaptive Server Anywhere テーブルは、Notes フォームに簡単にマップできます。

Address サンプル・ファイルにアクセスするよう Adaptive Server 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。

- Adaptive Server 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 Adaptive Server Anywhere テーブルにマップします。

```
CREATE EXISTING TABLE Person
 AT 'names...Person'
```

- テーブルを問い合わせます。

```
SELECT * FROM Person
```

パスワードのプロンプトを表示しない

Lotus Notes は、ODBC API を介したユーザ名とパスワードの送信をサポートしません。パスワードで保護された ID を使用して Lotus Notes にアクセスしようとすると、Adaptive Server Anywhere が稼働しているマシンに、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。

第 6 部　ストアド・プロシージャとトリガ

第 6 部では、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。ストアド・プロシージャ・デバッガは、すべての種類の論理をデバッグする強力なツールです。

第 20 章

プロシージャ、トリガ、バッチの使用

この章の内容

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。これ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。ユーザ定義関数は、クエリやその他の SQL 文で使うための結果を返すプロシージャの一種です。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。

さまざまな用途で、サーバ側 JDBC は、SQL ストアド・プロシージャよりも柔軟にデータベースにロジックを構築します。JDBC の詳細については、『ASA プログラミング・ガイド』> 「JDBC プログラミング」を参照してください。

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。

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。

プロシージャ・デバッガ

ストアド・プロシージャとトリガは、データベース・オブジェクト・デバッガを使用してデバッグできます。詳細については、「データベースでのデバッグ論理」889 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リングして、Sybase Central でパフォーマンスを分析できます。詳細については、「データベース・プロシージャのプロファイリング」259 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の定義はデータベース内にあり、データベース・アプリケーションから分離されています。これには、多くの利点があります。

標準化

プロシージャとトリガを使用すると、複数のアプリケーション・プログラムで実行するアクションを標準化できます。アクションをコーディングし、将来利用するためにデータベースに格納します。アプリケーションはプロシージャを呼び出すか、または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することができます。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することができます。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、すぐに実行することができます。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。

単純なクエリや多数の引数が含まれている場合、プロシージャは効率が低くなります。複雑なクエリの場合、プロシージャは効率が高くなります。

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、トリガを起動することができ

ます。同様に、プロシージャ (ユーザ定義関数を含む) はプロシージャの所有者のパーミッションにより実行されますが、パーミッションを与えられたユーザはプロシージャを呼び出すことができます。つまり、プロシージャとトリガのパーミッションはそれらを起動するユーザが持つパーミッションと異なる可能性があります。

プロシージャの概要

プロシージャを使用するには、次のことを行う方法を理解する必要があります。

- プロシージャの作成
- データベース・アプリケーションからの呼び出し
- プロシージャの削除
- パーミッションの制御

この項では、プロシージャを使用するための上記の方法とプロシージャの他の使用方法について説明します。

プロシージャの作成

Adaptive Server Anywhere では、新しいプロシージャを作成するためのツールが用意されています。

Sybase Central では、ウィザードを使用して必要な情報を指定できます。[プロシージャ作成] ウィザードには、プロシージャ・テンプレートを使用するオプションもあります。

または、CREATE PROCEDURE 文を使用してプロシージャを作成します。ただし、RESOURCE 権限を所有していなければなりません。文を入力する場所は、使用するツールによって異なります。

❖ **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 権限を所有するユーザとしてデータベースに接続します。
2. データベースの [プロシージャとファンクション] フォルダを開きます。
3. [ファイル] － [新規] － [プロシージャ] を選択します。

[プロシージャ作成] ウィザードが表示されます。

4 ウィザードの指示に従います。

5 ウィザードの終了後、右ウィンドウ枠の [SQL] タブでプロシージャ・コードを完了できます。

新しいプロシージャは、[プロシージャとファンクション] フォルダに表示されます。

❖ 新しいリモート・プロシージャを作成するには、次の手順に従います (Sybase Central の場合)。

1 DBA 権限を所有するユーザとしてデータベースに接続します。

2 データベースの [プロシージャとファンクション] フォルダを開きます。

3 [ファイル] － [新規] － [リモート・プロシージャ] を選択します。

[リモート・プロシージャ作成] ウィザードが表示されます。

4 ウィザードの指示に従います。

5 ウィザードの終了後、右ウィンドウ枠の [SQL] タブでプロシージャ・コードを完了できます。

新しいリモート・プロシージャは、[プロシージャとファンクション] フォルダに表示されます。

❖ 異なるツールを使用してプロシージャを作成するには、次の手順に従います。

- ツールの指示に従います。CREATE PROCEDURE 文を入力する前に、コマンド・デリミタをセミコロン以外のものに変更する必要がある場合もあります。

接続の詳細については、『ASA データベース管理ガイド』> 「データベースへの接続」を参照してください。

例

次に、サンプル・データベースの department テーブルに INSERT を行って、新しい部署を作成するプロシージャ new_dept の簡単な例を示します。

```
CREATE PROCEDURE new_dept (
    IN id INT,
    IN name CHAR(35),
    IN head_id INT )
 BEGIN
    INSERT
    INTO DBA.department ( dept_id,
        dept_name, dept_head_id )
    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。new_dept 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。new_dept プロシージャのパラメータはすべて、プロシージャによって変更されないので IN になります。

詳細については、『ASA SQL リファレンス・マニュアル』>「CREATE PROCEDURE 文」、『ASA SQL リファレンス・マニュアル』>「ALTER PROCEDURE 文」、「複合文の使用」842 ページを参照してください。

プロシージャの変更

Sybase Central または Interactive SQL を使って既存のプロシージャを変更することができます。DBA 権限を所有しているか、または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。

または、ALTER PROCEDURE 文を使用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。また、プロシージャにユーザ・パーミッションを再割り当てします。

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」44 ページを参照してください。

プロシージャのパーミッションの付与または取り消しの詳細については、『ASA データベース管理ガイド』> 「プロシージャに対するパーミッションの付与」と『ASA データベース管理ガイド』> 「ユーザ・パーミッションの取り消し」を参照してください。

❖ プロシージャのコードを変更するには、次の手順に従います (Sybase Central の場合)。

1. [プロシージャとファンクション] フォルダを開きます。
2. 対象のプロシージャを選択します。次のいずれかを行います。
   - 右ウィンドウ枠の [SQL] タブで、コードを直接編集する。
   - 編集する前に、コードを Watcom-SQL または Transact-SQL に変換する。
     - 対象のプロシージャを右クリックし、ポップアップ・メニューで [Watcom-SQL として開く] または [Transact-SQL として開く] を選択する。
     - 右ウィンドウ枠の [SQL] タブで、コードを編集する。
   - 右ウィンドウ枠でプロシージャを右クリックし、ポップアップ・メニューで [新しいウィンドウで編集] を選択して、コードを別のウィンドウで編集する。

ヒント
プロシージャ間でコードをコピーする場合は、プロシージャごとに別のウィンドウを開きます。

❖ **プロシージャのコードを変更するには、次の手順に従います (SQL の場合)。**

1. データベースに接続します。
2. ALTER PROCEDURE 文を実行します。この文には、新しいプロシージャ全体を追加します。

詳細については、『ASA SQL リファレンス・マニュアル』> 「ALTER PROCEDURE 文」、『ASA SQL リファレンス・マニュアル』> 「CREATE PROCEDURE 文」、「プロシージャの作成」815 ページを参照してください。

プロシージャの呼び出し

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

詳細については、『ASA SQL リファレンス・マニュアル』> 「CALL 文」を参照してください。

次に、new_dept 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 new_dept( 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、department テーブルを表示することができます。

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、department テーブルのパーミッションがなくても、new_dept プロシージャを呼び出すことができます。

EXECUTE パーミッションの詳細については、『ASA SQL リファレンス・マニュアル』> 「EXECUTE 文 [ESQL]」を参照してください。

結果セットを返すプロシージャを呼び出すもう 1 つの方法は、クエリ内で呼び出す方法です。プロシージャの結果セットに対してクエリを実行し、WHERE 句やその他の SELECT 機能を適用して、結果セットを制限できます。

```
SELECT t.id, t.quantity_ordered AS q
 FROM sp_customer_products( 149 ) t
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「FROM 句」を参照してください。

Sybase Central におけるプロシージャのコピー

Sybase Central では、データベース間でプロシージャをコピーできます。Sybase Central の左ウィンドウ枠でプロシージャを選択し、コピー先の接続済みデータベースの [プロシージャとファンクション] フォルダへドラッグします。新しいプロシージャが作成されて、元のプロシージャのコードがコピーされます。

新しいプロシージャにコピーされるのは、プロシージャのコードだけであることに注意してください。他のプロシージャ・プロパティ (パーミッションなど) はコピーされません。新しい名前を付ける場合、プロシージャは同じデータベースにコピーすることができます。

プロシージャの削除

作成したプロシージャは、明示的に削除されるまでデータベースに存在します。プロシージャの所有者か DBA 権限を所有するユーザだけがプロシージャをデータベースから削除できます。

❖ プロシージャを削除するには、次の手順に従います (Sybase Central の場合)。

1. DBA 権限を所有するユーザとして、またはプロシージャの所有者としてデータベースに接続します。

2　[プロシージャとファンクション] フォルダを開きます。

3　対象のプロシージャを右クリックし、ポップアップ・メニューで [削除] を選択します。

❖ **プロシージャを削除するには、次の手順に従います (SQL の場合)。**

1　DBA 権限を所有するユーザとして、またはプロシージャの所有者としてデータベースに接続します。

2　DROP PROCEDURE 文を実行します。

例

次の文は、new_dept プロシージャをデータベースから削除します。

```
DROP PROCEDURE new_dept
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「DROP 文」を参照してください。

プロシージャの結果をパラメータとして返す

プロシージャは次のいずれかの方法で、呼び出された環境に結果を返します。

- 個別の値を OUT または INOUT パラメータとして返す。
- 結果セットを返す。
- RETURN 文を使って 1 つの結果を返す。

ここでは、パラメータとしてプロシージャから結果を返す方法について説明します。

次に示す例は、サンプル・データベースの従業員の平均給与を OUT パラメータとして返します。

```
CREATE PROCEDURE AverageSalary( OUT avgsal
    NUMERIC (20,3) )
 BEGIN
    SELECT AVG( salary )
    INTO avgsal
    FROM employee;
 END
```

❖ このプロシージャを実行し、その出力を表示するには、次の手順に従います (SQL の場合)。

1. Interactive SQL からサンプル・データベースに、ユーザ ID **DBA**、パスワード **SQL** を使用して接続します。接続の詳細については、『ASA データベース管理ガイド』> 「データベースへの接続」を参照してください。

2. [SQL 文] ウィンドウ枠に上記のプロシージャ・コードを入力します。

3. プロシージャの結果を格納する変数を作成します。ここでは、結果は小数点以下 3 桁の数字なので、変数を次のように作成します。

   ```
   CREATE VARIABLE Average NUMERIC(20,3)
   ```

4. 作成した変数を使ってプロシージャを呼び出します。

   ```
   CALL AverageSalary(Average)
   ```

   プロシージャが正しく作成され、実行された場合、Interactive SQL の [メッセージ] ウィンドウ枠にエラーは表示されません。

5. SELECT Average 文を実行して変数の値を検査します。

   出力変数 Average の値を見ます。[結果] ウィンドウ枠の [結果] タブに、この変数の値 49988.623 が表示されます。これが従業員の給与の平均値です。

プロシージャの結果を結果セットとして返す

プロシージャは、個別のパラメータとして呼び出しを行った環境に結果を返すだけでなく、結果セットとして情報を返すこともできます。通常、結果セットになるのはクエリの結果です。次に示すプロシージャは、ある部署の従業員 1 人 1 人の給与をセットにして返します。

```
CREATE PROCEDURE SalaryList ( IN department_id INT)
 RESULT ( "Employee ID" INT, Salary NUMERIC(20,3) )
 BEGIN
      SELECT emp_id, salary
      FROM employee
      WHERE employee.dept_id = 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
105	62000
160	57490
243	72995
…	…

[オプション] ダイアログの [結果] タブでこのオプションを有効にした場合にのみ、Interactive SQL では複数の結果セットを返すことができます。各結果セットは [結果] ウィンドウ枠の個別のタブに表示されます。

詳細については、「プロシージャから複数の結果セットを返す」853 ページを参照してください。

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。Adaptive Server Anywhere では、すべてのユーザ定義関数は、NOT DETERMINISTIC と宣言されないかぎり「**冪等**」として扱われます。

冪等関数は、同じパラメータに対して一貫した結果を返し、副次効果はありません。同じパラメータを持つ冪等関数が連続して 2 回呼び出されている場合は、どちらの呼び出しでも同じ結果が返され、クエリのセマンティックに不要な副次効果は生じません。

非決定的関数と決定性関数の詳細については、「関数のキャッシュ」562 ページを参照してください。

ここでは、ユーザ定義関数の作成、使用、削除について説明します。

ユーザ定義関数の作成

CREATE FUNCTION 文を使用してユーザ定義関数を作成します。ただし、RESOURCE 権限を所有していなければなりません。

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
```

❖ **Interactive SQL を使用してこの例を作成するには、次の手順に従います。**

1 Interactive SQL からサンプル・データベースに、ユーザ ID **DBA**、パスワード **SQL** を使用して接続します。接続の詳細については、『ASA データベース管理ガイド』> 「データベースへの接続」を参照してください。

2 [SQL 文] ウィンドウ枠に上記の関数コードを入力します。

注意
Interactive SQL または Sybase Central 以外のツールを使用している場合は、コマンド・デリミタをセミコロン以外の文字に変更する必要がある場合があります。

詳細については、『ASA SQL リファレンス・マニュアル』> 「CREATE FUNCTION 文」を参照してください。

CREATE FUNCTION の構文は、CREATE PROCEDURE 文の構文と若干異なります。次に相違点を示します。

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

ユーザ定義関数の呼び出し

ユーザ定義関数は、集合関数以外の組み込み関数が使われていればどこでも使用できますが、適切なパーミッションが必要です。

次に示す例は、姓と名前の入った 2 つのカラムから氏名を表示します。

```
SELECT fullname (emp_fname, emp_lname)
 FROM employee;
```

fullname (emp_fname, emp_lname)
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');
```

fullname ('Jane','Smith')
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。

customer テーブルでは、アメリカの顧客の中にカナダの顧客が混ざっていますが、テーブルには country カラムはありません。このユーザ定義関数 nationality は、アメリカの郵便番号は数字だけですが、カナダの郵便番号はアルファベットで始まることを利用して、アメリカとカナダの顧客を区別します。

```
CREATE FUNCTION nationality( cust_id INT )
 RETURNS CHAR( 20 )
 BEGIN
     DECLARE natl CHAR(20);
     IF cust_id IN ( SELECT id FROM customer
     WHERE LEFT(zip,1) > '9') THEN
              SET natl = 'CDN';
     ELSE
           SET natl = 'USA';
     END IF;
     RETURN ( natl );
 END
```

この例では国名を入れる変数 natl を宣言し、SET 文を使用して値を natl に入れます。次に natl 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、customer 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 customer
 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。

特に郵便番号にインデックスがある場合、関数を使わずに同じクエリを書き換えると、性能が向上します。次に例を示します。

```
Select *
 FROM customer
 WHERE zip > '99999'
```

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 SELECT クエリを使用する場合、関数は 10,000 回呼び出されます。同じクエリの WHERE 句に関数を使用した場合は、100,000 回呼び出されます。

ユーザ定義関数の削除

作成したユーザ定義関数は、いずれかのユーザが明示的に削除するまでデータベースに保持されます。ユーザ定義関数の所有者または DBA 権限を所有するユーザのみが、データベースから関数を削除できます。

次に、fullname 関数をデータベースから削除する文を示します。

```
DROP FUNCTION fullname
```

ユーザ定義関数を実行するためのパーミッション

ユーザ定義関数の所有権はその関数を作成したユーザに所属し、そのユーザはパーミッションなしに実行することができます。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 another_user 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 fullname TO another_user
```

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 fullname FROM another_user
```

関数のユーザ・パーミッションの管理の詳細については、『ASA データベース管理ガイド』> 「プロシージャに対するパーミッションの付与」を参照してください。

トリガの概要

トリガとは、データを修正するクエリ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。

参照整合性の詳細については、「データ整合性の確保」101 ページと『ASA SQL リファレンス・マニュアル』>「CREATE TABLE 文」を参照してください。

検査項目を細かく設定して複雑な参照整合性を設定したり、既存のデータは制約の範囲から外れても許可するが新しいデータはチェックしたりする場合があります。トリガはこのようなときに使用すると便利です。また、データベースにアクセスするアプリケーションとは個別に、データベース・テーブルのアクティビティのログを取るときにもトリガを使います。

トリガを実行するためのパーミッション
トリガは、関連するテーブル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・イベント

トリガを起動するイベントのリストを次に示します。

動作	説明
INSERT	トリガの関連するテーブルの中に新しいローが挿入されたときに、トリガが起動される。
DELETE	トリガの関連するテーブル中のローが削除されたときに、トリガが起動される。
UPDATE	トリガの関連するテーブル中のローが更新されたときに、トリガが起動される。
UPDATE OF column-list	トリガの関連するテーブル中のローが、*column-list* 中のカラムが変更されるなどして更新されたときに、トリガが起動される。

処理が必要なイベントごとにトリガを個別に作成できます。または、共有するアクションや、イベントに応じたアクションが複数ある場合は、すべてのイベントに対して 1 つのトリガを作成し、IF 文を使用して実行するアクションを区別できます。

詳細については、『ASA SQL リファレンス・マニュアル』> 「トリガ・オペレーション条件」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。
- 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。

トリガ実行のタイミングは柔軟に設定できるので、トリガが参照整合性、たとえばカスケード更新または削除が実行されたかどうかによって実行するかどうかを決めるような場合に、特に有効です。

トリガの実行中にエラーが発生すると、トリガを起動した操作そのものがエラーになります。INSERT、UPDATE、DELETE はアトミック・オペレーションです (「アトミックな複合文」843 ページを参照)。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。

トリガの構文の詳細については、『ASA SQL リファレンス・マニュアル』> 「CREATE TRIGGER 文」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 文を使用することができます。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。

詳細については、この項の最後の相互参照のリストを参照してください。

❖ 指定されたテーブルに対するトリガを作成するには、次の手順に従います (Sybase Central の場合)。

1. 対象のテーブルの [トリガ] フォルダを開きます。
2. [ファイル] － [新規] － [トリガ] を選択します。

   [トリガ作成] ウィザードが表示されます。
3. ウィザードの指示に従います。
4. ウィザードの終了後、右ウィンドウ枠の [SQL] タブでトリガ・コードを完了できます。

❖ 指定されたテーブルに対するトリガを作成するには、次の手順に従います (SQL の場合)。

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

例 1 : ロー・レベルの INSERT トリガ

次にロー・レベルの INSERT トリガの例を示します。新しい従業員の生年月日が正しく入力されたかどうかをチェックします。

```
CREATE TRIGGER check_birth_date
      AFTER INSERT ON Employee
 REFERENCING NEW AS new_employee
 FOR EACH ROW
 BEGIN
       DECLARE err_user_error EXCEPTION
       FOR SQLSTATE '99999';
       IF new_employee.birth_date > 'June 6, 2001' THEN
             SIGNAL err_user_error;
       END IF;
 END
```

このトリガは、employee テーブルに新しいローが追加されると起動されます。2001 年 6 月 6 日以降の生年月日に対応する新しいローを検知し、エラーにします。

フレーズ REFERENCING NEW AS new_employee は、トリガ・コード中の文がエイリアス new_employee を使用して、新しいローのデータを参照できるようにします。

エラーを感知すると、トリガ文とトリガ以前のデータの変更が取り消されます。

employee 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、例文の最初の行を次のように変更します。

```
CREATE TRIGGER mytrigger BEFORE INSERT ON Employee
```

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 employee
 REFERENCING OLD AS oldtable
 FOR EACH ROW
 BEGIN
       ...
 END
```

REFERENCING OLD 句は、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。

このトリガをローが削除された後に起動されるようにするには、例文の最初の行を次のように変更します。

```
CREATE TRIGGER check_birth_date AFTER DELETE ON
employee
```

REFERENCING OLD 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。

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 employee
 REFERENCING NEW AS table_after_update
                OLD AS table_before_update
 FOR EACH STATEMENT
 BEGIN
       ...
 END
```

REFERENCING NEW 句と REFERENCING OLD 句は、UPDATE トリガのコマンド文が更新の前と後の両方の値を参照できるようにします。テーブル・エイリアス table_after_update は、新しいローのカラムを参照し、テーブル・エイリアス table_before_update は古いローのカラムを参照します。

REFERENCING NEW 句と REFERENCING OLD 句は、文レベルとロー・レベルのトリガで少し異なる意味を持ちます。文レベルではテーブルが対象になりますが、ロー・レベルでは変更されるローが対象になります。

詳細については、『ASA SQL リファレンス・マニュアル』>「CREATE TRIGGER 文」と「複合文の使用」842 ページを参照してください。

トリガの実行

トリガは指定したテーブルに INSERT、UPDATE、DELETE が行われたときに、自動的に実行されます。ロー・レベル・トリガは、ローが影響を受けるごとに起動され、文レベル・トリガは、文全体が一度に起動されます。

INSERT、UPDATE、DELETE がトリガを起動すると、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. 参照動作を行います。
3. 追加などの操作そのものが実行されます。
4. AFTER トリガが起動します。

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することができます。トリガを定義するテーブルの所有者であるか、DBA 権限を所有しているか、テーブルに対する ALTER パーミッションと RESOURCE 権限を所有してなければなりません。

Sybase Central では、既存のトリガの名前を直接変更することはできません。代わりに、新しい名前を付けて新しくトリガを作成し、このトリガに以前のコードをコピーしてから、元のトリガを削除します。

または、ALTER TRIGGER 文を使用して既存のトリガを修正できます。トリガを作成した CREATE TRIGGER 文と同じ構文で、この文に新しいトリガ全体を含めます。

データベース・オブジェクトのプロパティの変更については、「データベース・オブジェクトのプロパティの設定」44 ページを参照してください。

❖ トリガのコードを変更するには、次の手順に従います (Sybase Central の場合)。

1 [トリガ] フォルダを開きます。

2 対象のトリガを選択します。次のいずれかを行います。

- 右ウィンドウ枠の [SQL] タブで、コードを直接編集する。
- 編集する前に、コードを Watcom-SQL または Transact-SQL に変換する。
  - 対象のトリガを右クリックし、ポップアップ・メニューで [Watcom-SQL として開く] または [Transact-SQL として開く] を選択する。
  - 右ウィンドウ枠の [SQL] タブで、コードを編集する。
- 右ウィンドウ枠でトリガを右クリックし、ポップアップ・メニューで [新しいウィンドウで編集] を選択して、コードを別のウィンドウで編集する。

ヒント
トリガ間でコードをコピーする場合は、トリガごとに別のウィンドウを開きます。

❖ トリガのコードを変更するには、次の手順に従います (SQL の場合)。

1 データベースに接続します。

2 ALTER TRIGGER 文を実行します。この文に新しいトリガ全体を含めます。

詳細については、『ASA SQL リファレンス・マニュアル』> 「ALTER TRIGGER 文」を参照してください。

トリガの削除

作成したトリガは、明示的に削除されるまでデータベースに存在します。トリガを削除するには、トリガの関連するテーブルの ALTER パーミッションが必要となります。

❖ **トリガを削除するには、次の手順に従います (Sybase Central の場合)。**

1. [トリガ] フォルダを開きます。
2. 対象のトリガを右クリックし、ポップアップ・メニューで [削除] を選択します。

❖ **トリガを削除するには、次の手順に従います (SQL の場合)。**

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

例

次に、トリガ mytrigger をデータベースから削除する文を示します。

```
DROP TRIGGER mytrigger
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「DROP 文」を参照してください。

トリガを実行するためのパーミッション

ユーザはトリガを実行できないので、トリガを実行するパーミッションを与えることはできません。データベースに対するアクションに対応して Adaptive Server Anywhere がトリガを起動します。トリガが実行される場合は、トリガに関連するパーミッションがあり、その動作を実行する権利を定義します。

トリガは、トリガが定義されているテーブルの所有者のパーミッションを使用して実行します。トリガを起動する原因となったユーザのパーミッションや、トリガを作成したユーザのパーミッションではありません。

トリガがテーブルを参照するときは、そのテーブルの所有者名を特に指定しないで、テーブル作成者のグループ・メンバシップを使います。たとえば、user_1.Table_A にあるトリガが Table_B を参照し、Table_B の所有者の名前を指定しないとします。この場合、Table_B が user_1 によって作成されたか、user_1 が Table_B の所有者であるグ

ループの (直接または間接的に) メンバでなければなりません。どちらの条件も満たされない場合は、トリガを起動すると、「**テーブルが見つかりません**」というメッセージが表示されます。

また、user_1 はトリガに指定された操作を実行するためのパーミッションを持っていなければなりません。

バッチの概要

簡単なバッチは、SQL 文のセットをセミコロンまたはワード **go** だけが含まれる行で区切ったものです。**go** を使用することをおすすめします。たとえば、次に示す文はバッチを構成します。部署 Eastern Sales を作成し、マサチューセッツ州の営業担当者をすべてこの部署に異動するバッチの例を次に示します。

```
INSERT
 INTO department ( dept_id, dept_name )
 VALUES ( 220, 'Eastern Sales' )
 go
 UPDATE employee
 SET dept_id = 220
 WHERE dept_id = 200
 AND state = 'MA'
 go
 COMMIT
 go
```

これはアプリケーションに組み込んで、一緒に実行することもできます。

Interactive SQL とバッチ

セミコロンで区切られた文のリストの場合、Interactive SQL は、それを解析してからサーバに送ります。この場合、Interactive SQL は各文を個別にサーバに送り、バッチとしては処理しません。ただし、BEGIN と END で囲まれていればその部分を Interactive SQL は、バッチとして取り扱います。また、アプリケーションから利用する場合は、文はバッチとして送られ、処理されます。

プロシージャとトリガ内で使用される多くの文は、バッチ内でも使用できます。バッチ内では制御文 (CASE、IF、LOOP など) を複合文 (BEGIN と END) を含めて使用できます。複合文の中には変数の宣言、例外、テンポラリ・テーブル、カーソルを含めることができます。

次にテーブルを作成するバッチの例を示しますが、同じ名前のテーブルが存在していると作成は行われません。

```
IF NOT EXISTS (
     SELECT * FROM SYSTABLE
     WHERE table_name = 't1' ) THEN
         CREATE TABLE t1 (
         firstcol INT PRIMARY KEY,
         secondcol CHAR( 30 )
         )
         go
 ELSE
     MESSAGE 'Table t1 already exists' TO CLIENT;
 END IF
```

このバッチを Interactive SQL から 2 度実行すると、最初はテーブルを作成しますが、2 度目には Interactive SQL の [メッセージ] ウィンドウ枠にメッセージを表示します。

制御文

プロシージャまたはトリガの本文、またはバッチの中には、論理フローや、意思決定のための制御文がたくさんあります。使用可能な制御文は、次のとおりです。

制御文	構文

複合文
 	
詳細については、『ASA SQL リファレンス・マニュアル』>「BEGIN 文」を参照してください。
	`BEGIN [ ATOMIC ]`
	`  Statement-list`
	`END`

条件実行：IF
 	
詳細については、『ASA SQL リファレンス・マニュアル』>「IF 文」を参照してください。
	`IF condition THEN`
	`  Statement-list`
	`ELSEIF condition THEN`
	`  Statement-list`
	`ELSE`
	`  Statement-list`
	`END IF`

条件実行：CASE
 	
詳細については、『ASA SQL リファレンス・マニュアル』>「CASE 文」を参照してください。
	`CASE expression`
	`WHEN value THEN`
	`  Statement-list`
	`WHEN value THEN`
	`  Statement-list`
	`ELSE`
	`  Statement-list`
	`END CASE`

繰り返し：WHILE、LOOP
 	
詳細については、『ASA SQL リファレンス・マニュアル』>「LOOP 文」を参照してください。
	`WHILE condition LOOP`
	`  Statement-list`
	`END LOOP`

制御文	構文

繰り返し：FOR カーソル・ループ
 	
詳細については、『ASA SQL リファレンス・マニュアル』>「FOR 文」を参照してください。
	`FOR loop-name`
	`  AS cursor-name`
	`  CURSOR FOR select statement`
	`DO`
	`  Statement-list`
	`END FOR`

中断：LEAVE 詳細については、『ASA SQL リファレンス・マニュアル』>「LEAVE 文」を参照してください。	`LEAVE label`
CALL 詳細については、『ASA SQL リファレンス・マニュアル』>「CALL 文」を参照してください。	`CALL procname( arg, ... )`

各文の詳細については、『ASA SQL リファレンス・マニュアル』>「SQL 文」のエントリを参照してください。

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、『ASA SQL リファレンス・マニュアル』>「BEGIN 文」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することができます。

- 変数
- カーソル
- テンポラリ・テーブル
- 例外処理 (エラー識別子)

ローカル宣言は、複合文またはその中でネストされる複合文の中のどの文からでも参照できます。ローカル宣言は、複合文中から呼び出された他のプロシージャからは見えません。

アトミックな複合文

「アトミック」な文は、完全に実行されるか、まったく実行されません。たとえば、何千ものローを更新する UPDATE 文では、たくさんのローを更新した後にエラーが発生することがあります。文が完了しないと、すべての変更内容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 employee
      SET manager_ID = 501
      WHERE emp_ID = 467;
      UPDATE employee
      SET birth_date = '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 birth_date 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。

COMMIT 文と ROLLBACK 文、いくつかの ROLLBACK TO SAVEPOINT 文は、アトミックな複合文内で使用できません (「プロシージャとトリガでのトランザクションとセーブポイント」873 ページを参照)。

アトミックな複合文中のいくつかの文のみが実行されるときもあります。複合文の中で例外ハンドラがエラーを処理するときに、このようなことが起こります。

詳細については、「プロシージャとトリガでの例外ハンドラの使用」866 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」842 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。

プロシージャとトリガで使用できる SQL 文

プロシージャとトリガには、次に示すように、ほとんどの SQL 文を使用することができます。

- SELECT、UPDATE、DELETE、INSERT、SET 変数
- 他のプロシージャを実行する CALL 文
- 制御文 (「制御文」841 ページを参照)
- カーソル文 (「プロシージャとトリガでのカーソルの使用」856 ページを参照)
- 例外処理文 (「プロシージャとトリガでの例外ハンドラの使用」866 ページを参照)
- EXECUTE IMMEDIATE 文

プロシージャとトリガ内で使用できない SQL 文には次のようなものがあります。

- CONNECT 文
- DISCONNECT 文

COMMIT、ROLLBACK、SAVEPOINT 文はプロシージャとトリガ内で使用できますが、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」873 ページを参照)。

詳細については、『ASA SQL リファレンス・マニュアル』> 「 SQL 文」の章の各 SQL 文の使用方法を参照してください。

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 (『ASA SQL リファレンス・マニュアル』> 「SQL データ型」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     product.id,
                      sum( sales_order_items.quantity )
           FROM    product,
                      sales_order_items,
                      sales_order
           WHERE sales_order.cust_id = customer_id
           AND sales_order.id = sales_order_items.id
           AND sales_order_items.prod_id = product.id
           GROUP BY product.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 とおりの形式のどちらでも使用できます。

CREATE PROCEDURE 文の引数リストの末尾にオプションのパラメータがある場合、これらは CALL 文で省略できます。次に示すのは、INOUT パラメータを 3 つ持つプロシージャの例です。

```
CREATE PROCEDURE SampleProc( INOUT var1 INT
                                        DEFAULT 1,
                             INOUT var2 int DEFAULT 2,
                             INOUT var3 int DEFAULT 3 )
 ...
```

プロシージャを呼び出す環境では、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みなされます。

```
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、SampleProc を呼び出すことも可能です。

```
CALL SampleProc( 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、SampleProc プロシージャは次のように呼び出されます。

```
CALL SampleProc( var1 = V1, var3 = V3 )
```

または次のようになります。

```
CALL SampleProc( 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」825 ページで定義した、従業員の氏名を取り出す fullname 関数を使用した例を次に示します。

❖ 全従業員の名前をリストするには、次の手順に従います。

- 次のように入力します。

```
SELECT fullname(emp_fname, emp_lname) AS Name
 FROM employee
```

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」815 ページを参照してください。詳細については、次の項を参照してください。

RETURN 文を使って値を返す

RETURN 文は、呼び出しを行った環境に 1 つの整数値を返した後、すぐにプロシージャを終了します。次に RETURN 文を示します。

```
RETURN expression
```

式の値が、呼び出しを行った環境に返されます。返ってきた値を変数に保存するには、CALL 文の拡張機能を使います。

```
CREATE VARIABLE returnval INTEGER ;
 returnval = CALL myproc() ;
```

結果をプロシージャのパラメータとして返す

プロシージャは、プロシージャのパラメータで呼び出しを行った環境に結果を返すことができます。

次の文を使用して、プロシージャ内でパラメータと変数に値を割り当てることができます。

- SET 文
- INTO 句を持つ SELECT 文

SET 文の使用

次に示すプロシージャは、SET 文を使って割り当てた OUT パラメータに値を返します。

```
CREATE PROCEDURE greater (    IN a INT,
                                     IN b INT,
                                     OUT c INT)
 BEGIN
     IF a > b THEN
         SET c = a;
     ELSE
         SET c = b;
     END IF ;
 END
```

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」851 ページを参照してください。

クエリの結果、ローが取り出されなかった場合は、「**ローが見つかりません**」という警告が表示されます。

次にシングルローの SELECT 文の結果をパラメータに返すプロシージャの例を示します。

❖ **指定した顧客によって行われた発注の数を返すには、次の手順に従います。**

- 次のように入力します。

```
CREATE PROCEDURE OrderCount (IN customer_ID INT,
                                        OUT Orders INT)
 BEGIN
     SELECT COUNT(DBA.sales_order.id)
         INTO Orders
     FROM DBA.customer
         KEY LEFT OUTER JOIN "DBA".sales_order
     WHERE DBA.customer.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 company_name,
         CAST( sum(    sales_order_items.quantity *
                         product.unit_price)
                         AS INTEGER ) AS value
```

```
    FROM customer
        INNER JOIN sales_order
        INNER JOIN sales_order_items
        INNER JOIN product
    GROUP BY company_name
    ORDER BY value DESC;
END
```

- 次のように入力します。

```
CALL ListCustomerValue ()
```

Company	Value
Chadwicks	8076
Overland Army Navy	8064
Martins Landing	6888
Sterling & Co.	6804
Carmel Industries	6780
…	…

注意

- RESULT 句のリスト中の変数の数は、SELECT 文のリスト中の変数の数に一致しなければなりません。データ型が一致しない場合は、可能であれば自動的にデータ型の変換が行われます。
- RESULT 句は CREATE PROCEDURE 文の一部で、コマンド・デリミタは付きません。
- SELECT 文のリスト中の変数の名前は、RESULT 句のリスト中の変数の名前と一致する必要はありません。
- このプロシージャをテストするとき、デフォルトでは Interactive SQL は最初の結果セットのみを返します。[オプション] ダイアログの [結果] タブの [複数の結果セットを表示] オプションを設定して、複数の結果セットを表示するように Interactive SQL を設定できます。

- 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することができます。プロシージャの結果を修正する場合、プロシージャを呼び出すユーザは基本のテーブルに対して適切なパーミッションを持っている必要があります。この点が、通常のプロシージャの実行のパーミッションとは異なります。通常は、プロシージャの*所有者*がテーブルに対するパーミッションを持っていなければなりません。

  Interactive SQL で結果セットを修正する方法については、『SQL Anywhere Studio の紹介』> 「Interactive SQL でのテーブル値の編集」を参照してください。

- ストアド・プロシージャまたはユーザ定義関数が結果セットを返す場合、出力パラメータを設定したり戻り値を返したりすることはできません。

プロシージャから複数の結果セットを返す

Interactive SQL が複数の結果セットを返すように設定するには、[オプション] ダイアログの [結果] タブのこのオプションをオンに設定する必要があります。デフォルトでは、このオプションはオフに設定されます。設定を変更する場合、変更された設定は新しく接続を行うときに有効になります (新しいウィンドウなど)。

❖ 複数の結果セット機能を有効にするには、次の手順に従います。

1. [ツール] － [オプション] を選択します。
2. 表示される [オプション] ダイアログで、[結果] タブをクリックします。
3. [複数の結果セットを表示] チェック・ボックスをオンにします。

このオプションを有効にすると、プロシージャは呼び出しを行った環境に複数の結果セットを返すことができます。RESULT 句を使う場合、結果セットはそれに合わせなければなりません。つまり、結果セットは SELECT 文のリストと同じ数の項目を持ち、データ型は RESULT 句にリストされたデータ型に自動的に変換されます。

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 ( lname CHAR(36), fname CHAR(36) )
 BEGIN
     SELECT emp_lname, emp_fname
     FROM employee;
     SELECT lname, fname
     FROM customer;
     SELECT last_name, first_name
     FROM contact;
 END
```

注意

- Interactive SQL でこのプロシージャをテストするには、[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 emp_lname, emp_fname
         FROM employee
     ELSE
         SELECT emp_fname
         FROM employee
     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、『ASA SQL リファレンス・マニュアル』> 「DESCRIBE 文 [ESQL]」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。Adaptive Server 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。Adaptive Server 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 を使って開いたカーソルは、いずれかのユーザが明示的に閉じるまでの間、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、『ASA プログラミング・ガイド』> 「カーソル位置」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」851 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 "error 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 company_name,
             CAST( sum( sales_order_items.quantity *
                     product.unit_price ) AS INTEGER )
             AS value
     FROM customer
         INNER JOIN sales_order
         INNER JOIN sales_order_items
         INNER JOIN product
     GROUP BY company_name;

 -- 4. Initialize the values of TopValue
     SET TopValue = 0;
     -- 5. Open the cursor
     OPEN ThisCompany;

 -- 6. Loop over the rows of the query
     CompanyLoop:
     LOOP
         FETCH NEXT ThisCompany
             INTO ThisName, ThisValue;
         IF SQLSTATE = err_notfound THEN
             LEAVE CompanyLoop;
         END IF;
         IF ThisValue > TopValue THEN
             SET TopCompany = ThisName;
             SET TopValue = ThisValue;
         END IF;
     END LOOP CompanyLoop;
```

```
-- 7. Close the cursor
    CLOSE ThisCompany;
END
```

注意

この TopCustomerValue プロシージャには、次の特徴があります。

- 例外 "error not found" が宣言されます。この例外は、プロシージャの後でクエリの結果のループが完了するときに通知されます。

  例外の詳細については、「プロシージャとトリガでのエラーと警告」860 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 文は標準的なフォームで、最後のローを処理して終了します。FOR ループを使うと、このプロシージャはさらに簡潔になります。FOR 文は 1 つの文に、上記のプロシージャのいくつかの要素を組み込みます。

```
CREATE PROCEDURE TopCustomerValue2(
         OUT TopCompany CHAR(36),
         OUT TopValue INT )
 BEGIN
     -- Initialize the TopValue variable
     SET TopValue = 0;
     -- Do the For Loop
     FOR CompanyFor AS ThisCompany
         CURSOR FOR
         SELECT company_name AS ThisName ,
             CAST( sum( sales_order_items.quantity *
                     product.unit_price ) AS INTEGER )
             AS ThisValue
         FROM customer
             INNER JOIN sales_order
             INNER JOIN sales_order_items
             INNER JOIN product
         GROUP BY ThisName
     DO
         IF ThisValue > TopValue THEN
             SET TopCompany = ThisName;
             SET TopValue = ThisValue;
             END IF;
     END FOR;
 END
```

プロシージャとトリガでのエラーと警告

アプリケーションが SQL 文を実行した後、「**ステータス・コード**」をチェックできます。ステータス・コード (リターン・コード) は文が正しく実行されたかどうかを表示して、エラーの場合はその理由を提示します。プロシージャを呼び出す CALL 文にも同じメカニズムが使われます。

エラーのレポートには、SQLCODE か SQLSTATE のどちらかのステータス表示を使用します。SQLCODE と SQLSTATE のエラーと警告値とそれらの意味の詳細については、『ASA エラー・メッセージ』> 「ASA エラー・メッセージ」を参照してください。SQL 文が実行されると、SQLCODE と SQLSTATE と呼ばれる特別なプロシージャ変数に値が入ります。この値は、文の実行中に変わった状況が発生したかどうかを示します。SQLCODE と SQLSTATE の値は、IF 文を SQL 文の後に置いてチェックできます。その結果によって適切な動作が行われます。

たとえば、SQLSTATE 変数はローが正しくフェッチされたかどうかを示すのに使用できます。「プロシージャの SELECT 文でのカーソルの使用」856 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、Adaptive Server 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」866 ページで説明した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」865 ページを参照してください。

エラーを処理するには、特に指定しないかぎり次の 2 とおりの方法があります。

- **デフォルトのエラー処理**　プロシージャかトリガがエラーを起こしたときに、呼び出しを行った環境にエラー・コードが返されます。

- **ON EXCEPTION RESUME**　CREATE PROCEDURE 文に ON EXCEPTION RESUME 句が含まれていれば、プロシージャはエラーを起こした箇所の次の文から実行を再開します。

  ON EXCEPTION RESUME を使用するプロシージャの正確な動作は、ON_TSQL_ERROR オプション設定によって指定します。詳細については、『ASA データベース管理ガイド』>「ON_TSQL_ERROR オプション [互換性]」を参照してください。

デフォルトのエラー処理

通常、プロシージャまたはトリガの SQL 文がエラーを起こすと、そのプロシージャまたはトリガは実行を停止し、SQLCODE と SQLSTATE に適切な値が入った状態でアプリケーションに制御が戻されます。これは最初の文がエラーを起こしたときも同じです。トリガの場合は、トリガを起動した操作も取り消され、アプリケーションにエラーが返されます。

次の例のプロシージャは、アプリケーションからプロシージャ OuterProc を呼び出し、OuterProc が InnerProc を呼び出して、そこでエラーが発生した場合の処理を示します。

```
CREATE PROCEDURE OuterProc()
 BEGIN
     MESSAGE 'Hello from OuterProc.' TO CLIENT;
     CALL InnerProc();
     MESSAGE 'SQLSTATE set to ',
         SQLSTATE,' in OuterProc.' TO CLIENT
 END
 CREATE PROCEDURE InnerProc()
     BEGIN
         DECLARE column_not_found
             EXCEPTION FOR SQLSTATE '52003';
         MESSAGE 'Hello from InnerProc.' TO CLIENT;
         SIGNAL column_not_found;
     MESSAGE 'SQLSTATE set to ',
         SQLSTATE, ' in InnerProc.' TO CLIENT;
 END
```

注意

- InnerProc 内の DECLARE 文は、サーバが認識しているエラー条件に関連して事前に定義された SQLSTATE 値のうち、1 つの記号名を宣言します。
- MESSAGE 文は Interactive SQL の [メッセージ] ウィンドウ枠にメッセージを送ります。
- SIGNAL 文は InnerProc プロシージャ内から、エラーであることを外部に知らせる役割を持ちます。

OuterProc プロシージャを実行するには、次の文を入力してください。

```
CALL OuterProc();
```

Interactive SQL の [メッセージ] ウィンドウ枠に次のメッセージが表示されます。

```
Hello from OuterProc.

Hello from InnerProc.
```

InnerProc の SIGNAL 文の後の文は実行されず、InnerProc はすぐに呼び出しを行った環境 (この場合はプロシージャ OuterProc) に制御を戻します。OuterProc の CALL 文の後に続く文は実行されません。エラーは呼び出しを行った環境に戻され、処理されます。たとえば、Interactive SQL はエラー・メッセージをメッセージ・ウィンドウに表示してエラーの処理を行います。

TRACEBACK 関数はエラーが起きたときに実行していた文をリストします。Interactive SQL から TRACEBACK を使うには、次の文を入力します。

```
SELECT TRACEBACK(*)
```

ON EXCEPTION RESUME を使ったエラー処理

ON EXCEPTION RESUME 文が CREATE PROCEDURE 文に含まれていた場合、エラーが起きると、次の文が検査されます。その文がエラーを処理する場合、プロシージャの実行が続行され、エラーが発生した文の次の文を実行します。エラーが発生したとき、呼び出しを行った環境に制御を戻しません。

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することができます。詳細については、『ASA データベース管理ガイド』> 「ON_TSQL_ERROR オプション [互換性]」を参照してください。

エラー処理文には、次のようなものがあります。

- IF
- SELECT @variable =
- CASE
- LOOP
- LEAVE
- CONTINUE
- CALL
- EXECUTE
- SIGNAL
- RESIGNAL
- DECLARE
- SET VARIABLE

この機能を、次の例で説明します。

プロシージャの削除

[SQL 文] ウィンドウ枠に次のコマンドを入力して、必ずプロシージャ InnerProc と OuterProc を削除してから、チュートリアルを続けてください。

```
DROP PROCEDURE OuterProc;
 DROP PROCEDURE InnerProc
```

次に示すプロシージャは、アプリケーションからプロシージャ OuterProc を呼び出し、OuterProc が InnerProc を呼び出して、そこでエラーが発生した場合の処理を示します。例文は、この項の最初で使用したプロシージャを基にしています。

```
CREATE PROCEDURE OuterProc()
 ON EXCEPTION RESUME
 BEGIN
     DECLARE res CHAR(5);
     MESSAGE 'Hello from OuterProc.' TO CLIENT;
     CALL InnerProc();
     SELECT @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
```

OuterProc プロシージャを実行するには、次の文を入力してください。

```
CALL OuterProc();
```

Interactive SQL の [メッセージ] ウィンドウ枠に、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。

プロシージャの削除

[SQL 文] ウィンドウ枠に次のコマンドを入力して、必ずプロシージャ InnerProc と OuterProc を削除してから、チュートリアルを続けてください。

```
DROP PROCEDURE OuterProc;
 DROP PROCEDURE InnerProc
```

次に示す例は、デフォルトの警告処理を示します。例文は、「プロシージャとトリガでのデフォルトのエラー処理」860 ページで使用したプロシージャを基にしています。ここでは SIGNAL 文はエラーではなく、「ローが見つかりません」という警告を表示します。

```
CREATE PROCEDURE OuterProc()
 BEGIN
     MESSAGE 'Hello from OuterProc.' TO CLIENT;
     CALL InnerProc();
     MESSAGE 'SQLSTATE set to ',
         SQLSTATE,' in OuterProc.' TO CLIENT;
 END
 CREATE PROCEDURE InnerProc()
 BEGIN
     DECLARE row_not_found
         EXCEPTION FOR SQLSTATE '02000';
```

```
        MESSAGE 'Hello from InnerProc.' TO CLIENT;
        SIGNAL row_not_found;
        MESSAGE 'SQLSTATE set to ',
        SQLSTATE, ' in InnerProc.' TO CLIENT;
    END
```

OuterProc プロシージャを実行するには、次の文を入力してください。

```
CALL OuterProc();
```

Interactive SQL の [メッセージ] ウィンドウ枠に、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ほうが良い場合があります。これは「**例外ハンドラ**」を使用して行います。

例外ハンドラは、複合文の EXCEPTION で定義します (「複合文の使用」842 ページを参照)。複合文でエラーが起きた場合、例外ハンドラが実行されます。警告では、例外ハンドラは実行されません。ネストされた複合文の中でエラーが起きた場合、また、複合文の中から起動されたプロシージャやトリガの中でエラーが起きた場合は、例外処理コードが実行されます。

プロシージャの削除

[SQL 文] ウィンドウ枠に次のコマンドを入力して、必ずプロシージャ InnerProc と OuterProc を削除してから、チュートリアルを続けてください。

```
DROP PROCEDURE OuterProc;
 DROP PROCEDURE InnerProc
```

次に示す例では、「プロシージャとトリガでのデフォルトのエラー処理」860 ページで使用したプロシージャを基にして、例外ハンドラの処理を示します。ここでは InnerProc プロシージャ中の「カラムが見つかりません」エラーを処理するために、コードが追加されています。

```
CREATE PROCEDURE OuterProc()
 BEGIN
     MESSAGE 'Hello from OuterProc.' TO CLIENT;
     CALL InnerProc();
     MESSAGE 'SQLSTATE set to ',
         SQLSTATE,' in OuterProc.' TO CLIENT
 END
 CREATE PROCEDURE InnerProc()
 BEGIN
      DECLARE column_not_found
      EXCEPTION FOR SQLSTATE '52003';
      MESSAGE 'Hello from InnerProc.' TO CLIENT;
   SIGNAL column_not_found;
   MESSAGE 'Line following SIGNAL.' TO CLIENT;
      EXCEPTION
           WHEN column_not_found THEN
           MESSAGE 'Column not found handling.' TO
CLIENT;
           WHEN OTHERS THEN
        RESIGNAL ;
 END
```

EXCEPTION 文は例外ハンドラを宣言します。これ以降の文はエラーが起きないかぎり実行されません。WHEN 句は例外名 (DECLARE 文で宣言) と、その例外が起こったときに実行する文を定義します。WHEN OTHERS THEN 句はその前の WHEN 句以外で例外が起こったときに実行する文を定義します。

この例では、RESIGNAL は例外を上位レベルの例外ハンドラに渡します。WHEN OTHERS THEN が例外ハンドラ中に定義されていない場合は、RESIGNAL がデフォルト処理になります。

OuterProc プロシージャを実行するには、次の文を入力してください。

```
CALL OuterProc();
```

Interactive SQL の [メッセージ] ウィンドウ枠に、次のメッセージが表示されます。

```
Hello from OuterProc. Hello from InnerProc. Column not
found handling. SQLSTATE set to 00000 in OuterProc.
```

注意

- InnerProc の SIGNAL 文に続く行ではなく、EXCEPTION 文が実行されます。
- 「カラムが見つかりません」エラーのため、エラー処理のための MESSAGE 文が実行され、SQLSTATE は 0 にリセットされます (エラーは起こらなかったことを示します)。
- 例外処理コードが実行された後、制御は OuterProc に戻され、OuterProc はエラーがなかったかのように前へ進みます。
- ON EXCEPTION RESUME は指定した例外処理とは一緒に使えません。ON EXCEPTION RESUME が含まれていると、例外処理コードは実行されません。
- 「カラムが見つかりません」例外に対するエラー処理コードが単に RESIGNAL 文のとき、制御は OuterProc に戻され、SQLSTATE は値 52003 に設定されたままです。これは、InnerProc にはエラー処理コードがないのと同じです。OuterProc にはこれ以外のエラー処理コードはないため、プロシージャはエラーになります。

例外処理とアトミックな複合文

例外が複合文内で処理されるとき、複合文はアクティブな例外なしで完了し、例外より前の変更は取り消されません。これはアトミックな複合文でも同じです。アトミックな複合文の中でエラーが発生し、明示的に処理されると、複合文のいくつかの文は実行されます。

ネストされた複合文と例外処理

エラーを引き起こした文に続くコードは、プロシージャ定義に ON EXCEPTION RESUME 句が含まれる場合のみ、実行されます。

ネストされた複合文を使用すると、エラーの後にどの文が実行され、どの文が実行されないのかを制御できます。

プロシージャの削除

[SQL 文] ウィンドウ枠に次のコマンドを入力して、必ずプロシージャ InnerProc と OuterProc を削除してから、チュートリアルを続けてください。

```
DROP PROCEDURE OuterProc;
 DROP PROCEDURE InnerProc
```

次のプロシージャの例は、ネストされた複合文をどのように使用してフローを制御するかを示します。プロシージャは、「プロシージャとトリガでのデフォルトのエラー処理」860 ページの例に使用したものに基づいています。

```
CREATE PROCEDURE InnerProc()
 BEGIN
     BEGIN
         DECLARE column_not_found
             EXCEPTION FOR SQLSTATE VALUE '52003';
             MESSAGE 'Hello from InnerProc' TO CLIENT;
             SIGNAL column_not_found;
                 MESSAGE 'Line following SIGNAL' TO
CLIENT
                 EXCEPTION
             WHEN column_not_found THEN
                 MESSAGE 'Column not found handling' TO
                 CLIENT;
             WHEN OTHERS THEN
                 RESIGNAL;
     END;
         MESSAGE 'Outer compound statement' TO CLIENT;
 END
```

InnerPro プロシージャを実行するには、次の文を入力してください。

```
CALL InnerProc();
```

Interactive SQL の [メッセージ] ウィンドウ枠に、次のメッセージが表示されます。

```
Hello from InnerProc Column not found handling Outer
compound statement
```

エラーを引き起こした SIGNAL 文が検出されると、制御は複合文の例外ハンドラに渡されて、「`Column not found handling`」メッセージが出力されます。次に制御は外部複合文に渡され、「`Outer compound statement`」メッセージが出力されます。

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中に文を組み立てることができます。

次に示すのは、テーブルを作成する EXECUTE IMMEDIATE 文を含むプロシージャの例です。

```
CREATE PROCEDURE CreateTableProc(
         IN tablename char(30) )
 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',
    'SYSTABLE',
    'table_id <= 10')
```

table_id	table_name
1	SYSTABLE
2	syscolumns
3	SYSINDEX
...	...

アトミックな複合文中では、COMMIT を行う EXECUTE IMMEDIATE 文は使えません。COMMIT 文はこのコンテキストでは許可されていません。

EXECUTE IMMEDIATE 文の詳細については、『ASA SQL リファレンス・マニュアル』> 「EXECUTE IMMEDIATE 文 [SP]」を参照してください。

プロシージャとトリガでのトランザクションとセーブポイント

プロシージャまたはトリガ中の SQL 文は現在のトランザクションの一部です (「トランザクションと独立性レベル」133 ページを参照)。1 つのトランザクション中で複数のプロシージャを呼び出すことや、1 つのプロシージャ中に複数のトランザクションを持つことができます。

アトミックな文中では COMMIT と ROLLBACK は許可されません (「アトミックな複合文」843 ページを参照)。トリガはアトミックな文である INSERT、UPDATE、DELETE によって起動されることに注意してください。COMMIT と ROLLBACK はトリガまたはトリガから呼び出されたプロシージャ中では許可されません。

プロシージャまたはトリガではセーブポイント (「トランザクション内のセーブポイント」137 ページを参照) を使用できますが、ROLLBACK TO SAVEPOINT 文はアトミック・オペレーションが開始される以前のセーブポイントを参照することはできません。また、アトミック・オペレーション内のすべてのセーブポイントは、その操作が終了したときに解除されます。

プロシージャを作成するときのヒント

この項では、プロシージャを作成するためのヒントをいくつか説明します。

コマンド・デリミタを変更する必要があるかどうかをチェックする

Interactive SQL または Sybase Central でプロシージャを作成するときには、コマンド・デリミタの変更は必要ありません。他のブラウズ・ツールを使う場合には、コマンド・デリミタをセミコロンから他の文字に変更する必要がある場合があります。

プロシージャ内の各文はセミコロンで終わります。ブラウズするアプリケーションが CREATE PROCEDURE 文自体を解析するには、コマンド・デリミタにセミコロン以外の文字を使用する必要があります。

コマンド・デリミタを変更する必要があるアプリケーションを使用する場合は、コマンド・デリミタとして 2 つのセミコロン (;;) を使うか、複数の文字を使ったデリミタが許可されないシステムであれば疑問符 (?) が適切です。

プロシージャの中で文を区切る

プロシージャ中の各文はセミコロンで終わります。最後の文はセミコロンがなくてもかまいませんが、各文の後ろにはセミコロンを付けることを習慣にしてください。

CREATE PROCEDURE 文は本体である複合文と、RESULT 指定の両方を含みます。キーワード BEGIN または END の後と、RESULT 句の後には、セミコロンは必要ありません。

プロシージャ内のテーブル名

プロシージャがテーブルを参照する場合は、テーブル名に必ず所有者 (作成者) の名前をプレフィクスとして付けてください。

プロシージャがテーブルを参照するときは、プロシージャ作成者のグループ・メンバシップを使い、所有者の名前は指定しません。たとえば、user_1 が作成したプロシージャが Table_B を参照し、Table_B の所有者の名前を指定しないとします。この場合、Table_B が user_1 によって作成されたか、user_1 が Table_B の所有者であるグループの (直接または間接的に) メンバでなければなりません。どちらの条件も満たされない場合は、プロシージャが呼び出されると、「テーブルが見つかりません」というメッセージが表示されます。

テーブルの指定に相関名を使うと、テーブルの長い、完全に修飾された名前を入力しないですみます。相関名については、『ASA SQL リファレンス・マニュアル』> 「FROM 句」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。

プロシージャでデータ文字列を使用するときには、あいまいでない *yyyy-mm-dd* か *yyyy/mm/dd* の日付フォーマットを使用します。DATE_ORDER データベース・オプションの設定に関係なく、サーバはこれらの文字列を日付として正しく解釈します。

日付と時刻の詳細については、『ASA SQL リファレンス・マニュアル』> 「日付と時刻データ型」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] ウィンドウ枠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
```

ストアド・プロシージャ・デバッガを使用することもできます。

バッチに使用できる文

バッチには、ほとんどの SQL 文 (CREATE TABLE、ALTER TABLE などのデータ定義文を含む) を使用することができますが、次の文は使用できません。

- CONNECT 文または DISCONNECT 文
- ALTER PROCEDURE 文または ALTER FUNCTION 文
- CREATE TRIGGER 文
- INPUT、OUTPUT などの Interactive SQL コマンド
- バッチにホスト変数を使用することはできません。

CREATE PROCEDURE は許可されますが、必ずバッチの最後の文にします。したがって、バッチには単一の CREATE PROCEDURE 文しか含めることはできません。

バッチで SELECT 文を使用する

バッチには 1 つまたは複数の SELECT 文を含めることができます。

次に有効なバッチの例を示します。

```
IF EXISTS(    SELECT *
                 FROM SYSTABLE
                 WHERE table_name='employee' )
 THEN
     SELECT    emp_lname AS LastName,
                 emp_fname AS FirstName
     FROM employee;
     SELECT lname, fname
     FROM customer;
     SELECT last_name, first_name
     FROM contact;
 END IF
```

結果セットのエイリアスは、最初の SELECT 文でのみ必要です。サーバはバッチ中の最初の SELECT 文を結果セットの記述に使用するからです。

各クエリの後には、次の結果セットを取り出すための RESUME 文が必要です。

プロシージャからの外部ライブラリの呼び出し

ストアド・プロシージャまたはユーザ定義関数から外部ライブラリの関数を呼び出すことができます。Windows オペレーティング・システムでは DLL、NetWare では NLM、UNIX では共有オブジェクトの関数を呼び出すことができます。Windows CE では、外部関数を呼び出すことができません。

この項では、プロシージャ中で外部ライブラリ呼び出しを使用する方法について説明します。サンプルの外部ストアド・プロシージャと、プロシージャに含まれる DLL を構築するために必要なファイルは、フォルダ **%ASANY9%\Samples¥Asa¥ExternalProcedures** に格納されています。

警告
プロシージャから呼び出された外部ライブラリは、サーバのメモリを共有します。プロシージャから呼び出した外部ライブラリがメモリ処理のエラーを含んでいると、サーバそのものがクラッシュしたり、データベースが損傷したりする可能性があります。運用データベースに配備する前にライブラリをテストする必要があります。

この項で説明する API は、以前の API の代わりに使用します。バージョン 7.0 以前の古い API を使用して作成されたライブラリもサポートされますが、新しく開発する場合は、最新の API を使用してください。

Adaptive Server Anywhere は、MAPI 電子メールの送信などでこの機能を使用するシステム・プロシージャのセットを含みます。

システム・プロシージャの詳細については、『ASA SQL リファレンス・マニュアル』> 「システム・プロシージャとシステム関数」を参照してください。

外部呼び出しを使ったプロシージャと関数の作成

ここでは、外部呼び出しを使ったプロシージャと関数の例を示します。

DBA 権限
外部ライブラリを参照するプロシージャまたは関数を作成するには、DBA 権限が必要です。その他のプロシージャまたは関数の作成には RESOURCE 権限が必要ですが、外部プロシージャまたは関数の作成には DBA 権限が必須です。

構文

DDL ライブラリ *library.dll* の関数 *function_name* を呼び出すプロシージャは、次のように作成できます。

```
CREATE PROCEDURE dll_proc ( parameter-list )
 EXTERNAL NAME 'function_name@library.dll'
```

プロシージャから外部 DLL を呼び出す場合、そのプロシージャは他のタスクを実行することはできません。プロシージャは単に DLL の入れ物 (ラッパ) になってしまいます。

上記のプロシージャと同じ機能を果たすユーザ定義関数を次に示します。

```
CREATE FUNCTION dll_func ( parameter-list )
 RETURNS data-type
 EXTERNAL NAME 'function_name@library.dll'
```

これらの文中の *function_name* はダイナミック・リンク・ライブラリ (DLL) のエクスポートされる関数名で、*library.dll* はライブラリの名前です。*parameter-list* の引数の型と順序は、ライブラリ関数によって定義されている引数と対応しなければなりません。ライブラリ関数は、「外部関数のプロトタイプ」881 ページで説明した API を使ってプロシージャ引数にアクセスします。

外部関数から返された値は、プロシージャから、呼び出しを行った環境に返されます。

他の文は使用不可

外部関数を参照するプロシージャは、その他の文を含むことはできません。このプロシージャの目的は、関数の引数を取ること、関数を呼び出すこと、関数から返ってきた値と引数を呼び出し元の環境に返すことです。IN、INOUT、OUT パラメータは、通常のプロシージャの場合と同じように使用できます。入力された値は外部関数に渡され、関数によって変更を受けたパラメータは OUT または INOUT パラメータを通して呼び出しを行った環境に返されます。

オペレーティング・システムに依存する呼び出し

あるオペレーティング・システムではある関数を呼び出し、もう 1 つのオペレーティング・システムでは (おそらく同じ機能の) 別の関数を呼び出せます。この場合の構文は、関数名にオペレーティング・システム名をプレフィクスとして付けます。次に例を示します。

```
CREATE PROCEDURE dll_proc ( parameter-list )
 EXTERNAL NAME

'Windows95:95_fn@95_lib.dll;WindowsNT:nt_fn@nt_lib.dll'
```

オペレーティング・システムの識別子は **WindowsNT**、**Windows95**、**UNIX**、**NetWare** のいずれかでなければなりません。

関数のリストにサーバのオペレーティング・システムのエントリがなく、オペレーティング・システムが指定されていないエントリを含む場合、データベース・サーバはそのエントリのこの関数を呼び出します。

NetWare での呼び出しは他のオペレーティング・システムと少し異なります。NetWare では記号はすべてグローバルに知られており、エクスポートされる関数名のような記号は、システム上のすべての NLM に対してユニークでなければなりません。したがって NLM がすでにロードされているかぎり、NLM 名は呼び出し中には必要ありません。

NLM がすでにロードされているかどうかに関係なく、常にライブラリ名を使用することをおすすめします。NLM がまだロードされて*いない*場合は、ライブラリ名を指定する*必要があります*。ファイル拡張子 *.nlm* はオプションです。

CREATE PROCEDURE 文の構文の詳細については、『ASA SQL リファレンス・マニュアル』> 「CREATE PROCEDURE 文」を参照してください。

CREATE FUNCTION 文の構文の詳細については、『ASA SQL リファレンス・マニュアル』> 「CREATE FUNCTION 文」を参照してください。

外部関数のプロトタイプ

ここでは、外部ライブラリの関数のための API について説明します。

API は、SQL Anywhere Studio インストール・ディレクトリの *h* サブディレクトリのヘッダ・ファイル *extfnapi.h* によって定義されます。このヘッダ・ファイルは、外部関数のプロトタイプのプラットフォームに依存する機能を処理します。この API は、外部ライブラリの関数のための以前の API に代わるものです。

API バージョンの宣言

外部ライブラリが古い API を使って作成されていないことをデータベース・サーバに通知するには、次のように関数を指定します。

uint32 extfn_use_new_api( )

関数は符号なし 32 ビット整数値を返します。戻り値がゼロでない場合、データベース・サーバは古い API を使用していないとみなします。

関数が DLL によってエクスポートされない場合、データベース・サーバは古い API が使用中であるとみなします。新しい API を使用する場合、戻り値は *extfnapi.h* に定義された API バージョン番号とします。

NetWare の場合、外部プロシージャが新しい API を使用して作成されていることをデータベース・サーバに通知するには、NLM で extfn_use_new_api と呼ばれる関数か、*name*_use_new_api と呼ばれる関数のいずれかをエクスポートする必要があります。*name* は NLM の名前です。たとえば、*external.nlm* という名前の NLM は、関数 external_use_new_api をエクスポートします。

*name*_use_new_api をエクスポートすることで、一度に 2 つ以上の外部 NLM が使用される場合に、エクスポート名の競合を防げます。NLM から *name*_use_new_api と呼ばれる関数をエクスポートする場合は、CREATE PROCEDURE 文または CREATE FUNCTION 文に NLM 名を含める必要があります。

関数のプロトタイプ

関数名は、CREATE PROCEDURE または CREATE FUNCTION 文に参照された関数名と一致しなければなりません。関数宣言は、次のようにします。

void *function-name*(**an_extfn_api** *****_api_, **void** *****_argument-handle_)

この関数は、戻り値はなく、引数を渡すために使った構造体と SQL プロシージャによって指定された引数へのハンドルを引数として使用します。

an_extfn_api 構造体の形式は次のとおりです。

```
typedef struct an_extfn_api {
     short (SQL_CALLBACK *get_value)(
                     void *          arg_handle,
                     a_SQL_uint32    arg_num,
                     an_extfn_value *value
                     );
     short (SQL_CALLBACK *get_piece)(
                     void *          arg_handle,
                     a_SQL_uint32    arg_num,
                     an_extfn_value *value,
                     a_SQL_uint32    offset
                     );
     short (SQL_CALLBACK *set_value)(
                     void *          arg_handle,
                     a_SQL_uint32    arg_num,
                     an_extfn_value *value
                     short          append
                     );
 void (SQL_CALLBACK *set_cancel)(
                     void *    arg_handle,
                     void *    cancel_handle
                     );
 } an_extfn_api;
```

an_extfn_value 構造体の形式は次のとおりです。

```
typedef struct an_extfn_value {
     void *                data;
     a_SQL_uint32         piece_len;
     union {
         a_SQL_uint32    total_len;
         a_SQL_uint32    remain_len;
     } len;
     a_SQL_data_type        type;
 } an_extfn_value;
```

注意

OUT パラメータで **get_value** を呼び出すと、引数のデータ型が返され、データは NULL として返されます。

任意の引数の **get_piece** 関数は、同じ引数の **get_value** 関数の直後にのみ呼び出すことができます。

NULL を返すには、**an_extfn_value** 構造体において **data** を NULL に設定します。

set_value の **append** フィールドは、指定されたデータを既存のデータと置き換えるか (false)、または既存のデータに追加するか (true) を指定します。`append=FALSE` を指定して **set_value** を呼び出してから、同じ引数に対して `append=TRUE` を指定して呼び出してください。固定長データ型の場合、**append** フィールドは無視されます。

ヘッダ・ファイル自体が追加のノートを含みます。

次の表に、**an_extfn_api** に定義されている関数が false を返す条件を示します。

関数	true の場合は 0 を返し、それ以外の場合は 1 を返す条件
get_value()	- **arg_num** が無効な場合。たとえば、**arg_num** が **ext_fn** の引数の数より大きい場合。 – 外部関数呼び出しが正常に初期化される前に呼び出される場合。
get_piece()	- **arg_num** が無効な場合。たとえば、**arg_num** が最後の **get_value** に対応していない場合。| – オフセットが arg_num 引数の値の合計長より大きい場合。 – 外部関数呼び出しが正常に初期化される前に呼び出される場合。
set_value()	- **arg_num** が無効な場合。たとえば、**arg_num** が **ext_fn** の引数の数より大きい場合。- **arg_num** 引数が入力専用の場合。 – 与えられた値の型が **arg_num** 引数の型と一致しない場合。 – 外部関数呼び出しが正常に初期化される前に呼び出される場合。

a_sql_data_type フィールドに入力できる値の詳細については、『ASA プログラミング・ガイド』> 「Embedded SQL のデータ型」を参照してください。

外部関数にパラメータを渡す場合の詳細については、「パラメータを外部関数に渡す」885 ページを参照してください。

取り消し処理の実現

取り消されることが予想される外部関数は、**set_cancel** API 関数を呼び出してデータベース・サーバに通知します。外部動作を取り消すため特殊関数をエクスポートします。この関数は、次の形式にします。

void an_extfn_cancel(void * *cancel_handle* **)**

DLL がこの関数をエクスポートできない場合、データベース・サーバは DLL の関数のユーザ割り込みを無視します。この関数では、*cancel_handle* は上記の **an_extfn_api** 構造体に示した *set_cancel* API 関数による外部関数への呼び出しごとに取り消される関数によってデータベース・サーバへ提供されるポインタです。

パラメータを外部関数に渡す

データ型

外部ライブラリに渡されるのは、次に示す SQL データ型です。

SQL データ型	C データ型
CHAR	指定した長さの文字データ
VARCHAR	指定した長さの文字データ
LONG VARCHAR	指定した長さの文字データ
BINARY	指定した長さのバイナリ・データ
LONG BINARY	指定した長さの文字データ
TINYINT	1 バイト整数
[UNSIGNED] SMALLINT	[符号なし] 2 バイト整数
[UNSIGNED] INT	[符号なし] 4 バイト整数
[UNSIGNED] BIGINT	[符号なし] 8 バイト整数

SQL データ型	C データ型
VARBINARY	指定した長さのバイナリ・データ
REAL	単精度浮動小数点数
DOUBLE	倍精度浮動小数点数

日付または時刻データ型は使用できず、真数値データ型も使用できません。

INOUT または OUT パラメータに値を指定するには、**set_value** API 関数を使用します。IN と INOUT パラメータを読み取るには、**get_value** API 関数を使用します。

NULL を渡す

すべての引数に有効な値として NULL を渡すことができます。外部ライブラリの関数は、すべてのデータ型のリターン・タイプとして NULL を渡すことができます。

外部関数のリターン・タイプ

次の表に、サポートされているリターン・タイプと、それがどのように SQL 関数またはプロシージャにマップされているかを示します。

C データ型	SQL データ型
void	外部プロシージャに使用する
char *	CHAR() を返す関数
long	INTEGER を返す関数
float	FLOAT を返す関数
double	DOUBLE を返す関数

外部ライブラリの関数が NULL を返し、SQL 外部関数が CHAR() を返すように宣言された場合、SQL 拡張関数の戻り値は NULL 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、関連オブジェクトの内容にスクランブルをかけて読み取れないようにします。これらのオブジェクトは引き続き使用できます。また、アンロードして、別のデータベースに再ロードすることもできます。

修正を元に戻すことはできません。バージョン 8.0 以降を使用して作成したデータベースの場合は、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。

ストアド・プロシージャ・デバッガによるデバッグでは、プロシージャ定義が表示されず、プロシージャ・プロファイリングにもソースが表示されません。

すでに隠されているオブジェクトに対して、前述のいずれかの文を実行しても効果はありません。

特定の型のすべてのオブジェクトのテキストを隠すには、次のようなループを使用できます。

```
begin
     for hide_lp as hide_cr cursor for
         select proc_name,user_name
         from SYS.SYSPROCEDURE p, SYS.SYSUSERPERM u
         where p.creator = u.user_id
         and p.creator not in (0,1,3)
     do
         message 'altering ' || proc_name;
         execute immediate 'alter procedure "' ||
             user_name || '"."' || proc_name
             || '" set hidden'
     end for
 end
```

詳細については、『ASA SQL リファレンス・マニュアル』> 「ALTER FUNCTION 文」、『ASA SQL リファレンス・マニュアル』> 「ALTER PROCEDURE 文」、『ASA SQL リファレンス・マニュアル』> 「ALTER TRIGGER 文」、『ASA SQL リファレンス・マニュアル』> 「ALTER VIEW 文」を参照してください。

第 21 章

データベースでのデバッグ論理

この章の内容

この章では、Sybase デバッガを利用して、SQL ストアド・プロシージャ、トリガ、イベント・ハンドラ、Java ストアド・プロシージャの開発に役立てる方法について説明します。

データベースにおけるデバッグの概要

次のオブジェクトの開発中にデバッガを使用できます。

- SQL 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
- データベース内の Java ストアド・プロシージャ

この章では、デバッガを設定して使用する方法について説明します。

別途ライセンスを入手可能なコンポーネント

データベース内の Java に関するコンポーネントには別途ライセンスが必要です。インストールする場合はご注文ください。このコンポーネントのご注文については、ご購入いただいた販売代理店または弊社営業担当までご連絡ください。

デバッガの機能

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・プロシージャと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。インストールされた Java クラスのソース・コードも、ディスクに保存されているかぎりブラウズできます。
- **Java クラスのデバッグ**　データベースに格納する Java クラスをデバッグできます。
- **実行のトレース**　データベース内で実行している Java クラスまたはストアド・プロシージャのコードを、1 行ずつステップして移動します。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することができます。Java アプリケーションに特定の例外がスローされた場合、常に停止させることもできます。
- **ローカル変数の検査と修正**　実行がブレークポイントで停止したときに、ローカル変数の値を検査して変更できます。
- **式を検査してブレークする**　実行がブレークポイントで停止したときに、さまざまな式の値を検査できます。
- **ロー変数の検査と修正**　ロー変数はローレベル・トリガの OLD と NEW の値です。これらの値を検査して修正できます。
- **クエリの実行**　実行が SQL プロシージャのブレークポイントで停止したときに、クエリを実行できます。これによって、テンポラリ・テーブルに保持される中間結果を参照したり、ベース・テーブルの値をチェックして、クエリ実行プランを参照したりできます。

Java デバッガの稼働条件

デバッガを使用するには次のものが必要です。

- **パーミッション**　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。
- **Java クラスのソース・コード**　デバッガがアプリケーションのソース・コードを利用できるようにします。Java クラスでは、ハード・ディスクのディレクトリにソース・コードが保存されています。ストアド・プロシージャでは、データベースにソース・コードが保存されています。

- **コンパイル・オプション**　Java クラスをデバッグするには、デバッグ情報を含めてコンパイルします。たとえば、Sun Microsystems の JDK コンパイラ *javac.exe* を使用している場合には、-g コマンド・ライン・オプションを使ってコンパイルします。

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、デバッガの起動方法、データベースへの接続方法、簡単なストアド・プロシージャのデバッグ方法、Java クラスのデバッグ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

このチュートリアルは、デバッガの起動方法、データベースへの接続方法、デバッグのための接続を形成する方法について説明します。このチュートリアルでは、Adaptive Server Anywhere のサンプル・データベースを使用します。

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. Sybase Central を起動します。

   [スタート] － [プログラム] － [SQL Anywhere 9] － [Sybase Central] を選択します。

2. データベースに接続します。

   このチュートリアルでは、Adaptive Server Anywhere のサンプル・データベースに接続します。

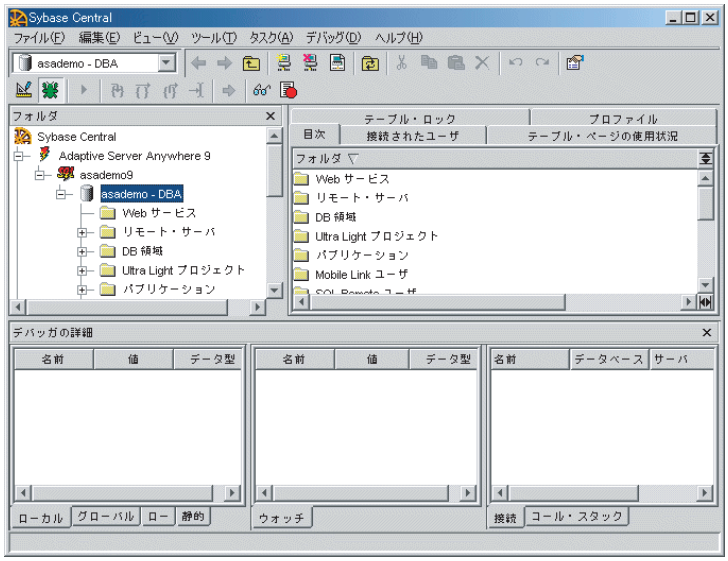
   a. 左ウィンドウ枠で、[Adaptive Server Anywhere 9] を右クリックし、ポップアップ・メニューで [接続] を選択します。

   [接続] ダイアログが表示されます。

3. デバッグ・モードを選択します。

Sybase Central は、設計モードまたはデバッグ・モードで使用できます。デバッグ・モードで動作中、デバッガ・ブレークポイントはアクティブになります。また、Sybase Central にデバッグ・メニューと [デバッガの詳細] ウィンドウ枠が表示されます。

デバッグ・モードを選択するには、[タスク] ー [デバッグ] をクリックします。Sybase Central の最下部に [デバッガの詳細] ウィンドウ枠が表示され、Adaptive Server Anywhere 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。



レッスン 2：ストアド・プロシージャのデバッグ

このチュートリアルでは、ストアド・プロシージャをデバッグするサンプル・セッションを説明します。これは、「レッスン 1：データベースへの接続とデバッガの起動」893 ページの続きです。

このチュートリアルでは、デバッガを使用してストアド・プロシージャのエラーを識別する方法を説明します。この目的のために、サンプル・データベースの debugger_tutorial ストアド・プロシージャ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] フォルダを開きます。

2. プロシージャを実行します。

   debugger_tutorial プロシージャを右クリックし、ポップアップ・メニューで [Interactive SQL から実行] を選択します。

   [Interactive SQL] ウィンドウが開き、次の結果セットが表示されます。

top_company	top_value
(NULL)	(NULL)

これは、明らかに正しい結果ではありません。チュートリアルの続きで、この結果をもたらしたエラーを診断します。

3 [Interactive SQL] ウィンドウを閉じて、作業領域をクリアします。

バグの診断

プロシージャのバグを診断するために、プロシージャにブレークポイントを設定して、コードをステップ・スルーし、プロシージャの実行とともに変化する変数の値を監視します。

ここでは、プロシージャ内の最初の実行可能な文にブレークポイントを設定します。

❖ バグを診断するには、次の手順に従います。

1 Sybase Central をデバッグ・モードに変更します。

[タスク] ー [デバッグ] を選択します。Sybase Central のメイン・ウィンドウの最下部に [デバッガの詳細] ウィンドウ枠が表示されます。

2 プロシージャ内の最初の実行可能な文にブレークポイントを設定します。

その文には次のテキストが含まれます。

```
open cur_this_cust;
```

この行の左側にあるグレーの縦線をクリックして、ブレークポイントを設定します。ブレークポイントは、赤い円で表示されます。[F9] キーを押してブレークポイントを設定する方法もあります。

3 プロシージャを再度実行します。

a. 左ウィンドウ枠で、debugger_tutorial プロシージャを右クリックし、ポップアップ・メニューで [Interactive SQL から実行] を選択します。

b. Interactive SQL からの接続をデバッグするかどうかをたずねるメッセージ・ボックスが表示されます。[はい] をクリックします。

プロシージャの実行がブレークポイントで停止します。[ソース・コード] ウィンドウ内の黄色の矢印は、現在位置を示します。そこがブレークポイントです。

4 変数を検査します。

[デバッガの詳細] ウィンドウ枠の [ローカル変数] ウィンドウに、プロシージャ内の変数が現在の値とデータ型とともにリストされます。変数 top_company、top_value、this_value、this_company は、いずれも初期化されていないため NULL です。

5 コードをステップ・スルーします。

次に示す行に達するまで、[F11] キーを何度か押してコードをステップ・スルーします。

```
if this_value > top_value then
```

ストアド・プロシージャの行をステップ・スルーするにつれて、変数の値が変化します。

`if` 文を見ると、this_value は 3000 ですが、top_value は NULL のままです。

6 もう 1 つ先の文にステップします。

[F11] キーをもう一度押して、分岐を確認します。黄色の矢印が、次のテキストを含む、ループの開始点のラベル文に戻ります。

```
customer loop: loop
```

`if` テストは TRUE を返しませんでした。テストが失敗したのは、NULL とどのような値を比較しても NULL が返されるためです。NULL 値によりテストは失敗し、`if...end if` 文内のコードは実行されませんでした。

このことから、top_value が初期化されていないことが原因とわかります。

診断内容を確認し、バグを修正する

top_value が初期化されていないことが原因であるという仮説を、プロシージャ・コードを変更せずに、デバッガでテストできます。

❖ 仮説をテストするには、次の手順に従います。

1. top_value に値を設定します。

   [ローカル] ウィンドウで、変数 top_value の [値] フィールドをクリックし、値 3000 を入力します。

2. ループを再度ステップ・スルーします。

   [F11] キーを押して命令を `if` 文までステップ・スルーし、this_value と top_value の値を確認します。ループを何度かステップ・スルーすると、top_value の値が 3000 より大きくなります。

3. ブレークポイントを無効にし、プロシージャを実行します。

   a. ブレークポイントをクリックして、グレーにします (無効になります)。

   b. [F5] キーを押してプロシージャの実行を終了します。

   [Interactive SQL] ウィンドウが再び表示されます。正しい結果が表示されています。

top_company	top_value
Chadwicks	8076

仮説が正しいことが確認されました。top_value が初期化されていないことが原因でした。

❖ バグを修正するには、次の手順に従います。

1. [タスク] － [設計] を選んで、デバッグ・モードから出ます。
2. 次のテキストが含まれる行の直後に、

   ```
   open cur_this_cust;
   ```

   top_value 変数を初期化する新しい行を作成します。

   ```
   set top_value = 0;
   ```

3. [CTRL+S] キーを押して、修正したプロシージャを保存します。
4. プロシージャを再度実行し、Interactive SQL に正しい結果が表示されることを確認します。

このレッスンは終了です。Sybase Central を閉じ、開いている [Interactive SQL] ウィンドウを閉じます。

レッスン 3 : Java クラスのデバッグ

このレッスンでは、Java クラスをデバッグするサンプル・セッションを説明します。

このレッスンでは、データベース・コンポーネントに Java が含まれている必要があります。

別途ライセンスを取得できるオプションが必要
データベース内の Java に関するコンポーネントには別途ライセンスが必要です。インストールする場合はご注文ください。

このコンポーネントのご注文については、『SQL Anywhere Studio の紹介』> 「別途ライセンスが入手可能なコンポーネント」を参照してください。

このレッスンでは、Interactive SQL から JDBCExamples.Query() を呼び出し、デバッガでの実行を中断してこのメソッドのソース・コードをトレースします。

JDBCExamples.Query() メソッドは、サンプル・データベースに対して次のクエリを実行します。

```
SELECT id, unit_price
 FROM product
```

その後、結果セットのすべてのローを検索し、最も高い単価を持つローを返します。

Java クラスをデバッグ用にコンパイルする

Java クラスをデバッグするには、javac -g オプションを指定してクラスをコンパイルします。デバッグ用にサンプル・クラスがコンパイルされています。

データベースの準備

このチュートリアルに従って作業するには、サンプル・データベースで Java を使用可能にし、*JDBCExamples.class* をサンプル・データベースにインストールします。

手順については、『ASA プログラミング・ガイド』> 「Java のサンプルの設定」を参照してください。

JDBCExamples クラスとそのメソッドの詳細については、『ASA プログラミング・ガイド』> 「JDBC プログラミング」を参照してください。

Java ソース・コードをデバッガで表示

デバッガは、いくつかのロケーションで (*.java* 拡張子が付いている) ソース・コード・ファイルを検索します。データベースで現在実行されているクラスのコードをデバッガが利用できるようにするためには、インストール・ディレクトリの *Samples¥ASA¥Java* サブディレクトリをロケーションのリストに追加する必要があります。

❖ Java のソース・コードをデバッガで表示するには、次の手順に従います。

1. Sybase Central を起動し、サンプル・データベース (ASA 9.0 Sample ODBC データ・ソース) に接続します。

2 Sybase Central のデバッグ・タスクを選択します。

[タスク] － [デバッグ] を選択します。

3 Java ソース・コードを検索するロケーションを設定します。

a. [デバッグ] － [Java ソース・パスの設定] を選択します。[Java ソース・パス] ウィンドウが表示されます。

b. [フォルダの参照] をクリックします。Java ソース・コード (*.java* ファイル) が格納されているフォルダに移動します。JDBCExamples クラスのソース・コードを参照するには、SQL Anywhere インストール・ディレクトリの *Samples¥ASA¥Java* サブディレクトリに移動します。SQL Anywhere をデフォルトのディレクトリにインストールしている場合は、次のディレクトリに移動します。

```
C:¥Program Files¥Sybase¥SQL Anywhere 9¥Samples¥
ASA¥Java
```

4 JDBCExamples クラスのソース・コードが表示されます。

a. Sybase Central の左ウィンドウ枠で [Java オブジェクト] フォルダを開きます。

b. 右ウィンドウ枠で、[すべての Java クラス] フォルダを開き、デバッグする JAR ファイルまたはクラスを見つけます。Sybase Central の設定によっては、[作成者] カラムをクリックすると便利です。これにより、リストが作成者によってソートされ、DBA が所有するクラスが SYS の所有するクラスより前に表示されます。

c. JDBCExamples クラスをダブルクリックします。

d. 右ウィンドウ枠で、[ソース] タブをクリックします。クラスのソース・コードが表示されます。

ブレークポイントの設定

❖ Java クラスに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 [ソース・コード] ウィンドウで、**Query** メソッドの最初の行が表示されるまでページを下にスクロールします。このメソッドはクラスの最後の方にあり、次の行で始まります。

```
public static int Query() {
```

2 行の左側にあるグレーのカラムに赤い円が表示されるまでクリックします。

```
int max_price = 0;
```

インジケータを繰り返しクリックすると、状態が切り替わります。

メソッドの実行

❖ Interactive SQL からメソッドを呼び出すには、次の手順に従います。

1 Interactive SQL を起動します。ユーザ ID **DBA** とパスワード **SQL** を入力して、サンプル・データベースに接続します。

2 Interactive SQL で、次のコマンドを入力してメソッドを実行します。

```
SELECT JDBCExamples.Query()
```

クエリは完了しません。代わりに、ブレークポイントに達すると実行はデバッガ内で停止します。Interactive SQL では、[SQL 文の中断] ボタンが有効になっています。デバッガの [ソース] ウィンドウでは、黄色の矢印が現在の行を示しています。

この状態になると、ソース・コードをステップして、デバッガでデバッグ処理を行うことが可能です。

ソース・コードのステップ・スルー

この項では、デバッガ内でコードをステップするいくつかの方法について説明します。

前の項で、デバッガはメソッド内の最初の文で **JDBCExamples.Query** の実行を停止しています。

例

次にステップ例をいくつか示します。

1. **次の行にステップする**　[デバッグ] － [ステップ] を選択するか、[F10] キーを押して、現在のメソッドの次の行にステップします。これを 2、3 回実行してみます。

2. **カーソルまで実行する**　マウスを使って次の行を選択し、[デバッグ] － [カーソルまで実行] を選択するか、[Ctrl + F10] キーを押して、その行まで実行してから停止します。

   ```
   max_price = price;
   ```

   黄色の矢印がその行に移動します。

3. **ブレークポイントを設定してそこまで実行する**　次に示す行 (行 292) にカーソルを置き、[F9] キーを押してその行にブレークポイントを設定します。

   ```
   return max_price;
   ```

   ブレークポイントをマーク付けする赤い停止記号が、左側のカラムに表示されます。[F5] キーを押してそのブレークポイントまで実行します。

4. **実験**　コードをステップするいくつかのメソッドを試してみます。最後に [F5] キーを押して実行を完了します。

   ソース・コードをステップするための一連の完全なオプションは [デバッグ] メニューに表示されます。

   動作が完了すると、Interactive SQL の [結果] タブにある [結果] ウィンドウ枠に 24 という値が表示されます。

変数の検査と変更

このレッスンでは、デバッガでローカル変数 (メソッドで宣言) とクラスの静的変数の両方の値を検査します。

ローカル変数の検査

コードをステップしながら、メソッド内のローカル変数の値を検査することで、ここでの動作をより良く理解できます。そのためには、javac -g オプションを指定してクラスをコンパイルしておいてください。

❖ **変数の値を検査して変更するには、次の手順に従います。**

1. **JDBCExamples.Query** メソッドの 1 行目にブレークポイントを設定します。この行を次に示します。

   ```
   int max_price = 0
   ```

2. Interactive SQL で、次の文を再び入力してメソッドを実行します。

   ```
   SELECT JDBCExamples.Query()
   ```

   クエリはブレークポイントまでしか実行しません。

3. [F10] キーを押して次の行にステップします。ここで **max_price** 変数が宣言され、0 に初期化されます。

4. [ローカル] タブのリストで、**max_price** を入力する [値] カラムをダブルクリックし、45 と入力して **max_price** の値を変更します。

   45 という値は最高価格です。これで、クエリは 24 の代わりに 45 を最高価格として返します。

5. [F10] キーをもう一度押してコードをステップ・スルーします。このとき、各変数の値が [ローカル] タブにリストされます。**stmt** と **result** 変数に値が設定されるまでステップします。

6 **result** オブジェクトを拡張するために、横のアイコンをクリックするか、カーソルをその行に置き [Enter] キーを押します。これにより、オブジェクトのフィールド値が表示されます。

7 変数の検査と修正を実践したら、[F5] キーを押してクエリの実行を完了し、チュートリアルを終了します。

静的変数の検査

ローカル変数の他に、デバッガの [静的] タブにクラスレベル変数 (静的変数) を表示し、[ウォッチ] タブでその値を監視できます。詳細については、デバッガのオンライン・ヘルプを参照してください。

ブレークポイントの活用

この項では、ブレークポイントを使用して、デバッガがソース・コードの実行をいつ中断するかを制御する方法について説明します。

ブレークポイントの設定

ブレークポイントは、指定した行で実行を中断するようデバッガに指示します。

❖ ブレークポイントを設定するには、次の手順に従います。

1. デバッグ・タスクを実行している Sybase Central では、ブレークポイントを設定するコードを表示します。
2. ブレークポイントを設定するには、ウィンドウ左側のグレーのカラムをクリックするか、行をクリックして [F9] キーを押します。赤い円は、ブレークポイントが設定された行を示します。

❖ 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. [ブレークポイント] ウィンドウを表示します。

   デバッグ・タスクを実行している Sybase Central で、[デバッグ] － [ブレークポイント] を選択します。

2. [ブレークポイント] ウィンドウで、[新しいブレークポイント] をクリックします。[新しいブレークポイント] ウィンドウが表示されます。
3. ドロップダウン・リストからプロシージャ名を選択し、任意で条件とカウント値を入力します。

   条件とは、ブレークポイントが処理を中断するために TRUE に評価されなければならない SQL 式または Java 式です。たとえば、特定のユーザによる接続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'
```

カウントとは、ブレークポイントのヒット数で、その回数を超えると実行が停止されます。0 を指定した場合、常にブレークポイントによって実行が停止されます。

4 [OK] をクリックしてブレークポイントを設定します。ブレークポイントが、プロシージャ内の最初の実行可能な文に設定されます。

ブレークポイントを設定すると、すべての接続に適用されます。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を設定するには、次の手順に従います。

1 ブレークポイント・マーカを右クリックし、ポップアップ・メニューで [ブレークポイントの編集] を選択します。[ブレークポイントの編集] ダイアログが表示されます。

2 [条件] フィールドに条件を入力します。たとえば、特定のユーザ ID からの接続にのみ適用するようにブレークポイントを設定するには、次の条件を入力します。

```
CURRENT USER='user-name'
```

user-name は、ブレークポイントをアクティブにする対象のユーザ ID です。

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 ステータスを変更するブレークポイントを含むプロシージャのソース・コードを表示します。

2 編集する行の左側にあるブレークポイント・インジケータをクリックします。行のステータスは、アクティブなブレークポイントから無効化されたブレークポイントに切り替わります。

❖ ブレークポイントのステータスを変更するには、次の手順に従います ([ブレークポイント] ウィンドウの場合)。

1 [ブレークポイント] ウィンドウを開きます。

2 ブレークポイントを編集します。

3 また、ブレークポイントを選択して [Delete] キーを押すとブレークポイントを削除できます。

ブレークポイント条件の編集

ブレークポイントに条件を追加して、特定の条件または回数を満たす場合だけそのブレークポイントで実行を中断するよう、デバッガに指示を出すことができます。

❖ ブレークポイントに条件または回数を追加するには、次の手順に従います。

- [ブレークポイント] ウィンドウを開きます。

  Java クラスについては、条件を Java ブール式にします。プロシージャとトリガについては、SQL 探索条件にします。

例

プロシージャ sp_contacts では、次のブレークポイントを使用します。

```
contact.id = contact.old_id
```

これを次の行で使用します。

```
DELETE FROM contact WHERE contact.id = contact.old_id
```

Java メソッド JDBCExamples.Query() で、次のブレークポイント条件を使用する例を考えてみます。

```
(price < 10)
```

これを次の行で使用します。

```
if (max.price == price) or (price == 10)
```

変数の編集

デバッガでは、コードをステップしながら変数の動作を表示して編集できます。デバッガには、ストアド・プロシージャと Java クラス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] ウィンドウが用意されています。[デバッガの詳細] ウィンドウは、Sybase Central でデバッグ・タスクを実行しているときに、[Sybase Central] ウィンドウの最下部に表示されます。

ローカル変数

❖ **変数の値を監視するには、次の手順に従います。**

1. 検査する変数があるプロシージャにブレークポイントを設定します。

   ブレークポイントの設定については、「ブレークポイントの設定」906 ページを参照してください。

2. [デバッガの詳細] ウィンドウの [変数] ウィンドウ枠にある [ローカル] タブをクリックします。

3. プロシージャを実行します。[ローカル] タブに変数が値とともに表示されます。

その他の変数

グローバル変数は Adaptive Server 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、[変数] ウィンドウの [グローバル] タブに表示されます。

グローバル変数のリストについては、『ASA SQL リファレンス・マニュアル』> 「グローバル変数」を参照してください。

ロー変数は、トリガで使用する値を格納します。ロー変数は、[変数] ウィンドウの [ロー] タブに表示されます。

トリガの詳細については、「トリガの概要」830 ページを参照してください。

静的変数は Java クラスで使用します。静的変数は [静的] タブに表示されます。

呼び出しスタック

ネストされたプロシージャや Java クラスをデバッグするときに、呼び出しの順番を検査すると便利なことがあります。[コール・スタック] タブで、プロシージャのリストを参照できます。

❖ 呼び出しスタックを表示するには、次の手順に従います。

1. 検査する変数があるプロシージャにブレークポイントを設定します。
2. コードをブレークポイントまで実行します。

   [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。すぐ下には現在のプロシージャを呼び出したプロシージャが表示され、また、同様の呼び出し情報が表示されます。

接続の活用

[接続] ウィンドウには、データベースへの接続が表示されます。常に複数の接続が実行されています。あるものはブレークポイントで停止し、あるものは停止していません。

[ソース・コード] ウィンドウには、1 接続の状態が表示されます。接続を切り替えるには、[接続] ウィンドウ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」908 ページと『ASA SQL リファレンス・マニュアル』> 「CURRENT USER 特別値」を参照してください。

索引

記号

291
*=
Transact-SQL 外部ジョイン 357
* (アスタリスク)
SELECT 文 279
=*
Transact-SQL 外部ジョイン 357
>
比較演算子 291
@@identity グローバル変数
IDENTITY カラム 612
1 対 1 の関係
解析 24
定義 7
1 対多の関係
解析 24
定義 7
2 つのテーブルのジョイン 341
2 フェーズ・ロック
原理 187
説明 187
3 つ以上のテーブルのジョイン 342

A

Adaptive Server Anywhere
他の SQL ダイアレクトとの違い 630
Adaptive Server Anywhere SQL の機能 630
Adaptive Server Anywhere の圧縮機能の使用 218
Adaptive Server Enterprise
ASA への移行 733
GROUP BY 互換性 332
互換性 590
データのインポート／エクスポートの互換性 744
Adaptive Server のアーキテクチャ 594
aggregates
アクセス・プラン内の項目 571
ALL
キーワードと UNION 句 324
キーワードと集合関数 306
サブクエリのテスト 457, 458
ALLOW_NULLS_BY_DEFAULT オプション
Transact-SQL 互換性の設定 605
ALL 演算子
説明 457
注意 458
ALTER TABLE 文
外部キー 62
検査制約 114
同時性 194
プライマリ・キー 59
例 55
AND
論理演算子の使用 302
ANSI
ANSI 以外のジョイン 344
SQL/92 標準と矛盾 140
ANSI update constraints
アクセス・プラン 568
ANSI 以外のジョイン 344
ANY 演算子
サブクエリのテスト 455
説明 455
問題 456
asademo.db ファイル
スキーマ 336
asajdbc サーバ・クラス 790
asaodbc サーバ・クラス 794

asejdbc サーバ・クラス　791
aseodbc サーバ・クラス　795
ASTMP 環境変数
　テンポラリ・ファイルのロケーション　216
AS キーワード
　エイリアス　282
AT 句
　CREATE EXISTING TABLE 文　762
AUTOINCREMENT
　Ultra Light アプリケーション　110
　使用する場合　193
　符号付データ型　110
　負の値　110
AUTOMATIC_TIMESTAMP オプション
　Transact-SQL 互換性の設定　605
AUTO モード
　使用　657
AvgDiskReads
　アクセス・プラン内の推定　567
AvgDiskReadTime
　アクセス・プラン内の推定　567
AvgDiskWrites
　アクセス・プラン内の推定　567
AvgRowCount
　アクセス・プラン内の推定　567
AvgRunTime
　アクセス・プラン内の推定　567

B

BCP フォーマット
　ASE でのインポートとエクスポート　744
BEGIN TRANSACTION 文
　リモート・データ・アクセス　776
BETWEEN キーワード
　範囲クエリ　293
BLOB
　説明　31
　挿入　488
BLOCKING オプション
　使用　148
B ツリー・インデックス
　説明　542

C

CacheHits
　アクセス・プラン内の統計　567
CacheRead
　アクセス・プラン内の統計　567
CacheReadIndLeaf
　アクセス・プラン内の統計　567
CacheReadTable
　アクセス・プラン内の統計　567
CALL 文
　構文　841
　説明　812
　パラメータ　847
　例　819
cardinality
　アクセス・プラン内の項目　569
CASCADE アクション
　説明　128
CASE 文
　構文　841
cdata ディレクティブ
　使用　673
CHECK 条件
　Transact-SQL　596
CLOSE 文
　プロシージャ　856
COMMIT 文
　参照整合性の検証　183
　複合文　843
　プロシージャとトリガ　873
　リモート・データ・アクセス　776
COMPUTE 句
　CREATE TABLE　64
　サポートされない　616
CONNECTION_PROPERTY 関数
　説明　246
count (*) の使用　310
COUNT 関数

NULL 311
説明 310
CREATE DATABASE 文
ASE 595
使用 40
CREATE DEFAULT 文
サポートされない 596
CREATE DOMAIN 文
Transact-SQL との互換性 596
使用 119
CREATE EXISTING TABLE 文
使用 765
CREATE FUNCTION 文
説明 825
CREATE INDEX 文
同時性 194
CREATE PROCEDURE 文
パラメータ 846
例 815
CREATE RULE 文
サポートされない 596
CREATE SERVER 文
JDBC と Adaptive Server Enterprise 791
ODBC と Adaptive Server Enterprise 796
リモート・サーバ 790
CREATE SERVER 文の USING パラメータ値 790
CREATE TABLE 文
Transact-SQL 614
外部キー 62
説明 51
同時性 194
プライマリ・キー 59
プロキシ・テーブル 766
CREATE TABLE 文を使用したプロキシ・テーブルの作成 766
CREATE TRIGGER 文
説明 831
CREATE VIEW 文
WITH CHECK OPTION 句 72
CUBE 演算
NULL 424
説明 432
例 433
CUBE の概要 434
CUBE の使用 432

D

DataSet
XML をインポートするために使用 648
リレーショナル・データを XML としてエクスポートするために使用 640
DataSet オブジェクトを使用した XML のインポート 648
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 640
DataWindows
リモート・データ・アクセス 747
DB_PROPERTY 関数
説明 246
db2odbc サーバ・クラス 797
DB2 リモート・データ・アクセス 797
dberase ユーティリティ
使用 42
dbinit ユーティリティ
使用 41
dbisql ユーティリティ
データベースの再構築 727
dbo ユーザ
ASE 597
dbunload ユーティリティ
再構築ツール 727
データのエクスポート 709
DB 領域
管理 595
DDL
説明 36
同時性 192, 194
debugger_tutorial プロシージャ
説明 895
DECLARE 文
複合文 843

プロシージャ　856, 861
DELETE 文
使用　492
ロック　185
DELETE 文、UPDATE 文、INSERT 文でのジョインの使用　343
depth
アクセス・プラン内の項目　569
direction
アクセス・プラン内の項目　569
DISCONNECT 文
使用　43
DiskRead
アクセス・プラン内の統計　567
DiskReadIndInt
アクセス・プラン内の統計　567
DiskReadIndLeaf
アクセス・プラン内の統計　567
DiskReadTable
アクセス・プラン内の統計　567
DiskWrite
アクセス・プラン内の統計　567
DISK 文
サポートされない　595
distinct list
アクセス・プラン内の項目　571
DISTINCT キーワード
集合関数　306, 310
DISTINCT 句
SELECT 文　286
不要な DISTINCT の削除　550
DISTINCT の削除
説明　550
DISTINCT を伴う集合関数の使用　310
DLL
外部プロシージャの呼び出し　881
プロシージャからの呼び出し　879
DML
説明　482
DROP CONNECTION 文
使用　43
DROP DATABASE 文
ASE　595
使用　41
DROP TABLE 文
例　56
DROP TRIGGER 文
説明　836
DROP VIEW 文
例　76
DROP 文
同時性　194
DUMP DATABASE 文
サポートされない　595
DUMP TRANSACTION 文
サポートされない　595

E

element ディレクティブ
使用　669
ER 図
説明　5
読み込み　8
ER 図の読み込み　8
EstCpuTime
アクセス・プラン内の推定　567
EstDiskReads
アクセス・プラン内の推定　567
EstDiskReadTime
アクセス・プラン内の推定　567
EstDiskWrites
アクセス・プラン内の推定　567
estimated leaf pages
アクセス・プラン内の項目　569
estimated pages
アクセス・プラン内の項目　569
estimated rows
アクセス・プラン内の項目　569
estimated row size
アクセス・プラン内の項目　569
EstRowCount
アクセス・プラン内の推定　567

EstRunTime
アクセス・プラン内の推定　567
Excel とリモート・アクセス　804
EXCEPT 演算
NULL　328
クエリの結合　324
使用　325
ルール　326
EXECUTE IMMEDIATE 文
プロシージャ　871
EXISTS 演算子　461
EXISTS 述部
サブクエリを書き換え　505
EXPLICIT モード
cdata ディレクティブの使用　673
element ディレクティブの使用　669
hide ディレクティブの使用　670
xml ディレクティブの使用　672
クエリの記述　664
構文　661
使用　661
EXPLICIT モードのクエリの記述　664
expression SQL
アクセス・プラン内の項目　572

F

FALSE 条件
NULL　301
FALSE と UNKNOWN の違い　301
fetchtst　257
FETCH 文
プロシージャ　856
FIRST 句
説明　321
FOR BROWSE 句
サポートされない　616
FOR READ ONLY 句
無視　616
FOR UPDATE 句
サポートされない　616
FORWARD TO 文
ネイティブ文　772
FOR XML AUTO
使用　657
FOR XML AUTO の使用　657
FOR XML EXPLICIT
cdata ディレクティブの使用　673
element ディレクティブの使用　669
hide ディレクティブの使用　670
xml ディレクティブの使用　672
構文　661
using　661
FOR XML EXPLICIT の使用　661
FOR XML RAW
使用　654
FOR XML RAW の使用　654
FOR XML 句
AUTO モードの使用　657
BINARY データ型　651
EXPLICIT モードの構文　661
EXPLICIT モードの使用　661
IMAGE データ型　651
LONG BINARY データ型　651
RAW モードの使用　654
VARBINARY データ型　651
クエリ結果を XML として取得　650
使用方法　651
制限　651
FOR 句
FOR XML AUTO の使用　657
FOR XML EXPLICIT の使用　661
FOR XML RAW の使用　654
クエリ結果を XML として取得　650
FOR 文
構文　841
FoxPro
リモート・データ・アクセス　806
FROM 句
FROM 句内のストアド・プロシージャ　288
FROM 句内の抽出テーブル　288
概要　288
ジョインの説明　338
FullCompare

アクセス・プラン内の統計　567

G

go
バッチ文デリミタ　839
GRANT 文
Transact-SQL　599
同時性　194
GROUP BY ALL 句
サポートされない　616
group by list
アクセス・プラン内の項目　571
GROUP BY 句
Adaptive Server Enterprise との互換性　332
order by　322
SQL/92 標準　332
SQL 規格準拠　332
WHERE 句　317
実行　314
集合関数　312
説明　312
GROUP BY 句の概要　314
GROUP BY のあるクエリを実行する方法　314
GROUPING SETS
説明　435
連結　437
GROUPING SETS の使用　435
GROUPING 関数
NULL　424
ROLLUP 演算　424
GROUP 化されたビューまたは UNION ビューへの述部のプッシュダウン　553
GUID
グローバル・オートインクリメントとの比較　111
生成　193
デフォルトのカラム値　111
-gx オプション
スレッド　785

H

HAVING 句
GROUP BY　318
サブクエリ　450
集合関数を持つ場合と持たない場合　318
データ・グループの選択　318
パフォーマンス　547
論理演算子　319
HAVING 句でのサブクエリ　450
hide ディレクティブ
使用　670
HOLDLOCK キーワード
Transact-SQL　616

I

I/O
ビットマップのスキャン　533
IBM DB2
ASA への移行　733
DB2 へのリモート・データ・アクセス　797
IDENTITY カラム
値の検索　612
特殊な IDENTITY　611
IF 文
構文　841
IndAdd
アクセス・プラン内の統計　567
IndLookup
アクセス・プラン内の統計　567
IN list
アクセス・プラン内の項目　572
INOUT パラメータ
定義　846
INPUT 文
使用　696
説明　695
INSERT 文
SELECT　484
説明　484
ロック　182

instest 257
Interactive SQL
.sql ファイルのデフォルト・エディタ 741
クエリ結果のエクスポート 711
コマンド・デリミタ 874
コマンドのロード 743
コマンド・ファイル 741
スクリプトの実行 742
データベースの再構築 727
バッチ処理 742
バッチ・モード 742
リレーショナル・データを XML としてエクスポート 639
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 639
Interactive SQL でのプロシージャ・プロファイリング情報の表示 268
Interactive SQL の [インポート] ウィザード
説明 693
Interactive SQL を使用した特定のプロシージャに関するプロファイリング情報の表示 270
INTERSECT 演算
NULL 328
クエリの結合 324
使用 325
ルール 326
INTO 句
使用 850
Invocations
アクセス・プラン内の統計 567
IN キーワード
マッチング・リスト 294
IN 条件
サブクエリ 459
IN 条件によるセット・メンバシップのテスト 459
IN パラメータ
定義 846
IN リスト
アルゴリズム 514
最適化 556
ISNULL 関数
説明 301
IS NULL キーワード 301
ISO SQL/92 標準
典型的な矛盾 140
同時性 140

J

Java
Java クラスのデバッグ 899
デバッグに関する説明 890
Java デバッガ
Java クラスのデバッグ 899
稼働条件 891
起動 893
ストアド・プロシージャのデバッグ 894
ソース・コードの表示 900
チュートリアル 893
Java デバッガの稼働条件 891
jConnect メタデータ・サポート
インストール 49
JDBC クラスの設定上の注意 789
JDBC ベースのサーバ・クラス 789

K

key type
アクセス・プラン内の項目 569

L

LEAVE 文
構文 841
LIKE 演算子
ワイルドカード 296
LIKE 最適化 557
LIKE 条件
LIKE 最適化 557
LOAD DATABASE 文

サポートされない　595
LOAD TABLE 文
使用　700
説明　697
LOAD TRANSACTION 文
サポートされない　595
locked tables
アクセス・プラン内の項目　569
LONG VARCHAR データ型
XML の格納　637
LOOP 文
構文　841
プロシージャ　856
Lotus Notes
パスワード　807
リモート・データ・アクセス　806

M

master データベース
サポートされない　594
MAX 関数
リライト最適化　555
MESSAGE 文
プロシージャ　861
Microsoft Access
ASA への移行　733
リモート・データ・アクセス　805
Microsoft Excel
リモート・データ・アクセス　804
Microsoft FoxPro
リモート・データ・アクセス　806
Microsoft SQL Server
ASA への移行　733
リモート・アクセス　801
minimum 関数または maximum 関数の最適化　555
MIN 関数
リライト最適化　555
msodbc サーバ・クラス　801

N

NetWare
外部関数　882
外部プロシージャの呼び出し　881
NEWID
使用する場合　193
NEWID 関数
デフォルトのカラム値　111
NLM
外部関数　882
外部プロシージャの呼び出し　881
プロシージャからの呼び出し　879
NOHOLDLOCK キーワード
無視　617
NOT
論理演算子の使用　302
Notes とリモート・アクセス　806
NOT キーワード
探索条件　292
NULL
CUBE 演算　424
DISTINCT 句　287
EXCEPT 演算　328
INTERSECT 演算　328
ROLLUP 演算　424
Transact-SQL との互換性　614
UNION 演算　328
カラム定義　302
カラムのデフォルト　605
集合演算子　328
集合関数　311
出力　716
説明　299
ソート順　321
デフォルト　112
デフォルト・パラメータ　300
比較　300
プロパティ　301
NULL 値
Transact-SQL 外部ジョイン　359
カラムでの許可　32

データのインポート　702
変換エラーの無視　700
例　424
NULL 値と小計ロー　424
NULL 入力テーブル
外部ジョイン　352
NULL のためのカラムのテスト　300
NULL のプロパティ　301
NULL への値の代入　301

O

ODBC
アプリケーション　145
アプリケーションとロック　145
外部サーバ　793
odbcfet　257
ODBC サーバ・クラス　793, 804
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　145
OLAP
CUBE 演算　432
GROUPING SETS 演算　435
NULL 値　424
ROLLUP 演算　426
機能　420
小計ロー　421, 430
説明　419
OLAP 演算
説明　421
OLAP 演算について　421
OLAP 関数
RANK 関数　438
Window 関数　444
説明　438
レポート関数　442
OLAP 関数の活用　438
OLAP の概要　420
OLAP の仕組み　421
OLAP の使用　419
ON EXCEPTION RESUME 句
Transact-SQL　628
ストアド・プロシージャ　860
説明　862
例外処理を伴わない　868
ON 句
ジョイン　345
OPENXML 関数
使用　641
OPENXML を使用した XML のインポート　641
OPEN 文
プロシージャ　856
OPTIMIZATION_WORKLOAD オプション
クラスタード・ハッシュ GROUP BY　524
optimization goal
アクセス・プラン　568
OR
論理演算子の使用　302
Oracle
ASA への移行　733
Oracle とリモート・アクセス　800
oraodbc サーバ・クラス　800
order-by
アクセス・プラン内の項目　572
ORDER BY 句
GROUP BY　322
クエリ結果のソート　320
結果の制限　321
パフォーマンス　242
ORDER BY と GROUP BY　322
OUTPUT 文　711
説明　706
データを XML としてエクスポートするために使用　639
OUT パラメータ
定義　846

P

page map
アクセス・プラン内の項目　569
PerformanceFetch　257
PerformanceInsert　257

PerformanceTraceTime 257
PerformanceTransaction 257
PowerBuilder
リモート・データ・アクセス 747
PowerBuilder DataWindows からのリモート・データへのアクセス 747
predicates
アクセス・プラン内の項目 570
PREPARE 文
リモート・データ・アクセス 776
PROPERTY 関数
説明 246

Q

QUOTED_IDENTIFIER オプション
Transact-SQL 互換性の設定 605
説明 297

R

RAISERROR 文
ON EXCEPTION RESUME 628
Transact-SQL 627
RANK 関数
説明 438
例 439, 440, 441
RAW モード
使用 654
READ 文
コマンド・ファイルの実行 741
reload.sql
データベースの再構築 719
データベースの再ロード 730
テーブル・データのエクスポート 729
テーブルのエクスポート 717
レプリケートするデータベースの再構築 724
RESIGNAL 文
説明 867
RESTRICT アクション
説明 128
RETURN 文
説明 849
RETURN 文を使って値を返す 849
REVOKE 文
Transact-SQL 599
同時性 194
ROLLBACK 文
トリガ 620
複合文 843
プロシージャとトリガ 873
ROLLUP 演算
GROUP BY 句の概要 315
NULL 424
小計ロー 430
説明 426
例 428
ROLLUP の使用 426
row limit count
アクセス・プラン内の項目 572
RowsReturned
アクセス・プラン内の統計 567
RunTime
アクセス・プラン内の統計 567

S

SA_DEBUG グループ
デバッガ 891
sa_migrate_create_fks システム・プロシージャ
使用 737
sa_migrate_create_remote_fks_list システム・プロシージャ
使用 737
sa_migrate_create_remote_table_list システム・プロシージャ
使用 737
sa_migrate_create_tables システム・プロシージャ
使用 737
sa_migrate_data システム・プロシージャ
使用 737

sa_migrate_drop_proxy_tables システム・プロシージャ
使用 737
sa_migrate システム・プロシージャ
使用 736
sa_pause_workload_capture プロシージャ
インデックス・コンサルタントの一時停止 90
sa_stop_index_consultant プロシージャ
インデックス・コンサルタントの終了 90
sa_stop_workload_capture プロシージャ
インデックス・コンサルタントの終了 90
selectivity
アクセス・プラン内の項目 569
select list
アクセス・プラン 568
SELECT 文
INSERT 484
INTO 句 850
Transact-SQL 615
エイリアス 282
カーソル 856
カラムの順序 281
カラムの見出し 282
キーとクエリ・アクセス 240
説明 276
表示文字列 283
変数 617
文字データ 297
ローの指定 290
select リスト
EXCEPT 文 324
INTERSECT 文 324
UNION 演算 325
UNION 文 324
エイリアス 282
計算カラム 284
説明 279
SELECT リストの値の計算 284
SELECT を使用した新しいローの追加 486
SELF_RECURSION オプション
Adaptive Server Enterprise 620
SET DEFAULT アクション
説明 128
SET NULL アクション
説明 128
SET OPTION 文
Transact-SQL 605
SET 句
UPDATE 文 490
SIGNAL 文
Transact-SQL 627
プロシージャ 861
sp_addgroup システム・プロシージャ
Transact-SQL 599
sp_addlogin システム・プロシージャ
Transact-SQL 599
サポート 594
sp_adduser システム・プロシージャ
Transact-SQL 599
sp_bindefault プロシージャ
Transact-SQL 596
sp_bindrule プロシージャ
Transact-SQL 596
sp_changegroup システム・プロシージャ
Transact-SQL 599
sp_dboption システム・プロシージャ
Transact-SQL 605
sp_dropgroup システム・プロシージャ
Transact-SQL 599
sp_droplogin システム・プロシージャ
Transact-SQL 599
sp_dropuser システム・プロシージャ
Transact-SQL 599
SQL/XML
説明 649
SQL
他の SQL ダイアレクトとの違い 630
入力 277
SQL_TXN_ISOLATION 145
SQL_TXN_READ_COMMITTED
独立性レベル 145
SQL_TXN_READ_UNCOMMITTED
独立性レベル 145
SQL_TXN_REPEATABLE_READ

独立性レベル　145
SQL_TXT_SERIALIZABLE
独立性レベル　145
SQL Anywhere Studio
マニュアル　x
SQLCA.lock
独立性レベル　141
独立性レベルの選択　145
SQLCODE 変数
概要　860
SQL Remote
リモート・データ・アクセス　783
レプリケーションと同時トランザクション　195
SQL Server とリモート・アクセス　801
SQLSetConnectOption　145
SQLSTATE 変数
概要　860
SQL 規格　3
ANSI 以外のジョイン　344
GROUP BY 句　332
準拠　629
SQL クエリ　277
SQL コマンド・ファイル
Interactive SQL で開く　741
作成　741
実行　741, 742
出力の書き込み　743
説明　741
SQL 文
Sybase Central でのロギング　47
互換性のある SQL 文の記述方法　613
SQL 文のロギング　47
SQL を使用したプロキシ・テーブルの作成　765
Sybase Central
jConnect メタデータ・サポートのインストール　49
SQL 文のロギング　47
インデックスの検証　84
インデックスの作成　82
外部キーの管理　60
カラム制約　116
カラム・デフォルト　108
システム・オブジェクトの表示　47
システム・テーブルの表示　67
接続しないでデータベースを起動　48
データベース・オブジェクトのコピー　98
データベース・オプションの設定　45
データベースの再構築　727
データベースの削除　42
データベースの作成　39
データベースの消去　42
テーブルのコピー　67
テーブルの削除　56
テーブルの作成　52
テーブルの変更　54
統合データベースの設定　46
ビューの削除　76
ビューの修正　75
プライマリ・キーの管理　58
プロシージャの変換　622
Sybase Central からのデータベース統計値のモニタリング　248
Sybase Central でカラム・デフォルトを設定する　108
Sybase Central でテーブル制約およびカラム制約を編集する　116
Sybase Central におけるデータベース・オブジェクトのコピー　98
Sybase Central パフォーマンス・モニタ　248
Sybase Central パフォーマンス・モニタの設定　250
Sybase Central パフォーマンス・モニタを開く　248
Sybase Central を使用したプロキシ・テーブルの作成　764
Sybase Central を使用するストアド・プロシージャの変換　622
SYSCOLSTAT
カラム統計の更新　501
SYSCOLUMNS ビュー
重複する名前　604
SYSINDEXES ビュー

インデックス情報　96
重複する名前　604
SYSINDEX テーブル
インデックス情報　96
SYSIXCOL テーブル
インデックス情報　96
SYSSERVERS システム・テーブル
リモート・サーバ　751
SYSTABLE システム・テーブル
ビュー情報　78
SYSVIEWS ビュー
ビュー情報　78

T

TEMP 環境変数
テンポラリ・ファイルのロケーション　216
TIMESTAMP データ型
Transact-SQL　609
TMPDIR 環境変数
テンポラリ・ファイルのロケーション　216
TMP 環境変数
テンポラリ・ファイルのロケーション　216
TOP 句
説明　321
TRACEBACK 関数　861
Transact-SQL
IDENTITY カラム　611
NULL　614
timestamp カラム　609
移植可能な SQL の記述　613
外部ジョイン　357
外部ジョインの制限事項　358
概要　590
結果セット　624
後続ブランク　603
互換性のある SQL 文の記述方法　613
ジョイン　617
説明　590
データベースの作成　602
トリガ　620
バッチ　621
プロシージャ　619
変数　625
Transact-SQL 外部ジョインに対する NULL の影響　359
Transact-SQL 外部ジョインを使ったビューの使用　359
Transact-SQL 互換性
データベース・オプションの設定　605
Transact-SQL と互換性のあるデータベースの作成　602
Transact-SQL との互換性　589
データベース　606
Transact-SQL との互換性を意識したデータベースの設定　602
Transact-SQL との互換性を維持するためのオプション設定　605
Transact-SQL のストアド・プロシージャの概要　619
Transact-SQL の特殊な timestamp カラムとデータ型　609
Transact-SQL のトリガの概要　620
Transact-SQL のバッチの概要　621
Transact-SQL のプロシージャ言語の概要　619
Transact-SQL プロシージャから返される結果セット　624
Transact-SQL プロシージャでのエラー処理　626
trantest　257
TRUNCATE TABLE 文
説明　493
tsequal 関数　610

U

UNION
クエリ実行アルゴリズム　526
UNION ALL
クエリ実行アルゴリズム　527
UNION 演算
NULL　328

クエリの結合　324
ルール　326
UNKNOWN
NULL　301
UNLOAD TABLE 文
説明　707
UNLOAD 文
説明　708
UPDATE 文
使用　489
ロック　184
UUID
グローバル・オートインクリメントとの比較　111
生成　193
デフォルトのカラム値　111

V

ValuePtr パラメータ
説明　145

W

Watcom-SQL
互換性のある SQL 文の記述方法　613
説明　590
WHERE 句
GROUP BY 句　317
HAVING との比較　318
NULL 値　300
UPDATE 文　490
サブクエリ　449
ジョイン　347
説明　290
パフォーマンス　242, 547
文字列比較　295
WHERE 句でのサブクエリの使用　449
WHERE 句での範囲 (between と not between) の使用　293
WHERE 句での比較演算子の使用　291
WHERE 句での文字列のマッチング　295
WHERE 句でのリストの使用　294
WHERE 句のサブクエリのジョインへの変換　470
WHILE 文
構文　841
Windows CE
データベースの作成　39
Windows パフォーマンス モニタ
説明　251
Windows パフォーマンス モニタからのデータベース統計値のモニタリング　251
Window 関数
クエリ実行アルゴリズム　530
説明　444
例　445
WITH CHECK OPTION 句
説明　72
WITH CHECK OPTION 句の使い方　72
WITH EXPRESS CHECK
パフォーマンス　218
WITH 句　391
オプティマイザのアルゴリズム　515

X

XML
Adaptive Server Anywhere データベースでの使用　635
DataSet オブジェクトを使用したインポート　648
DataSet オブジェクトを使用してデータを XML としてエクスポート　640
FOR XML AUTO の使用　657
FOR XML EXPLICIT の使用　661
FOR XML RAW の使用　654
Interactive SQL からデータを XML としてエクスポート　639
OPENXML を使用したインポート　641
クエリ結果を XML として取得　650
定義　636

リレーショナル・データからクエリ結果を XML として取得 649
リレーショナル・データとしてインポート 641
リレーショナル・データベースにおける格納 637
リレーショナル・データを XML としてエクスポート 639
XMLAGG 関数
使用 675
XMLCONCAT 関数
使用 676
XMLELEMENT 関数
使用 677
XMLFOREST 関数
使用 679
XMLGEN 関数
使用 680
xml ディレクティブ
using 672
XML データ型
使用 637
XML と Adaptive Server Anywhere 636
XML 文書をリレーショナル・データとしてインポートする 641

あ

アーキテクチャ
Adaptive Server 594
アイコン
マニュアルで使用 xv
アクション
CASCADE 128
RESTRICT 128
SET DEFAULT 128
SET NULL 128
アクセス・プラン
Interactive SQL 584
SQL 関数 584
印刷 575
解釈 564
カスタマイズ 575
キャッシュ 507
グラフィカルなプラン 575
コンテキスト別のヘルプ 576
省略形 564
説明 496
統計の説明 567
長いテキスト・プラン 574
短いテキスト・プラン 573
アクセス・プランに使用される省略形 564
アクセス・プランの解釈 564
主な統計 579
アクセス・プランのキャッシュ 507
アスタリスク (*)
SELECT 文 279
値の格納
共通テーブル式 399
圧縮
パフォーマンス 218
圧縮 B ツリー・インデックス
説明 544
アップグレード
データベース・ファイル・フォーマット 720
アトミックなトランザクション 134
アトミックな複合文 843
アポストロフィ
文字列 297
アルゴリズム
インデックス DISTINCT 522
クエリ実行 510
暗号化
オブジェクトを隠す 887
キャッシュ・サイズ 212
アンロード・ツール
説明 710
[データのアンロード] ダイアログ 711
[データベース・アンロード] ウィザード 710
アンロードと再ロード
データベース 722, 724, 730

い

以下
　比較演算子　291
意思決定支援
　独立性レベル　154
以上
　比較演算子　291
移植可能な SQL　613
移植可能な SQL を記述するための一般的な
　　ガイドライン　613
一意性
　インデックスを使用した強化　542
一意性制約
　生成されたインデックス　535
一貫性
　2 フェーズ・ロック　187
　ISO SQL/92 標準　140
　繰り返し可能読み出し　140, 160, 180
　繰り返し不可能読み出しの例　162
　実際のロックの意味　169
　正当性とスケジューリング　152
　説明　134
　ダーティ・リード　140, 180
　ダーティ・リードのチュートリアル　157
　直列不可能なスケジュールの影響　153
　典型的なトランザクション　154
　独立性レベル　139, 141, 166, 169, 188
　トランザクション中　140
　幻ロー　141, 166, 180, 188
　ロックを使用して保証　175
イベント
　個々のプロファイリング情報の表示　267
　プロファイリング・データ一覧の表示　265
　プロファイリング・データの表示　259
イメージ
　挿入　488
インストール
　jConnect メタデータ・サポート　49
インデックス
　B ツリー　542
　HAVING 句のパフォーマンス　547
　Transact-SQL　608
　WHERE 句のパフォーマンス　547
　圧縮 B ツリー　544
　インデックス間の相関関係　95
　インデックス・コンサルタント　79, 86, 89
　インデックス・コンサルタントの知識　90
　インデックスのタイプ　542
　インデックス・ヒント　81
　仮想　88
　カラムの順序の効果　539
　クラスタード　81
　計算カラム　83
　検査　96
　検索引数可能な述部　547
　検証　84
　構造　537
　候補　88
　コスト　86
　最適化　535
　削除　85
　作成　82
　支援　205
　述部の分析　547
　使用する場合　79
　処理　79
　生成　535
　説明　535
　断片化　256
　通常、述部を満たす　504
　テンポラリ・テーブル　536
　同時性の改善　190
　ハッシュ B ツリー　543
　ハッシュ値　543
　パフォーマンス　213
　パフォーマンスの改善　537
　頻繁に検索するカラム　80
　ファンアウトとページ・サイズ　545
　複合　538, 544
　ページ・サイズの推奨値　545
　未使用　95
　リーフ・ページ　537
　利益　86
　利点とロック　155

インデックス DISTINCT
クエリ実行アルゴリズム 522
インデックス GROUP BY
クエリ実行アルゴリズム 525
インデックス・コンサルタント
概要 79, 86, 205
起動 89
サーバの状態 96
終了 90
推奨内容 93
推奨内容の実装 96
推奨内容の評価 95
接続の状態 96
説明 86
知識 90
負荷 87, 91
[インデックス作成] ウィザード
使用 82
インデックス・スキャン
説明 510
並列インデックス・スキャン 513
並列インデックス・スキャン・アルゴリズム 513
インデックスの選択性
説明 537
インデックスのその他の使用法 542
インデックスのタイプ 542
インデックスの断片化 256
インデックスのパフォーマンス向上 537
インデックスの編集 79
インデックスの有効な使用 213
インデックスはいつ使うか 79
インデックス・ヒント
使用 81
インポート
ASE 互換性 744
DataSet オブジェクトを使用して XML 文書をリレーショナル・データとしてインポートする 648
NULL 値 702
OPENXML を使用して XML 文書をリレーショナル・データとしてインポートする 641
XML 文書をリレーショナル・データとしてインポートする 641
一致しないテーブル構造 702
概要 691
ツール 693
テンポラリ・テーブルの使用 97
[インポート] ウィザード
使用 693
説明 693
インポート中の変換エラー 700
インポート・ツール
INPUT 文 695
INSERT 文 698
Interactive SQL の [インポート] ウィザード 693
LOAD TABLE 文 697
説明 693
プロキシ・テーブル 699
インポート用のテーブル構造 702
引用符
Adaptive Server Enterprise 297
文字列 297

う

ウィザード
[インデックス作成] 82
[インポート] 693
外部キー作成 60
データベース・アップグレード 49
[データベース移行] 733
[データベース作成] 39
[データベース消去] 42
ドメイン作成 119
[トリガ作成] 832
[ビュー作成] 70
プロキシ・テーブル作成 764
[プロシージャ作成] 815
リモート・サーバ作成 753
[リモート・プロシージャ作成] 773, 816

え

影響
直列不可能なトランザクション・スケジューリング 153
トランザクション・スケジューリング 153
エイリアス
説明 282
相関名 288
エクスプロード
クエリ実行アルゴリズム 529
エクスポート
ASE 互換性 744
NULL 値 716
概要 704
クエリ結果 711
スキーマ 729
テーブル 717
リレーショナル・データを XML としてエクスポートする 640
エクスポート・ツール 705
dbunload ユーティリティ 709
Interactive SQL ウィザード 705
OUTPUT 文 706
UNLOAD TABLE 文 707
UNLOAD 文 708
エラー
Transact-SQL 626, 628
プロシージャとトリガ 860
変換 700
エラー処理
ON EXCEPTION RESUME 862
プロシージャとトリガ 860
演算子
NOT キーワード 292
算術 285
条件の接続 302
ハッシュ・テーブル・スキャン 512
優先度 285
エンティティ
整合性の確保 124
説明 5
選択 13
属性 5
定義 5
エンティティ整合性
概要 104
クライアント・アプリケーションによる違反 124
プライマリ・キー 630
エンティティと関係の決定 13

お

オートインクリメント
IDENTITY カラム 611
デフォルト 110
オートコミット
トランザクション 135
パフォーマンス 217
オートコミット・モードをオフにする 217
オートメーション
ユニーク・キーの生成 192
オーファンと参照整合性 183
大文字と小文字の区別
ASE 互換データベースの作成 603
Transact-SQL との互換性 606
識別子 277, 607
ソート順 321
データ 607
データベース 606
ドメイン 606, 607
パスワード 607
ユーザ ID 607
リモート・アクセス 784
大文字と小文字の不要な変換の排除 561
オブジェクト
隠す 887
ロック可能なオブジェクト 175
オブジェクト名の互換性 608
オプション
BLOCKING 148
DEFAULTS 703
ISOLATION_LEVEL 143
データベース・オプションの設定 45

オプションの外部キー　126
オプションの関係　9
オプティマイザ
　仮定条件　502
　述部の分析　547
　省略　509
　説明　496, 498
　セマンティック・サブクエリ変形　546
　セマンティック変形　550
　役割　496
オプティマイザの仕組み　498
オプティマイザの推定
　説明　498
オプティマイザの役割　496
オンライン分析処理
　CUBE 演算　432
　GROUPING SETS　435
　NULL 値　424
　ROLLUP 演算　426
　関数　438
　機能　420
　仕組み　421
　小計ロー　421
　説明　419

か

カーソル
　LOOP 文　856
　SELECT 文　856
　安定性　142
　ジョインでの更新　343
　不安定性　142
　プロシージャ　856
　プロシージャとトリガ　856
カーソル安定性　142
カーソル管理
　概要　856
カーソル不安定性　142
カーディナリティ
　関係　7
改行
　SQL　277
解析
　関係　24
階層データ構造　402, 405
概念データベース・モデル
　定義　5
概念データ・モデリング
　説明　3
外部関数
　取り消し　885
　パラメータを渡す　885
　プロトタイプ　881
　リターン・タイプ　886
外部キー
　Sybase Central で表示　60
　管理　60
　キー・ジョイン　374
　削除　60
　作成　60, 62
　参照整合性　126
　修正　62
　整合性　630
　生成されたインデックス　535
　パフォーマンス　240
　必須／オプション　126
　役割名　375
[外部キー作成] ウィザード
　使用　60
外部キーの管理　60
外部キーを使ったクエリのパフォーマンス改善　241
外部サーバ
　ODBC　793
外部参照
　HAVING 句　450
　集合関数　307
　説明　463
外部ジョイン
　Transact-SQL　357, 617
　Transact-SQL の制限　358
　Transact-SQL ビュー　359

ジョイン条件　353
スター・ジョインの例　365
制限　358
説明　352
ビューと抽出テーブル　355
複雑　354
外部ジョインから内部ジョインへの変換　558
外部呼び出し
プロシージャと関数の作成　879
外部ロード　689
外部ログイン
削除　760
作成　759
説明　759
[外部ログイン作成] ウィザード
使用　759
外部ログインの削除　760
外部ログインの作成　759
外部ログインの使用　759
書き込みロック　177
各種独立性レベルでの典型的なトランザクション　154
仮想インデックス
インデックス・コンサルタント　88
インデックス・コンサルタントの分析手順　92
説明　88
仮想メモリ
貴重なリソース　504
カタログ
ASE 互換性　596
カッコ
UNION 演算子　324
算術文内　285
カラム
GROUP BY 句　312
IDENTITY　611
NULL 値の許可　32
SELECT 文　281
select リスト　281
timestamp　609
検査制約　117
制約　32
データ型　31
データ型とドメインの割り当て　119
デフォルト　107
プロパティ　31
変更　54, 55
命名　31
カラム一意性の強化　542
カラム属性
AUTOINCREMENT　193
NEWID　193
デフォルト値の生成　193
カラム・デフォルトの修正と削除　108
カラム・デフォルトの使い方　107
カラム統計　498
ヒストグラムを参照
更新　501
カラム統計の更新　501
カラムに対する検査制約の使い方　114
カラムのデータ型の選択　31
カラム名の選択　31
関係
1 対 1　7
1 対多　7
エンティティに変更　10
カーディナリティ　7
解析　24
再帰　9
説明　6
選択　13
多対多　8
定義　5
必須／オプション　9
役割　8
環状ブロックの競合　149
関数
TRACEBACK　861
tsequal　610
外部　879
キャッシュ　562
決定性　562

個々のプロファイリング情報の表示　266
プロファイリング・データ一覧の表示　264
プロファイリング・データの表示　259
べき等　562
ユーザ定義　825
完全比較
説明　537
カンマ
スター・ジョイン　362
テーブル式のジョイン時　378
テーブル式リスト　349
管理者の役割
ASE　597

き

キー
パフォーマンス　240
割り当て　20
キー・ジョイン
2 つ以上の外部キー　375
ON 句　346, 375
カンマを含まないテーブル式　379
カンマを含まないテーブル式とリスト　383
規則　388
説明　374
テーブル式　378
テーブル式リスト　381
ビューと抽出テーブル　385
キー・ジョイン操作規則　388
キーワード
HOLDLOCK　616
NOHOLDLOCK　617
リモート・サーバ　783
キーを使ったクエリのパフォーマンス改善　240
記号
文字列比較　295
規則
表記　xiii
起動
インデックス・コンサルタント　89
基本となる仮定条件
オプティマイザ　502
キャッシュ
UNIX　234
UNIX での Java アプリケーション　236
Windows 95/98　234
Windows NT　234
アクセス・プラン　507
暗号化データベースには大きいキャッシュが必要　212
サイズのモニタリング　236
サブクエリ　562
準備　237
初期サイズ、最小サイズ、最大サイズ　232
説明　231
動的サイズ決定　231
パフォーマンス　212
ユーザ定義関数　562
読み込みヒット率　581
キャッシュ・ウォーミング
説明　237
キャッシュが使用するメモリの制限　232
キャッシュ・サイズ
UNIX　234
UNIX での Java アプリケーション　236
Windows 95/98　234
Windows NT　234
初期サイズ、最小サイズ、最大サイズ　232
ページ・サイズ　533
モニタリング　236
キャッシュ・サイズの拡大　212
キャッシュ・サイズのモニタリング　236
競合
環状ブロッキング　149
トランザクション・ブロック　148, 165
ロック　148
ロック間　179
共通テーブル式
一般的な使用例　397
階層データ構造の探索　402
構成部品の問題　405
再帰でのデータ型　409
再帰に関する制限　403

最短距離の問題　411
使用できる条件　395
説明　391
定数セットの格納　399
複数の集合レベル　397
共有オブジェクト
プロシージャからの呼び出し　879
共有ロックと排他ロック　178

く

句
COMPUTE　616
FOR BROWSE　616
FOR READ ONLY　616
FOR UPDATE　616
GROUP BY ALL　616
INTO　850
ON EXCEPTION RESUME　628, 862, 868
説明　276
クエリ
Transact-SQL　615
アクセス・プラン　564
エクスポート　711
オプティマイザの省略　509
共通テーブル式　391
最適化　496, 498
集合操作　324
説明　276
テーブルからのデータの選択　275
プレフィクス　421
クエリ結果
エクスポート　711
クエリ結果にあるカラム名の変更　282
クエリ結果のグループへの編成　312
クエリ結果のソート　242
クエリ結果の文字列　283
クエリ結果の要約、グループ化、ソート　305
クエリ結果を XML として取得する　649
クエリ最適化
オプティマイザのバイパス　509
クエリ実行アルゴリズム　510
IN リスト　514
UNION ALL　527
Window 関数　530
アクセス・プランに使用される省略形　564
インデックス DISTINCT　522
インデックス GROUP BY　525
インデックス・スキャン　510
エクスプロード　529
クラスタード・ハッシュ GROUP BY　524
グループ化アルゴリズム　523
順序付き DISTINCT　522
順序付き GROUP BY　525
順序付き GROUP BY SETS　525
ジョイン　515
セミジョイン　515, 516, 518
ソート GROUP BY SETS　526
ソートと UNION　526
ソート・ブロック・ジョイン　516
単一の GROUP BY　526
単一の XML GROUP BY　526
逐次テーブル・スキャン　511
抽出テーブル　514
重複排除　521
ネスト・ブロック・ジョイン　516
ネスト・ループ・ジョイン　515
ネスト・ループ・セミジョイン　516
ハッシュ DISTINCT　521
ハッシュ GROUP BY　523
ハッシュ GROUP BY SETS　524
ハッシュ NOT EXIST　519
ハッシュ・ジョイン　517, 518, 519
ハッシュ・セミジョイン　518
ハッシュ・テーブル・スキャン　512
ハッシュ非セミジョイン　519
非セミジョイン　519
フィルタと事前フィルタ　528
ブルーム・フィルタ　529
ブロック・ネスト・ループ・ジョイン　516
並列インデックス・スキャン　513
マージ・ジョイン　520
マージ・ソート　526

ローの制限　529
ロック　529
クエリ時のロック　180
クエリ上でブロックされるクエリ
リモート・データ・アクセス　785
クエリ処理中のワーク・テーブルの使用　243
クエリに関する一般的な問題
リモート・データ・アクセス　785
クエリの解析
リモート・データ・アクセス　778
クエリの最初のローの検索　321
クエリの最適化
アクセス・プランの解釈　564
仮定条件　502
サブクエリを EXISTS 述部として書き換え　505
説明　496
手順　508
クエリの最適化と実行　495
クエリの正規化
リモート・データ・アクセス　778
クエリのパフォーマンス
アクセス・プランの解釈　564
クエリのパフォーマンスのモニタリング　257
クエリの前処理
リモート・データ・アクセス　778
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　124
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　127
クライアント・アプリケーションからのデータベース統計値の取得　246
クライアント側ロード　689
クライアントとサーバとの間の要求数の削減　219
クラス
リモート・サーバ　787
クラスタード・インデックス
インデックス・コンサルタントの推奨内容　93, 96
使用　81
クラスタード・ハッシュ GROUP BY
クエリ実行アルゴリズム　524
グラフィカルなプラン
Interactive SQL　584
SQL 関数　584
印刷　575
カスタマイズ　575
コンテキスト別のヘルプ　576
述部　582
説明　575, 576
グラフィカルなプランのカスタマイズ　575
グラフィカル・プラン
単純なクエリに対してはすべてのセクションが存在するわけではない　576
グラフの種類
パフォーマンス・モニタの設定　250
グラフ表示
パフォーマンス・モニタの使用　248
繰り返し可能読み出し　140, 160, 180
繰り返し不可能読み出し
説明　140
チュートリアル　160
独立性レベル　141, 188
例　162
繰り返し不可能読み出しによる矛盾　140, 160, 180
グループ
ASE　599
グループ化アルゴリズム
クエリ実行アルゴリズム　523
グループ読み込み
テーブル　533
グローバル・オートインクリメント
GUID および UUID との比較　111
グローバル・テンポラリ・テーブル
説明　97
データのインポート　703
グローバル変数
デバッガ　910
クロス・ジョイン　349

け

警告
プロシージャとトリガ 865
計算カラム
INSERT 文 66
UPDATE 文 66
インデックス 83
検索引数可能な関数を使用したクエリ 548
再計算 66
作成 64
制限事項 66
トリガ 66
計算値 312
説明 306
計算ロー
ROLLUP 演算 430
結果セット
Transact-SQL 624
パーミッション 851
ファイルへの保存 743
複数 853
プロシージャ 823, 851
変数 854
ロー数の制限 321
結果セットの実体化
クエリ処理 243
結果をプロシージャのパラメータとして返す 849
決定性関数
定義 562
現在の日付／時刻デフォルト 109
検索引数可能な述部
説明 547
検査制約
カラム 114
削除 117
修正 117
選択 32
テーブル 115
ドメイン 116
ドメインでの使用 121
検査制約の修正と削除 117
検証
XML 673
インデックス 84
カラム制約 32
データベース設計 28
限定比較テスト 472
サブクエリ 455
原理
2 フェーズ・ロック 187

こ

降順
ORDER BY 句 320
更新時のロック 184
[更新] タブ
インデックス・コンサルタントの推奨内容 94
構成部品の問題 405
後続ブランク
Transact-SQL 603
データベースの作成 603
比較 291
構文に依存しない最適化 496
候補インデックス
インデックス・コンサルタント 88
説明 88
効率
インデックスを使用した改善 190
改善とロック 155
データのインポート時の時間の節約 692
互換性
ANSI 以外のジョイン 344
ASE 590
ASE でのインポートとエクスポート 744
GROUP BY 句 332
NULL の出力 716
Transact-SQL との互換性を維持するためのオプション設定 605
大文字と小文字の区別 606
サーバとデータベース 594
互換性のある SQL 文の記述方法 613

互換性のあるクエリの記述方法　615
互換性のあるデータ型のジョイン　343
互換性のあるテーブルの作成　614
コスト
　インデックス・コンサルタントの推奨内容　94
コストベースの最適化　496
コスト・モデル
　インデックス・コンサルタントが使用するコスト・モデル　92
　説明　498
コスト利益の合計
　インデックス・コンサルタントの推奨内容　94
異なるファイルの異なるデバイスへの配置　214
コピー
　INSERT を使用したデータ　487
　Sybase Central におけるデータベース・オブジェクト　98
　テーブル　67
　ビュー　72
　プロシージャ　820
コマンド
　Interactive SQL でのロード　743
コマンド・デリミタ
　設定　874
コマンド・デリミタを変更する必要があるかどうかをチェックする
　プロシージャ　874
コマンド・ファイル
　Interactive SQL で開く　741
　[SQL 文] ウィンドウ枠　741
　概要　741
　構築　741
　作成　741
　実行　741, 742
　出力の書き込み　743
　説明　741
コミット時の参照整合性の検査　183
混合負荷または OLAP 負荷の最適化　503

さ

サーバ
　接続しないでデータベースを起動　48
　パフォーマンス・モニタでのグラフ表示　248
サーバ側ロード　689
サーバ・クラス
　asajdbc　790
　asaodbc　794
　asejdbc　791
　aseodbc　795
　db2odbc　797
　msodbc　801
　ODBC　793, 804
　oraodbc　800
　説明　749
　定義　748
サーバ上のリモート・テーブルのリスト　756
サーバとデータベース
　互換性　594
サーバの機能
　リモート・データ・アクセス　779
サーバの状態
　インデックス・コンサルタント　96
再帰関係　9
再帰クエリ
　制限　403
再帰サブクエリ
　最短距離の問題　411
　説明　391
　データ型宣言　409
　複数の集合レベル　397
　分解部品の問題　405
再帰テーブル
　アルゴリズム　515
再構築
　ダウン時間の最短化　730
　ツール　727
　データベース　719
　目的　719
再構築中のダウン時間の最短化　730
再構築ツール　727

dbisql ユーティリティ 727
dbunload ユーティリティ 727
UNLOAD TABLE 文 728
最高のパフォーマンスを得るヒント 211
最小キャッシュ・サイズ 232
最小限の管理作業
オプティマイザ 502
最初のローの最適化または結果セット全体の最適化 503
最大
キャッシュ・サイズ 232
最短距離の問題 411
最適化
アクセス・プランの解釈 564
インデックスの使用 190
コストベース 496
最適化時のサブクエリ変形 546
最適化に影響する仮定条件 502
最適化の手順 508
削除
インデックス 85
カラム・デフォルト 108
検査制約 117
データ型 122
データベース・ファイル 41
テーブル 56
ドメイン 122
トリガ 836
ビュー 76
プロシージャ 820
ユーザ定義データ型 122
削除時のロック 185
作成
SQL のデータベース 40
Windows CE 用データベース 39
インデックス 82
外部呼び出しを使ったプロシージャと関数 879
カラム・デフォルト 107
コマンド・ラインからのデータベース 41
データ型 119, 120
テーブル 51
ドメイン 119, 120
トリガ 831
ビュー 70
プロシージャ 815
ユーザ定義関数 825
リモート・プロシージャ 815
サブクエリ
ALL テスト 457
ANY 演算子 456
ANY テスト 455
EXISTS 述部として書き換え 505
GROUP BY 450
HAVING 句 450
IN キーワード 294
WHERE 句 449, 470
演算子のタイプ 452
オプティマイザ内部 469
外部参照 450
キャッシュ 562
限定比較テスト 452, 455
ジョインとして書き換え 464
ジョインへの変換 464, 470
セット・メンバシップ・テスト 452, 459
説明 447, 448
相関 463, 469
存在テスト 452, 461
ネスト 467
比較演算子 471
比較テスト 452, 453
ロー・グループ選択 450
ロー選択 449
サブクエリ・セット・メンバシップ・テスト
説明 459
サブクエリ操作 469
サブクエリと関数のキャッシュ 562
サブクエリとジョイン 464
サブクエリの使用 447
サブクエリのテスト 452
サブクエリのネスト解除 551
サブクエリ比較テスト 453
サブクエリを EXISTS 述部として書き換え

505
サブスクリプション
データのレプリケーションと同時性 195
サブトランザクション
セーブポイント 137
プロシージャとトリガ 873
サポート
ニュースグループ xviii
算術
演算 306
式と演算子の優先度 285
参照整合性
アクション 127
オーファン 183
外部キー 126
クライアント・アプリケーションによる違反 127
検査 128
コミット時の検証 183
システム・テーブル内の情報 130
喪失 127
プライマリ・キー 630
参照整合性アクション
システム・トリガで実装 128
参照整合性検証の遅延 183
参照整合性の確保 125
参照整合性の喪失 127
サンプル・データベース
asademo.db のスキーマ 336

し

時間
プロシージャとトリガ 875
時間節約の方式
データのインポート 692
式
NULL 値 302
識別子
大文字と小文字の区別 607
修飾 278
ドメインでの使用 121
ユニーク 608
システム・オブジェクト 47
表示 67
システム・カタログ
ASE 596
システム関数
tsequal 610
システム管理者
ASE 597
システム・セキュリティ担当者
ASE 597
システム・テーブル
ASE 596
Transact-SQL の名前の重複 604
インデックス 96
参照整合性についての情報 130
所有者 597
ビュー 78
表示 67
システム・テーブル内のインデックス 96
システム・テーブル内のビュー 78
システム・テーブルの整合性ルール 130
システム・テーブルの表示 67
システム・トリガ
個々のプロファイリング情報の表示 268
参照整合性アクションの実装 128
プロファイリング・データ一覧の表示 265
プロファイリング・データの表示 259
システム・ビュー
インデックス 96
参照整合性についての情報 130
実行
SQL スクリプト 741, 742
コマンド・ファイル 741
実際のロックの重要性についてのチュートリアル 169
実装
ロック 188
指定カラムへの値の挿入 485
自動ジョイン
外部キー 631
集計情報

CUBE 演算　432
ROLLUP 演算　426
小計ロー　421
集合関数
Adaptive Server Enterprise との互換性　332
ALL キーワード　306
DISTINCT キーワード　306, 310
GROUP BY 句　312
NULL　311
ORDER BY と GROUP BY　322
外部参照　307
スカラ集合　307
説明　306
データ型　309
複数レベル　397
ベクトル集合　312
集合関数に伴う GROUP BY の使用　312
集合関数を使用したクエリ結果の要約　306
集合関数を使用できる場所　307
集合操作
NULL　328
説明　324
ルール　326
修飾
説明　290
修飾された名前
データベース・オブジェクト　278
修正
カラム・デフォルト　108
検査制約　117
ビュー　75
述部
アクセス・プランでの解釈　582
説明　547
パフォーマンス　547
述部の分析
説明　547
出力
NULL　716
エクスポート・ツール　705
クエリ結果のエクスポート　711
データのエクスポート　704
データベースのエクスポート　715
テーブルのエクスポート　717
出力リダイレクション　711
順序付き DISTINCT
クエリ実行アルゴリズム　522
順序付き GROUP BY
クエリ実行アルゴリズム　525
順序付き GROUP BY SETS
クエリ実行アルゴリズム　525
ジョイン
2 つのテーブル　341
3 つ以上のテーブル　342
ANSI 以外のジョイン　344
DELETE 文、UPDATE 文、INSERT 文　343
FROM 句　338
NULL 入力テーブル　352
ON 句　345
Transact-SQL　617
Transact-SQL 外部ジョインと NULL 値　359
Transact-SQL 外部ジョインとビュー　359
Transact-SQL 外部ジョインの制限　358
WHERE 句　347
カーソルの更新　343
外部　352
外部ジョインから内部ジョインへの変換　558
カンマ　349
キー　631
キー・ジョイン　374
クエリ実行アルゴリズム　515
クロス・ジョイン　349
異なるデータベースのテーブル　770
サブクエリからジョインへの変換　470
サブクエリの変換　464
自動　631
ジョインしたテーブル　340
ジョイン条件　339
ジョインの削除によるリライト最適化　554
スター・ジョイン　362
説明　338
セルフジョイン　360
探索条件　347
抽出テーブル　368

重複する相関名　362
直積　349
データ型変換　343
テーブル式　342
デフォルトは KEY JOIN　340
等価ジョイン　347
内部　351
内部ジョインの計算方法　341
内部と外部　351
ナチュラル　631
ナチュラル・ジョイン　369
ネスト　342
保護されたテーブル　352
リモート・テーブル　768
ジョイン・アルゴリズム
説明　515
ジョイン演算子
Transact-SQL　617
ジョイン条件
種類　347
ジョイン条件に対する WHERE 句の使用　347
ジョイン操作　337
ジョインで重複する相関名 (スター・ジョイン)　362
ジョインの互換性　617
ジョインの削除によるリライト最適化　554
ジョイン方式の区切りにコロンを使用　572
消去
検査制約　117
データベース　41
テーブル　56
消去ユーティリティ [dberase]
使用　42
小計の概要　421
小計ロー
GROUPING SETS　435
NULL 値　424
ROLLUP 演算　430
構成　430
説明　421
定義　426, 432
連結された GROUPING SETS　437
条件
論理演算子との接続　302
昇順
ORDER BY 句　320
初期化
データベース　37
初期化ユーティリティ [dbinit]
使用　41
初期キャッシュ・サイズ　232

す

推奨内容
インデックス・コンサルタント　93
推奨内容の実装
インデックス・コンサルタント　96
スカラ集合　307
スキーマ
エクスポート　729
スクリプト
Interactive SQL で実行　742
コマンド・ファイルの作成　741
コマンド・ファイルの説明　741
コマンド・ファイルのロード　743
スケジュール
2 フェーズ・ロック　187
直列可能　152
直列可能性の影響　153
直列可能とロックの早期解放　189
直列不可能なスケジュールの影響　153
スター・ジョイン　362
ストアド・プロシージャ
FROM 句内での使用　288
Transact-SQL ストアド・プロシージャの概要　619
共通テーブル式を含む　397
個々のプロファイリング情報の表示　266
デバッグ　894
プロファイリング・データ一覧の表示　264
プロファイリング・データの表示　259

ストアド・プロシージャ言語
概要 619
ストアド・プロシージャの自動変換 622
スレッド
使用できるスレッドがない場合のデッドロック 149
スワップ領域
データベース・キャッシュ 234

せ

正規化
説明 19
パフォーマンス向上 213
正規形
第 1 正規形 20
第 2 正規形 22
第 3 正規形 23
データベース設計の正規化 19
制御文
リスト 841
制限
リモート・データ・アクセス 783
整合性
確保 124
カラム・デフォルト 107
検査 128
システム・テーブル内の情報 130
整合性確保のためのトリガの使用 105
制約 105, 114
説明 102
喪失 127
ツール 103
整合性制約を実装するための SQL 文 105
生成
物理データ・モデル 24
ユニーク・キー 192
生成されたジョイン条件
説明 339
生成されたジョインと ON 句 346
制約 116
一意性制約 116
概要 104
カラムとテーブル 32
検査制約 116
制約の選択 32
セーブポイント
トランザクション内 137
ネストと命名 138
プロシージャとトリガ 873
セーブポイントのネストと命名 138
セーブポイントの命名とネスト 138
セキュリティ
オブジェクトを隠す 887
プロシージャ 813
[設計の詳細] ウィンドウ枠
説明 47
設計のプロセス 13
接続
接続しないでデータベースを起動 48
デバッガ 912
デバッグ 893
リモート 776
接続しないでデータベースを起動する 48
接続の削除
リモート・データ・アクセス 786
接続の切断
データベース 43
別のユーザとデータベースの接続 43
接続の問題点
リモート・データ・アクセス 784
設定
Sybase Central パフォーマンス・モニタ 250
セット・メンバシップ・テスト 476
=ANY 459
否定 460
セマンティック・クエリ変形 546
セマンティック変形
説明 550
セマンティック変形の種類 550
セミコロン
コマンド・デリミタ 874
セミジョイン

クエリ実行アルゴリズム　515, 516, 518
セルフジョイン　360
全外部ジョイン
説明　352
選択性
アクセス・プランでの解釈　579, 582
選択性推定
アクセス・プランでの解釈　582
部分インデックス・スキャンの使用　542
全比較
アクセス・プラン内の統計　567

そ

相関サブクエリ
外部参照　463
説明　462, 469
相関名
スター・ジョイン　362
説明　375
セルフジョイン　360
テーブル名　288
相対利益
インデックス・コンサルタントの推奨内容　94
挿入されたローに対するディスク割り付け　531
挿入時のロック　182
双方向レプリケーション　195
総利益
インデックス・コンサルタントの推奨内容　94
[ソース・コード] ウィンドウ
ブレークポイントの設定　906
ソート
インデックスの使用　242
クエリ実行アルゴリズム　526
ソート GROUP BY SETS
クエリ実行アルゴリズム　526
ソート順
ORDER BY 句　320
ソートと UNION　526
ソート・ブロック・ジョイン
クエリ実行アルゴリズム　516
属性
SQLCA.lock　145
クエリ結果を XML として取得　650
選択　16
定義　5
存在テスト
説明　461
否定　462

た

ダーティ・リード
クエリ中のロック　180
チュートリアル　157
独立性レベル　141
矛盾　140
待機
参照整合性の検証　183
ロックされたローへのアクセス　165
対挿入
ロック　177, 186
多対多の関係
解析　24, 27
定義　8
多対多の関係をエンティティに変更　10
他の SQL ダイアレクトとの違い　629
単一の GROUP BY
クエリ実行アルゴリズム　526
単一の XML GROUP BY
クエリ実行アルゴリズム　526
断片化
インデックス　256
説明　254
テーブル　255
ファイル　254
ファイル、テーブル、インデックス　224

ち

逐次スキャン
説明 511
ディスク割り付けとパフォーマンス 531
逐次テーブル・スキャン
説明 511
ディスク割り付けとパフォーマンス 531
抽出
SQLRemote のデータベース 732
抽出テーブル
FROM 句内での使用 288
アルゴリズム 514
外部ジョイン 355
キー・ジョイン 385
ジョイン 368
ナチュラル・ジョイン 372
中断
メニューの実行 906
チュートリアル
繰り返し不可能読み出し 160
ダーティ・リード 157
独立性レベル 157
幻ロー 166
幻ロック 169
ロックの意味 169
重複結果
削除 286
重複したクエリ結果の削除 286
重複するロー
UNION による削除 324
重複排除
クエリ実行アルゴリズム 521
直積 349
直列可能なスケジュール
2 フェーズ・ロック 187
影響 153
説明 152
ロックの早期解放 189
直列不可能なトランザクションのスケジューリング
影響 153

つ

追加
データベースへのデータの追加 687, 691
ツール
データのアンロード 710
データのインポート 693
データのエクスポート 705
データの再ロード 727
データのロード 699
データベースの再構築 727
常にトランザクション・ログを使用する 211

て

定数式デフォルト 113
ディスク・アクセスのコスト・モデル
説明 498
データ
XML としてエクスポート 639
一貫性 140
インポート 691
インポートとエクスポート 688
エクスポート 704, 705
大文字と小文字の区別 607
検索 80
整合性と正当性 152
追加、変更、削除 481
データの修正に必要なパーミッション 482
表示 57
無効 102
データ一貫性
実際のロックの意味 169
データ型
EXCEPT 演算 324
INTERSECT 演算 324
SQL と C 885
Transact-SQL の timestamp 609
UNION 演算 324
カラムへの割り当て 119
再帰サブクエリ 409

削除　122
作成　119, 120
集合関数　309
選択　31
ユーザ定義　119
リモート・プロシージャ　774
データが有効でなくなる場合　102
データ修正のパーミッション　482
データ修正文
説明　482
データ整合性
確保　101, 124
カラム制約　32
カラム・デフォルト　107
検査　128
システム・テーブル内の情報　130
制約　105, 114
説明　102
喪失　127
直列不可能なスケジュールの影響　153
ツール　103
データ整合性の確保　101
データ・ソース
外部サーバ　793
データ入力
独立性レベル　154
[データのアンロード] ダイアログ
使用　711
データの一貫性
2 フェーズ・ロック　187
ISO SQL/92 標準　140
繰り返し可能読み出し　140, 160, 180
正当性　152
ダーティ・リード　140, 180
ダーティ・リードのチュートリアル　157
幻ロー　141, 166, 180, 188
ロックを使用して保証　175
データの移動
インポート　691
インポートとエクスポート　688
エクスポート　704
説明　687
パフォーマンス　689
データのインポート
DEFAULTS オプション　703
INPUT 文　695
INSERT 文　698
LOAD TABLE 文　697, 700
NULL 値を含む　702
一致しないテーブル構造　702
[インポート] ウィザード　693
インポート／エクスポートの状況　688
概要　687
考慮事項　691
説明　688
他のデータベースから　699
ツール　693, 699
データベースのバックアップ　690
データベースへのデータのインポート　691
テンポラリ・テーブル　702, 703
パフォーマンス　689
パフォーマンスのヒント　692
プロキシ・テーブル　699
変換エラー　700
データのインポートとエクスポート
説明　687
データのインポートのパフォーマンスのヒント　692
データのエクスポート
dbunload ユーティリティ　709
Interactive SQL ウィザード　705
OUTPUT 文　706
UNLOAD TABLE 文　707
UNLOAD 文　708
概要　687
考慮事項　704
スキーマ　729
説明　688, 704
ツール　705
データベースのバックアップ　690
データの削除
DELETE 文　492
TRUNCATE TABLE 文　493
説明　481

データの出力
インポートとエクスポート 688
ファイル 708
ファイルへの保存 743
データの挿入
BLOB 488
INPUT 文 695
INSERT の使用 484
INSERT 文 698
NULL の追加 485
SELECT の使用 486
カラム・データ INSERT 文 485
制約 485
全カラム 484
デフォルト 485
データの追加
BLOB 488
INSERT の使用 484
NULL の追加 485
SELECT の使用 486
カラム・データ INSERT 文 485
制約 485
説明 481
全カラム 484
デフォルト 485
データの追加、変更、削除
説明 481
データの転送
INPUT 文 695
INSERT 文 698
LOAD TABLE 文 697
OUTPUT 文 706
UNLOAD TABLE 文 707
UNLOAD 文 708
インポート 691
[インポート] ウィザード 693
インポート／エクスポートの状況 688
インポートとエクスポート 688
説明 687
パフォーマンス 689
パフォーマンスのヒント 692
変換エラー 700
データの入力
INPUT 文 695
INSERT 文 698
LOAD TABLE 文 697
インポートとエクスポート 688
データの変更
UPDATE 文 489
説明 481
パーミッション 482
複数のテーブルを使用したデータの更新 490
データのロード
ツール 699
変換エラー 700
データベース
ASA への移行 733
jConnect メタデータ・サポートのインストール 49
SQL からの作成 40
SQL からの初期化 40
Sybase Central からの作成 39
Sybase Central からの初期化 39
Transact-SQL との互換性 602
Windows CE 用に作成 39
XML のインポート 641
XML の格納 637
アンロード 715
アンロードと再ロード 722, 724, 730
エクスポート 715
大文字と小文字の区別 603, 606
オブジェクトを使用した作業 35
オプションの設定 45
コマンド・ラインからの作成 41
コマンド・ラインからの消去 42
再構築 722
再ロード 730
作業 37
削除 41
作成 37
システム・オブジェクトの表示 47
システム・テーブルの表示 67
消去 41
初期化 37

正規化　19
設計　3
設計概念　5
設計の検証　28
接続しないで起動　48
抽出　732
データベースとの接続の切断　43
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　719
統合データベースの設定　46
トランザクション・ログ　37
ファイルの互換性　37
プロパティの表示と編集　44
[データベース・アップグレード] ウィザード
jConnect メタデータ・サポートのインストール　49
[データベース・アンロード] ウィザード
使用　715
説明　710
[データベース移行] ウィザード
使用　733
[データベース移行] ウィザード
説明　733
データベース・オブジェクト
Sybase Central におけるコピー　98
プロパティの編集　44
データベース・オブジェクトの使用　35
データベース・オブジェクトのプロパティの設定　44
データベース・オプション
Transact-SQL 互換性のための設定　605
インデックス・コンサルタント　96
データベース・オプションの設定　45
データベース管理者
役割　598
[データベース作成] ウィザード
Transact-SQL と互換性のあるデータベースの作成　602
使用　39
[データベース消去] ウィザード
使用　42
データベース・スレッド
ブロック　149
データベース設計の概念　5
[データベース抽出] ウィザード
説明　732
データベース・テーブル・プロパティの設計　31
データベースでのデバッグ論理　889
データベース統計
説明　251
データベース内の整合性制約　103
データベースにおける XML の使用　635
データベースのアップグレード　720
データベースのアンロード
カンマ区切りの ASCII フォーマット　727
使用　715
説明　719
データベースの移行
sa_migrate ストアド・プロシージャ　735
説明　733
[データベース移行] ウィザード　733
データベースのエクスポート
使用　715
データベースのコピー
データのレプリケーションと同時性　195
データベースの再構築
dbunload の使用　725
UNLOAD TABLE 文　728
エクスポートのコマンド　719
考慮事項　722
コマンド・ライン　730
説明　719
ツール　727
非レプリケート・データベース　722
ユーザに対する影響　724
理由　720
レプリケート・データベース　724
データベースの再ロード
コマンド・ライン　730
説明　719
データベースの作成　37
Sybase Central　39

データベースのシステム・オブジェクトの表示 47
データベースの設計 3
概念 5
説明 3
手順 13
データベースのデータが変わる場合 103
データベースのパフォーマンスのモニタリング 246
データベースのブラウズ
独立性レベル 154
データベースの編集 37
データベース・ファイル
断片化 224, 254
パフォーマンス 214
データベース・プロシージャ
プロファイリング・データの表示 259
データベース・プロシージャのプロファイリング
説明 259
データベース・ページ
インデックス・コンサルタントの推奨内容 93
データ編成
物理的 531
データ・モデリング
説明 3
データ・モデルの正規化 19
データ・リカバリ
インポートとエクスポート 690
テーブル
SQL を使用したプロキシ・テーブルの作成 765, 766
Sybase Central からのアンロード 711
Sybase Central でのプライマリ・キーの表示 58
Sybase Central を使用したプロキシ・テーブルの作成 764
Transact-SQL 614
インポート 700
エクスポート 717
外部キーの管理 60, 62
カラム名 31
キーの追加 58, 59, 60, 62
クエリでの命名 288
グループ読み込み 533
検査制約 117
構造 53
コピー 67
削除 56
作成 51
システム・テーブル 67
処理 51
制約 32
相関名 288
断片化 255
データのエクスポート 729
データのブラウズ 57
データの編集 57
データベース内の個々のテーブルのデフラグメンテーション 256
データベース内のすべてのテーブルのデフラグメンテーション 256
テーブル制約の管理 116
ビットマップ 533
プライマリ・キーの管理 58, 59
プロキシ 762
プロパティ 31
変更 53, 54, 55
リモート・アクセス 748
リモートのリスト 756
ローのコピー 487
ワーク・テーブル 243
テーブルから全ローを削除 493
テーブルからのすべてのカラムの選択 279
テーブルからの特定カラムの選択 281
テーブル検証時の WITH EXPRESS CHECK オプションの使用 218
テーブル構造の正規化 213
テーブル・サイズ
説明 533
テーブル式
キー・ジョイン 378
構文 338

ジョイン方法　342
テーブル・スキャン
説明　511
ディスク割り付けとパフォーマンス　531
ハッシュ・テーブル・スキャン・アルゴリズム　512
テーブル制約とカラム制約の使い方　114
テーブルとページのサイズ　533
テーブルに対する検査制約の使い方　115
テーブルのエクスポート
スキーマ　729
説明　717
テーブルの検証
WITH EXPRESS CHECK　218
テーブルの断片化
説明　255
テーブルの編集　51
テーブルへのアクセス
インデックス・スキャンと逐次スキャン　510
テーブル名
識別　277
プロシージャでの完全修飾　874
プロシージャとトリガ　874
テキスト・プラン
アクセス・プランの解釈　572
適切なページ・サイズの使用　222
テクニカル・サポート
ニュースグループ　xviii
テスト
データベース設計　19, 28
デッドロック
診断　150
説明　148
トランザクションのブロック　149
理由　149
デッドロック・レポート
説明　150
デバイス
管理　595
デバッガ
Java クラスのデバッグ　899
稼働条件　891
起動　893
機能　890
ストアド・プロシージャのデバッグ　894
接続　893
接続の活用　912
説明　890
ソース・コードの表示　900
チュートリアル　893
はじめに　893
ブレークポイントの活用　906
変数の検査　910
デバッグ
Java　899
概要　890
稼働条件　891
クラスのコンパイル　900
ストアド・プロシージャ　894
説明　889
チュートリアル　894, 899
パーミッション　891
ブレークポイント　903
ローカル変数　904
デフォルト
INSERT 文　485
NEWID　111
NULL　112
Sybase Central で作成　108
Transact-SQL　596
オートインクリメント　110
概要　103
カラム　107
現在の日付と時刻　109
作成　107
定数式　113
ドメインでの使用　121
トランザクションとロック　193
文字列と数値　113
ユーザ ID　110
デフラグメンテーション
説明　254
データベース内の個々のテーブル　256
データベース内のすべてのテーブル　256

ハード・ディスク　254
典型的な矛盾のケース　140
テンポラリ・テーブル　51
Transact-SQL　614
インデックス　536
クエリ処理中のワーク・テーブル　243
説明　97
データのインポート　702, 703
ローカルとグローバル　97
テンポラリ・ファイル
ワーク・テーブル　214

と

等価ジョイン
説明　347
統計
アクセス・プラン　564, 567
カラム統計の更新　501
パフォーマンス　251
パフォーマンス・モニタからの削除　249
パフォーマンス・モニタに追加　249
表示　251
モニタリング　246
有効　251
統計が存在し、正確である　503
統計情報付きのグラフィカルなプラン
Interactive SQL　584
SQL 関数　584
説明　575
表示　578
統計値の削除　249
統計値の追加と削除　249
等号演算子
比較演算子　291
統合データベース
設定　46
統合データベースの指定　46
同時性　136
DDL 文　194
ISO SQL/92 標準　140
一貫性　140
インデックスの使用による改善　155
改善　155
改善とインデックス　190
説明　140, 192
パフォーマンス　137
プライマリ・キー　192
矛盾　140
利点　136
レプリケーション　195
ロックの仕組み　175
ロックの種類　177
同時トランザクション
ブロック　148, 165
同時に複数のトランザクション　136
動的キャッシュ・サイズ決定
UNIX の説明　234
Windows の説明　234
説明　231
ときどき接続されるユーザ
データのレプリケーションと同時性　195
ドキュメント
挿入　488
ドキュメントとイメージの挿入　488
特殊な IDENTITY カラム　611
特殊なジョイン　360
特殊な同時性の問題　192
独立性のレベル
説明　139
独立性レベル
ODBC　145
各レベルの典型的なトランザクション　154
参照　147
設定　143
説明　139
選択　152
チュートリアル　157
デフォルトの設定　143
典型的なトランザクション　154
典型的な矛盾　141, 166, 169, 188
トランザクション内の変更　146
変更　143

レベル 0 で実装　180
レベル 1 で実装　180
レベル 2 で実装　180
レベル 2 と 3 での同時性の改善　155
レベル 3 で実装　181
ロック・タイプの選択のチュートリアル　164
独立性レベル 0
SELECT 文のロック　180
例　157
独立性レベル 1
SELECT 文のロック　180
例　160
独立性レベル 2
SELECT 文のロック　180
例　166, 169
独立性レベル 2 と 3 での同時性の改善　155
独立性レベル 3
SELECT 文のロック　181
例　168
独立性レベルと一貫性　139
独立性レベルの参照　147
独立性レベルの設定　143
独立性レベルの選択　152
独立性レベルの変更　143
どのようなときにインデックスを作成するか　535
ドメイン
大文字と小文字の区別　607
カラムへの割り当て　119
検査制約　116
削除　122
作成　119, 120
使用　119
使用例　119
[ドメイン作成] ウィザード
使用　119
ドメインのカラム検査制約　116
ドメインの使い方　119
トラブルシューティング
ANY 演算子　456
クラスのデバッグ　900
デッドロック　150
パフォーマンス　211
リモート・データ・アクセス　783
トランザクション
開始　135
干渉　148, 165
管理　777
完了　135
サブトランザクションとセーブポイント　137
使用　135
セーブポイント　137
説明　134
データ修正　482
デッドロック　149
同時性　136
同時に複数　136
同時レプリケーション　195
プロシージャとトリガ　873
ブロック　148, 149, 165
リモート・データ・アクセス　776
トランザクション間の干渉　148, 165
トランザクション管理の制限　777
トランザクション結果の保存　135
トランザクション処理
2 フェーズ・ロック　187
スケジューリング　152
スケジューリングの影響　153
直列可能なスケジューリング　152
トランザクション・ログ・ベースのレプリケーション　196
パフォーマンス　137
ブロック　148, 165
トランザクション・スケジューリング
影響　153
トランザクションとデータ修正　482
トランザクションと独立性レベル　133
トランザクションと独立性レベルの使用　133
トランザクション内の独立性レベルの変更　146
トランザクションのインタリーブ　152
トランザクションの管理　777
リモート・データ　776

トランザクションの完了　135
トランザクションの終了　135
トランザクションの順序付け　152
トランザクションの使用　135
トランザクションのスケジューリング　152
トランザクションのロールバック　135
トランザクション・ブロック
　説明　148
トランザクション・ログ
　データベースのレプリケート　724
　データ・リカバリ　690
　データ・レプリケーションの役割　196
　パフォーマンス　211
トリガ
　ROLLBACK 文　620
　Transact-SQL　608, 620
　エラー処理　860
　カーソル　856
　概要　812
　許可される SQL 文　845
　警告　865
　構造　845
　個々のプロファイリング情報の表示　267
　コマンド・デリミタ　874
　再帰　620
　削除　836
　作成　831
　時間　875
　実行　834
　実行パーミッション　837
　使用　830
　セーブポイント　873
　説明　811
　日付　875
　プロファイリング・データ一覧の表示　265
　プロファイリング・データの表示　259
　文レベル　620
　変更　835
　利点　813
　例外ハンドラ　866
[トリガ作成] ウィザード
　使用　832
トリガの実行　834
取り消し
　外部関数　885

な

内部オペレーション
　リモート・データ・アクセス　778
内部ジョイン
　説明　351
内部ジョインと外部ジョイン　351
内部ロード　689
長いプラン
　Interactive SQL　584
　SQL 関数　584
　説明　574
ナチュラル・ジョイン
　ON 句　370
　説明　369
　テーブル式　370
　ビューと抽出テーブル　372

に

二重引用符
　文字列　297
ニュースグループ
　テクニカル・サポート　xviii
入力
　データのインポート　691
　テーブルのインポート　700

ね

ネイティブ文のリモート・サーバへの送信　772
ネーム・スペース
　インデックス　608
　トリガ　608
ネスト

外部ジョイン　354
ジョイン　342
ジョインでの抽出テーブル　368
ネストされたサブクエリ
説明　467
ネストされた複合文と例外処理　868
ネスト・ブロック・ジョイン
クエリ実行アルゴリズム　516
ネスト・ブロック・ジョインとソート・ブロック　516
ネスト・ループ・ジョイン
クエリ実行アルゴリズム　515
ネスト・ループ・セミジョイン
クエリ実行アルゴリズム　516

は

パーミッション
ASE　599
外部関数を呼び出すプロシージャ　879
データの修正　482
デバッグ　891
トリガ　837
プロシージャの結果セット　851
ユーザ定義関数　829
排他ロック　177
排他ロックと共有ロック　178
バイナリ・ラージ・オブジェクト
説明　31
挿入　488
バケット
ヒストグラム　498
パスワード
Lotus Notes　807
大文字と小文字の区別　607
ハッシュ B ツリー・インデックス
説明　543
ハッシュ DISTINCT
クエリ実行アルゴリズム　521
ハッシュ GROUP BY
クエリ実行アルゴリズム　523
ハッシュ GROUP BY SETS
クエリ実行アルゴリズム　524
ハッシュ NOT EXIST
クエリ実行アルゴリズム　519
ハッシュ・ジョイン
クエリ実行アルゴリズム　517, 518, 519
ハッシュ・セミジョイン
クエリ実行アルゴリズム　518
ハッシュ値
インデックス　543
ハッシュ・テーブル・スキャン
アルゴリズム　512
ハッシュ非セミジョイン
クエリ実行アルゴリズム　519
ハッシュ・マップ
クエリ実行アルゴリズム　529
バッチ
SELECT 文の使用　877
Transact-SQL の概要　621
許可される SQL 文　877
許可される文　877
作成　621
制御文　839
説明　839
データ定義文　839
バッチ処理
Interactive SQL　742
バッチで SELECT 文を使用する　877
バッチに使用できる文　877
バッチ・モード
Interactive SQL　742
パフォーマンス
Windows でモニタリング　251
Windows パフォーマンス モニタの統計　251
WITH EXPRESS CHECK　218
アクセス・プランの解釈　564
圧縮　218
インデックス　79, 213
インデックスの断片化　256
オートコミット　217
オプティマイザによる推定と実際の統計の比較　579
改善　80, 81

改善とロック　155
キー　240
キャッシュ　212
キャッシュ読み込みヒット率　581
クエリ速度の測定　257
実行時間の実際値と推定値　581
自動チューニング　501
述部の分析　547
推定ソース　580
説明　211
選択性　580
データベース設計　213
テーブルとページのサイズ　533
テーブルの断片化　255
トランザクション・ログ　211
バルク・オペレーション　218
バルク・ロード　689
ヒント　211
ファイル管理　214
ファイルの断片化　224, 254
分散読み込み　224
ページ・サイズ　222
ページ・サイズの推奨値　545
モニタリング　246
要求の削減　219
ワーク・テーブル　243
パフォーマンス改善　211
パフォーマンス向上へのキャッシュの使用　231
パフォーマンスの自動チューニング　501
パフォーマンスのモニタ
アクセス・プランに使用される省略形　564
アクセス・プランの解釈　564
パフォーマンスのモニタリング
クエリ測定ツール　257
パフォーマンスのモニタリングと改善　201
パフォーマンス・モニタ
Sybase Central　248
Sybase Central で開く　248
Windows　251
概要　248
間隔時間の設定　250
起動　251
設定　250
統計値の追加と削除　249
複数コピーの実行　251
パブリケーション
データのレプリケーションと同時性　195
パラメータを外部関数に渡す　885
パラメータを関数に渡す　848
パラメータをプロシージャに渡す　847
バルク・オペレーション
データのリカバリの問題　690
パフォーマンス　218
バルク・オペレーションのパフォーマンスの側面　689
バルク・オペレーション方法の使用　218
バルク・ロード
パフォーマンス　689
範囲クエリ　293

ひ

比較
NULL 値　300
後続ブランク　291
比較演算子
NULL 値　300
記号　291
サブクエリ　471
比較テスト
サブクエリ　453
ヒストグラム
更新　501
説明　498
非セミジョイン
クエリ実行アルゴリズム　519
非相関サブクエリ
説明　469
非ダーティ・リード
チュートリアル　157
左外部ジョイン
説明　352

日付
検索 297
入力規則 297
プロシージャとトリガ 875
必須
外部キー 126
必須の関係 9
ビットマップ 31
スキャン 533
等しくない
比較演算子 291
非排他ロック 177
ビュー
FROM 句 288
SYSCOLUMNS 604
SYSINDEXES 604
エクスポート 706
外部ジョイン 355
キー・ジョイン 385
共通テーブル式 391
更新 71
コピー 72
削除 76
作成 70
修正 75
使用 71
処理 69
チェック・オプション 72
データのブラウズ 77
ナチュラル・ジョイン 372
[ビュー作成] ウィザード
使用 70
ビューのエクスポート
説明 717
ビューの使い方 71
ビューの編集 69
表記
規則 xiii
表示
テーブル・データ 57
ビュー・データ 77
標準出力
ファイルへのリダイレクト 711
ヒント
パフォーマンス 211

ふ

ファイル
断片化 224
パフォーマンス 214
ファイル、テーブル、インデックスの断片化の確認 224
ファイルの断片化
説明 254
ファイルへの出力の書き込み 743
フィードバック
提供 xviii
マニュアル xviii
フィルタ
クエリ実行アルゴリズム 528
フィルタと事前フィルタ 528
負荷
インデックス・コンサルタント 87
取得 87, 91
知識 91
定義 86
分析 87
負荷の取得
一時停止 90
インデックス・コンサルタント 87, 91
終了 90
負荷の分析
インデックス・コンサルタント 87
複合インデックス
カラムの順序の効果 539
説明 538
ハッシュ値 544
複合文
アトミック 843
使用 842
宣言 843
複合文での宣言 843

複合文の使用　842
複雑な外部ジョイン　354
複雑な外部ジョインについて　354
複数のカラムを持つ GROUP BY　316
複数のデータベース
　ジョイン　770
複数のトランザクション
　同時性　136
複数のローカル・データベースのテーブルのジョイン　770
物理データの編成　531
物理データ・モデル
　生成　24
不定の値
　説明　299
部分インデックス・スキャン
　説明　542
不要な DISTINCT の削除　550
プライマリ・キー
　NEWID を使用して UUID を作成　111
　エンティティ整合性　125
　オートインクリメント　110
　管理　58
　作成　58, 59
　修正　58, 59
　整合性　630
　生成　192
　生成されたインデックス　535
　同時性　192
　パフォーマンス　240
プライマリ・キーによるエンティティ整合性の確保　125
プライマリ・キーの管理　58
プライマリ・キーを使ったクエリのパフォーマンス改善　240
ブラウズ
　テーブル・データ　57
　ビュー　77
プラス演算子
　NULL 値　302
プラン
　Interactive SQL　584
　SQL 関数　584
　印刷　575
　解釈　564
　カスタマイズ　575
　キャッシュ　507
　グラフィカルなプラン　575
　コンテキスト別のヘルプ　576
　使用される省略形　564
　長いテキスト・プラン　574
　短いテキスト・プラン　573
プランに使用される一般的な推定　567
プランに使用される一般的な統計　567
ブルーム・フィルタ
　クエリ実行アルゴリズム　529
ブレーク条件
　設定　906
ブレークポイント
　カウント　908
　個々の接続　907
　個々のユーザ　907
　条件　908
　ステータス　907
　設定　906
　説明　906
　デバッグ　903
　無効化　907
　有効化　907
ブレークポイント条件の編集　908
ブレークポイントの活用　906
ブレークポイントの設定
　デバッガ　906
ブレークポイントの無効化
　説明　907
　有効化　907
ブレークポイントの有効化
　説明　907
プレフィクス
　ROLLUP 演算　430
　小計ロー　430
プレフィクス・カラム
　ROLLUP 演算　430
プロキシ・テーブル

作成　748, 764, 765, 766
説明　748, 762
データのインポート　699
ロケーション　762
[プロキシ・テーブル作成] ウィザード
使用　764
プロキシ・テーブルの編集　762
プロキシ・テーブルのロケーションの指定　762
プログラム変数
共通テーブル式　397
プロシージャ
EXECUTE IMMEDIATE 文　871
FROM 句内での使用　288
Transact-SQL　622
Transact-SQL の概要　619
エラー処理　626, 628, 860
カーソル　856
カーソルの使用　856
外部関数　879
概要　812
許可される SQL 文　845
警告　865
結果セット　823, 851
結果セットに対するパーミッション　851
結果を返す　821, 849
構造　845
コピー　820
コマンド・デリミタ　874
削除　820
作成　815
作成のヒント　874
時間　875
使用　815
セーブポイント　873
セキュリティ　813
説明　811
テーブル名　874
デフォルトのエラー処理　860
入力検証　875
パラメータ　846, 847
日付　875
複数の結果セット　853
変換　622
変更　817
変数結果セット　854
戻り値　626
呼び出し　819
利点　813
リモート・プロシージャの削除　775
リモート・プロシージャの追加　773
例外ハンドラ　866
プロシージャから返される結果　849
プロシージャから結果セットを返す　851
プロシージャからの外部ライブラリの呼び出し　879
プロシージャから複数の結果セットを返す　853
プロシージャから変数結果セットを返す　854
プロシージャ言語
概要　619
[プロシージャ作成] ウィザード
使用　815
説明　815
プロシージャでの EXECUTE IMMEDIATE 文の使用　871
プロシージャとトリガで使用できる SQL 文　845
プロシージャとトリガでのエラーと警告　860
プロシージャとトリガでのカーソルの使用　856
プロシージャとトリガでのデフォルトのエラー処理　860
プロシージャとトリガでのデフォルトの警告処理　865
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　873
プロシージャとトリガでの例外ハンドラの使用　866
プロシージャとトリガの構造　845
プロシージャとトリガの利点　813
プロシージャ、トリガ、バッチの使用　811

プロシージャ内のテーブルでは完全に修飾された名前を使用　874
プロシージャの SELECT 文でのカーソルの使用　856
プロシージャの結果を結果セットとして返す　823
プロシージャの結果をパラメータとして返す　821
プロシージャの中での RAISERROR 文の使用　627
プロシージャの中で日付と時刻を指定する　875
プロシージャの中で文を区切る　874
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　875
プロシージャの呼び出し　819
プロシージャ・パラメータの宣言　846
プロシージャ・プロファイリング
　Interactive SQL でのデータ表示　268
　SQL を使用したクリア　262
　SQL を使用した無効化　262
　SQL を使用した有効化　259
　SQL を使用したリセット　261
　Sybase Central でのクリア　262
　Sybase Central でのデータ表示　259, 263
　Sybase Central での無効化　262
　Sybase Central での有効化　259
　Sybase Central でのリセット　261
　イベント　265, 267
　個々のプロシージャの情報　266, 270
　システム・トリガ　265, 268
　ストアド・プロシージャと関数　264, 266
　トリガ　265, 267
　プロシージャの概要　268
　要約情報　263
プロシージャ・プロファイリング・データの表示
　Sybase Central　263
プロシージャ・プロファイリングのクリア
　SQL　262
　Sybase Central　262
プロシージャ・プロファイリングの無効化
　SQL　262
　Sybase Central　262
プロシージャ・プロファイリングの有効化
　SQL　259
　Sybase Central　259
プロシージャ・プロファイリングのリセット
　SQL　261
　Sybase Central　261
プロシージャを作成するときのヒント　874
ブロック　148, 165
　デッドロック　149
　トラブルシューティング　150
　トランザクション　148
ブロック・ネスト・ループ・ジョイン
　クエリ実行アルゴリズム　516
プロトコル
　2 フェーズ・ロック　187
プロトタイプ
　外部関数　881
プロパティ
　すべてのデータベース・オブジェクトのプロパティを設定する　44
　定義　5
プロパティ・シート　44
プロファイリング情報
　イベント　267
　システム・トリガ　268
　ストアド・プロシージャと関数　266
　トリガ　267
プロファイリング・データ一覧
　イベント　265
　システム・トリガ　265
　ストアド・プロシージャと関数　264
　トリガ　265
文
　CALL　812, 819, 841, 847
　CASE　841
　CLOSE　856
　COMMIT　843, 873
　CREATE DATABASE　595
　CREATE DEFAULT　596
　CREATE DOMAIN　596

CREATE PROCEDURE 815, 846
CREATE RULE 596
CREATE TABLE 614
CREATE TRIGGER 831
DECLARE 843, 856, 861
DISK 595
DROP DATABASE 595
DUMP DATABASE 595
DUMP TRANSACTION 595
EXECUTE IMMEDIATE 871
FETCH 856
FOR 841
GRANT 599
IF 841
LEAVE 841
LOAD DATABASE 595
LOAD TRANSACTION 595
LOOP 841, 856
MESSAGE 861
OPEN 856
OUTPUT 711
RAISERROR 627, 628
RETURN 849
REVOKE 599
ROLLBACK 620, 873
SELECT 615, 617, 850
SIGNAL 627, 861
WHILE 841
最適化 496
複合 842
ロギング 47
分散読み込み
パフォーマンス 224
分析
インデックス・コンサルタント 92
知識 92
文の完全なパススルー
リモート・データ・アクセス 780
文の部分的なパススルー
リモート・データ・アクセス 780
文レベルのトリガ 620
文をプロシージャ内で区切る 874

へ

並列インデックス・スキャン
アルゴリズム 513
テーブルのスキャン 513
ページ
挿入されたローに対するディスク割り付け 531
ページ・サイズ
インデックス 534, 545
説明 533
挿入されたローに対するディスク割り付け 531
パフォーマンス 222
ページ・サイズの推奨値 545
ページ・マップ
スキャン 533
ベース・テーブル 51
べき等関数
定義 562
ベクトル集合 312
別個のプライマリ・キー・インデックスと外部キー・インデックス 241
変形
リライト最適化 550
変更
カラム 54, 55
テーブル 53, 54, 55
トリガ 835
ビュー 75
プロシージャ 817
編集
データベース・オブジェクトのプロパティ 44
テーブル・データ 57
変数
SELECT 文 617
SET 文 617
Transact-SQL 625
デバッグ 904
ローカル 617
割り当て 617
変数の検査

デバッガ 910
編成
物理データ 531

ほ

ポータブル・コンピュータ
データベースのレプリケーション 195
保護されたテーブル
外部ジョイン 352
保存
結果セット 743

ま

マージ・ジョイン
クエリ実行アルゴリズム 520
マージ・ソート
クエリ実行アルゴリズム 526
マニュアル
SQL Anywhere Studio x
幻
ロー 141, 180, 188
幻ロー
チュートリアル 166
独立性レベル 141, 169, 188
幻ロック 169

み

右外部ジョイン
説明 352
短いプラン
Interactive SQL 584
SQL 関数 584
説明 573
未使用インデックス
インデックス・コンサルタントの推奨内容 95

む

無効なデータ 102
矛盾
ISO SQL/92 標準 140
繰り返し不可能読み出しの例 162
実際のロックの意味 169
ダーティ・リード 140, 180
ダーティ・リードのチュートリアル 157
直列不可能なスケジュールの影響 153
幻ロー 141, 166, 180, 188
ロックを使用して防止 175

め

明示的ジョイン条件 339
明示的ジョイン条件の種類 347
明示的なジョイン条件 (ON 句) 345
命令グラフ 411
メタデータ・サポート
jConnect に対するインストール 49
メンテナンス
パフォーマンス 211

も

文字データ
検索 297
文字列
select リストの使用 284
引用符 297, 298
説明 297
マッチング 295
文字列と引用符 297
文字列と数値のデフォルト 113
文字列の連結
NULL 302
戻り値
プロシージャ 626
モニタ
Sybase Central パフォーマンス・モニタの設定

250
Sybase Central パフォーマンス・モニタを開く 248
パフォーマンス・モニタの概要 248

や

役割
ASE 597
定義 8
役割名
説明 375

ゆ

ユーザ
ときどき接続される 195
ユーザ ID
ASE 599
大文字と小文字の区別 607
デフォルト 110
ユーザ定義関数
外部関数 879
キャッシュ 562
削除 828
作成 825
実行パーミッション 829
説明 825
パラメータ 848
呼び出し 826
ユーザ定義関数の呼び出し 826
ユーザ定義データ型 119
検査制約 116
削除 122
作成 119, 120
ユニーク・キー
生成と同時性 192
ユニークな結果
制限 286
ユニークな識別子
テーブル 58

よ

要求
数の削減 219
[要求] タブ
インデックス・コンサルタントの推奨内容 94
要求のキャンセル
リモート・データ・アクセス 786
要求ロギング
説明 202
要求ログ
説明 202
要素
クエリ結果を XML として取得 650
データベースへの XML の格納 637
リレーショナル・データからの XML の生成 639
呼び出しスタック
デバッガ 910
読み込みロック
説明 177
対挿入ロック 186
予約語
リモート・サーバ 783
より大きい
範囲指定 293
比較演算子 291
より大きくない
比較演算子 291
より小さい
範囲指定 293
比較演算子 291
より小さくない
比較演算子 291

ら

ラップトップ・コンピュータ
トランザクション 195

り

リーフ・ページ　537
利益
　インデックス・コンサルタントの推奨内容　94
リカバリ
　インポート／エクスポート　690
リターン・タイプ
　外部関数　886
リダイレクト
　ファイルへの出力　711
リテラル値
　NULL　302
リモート・サーバ
　外部ログイン　759
　クラス　787, 788
　削除　755
　作成　751
　説明　751
　トランザクション管理　776
　ネイティブ文の送信　772
　プロパティのリスト　757
　変更　755
[リモート・サーバ作成] ウィザード
　使用　753
リモート・サーバの機能のリスト　757
リモート・サーバの削除　755
リモート・サーバの作成　751
リモート・サーバの使用　751
リモート・サーバの変更　755
リモート・データ
　ロケーション　762
リモート・データ・アクセス
　Lotus Notes SQL 2.0　806
　Microsoft Access　805
　Microsoft Excel　804
　Microsoft FoxPro　806
　PowerBuilder DataWindows　747
　大文字と小文字の区別　784
　概要　746
　サポートされていない SQL Remote　783
　サポートされていない機能　783
　接続名　786
　説明　745
　トラブルシューティング　783
　内部オペレーション　778
　パススルー・モード　772
　パフォーマンスの制限　746
　リモート・サーバ　751
リモート・データ・アクセスの基本概念　748
リモート・データ・アクセスのサーバ・クラス　787
リモート・データにサポートされていない機能　783
リモート・データベース
　レプリケーション　195
リモート・データへのアクセス　745
　基本概念　748
リモート・テーブル
　アクセス　745
　カラムのリスト　767
　説明　748
　リスト　756
リモート・テーブルのカラムのリスト　767
リモート・テーブルのジョイン　768
リモート・テーブルのマッピング　748
リモート・トランザクション管理の概要　776
リモート プロシージャ
　コール　773
リモート・プロシージャ
　削除　775
　作成　815
　追加　773
　データ型　774
リモート・プロシージャ・コール
　説明　773
リモート・プロシージャ・コールの使用　773
[リモート・プロシージャ作成] ウィザード
　使用　773, 816
リモート・プロシージャの削除　775
リモート・プロシージャの作成　773

利用可能な条件の発見　559
リライト最適化
　説明　550
リレーショナル・データ
　XML としてエクスポート　639
リレーショナル・データベースにおける
　　XML 文書の格納　637
リレーショナル・データを XML としてエク
　　スポートする　639

る

ルール
　Transact-SQL　596

れ

例
　繰り返し不可能読み出し　160
　ダーティ・リード　157
　幻ロー　166
　幻ロック　169
　ロックの意味　169
例外
　宣言　861
例外ハンドラ
　ネストされた複合文　868
　プロシージャとトリガ　866
レプリケーション
　データベースの再構築　724
　同時性　195
　同時性の問題点　192
レプリケーションと同時性　195
レポート関数
　説明　442
　例　442, 443
連結された GROUPING SETS
　説明　437
連結された GROUPING SETS の使用　437

ろ

ロー
　INSERT を使用したコピー　487
　削除　533
　小計ロー　421
　選択　290
ローカル・テンポラリ・テーブル
　説明　97
ローカル変数
　デバッガ　910
ロード
　Interactive SQL 内のコマンド　743
ロード・ツール　699
ローの制限
　FIRST 句　321
　TOP 句　321
　クエリ実行アルゴリズム　529
ローの全カラムへの値の挿入　484
ローの連続記憶領域　531
ロールバック・ログ
　セーブポイント　138
　データ・リカバリ　690
ログ
　ロールバック・ログ　138
ロック
　2 フェーズ・ロック　187
　Sybase Central での表示　175
　インデックスによって影響を削減　155
　インデックスによる減少　542
　オーファンと参照整合性　183
　書き込み　177
　競合処理　148, 165
　競合のタイプ　179
　共有と排他　178
　クエリ実行アルゴリズム　529
　更新手順　184
　削除手順　185
　種類　177
　使用　177
　説明　175
　早期解放　154, 187
　挿入　177

挿入手順　182
対挿入　177, 186
デッドロック　149
典型的なトランザクションと独立性レベル　154
独立性レベル　139
独立性レベルの選択のチュートリアル　164
トランザクションのブロックとデッドロック　148
排他　177
非排他　177
ブロック　148, 165
幻ローと独立性レベル　166, 169
矛盾と典型的な独立性レベル　141, 188
読み込み　177, 186
読み込みロックの早期解放　189
レベル 0 で実装　180
レベル 1 で実装　180
レベル 2 で実装　180
レベル 3 で実装　181
ロックできるオブジェクト　175
ロック可能なオブジェクト　175
ロックされたローへのアクセス待機
デッドロック　148
ロックできるオブジェクト　175
ロックの影響の削減　155
ロックの仕組み　175
ロックの種類　177
ロックの使用　177
ロックの設定方法　188
ロックの早期解放　187
トランザクション　154
例外　189
論理演算子
HAVING 句　319
条件の接続　302
論理演算子による複数の条件の接続　302

わ

ワーク・テーブル
クエリ処理　243
説明　243
パフォーマンスのヒント　214
ワイルドカード
LIKE 演算子　296
文字列比較　295
割り当て
カラムのデータ型　119
カラムのドメイン　119