iAnywhere
A SYBASE COMPANY

SQL Anywhere® Studio 9

デベロッパーズガイド

Breck Carter 著
株式会社トップスタジオ 訳
アイエニウェア・ソリューションズ株式会社 監修

SE
SHOEISHA

iAnywhere
A SYBASE COMPANY

SQL Anywhere® Studio 9

デベロッパーズガイド

Breck Carter 著
株式会社トップスタジオ 訳
アイエニウェア・ソリューションズ株式会社 監修

SE
SHOEISHA

本書内容に関するお問い合わせについて

このたびは翔泳社の書籍をお買い上げいただき、誠にありがとうございます。弊社では、読者の皆様からのお問い合わせに適切に対応させていただくため、以下のガイドラインへのご協力をお願いしております。下記項目をお読みいただき、手順に従ってお問い合わせください。

●お問い合わせの前に

弊社Webサイトの「正誤表」や「出版物Q&A」をご確認ください。これまでに判明した正誤や追加情報、過去のお問い合わせへの回答（FAQ）、的確なお問い合わせ方法などが掲載されています。

正誤表　　http://www.seshop.com/book/errata/
出版物Q&A　http://www.seshop.com/book/qa/

●ご質問方法

弊社Webサイトの書籍専用質問フォーム（http://www.seshop.com/book/qa/）をご利用ください。記載漏れ、お電話や電子メールによるお問い合わせ、本書にはさみ込まれたアンケートはがきなど別紙に記入されたご質問は、お受けしておりません。

※質問専用シートのお取り寄せについて

Webサイトにアクセスする手段をお持ちでない方は、ご氏名、ご送付先（ご住所／郵便番号／電話番号またはFAX番号／電子メールアドレス）および「質問専用シート送付希望」と明記のうえ、電子メール（qaform@shoeisha.com）、FAX、郵便（80円切手同封）のいずれかにて“編集部読者サポート係”までお申し込みください。お申し込みの手段によって、折り返し質問シートをお送りいたします。

シートに必要事項を漏れなく記入し、“編集部読者サポート係”までFAXまたは郵便にてご返送ください。

●回答について

回答は、ご質問いただいた手段によってご返事申し上げます。ご質問の内容によっては、回答に数日ないしはそれ以上の期間を要する場合があります。

●ご質問に際してのご注意

本書の対象を越えるもの、記述個所を特定されていないもの、また読者固有の環境に起因するご質問等にはお答えできませんので、予めご了承ください。

●郵便物送付先およびFAX番号

送付先住所　〒160-0006　東京都新宿区舟町5
FAX番号　03-5362-3818
宛先　（株）翔泳社 出版局 編集部読者サポート係

※本書に記載されたURL等は予告なく変更される場合があります。
※本書の出版にあたっては正確な記述に努めましたが、著者および出版社のいずれも、本書の内容に対してなんらかの保証をするものではなく、内容やサンプルに基づくいかなる運用結果に関してもいっさいの責任を負いません。
※本書に掲載されている画面イメージなどは、特定の設定に基づいた環境にて再現される一例です。

※iAnywhere、iAnywhere Solutions、iAnywhere Solutionsのロゴ、Sybase、SQL Anywhere、Adaptive Server、Sybase Central、SQL Remote、jConnect、Physical Architect、InfoMaker、Watcom、Watcom SQL、RFID Anywhere、Manage Anywhere Studio、M-Business Anywhere、Powersoft、Embedded SQLおよびOpen Clientは、米国法人iAnywhere Solutions, Inc. または米国法人Sybase, Inc. の米国あるいは日本における登録商標または商標です。
※その他、本書に記載されている会社名、商品名、製品名などは、一般に各社の商標、または登録商標です。
※本書ではTM、®、©は割愛させていただいております。

SQL Anywhere Studio 9 developer's guide by Breck Carter
Copyright © 2004 by Wordware Publishing, Inc.

Japanese translation rights arranged with Waterside Productions, Inc. through Japan UNI Agency, Inc., Tokyo.

著者まえがき ～日本語版刊行にあたって

私の本の日本語訳版が発売されるということで、たいへん光栄に思います。本書の執筆にあたっては、当初、読者をだれに設定すべきか悩みましたが、結局1人の読者のために書くことにしました。それは「私」です。その甲斐あって、私が顧客訪問する際は本書を必ず携えますし、本書を開かない日は一日もありません。そして、多くの方にご購入いただいた結果、私と同じように、SQL Anywhereに関する明確で正確な記述を求めている方がとても多かったことを知り、心からうれしく思います。

今、私の本は日本語に翻訳され、新たな世界へと踏み出そうとしています。翻訳の苦労を乗り越えてでも出版する価値が本書にあると思っていただけたことは、とてもエキサイティングで、本当に光栄です。

本書（原書）の出版以来、最も好評なのは「第10章　チューニング」です。だれもがSQLを早く動かしたいと思っていて、この章がそのために書かれているからでしょう。要求レベルのロギングや実行プロファイルを使って遅いクエリを発見し、グラフィカルなプランを用いてそのクエリが遅い原因を見つける方法が書かれています。さらには、テーブルとインデックスの断片化やデータベースの再構築、インデックスの動作に関するTipやテクニックについても説明されています。

私のお気に入りは、本書で2番目に長い「第9章　保護」ですが、執筆に苦労したことを思い出します。「それは知らなかった、そうだったのか」と何度も驚きながら、難しい内容の解説に挑みました。データベース・オプション、トランザクション制御、例外処理、セキュリティ、バックアップとリカバリなどです。特に、ロック、ブロック、デッドロックの部分は難儀しました。しかし、それでもこの章が一番好きなのは、使用方法がわかった今でも、正確に覚えきれないところがあって、頻繁に参照する章だからです。また、章の冒頭の文も気に入っています：「この章は、システムの障害、不正なユーザからのアクセス、システム処理が固まってしまうような事態、およびそれらの予防策に関係します。つまり、お使いのデータベースをさまざまな悪い事象から保護するあらゆる手段について説明します。」だからこそ、この章は重要であり、たとえ第10章を先に読んだとしても、この章に戻っていただければと思います。

これらとは別の章を好む読者もいらっしゃるでしょう。たとえば、Mobile Linkを初めて扱う方であれば、「第7章　同期」に多くの時間を割くことと思います。

あるいは、本書で最長の章「第3章　選択」が、最も人気の高い章になるかもしれません。SQLの中でSELECT文が最も重要で強力ですが、同時に、最も複雑でややこしくもあるからです。この章では、単純な「SELECT 1」から始まり、GROUP BY ROLLUPや再帰ユニオンまで、SELECT文の全貌に触れます。SELECT文が存在しなければ、SQLもRDBMSも本書も存在していなかったと言っても過言ではないでしょう。

他にも6つの章があり、計10章が本書に含まれています。きっと、読者の皆様にとって、おもしろく役に立つトピックがたくさんあることでしょう。この10章を自信をもってお勧めします。

Breck Carter
2006年2月　オンタリオ州ノブレトンにて

Breck Carterプロフィール

Breck Carterは1973年以来、リレーショナル・データベースに携わってきました。独立系コンサルタントとして、長年にわたってIMS DB/DC、DB2、Oracle、SQL Server、ASEなどを扱い、1995年よりSQL Anywhereを使用した企業のデータベースやアプリケーションの開発を行ってきました。1993年からは、SybaseおよびiAnywhere Solutionsを代表してSQL Anywhereニュースグループの質問に答えるTeam Sybaseのメンバとしても活躍しています。また、iAnywhere Solutions Customer Advisory Boardのメンバでもあり、iAnywhere Solutionsの今後の製品方針の策定に寄与しています。

監修者まえがき

本書は、SQLの書き方から始まり、ストアドプロシージャの作り方やチューニング方法、Mobile Link同期などを扱っています。SQLの説明は、入門レベルの内容から、複雑な動作を解き明かす実践的な詳細まで盛り込まれており、本書の特徴となっています。SQL AnywhereのSQLは、ストアドプロシージャの結果セットをテーブルのようにFROM句で使えるなど、とても柔軟であり、本書でもその良さが十分に解説されています。また、コード例が具体的に挙げられているので、ストアドプロシージャやチューニングの説明も理解しやすく、サンプルとして読者が利用できるようになっています。Mobile Link同期は、図が少ないのが残念ですが、丁寧で正確に綴られています。さらに、TipやNoteとして、著者のノウハウが数多く書かれており、本書を読めば、SQL Anywhereの能力を十分に引き出したシステム開発ができることでしょう。

原書はSQL Anywhere 9.0.0をベースにしていますが、現在（2006年2月）の最新バージョンは9.0.2です。マイナー・バージョンアップですので、大きな機能変更があったわけではありませんが、翻訳にあたり、必要な部分については注釈を加えました。

著者のBreck Carter氏は、iAnywhere Solutions, Inc.の社員ではありません。しかし、SQL Anywhereを使ったシステム構築のコンサルティング業務に長年携わっていて、SQL Anywhereの知識と経験の豊富な方です。また、ボランティア・コミュニティであるTeam Sybaseの一員でもあり、SQL Anywhereのニュースグループでは回答する側としても活躍しています。毎年夏に行われるトレーニングでは、彼が質問しようと手を挙げると、社員である講師がファーストネームで呼びかけ、議論が盛り上がるなど、社員からも一目置かれている存在です。

なお、SQL Anywhereの運用／管理に焦点を当てた『使える！ SQL Anywhere』（アイエニウェア・ソリューションズ株式会社 森脇大悟著、翔泳社刊）も出版されていますので、本書と共に、SQL Anywhereを使ったシステム開発／運用に役立てていただければ幸いです。

2006年2月　アイエニウェア・ソリューションズ株式会社

目次

著者まえがき ～日本語版刊行にあたって iii
監修者まえがき v
はじめに xiv
謝辞 xviii

Chapter 1 作成 1

1.1 概要 2
1.2 リレーショナル・データベースにおけるコッドの 12 のルール 4
1.3 テーブルの 5 つのタイプ 6
1.4 グローバル永続テーブル 8
1.5 データ型 10
1.5.1 文字列は文字列：BINARY、CHARACTER、LONG 10
1.5.2 日付と時刻 13
1.5.3 真数（exact decimal） 14
1.5.4 浮動小数点数 14
1.5.5 バイナリ整数 15
1.5.6 リテラル 16
1.6 カラム・プロパティ 18
1.7 計算カラム 19
1.8 DEFAULT 値 21
1.8.1 DEFAULT AUTOINCREMENT 22
1.8.2 DEFAULT GLOBAL AUTOINCREMENT 23
1.8.3 リテラル・デフォルト 24
1.8.4 特殊更新デフォルト 26
1.8.5 デフォルトとしての式 27
1.9 NULL プロパティ 28
1.10 カラム制約 28
1.10.1 NOT NULL 制約 29
1.10.2 カラム CHECK 制約 29

1.10.3 PRIMARY KEY カラム制約 30
1.10.4 外部キー・カラム制約 31
1.10.5 UNIQUE カラム制約 32
1.11 ユーザ定義データ型 33
1.12 空き領域 35
1.13 テーブル制約 35
1.13.1 テーブル CHECK 制約 36
1.13.2 PRIMARY KEY テーブル制約 36
1.13.3 FOREIGN KEY テーブル制約 37
1.13.4 UNIQUE テーブル制約 39
1.14 リモート・データ・アクセス 40
1.14.1 CREATE SERVER 41
1.14.2 CREATE EXTERNLOGIN 42
1.14.3 リモート・テーブルとプロキシ・テーブルの作成 42
1.14.4 CREATE EXISTING TABLE 44
1.15 テンポラリ・テーブル 46
1.15.1 グローバル・テンポラリ・テーブル 46
1.15.2 ローカル・テンポラリ・テーブル 48
1.16 設計の正規化 53
1.16.1 第 1 正規形 54
1.16.2 第 2 正規形 55
1.16.3 第 3 正規形 56
1.16.4 ボイスコッド正規形 57
1.16.5 第 4 正規形 58
1.16.6 第 5 正規形 59
1.17 この章のまとめ 61

Chapter 2 挿入 63

2.1 概要 64
2.2 INSERT 64
2.2.1 すべてのカラムへの INSERT 65
2.2.2 指定のカラムへの INSERT 67
2.2.3 select を使用したすべてのカラムへの INSERT 69
2.2.4 select を使用した指定のカラムへの INSERT 71
2.2.5 select を使用し、カラム名を自動的に設定する INSERT 72

2.3 LOAD TABLE 74
2.4 ISQL INPUT 84
2.5 この章のまとめ 89

Chapter 3 選択 91

3.1 概要 92
3.2 SELECT の論理的実行 92
3.3 FROM 句 105
3.4 JOIN 108
3.4.1 CROSS JOIN 108
3.4.2 INNER JOIN 109
3.4.3 LEFT OUTER JOIN 110
3.4.4 RIGHT OUTER JOIN 112
3.4.5 FULL OUTER JOIN 113
3.5 抽出テーブル 114
3.6 マルチテーブル・ジョイン 118
3.7 SELECT FROM プロシージャ・コール 125
3.8 ラテラル・プロシージャ・コール 130
3.9 SELECT リスト 132
3.10 式と演算子 134
3.10.1 IF 式と CASE 式 137
3.11 上位 15 位のスカラ組み込み関数 140
3.12 ブール式と WHERE 句 147
3.12.1 比較述部 150
3.12.2 EXISTS 述部 153
3.12.3 IN 述部 154
3.12.4 BETWEEN 述部 155
3.12.5 NULL テスト述部 156
3.12.6 LIKE 述部 157
3.12.7 トリガ述部 159
3.13 GROUP BY 句 160
3.14 集合関数呼び出し 163

3.15 GROUP BY ROLLUP 句 171
3.16 HAVING 句 175
3.17 ORDER BY 句 176
3.18 SELECT DISTINCT 178
3.19 FIRST と TOP 179
3.20 NUMBER(*) 181
3.21 INTO 句 183
3.22 UNION、EXCEPT、INTERSECT 184
3.23 CREATE VIEW 190
3.24 WITH 句 194
3.24.1 再帰ユニオン 195
3.25 UNLOAD TABLE と UNLOAD SELECT 203
3.26 ISQL OUTPUT 208
3.27 この章のまとめ 214

Chapter 4 更新 215

4.1 概要 216
4.2 単一のローの更新 216
4.3 複数のローの更新 218
4.4 集合型 UPDATE の実行ロジック 220
4.4.1 集合型 UPDATE 225
4.5 UPDATE WHERE CURRENT OF カーソル 228
4.6 この章のまとめ 232

Chapter 5 削除 233

5.1 概要 ... 234
5.2 単一のローの削除 ... 234
5.3 複数のローの削除 ... 236
5.4 集合型 DELETE の実行ロジック ... 237
5.4.1 集合型 DELETE ... 240
5.5 DELETE WHERE CURRENT OF カーソル ... 244
5.6 TRUNCATE TABLE ... 246
5.7 この章のまとめ ... 252

Chapter 6 フェッチ 253

6.1 概要 ... 254
6.2 カーソルの FETCH ループ ... 254
6.2.1 DECLARE CURSOR FOR select ... 258
6.2.2 DECLARE CURSOR USING select ... 264
6.2.3 DECLARE CURSOR FOR CALL ... 265
6.2.4 OPEN と CLOSE カーソル ... 267
6.2.5 FETCH カーソル ... 268
6.3 カーソルの FOR ループ ... 269
6.4 この章のまとめ ... 272

Chapter 7 同期 273

7.1 概要 ... 274
7.2 Mobile Link のしくみ ... 275
7.3 Mobile Link ASA クライアント ... 280
7.4 Mobile Link クライアントの設定 ... 281
7.4.1 CREATE PUBLICATION ... 282

7.4.2 CREATE SYNCHRONIZATION USER 283
7.4.3 CREATE SYNCHRONIZATION SUBSCRIPTION 285
7.4.4 DBMLSYNC コマンド・ライン 286
7.4.5 SYSSYNC とパブリケーション・デフォルト 291
7.5 Mobile Link サーバ 293
7.6 Mobile Link サーバの設定 294
7.6.1 接続レベルのスクリプト 295
7.6.2 セッション・レベルのスクリプト 296
7.6.3 テーブル・レベルのスクリプト 298
7.6.4 ロー・レベルのスクリプト 300
7.7 Mobile Link のシステム・テーブル 328
7.8 Mobile Link モニタ 333
7.9 この章のまとめ 335

Chapter 8 パッケージ 337

8.1 概要 338
8.2 BEGIN ブロック 338
8.3 例外ハンドラ 341
8.4 代入文 345
8.5 IF 文 346
8.6 CASE 文 347
8.7 WHILE ループ 349
8.8 EXECUTE IMMEDIATE 350
8.9 CREATE PROCEDURE、CALL、RETURN 354
8.10 CREATE FUNCTION 362
8.11 CREATE TRIGGER 367
8.12 CREATE EVENT 378
8.13 TRIGGER EVENT 388
8.14 CREATE VARIABLE 392
8.15 この章のまとめ 393

Chapter 9 保護 395

9.1 概要 ... 396
9.2 オプション ... 397
9.3 トランザクション ... 403
9.4 セーブポイントとサブトランザクション ... 413
9.5 エラー処理 ... 418
9.5.1 SIGNAL と RESIGNAL ... 418
9.5.2 RAISERROR と CREATE MESSAGE ... 422
9.5.3 ROLLBACK TRIGGER ... 430
9.6 ロック ... 433
9.7 ブロックと独立性レベル ... 437
9.7.1 独立性レベル 0 ... 438
9.7.2 独立性レベル 1 ... 442
9.7.3 独立性レベル 2 ... 446
9.7.4 独立性レベル 3 ... 448
9.8 デッドロック ... 455
9.9 ミューテックス ... 460
9.10 GRANT CONNECT ... 462
9.10.1 テーブル権限 ... 467
9.10.2 ビュー権限 ... 470
9.10.3 実行権限 ... 472
9.10.4 特別権限 ... 473
9.10.5 GRANT GROUP ... 476
9.11 ロギングとリカバリ ... 480
9.12 バックアップ ... 485
9.12.1 フル・オフライン・イメージ・バックアップ ... 485
9.12.2 フル・オンライン・イメージ・バックアップ ... 488
9.12.3 差分オンライン・ログ・バックアップ ... 491
9.12.4 インクリメンタル・オンライン・ログ・バックアップ ... 492
9.12.5 ライブ・ログ・バックアップ ... 494
9.13 リストア ... 495
9.14 検証 ... 503
9.15 この章のまとめ ... 508

Chapter 10 チューニング 511

10.1 概要 512
10.2 要求レベルのロギング 513
10.3 インデックス・コンサルタント 524
10.4 プロファイリング 530
10.5 グラフィカルなプラン 532
10.6 断片化と再編成 538
10.6.1 ファイルの断片化 539
10.6.2 テーブルの断片化 541
10.6.3 テーブルの再編成 547
10.6.4 インデックスの断片化 549
10.6.5 インデックスの再編成 552
10.6.6 アンロード／再ロードによるデータベースの再構築 553
10.7 インデックスの作成 559
10.8 データベースのパフォーマンス・カウンタ 566
10.9 ヒントとテクニック 570
10.10 この章のまとめ 578

付属 CD-ROM について 579

収録内容 579
Developer Edition について 579
SQL Anywhere のインストール 579
特定製品に関するライセンス条件 580
サンプル・コードの使用に関するライセンス条件 584

索引 586

はじめに

SQL Anywhere がモバイル・データベース市場の 70% を超えるシェアを占めているのには、確かな理由があります。実際、使いやすい、管理が簡単、省リソースで動作するなど、多くの理由があり、これらはすべて数百あるいは数千のデータベースを配備するのに好都合です。また、何万ものルータやスイッチ、またはその他のハードウェア・デバイスにリレーショナル・データベースを組み込むのにも適しています。

しかし、SQL Anywhere は、モバイル・データベースや組み込みデータベースだけのものではありません。数百人ものアクティブ・ユーザが1つのデータベースを使用する過酷な環境のネットワーク・サーバとしても動作する、機能と柔軟性を備えています。さらに、1980 年代の Watcom に始まり、1990 年代の Powersoft と Sybase、そして 21 世紀の iAnywhere Solutions へと続く、揺るぎない基盤と長い歴史を誇っています。

この長い年月の中で、SQL Anywhere の開発チームは、常にある 1 つの道を歩んできました。VMI Communication and Learning Systems の Richard Williams は、これを「Watcom は、何事も行われるべき方法で行う」と表現しました。

SQL Anywhere Studio 9 の歴史の概要を、次に示します。

- **1965**：ウォータールー大学で最初に IBM 7040 用の WATFOR FORTRAN コンパイラを開発
- **1981**：Watcom の設立
- **1988**：Watcom Information Workbench の一部として PACEBase SQL Database System をリリース。これは、DOS を使用する PC 上で動作し、ISQL、トランザクション処理、宣言参照整合性、CREATE TABLE 文、ALTER TABLE 文、内部ジョイン、SELECT リスト内のサブクエリ、複数のユーザ ID、リカバリ用の個別のロールバック・ファイルを備えていた。これは「バージョン 1」で、バージョン 2 はリリースされなかった
- **1992**：Watcom SQL 3 の市販。Watcom SQL 3 は、ローレベル・ロック、個別のトランザクション・ログ、複数テーブルの更新を備えるマルチユーザ・サーバであった。以降のバージョンでは、Windows 3 のサポート、ODBC ドライバ、外部ジョイン、DEFAULT 値、CHECK 制約、TIME 型、DATE 型、LONG 型、オンライン・バックアップが追加された
- **1993**：Powersoft が Watcom SQL 3.2 を PowerBuilder 3 にバンドル
- **1994**：Powersoft が Watcom を買収
- **1994**：Watcom SQL 4 のリリース。ストアド・プロシージャ、トリガ、カスケード更新、カスケード削除、テンポラリ・テーブル、1 台のサーバにおける複数のデータベース、セルフチューニングのクエリ・オプティマイザ、TCP/IP サポートを備えていた
- **1995**：Sybase が Powersoft を買収

- **1995**：SQL Anywhere 5のリリース。SQL Remote、SQL Central（現在のSybase Central）、Transact SQL構文、Sybase Replication Serverのサポートを備えていた
- **1996**：SQL Anywhere 5.5のリリース。InfoMaker、NetImpact Dynamo（後のPowerDynamo）、Sybase Centralのプラグイン・アーキテクチャを含む製品にバンドルされてリリースされた
- **1998**：SQL Anywhere 6のリリース。エンジン自体には「Adaptive Server Anywhere（ASA）」という名前が付けられ、製品バンドルには「SQL Anywhere Studio」」という名前が付けられた。製品バンドルにはSQL Modeler（後にPowerDesignerと呼ばれる）が含まれていた。SQL Anywhere 6はデータベースにJavaを導入したバージョンであり、この後のリリースでは、プロキシ・テーブル、Ultra Light配備オプション、Mobile Link同期、対話型デバッガが追加された
- **2000**：SQL Anywhere 7のリリース。ASAデータベースをMobile Linkリモート・データベースとしてサポートし、Ultra Lightの向上、OLEDBドライバ、キャッシュの動的なサイズ変更の機能を備えていた。バージョン7では、ISQLおよびSybase CentralのJavaバージョンも導入された
- **2000**：iAnywhere Solutions, Inc.をSybaseの子会社として設立
- **2001**：SQL Anywhere 8のリリース。このバージョンでは数百もの強化が行われたが、その中心となるのはすべてクエリ・エンジンに関するものであった。パフォーマンスを向上させ、今後の強化をサポートするために、クエリ・エンジンが完全に作り直された
- **2003**：SQL Anywhere 9のリリース。パフォーマンスの向上、インデックス・コンサルタント、OLAのサポート、XMLのサポート、64ビット・プロセッサのサポートに加えて、再帰UNIONや、FROM句内でのストアド・プロシージャ・コールを含む多くの新しいSQL機能など、数百もの強化が成された

今後、コンピュータとハード・ドライブの価格が低下を続け、SQL Anywhereのパフォーマンスと機能が向上を続けるのに伴い、SQL Anywhereは、より大規模で高度なアプリケーションで使用されることが予想されます。本書は、このような展望と、データベース設計者やアプリケーション開発者が、SQL Anywhereの機能を知り、その特徴を最大限に活用することを望んでいるという信念に基づいています。

本書は10章に分かれており、各章のタイトルは、典型的なデータベースのライフサイクルの中で行われるタスクを大まかに表しています。

第1章　作成
第2章　挿入
第3章　選択
第4章　更新
第5章　削除
第6章　フェッチ
第7章　同期

第8章　パッケージ
第9章　保護
第10章　チューニング

「第1章 作成」は、SQL Anywhereのようなリレーショナル・データベース管理システムが実際にはどのようなものであるかを定義するコッド博士の12のルールで始まります。次に、SQL Anywhereが使用する5種類のテーブル（グローバル永続、グローバル・テンポラリ、ローカル・テンポラリ、リモート、プロキシ）について説明します。第1章では、データ型、カラム・プロパティ（DEFAULTなど）、CHECKやFOREIGN KEYなどのカラム制約とテーブル制約についても説明し、最後に正規化設計について説明します。

「第2章 挿入」では、データベースへのデータの挿入について説明します。具体的にはINSERTの5種類の形式に加えて、LOAD TABLE文と、ISQLのクライアント側の代替であるINPUT文について説明します。

「第3章 選択」は第2章とは逆の内容で、SELECT文、UNLOAD文、ISQL OUTPUT文によるデータベースからのデータの取得について説明します。この章は本書の中で最も長い章ですが、それは驚くべきことではありません。SQLのSELECT文は、あらゆるプログラミング言語の中で、最も強力かつ最も複雑な単一の構文構造だからです。

第3章では、最初にSELECT文について、つまりSELECT文の論理的な実行について説明します。ここでは、SQL AnywhereがSELECT文を実際にはどのように実行するかは考慮せずに、各種の句が全体としてどのように結合し、最終的な結果セットにどのように影響を及ぼすかについて説明します。第3章の残りの大部分では、章の先頭に提示し、説明したステップに従います。GROUP BY句で問題が生じたことがある場合、SELECT文が目的とする3つのローではなく、10,000ものローを返してきたことがある場合、またはWHERE句でNUMBER(*)を呼び出せない理由がわからない場合、ぜひこの章を参照してください。

「第4章 更新」では、UPDATE文について説明し、UPDATE文を使用して1つのテーブル内の1つのローまたは複数のロー、あるいは複数のテーブル内の複数のローを更新する方法について説明します。この章では、集合指向型のUPDATE文の論理的な実行と、WHERE CURRENT OF句を使用するカーソル指向型のUPDATE文についても説明します。

「第5章 削除」は第4章の構成と非常によく似ており、1つのローまたは複数のローの削除について説明します。集合指向型のDELETE文の論理的な実行と、WHERE CURRENT OF句を使用するカーソル指向型のDELETE文についても説明します。第5章には、TRUNCATE TABLE文の項もあり、DELETE文との比較（速い場合、速くない場合、およびその理由）について説明します。

「第6章 フェッチ」では、SELECTクエリとプロシージャ・コールを使用してカーソルに適用されるDECLARE文、OPEN文、FETCH文、CLOSE文について説明します。この章では、カーソルのFORループについても説明します。カーソルのFORループでは、すべてのローカル変数を宣言（DECLARE）したり、無限に続くFETCHリストをコーディングする必要がなくなるため、SQLにおけるカーソル・ループが非常に単純化されます。

「第7章 同期」はMobile Linkに関する章であり、Mobile LinkサーバがASA対応のMobile Linkクライアントと連携して、1つの統合データベースと多数のリモート・データベースの間で互いにデータを同期する方法について説明します。Mobile Linkスクリプトについては、レベル（接続、セッション、テーブル、ロー）と方向（アップロード、ダウンロード）で分類して説明します。また、サーバ側とクライアント側のエラー処理についても説明します。

「第8章 パッケージ」では、BEGINブロックと、BEGINブロックに基づくSQLモジュール（プロシージャ、関数、トリガ、イベント）を使用して、アプリケーション・ロジックをデータベースに移動する方法について説明します。また、SET代入文、フロー制御構造（IF、CASE、WHILE）、EXECUTE IMMEDIATE文、接続レベルのCREATE VARIABLE文について説明します。この章では、例外処理の基本についても紹介します（第9章で詳しく説明します）。

「第9章 保護」は、本書の中で2番目に長い章で、オプション、トランザクション、例外、エラー、サブトランザクション、ロック、ブロック、独立性レベル、デッドロック、ミューテックス、ユーザID、権限、ユーザ・グループ、ロギング、リカバリ、バックアップ、リストア、データベース検証など、多様なトピックについて説明します。これらのトピックの多くは、相互の結び付きはそれほど強くありませんが、すべてが「あらゆる破壊行為とその行為からの保護」に分類されます。

「第10章 チューニング」は、読者が最も多い章かもしれません。だれもがスピードに興味を持っています。この章は、最後に読むべき章でもあります。なぜなら、最適化の1番目のルールは、「最適化をしてはならない」だからです。読者が専門家の場合は、「まだ最適化してはならない」という最適化ルール2が当てはまります。

第10章では、最初に、SQL Anywhereが備えている、要求レベルのロギング、インデックス・コンサルタント、プロファイリング、グラフィカル・プランという4つの大きなパフォーマンス・チューニング・ツールについて説明します。次に、ファイル、テーブル、インデックスの断片化や、テーブル、インデックス、データベースの再編成など、難解なトピックについて説明します。CREATE INDEX文の構文と、データベース・パフォーマンス・カウンタに関する項もあります。本書の最後の項は、パフォーマンス・チューニングの「ヒントと方法」です。ここには、本書でこれまでヒントとして記載する適切な場所がなかった36項目（推奨事項と禁止事項）が示されています。

本書で説明するほとんどすべての内容に対して、サンプル・コードが用意されています。SQL Anywhere独自のデモンストレーション・データベース（ASADEMO.DB）と、再帰UNION機能に関する第3章の簡単な組織図への参照を除いて、大きなケース・スタディを扱ったり、実世界のアプリケーションの細部や複雑さを再現することはありません。ほとんどの例では、「t1」や「t2」といった人工的な名前の単純なテーブルを使用しています。サンプル・コードは、SQL Anywhereの構文とその意味に的を絞っています。行数は短く、内容は現在のトピックに限定し、そして数多くの例が用意されています。

付属CD-ROMには、本書のすべてのサンプル・コードを記載したHTMLファイルが収録されており、簡単に参照できます。また、CD-ROMは2枚あり、SQL Anywhere Studio 9.0.2のWindows版とLinux版それぞれのDeveloper Editionが収録されています。

謝辞

iAnywhere Solutions の Chris Kleisath 氏、Dave Neudoerffer 氏、Michael Paola 氏、Glenn Paulley 氏、Tom Slee 氏の、何年間にもわたる励ましに感謝します。また、筆者を Wordware Publishing に引き合わせ、プロジェクトの開始に尽力してくださった Steven McDowell 氏に感謝します。Wordware の Wes Beckwith 氏と Beth Kohler 氏の情熱と忍耐にも感謝の意を表します。

iAnywhere Solutions の公式な「技術チェック」プロセスを調整してくれた Ruth Haworth 氏と、実際の作業を行ってくれた Peter Bumbulis 氏、Russ Chamberlain 氏、Reg Domaratzki 氏、Bruce Hay 氏、Glenn Paulley 氏にも感謝します。特に Glenn Paulley 氏には、BNF の手ほどきを受け、あれやこれやと数週間、ときには数か月にわたって毎日のようにかけ続けた電話に耐えてもらいました。この方たちには、改めて賞賛の意を表します。

Hartmut Branz 氏、Kerry Ferguson 氏、Jonathan P. Groves 氏、Ruth Haworth 氏、Margaret Kammermayer 氏、Tom Slee 氏、Carol Stone 氏は、原稿をレビューし、価値あるフィードバックを返してくれました。

Tom Slee 氏は、ネックであった本書全体の構成に関して、リレーショナル・データベース・システムについて説明する場合は、「何よりも先に、すべてを紹介すべきだ」ということを指摘して、問題解決の糸口を与えてくれました。つまり、完璧な順番などないのだから、たとえ SELECT の構文が第 3 章まで登場しなくても、第 1 章に SELECT INTO #table_name を記載しても問題ないというのです。なぜなら、この文によってテーブルが作成され、第 1 章ではそれがすべてだからです。

iAnywhere Solutions の Tom Slee 氏と同僚たちが作成した SQL Anywhere のオンラインヘルプは非常に優れているため、ある意味で、本書の執筆は困難な作業でした。本書とオンラインヘルプの間の相違の多くは、単に表現が異なるというだけです。どちらがよいわけでも悪いわけでもなく、表現が異なるだけです。一例が、本書で構文を表すのに使用している BNF（Backus-Naur Form）表記法です（BNF については、第 1 章の概要で説明されています）。オンラインヘルプでは、構文を表すのに別の表記法が使用されています。このような違いがあることによって、それぞれが役に立ち、そしていずれか一方でわかりにくいことがあっても、他方、つまり本書またはオンラインヘルプを参照し直すことによって、答えを見つけることができるでしょう。

多くの仲間がアイディアを提供し、質問に答えてくれました。その名前をここに記します。Richard Biffl、Ivan Bowman、Mark Culp、Mahesh Dixit、Nick Elson、Dan Farrar、Greg Fenton、David Fishburn、Anil Goel、Jason Hinsperger、Graham Hurst、Chris Irie、Ian McHardy、Martin Neumann、Graeme Perrow、John Smirnios、Dave Wellstood、Matthew Young-Lai（敬称略、アルファベット順）。

競合会社の NASCAR Nextel Cup の Ken Schrader 氏にも感謝します。「second place」（準優勝）の同義語として、氏の言葉から「best loser」を借用しました。この単語は、第 3 章の「SELECT FROM プロシージャ・コール」におけるストアド・プロシージャ名の中で使用されています。

最後に、本書の執筆期間に限らず、これまで長い間にわたって筆者を支え、励ましてきてくれた妻の Bonnie に心から感謝します。

Chapter

1

作成

Section

1.1 概要

この章では、SQL Anywhere 9リレーショナル・データベースのライフサイクルの1番目のステップである、テーブルとカラムの作成について説明します。

リレーショナル・データベースはわかりやすいため、広く使用されています。また、優れた機能も備えています。同じような機能を持つデータベースやファイル・システムは他にもありますが、リレーショナル・データベースはその単純さゆえに、他のテクノロジより広く使用されています。

リレーショナル・データベースがわかりやすいのは、テーブルと呼ばれる非常に単純な構成要素で構築されているためです。テーブルは0個以上のローで構成され、各ローは1つ以上のデータ要素、つまりカラムで構成されます。ローの構成はすべて同じです。各々のローは、名前とデータ型が同じカラムが、同じ数だけ同じ順序で並びます。

Note

「カラムの順序は重要か？」という問題が、議論の対象になることがあります。ここでの説明の趣旨は、どのようなコンテキスト（たとえばSELECT * FROM t）であっても、またカラムの順序がどのようなものであっても、テーブル内のローはすべて同じということです。

この章では、まず、「リレーショナル・データベースとは何か」という質問の答えに相当する、12のルールを示します。そして、章の最後では、「優れたリレーショナル・データベースとは何か」という質問の答えに相当する、正規化について説明します。その間では、7種類の形式のCREATE TABLE文と、5種類のテーブルについて詳しく説明します。カラムのデータ型に加えて、各種のテーブルとカラムのプロパティや制約についても説明します。

本書では、BNF（Backus-Naur Form）表記法を使用してSQL構文を記述しています。この表記法は、単純なオブジェクトから複雑なオブジェクトを形成することに関する、簡潔で正確なルールの形式をとります。各ルールは生成規則と呼ばれ、左から右に向かって、1つの項目、「左辺は右辺のように定義される」を意味する特殊な演算子"::="、項目の展開定義で構成されます。

たとえば、次の生成規則は、「<in_dbspace_clause>項目は、INキーワードまたはOUTキーワードの後に、DEFAULTキーワードまたは<dbspace_name>項目が続くものとして定義される」ことを意味しています。

```
<in_dbspace_clause> ::= ( IN | ON ) ( DEFAULT | <dbspace_name> )
<dbspace_name>      ::= <identifier>
```

次に、2番目の生成規則で、<dbspace_name>項目は<identifier>に等しいことが定義されます。別

の場所には、<identifier> を説明する生成規則があります。

表 1-1 は、BNF 表記法で使用される記号を示します。

表 1-1　BNF 表記法

記号	説明
::=	「左辺は右辺のように定義される」を意味する
<name>	"::=" 生成規則で定義される項目
|	"or"（または）を意味する（例：<color> ::= RED | ORANGE | YELLOW）
{ items }	「0 個以上の項目の繰り返し」を意味する
(items)	「いずれかの項目を選択する」を意味する
[items]	「これらの項目はオプションである」を意味する
WORD	そのまま表示されるキーワード
","	そのまま表示される句読点や特殊記号（二重引用符は除く）
words	自由形式の説明。より簡潔にするために、BNF の規則を曲げている
-- words	コメント

本書の BNF は、オンラインヘルプの表記法とは異なります。これは意図的に行われています。あるコマンドについて、オンラインヘルプの内容が理解できなくても、BNF では理解できるかもしれません。逆についても同様です。BNF が紛らわしい場合でも、オンラインヘルプが参考になるかもしれません。ただし、両者の相違には注意してください。たとえば、本書の中カッコ ⦃ は「0 個以上の繰り返し」を意味しますが、オンラインヘルプの中カッコ ⦃ は「いずれかの項目を選択する」を意味します。

BNF は、最初、読みにくいかもしれません。SQL のきわめて一般的な構成である、カンマで区切られたリストの例を次に示します。

```
<user_name_list> ::= <identifier> { "," <identifier> }
```

この生成規則を厳密に解釈すると、<user_name_list> は、<identifier> の後に、カンマと <identifier> の組み合わせが 0 個以上続くものとして定義されます。慣れてくると、これが単に「カンマで区切られた識別子のリスト」であることがわかります。

付属 CD-ROM では、本書で使用されているすべての BNF が HTML 形式で表示され、すべての <item> とその定義の間はハイパーテキスト・リンクで結ばれています。

Section

1.2 リレーショナル・データベースにおけるコッドの12のルール

1970年代にコッド博士（E. F. Codd）が最初に提唱したデータのリレーショナル・モデルでは、「テーブル」は「リレーション」を表す語句です。「リレーション」は、リレーショナル・データベースの理論上の基礎について説明するときに使用され、「テーブル」は、SQL Anywhere 9などの実際の実装について説明するときに使用されます。

コッド博士は、何が本当のリレーショナル・データベースで、何が本当のリレーショナル・データベースでないかについて、厳密なルールを定めました。

ルール0 リレーショナル・データベースは、ルール1～12で定義される機能を通じて、すべてのデータを管理します。
パフォーマンスの向上など、いかなる理由があっても、これらのルールを逸脱してはなりません。つまり、特殊な例外について心配することなくルールを信頼できます。

ルール1 テーブルを使用して、データベース内のすべてのデータを論理的に表します。
すべては、テーブルにあります。別のレイアウトや配置について理解する必要はありません。可変形式のレコード、配列、ポインタ、ヒープ、スタック、またはリストは使用しません。

ルール2 テーブル名、カラム名、プライマリ・キー値を使用して、すべてのデータ要素をユニークに識別できます。
これは、どのテーブルにもプライマリ・キーが必要であることを意味します。プライマリ・キーは、null以外の値を持つ1つ以上のカラムから構成され、各ローにおけるそれらの値の組み合わせがテーブル内でユニークになります。SQL Anywhere 9では、プライマリ・キーを使用せずにテーブルを作成できますが、パフォーマンスなど多くの理由により、これは不適切な考え方だと言えるでしょう。ルール2に従うかどうかはユーザしだいです。また、ビジネス関連のデータに基づくプライマリ・キーをユーザが指定できない場合、または指定しない場合、SQL Anywhere 9では機械的にプライマリ・キーを定義することもできます。

ルール3 不足しているデータや適用できないデータは、null値を使用して系統的に表します。null値はデータ型に依存せず、空の文字列、ブランク文字列、0など、他の値と区別されます。プライマリ・キーと他のカラムに対して、「nullを使用できない」ことを指定できる必要があります。

ルール4 テーブルを使用して、データベースの説明やシステム・カタログを論理的に表します。認証されたユーザは、通常のテーブルと同じ方法でシステム・カタログを問い合わせる

ことができます。
つまり、システム・カタログはデータベース内のデータとして格納されており、かつルール 1 に従います。

ルール 5 データ定義、データ操作、整合性制約、ユーザ認証、トランザクション制御操作をすべてサポートする、少なくとも 1 つのプログラミング言語が必要です。
現在、その言語は SQL です。SQL は、CREATE、UPDATE、GRANT、COMMIT など、各種のコマンドをすべて備えています。

ルール 6 すべてのビューは、理論上更新可能の場合は、更新できる必要があります。
このルールは、更新に加えて、挿入と削除に適用されます。ビューは、1 つ以上のテーブルを含むクエリです。要求された効果をビューで実現するために、どのベース・テーブル内のどのローを更新する必要があるか判断できる場合、ビューは理論上更新可能です。たとえば、「SELECT * FROM t1」として定義されているビューでは、ベース・テーブル t1 に関するすべての情報をビューから参照できるため、ビューは更新可能です。これに対して、SELECT 1 として定義されているビューでは、テーブルさえも参照していないため更新可能ではありません。

ルール 7 集合指向の挿入、更新、削除の操作を実行できます。
多くの場合、集合指向の操作は、ロー単位の操作より、書き込みと実行が効率よく行われます。

ルール 8 アプリケーション・プログラムは、物理的なデータ表現の変更の影響を受けてはなりません。
アプリケーション・プログラムは、基になる物理的なレイアウトやアクセス方法ではなく、SQL によって提供される論理ビューのみを処理する必要があります。特に、アプリケーション・プログラムは、テーブル内のローの物理的な順序や、ロー内のカラムの順序を意識すべきではありません。これは、ファイル名、ロケーション、レコード・レイアウト、ソート順が変更されたときに、プログラムの変更が要求されるような他のテクノロジに勝る大きな利点です。

ルール 9 アプリケーション・プログラムは、情報を維持する論理的なデータ設計の変更の影響を必ずしも受けません。
たとえば、元のデータを維持するようにテーブルを分割または結合する場合、アプリケーション・プログラムのロジックを変更せずにこれを実現できる必要があります。これは、アプリケーションを再作成せずにデータベース設計を拡張または修復できることをしばしば意味します。

ルール 10 整合性制約定義は、アプリケーション・プログラムではなく、システム・カタログ・テーブルに保存します。まず、テーブルごとにエンティティの整合性を満たす必要がありま

す――つまり、プライマリ・キーのコンポーネントがnullになることはありません。また、すべての外部キー関係で、参照整合性を満たす必要があります――つまり、null でない外部キー値ごとに、それに対応するプライマリ・キー値がなければなりません。
整合性制約は、データベースに組み込まれ、回避できないため、長期にわたってデータの一貫性を確保します。短期的な観点においても、データの整合性に違反するアプリケーション・プログラミング・エラーを検出できるというメリットがあります。

ルール 11 分散データをサポートする場合は、データが分散されてもアプリケーション・プログラムが影響を受けてはなりません。
たとえば、テーブルが同じロケーションにあっても異なるロケーションにあっても、複数のテーブルに関係するプログラムを同じように作成できる必要があります。SQL Anywhere 9 では、プロキシ・テーブルを使用して分散データをサポートし、このルールを満たしています。

ルール 12 データの操作時に整合性ルールや整合性制約を回避できてはなりません。
集合指向の操作またはロー単位の操作のいずれを実行する場合にも同じルールが適用され、このルールを破る低レベルのアクセス方法があってはなりません。ただし、実際には、アプリケーション設計の問題を解決するために、いくつかのルール・チェックを一時的に延期することはあります。

Section

1.3 テーブルの5つのタイプ

CREATE TABLE 文には7種類の形式があり、5つの異なるタイプのテーブルを作成します。文の形式は異なりますが、新しいテーブルの名前とレイアウトを定義する、各ロー内のカラムの名前とデータ型を定義する、テーブルとカラムに適用される制約を定義するという基本的な目的は同じです。テーブルのタイプは、テーブルが異なる接続で使用できるかどうか（グローバル対ローカル）、スキーマとデータが永続的（パーマネント）か一時的（テンポラリ）かに加えて、データが位置する場所（リモート対プロキシ）によって異なります。

```
<create_table> ::= <create_global_permanent_table>
                 | <create_remote_and_proxy_table>
                 | <create_proxy_table>
                 | <create_global_temporary_table>
                 | <create_local_temporary_table>
                 | <declare_local_temporary_table>
                 | <select_into_local_temporary_table>
```

最も一般的なタイプはグローバル永続テーブルです。このテーブルは現在のデータベース内に存在し、スキーマとデータの両方とも、適切なパーミッションを持つすべての接続でグローバルに使用できます。スキーマはテーブルが明示的に削除されるまで永続的であり、データはデータが明示的に削除されるまで永続的です。

Note

「グローバル永続テーブル」は本書独自の用語です。オンラインヘルプや他のソースでは、これを単に「テーブル」、場合によっては「ベース・テーブル」と呼んでいます。

リモート・テーブルは、現在のデータベース以外のデータベースやファイル内に存在し、プロキシ・テーブルを通じて現在のデータベースでのみ参照できます。最も厳密な意味では、リモート・テーブルは実際にはテーブルである必要はありません。プロキシ・テーブル・インタフェースに関する限り、単に外観がテーブルのようであればかまいません。リモート・テーブルは、スプレッドシート、テキスト・ファイル、場合によっては他のテーブルやビューを含むビューでもかまいません。その他の点では、リモート・テーブルは、スキーマと基になるデータが永続的だという点でグローバル永続テーブルに似ています。

プロキシ・テーブルは、一度に2つの場所に存在します。スキーマは現在のデータベースに存在しますが、実際のデータは、別のデータベースやファイル内のリモート・テーブルなど、別の場所にあります。「プロキシ」という用語は、「代理」または「代用」を意味します。各プロキシ・テーブルは、リモート・テーブル内のデータと、現在のデータベース内のスキーマの間のマッピングです。

プロキシ・テーブルは、リモート・テーブルがグローバル永続テーブルであるかのように扱うためのメカニズムで、すべての接続に対して有効であり、スキーマとデータは永続的です。レガシー・データベース、分散データベース、スプレッドシート、フラット・ファイルなど、他のソースからデータを収集するために使用されることがあります。また、問い合わせだけでなく更新も可能であり、変更が他のデータ・ソースに反映されます。

グローバル・テンポラリ・テーブルは、現在のデータベース内に存在します。スキーマは永続的であり、すべての接続に対して有効ですが、データは一時的です。各接続は、その接続でのみ参照できるデータを挿入し、接続が切断されるとそのデータは自動的に削除されます。データは、COMMIT または ROLLBACK の実行のたびにも削除されます。

ローカル・テンポラリ・テーブルは、現在のデータベース内に存在します。スキーマとデータは両方とも一時的です。各接続がテーブルを作成し、データを挿入する必要があります。テーブルとデータは、その接続でのみ参照できます。接続が切断された場合、またはテーブル名がスコープから外れた場合、テーブルは自動的に削除されます。データは、COMMIT または ROLLBACK の実行のたびに自動的に削除されます。それ以外の場合は、テーブルが削除されるまで継続します。テンポラリ・テーブルを作成する文は3種類あり、その中の1つはテーブルを作成すると同時にデータを挿入します。

Section

1.4 グローバル永続テーブル

グローバル永続テーブルのCREATE TABLE文は、テーブル名、オプションの所有者名（現在接続しているユーザと異なる場合）、テーブル要素のリストに加えて、物理DBSPACEのオプション名、つまりデータが位置する場所を指定します。

```
<create_global_permanent_table> ::= CREATE TABLE
                                    [ <owner_name> "."] <permanent_table_name>
                                    <table_element_list>
                                    [ <in_dbspace_clause> ]
<owner_name>             ::= <identifier>
<identifier>             ::= <alphabetic> { ( <alphabetic> | <numeric> ) }
                           | 二重引用符または角カッコで囲まれた予約語
                           | 二重引用符または角カッコで囲まれた任意の文字列
<alphabetic>             ::= 英字、下線（_）、アットマーク（@）、ドル記号（$）、またはシャープ記号（#）
<numeric>                ::= 0～9の数字
<permanent_table_name>   ::= <non_temporary_identifier>
<non_temporary_identifier> ::= <non_temporary_alphabetic>
                                 { ( <alphabetic> | <numeric> ) }
                             | 二重引用符または角カッコで囲まれた予約語
                             | 二重引用符または角カッコで囲まれた任意の文字列
<non_temporary_alphabetic> ::= 英字、下線（_）、アットマーク（@）、またはドル記号（$）
<table_element_list>       ::= "(" <table_element> { "," <table_element> } ")"
<in_dbspace_clause>      ::= ( IN | ON ) ( DEFAULT | <dbspace_name> )
<dbspace_name>           ::= <identifier>
```

グローバル永続テーブルの名前は、所有者名とテーブル名の2つから成ります。ほとんどのSQL Anywhereデータベースでは、テーブル名のセットは1つだけで、DBAが所有者です。しかし、テーブル名が同じで所有者は異なる別のテーブルを作成することができます。他の文は、該当の所有者名を明示的に指定することによって、このようなテーブルを参照できます。これによって、各ユーザが自身のテーブル・セットを「所有」し、同じテーブルの異なるバージョンが同時に存在できるようになります。

CREATE TABLEは、デフォルトで現在のユーザを所有者として使用します。異なる所有者名を明示的に指定することも可能ですが、実際にそうすることはまれです。テーブルの所有権の詳細については、「9.10.1 テーブル権限」および「9.10.5 GRANT GROUP」を参照してください。

次に、単純なCREATE TABLE文の例を示します。この文は、3つのカラムから成るグローバル永続テーブルt1を作成します。

```
CREATE TABLE t1 (
   key_1          INTEGER NOT NULL PRIMARY KEY,
   non_key_1      VARCHAR ( 100 ) NOT NULL,
   non_key_2      VARCHAR ( 100 ) NOT NULL );
```

所有者名、テーブル名、DBSPACE名はすべて、標準のSQL Anywhere識別子です。「識別子」は128文字以下の文字列で、場合によって二重引用符（" "）または角カッコ（[ ]）で囲みます。文字列がSQL Anywhereの予約語の場合は、引用符またはカッコが必要です。先頭が英字で、その後に英数字が続くような文字列でない場合にも、引用符またはカッコが必要です。なお、英字には、アンダースコア（_）、アットマーク（@）、シャープ記号（#）などの特殊文字が含まれます。

先頭がシャープ記号（#）のテーブル名は特別なケースで、「1.15.2 ローカル・テンポラリ・テーブル」で説明します。グローバル永続テーブルの名前は、CREATE TABLE文の中で二重引用符または角カッコで囲まない限り、シャープ記号で始めることはできません。

dbspace句は、このテーブルのデータを保持する、最大で13個の物理ファイルのうちの1つを指定します。初めて作成するデータベースのdbspaceファイルは1つで、作成時に指定したファイル名を持ち、dbspace名はSYSTEMです。新しいテーブルは、デフォルトでこのファイルを使用します。CREATE DBSPACE文を使用すると、これ以外に12個のファイルを作成し、dbspace名を指定できます。オペレーティング・システムによっては、複数のファイルを使用することで、記憶領域の容量を増やしたり、あるいは異なるデータ・セットを異なる物理ディスク・ドライブに配置してパフォーマンスを向上させたりすることができます。dbspaceについては「第10章 チューニング」で説明します。CREATE DBSPACE文はオンラインヘルプで説明されています。

カンマで区切られたテーブル要素のリスト（<table_element_list>）は、1つ以上のカラム定義と、0個以上のテーブル・プロパティおよびテーブル制約で構成されます。各カラム定義は、カラム名、データ型、オプションのカラム・プロパティで構成されます。カラム名はSQL Anywhereの識別子であり、データ型は、組み込みのデータ型、またはユーザ定義のデータ型への参照です。

```
<table_element>     ::= <column_definition>
                      | <table_property>
                      | <table_constraint>
<column_definition> ::= <column_name> <data_type> [ <column_definition_list> ]
<column_definition_list> ::= <column_definition_term> { <column_definition_term> }
<column_definition_term> ::= <column_property>
                           | <column_constraint>
<column_name>       ::= <identifier>
<data_type>         ::= <builtin_data_type>
                      | <user_defined_data_type>
                      | <builtin_user_defined_data_type>
```

Section

1.5 データ型

SQL Anywhere には、文字列、日時、数値を表すさまざまなデータ型が組み込まれています。データ型には、4つの重要なプロパティがあります。

- データベース内での物理的な格納方法
- データベース・サーバが実行する SQL 操作による処理方法
- クライアント・アプリケーションから入力として値を受け取る方法
- クライアント・アプリケーションへの出力として値を表示する方法

この項では、先頭の2つのプロパティ、つまり、物理記憶領域と操作規則について特に説明します。入力と出力については、後の章で説明します。

```
<builtin_data_type> ::= <string_type>
                      | <date_time_type>
                      | <exact_numeric_type>
                      | <float_numeric_type>
                      | <integer_type>
```

リレーショナル・データベースのルールが示すように、アプリケーション・プログラマにとっては、データ型の物理的な記憶方法の詳細は、操作規則、入力、出力の3点に比べればあまり重要ではありません。特に、ここではルール0とルール12が当てはまります。すなわち、リレーショナル・データベースがすべてのデータを管理するのであり、整合性や制約を破るようなデータベース操作ができてはならないということです。

1.5.1 文字列は文字列：BINARY、CHARACTER、LONG

すべての文字カラムとバイナリ・カラムは、宣言の仕方にかかわらず、可変長の文字列として格納されます。最大長（<maximum_length>）は、文字列のデータ部分のバイト・サイズの制限値を指定し、デフォルト値は1バイトです。LONG VARCHAR 型および LONG BINARY 型の暗黙の最大長は 2GB です。

```
<string_type>    ::= <char_type> [ "(" <maximum_length> ")" ]
                   | LONG BINARY
                   | LONG VARCHAR
<char_type>      ::= BINARY
                   | CHAR [ VARYING ]
```

```
                    | CHARACTER [ VARYING ]
                    | VARBINARY
                    | VARCHAR
<maximum_length> ::= 1～32767の範囲の整数リテラル
```

Tip

LONG VARCHARとLONG BINARYを含むこれらすべてのデータ型は、テーブル内のカラムに加えて、ストアド・プロシージャ内のローカル変数とグローバル変数でも使用できます。

必要な記憶領域は、最大長ではなく、各カラムの値の現在の長さに依存します。長い文字列は分割されて格納されるので、短い文字列よりオーバヘッドがかかります。一方、短い文字列は、LONG VARCHARとして宣言されていても、効率的に格納されます。動作の仕組みは次のとおりです。長さが254バイトまでの文字列値は、ロー内の他のカラムと常に一緒に格納されます。長さが255バイト以上になると、値は2つの部分に分けられます。最初の部分の長さは254バイトで、元の位置にそのまま格納されます。一方、残りはblob継続と呼ばれ、拡張ページと呼ばれる1つ以上の別ページに配置されます。これらの拡張ページが個別に保持されるのは、long値を参照する必要がないクエリまたは逐次スキャンが、すべてのページを検索しなくて済むためです。この配置の詳細については、「10.6.2 テーブルの断片化」を参照してください。

SQLプログラミングの観点からすると、SQL Anywhere 9では文字列はあくまでも文字列であり、宣言されたデータ型を意識する必要はありません。たとえば、すべての会社名が30文字以内【監注1】に収まるとは思うが、例外が心配、という場合があるかもしれません。しかし、CHARACTER (100) またはCHARACTER (1000) のいずれを使用しても、パフォーマンスが低下することはありません。同様に、通常は数百文字程度が必要な説明用のカラムをLONG VARCHARとして宣言して、特殊な場合に対処することができます。この場合、実際に非常に長い値を格納しない限り、データベースのサイズが増大することはありません。

CHARACTERカラムまたはBINARYカラムに、まったく同じデータが格納されることがあります。特に、CHARACTERカラムに格納された0バイト（16進数の00）は、文字列ターミネータではなく、データとして扱われます。

Tip

プログラミング環境によっては、0バイトの文字列ターミネータを"null"と呼びます。これは、SQL Anywhere 9で実装されるデータベースのNULL値と同じものではありません。データベースのNULLは、アプリケーション内で使用するときに特殊な処理が必要です。

【監注1】文字列型のサイズ指定はバイト数なので、正確には30バイトを意味する。日本語はマルチバイトとして扱われる。たとえば、文字コードセットが932JPN（Shift_JIS）やEUC_JAPANであれば、日本語文字は2バイトとなる。UTF-8であれば、3バイト。

「文字列は文字列」という前提に対して、いくつかの例外があります。第1に、BINARY カラムに関するソートと比較では、常に実際のバイナリ値を使用するのに対して、CHARACTER カラムは、データベースの照合順と大文字・小文字の区別に基づいてソートおよび比較されます。たとえば、大文字と小文字を区別しない（デフォルト）データベースでは、CHARACTER 値の 'a' と 'A' は等しいものとして扱われるのに対して、BINARY の 'a' と 'A' を比較またはソートする場合は、BINARY の 'a' の方が BINARY の 'A' より小さいものとして扱われます。

Tip

大文字と小文字を区別しないデータベースで、大文字と小文字を区別する比較を実行する必要がある場合は、次に示すように、CAST 関数を使用します。

```
IF CAST ( char1 AS BINARY ) = CAST ( char2 AS BINARY )
```

これは、WHERE 句と ORDER BY 句でも動作し、データベースの照合順を無視する必要がある場合に使用できます。

Note

本書では、原則としてマルチバイト文字セットについては言及しません。そのため、上記のヒントのように、シングルバイト文字セットのみを対象とする技法（マルチバイト文字セットには適用できない技法）を説明する場合があります。

第2に、一部の関数は、文字列引数の先頭の 255 バイトに対してのみ動作します。SOUNDEX、SIMILAR、およびすべての日時関数は、255 バイトを超える部分を無視します。

第3に、文字列から数値への変換では、やはり 255 バイトを超えるデータが無視されます。

第4に、長い文字列を、それより短い最大長が宣言されているカラムに代入しようとすると、右側がトランケート（切り詰め）されます。INSERT コマンドまたは UPDATE コマンドによってブランクだけがトランケートされる場合、またはブランク以外のデータがトランケートされるが、STRING_RTRUNCATION オプションがデフォルトの 'OFF' に設定されている場合は、メッセージが何も表示されません。ブランク以外のデータがトランケートされる場合にエラー・メッセージを生成するには、STRING_RTRUNCATION を 'ON' に設定する必要があります。後続ブランクは重要であり、宣言された最大長に収まらない限り、トランケートされることはありません。

Tip

先行ブランクと後続ブランクを取り除くには、LTRIM、RTRIM、TRIM の各関数を使用します。

第5に、アプリケーション開発ツールによっては、文字列の宣言方法に応じて、異なるコードとユーザ・

インタフェース要素を生成します。CHARとして宣言されているカラムが、固定長文字列として扱われる場合があります。ただし、SQL Anywhere 9では、そのような実装はしていません。

Note

データベース製品によってはCHARカラムを固定長文字列として実装していることがあるため、プロキシ・テーブルまたはMobile Link同期を通じて相互にデータを送信すると、影響を受ける可能性があります。

最後に、必要以上に大きな最大長で文字列カラムを宣言すると、パフォーマンスに影響が出ることもあります。インデックス内の宣言されたカラム幅によって、ハッシュBツリー・インデックスの代わりに圧縮Bツリー・インデックスが使用されます。また、カラムと引数に宣言された最大長の合計が非常に大きい場合は、サブクエリと関数のキャッシュがオフになることがあります。また、クエリ・オプティマイザがカラム統計からより適切な情報を取得できる可能性があるのは、カラムの最大サイズが8バイト未満に宣言されている場合です。これらのトピックのいくつかについては、「第10章 チューニング」で説明します。

それ以外の場合、文字列はあくまでも文字列です。異なる文字列型が宣言されていても、その値を何の問題もなく格納、コピー、代入できます。

1.5.2 日付と時刻

日付と時刻の値は、多くの異なる外部フォーマットで入力および表示できますが、常に、3種類の固定物理フォーマットの1つで格納されます。DATEデータ型では、年、月、日を数値として格納するのに4バイトが必要です。TIMEデータ型では、時、分、秒、秒の小数部（小数点第6位まで）を数値として格納するのに8バイトが必要です。TIMESTAMPデータ型も、年、月、日、時、分、秒、秒以下を数値として格納するのに8バイトが必要です。

```
<date_time_type> ::= DATE
                   | TIME
                   | TIMESTAMP
```

TIMEデータ型は00:00:00.000000～23:59:59.999999の範囲をサポートし、DATE型は0001-01-01～9999-12-31の範囲をサポートします。TIMESTAMPデータ型は、1600-02-28 23:59:59～7911-01-01 00:00:00の範囲を完全にサポートします。この範囲を外れた場合、日付部分は1～9999の年をサポートしますが、時間部分は完全でない可能性があります。

一部のプラットフォームでは、CURRENT TIMEとCURRENT TIMESTAMPによって返される秒の部分において、小数点の右側の有効桁数は2または3だけです。ただし、TIMEカラムとTIMESTAMPカラムは、常に6桁すべてを格納できます。

Tip

小数点以下の６桁を、値が０の場合も含めて、データベースからすべて取得するには、「SET OPTION PUBLIC.TIME_FORMAT = 'HH:NN:SS.SSSSSS'」と「SET OPTION PUBLIC.TIMESTAMP_FORMAT = 'YYYY MM DD HH:NN:SS.SSSSSS'」を使用します。デフォルトは３桁で、この場合、それより多くの桁を記録したとしても、その部分はアプリケーションから取得できません。

1.5.3 真数（exact decimal）

真数（exact decimal）は、算術式の中で使用される場合に正確な結果を導くものであり、財務計算に適しています。

DECIMAL データ型と NUMERIC データ型は同じものであり、真数（exact decimal）の長さ（精度）は最大 127 桁です。小数点の右側の桁数（位取り）は、0 ～すべての桁数の範囲です。

```
<exact_numeric_type> ::= <exact_keyword>
                           [ "(" <exact_precision> [ "," <exact_scale> ] ")" ]

<exact_keyword>      ::= DEC
                       | DECIMAL
                       | NUMERIC
<maximum_length> ::= 1 ～ 127 の範囲の整数リテラル
<exact_scale>    ::= 1 ～ <exact_precision> の範囲の整数リテラル
```

精度と位取りを両方とも省略した場合、デフォルトは（30,6）です。精度を指定し、位取りを省略した場合、位取りのデフォルト値は 0 です。精度と位取りを両方とも指定する場合、位取りは 0 以上精度以下でなければなりません。つまり、精度の実際の桁数を超えて小数点を移動させることはできません。

必要な記憶領域は、宣言された精度と位取りではなく、実際に格納される値の精度と位取りに依存します。たとえば、123456789 は 123 より多くの記憶領域を使用し、1.123456789 は 1.1 より多くのディスク容量を使用します。必要なバイトを求める計算式は、次のとおりです。

```
2 + TRUNCNUM ( ( p - s + 1 ) / 2, 0 ) + TRUNCNUM ( ( s + 1 ) / 2, 0 )
```

p および s は、先行ゼロと後続ゼロが無視された後の、実際の精度と位取りです。

1.5.4 浮動小数点数

浮動小数点数は、真数（exact decimal）よりその範囲がはるかに広いため、値の大小にかかわらず最小精度が要求される科学計算で役立ちます。ただし、丸め誤差が生じるため、正確さが要求される計算には適していません。特に、浮動小数点数を使用すると、同じ結果が生じるはずの２つの異なる計算

が、実際にはわずかに異なる解を生じることがあります。

REAL データ型、DOUBLE データ型、FLOAT データ型は、単精度と倍精度の概数浮動小数点バイナリ値として実装されています。

```
<float_numeric_type> ::= REAL
                       | DOUBLE [ PRECISION ]
                       | FLOAT "(" <single_precision> ")"
                       | FLOAT "(" <double_precision> ")"
                       | FLOAT
<single_precision>   ::= 1 ～ 24 の範囲の整数リテラル
<double_precision>   ::= 25 ～ 53 の範囲の整数リテラル
```

REAL は、FLOAT (24) とまったく同じ方法で格納されます。FLOAT (1) ～ FLOAT (23) のすべての宣言についても同様です。24 桁のバイナリの精度を持つ単精度の値は、6 桁の 10 進数にほぼ相当します。REAL 値の記憶領域は常に 4 バイトで、絶対値で 1.175495e-3 ～ 3.402823e+38 の範囲の正の値と負の値を表すことができます。6 桁の 10 進数を超える値を表すこともできますが、6 桁以降については丸め誤差が生じます。

DOUBLE は FLOAT (53) と同じで、FLOAT (25) ～ FLOAT (52) についても同様です。倍精度値は、53 ビットの精度を持ち、15 桁の 10 進数に相当します。DOUBLE の記憶領域は常に 8 バイトで、絶対値で 2.22507385850721e-308 ～ 1.79769313486231e+308 の範囲の正の値と負の値を表すことができます。15 桁以降については丸め誤差が生じます。

精度を指定せずに FLOAT を指定すると、REAL と見なされます。FLOAT_AS_DOUBLE オプションを 'ON' に設定すると、REAL ではなく DOUBLE と見なされます。

1.5.5 バイナリ整数

バイナリ整数は、計算効率が重要な場合によく使用されます。バイナリ・コンピュータは、2 バイトおよび 4 バイトのバイナリ整数に関する計算を、他のデータ型より格段に高速に実行できます。

SQL Anywhere は、1、8、16、32、64 桁のバイナリから成る符号なしバイナリ整数と、15、31、63 桁と符号ビットから成る符号付きの値を利用できます。これらは真数データ型で、実用上は DECIMAL (p, 0) 値に似ています。

```
<integer_type> ::= BIT
                 | [ UNSIGNED ] TINYINT
                 | [ UNSIGNED ] SMALLINT
                 | [ UNSIGNED ] ( INT | INTEGER )
                 | [ UNSIGNED ] BIGINT
```

BIT データ型はブール・データを表すのに役立ち、0 は FALSE、1 は TRUE をそれぞれ表します。各値に対して物理記憶領域のすべてのバイトが使用されますが、使用できる値は 0 と 1 だけです。

デフォルトでは、BIT カラムは NOT NULL として作成されますが、明示的に NULL として作成することもできます。これは、CREATE TABLE のデフォルトが NULL で、NOT NULL は明示的に指定する必要がある他のデータ型と異なります。

TINYINT データ型と UNSIGNED TINYINT データ型は同じです。0 ～ 255 の範囲の符号なしバイナリ整数で、記憶領域は 1 バイトです。このデータ型は、小さな「コード」テーブルのプライマリ・キーなど、非常に小さな正の値に適しています。

SMALLINT データ型の記憶領域は 2 バイトで、-32,768 ～ 32,767 の範囲の値を表します。

UNSIGNED SMALLINT 値の記憶領域も 2 バイトですが、値の範囲は 0 ～ 65,535 です。この値は、SMALLINT の正の範囲を 2 倍にした値です。

INTEGER データ型の記憶領域は 4 バイトで、-2,147,483,648 ～ 2,147,483,647 の範囲の値を表します。このデータ型は、大きなテーブルの DEFAULT AUTOINCREMENT プライマリ・キーでよく使用されます。

UNSIGNED INTEGER 値の記憶領域も 4 バイトですが、値の範囲は 0 ～ 4,294,967,295 です。これは、大きな正のプライマリ・キーに非常に適しています。

BIGINT データ型は、最も大きなバイナリ整数です。記憶領域は 8 バイトで、-9,223,372,036,854,775,808 ～ 9,223,372,036,854,775,807 の範囲の値を表します。多数の大きなパーティションに分ける余裕があるため、DEFAULT GLOBAL AUTOINCREMENT を指定したプライマリ・キーに適しています。

UNSIGNED BIGINT 値の記憶領域も 8 バイトですが、値の範囲は 0 ～ 18,446,744,073,709,551,615 です。他の UNSIGNED 整数型と同様、この値も正の範囲を 2 倍にした値です。

Tip

プログラミング言語によっては、これらのデータ型に対して別の名前を使用しています。SQL Anywhere 9 の SMALLINT は、"short integer"、または単に "integer" と呼ばれることがあります。INTEGER は、"long integer"、"long"、または "integer" と呼ばれることがあります。BIGINT は使用できない可能性があります。そのため、真数（exact decimal）型を多少変更したデータ型を使用する必要があります。

1.5.6 リテラル

識別子は二重引用符（" "）で囲みますが、文字列リテラルは一重引用符（' '）で囲んで記述します。一重引用符を 2 つ続けることで、一重引用符自体を表すことができます（例：Fred''s Not Here）。円記号はエスケープ文字として解釈されることがあるため、注意してください。円記号に続く 1 文字または複数の文字の意味が変更されてしまいます。

¥xnn と ¥Xnn は、00 ～ FF の範囲の 16 進数値 nn で、1 つの文字を表します。たとえば、'¥x31¥x32¥x33' は '123' と同じです。'Fred¥X27s Not Here' と 'Fred''s Not Here' は同じです。

¥n は改行文字を表し、¥x0A と同じです。'c:¥new' と記述するのは危険です。しかし、'c:¥New' ならば、'¥N' に意味はないので問題ありません。

¥¥ は 1 つの円記号を表します。このため、'c:¥¥new' と記述すれば問題を解決できます。

Tip

円記号は常にエスケープ文字として解釈されるわけではありませんが、1 つの円記号を表す場合は、常に '¥¥' を使用するのが安全です。たとえば、'¥abc' と '¥¥abc' は同じ値です。

バイナリ文字列リテラルは文字列リテラルと同じように記述されることもあれば、0xnn... のように、引用符で囲まずに記述されることもあります。nn... は、00 ～ FF の範囲の 1 つ以上の 16 進数値です。たとえば、0x48656C6C6F2C20776F726C6421 は 'Hello, world!' と同じです。これは、文字列リテラルの '¥xnn' フォーマットと同じではないことに注意してください。'¥xnn' フォーマットには引用符が必要で、すべての文字の前に ¥x を繰り返す必要があります。

Tip

アプリケーション・プログラムがバイナリ値を文字データとして処理するようにするには、CAST 関数を使用してデータ型を変更します。たとえば、ISQL で「SELECT CAST (0x48656C6C6F2C20776F726C6421 AS VARCHAR)」を実行すると、不可解な 16 進数値ではなく、'Hello, world!' と表示されます。バイナリ文字列と文字列の格納方法には相違がないことを忘れないでください。異なるのは解釈方法だけです。

日付はさまざまなフォーマットの文字列リテラルとして記述されますが、year-month-day フォーマットが最も安全です。たとえば、'2003-07-06'、'2003 07 06'、'2003/07/06'、'2003 07/06'、'2003/7-6'、'2003 7 6'、'20030706'、'2003/007-6'、'2003 July 6' はすべて、DATE に変換されると常に 2003 年 7 月 6 日として解釈されます。

'7-6-2003' などのフォーマットはあいまいで、DATE_ORDER データベース・オプションに従って解釈されます。DATE_ORDER が 'MDY' に設定されている場合は、'7-6-2003' は 7 月 6 日と解釈されます。しかし、DATE_ORDER が 'DMY' に設定されている場合は、'7-6-2003' は 6 月 7 日と解釈されます。

時間は、'h:m:s.ssssss' のフォーマットで文字列リテラルとして記述されます。時、分、秒の位置に 0 が先行する場合と先行しない場合があります。秒は省略できますが、時と分は必須です。12 時間表記でリテラルを記述する場合は、後続の AM または PM を追加することもできます。たとえば、正午 1 分過ぎは、'12:1'、'0:1:0.000000PM'、'12:01:00.000000'、'0012:0001:0000.00000000' として記述できます。

タイムスタンプは、日付と時刻のリテラルを連結したものとして記述できます。たとえば、'20030706 14:30' は、2003 年 7 月 6 日 2:30PM を表します。

Tip

日付と時刻は、常に同じ安全なフォーマットで格納されることに注意してください。紛らわしいのは入力フォーマットと出力フォーマットだけで、入力フォーマットの一例が文字列リテラルのレイアウトです。

数値リテラルは、123 などの整数リテラル、1.23 などの真数（exact decimal）、1.23 を表す 123E-2 などの科学表記で記述されます。

Section

1.6 カラム・プロパティ

カラムは、データ型に加えて、便利なプロパティを持つことができます。スプレッドシートのセルのように計算式を設定したり、デフォルト値を設定したり、トリガを作成しなくても自動的に更新したりできます。

```
<column_definition>      ::= <column_name> [ AS ] <data_type>
                               [ <column_definition_list> ]
<column_definition_list> ::= <column_definition_term> { <column_definition_term> }
<column_definition_term> ::= <column_property>
                           | <column_constraint>
<column_property>        ::= COMPUTE "(" <expression> ")"
                           | DEFAULT <default_value>
                           | IDENTITY
                           | NULL
```

Note

IDENTITY プロパティは、DEFAULT AUTOINCREMENT とまったく同じように実装されているため、1.8.1 項で DEFAULT AUTOINCREMENT と併せて説明します。

Section

1.7 計算カラム

COMPUTE 句を使用したカラムは、その値が計算式に基づく読み取り専用の計算カラムになります。計算カラムには、INSERT 文や UPDATE 文で明示的な値を指定することはできませんが、インデックスは作成できます。また、通常はトリガを必要とするような計算を、トリガを使用せずに実行できます。

次に、クエリの最適化時に、テーブルあたり 1 つのインデックスのみ使用するという制限にぶつかるクエリの例を示します。この WHERE 句には、個別のインデックスが適している 2 つのカラムが含まれています。

```
CREATE TABLE street (
   pkey           INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   street_name    VARCHAR ( 100 ) NOT NULL,
   range_part_1   INTEGER NOT NULL,
   range_part_2   VARCHAR ( 100 ) NOT NULL,
   postal_code    VARCHAR ( 6 ) NOT NULL,
   PRIMARY KEY ( pkey ) );

SELECT *
  FROM street
 WHERE street_name LIKE 'GRAND%'
   AND postal_code LIKE 'L6H5%';
```

Note

アプリケーション・プログラムでは、ホスト変数は通常、'GRAND%' のような値を保持するのに使用されますが、それが SQL Anywhere 9 に渡ると文字列リテラルのように扱われます。

リストを表示する前に、ユーザが目的の郵便番号の最初の少なくとも 3 文字と、通りの名前の最初の 2 文字を入力するものと想定すれば、この例のように fast_search という名前の計算カラムを追加することが考えられます。

```
CREATE TABLE street (
   pkey           INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   fast_search    VARCHAR ( 5 ) NOT NULL
                  COMPUTE ( STRING ( LEFT ( postal_code, 3 ),
                                     LEFT ( street_name, 2 ) ) ),
   street_name    VARCHAR ( 100 ) NOT NULL,
   range_part_1   INTEGER NOT NULL,
   range_part_2   VARCHAR ( 100 ) NOT NULL,
   postal_code    VARCHAR ( 6 ) NOT NULL,
   PRIMARY KEY ( pkey ) );
```

これによって、効果的なインデックスを作成できるようになり、WHERE 句に新しいカラムを追加できます。オプティマイザは新しいインデックスを使用して、処理速度を高速化します。

```
CREATE CLUSTERED INDEX xfast ON street ( fast_search );

SELECT *
  FROM street
 WHERE fast_search = 'L6HGR'
   AND street_name LIKE 'GRAND%'
   AND postal_code LIKE 'L6H5%';
```

ローが挿入されるたびに、1つのロー内のすべての計算カラムが自動的に計算され、ローが更新されると再計算されます。テーブル・スキーマが変更されて、カラムのデータ型やCOMPUTE 句の変更、カラムの追加、カラムの削除、カラムの名前変更、またはテーブルの名前変更が行われると、すべてのロー内のすべての計算カラムが再計算されます。計算カラムは、ローが取得されても再計算されません。クエリとインデックスにとっては効率的ですが、時間に依存する式には適さない場合があります。

Tip

インデックス付きの計算カラムを使用すると、既存のクエリを最適化できます。たとえば、WHERE (x + y + z) を含む多くのクエリは、(x + y + z) のインデックスから恩恵を受けるとします。ここで、x、y、zは、同じテーブル内のカラムです。2つのルールに従えば、クエリ・オプティマイザは、式を、計算カラムへの参照に自動的に置換できます。まず、WHERE 句内の引数と演算子と同じ順序で、COMPUTE 式の引数と演算子を記述します。次に、計算カラムのデータ型を、式のデータ型と同じにします。データ型を確認するには、EXPRTYPE 関数を使用します。

次に、SELECT がフルテーブル・スキャンとなるときに、計算カラムの恩恵を受けられるテーブルを示します。

```
CREATE TABLE t (
   pkey INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   x    INTEGER,
   y    INTEGER,
   z    INTEGER,
   PRIMARY KEY ( pkey ) );

SELECT * FROM t WHERE ( x + y + z ) BETWEEN 1501 AND 1510;
```

ISQL で次のコマンドを実行して、(x + y + z) が 'int' を返すことを確認します。

```
SELECT EXPRTYPE ( 'SELECT ( x + y + z ) FROM t', 1 );
```

このテーブルに計算カラムを追加し、クラスタード・インデックスを作成すると、BETWEEN 範囲クエリを効率化できます。

```
CREATE TABLE t (
   pkey INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   x    INTEGER,
   y    INTEGER,
   z    INTEGER,
   xyz  INTEGER COMPUTE ( x + y + z ),
   PRIMARY KEY ( pkey ) );

CREATE CLUSTERED INDEX xyz ON t ( xyz );

SELECT * FROM t WHERE ( x + y + z ) BETWEEN 1501 AND 1510;
```

WHERE 句にカラム xyz はありませんが、SELECT では新しいインデックスを使用できます。

Note

インデックスの使用の有無は、データの条件とクエリ・オプティマイザによる選択に依存します。クエリ・オプティマイザが該当の状況で実際に何を実行するかを正確に予測することは、不可能ではありませんが困難です。しかし、この例の趣旨は、計算カラムがあることでインデックスが利用されるという点を示すことです。

クラスタード・インデックスの詳細については、「10.7 インデックスの作成」を参照してください。

Section

1.8 DEFAULT値

ローの挿入時に明示的な値が指定されない場合は、DEFAULT 値が割り当てられます。ローの更新時には、特別なデフォルト値も適用されます。

```
<default_value>  ::= NULL
                   | AUTOINCREMENT
                   | GLOBAL AUTOINCREMENT
                   | GLOBAL AUTOINCREMENT "(" <partition_size> ")"
                   | <string_literal>
                   | [ "-" ] <number_literal>
                   | <special_literal>
```

```
                    | <special_update_default>
                    | <constant_function_call>
                    | "(" <constant_expression> ")"
<partition_size> ::= 1～4611686018427387903の範囲の整数リテラル
<string_literal> ::= 一重引用符で囲まれた文字の並び
<number_literal> ::= 整数、真数、または浮動小数点の数値リテラル
```

DEFAULT NULLは、いわばデフォルトのDEFAULTで、値を指定しなかった場合にこのカラムがNULLに設定されることを意味します。

1.8.1 DEFAULT AUTOINCREMENT

DEFAULT AUTOINCREMENTは、連続する各ローに値1、2、3、…を割り当てます。このプロパティは、単一のカラムから成る効率的で機械的なプライマリ・キーを作成するのに非常に役立ちます。ビジネス関連のプライマリ・キーに複数のカラムが含まれる場合、ビジネス関連のカラムが非常に大きい場合、時間の経過に伴い値が変化することがある場合、または単に明らかなプライマリ・キーがない場合に、この方法がよく使われます。また、テキスト・データが行単位に格納されているために、元のINSERTまたはLOAD TABLEの入力順序を保持する必要がある場合にも役立ちます。

DEFAULT AUTOINCREMENTを指定したカラムは、同じテーブルに複数存在することができます。各カラムにおける開始ポイント（次の値）はSYSCOLUMNカタログ・テーブルに格納されるため、このカラムにインデックスを作成することはパフォーマンス上必須というわけではありませんが、さまざまな理由によりインデックスが作成されることが多いです。DEFAULT AUTOINCREMENTプロパティを使用しても、同時実行性の問題は生じません。つまり、新しい値の計算にはロックやブロックが発生しないので、異なるユーザが同じテーブルにローを挿入できます。

挿入時に明示的な値を指定することによって、DEFAULT AUTOINCREMENTを上書きできます。指定した値が既存の値より大きい場合は、指定した値が新しい開始ポイントとして使用され、ギャップが生じます。たとえば、最大値が3の場合に明示的に7を挿入すると、次のDEFAULT AUTOINCREMENT値は8になります。ローが削除されてギャップが生じることもあります。欠如している値を明示的に挿入すれば、次のデフォルト値に影響を及ぼすことなく、ギャップを自分で埋めることができます。

ギャップを埋めるもう1つの方法は、sa_reset_identityシステム・プロシージャを呼び出して、次のINSERTで異なるデフォルト値が使用されるように、特定のテーブルの最後に使用された値をリセットする方法です。ただし、このプロシージャは、指定されたとおりの操作を実行し、実際に使用されている値をチェックしないので、十分に注意してください。大きい値の挿入／削除を実行した後で、安全にsa_reset_identityシステム・プロシージャを使用できる例を次に示します。この例では、実際に使用されている最も大きい値をSELECTを使用して確かめ、その値をsa_reset_identityに渡すことによって、t1.key_1内の最も高い値より上にあるギャップを削除します。

```
BEGIN
   DECLARE @max_key_1 INTEGER;
   SELECT MAX ( key_1 ) INTO @max_key_1 FROM t1;
   CALL sa_reset_identity ( 't1', 'DBA', @max_key_1 );
END;
```

BEGIN ブロック、DECLARE 文、CALL 文の詳細については、「第 8 章 パッケージ」を参照してください。

Note

sa_reset_identity システム・プロシージャは、テーブル名と所有者名を要求しますが、カラム名は要求しません。テーブルに複数の DEFAULT AUTOINCREMENT カラムがある場合は、すべての DEFAULT AUTOINCREMENT カラムに対して、同じ最後に使用された値が用意されます。

IDENTITY プロパティは、DEFAULT AUTOINCREMENT によって実装されます。ただし、Sybase Adaptive Server Enterprise の IDENTITY プロパティとは異なり、カラムが NOT NULL である必要はありません。また、サーバが異常終了したときにも、ギャップは生じません。

1.8.2 DEFAULT GLOBAL AUTOINCREMENT

DEFAULT GLOBAL AUTOINCREMENT プロパティは、DEFAULT AUTOINCREMENT にパーティションを追加したものです。これは、Mobile Link を使用する分散データベース内のプライマリ・キーにとって非常に役立ちます。各データベースにそれぞれ異なる開始ポイントを設定することによって、すべてのデータベースにわたってすべてのローがユニークな値を持つことができます。この環境では、GLOBAL_DATABASE_ID オプションを通じて、各リモート・データベースにユニークなグローバル・データベース識別子 1、2、3、…が割り当てられます。中央の統合データベースを識別するには、値 0 がよく使用されます。

各データベース内の各 DEFAULT GLOBAL AUTOINCREMENT カラムの開始ポイントは、GLOBAL_DATABASE_ID オプション値 g に、カラムの分割（パーティション）サイズ p を掛けて、1 を追加する―つまり、「(g * p) + 1」を計算する―ことによって求められます。

次の例では、開始ポイントは (5 * 1000) + 1 であり、先頭の 2 つのローが 5001 と 5002 で挿入されます。

```
SET OPTION PUBLIC.GLOBAL_DATABASE_ID = '5';
CREATE TABLE t (
   auto UNSIGNED BIGINT DEFAULT GLOBAL AUTOINCREMENT ( 1000 ) );
INSERT t VALUES ( DEFAULT );
INSERT t VALUES ( DEFAULT );
COMMIT;
SELECT * FROM t ORDER BY t.auto;
```

パーティションの範囲を外れた値を明示的に使用することも可能で、次のデフォルト値にも影響は及びません。ただし、パーティションの範囲内の値を明示的に使用した場合は、DEFAULT AUTOINCREMENT と同じように動作します。つまり、それによってギャップが生じ、またそのギャップを埋めることも可能です。

たとえば、次のコードを実行すると、値 9999（パーティションの範囲外）、5003（パーティションの範囲内の次のデフォルト）、5100（ギャップが生じる）、5004（ギャップを埋める）、5101（次のデフォルト）が挿入されます。

```
INSERT t VALUES ( 9999 );
INSERT t VALUES ( DEFAULT );
INSERT t VALUES ( 5100 );
INSERT t VALUES ( 5004 );
INSERT t VALUES ( DEFAULT );
COMMIT;
SELECT * FROM t ORDER BY t.auto;
-- 5001、5002、5003、5004、5100、5101、9999を表示する
```

パーティションの終端は「(g + 1) * p」にあり、上の例の場合、これは「(5 + 1) * 1000 = 6000」です。その後、次のデフォルト値は NULL になりますが、通常、これは許可されません（プライマリ・キーは NULL であってはいけません）。値が足りなくなる可能性がある場合、GlobalAutoincrement 型の EVENT を作成して、新しい値を GLOBAL_DATABASE_ID に割り当てることで対処できます。

ただし、ほとんどのアプリケーションではパーティションが 1000 より大きいため、値が足りなくなることは実質的にありません。たとえば、UNSIGNED BIGINT を使用すると、分割サイズの最大値は 4,294,967,298 で、GLOBAL_DATABASE_ID の最大値は 2,147,483,646 です。つまり、20 億もの各データベースに、すべてユニークな 40 億を超える異なるデフォルト値を持つことができます。データベースの数が少なくなれば、分割サイズはさらに大きくできます。GLOBAL_DATABASE_ID を 1000 に制限すると、各データベース内に 1000 兆ものユニークな値を作成できます。

```
SET OPTION PUBLIC.GLOBAL_DATABASE_ID = '1000';
CREATE TABLE t (
   auto UNSIGNED BIGINT DEFAULT GLOBAL AUTOINCREMENT ( 1000000000000000 ) );
INSERT t VALUES ( DEFAULT );                -- 1000000000000001
INSERT t VALUES ( DEFAULT );                --1000000000000002
INSERT t VALUES ( 1001 * 1000000000000000 ); -- 1001000000000000
INSERT t VALUES ( DEFAULT );                -- NULL
```

1.8.3 リテラル・デフォルト

文字列、数値、日付／時刻の各カラムには、単純なリテラル DEFAULT 値を指定できます。次にその例を示します。

```
CREATE TABLE t (
   c1  INTEGER,
   c2  VARCHAR ( 1 )    DEFAULT 'Y',                -- Y
   c3  BINARY ( 20 )    DEFAULT 0x48656C6C6F,       -- Hello
   c4  VARCHAR ( 1 )    DEFAULT '¥n',               -- 改行
   c5  VARCHAR ( 100 )  DEFAULT 'c:¥¥new',          -- c:¥new
   c6  LONG VARCHAR     DEFAULT '¥x61¥x62¥x63',     -- abc
   c7  INTEGER          DEFAULT 0,                  -- 0
   c8  DECIMAL ( 9, 2 ) DEFAULT 27.95,              -- 27.95
   c9  DOUBLE           DEFAULT -123.456E-2,        -- -1.23456
   c10 DATE             DEFAULT '2003 07 06',       -- 2003年7月6日
   c11 TIME             DEFAULT '00:01',            -- 真夜中1分過ぎ
   c12 TIMESTAMP        DEFAULT '20030706 14:30' ); -- 2003年7月6日2:30 PM
```

SQL Anywhereには、式とDEFAULT指定で使用できる、いくつかの特殊リテラルが用意されています。これらのリテラルは、「特殊定数」と呼ばれることがありますが、実際には定数ではありません。時間の経過と共に変化するものもあれば、プログラム実行の状況を反映して変化するものもあります。ただし、DEFAULT値として使用する場合は、挿入されるローにコピーされた時点で、その値が固定されます。

```
<special_literal> ::= CURRENT DATABASE
                    | CURRENT DATE
                    | CURRENT TIME
                    | CURRENT TIMESTAMP
                    | CURRENT USER
                    | CURRENT UTC TIMESTAMP
                    | SQLCODE
                    | SQLSTATE
                    | USER
```

特殊リテラルのCURRENT DATABASEは、データベースのVARCHAR (128)ランタイム名を返します（例：'test'）。

CURRENT DATEは、現在の日付を含むDATE値を返します（例：2003 06 11）。

CURRENT TIMEは、現在の時刻を含むTIME値を返します（例：10:16:40.940000）。プラットフォームによっては、秒の小数点以下の有効桁数は2または3だけです。

CURRENT TIMESTAMPは、現在の日付と時刻を含むTIMESTAMP値を返します（例：2003 06 11 10:16:40.940000）。これをDEFAULTに使用すると、「このローはいつ挿入されたか」という質問に答えることができます。プラットフォームによっては、秒の小数点以下の有効桁数は2または3だけです。

CURRENT USERは、現在の接続のユーザID（VARCHAR (128)値）を返します（例：'DBA'）。これをDEFAULTに設定したカラムを使用すれば、「このローはだれが挿入したか」という質問に答えることができます。

CURRENT UTC TIMESTAMPは、サーバのタイム・ゾーンの調整値によって調整されたCURRENT TIMESTAMP値を返し、協定世界時（UTC：Coordinated Universal Time）の値を形成します。たと

えば、TIME_ZONE_ADJUSTMENT オプションが '-240' で、CURRENT TIMESTAMP が 2003 06 11 10:16:40.940000 の場合、CURRENT UTC TIMESTAMP は 2003 06 11 14:16:40.940000 です。

SQLCODE は、前の文による数値警告またはエラー・コードを含む INTEGER を返します（例：100 は「ローが見つかりません」、-306 は「デッドロックが検出されました」を表します）。

SQLSTATE は、前の文による英数字警告またはエラー・コードを含む VARCHAR (5) 値を返します（例：'02000' は「ローが見つかりません」、'40001' は「デッドロックが検出されました」を表します）。

USER は、CURRENT USER とまったく同じで、現在接続しているユーザ ID を表します。

Tip

これらの特殊リテラルの用途は、DEFAULT 値に限定されません。リテラルを使用できる、SQL 内の任意の場所で使用できます。たとえば、SELECT CURRENT TIMESTAMP を発行することによって、アプリケーションはサーバ上で日付と時刻を見つけることができます。

1.8.4 特殊更新デフォルト

SQL Anywhere には、カラム値が明示的に指定されない場合に、更新と挿入の操作に適用される 3 つの特殊な DEFAULT 値が用意されています。前述の特殊リテラルとは異なり、この特殊値は、DEFAULT 値としてのみ使用できます。

```
<special_update_default> ::= LAST USER
                           | TIMESTAMP
                           | UTC TIMESTAMP
```

DEFAULT LAST USER は、現在の接続のユーザ ID（VARCHAR (128) 値）を返します。このデフォルトを持つカラムは、「このローを挿入または更新したのはだれか」という質問に答えられます。

DEFAULT TIMESTAMP は、現在の日付と時刻を含む TIMESTAMP 値を返し、「このローが挿入または更新されたのはいつか」という質問に答えられます。

DEFAULT UTC TIMESTAMP は、DEFAULT TIMESTAMP と同様に動作し、協定世界時を調整します。

これらの特殊更新デフォルトを使用すると、特殊なトリガ・ロジックを作成せずに、簡単な監査証跡を実装できます。DEFAULT TIMESTAMP は、Mobile Link の分散環境において特に有用であり、「前回の同期以降にローが挿入または更新されたため、このローをダウンロードする必要があるかどうか」という質問に答えることができます。

1.8.5 デフォルトとしての式

いくつかの限られた形式の関数呼び出しと式を、DEFAULT 値として使用することができます。計算カラムとは異なり、これらの式は、ローの挿入時に 1 度だけ評価されます。また、カラムやユーザ定義関数も含めて、データベース・オブジェクトを参照できません。つまり、リテラル値と、リテラル値をパラメータとして使用する組み込み関数にすぎません。

```
<constant_function_call> ::= データベース・オブジェクトを参照しない組み込み関数呼び出し
<constant_expression>    ::= データベース・オブジェクトを参照しない式
```

次にその例を示します。

```
CREATE TABLE t (
   c1  INTEGER,
   server   VARCHAR ( 100 ) DEFAULT PROPERTY ( 'MachineName' ),
   today    VARCHAR ( 100 ) DEFAULT DAYNAME ( CURRENT DATE ),
   tomorrow DATE            DEFAULT ( CURRENT DATE + 1 ),
   guid     BINARY ( 16 )   DEFAULT NEWID() );
```

t.server カラムは、サーバを実行しているコンピュータのマシン名または IP アドレスで初期化されます。t.today カラムのデフォルト値は、現在の日付の曜日です（例：'Thursday'）。t.tomorrow カラムは、CURRENT DATE に 1（日）を加えた値に初期化されます。

t.guid の初期値はグローバル・ユニーク識別子で、UUIDTOSTR() 関数によってフォーマットした場合、'28c47c41-9cb8-11d7-88d6-0000863a7c57' のような形式になります。

Tip

GLOBAL_DATABASE_ID オプションにユニークな値を割り当てることができない場合は、DEFAULT GLOBAL AUTOINCREMENT の代わりに、NEWID() デフォルトを使用できます。得られる値は、必ずしも最も効率的なプライマリ・キー・インデックスではありませんが【監注 2】、追加のプログラミングを行わなくても、グローバルにユニークであることが保証されています。

【監注 2】16 バイトの値なので、INTEGER 型などに比べて保存効率が悪いことを指している。

Section

1.9 NULLプロパティ

NULLプロパティは、このカラムにNULL値が格納される可能性があることを宣言するものです。NOT NULL制約が使用されない限り、すべてのカラムに対して、このルールがデフォルトで適用されます。ただし、BITカラムは唯一の例外です。BITカラムはデフォルトでNOT NULLであるため、BITカラムにnull値を入力できるようにするためには、NULLプロパティを使用する必要があります。

NULLとDEFAULT NULLは、それぞれ異なります。NULLは、ローの挿入時または更新時に、このカラムにNULL値を明示的に割り当てられることを意味するのに対し、DEFAULT NULLは、ローの挿入時に値を明示的に設定しない限り、このカラムにNULL値が割り当てられることを意味します。

Section

1.10 カラム制約

カラム制約とは、カラムへの値の格納に関するルールです。カラム制約は妥当性に関するアサーションとして記述され、ローが挿入、更新、削除されるときに評価されます。制約が失敗した場合（FALSEと評価されると）、2つの状態のいずれかが発生します。ほとんどの場合は、エラー・メッセージが表示され、違反操作がキャンセルされます。場合によっては、メッセージも表示されずに、データが自動的に修復されます。制約が成功した場合、処理が続行します。

プログラミングをするうえで、カラム制約は非常に有用です。開発の早い段階で、データに関する誤りを知ることができます。データベース内に制約が多いほど、開発プロセスのきわめて初期の段階でプログラミング・エラーが見つかる可能性が高まります。これは、とてもよいことです。

制約があると、無効なデータを格納するのが非常に困難になるため、データベース管理者にとっても有用です。ビジネス関連の検証ルールをカラム制約として1つの中央のロケーション（データベース）に格納することで、それをすべてのアプリケーションや特定の更新に適用できます。どのようなプログラムあるいは上級ユーザであっても、カラム制約を回避したり省略したりすることはできません。

```
<column_constraint>  ::= NOT NULL
                       | [ <constraint_prefix> ]
                           CHECK "(" <boolean_expression> ")"
                       | [ <constraint_prefix> ]
                           PRIMARY KEY [ <clustering> ]
                       | [ <constraint_prefix> ]
                           REFERENCES
                           [ <owner_name> "." ]
```

```
                          <table_name>
                          [ "(" <column_name> ")" ]
                          [ <on_action> ]
                          [ <clustering> ]
                        | [ <constraint_prefix> ]
                          UNIQUE [ <clustering> ]
<constraint_prefix>  ::= CONSTRAINT [ <constraint_name> ]
<constraint_name>    ::= <identifier>
<boolean_expression> ::=「第3章 選択」の<boolean_expression>を参照
<clustering>         ::= CLUSTERED
                       | NONCLUSTERED
<table_name>         ::= <identifier>
<on_action>          ::= ON UPDATE <action>
                       | ON UPDATE <action> ON DELETE <action>
                       | ON DELETE <action>
                       | ON DELETE <action> ON UPDATE <action>
<action>             ::= CASCADE | SET NULL | SET DEFAULT | RESTRICT
```

1.10.1 NOT NULL制約

NOT NULL制約を指定したカラムには、NULL値を格納できません。この制約は、式、クエリ、他のSQL文を簡素化するために、可能な限りすべてのカラムに指定することをお勧めします。NULL、およびTRUE、FALSE、UNKNOWNの各値を使用する特殊な3値的論理システムとの関係については、「3.12 ブール式とWHERE句」を参照してください。

1.10.2 カラムCHECK制約

カラムCHECK制約を使用すると、カラムが変更されるたびに探索条件をテストできます。探索条件は、「CHECK (salary >= 0)」のような単純な比較から、クエリのWHERE句で使用できるすべての機能が含まれる非常に複雑な式まで、多岐にわたります。探索条件の詳細については、「第3章 選択」を参照してください。

次のコード例では、データベースのスキーマが最初に作成された日時を記録します。CHECK制約によって、このテーブルには1つのローのみ格納されることが保証されます。

```
CREATE TABLE schema_created (
   pkey    INTEGER NOT NULL DEFAULT 1 CHECK ( pkey = 1 ),
   created TIMESTAMP NOT NULL DEFAULT CURRENT TIMESTAMP,
   PRIMARY KEY ( pkey ) );

INSERT schema_created VALUES ( DEFAULT, DEFAULT );
```

次に示すのは、新しい値が常に増加することを保証するCHECK制約の例です。

```
CREATE TABLE t (
   c1 INTEGER CHECK ( c1 > ( SELECT MAX ( c1 ) FROM t ) ) );

INSERT t VALUES ( 3 ); -- OK
INSERT t VALUES ( 4 ); -- OK
INSERT t VALUES ( 1 ); -- 失敗
```

CHECK 制約は、FALSE を返す場合のみ失敗します。TRUE 値と UNKNOWN 値は、成功として処理されます。これが、上の例で、最初の INSERT が動作する理由です。テーブルは空なので、「SELECT MAX (c1) FROM t」によって NULL が返り、CHECK 制約は UNKNOWN を返します。

2 番目の INSERT では、4 は 3 より大きく、CHECK 制約は TRUE であるため、2 番目の INSERT も動作します。

3 番目の INSERT は失敗します。新しい最大値の 4 より、1 が小さいためです。このとき、「制約 'ASA92' の違反です：テーブル 't' 内のカラム 'c1' 内の値が無効です」というエラー・メッセージが表示されます。'ASA92' は、CREATE TABLE に制約名が指定されていないため、この CHECK 制約に自動的に割り当てられた制約名です。

独自の制約名を割り当てることによって、このエラー・メッセージの意味をわかりやすくすることができます。

```
CREATE TABLE t (
   c1 INTEGER CONSTRAINT "c1 must increase in value"
              CHECK ( c1 > ( SELECT MAX ( c1 ) FROM t ) ) );
```

これで、「制約「c1 の値は増加し続けなければならない」の違反です：テーブル 't' 内のカラム 'c1' 内の値が無効です」のようなエラー・メッセージが表示されます。

Tip

SQL Anywhere の名前と識別子はすべて、最大 128 文字まで指定できます。また、ブランクや特殊文字を含む長い語句を使用する場合は、二重引用符や角カッコで囲みます。

1.10.3 PRIMARY KEYカラム制約

PRIMARY KEY カラム制約は、このカラムの値が NULL 値以外であり、かつすべての値が相互に異なることを指定します。PRIMARY KEY 制約ごとにユニーク・インデックスが自動的に作成されるため、明示的にユニーク・インデックスを定義する必要はありません。このインデックスは CLUSTERED または NONCLUSTERED として定義され、NONCLUSTERED がデフォルトです。クラスタード・インデックスの詳細については、「10.7 インデックスの作成」を参照してください。

どのテーブルにもプライマリ・キーがなければなりません。プライマリ・キーは、テーブル間の階層関係や他の関係を定義するうえで非常に貴重であり、パフォーマンスの点でも非常に重要です。

PRIMARY KEY 制約は、ユニーク・インデックスとは異なります。インデックスではNULL 値を使用できますが、PRIMARY KEY では使用できません。

1.10.4 外部キー・カラム制約

外部キー・カラム制約では、REFERENCES 句を使用して、このテーブル（子テーブルまたは外部テーブルと呼ばれます）と他のテーブル（親テーブルまたはプライマリ・テーブルと呼ばれます）との関係を定義します。外部キー制約は、親テーブルと子テーブルの間の1対多の関係を表すのに使用されます。次の例では、1つの国に複数の支店があるという関係を表しています。

```
CREATE TABLE country (
   country_code  VARCHAR ( 2 ) PRIMARY KEY,
   name          VARCHAR ( 100 ) NOT NULL );

CREATE TABLE office (
   office_code  VARCHAR ( 10 ) PRIMARY KEY,
   country_code VARCHAR ( 2 ) NULL
                CONSTRAINT "office.country_code must be valid or NULL"
                REFERENCES country ( country_code )
                ON UPDATE CASCADE ON DELETE SET NULL );
```

この例では、office 内の country_code カラムは、NULL の場合があります。NULL でない場合は、country テーブルの country_code カラムに対応する値を格納する必要があります。これがREFERENCES 句の目的であり、親テーブルとそのPRIMARY KEY カラムを名前で指定しています。

REFERENCES 句は、子テーブル内のこのローに対応する、親テーブル内の1つのローを識別するために使用されます。REFERENCES 句の参照先としては、PRIMARY KEY カラムか、UNIQUE 制約が設定された親カラムを指定できますが、ほとんどの場合、PRIMARY KEY カラムが使用されます。REFERENCES 句に親カラム名が指定されていない場合は、PRIMARY KEY カラムががデフォルトで使用されます。

上記の例では、外部キー制約の名前を指定しています。制約が失敗すると、この名前がエラー・メッセージに表示されます。たとえば、次に示すINSERT の4番目は失敗し、「テーブル 'office' の外部キー 'office.country_code は有効な値かNULL でなければならない ' に対応するプライマリ・キーの値がありません」というエラー・メッセージが表示されます。

```
INSERT country VALUES ( 'CA', 'Canada' );
INSERT office VALUES ( '001', 'CA' ); -- OK
INSERT office VALUES ( '002', NULL ); -- OK
INSERT office VALUES ( '003', 'XX' ); -- 失敗
```

外部キー制約が失敗する主な理由は2つあります。1つは、制約を満たさない値で、子ローを挿入（INSERT）または更新（UPDATE）しようとした場合です。もう1つは、1つ以上の子ローが制約違反となるような方法で、親ローを更新（UPDATE）または削除（DELETE）しようとした場合です。子テーブルへの変更が原因で生じるエラーを回避することはできませんが、親テーブル内での変更が原因の違反は、ON UPDATE句とON DELETE句で簡単に修正できます。

ON UPDATE句では、親カラムの値を変更したときに、対応するすべての子ローのうち外部キー制約違反となったものに対して、3種類のいずれかの操作を実行することを指定します。ON UPDATE CASCADEは、新しい親カラムに対応するように、子カラムに同じ変更を加えます。ON UPDATE SET NULLは、子カラムをNULLに変更して、少なくとも制約に違反しないようにします。ON UPDATE SET DEFAULTは、子カラムをDEFAULT値（親テーブル内のいずれかのローに対応すると思われる値）に変更します。

ON DELETE句では、親ローが削除されたときに、対応するすべての子ローで、3種類のいずれかの操作を実行することを指定します。ON DELETE CASCADEは、対応するすべての子ローを削除するという強引な方法で、違反を排除します。ON DELETE SET NULLは、子カラムをNULLに変更して、制約に違反しないようにします。ON DELETE SET DEFAULTは、子カラムを、まだ削除されていない親テーブル内の他のローに対応するDEFAULT値に変更します。

これらの修復方法はすべて、エラー・メッセージの通知なしに行われますが、UPDATEとDELETEのいずれにも、さらに4番目の方法があります。それは、デフォルトの動作であるON UPDATE RESTRICTとON DELETE RESTRICTです。どちらもエラー・メッセージが表示され、親テーブルでの操作が阻止されます。

パフォーマンス上の理由から、外部キー制約ごとにインデックスが作成されるため、明示的にインデックスを定義する必要はありません。このインデックスはCLUSTEREDまたはNONCLUSTEREDとして定義され、NONCLUSTEREDがデフォルトです。クラスタード・インデックスの詳細については、「10.7 インデックスの作成」を参照してください。

1.10.5 UNIQUEカラム制約

UNIQUEカラム制約は、すべての値がNULL以外の値で、すべての値が異なることを指定します。この制約はユニーク・インデックスを使用して実装されますが、ユニーク・インデックスではNULL値を使用できるため、ユニーク・インデックスとUNIQUE制約は異なります。また、外部キー制約を別のテーブルで定義する際、UNIQUE制約をPRIMARY KEYと同じように扱うことができます。テーブルには、「候補キー」が複数存在する可能性がありますが、PRIMARY KEYとして定義できるのはその中の1つだけであり、他はUNIQUE制約です。

UNIQUE制約に対応するインデックスはCLUSTEREDまたはNONCLUSTEREDとして定義され、NONCLUSTEREDがデフォルトです。クラスタード・インデックスの詳細については、「10.7 インデックスの作成」を参照してください。

Section

1.11 ユーザ定義データ型

SQL Anywhere では、組み込みデータ型に DEFAULT 値、CHECK 条件、NULL プロパティを組み合わせて、ユーザ定義データ型を作成できます。このユーザ定義データ型は、組み込みデータ型と同じように使用できます。

```
<create_domain>          ::= CREATE DOMAIN <domain_definition>
                           | CREATE DATATYPE <domain_definition>
<domain_definition>      ::= <user_defined_data_type> [ AS ] <data_type>
                                [ <domain_property_list> ]
<user_defined_data_type> ::= <identifier>
<domain_property_list>   ::= <domain_property> { "," <domain_property> }
<domain_property>        ::= DEFAULT <default_value>
                           | IDENTITY
                           | NULL
                           | NOT NULL
                           | CHECK "(" <boolean_expression> ")"
```

ユーザ定義データ型は、簡単に言うと、完全なカラム定義の省略形です。簡単に記述できることが唯一の利点ですが、その一方で多数の欠点があります。第1に、ユーザ定義データ型は、ユーザ定義データ型へのすべての参照を削除してからでないと、削除または変更ができません。第2に、ユーザ定義データ型によって、アプリケーション・プログラマは本当のカラム定義を目にすることができません。そしてその情報は、多くの場合、アプリケーションを作成するうえで重要です。第3に、制約名を使用できないため、エラー・メッセージを意味のあるものにすることが困難です。

なお、組み込みの基本データ型を除いて、ユーザ定義データ型のすべてのプロパティは、CREATE TABLE 内で上書きできます。次に、INSERT によって address_2 に空の文字列を挿入し、address_3 に NULL を挿入する例を示します。これらの値は CREATE DOMAIN の定義に反しますが、挿入は成功します。

```
CREATE DOMAIN address AS VARCHAR ( 100 )
                         NOT NULL
                         DEFAULT ''
                         CHECK ( LENGTH ( TRIM ( @col ) ) > 0 );

CREATE TABLE office (
   office_code INTEGER PRIMARY KEY,
   address_1   address,
   address_2   address CHECK ( address_2 IS NOT NULL ),
   address_3   address NULL DEFAULT ( NULL ) );

INSERT office ( office_code, address_1 ) VALUES ( 1, '123 Main Street' );
```

CREATE DOMAIN の CHECK 条件で現在のカラム名を参照するには、@ で始まる任意の識別子を使用します。この例の場合、CREATE TABLE が処理されると、@col が address_1、address_2、address_3 に置き換えられます。

SQL Anywhere 9 は、Microsoft SQL Server および Sybase Adaptive Server Enterprise との互換性を維持するために、いくつかの単純なユーザ定義データ型を備えています。たとえば、DATETIME は組み込みデータ型の TIMESTAMP に対応し、TEXT は LONG VARCHAR として定義されています。

```
<builtin_user_defined_data_type> ::= DATETIME
                                   | IMAGE
                                   | MONEY
                                   | OLDBIT
                                   | SMALLDATETIME
                                   | SMALLMONEY
                                   | SYSNAME
                                   | TEXT
                                   | UNIQUEIDENTIFIER
                                   | UNIQUEIDENTIFIERSTR
                                   | XML
```

組み込みのユーザ定義データ型を表 1-2 に示します。

表 1-2　組み込みのユーザ定義データ型

データ型	定義
DATETIME	TIMESTAMP
IMAGE	LONG BINARY
MONEY	NUMERIC (19, 4)
OLDBIT	TINYINT
SMALLDATETIME	TIMESTAMP
SMALLMONEY	NUMERIC (10, 4)
SYSNAME	VARCHAR (30) NOT NULL
TEXT	LONG VARCHAR
UNIQUEIDENTIFIER	BINARY(16)
UNIQUEIDENTIFIERSTR	CHAR (36)
XML	LONG VARCHAR

Section

1.12 空き領域

CREATE TABLEの要素のうち、「テーブル・プロパティ」に分類されるのは、現在のところPCTFREEで指定する空き領域の割合だけです。

```
<table_element>  ::= <column_definition>
                   | <table_property>
                   | <table_constraint>
<table_property> ::= PCTFREE <free_percent>
<free_percent>   ::= 0～100の範囲の整数リテラル
```

データベースに新しいローが挿入される場合、ページ・サイズが1Kのときは、デフォルトで100バイトの空き領域が各ページに残るように挿入されます。ページ・サイズがこれより大きいときは、空き領域は200バイトです。空き領域を残すことで、ローのサイズが増えても、ローを分割して個別のページに格納しなくて済みます。カラムが大きくなるとローのサイズが大きくなります。ローの分割が多くなり、内部的に断片化すると、データベースのI/Oが低下します。

0～100の範囲でPCTFREEの割合を指定することで、デフォルトの空き領域の大きさを上書きできます。0は、断片化を考慮しないことを意味します。データの挿入時に、サーバは各ページを満杯にできます。新しい各ローを別々のページに挿入するには、100を使用します。ただし、最低でも1ページに1つのローを格納しなければならないため、100%の空き領域を確保することは論理的に不可能です。

空き領域の設定をどのようにしても、大きな文字列を格納したときに必ずページがいっぱいになるとは限りません。文字列の254バイトを超えた部分は、1つ以上の拡張ページに格納されるからです。長いカラムと拡張ページの詳細については、「10.6.2 テーブルの断片化」を参照してください。

また、新しいローが挿入され、そのローが1つのページ全体を占める場合は、空き領域の設定にかかわらず、ローは分割されません。

Section

1.13 テーブル制約

カラム制約の代わりにテーブル制約を使用して、同じ内容を定義できます。テーブル制約は、1つの大きな例外を除いて、カラム制約とまったく同じです。それは、テーブル制約は、同時に複数のカラムに適用できることです。

```
<table_constraint> ::= <table_check_constraint>
                     | <primary_key_table_constraint>
                     | <foreign_key_table_constraint>
                     | <unique_table_constraint>
```

1.13.1 テーブルCHECK制約

```
<table_check_constraint> ::= [ <constraint_prefix> ] CHECK
                               "(" <boolean_expression> ")"
```

テーブル CHECK 制約は、カラム CHECK 制約とは異なり、現在のロー内の複数のカラムを参照できます。次に、各カラムの値を前のカラムより必ず大きくさせる例を示します。

```
CREATE TABLE t (
   c1 INTEGER,
   c2 INTEGER,
   c3 INTEGER,
   CONSTRAINT "0 < c1 < c2 < c3"
   CHECK ( 0 < c1 AND c1 < c2 AND c2 < c3 ) );

INSERT t VALUES ( 1, 2, 3 ); -- OK
INSERT t VALUES ( 2, 2, 2 ); -- 失敗
```

2番目の INSERT は失敗し、「制約「0 < c1 < c2 < c3」の違反です。：テーブル 't' 内のカラム 'c2' 内の値が無効です」というエラー・メッセージが表示されます。

1.13.2 PRIMARY KEYテーブル制約

PRIMARY KEY テーブル制約には、プライマリ・キーを構成する1つ以上のカラム名のリストが含まれます。これらのどのカラムも NULL 値を保持できず、値の組み合わせはすべてのローでユニークです。

```
<primary_key_table_constraint> ::= [ <constraint_prefix> ] PRIMARY KEY
                                     [ <clustering> ]
                                     "(" <column_name_list> ")"
<column_name_list>             ::= <column_name> { "," <column_name> }
```

複数のカラムから構成されるプライマリ・キーは、通常、「複合プライマリ・キー」と呼ばれます。次に、監査証跡テーブルの例を示します。このテーブルには、別のテーブルから同じローのコピーを複数格納することができます。日付／時刻カラムによって、各コピーはユニークなプライマリ・キーを持ちます。

```
CREATE TABLE audit_office (
   copied       TIMESTAMP DEFAULT TIMESTAMP,
   office_code  VARCHAR ( 10 ),
   country_code VARCHAR ( 2 ),
   PRIMARY KEY ( copied, office_code ) );
```

Tip

ローがどれほど速く挿入されても、異なる値を割り当てる必要がある場合は、DEFAULT CURRENT TIMESTAMPの代わりにDEFAULT TIMESTAMPを使用します。DEFAULT_TIMESTAMP_INCREMENTオプションによって、各値は少なくとも1マイクロ秒だけ区別されますが、これはDEFAULT TIMESTAMPにのみ適用され、DEFAULT CURRENT TIMESTAMPには適用されません。

プライマリ・キー・カラム上に自動的に作成されるユニーク・インデックスはCLUSTEREDまたはNONCLUSTEREDとして定義され、NONCLUSTEREDがデフォルトです。クラスタード・インデックスの詳細については、「10.7 インデックスの作成」を参照してください。

Tip

必ずPRIMARY KEYを指定するか、あるいは少なくともUNIQUE制約またはNULLを許容しないユニーク・インデックスを指定してください。キーまたはインデックス・エントリによってローをユニークに識別できるため、トランザクション・ログが小さくて済みます。このような識別ができない場合、変更のたびにロー全体をトランザクション・ログに書き込む必要があります。そうなると、ログ・ファイルがふくれ上がり、サーバの速度が低下します。

1.13.3 FOREIGN KEYテーブル制約

FOREIGN KEYテーブル制約には、対応するカラム制約より多くのオプションがあり、外部キーを構成するカラム名のリストを指定する必要もあるため、指定に多少手間がかかります。

```
<foreign_key_table_constraint> ::= [ <constraint_or_prefix> ]
                                     [ NOT NULL ]
                                     FOREIGN KEY [ <role_name> ]
                                     "(" <column_name_list> ")"
                                     REFERENCES [ <owner_name> "."] <table_name>
                                     [ "(" <column_name_list> ")" ]
                                     [ <on_action> ]
                                     [ CHECK ON COMMIT ]
                                     [ <clustering> ]
<constraint_or_prefix>         ::= CONSTRAINT
```

```
                                | CONSTRAINT <constraint_name>
                                | <constraint_name>
<role_name>                   ::= <identifier>
```

次に、3つのレベルから成る階層構造の例を示します。この階層構造では、プライマリ・キーと外部キーのサイズが各レベルごとに大きくなっていきます。

```
CREATE TABLE country (
   country_code  VARCHAR ( 2 ),
   name          VARCHAR ( 100 ),
   PRIMARY KEY ( country_code ) );

CREATE TABLE office (
   country_code VARCHAR ( 2 ),
   office_code  VARCHAR ( 10 ),
   address      VARCHAR ( 1000 ),
   PRIMARY KEY ( country_code, office_code ),
   FOREIGN KEY ( country_code ) REFERENCES country );

CREATE TABLE representative (
   country_code      VARCHAR ( 2 ),
   office_code       VARCHAR ( 10 ),
   representative_id INTEGER,
   name              VARCHAR ( 100 ),
   PRIMARY KEY ( country_code, office_code, representative_id ),
   FOREIGN KEY ( country_code, office_code ) REFERENCES office );
```

制約名は、2つの場所のいずれかに指定できます。1つは先頭に（CONSTRAINT キーワードの指定は任意）、もう1つはFOREIGN KEY キーワードに続く役割名として指定できます。どちらの場合も、その名前がエラー・メッセージに表示されるため、この名前を工夫してメッセージの意味をわかりやすくすることができます。

NOT NULL句は、個々のカラムがNULLまたはNOT NULLとして定義されているかどうかにかかわらず、すべての外部キー・カラムがNOT NULLであることを指定する方法です。

REFERENCES句には、親テーブル名を指定します。オプションでカラム名のリストを指定できます。デフォルトは親テーブルのプライマリ・キー・カラムです。プライマリ・キーではなく親テーブルのUNIQUE制約を参照する場合は、UNIQUE制約のあるカラムの名前をREFERENCES句に指定します。

ON UPDATE句とON DELETE句は、外部キー・カラム制約の場合と同じように動作します。これらは、対応する親ローの更新と削除によって生じた制約違反を修復する操作を指定します。指定した操作は、通知なしで実行されます。デフォルトはRESTRICTで、エラー・メッセージが表示され、操作が阻止されます。

CHECK ON COMMIT句は、COMMITが実行されるまで、この制約のチェックを延期します。この機能は、「データの操作時に整合性ルールや整合性制約を省略できてはならない」というコッド博士のリ

レーショナル・データベースのルール 12 を曲げるものです。ただし、FOREIGN KEY 制約が完全に省略されるわけではなく、その適用が単に延期されるだけです。

CHECK ON COMMIT 句は、「正しい順序」（親を最初に挿入する、親を最後に削除する、など）で変更を加えることが難しい場合に役立ちます。変更が完了して COMMIT が発行される時点で FOREIGN KEY 制約に違反がなければ、アプリケーションは任意の順序でローの挿入、削除、更新ができます。

外部キー・カラムに自動的に作成されるインデックスは CLUSTERED または NONCLUSTERED として定義されます。デフォルトは NONCLUSTERED です。クラスタード・インデックスの詳細については、「10.7 インデックスの作成」を参照してください。

1.13.4 UNIQUEテーブル制約

```
<unique_table_constraint> ::= [ <constraint_prefix> ] UNIQUE
                                [ <clustering> ]
                                "(" <column_name_list> ")"
```

UNIQUE テーブル制約は、1 つ以上のカラム名のリストが必要なことを除いて、UNIQUE カラム制約とまったく同じです。2 つ以上のカラム名を指定した場合、ユニークでなければならないのは、個々のカラムの値ではなく、カラムの値の組み合わせです。次に、問題なく動作する 3 つの INSERT と、UNIQUE 制約に違反する 1 つの INSERT の例を示します。

```
CREATE TABLE t (
   c1 INTEGER PRIMARY KEY,
   c2 INTEGER,
   c3 INTEGER,
   UNIQUE ( c2, c3 ) );

INSERT t VALUES ( 1, 1, 1 ); -- OK
INSERT t VALUES ( 2, 1, 2 ); -- OK
INSERT t VALUES ( 3, 2, 1 ); -- OK
INSERT t VALUES ( 4, 1, 1 ); -- 失敗
```

UNIQUE 制約を指定したカラムに自動的に作成されるユニーク・インデックスは、CLUSTERED または NONCLUSTERED として定義されます。デフォルトは NONCLUSTERED です。クラスタード・インデックスの詳細については、「10.7 インデックスの作成」を参照してください。

Section

1.14 リモート・データ・アクセス

リモート・テーブルは、現在のデータベースとは別のロケーションにあるテーブルのことです。リモート・テーブルの例としては、Oracle テーブル、テキスト・ファイル、Excel スプレッドシート、別の SQL Anywhere データベース内のテーブルなどが挙げられます。リモート・テーブルのスキーマとデータは永続的です。

プロキシ・テーブルは、リモート・テーブル内のデータへの間接的なアクセスを提供します。プロキシ・テーブルのスキーマはローカル・データベースにありますが、データはリモート・ロケーションにあります。各プロキシ・テーブルは1つのリモート・テーブルに相当し、プロキシ・テーブルのスキーマは、リモート・テーブルのスキーマと同一か、または非常に似ています。

Tip

プロキシ・テーブルは、リモート・ビューにも定義できます。なぜなら、プロキシ・テーブルのインタフェースに関する限り、「リモート・テーブル」はテーブルのような外観を持つあらゆるものを表し、ビューもそれに含まれるためです。ビューを利用すると、リモート・サーバにクエリの処理を任せられるため、リモート・サーバ上のインデックスを有効に利用できるようになり、パフォーマンスが向上します。また、リモート・サーバ上でのみサポートされる構文を利用できるため、柔軟性も向上します。

Note

プロキシ・テーブルは優秀ですが、適切に動作しないこともあります。実際、その実行速度が、クエリの実行速度に比べてはるかに遅いことがあります。そのため、プロキシ・テーブルのパフォーマンスを向上させることは非常に重要です。

プロキシ・テーブルは、SQL 文の中で他のテーブルとまったく同じように参照できるため、複数のデータベースにまたがるジョインと更新が可能です。たとえば、次のクエリでは、2つの異なるリモート・データベースから従業員テーブルと販売予測テーブルをジョインして、SQL Anywhere の結果セットを生成します。

```
SELECT *
  FROM Oracle_employee
  JOIN DB2_sales_projection
    ON Oracle_employee.employee_id
     = DB2_sales_projection.employee_id;
```

外部ロケーションとの間でデータをコピーする場合にも、プロキシ・テーブルはよく利用されます。

次に、DB2_sales_projection プロキシ・テーブルを通じて、リモート・テーブルから sales_projection 永続テーブルに販売予測データをコピーする例を示します。

```
INSERT sales_projection
SELECT *
  FROM DB2_sales_projection;
```

プロキシ・テーブルを通じたリモート・テーブルへのアクセスは、特殊なミドルウェア（ソフトウェア）によって実現されています。このミドルウェアは、これまで OmniConnect という名前の別製品でしたが、現在は SQL Anywhere データベース・エンジンに組み込まれています。アプリケーションがプロキシ・テーブルに対して問い合わせや更新を行うと、このミドルウェアがリモート・ロケーションに処理を渡し、プロキシ・テーブルがデータそのものを保持しているかのように結果を返します。

CREATE TABLE 文に AT 句を指定すると、新しいリモート・テーブルと対応するプロキシ・テーブルを 1 回の操作で作成できます。また、既存のリモート・テーブルに対応するプロキシ・テーブルを定義することもできます。ただし、どちらの場合も、リモート・テーブルの検索方法をミドルウェアに指示する必要があります。これが、CREATE SERVER コマンドと CREATE EXTERNLOGIN コマンドの目的です。

1.14.1 CREATE SERVER

CREATE SERVER コマンドは、リモート・データ・サーバのローカル名を定義します。

```
<create_server>       ::= CREATE SERVER <server_local_name>
                            CLASS <server_access_class>
                            USING <connection_string>
                            [ READ ONLY ]
<server_local_name>   ::= <identifier>
<server_access_class> ::= 'ASAJDBC'
                        | 'ASEJDBC'
                        | 'ASAODBC'
                        | 'ASEODBC'
                        | 'DB2ODBC'
                        | 'MSSODBC'
                        | 'ORAODBC'
                        | 'ODBC'
<connection_string>   ::= データ・ソース名を含む ODBC 文字列リテラル
                        | UNIX ODBC 文字列リテラル 'driver=file_spec;dsn=dsn_name'
                        | JDBC 文字列リテラル 'machinename:portnumber[/dbname]'
```

サーバ・アクセス・クラスは、アクセスのメカニズム（ODBC または JDBC）を指定し、場合によってサーバ・タイプを指定します。現在、リストには ASA、ASE、DB2、Oracle、Microsoft SQL Server が含まれています。リストに名前がないサーバ・タイプに対しては、'ODBC' を使用してください（たと

えば、Excel ファイルとテキスト・ファイルに対しては 'ODBC' を使用します)。

接続文字列は、リモート・サーバの正確なロケーションを指定するもので、ODBC データ・ソース名を指定するか、JDBC マシン、ポート、データベース名から成る文字列を指定します。

READ ONLY 句を指定した場合、このサーバ上のデータを変更を加えることはできません。

Tip

CREATE SERVER コマンドと CREATE EXTERNLOGIN コマンドは、実際にはリモート・データベースに接続しません。プロキシ・テーブルを作成または使用するまで、この接続が動作するかどうかは判断できません。

1.14.2 CREATE EXTERNLOGIN

CREATE EXTERNLOGIN 文は、テキスト・ファイルなど一部のリモート・データではオプションですが、SQL Anywhere や Oracle など、アクセス制御を行うリモート・サーバでは必須です。

CREATE EXTERNLOGIN は、ローカル・ユーザ ID を指定します。このローカル・ユーザ ID は、CREATE SERVER コマンドで指定したサーバ名を通じて、リモート・テーブルとプロキシ・テーブルを操作するときに使用されます。ミドルウェアがリモート・サーバに接続できるように、リモート・ユーザ ID とリモート・パスワードを一緒に指定できます。

```
<create_external_login>   ::= CREATE EXTERNLOGIN <local_user_id>
                                  TO <server_local_name>
                                  [ REMOTE LOGIN <remote_user_id>
                                    [ IDENTIFIED BY <remote_password> ] ]
<local_user_id>           ::= <identifier>
<remote_user_id>          ::= <identifier>
<remote_password>         ::= <identifier> -- パスワードとして使用
                            | パスワードを含む空でない文字列リテラル
```

1.14.3 リモート・テーブルとプロキシ・テーブルの作成

CREATE TABLE … AT コマンドは、リモート・テーブルとプロキシ・テーブルを同時に作成します。グローバル永続テーブルの場合とまったく同じように、カラム定義と他のプロパティのリストを指定する必要がありますが、このリストがリモート・サーバにも送られて、そこでもテーブルが作成される点が異なります。リモート・サーバを指し、そのサーバ上のテーブル名を指定する文字列リテラルも指定する必要があります。

```
<create_remote_and_proxy_table> ::= CREATE TABLE [ <owner_name> "."] <table_name>
                                        <table_element_list>
                                        AT <remote_location>
<remote_location>    ::= リテラル 'server_local_name;[db_name];[owner];object_name'
```

リモート・ロケーションの文字列は、セミコロンまたはピリオドで区切られた4つの要素で構成されます。要素自体にピリオドが含まれる場合は、セミコロンを使う必要があります。最初の要素は、必ずCREATE SERVERコマンドで定義されるリモート・サーバのローカル名で、最後の要素は、必ずそのサーバ上のテーブル、ビュー、またはオブジェクト名です。他の2つの要素は、リモート・サーバごとにそれぞれ異なります。たとえば、Excelの場合は、2番目の要素はファイル仕様です。3番目の要素は省略され、最後の要素はシート名です。DB2、SQL Anywhere、他のほとんどのリレーショナル・データベースの場合は、2番目、3番目、4番目の要素は、それぞれデータベース名、所有者名、テーブル名またはビュー名です。Oracleは例外で、データベース名が省略されます。

次に、ブックasaexcel.xls内にtest1という名前の新しいExcelスプレッドシートを作成する例を示します。etestという名前のプロキシ・テーブルを通じて、ローを挿入および取得します。CREATE EXTERNLOGINが必要のないことに注意してください。

```
CREATE SERVER ASAEXCEL CLASS 'ODBC' USING 'EXCEL SAMPLE';

CREATE TABLE etest
   ( pkey INTEGER NOT NULL,
     fld1 VARCHAR ( 20 ) NOT NULL,
     fld2 INTEGER NOT NULL )
   AT 'ASAEXCEL;c:¥¥temp¥¥asaexcel.xls;;test1';

INSERT INTO etest VALUES ( 1, 'Hello, World', 9 );

SELECT * FROM etest;
```

Tip

ExcelのODBC DSNを作成してプロキシ・テーブルを使用する場合は、ODBCアドミニストレータの［ブックの選択］ボタンで、ダミーのブック仕様を指定する必要があります。このファイル仕様は、CREATE TABLEでは無視され、代わりにリモート・ロケーション文字列で指定されているファイル仕様が使用されます。

Tip

ほとんどのリモート・サーバは、プロキシ・テーブルを作成する際に実行されている必要がありますが、Excelにはこれが当てはまりません。

1.14.4 CREATE EXISTING TABLE

CREATE EXISTING TABLEコマンドは、リモート・データベース上の既存のテーブルの定義を取得し、その定義を使用してプロキシ・テーブルを作成します。

```
<create_proxy_table> ::= CREATE EXISTING TABLE [ <owner_name> "."] <table_name>
                            [ <proxy_column_list> ]
                            AT <remote_location>
<proxy_column_list>  ::= "(" <proxy_column> { "," <proxy_column> } ")"
<proxy_column>       ::= <column_name> <data_type> [ [ NOT ] NULL ]
```

ほとんどの場合、CREATE EXISTING TABLEはカラム・リストの指定なしで、リモート・テーブルとまったく同じレイアウトのプロキシ・テーブルを作成できます。次に、ora_deptという名前のプロキシ・テーブルを通じてSCOTT.DEPTという名前のOracleテーブルを使用する例を示します。

```
CREATE SERVER ORASAMPLE CLASS 'ORAODBC' USING 'ORA SAMPLE';

CREATE EXTERNLOGIN DBA TO ORASAMPLE
  REMOTE LOGIN system IDENTIFIED BY 'manager';

CREATE EXISTING TABLE ora_dept AT 'ORASAMPLE..SCOTT.DEPT';

SELECT * FROM ora_dept;
```

次に、DB2の例を示します。この例では、db2_departmentプロキシ・テーブルを通じて、リモート・テーブルのADMINISTRATOR.DEPARTMENTにクエリを実行しています。

```
CREATE SERVER DB2DELL180 CLASS 'DB2ODBC' USING 'DB2 SAMPLE';

CREATE EXTERNLOGIN DBA TO DB2DELL180
  REMOTE LOGIN db2admin IDENTIFIED BY 'secret';

CREATE EXISTING TABLE db2_department
  AT 'DB2DELL180.SAMPLE.ADMINISTRATOR.DEPARTMENT';

SELECT * FROM db2_department;
```

カラム名、データ型、NULL/NOT NULLプロパティのリストを明示的に指定することもできます。これらのリストを指定すれば、カラムを省略したり、カラムの順序を並べ替えたり、データ型を操作するといったことができます。この変更はリモート・テーブルには影響せず、対応するプロキシ・テーブルのレイアウトにのみ影響します。ただし、カラムによってリモート・テーブルとプロキシ・テーブルが対応しているため、カラム名を変更することはできません。次に、先ほど作成したExcelスプレッドシートを基に、etest2という別のプロキシ・テーブルを作成する例を示します。etest2とetestのレイアウト

は異なりますが、INSERT はそのまま動作します。

```
CREATE EXISTING TABLE etest2
   ( pkey NUMERIC ( 10 ) NOT NULL,
     fld1 VARCHAR ( 100 ) NOT NULL )
   AT 'ASAEXCEL;c:¥¥temp¥¥asaexcel.xls;;test1';

INSERT INTO etest2 VALUES ( 2, 'A value that is longer than 20 characters.' );

SELECT * FROM etest2;
```

以降では、2つの SQL Anywhere 9 データベースを使用する例を、最初から最後まで通して説明します。最初のデータベースはリモート・データベースで、2番目のデータベースにはプロキシ・テーブルが含まれます。まず、最初のデータベースでグローバル永続テーブル t1 を作成します。

```
CREATE TABLE t1 (
   pkey INTEGER NOT NULL,
   c1   VARCHAR ( 20 ) NOT NULL,
   PRIMARY KEY ( pkey ) );

INSERT INTO t1 VALUES ( 1, 'Hello, World' );
COMMIT;
```

2番目のデータベースでは、次のコマンドで、サーバ、外部ログイン、2つのプロキシ・テーブルを作成します。1番目のプロキシ・テーブル t1proxy は既存のテーブル t1 を指し、2番目のプロキシ・テーブル t2proxy を作成すると、リモート・データベース上に新しいテーブル t2 が作成されます。

```
CREATE SERVER other CLASS 'ASAODBC' USING 'otherdsn';

CREATE EXTERNLOGIN DBA TO other
  REMOTE LOGIN DBA IDENTIFIED BY "SQL";

CREATE EXISTING TABLE t1proxy AT 'other;otherdb;dba;t1';

SELECT * FROM t1proxy; -- 'Hello, World' を表示する

CREATE TABLE t2proxy (
   pkey INTEGER NOT NULL,
   c1   VARCHAR ( 20 ) NOT NULL,
   PRIMARY KEY ( pkey ) )
   AT 'other;otherdb;dba;t2';

INSERT INTO t2proxy VALUES ( 1, 'Goodbye' );

SELECT * FROM t2proxy; -- 'Goodbye' を表示する
```

Section

1.15 テンポラリ・テーブル

テンポラリ・テーブルは、データのロケーションと有効期間が永続テーブルと異なります。永続テーブルのデータはデータベース・ファイルに格納され、明示的に削除されるまで残ります。テンポラリ・テーブルのデータは、実際のデータベース・ファイルには格納されません。また、テンポラリ・テーブルのデータは最初に挿入した接続より、長く続くことはありません。サーバは、テンポラリ・テーブルのデータをメモリに保持しますが、メモリに余裕がない場合は、データベース・ファイルとは別の特殊な「テンポラリ・ファイル」を使用します。1つのデータベースにつき1つだけテンポラリ・ファイルが作成され、サーバが停止するときにファイルが削除されます。

テンポラリ・テーブルは、複雑なアプリケーションにおいて非常に有効です。テンポラリ・テーブルを使用すると、未加工の入力データをロードおよび処理してから、永続テーブルに格納できます。また、大規模なクエリを作成するのに、「分断統治」戦術をとることができます。つまり、元のクエリの一部をいくつかのテンポラリ・テーブルにロードし、最後にそれらのテンポラリ・テーブルを単純なクエリで結合します。いくつかの小さなクエリに分けることで、オプティマイザが適切に動作して全体の実行速度が上がる場合があります。

テンポラリ・テーブルは、通常、結果セットを返すストアド・プロシージャで使用されます。プロシージャでは、カーソル・ループや、テンポラリ・テーブルをロードするのに必要な他の複雑な技術を使用し、そのテーブルでSELECT * FROMを実行して、単純な結果セットを返します。この方法を使用すると、外部ソースの非リレーショナル・データをリレーショナル・データに変換してから処理できるようになります。

Tip

SQL Anywhere 9では、SELECT FROM句の中でプロシージャを呼び出すことができます。つまり、プロシージャをテーブルと同じように扱い、他のテーブルにジョインできます。

1.15.1 グローバル・テンポラリ・テーブル

グローバル・テンポラリ・テーブルは、スキーマがシステム・カタログ・テーブルに永久的に記録される唯一のテンポラリ・テーブルです。スキーマは、テーブルが明示的に削除されるまで保持され、テーブルはすべての接続で使用できます。ただし、データは接続ごとに分割され、各接続がテーブルのコピーをそれぞれ所有しているかのような状態になります。

グローバル永続テーブルと同様、グローバル・テンポラリ・テーブルは事前に作成し、接続で必要と

されるときに使用されます。また、ローカル・テンポラリ・テーブルと同様、ある接続によって挿入されたデータは、他のどの接続からも参照できません。

```
<create_global_temporary_table> ::= CREATE GLOBAL TEMPORARY TABLE
                                        [ <owner_name> "."] <table_name>
                                        <table_element_list>
                                        [ <commit_action> ]
<table_name>                    ::= <identifier>
<commit_action>                 ::= ON COMMIT DELETE ROWS
                                  | ON COMMIT PRESERVE ROWS
                                  | NOT TRANSACTIONAL
```

テーブル名の先頭をシャープ記号にするかどうかは任意です。どちらの場合でも、グローバル・テンポラリ・テーブルは作成されます。

グローバル・テンポラリ・テーブルは、グローバル永続テーブルと同じように、1つ以上のカラム定義に加えて、適用されるテーブル制約とプロパティから成るテーブル要素を設定できます。

コミット・アクションは、COMMITまたはROLLBACKが実行されたときにデータに対して行われる処理を制御します。ON COMMIT DELETE ROWSを指定すると、COMMITが実行されたときにすべてのデータが削除され、ROLLBACKが実行されたときに変更がロールバックされます。それ以外の場合、データは明示的に削除されるか接続が切断されるまで保持されます。デフォルトの動作がON COMMIT DELETE ROWSであるため、テストの際に「データはいったいどこに行ってしまったのか。コミットしただけなのに!」とあわてる原因になることがあります。

ON COMMIT PRESERVE ROWSを指定すると、COMMITの実行時にすべてのデータが削除されるのではなく、変更がコミットされます。これは、ロックを解放するためにコミットが頻繁に行われる長いプロセスで役に立ちます。

次の例は、ON COMMIT PRESERVE ROWSを指定すると、COMMITとROLLBACKが通常どおり動作することを示しています（したがって、2番目のローだけがSELECTで表示されます）。

```
CREATE GLOBAL TEMPORARY TABLE t (
   c1 INTEGER )
   ON COMMIT PRESERVE ROWS;

INSERT t VALUES ( 1 ); -- ロールバックされる
ROLLBACK;
INSERT t VALUES ( 2 ); -- コミットされる
COMMIT;
INSERT t VALUES ( 3 ); -- ロールバックされる
ROLLBACK;
SELECT * FROM t;       -- ロー #2 だけが表示される
```

Tip

同期のアップロード側とダウンロード側でMobile Linkスクリプトを実行し、グローバル・テンポラリ・テーブルを使用してデータを受け渡しする場合は、ON COMMIT PRESERVE ROWSを使用してください。これは、Mobile Linkがアップロードとダウンロードの間でCOMMITを発行するためです。ただし、新しい同期を開始するときには、古いデータを忘れずに削除してください。Mobile Linkは、次の同期で同じ接続を再利用します。

NOT TRANSACTIONALは、COMMITコマンドとROLLBACKコマンドがデータに影響しないことを意味します。COMMITが実行されても、データが自動的に削除されることはありません。NOT TRANSACTIONALにはコミットやロールバックの概念がなく、データは明示的に削除されるか接続が切断されるまで保持されます。

ROLLBACKとCOMMITがローに影響しない例を次に示します。挿入されたローは、SELECTで表示されます。

```
CREATE GLOBAL TEMPORARY TABLE t (
   c1 INTEGER )
   NOT TRANSACTIONAL;

INSERT t VALUES ( 1 );
ROLLBACK;              -- 影響しない
INSERT t VALUES ( 2 );
COMMIT;                -- 影響しない
SELECT * FROM t;       -- 両方のローが表示される
```

Tip

長時間にわたって実行するカーソル・ループの中でテンポラリ・テーブルを使用する場合、CREATEではON COMMIT PRESERVE ROWSを使用し、カーソルのOPENではWITH HOLDを使用します。これによって、ローを失ったり、カーソルが閉じたりすることなく、ループの中でCOMMITを実行できます。カーソルのループが失敗した場合に、ループを途中から再開するのではなく最初からやり直したい場合は、NOT TRANSACTIONAL句の方が適しています。

1.15.2 ローカル・テンポラリ・テーブル

ローカル・テンポラリ・テーブルは、システム・カタログには現れません。スキーマとデータは、テーブルを作成し、データを挿入した接続のみが参照できます。スキーマもデータも、現在の接続より長く保持されることはなく、それより前になくなることもあります。

ローカル・テンポラリ・テーブルを作成する方法には、CREATE TABLE文で#から始まるテーブ

ル名を指定する方法、DECLARE LOCAL TEMPORARY TABLEを使用する方法、INTO句を伴うSELECT文で#から始まるテーブル名を指定する方法の3種類があります。

1.15.2.1 CREATE TABLE #table_name

```
<create_local_temporary_table> ::= CREATE TABLE <temporary_table_name>
                                       <table_element_list>
<temporary_table_name>         ::= "#" { ( <alphabetic> | <numeric> ) }
```

この形式の場は、テーブル名の先頭をシャープ記号（#）にすることで、これがグローバル永続テーブルではなくローカル・テンポラリ・テーブルであることをSQL Anywhereに知らせます。

CREATE GLOBAL TEMPORARY TABLEとは異なり、コミット・アクション句はありません。デフォルトの動作は、ON COMMIT PRESERVE ROWSと同じです。つまり、COMMITとROLLBACKは、その命令どおり変更をコミットおよびロールバックし、またコミット時にローが自動的に削除されることはありません。次に例を示します。

```
CREATE TABLE #t ( c1 INTEGER );

INSERT #t VALUES ( 1 ); -- ロールバックされる
ROLLBACK;
INSERT #t VALUES ( 2 ); -- コミットされる
COMMIT;
INSERT #t VALUES ( 3 ); -- ロールバックされる
ROLLBACK;
SELECT * FROM #t;       -- ロー2だけが表示される
```

CREATE TABLE #table_nameが、ストアド・プロシージャ、またはBEGINブロックを使用する他の複合文の中で実行されると、その複合文の終了時に、テーブルが自動的に削除されます。複合文の外側で単独で実行される場合、テーブルとそのデータは、明示的に削除されるまで、または接続が終了するまで保持されます。

テンポラリ・テーブル名は、ネストのスコープがあります。つまり、複合文の中で同じ#table_nameを使用してテーブルを作成（CREATE）した場合、複合文が終了するまで、ネストの内側のテーブルしか参照できません。その後、ネストの内側のテーブルを削除すると、ネストの外側のテーブルを再び参照できるようになります。

次に、同じSELECTを実行しても、ネスト・テーブルのスコープの内側と外側で結果が異なる例を示します。CREATE TABLEは、複合文の内側の任意の位置に指定できますが、いったん実行されると、外部テーブルが参照できなくなることに注意してください。

```
CREATE TABLE #t ( c1 INTEGER );
INSERT #t VALUES ( 1 );
SELECT * FROM #t; -- 1が表示される
```

```
BEGIN
   SELECT * FROM #t; -- まだ1が表示される
   CREATE TABLE #t ( c1 INTEGER );
   INSERT #t VALUES ( 2 );
   SELECT * FROM #t; -- 2が表示される
END;

SELECT * FROM #t; -- 再び1が表示される
```

この形式のCREATE TABLEは、副次的な効果として自動COMMITを実行しません。つまり、トランザクション内でこの種類のテーブルを作成しても問題はなく、コミットとロールバックの論理が混乱することはありません。

Tip

ローカル・テンポラリ・テーブルは、ストアド・プロシージャ専用ではありません。クライアント側のアプリケーション・コードから作成し、使用することもできます。たとえば、PowerBuilderのEXECUTE IMMEDIATEを使用してテンポラリ・テーブルを作成し、DataWindow SELECTで参照できます。

1.15.2.2 DECLARE LOCAL TEMPORARY TABLE

```
<declare_local_temporary_table> ::= DECLARE LOCAL TEMPORARY TABLE <table_name>
                                        <table_element_list>
                                        [ <commit_action> ]
<table_name>                    ::= <identifier>
```

この形式の場合、テーブル名の先頭をシャープ記号にするかどうかは任意です。どちらの場合でも、ローカル・テンポラリ・テーブルは作成されます。

プロシージャまたは他の複合文の中でもDECLARE LOCAL TEMPORARY TABLEを使用できますが、その場合は、他のDECLARE文より前に指定する必要があります。テーブル名には、ネストのスコープがあります。複合文が終了するまでは、内部テーブルだけが参照可能です。そのテーブル名が、複合文の外側で作成されたグローバル永続テーブル、グローバル・テンポラリ・テーブル、または他のローカル・テンポラリ・テーブルと同じ名前であっても同様です。

次に、BEGIN/ENDブロックの中で、ローカル・テンポラリ・テーブル名が永続テーブルを上書きする例を示します。

```
CREATE TABLE t ( c1 INTEGER ); -- 永続テーブル
INSERT t VALUES ( 1 );
SELECT * FROM t;    -- 1が表示される
```

```
BEGIN
   DECLARE LOCAL TEMPORARY TABLE t ( c1 INTEGER );
   INSERT t VALUES ( 2 );
   SELECT * FROM t; -- 2が表示される
END;

SELECT * FROM t;    -- 再び1が表示される
```

コミット・アクション句は、CREATE GLOBAL TEMPORARY TABLEの場合と同じように動作します。デフォルトはON COMMIT DELETE ROWSです。ON COMMIT PRESERVE ROWSは、コミット実行時の自動削除をオフにします。NOT TRANSACTIONALを指定すると、コミット・コマンドとロールバック・コマンドはこのテーブル内のローを無視します。

パフォーマンスが気になる場合は、できるだけNOT TRANSACTIONALを使用してください。テンポラリ・テーブルに対する変更がトランザクション・ログに記録されることはありませんが、NOT TRANSACTIONALを指定しないと、ロールバック・ログには記録されます。テンポラリ・テーブルに対してロールバック・ログが使用されないようにすることで、パフォーマンスが向上する可能性があります。

DECLARE LOCAL TEMPORARY TABLEは、複合文の外側で実行可能な文とまったく同じように使用できます。この場合、新しいテーブルが、同じ名前のグローバル永続テーブルやグローバル・テンポラリ・テーブルを上書きします。次に示す例では、DECLARE LOCAL TEMPORARY TABLEで作成される新しいテーブルを明示的に削除するまで、グローバル・テンポラリ・テーブルが上書きされます。

```
CREATE GLOBAL TEMPORARY TABLE t ( c1 INTEGER );
INSERT t VALUES ( 1 );
SELECT * FROM t; -- 1が表示される

DECLARE LOCAL TEMPORARY TABLE t ( c1 INTEGER );
INSERT t VALUES ( 2 );
SELECT * FROM t; -- 2が表示される

DROP TABLE t;    -- テンポラリ・テーブルを削除する
SELECT * FROM t; -- 再び1が表示される
```

同じ名前のグローバル永続テーブルでも、同じ状態が発生します。つまり、既存のテーブルをテンポラリ・テーブルとして一時的に再定義できます。

DECLARE LOCAL TEMPORARY TABLEは、副次的な効果として自動COMMITを実行しません。そのため、トランザクションの内側で使用しても安全です。

1.15.2.3　SELECT INTO #table_name

```
<select_into_local_temporary_table> ::= SELECT              -- 第3章の<select>も参照
                                        [ <summarizer> ]
                                        [ <row_range> ]
                                        <select_list>
                                        INTO <temporary_table_name>
                                        [ <from_clause> ]
                                        [ <where_clause> ]
                                        [ <group_by_clause> ]
                                        [ <having_clause> ]
<temporary_table_name>              ::= "#" { ( <alphabetic> | <numeric> ) }
```

この形式の場合、テーブル名の先頭をシャープ記号（#）にしなければなりません。これによって、INTO 句に指定したものが変数名ではなくテーブル名であることを SQL Anywhere に伝えます。

SELECT INTO #table_name は、非常に強力です。テーブルを作成するだけでなく、同時にデータもロードします。テンポラリ・テーブルのカラム名とデータ型が select リストから取得され、SELECT を実行することでローがテンポラリ・テーブルに格納されます。したがって、select リスト内のカラム名は実際に存在していなければなりません。式の場合は、「AS 識別子」を使用して名前を指定できます。SELECT 文の詳細については、「第 3 章 選択」を参照してください。

次に、テーブル t と同一のコピーをテンポラリ・テーブル #t に作成する例を示します。両テーブルでは、カラム名、データ型、データのローが同じです。

```
CREATE TABLE t ( -- 永続テーブル
   c1 INTEGER,
   c2 VARCHAR ( 10 ),
   c3 TIMESTAMP );

INSERT t VALUES ( 1, 'AAA', CURRENT TIMESTAMP );
INSERT t VALUES ( 2, 'BBB', CURRENT TIMESTAMP );

SELECT * INTO #t FROM t; -- テンポラリ・テーブルへコピー
```

Tip

カラムの実際のデータ型を確認するには、それを SELECT に記述し、EXPRTYPE 関数を呼び出します。たとえば、「SELECT EXPRTYPE ('SELECT * FROM #t', 2)」を実行すると、#t の 2 番目のカラムが 'varchar(10)' として表示されます。

SELECT INTO #table_name で作成されるテーブルには、CREATE TABLE #table_name で作成されるテーブルと同様に、ネストのスコープがあります。SELECT INTO #table_name は、副次的な効果として自動 COMIIT を実行しないため、トランザクションの内側でも安全に使用できます。

Tip

SELECT INTO #t コマンドで作成されたテーブルに、INSERT #t SELECT * FROM t コマンドでさらにローを追加することもできます。このとき、どちらのコマンドにもカラム名をリストする必要はありません。INSERT 文の詳細については、「第２章 挿入」を参照してください。

Section

1.16 設計の正規化

「正規化」とは、手間とエラーの発生を減らすために、不要な冗長性を排除し、データベース設計を改善することです。データが冗長だと、複数のロケーションで同じデータを変更しなければならないため、必要な作業が増え、そこから発生するエラーとデータの矛盾に対処するために、さらなる労力が必要になります。

冗長性は、安全性と信頼性を高めるうえでは役に立ちます。たとえば、検算用の数字は他のデータから求めることができるため冗長ですが、入力エラーを検出できるというメリットがあります。しかし、冗長データも偶然にエラーになることがあり、その意味では何の役にも立ちません。

正規化の各ステップによって、1つのテーブルが、外部キー関係を持つ2つ以上のテーブルに分割されます。このプロセスは「正規形」という用語で定義され、より高度な改善を達成するためのガイドラインとなります。正規形には6種類あり、1から5の番号が付けられています。加えて、3番目と4番目の間に位置する、ボイスコッド正規形と呼ばれる中間レベルがあります。

重要なのは、各正規形を、正規化の過程として認識することではなく、冗長性を排除し、データの矛盾を回避することです。ここで正規形について説明するのは、正規形が、よく発生するさまざまな問題と、それらに対処するために必要な変更の方法を明らかにしてくれるからです。

次に示すのは、最初の3つの正規形に違反するテーブルです。このテーブルは、単純な紙ベースの注文フォームを表しており、ユニークな注文番号に加えて、顧客、営業部員、注文された製品に関する情報で構成されています。

```
CREATE TABLE order_form (
   order_number              INTEGER NOT NULL PRIMARY KEY,
   client_name               VARCHAR ( 100 ) NOT NULL,
   shipping_address          VARCHAR ( 1000 ) NOT NULL,
   salesperson_name          VARCHAR ( 100 ) NOT NULL,
   salesperson_phone         VARCHAR ( 100 ) NOT NULL,
   salesperson_commission    NUMERIC ( 6, 3 ) NOT NULL,
   product_number_1          INTEGER NOT NULL,
   product_description_1     VARCHAR ( 100 ) NOT NULL,
```

```
   requested_quantity_1      INTEGER NOT NULL,
   estimated_shipping_date_1 DATE NOT NULL,
   product_number_2          INTEGER NULL,
   product_description_2     VARCHAR ( 100 ) NULL,
   requested_quantity_2      INTEGER NULL,
   estimated_shipping_date_2 DATE NULL,
   product_number_3          INTEGER NULL,
   product_description_3     VARCHAR ( 100 ) NULL,
   requested_quantity_3      INTEGER NULL,
   estimated_shipping_date_3 DATE NULL );
```

1.16.1 第1正規形

第1正規形（1NF）では、カラムの数が可変のローと、カラム／カラム・グループの繰り返しをすべて排除します。リレーショナル・データベースでは、カラムの数が増減することは許可されませんが、異なるカラムが同じ種類のデータを格納することは可能です。order_form テーブルには、このようなデータ・グループが3つあり、製品番号と製品説明、注文数、出荷日がそれにあたります。これは、第1正規形に違反します。

カラムの繰り返しは、さまざまな問題を引き起こします。第1に、スキーマを変更せずにエントリの最大数を増やすのが困難です。第2に、カラムがそれぞれ異なる名前を持つため、複数のエントリを処理するアプリケーション・コードを作成するのが困難です。最後に、カウンタ・カラムを別に定義したり、特別な値を設けることなしに、実際に含まれるエントリ数を判断するのが困難です。この例では、NULL を使用して不足データを表しています。

この問題を解決するには、order_form を order_header と order_detail に分割し、繰り返しカラムを order_detail に移動します。order_detail 内の order_number カラムは order_header テーブルを指す外部キーで、このため order_detail が order_header の繰り返しの子になります。product_number カラムはプライマリ・キーの一部で、同じ注文を構成する各詳細ローを識別します。

```
CREATE TABLE order_header (
   order_number              INTEGER NOT NULL PRIMARY KEY,
   client_name               VARCHAR ( 100 ) NOT NULL,
   shipping_address          VARCHAR ( 1000 ) NOT NULL,
   salesperson_name          VARCHAR ( 100 ) NOT NULL,
   salesperson_phone         VARCHAR ( 100 ) NOT NULL,
   salesperson_commission    NUMERIC ( 6, 3 ) NOT NULL );

CREATE TABLE order_detail (
   order_number              INTEGER NOT NULL REFERENCES order_header,
   product_number            INTEGER NOT NULL,
   product_description       VARCHAR ( 100 ) NOT NULL,
   requested_quantity        INTEGER NOT NULL,
   estimated_shipping_date   DATE NOT NULL,
   PRIMARY KEY ( order_number, product_number ) );
```

これで、1つの注文内のorder_detailローの数が本当の意味で可変になり、人為的な制限数がなくなります。各order_detailローは、アプリケーション・プログラムのループの中で他と同じように処理され、ローの数を簡単にカウントできます。

1.16.2 第2正規形

第2正規形(2NF)は、プライマリ・キー全体ではなく、プライマリ・キーの一部にのみ依存する非キー・カラムを排除します。order_detailテーブルには2つのカラムから成るプライマリ・キー(order_number、product_number）がありますが、product_descriptionカラムはproduct_numberにのみ依存します。これは、第2正規形に違反します。

ここでの問題の1つは冗長性です。製品の説明が変更された場合、その値が格納されているすべてのorder_detailローで変更が必要です。もう1つの問題は、その製品が注文されるまで、新しい製品番号と説明を格納する場所がないことです。

この問題を解決するには、注文とは別に製品に関する情報が保存される新しいproduct_catalogテーブルに、product_descriptionを移動します。order_detailテーブルはproduct_orderになり、product_numberカラムが、新しいproduct_catalogテーブルを指す外部キーになります。

```
CREATE TABLE order_header (
   order_number              INTEGER NOT NULL PRIMARY KEY,
   client_name               VARCHAR ( 100 ) NOT NULL,
   shipping_address          VARCHAR ( 1000 ) NOT NULL,
   salesperson_name          VARCHAR ( 100 ) NOT NULL,
   salesperson_phone         VARCHAR ( 100 ) NOT NULL,
   salesperson_commission    NUMERIC ( 6, 3 ) NOT NULL );

CREATE TABLE product_catalog (
   product_number            INTEGER NOT NULL PRIMARY KEY,
   product_description       VARCHAR ( 100 ) NOT NULL );

CREATE TABLE product_order (
   order_number              INTEGER NOT NULL REFERENCES order_header,
   product_number            INTEGER NOT NULL REFERENCES product_catalog,
   requested_quantity        INTEGER NOT NULL,
   estimated_shipping_date   DATE NOT NULL,
   PRIMARY KEY ( order_number, product_number ) );
```

異なる各製品のproduct_descriptionが1箇所にだけ格納されるため、冗長性が排除されます。加えて、最初の注文を受け取る前であっても、最後の注文が削除された後であっても、製品情報を格納する場所が確保されます。

1.16.3 第3正規形

第3正規形(3NF)は、プライマリ・キーに依存しない非キー・カラムを排除します。order_header テーブルの中で、salesperson_phone カラムは salesperson_name に依存しますが、これはプライマリ・キーの一部ではありません。これは、第3正規形に違反します。

問題は、第2正規形の場合と同じです。第1の問題は冗長性です。営業部員の電話番号が変更された場合、その値が格納されているすべての注文のローを変更しなければなりません。第2の問題は、新しい営業部員に関する情報は、その営業部員が注文を受けるまで格納場所がないことです。

この問題を解決するには、営業部員のカラムを新しい salesperson テーブルに移動し、新たに salesperson_id カラムをプライマリ・キーとします。order_header テーブルは sales_order になり、新しい salesperson テーブルを指す外部キーとして salesperson_id カラムを追加します。

```
CREATE TABLE salesperson (
   salesperson_id            INTEGER NOT NULL PRIMARY KEY,
   name                      VARCHAR ( 100 ) NOT NULL,
   phone                     VARCHAR ( 100 ) NOT NULL );

CREATE TABLE sales_order (
   order_number              INTEGER NOT NULL PRIMARY KEY,
   client_name               VARCHAR ( 100 ) NOT NULL,
   shipping_address          VARCHAR ( 1000 ) NOT NULL,
   salesperson_id            INTEGER NOT NULL REFERENCES salesperson,
   salesperson_commission    NUMERIC ( 6, 3 ) NOT NULL );

CREATE TABLE product_catalog (
   product_number            INTEGER NOT NULL PRIMARY KEY,
   product_description       VARCHAR ( 100 ) NOT NULL );

CREATE TABLE product_order (
   order_number              INTEGER NOT NULL REFERENCES sales_order,
   product_number            INTEGER NOT NULL REFERENCES product_catalog,
   requested_quantity        INTEGER NOT NULL,
   estimated_shipping_date   DATE NOT NULL,
   PRIMARY KEY ( order_number, product_number ) );
```

各営業部員に関する情報が1箇所にだけ格納されるため、冗長性が排除されます。また、最初の注文を受ける前であっても、最後の注文が削除された後であっても、営業部員の情報を格納する場所が確保されます。

正規化は、データを規定するビジネス・ルールに依存します。スキーマを見るだけで、常に設計を正規化できるとは限りません。たとえば、営業部員が販売実績に応じて一定の歩合を受け取る場合は、salesperson_commission カラムも salesperson テーブルに移動します。しかし、この例の場合、歩合は注文によって変化するものとしているため、salesperson_commission は sales_order テーブルに残しま

す。

正規化は必ずしも明解ではなく、誤りを犯す恐れが常にあり、やりすぎないことが大切です。たとえば、client_name カラムも別個のテーブルにできるカラムの1つです。電話番号や請求書の宛先など、顧客関連のカラムがある場合は特にそうです。しかし、shipping_address カラムは、このようなカラムの1つではありません。このカラムは、顧客よりも注文に密接に関係します。特に、1つの顧客に複数の配送先があったり、注文の製品を第3者に発送できるような場合はそれが当てはまります。

1.16.4 ボイスコッド正規形

ボイスコッド正規形（BCNF）は、候補キーに依存しない従属カラムを排除します。候補キーは、テーブル内のローをユニークに識別する1つまたは複数のカラムです。テーブルには複数の候補キーがある可能性がありますが、プライマリ・キーとして選択されるのはその中の1つだけです。

BCNF は、3NF よりわずかに厳格です。3NF では「非キー・カラム」に注目するのに対し、BCNF では「従属カラム」に注目します。もう1つの相違は、BCNF では、プライマリ・キーだけでなく、候補キーについても注目します。

次の例では、営業部員がどのスキルを持っているかが salesperson_skill でわかります。salesperson_id と salesperson_name は、すべての営業部員についてユニークです。つまり、salesperson_name と sales_skill_id の組み合わせは、salesperson_skill の候補キーを形成します。これは、PRIMARY KEY とは別に、UNIQUE 制約として示されています。

```
CREATE TABLE sales_skill (
   sales_skill_id            INTEGER NOT NULL PRIMARY KEY,
   description               LONG VARCHAR );

CREATE TABLE salesperson_skill (
   salesperson_id            INTEGER NOT NULL,
   salesperson_name          VARCHAR ( 100 ) NOT NULL,
   sales_skill_id            INTEGER NULL REFERENCES sales_skill,
   PRIMARY KEY ( salesperson_id, sales_skill_id ),
   UNIQUE ( salesperson_name, sales_skill_id ) );
```

salesperson_skill テーブルは、第3正規形です。非キー・カラムが単にまったくないため、非キー・カラムはプライマリ・キーに依存する必要があるというルールに違反するカラムがありません。salesperson_skill 内のすべてのカラムは、いずれかの候補キーの一部です。

しかし、salesperson_skill は、ボイスコッド正規形ではありません。salesperson_name と salesperson_id は相互に依存し、どちらのカラムも単独では候補キーを形成しないためです。この問題を解決するには、違反しているカラム、つまり salesperson_id と salesperson_name のいずれかを salesperson テーブルに移動します。

```
CREATE TABLE salesperson (
   salesperson_id            INTEGER NOT NULL PRIMARY KEY,
   salesperson_name          VARCHAR ( 100 ) NOT NULL UNIQUE );

CREATE TABLE sales_skill (
   sales_skill_id            INTEGER NOT NULL PRIMARY KEY,
   description               LONG VARCHAR );

CREATE TABLE salesperson_skill (
   salesperson_id            INTEGER NOT NULL REFERENCES salesperson,
   sales_skill_id            INTEGER NULL REFERENCES sales_skill,
   PRIMARY KEY ( salesperson_id, sales_skill_id ) );
```

実際問題として、第3正規形とボイスコッド正規形の違いを説明するのは困難です。テーブルを第3正規形に変換すると、定義に多少の違いはありますが、すべての冗長性が排除されるため、ボイスコッド正規形にもなる可能性が十分にあります。

実際、第3正規形のデータベース設計は、第4正規形と第5正規形になる可能性もあります。次の3つの項では、第4正規形と第5正規形が第3正規形とは異なるまれな状況について説明します。

1.16.5 第4正規形

第4正規形（4NF）は、同じテーブル内で別個に存在する複数の多対多関係を排除します。次に示す例では、salesperson_skillテーブルが2つの多対多関係を表しています。1つ目は、1人の営業部員が複数の販売スキルを持つという関係で、これは逆に、1つの販売スキルを複数の営業部員で共有することもできます。2つ目は、営業部員と技術スキルの多対多の関係です。この2つの多対多関係は独立しています。営業部員の技術スキルと販売スキルは、この設計に関する限り、相互に依存しません。

```
CREATE TABLE salesperson (
   salesperson_id            INTEGER NOT NULL PRIMARY KEY,
   salesperson_name          VARCHAR ( 100 ) NOT NULL );

CREATE TABLE sales_skill (
   sales_skill_id            INTEGER NOT NULL PRIMARY KEY,
   description               LONG VARCHAR );

CREATE TABLE technical_skill (
   technical_skill_id        INTEGER NOT NULL PRIMARY KEY,
   description               LONG VARCHAR );

CREATE TABLE salesperson_skill (
   salesperson_id            INTEGER NOT NULL REFERENCES salesperson,
   sales_skill_id            INTEGER NOT NULL REFERENCES sales_skill,
   technical_skill_id        INTEGER NOT NULL REFERENCES technical_skill,
   PRIMARY KEY ( salesperson_id, sales_skill_id, technical_skill_id ) );
```

この例では、営業部員が異なる数の販売スキルと技術スキルを持つ場合に、salesperson_skill 内のローをどう設定すべきかがはっきりしません。欠如しているスキルに対しては、特別な「空白」値を使用すべきでしょうか。関係の途切れたローに、販売スキルと技術スキルのいずれかを設定すべきでしょうか。販売スキルと技術スキルのすべての組み合わせから成る直積を作成すればよいでしょうか。どの方法も、冗長性が生じたり、更新するには複雑なルールが必要になるなど、問題を抱えています。

この問題を解決するには、次に示すように、salesperson_skill を 2 つの個別のテーブルに置き換えます。

```
CREATE TABLE salesperson_sales_skill (
   salesperson_id            INTEGER NOT NULL REFERENCES salesperson,
   sales_skill_id            INTEGER NOT NULL REFERENCES sales_skill,
   PRIMARY KEY ( salesperson_id, sales_skill_id ) );

CREATE TABLE salesperson_technical_skill (
   salesperson_id            INTEGER NOT NULL REFERENCES salesperson,
   technical_skill_id        INTEGER NOT NULL REFERENCES technical_skill,
   PRIMARY KEY ( salesperson_id, technical_skill_id ) );
```

異なる多対多の関係が異なるテーブルで表されるため、これらのテーブルは第 4 正規形となります。

1.16.6 第5正規形

第 5 正規形（5NF）は、新しいテーブルのプライマリ・キーが小さく、冗長性が低く、ジョインして元の状態に再構築できる場合に、1 つのテーブルを 3 つ以上に分割します。これは、1 つのテーブルを 2 つに分割する他の正規形とは異なります。

次に示す例では、業者の製品ラインとそれを担当する営業部員に関する情報が salesperson_company_line に格納されています。ここでは、特別なビジネス・ルールが適用されます。ある営業部員がある製品ラインを担当し、業者がその製品ラインを製造する場合、その営業部員はその業者が製造した製品ラインを担当します。これは、個々の多対多関係が独立していない 3 方向の関係であり、salesperson_company_line は第 4 正規形です。

```
CREATE TABLE salesperson (
   salesperson_id            INTEGER NOT NULL PRIMARY KEY,
   salesperson_name          VARCHAR ( 100 ) NOT NULL );

CREATE TABLE company (
   company_id                VARCHAR ( 10 ) NOT NULL PRIMARY KEY,
   company_name              VARCHAR ( 100 ) NOT NULL );

CREATE TABLE product_line (
   product_line_id           VARCHAR ( 10 ) NOT NULL PRIMARY KEY,
   product_line_description  VARCHAR ( 100 ) NOT NULL );
```

```
CREATE TABLE salesperson_company_line (
   salesperson_id            INTEGER NOT NULL REFERENCES salesperson,
   company_id                VARCHAR ( 10 ) NOT NULL REFERENCES company,
   product_line_id           VARCHAR ( 10 ) NOT NULL REFERENCES product_line,
   PRIMARY KEY ( salesperson_id, company_id, product_line_id ) );
```

この場合、salesperson_company_lineが冗長な値を格納する必要があるだけでなく、次のローで特別なビジネス・ルールに違反する可能性があります。

```
INSERT salesperson_company_line VALUES ( 1, 'Acme', 'cars' );
INSERT salesperson_company_line VALUES ( 2, 'Acme', 'trucks' );
INSERT salesperson_company_line VALUES ( 2, 'Best', 'cars' );
```

1番目のローはsalesperson 1が対象で、Acmeが乗用車(cars)を作成することを示します。2番目のローは、salesperson 2もAcmeを担当することを示します。ただし、この場合はトラック（trucks）です。3番目のローは、salesperson 2がBestの乗用車を担当することを示します。そうすると、salesperson 2がAcmeの乗用車を担当することを示すローはどこにあるのでしょうか。

salesperson_company_lineテーブルは、次の3つのテーブルに分割できるため（多くの場合分割する必要があるため)、第5正規形ではありません。

```
CREATE TABLE salesperson_company (
   salesperson_id            INTEGER NOT NULL REFERENCES salesperson,
   company_id                VARCHAR ( 10 ) NOT NULL REFERENCES company,
   PRIMARY KEY ( salesperson_id, company_id ) );

CREATE TABLE company_line (
   company_id                VARCHAR ( 10 ) NOT NULL REFERENCES company,
   product_line_id           VARCHAR ( 10 ) NOT NULL REFERENCES product_line,
   PRIMARY KEY ( company_id, product_line_id ) );

CREATE TABLE salesperson_line (
   salesperson_id            INTEGER NOT NULL REFERENCES salesperson,
   product_line_id           VARCHAR ( 10 ) NOT NULL REFERENCES product_line,
   PRIMARY KEY ( salesperson_id, product_line_id ) );
```

これらの3つの新しいテーブルにデータを挿入する方法を次に示します。SELECTで元のテーブルを再構築すると、salesperson 2が、実際にはAcmeの乗用車を担当することを示すローが含まれます。

```
INSERT salesperson_company VALUES ( 1, 'Acme' );
INSERT salesperson_company VALUES ( 2, 'Acme' );
INSERT salesperson_company VALUES ( 2, 'Best' );
INSERT company_line VALUES ( 'Acme', 'cars' );
INSERT company_line VALUES ( 'Acme', 'trucks' );
```

```
INSERT company_line VALUES ( 'Best', 'cars' );
INSERT salesperson_line VALUES ( 1, 'cars' );
INSERT salesperson_line VALUES ( 2, 'cars' );
INSERT salesperson_line VALUES ( 2, 'trucks' );

SELECT DISTINCT
       salesperson_company.salesperson_id,
       company_line.company_id,
       salesperson_line.product_line_id
  FROM salesperson_company
  JOIN company_line
    ON salesperson_company.company_id = company_line.company_id
  JOIN salesperson_line
    ON salesperson_company.salesperson_id = salesperson_line.salesperson_id
   AND company_line.product_line_id = salesperson_line.product_line_id;
```

第5正規形にするために特別な作業を要するテーブルは、めったにありません。この例でも、特別なビジネス・ルールがなければ、元の salesperson_company_line テーブルは、salesperson、company、product line…との間に3方向の多対多関係を実装しているため、正しい選択であり、既に第5正規形です。ほとんどの場合、第3正規形に達すると、ボイスコッド正規形、第4正規形、第5正規形にも達しています。

Section

1.17 この章のまとめ

この章では、SQL Anywhere 9でサポートされている5つの異なるタイプのテーブル（グローバル永続、グローバル・テンポラリ、ローカル・テンポラリ、リモート、プロキシ）の作成方法について説明しました。また、基本のカラム・データ型、COMPUTE や DEFAULT などのカラム・プロパティ、CHECK、PRIMARY KEY、外部キー、UNIQUE などのカラム制約とテーブル制約についても説明しました。リレーショナル・データベースの 12 のルールと、適切なデータベース設計の 6 種類の正規形について説明しました。

次の章では、データベースのライフサイクルの 2 番目のステップであるローの挿入について説明します。

Chapter

2

挿入

Section

2.1 概要

リレーショナル・データベースのライフサイクルの2番目のステップは、データベース・テーブルへのデータの挿入です。SQL Anywhereには、INSERT文、LOAD TABLE文、ISQL INPUT文の3つの異なる方法があります。

INSERT文には、カラムごとにVALUESリストの値を明示的に指定して、INSERT 1回につき1つのローを挿入するタイプと、1回のINSERTで他のソースからテーブルにロー・セット全体をコピーするタイプがあります（ここで、「他のソース」とはSELECTで生成できるすべてのソースです）。

この章では、値をターゲット・テーブルのカラムの一部またはすべてに入力するか、あるいはAUTO NAME機能を使用するかによって、INSERTの2つのタイプをさらに5つの形式に分けて説明します。

外部ファイルからテーブルにデータを挿入する方法としては、LOAD TABLEとISQL INPUTの2つがあります。

この章では、これらの方法に用意されている各種の機能について解説します。たとえば、ON EXISTING UPDATE句を使えば、プライマリ・キーが競合する場合にINSERTをUPDATEに変換できます。LOAD TABLE文では、パフォーマンスを向上させるためにショートカットが行われます。ISQL INPUT文を使えば、レガシー・ソースから固定レイアウト・レコードやその他のファイル・フォーマットをロードできます。

Section

2.2 INSERT

INSERT文には、次の5つのフォーマットがあります。

- すべてのカラムの値を指定するVALUESリストを使って、1つのローを挿入する
- カラム名リストと、それに対応するVALUESリストを使って、1つのローを挿入する
- すべてのカラムの値を取り出すSELECTを使って、複数のローを挿入する
- カラム名リストと、それに対応するSELECTを使って、複数のローを挿入する
- カラム名リストの代わりにWITH AUTO NAME句を使って、複数のローを挿入する

以降では、この各フォーマットについて順に見ていきます。

2.2.1 すべてのカラムへのINSERT

INSERT 文の最も簡単な形式は、すべてのカラムの値を指定して1つのローを挿入する文です。

```
<insert_all_values> ::= INSERT [ INTO ]
                          [ <owner_name> "." ] <target_table_name>
                          [ <on_existing> ]
                          VALUES "(" <all_values_list> ")"
<owner_name>        ::= <identifier>
<target_table_name> ::= <identifier>
<identifier>        ::=「第1章　作成」の<identifier>を参照
<on_existing>       ::= ON EXISTING ERROR  -- デフォルト
                      | ON EXISTING UPDATE
                      | ON EXISTING SKIP
<all_values_list>   ::= <value_list> -- テーブル内のすべてのカラムに対応
<value_list>        ::= <value> { "," <value> }
<value>             ::= <expression>
                      | DEFAULT
<expression>        ::=「第3章　選択」の<expression>を参照
```

Note

ビューが更新可能ビューであり、1つのテーブルしか含まない場合は、そのビューにローを挿入できます。更新可能ビューの詳細については、「3.23　CREATE VIEW」を参照してください。

VALUES リスト内の式の順序は、その式が適用される CREATE TABLE のカラムの順序と同じでなければなりません。また、テーブルのすべてのカラムの値を指定しなければなりません。

Note

既存のテーブルにカラムを追加する場合は、ALTER TABLE コマンドを使用します。新しいカラムは、CREATE TABLE に最初に定義されたすべてのカラムと、以前の ALTER TABLE コマンドで追加されたその他のカラムの後ろに挿入されます。「CREATE TABLE のカラムの順序」という場合は、「CREATE TABLE に最初にリストされたカラムと、後で ALTER TABLE コマンドで追加されたカラムからなる順序」と解釈してください。各種の ALTER コマンドは非常に強力で便利ですが、紙面上の理由から本書では説明を割愛します。

次の例では、key_1 に値1が挿入され、non_key_1 に 'first row' が挿入され、last_updated に '2003 09 29 13:21' が挿入されます。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   non_key_1     VARCHAR ( 100 ) NOT NULL,
   last_updated  TIMESTAMP NOT NULL DEFAULT TIMESTAMP,
   PRIMARY KEY ( key_1 ) );

INSERT t1 VALUES ( 1, 'first row', '2003 09 29 13:21' );
```

Tip

テーブルt1のカラムの順序を確認するには、ISQLでコマンド「SELECT * FROM t1 WHERE 1 = 0」を実行します。データを取り出さずにカラム名のみが表示されます。パフォーマンスの心配は無用です。データベース・エンジンは、「WHERE 1 = 0」を、ローを戻さないという意味と認識するので、テーブルの大きさに関係なくこのクエリは短時間で実行されます。

VALUESリストは、DEFAULTという特殊キーワードをサポートしています。このキーワードは、DEFAULTの値を使用したい場合に明示的な値の代わりとして使用できます。次の例では、key_1にDEFAULT AUTOINCREMENTの値2が挿入され、non_key_1に'second row'が挿入され、last_updatedにDEFAULT TIMESTAMPのカレント日時が挿入されます。

```
INSERT t1 VALUES ( DEFAULT, 'second row', DEFAULT );
```

CREATE TABLEで2番目のカラムが明示的なDEFAULTの値なしでNOT NULLとして宣言されているので、このテーブルの2番目のカラムにDEFAULTキーワードを使用することはできません。DEFAULTのデフォルトはNULLですが、このカラムにはNULLを入れることができないため、このキーワードは機能しません。つまり、「INSERT t1 VALUES (DEFAULT, DEFAULT, DEFAULT)」は失敗し、「テーブル't1'のカラム'non_key_1'はNULL値を持つことはできません」というエラーが発生します。

Tip

INSERTのVALUESリストでは、クライアント・アプリケーションからのINSERTの場合でも、CURRENT TIMESTAMP、CURRENT TIMEなどの特殊リテラルを使用できます。SQL Anywhereデータベース・エンジンは、ローが挿入されるときに実際の値を入力します。特殊リテラルの詳細については、「1.8.3　リテラル・デフォルト」を参照してください。

3つのON EXISTING句では、1つのテーブルに同じプライマリ・キー値を持つローが既に存在する場合の動作を指定できます。これは、テーブルに明示的なPRIMARY KEYがなければならないことを意味します。PRIMARY KEYがUNIQUE制約で置き換えられていたとしても、そのテーブルでON

EXISTING 句を使用することはできません。

デフォルトは ON EXISTING ERROR で、「テーブル 't1' のプライマリ・キーがユニークではありません」というお馴染みのエラーが発生し、挿入が拒否されます。

ON EXISTING SKIP 句では、エラーは発生せず、INSERT が無視されます。これは、挿入するローが重複していて、単にその重複するローを無視したい場合に便利です。ON EXISTING UPDATE 句は、エラーが発生せず、INSERT が UPDATE に変換されます。これは、挿入するローが重複していて、古い値を新しい値で上書きしたい場合に便利です。

次の例では、3 つの INSERT 文すべてで同じプライマリ・キー値が指定されていますが、エラーは発生せず、すべての文が実行されます。実行が終了したときには、カラム non_key_1 に 'replaced' という値が入力された 1 つのローのみが存在します。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   non_key_1     VARCHAR ( 100 ) NOT NULL,
   last_updated  TIMESTAMP NOT NULL DEFAULT TIMESTAMP,
   PRIMARY KEY ( key_1 ) );

INSERT t1 VALUES ( 1, 'first row', DEFAULT );
INSERT t1 ON EXISTING UPDATE VALUES ( 1, 'replaced', DEFAULT );
INSERT t1 ON EXISTING SKIP  VALUES ( 1, 'ignored', DEFAULT );
```

ON EXISTING 句は、この章で説明するすべての形式の INSERT 文で使用できます。また、「7.6.4.5 アップロード・エラーの処理」でも、アップロードされるデータでプライマリ・キーの競合を避ける方法として紹介します。

2.2.2 指定のカラムへのINSERT

VALUES リストは使用したいが、すべての値をリストする必要がない場合や、値の順序を変更したい場合は、INSERT にカラム名リストを追加できます。

```
<insert_named_values> ::= INSERT [ INTO ]
                            [ <owner_name> "." ] <target_table_name>
                            "(" <column_name_list> ")"
                            [ <on_existing> ]
                            VALUES "(" <named_values_list> ")"
<column_name_list>    ::= <column_name> { "," <column_name> }
<column_name>         ::= <identifier>
<named_values_list>   ::= <value_list> -- 指定されたカラムに対応
```

このタイプの INSERT を使用する場合、VALUES リスト内の式は、カラム名リストに指定されている順にカラムに適用されるので、値の数とカラム名の数を一致させる必要があります。指定のないカラ

ムについては、そのカラムのデフォルト値が割り当てられます。

Note

テーブルを作成するときに明示的なDEFAULT句をコーディングしているかどうかに関係なく、すべてのカラムにデフォルト値があります。DEFAULTのデフォルトはNULLです。

順序を変更して、col_2に'A'を、col_3に'B'を挿入し、その他のすべてのカラムにデフォルト値を割り当てる例を次に示します。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   col_2         VARCHAR ( 100 ) NOT NULL DEFAULT 'X',
   col_3         VARCHAR ( 100 ) NOT NULL DEFAULT 'Y',
   updated_by    VARCHAR ( 128 ) NOT NULL DEFAULT LAST USER,
   last_updated  TIMESTAMP NOT NULL DEFAULT TIMESTAMP,
   PRIMARY KEY ( key_1 ) );

INSERT t1 ( col_3, col_2 ) VALUES ( 'B', 'A' );
```

このCREATE TABLEでは、すべてのカラムにDEFAULT値を指定しています。したがって、次の挿入は正常に機能し、エラーが発生せずに繰り返し実行できます。

```
INSERT t1 ( key_1 ) VALUES ( DEFAULT );
```

ただし、「INSERT t1() VALUES()」と記述することはできません。

Note

SQL Anywhereに希望する機能がある場合は、遠慮なくご意見をお寄せください。INSERT文にON EXISTINGが追加されたのは、forums.sybase.comのNNTPサーバにあるsybase.public.sqlanywhere.product_futures_discussionというパブリック・ニュースグループにリクエストが投稿されたためです。このニュースグループには、Forte AgentなどのNNTPクライアント・ソフトウェアを使って投稿できます。または、Webブラウザを使ってhttp://www.ianywhere.com/developerにアクセスし、[Newsgroups]リンクをクリックしてください。すべての意見が取り入れられるわけではありませんが、製品開発担当者がすべての意見を慎重に検討します。
アイエニウェア・ソリューションズ社が管理・運営している日本語のニュースグループもあります(http://www.ianywhere.jp/developers/newsgroups.html を参照してください)。

2.2.3　selectを使用したすべてのカラムへのINSERT

どこか他の場所にあるデータをコピーしてテーブルに挿入する場合は、VALUESリストの代わりにselectを使用できます。最も簡単な形式では、テーブルのすべてのカラムに値を提供するselectを使用します。

```
<insert_select_all_columns> ::= INSERT [ INTO ]
                                [ <owner_name> "." ] <target_table_name>
                                [ <on_existing> ]
                                <select_all_columns>
<select_all_columns> ::= <select> -- ターゲット・テーブル内のすべてのカラムに対する値
<select>             ::=「第3章　選択」の<select>を参照
```

selectを使用するINSERT文には、VALUESリストを使用する文よりも便利な点が2つあります。まず、1つは、INSERTで複数のローを挿入できることです。もう1つは、値を明示的に指定せずにデータを挿入できることです。

次に示すのは、t1のすべてのローとカラムをt2にコピーする例です。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   non_key_1     VARCHAR ( 100 ) NOT NULL DEFAULT 'xxx',
   last_updated  TIMESTAMP NOT NULL DEFAULT TIMESTAMP,
   PRIMARY KEY ( key_1 ) );

CREATE TABLE t2 (
   key_1         INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   non_key_1     VARCHAR ( 100 ) NOT NULL DEFAULT 'xxx',
   last_updated  TIMESTAMP NOT NULL DEFAULT TIMESTAMP,
   PRIMARY KEY ( key_1 ) );

INSERT t2
SELECT key_1, non_key_1, last_updated
  FROM t1;
```

この2つのテーブルのカラムの数と順序はまったく同じなので、INSERTはもっと簡単に指定できます。

```
INSERT t2 SELECT * FROM t1;
```

Tip

この形式のINSERTは、プロキシ・テーブルを介して外部ソースからデータをロードする場合によく使われます（たとえば、「INSERT local_table SELECT * FROM proxy_table」など）。プロキシ・テーブルの詳細については、「1.14　リモート・データ・アクセス」を参照してください。

selectを使用するINSERT文には、守らなければならない規則がいくつかあります。selectが戻すカラムは、ターゲット・テーブルのカラムの数と同じでなければなりません。また、ターゲット・テーブルに対するCREATE TABLEのカラムと同じ順序でなければならず、さらに、同一のデータ型か互換性のあるデータ型でなければなりません。つまり、結果セットが一致した場合に挿入が行われるということです。

この規則に従う限り、「第3章　選択」で説明する高度な機能をINSERTですべて使用できます。次に示すのは、t1から選択したローに、UNIONを使ってさらに2つのローを追加する例です。

```
INSERT t2
SELECT 0, 'first', '2001-01-01'
UNION
SELECT * FROM t1
 WHERE key_1 BETWEEN 1 AND 9998
UNION
SELECT 9999, 'last', CURRENT TIMESTAMP;
```

Note

selectには、単なるSELECT以外の意味があります。本書では、結果セットを戻すクエリを表す場合に小文字の"select"を使用しています。すべてのクエリ、つまりselectは、少なくとも1つの"SELECT"キーワード（本書では大文字で表記）を含みます。ただし、前述の例のように、1つのselectに複数のSELECTが含まれていることもあります。クエリの詳細については、「第3章　選択」を参照してください。

INSERT文内のSELECTからの結果セットが完全に実体化されてから、ローが挿入されます。ターゲット・テーブル自体がSELECTのいずれかの句で指定されている場合は、テーブルに既にあるローのみがSELECTで生成される結果セットに影響します。次の例でそのポイントを示します。最後のINSERT文で、t2.non_key_1からt1.key_1に値をコピーし、WHERE句で、まだ存在しない値のみを選択することを指定します。これは、重複する値をt1.key_1に挿入しないようにする試みです。値1はt1.key_1に既にあるので機能しますが､値2はINSERTの実行前にt1.key_1に存在せず､t2に値2のローが2つあるため、文の実行は失敗し、「テーブル't1'のプライマリ・キーがユニークではありません」というエラーが発生します。

```
CREATE TABLE t1 (
   key_1     INTEGER NOT NULL PRIMARY KEY );

INSERT t1 VALUES ( 1 );

CREATE TABLE t2 (
   key_1     VARCHAR ( 10 ) NOT NULL PRIMARY KEY,
   non_key_1 INTEGER NOT NULL );
```

```
INSERT t2 VALUES ( 'A', 1 );
INSERT t2 VALUES ( 'B', 2 );
INSERT t2 VALUES ( 'C', 2 );

INSERT t1
SELECT t2.non_key_1
  FROM t2
 WHERE NOT EXISTS ( SELECT *
                      FROM t1
                     WHERE t1.key_1 = t2.non_key_1 );
```

2.2.4 selectを使用した指定のカラムへのINSERT

selectは使用したいが、すべてのカラムをリストする必要がない場合や、カラムの順序を変更したい場合は、INSERTにカラム名リストを指定できます。

```
<insert_select_column_list> ::= INSERT [ INTO ]
                                  [ <owner_name> "." ] <target_table_name>
                                  "(" <column_name_list> ")"
                                  [ <on_existing> ]
                                  <select_column_list>
<select_column_list>        ::= <select> -- 指定されたカラムに対する値
```

このタイプのINSERTを使用する場合、selectから戻される値は、カラム名リストで指定されている順にカラムに適用されます。selectが戻すカラムは、カラム名リストのカラムと同じ数、同じ順序でなければならず、また同一のデータ型か互換性のあるデータ型でなければなりません。

データは、スキーマが異なる場合でもテーブル間でコピーできます。次は、t1.key_1をVARCHARに変換してt2.col_3に挿入し、t1.non_key_1をt2.col_2にコピーし、t2のその他のすべてのカラムにデフォルト値を割り当てる例です。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL,
   non_key_1     VARCHAR ( 100 ) NOT NULL,
   PRIMARY KEY ( key_1 ) );

CREATE TABLE t2 (
   key_col       INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   col_2         VARCHAR ( 100 ) NOT NULL,
   col_3         VARCHAR ( 100 ) NOT NULL,
   updated_by    VARCHAR ( 128 ) NOT NULL DEFAULT LAST USER,
   last_updated  TIMESTAMP NOT NULL DEFAULT TIMESTAMP,
   PRIMARY KEY ( key_col ) );
```

```
INSERT t2 ( col_3, col_2 )
SELECT key_1, non_key_1
  FROM t1;
```

Tip

異なるデータ型のカラムに値を挿入するときに、暗黙的に行われるデータ変換によって発生する問題に注意してください。最大長が短いカラムに長い文字列を挿入すると、通知なしで切り捨てが発生することがあります。また、NUMERIC データ型と FLOAT データ型の間で変換する場合は精度に関する問題が発生します。さらに、データ型が異なる値を含む式を計算してその結果を挿入する場合は、データ型に関する問題が発生します。たとえば、「1 + 32767」という式を SMALLINT 型のカラムに挿入すると、-32768 として格納されます。

2.2.5 selectを使用し、カラム名を自動的に設定するINSERT

WITH AUTO NAME 句では、INSERT からカラム名リストを省き、select を使って一部のカラムを除外したり、カラムの指定順序を変更したりすることができます。ターゲット・テーブルのカラム名は、select から戻された値と自動的に照合されます。つまり、select から戻される各値には、カラム名またはエイリアス名のいずれかの名前がなければならないと同時に、その名前がターゲット・テーブルのカラム名と一致していなければなりません。

```
<insert_select_auto_name> ::= INSERT [ INTO ]
                              [ <owner_name> "." ] <target_table_name>
                              [ <on_existing> ]
                              WITH AUTO NAME
                              <select_auto_name>
<select_auto_name> ::= <select> -- ターゲット・カラムに一致する名前またはエイリアスを持つ
```

次の例は、SELECT でエイリアス名を使って t2 の col_2、col_3、col_4 の値を指定する方法を示します。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL,
   non_key_1     VARCHAR ( 100 ) NOT NULL,
   PRIMARY KEY ( key_1 ) );

CREATE TABLE t2 (
   key_col       INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   col_2         VARCHAR ( 100 ) NOT NULL,
   col_3         VARCHAR ( 100 ) NOT NULL,
   col_4         VARCHAR ( 100 ) NOT NULL,
```

```
   PRIMARY KEY ( key_col ) );

INSERT t2 WITH AUTO NAME
SELECT key_1             AS col_3,
       non_key_1         AS col_2,
       CURRENT TIMESTAMP AS col_4
  FROM t1
 WHERE key_1 > 1;
```

WITH AUTO NAME 句は、テーブル間でデータを単にコピーするためだけでなく、VALUES リストの代用とすることもできます。WITH AUTO NAME 句のほうが、対応関係がわかりやすいかもしれません。たとえば、次の形式の INSERT では、どの値をどのカラムに挿入するかがわかりにくく、またすべてのカラム値を指定しなければならないため、この形式を好まない人もいます。

```
INSERT t1 VALUES ( DEFAULT, 'aaa', 1, 'bbb', 2 );
```

次の代替方法では、デフォルトを使用するカラムを省くことで問題の1つは解決されます。しかし、テーブルが大きい場合、どの値をどこに挿入するかわかりにくいという問題は相変わらず解決しません。

```
INSERT t1 ( non_key_1,
            non_key_2,
            non_key_3,
            non_key_4 )
   VALUES ( 'aaa',
            1,
            'bbb',
            2 );
```

WITH AUTO NAME 句によって、ターゲットのカラム名を対応する値と並べてコーディングすると、こういった問題も解決できます。

```
INSERT t1 WITH AUTO NAME
SELECT 'aaa' AS non_key_1,
       1     AS non_key_2,
       'bbb' AS non_key_3,
       2     AS non_key_4;
```

Section

2.3 LOAD TABLE

LOAD TABLE 文は、フラット・ファイルからデータベース・テーブルにデータをコピーする非常に効率の良い方法です。

```
<load_table>       ::= LOAD [ INTO ] TABLE
                         [ <owner_name> "." ] <target_table_name>
                         [ "(" <input_name_list> ")" ]
                         FROM <load_filespec>
                         [ <load_option_list> ]
<input_name_list>  ::= <input_name> { "," <input_name> }
<input_name>       ::= <column_name> -- ターゲット・テーブル内
                     | "filler()"    -- 入力フィールドを無視するため
<load_filespec>    ::= サーバを基準とする文字列リテラルのファイル指定
<load_option_list> ::= <load_option> { <load_option> }
<load_option>      ::= CHECK CONSTRAINTS ( ON | OFF )      -- デフォルトは ON
                     | COMPUTES ( ON | OFF )               -- デフォルトは ON
                     | DEFAULTS ( ON | OFF )               -- デフォルトは OFF
                     | DELIMITED BY <load_delimiter>       -- デフォルトは ','
                     | ESCAPE CHARACTER <escape_character> -- デフォルトは '¥'
                     | ESCAPES ( ON | OFF )                -- デフォルトは ON
                     | FORMAT ( ASCII | BCP )              -- デフォルトは ASCII
                     | HEXADECIMAL ( ON | OFF )            -- デフォルトは ON
                     | ORDER ( ON | OFF )                  -- デフォルトは ON
                     | PCTFREE <free_percent>
                     | QUOTES ( ON | OFF )                 -- デフォルトは ON
                     | STRIP ( ON | OFF )                  -- デフォルトは ON
                     | WITH CHECKPOINT ( ON | OFF )        -- デフォルトは OFF
<load_delimiter>   ::= 長さが 1 ～ 255 文字の文字列リテラル
<escape_character> ::= 長さが 1 文字のみの文字列リテラル
<free_percent>     ::= 0 ～ 100 の範囲の整数リテラル
```

ターゲット・テーブルの名前は必須ですが、入力名リストはオプションです。入力名リストを省略した場合、入力ファイルのレイアウトはテーブルのレイアウトと一致する（つまり、テーブルのカラムと同じ数の入力フィールドがあり、入力フィールドは各入力レコード内で左から右に、CREATE TABLE のカラムと同じ順序で配置されている）と見なされます。

デフォルトの入力ファイル・フォーマットは、各行が個々のローを表す、カンマで区切られた ASCII テキストです。次に示すのは、簡単な LOAD TABLE 文の例です。ファイル指定では、1 つの円記号を 2 つ 1 組の円記号で表すので、エスケープ文字（¥）の解釈に関する問題はありません。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL,
   col_2         VARCHAR ( 100 ) NULL,
   col_3         DECIMAL ( 11, 2 ) NULL,
   col_4         TIMESTAMP NULL,
   col_5         INTEGER NOT NULL,
   PRIMARY KEY ( key_1 ) );

LOAD TABLE t1 FROM 'c:¥¥temp¥¥t1_a.txt';
```

t1_a.txt ファイルには、次の 4 行のデータが入っています。

```
1,'Hello, World',67.89,2003-09-30 02:15PM,999
2,   stripped string without quotes    ,  0  ,  ,  0
3,,,,
4,"    double quoted padded string   ",0,2003 9 30,-111
```

LOAD TABLE の実行後、t1 の 4 つのローは次のようになります。

```
key_1 col_2                                 col_3 col_4                   col_5
===== ===================================== ===== ====================== =====
1     'Hello, World'                        67.89 2003-09-30 14:15:00.000 999
2     'stripped string without quotes'      0.00  NULL                    0
3     NULL                                  NULL  NULL                    0
4     '    double quoted padded string   ' 0.00  2003-09-30 00:00:00.000 -111
```

入力名リストを使用する目的は 3 つあります。入力フィールドをテーブルのカラムに適用する順序を変更するため、対応する入力フィールドがないカラムを省略するため、そしてカラムにコピーしない入力フィールドを省略するためです。

- 順序を変更する場合は、対応する入力フィールドがファイル内で実際に現れる順にカラム名をコーディングする
- テーブルにあるカラムを省略する場合は、リストからそのカラム名を省く
- 入力フィールドを省略する場合は、入力名リストのその位置に "filler()" を使用する

次は、明示的な入力名リストを使用する LOAD TABLE の例です。2 番目の入力フィールドは無視され、3 番目、4 番目、5 番目の入力フィールドは、それぞれ col_4、col_3、col_2 に適用され、col_5 に対応する入力フィールドはありません。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL,
   col_2         INTEGER NOT NULL,
```

```
   col_3         TIMESTAMP NULL,
   col_4         DECIMAL ( 11, 2 ) NULL,
   col_5         VARCHAR ( 100 ) NULL,
   PRIMARY KEY ( key_1 ) );

LOAD TABLE t1 ( key_1, filler(), col_4, col_3, col_2 ) FROM 'c:¥¥temp¥¥t1_b.txt';
```

入力ファイルに次のレコードが入っている場合、

```
1, 'Hello, World', 67.89, 2003-09-30 02:15PM, 999
```

t1 に挿入されるローは次のようになります。

```
key_1 col_2 col_3                    col_4 col_5
===== ===== ======================== ===== =====
1     999   '2003-09-30 14:15:00.000' 67.89 NULL
```

LOAD TABLE の入力ファイル指定は、クライアントではなくサーバが基準となります。もっと具体的に言えば、ドライブとパスは、SQL Anywhere データベース・エンジンが起動された時点のカレント・フォルダが基準となります。これは、デスクトップ上の ISQL から LOAD TABLE を実行しているが、データベース・サーバはネットワーク上のどこか他の場所にある、というときに問題となります。入力ファイルが置かれている同じコンピュータで LOAD TABLE 文を実行した場合でも、LOAD TABLE 文が実行されるのはサーバ上であり、ファイルを検索するのは ISQL ではなく、データベース・サーバです。

LOAD TABLE では、ネットワーク・ファイルに対する UNC（Universal Naming Convention）を使用できます。次は、UNC ファイル指定のレイアウトです。

```
¥¥<server-name>¥<share-name>¥<directory/filename>
```

たとえば、ファイル C:¥temp¥t1_c.txt がマシン TECRA にあり、C: ドライブを "TecraC" として共有している場合、次の LOAD TABLE を使ってネットワークを検索し、ファイルを ¥¥TECRA¥TecraC¥temp¥t1_c.txt として読み取ることができます。前にも説明したように、1 つの円記号（¥）は、ファイル指定では 2 つ 1 組の円記号で表します。

```
LOAD TABLE t1 FROM '¥¥¥TECRA¥¥TecraC¥¥temp¥¥t1_c.txt';
```

データベース・エンジンを実行しているコンピュータを基準として見たとき、ファイルが上記の場所に存在するのであれば、ISQL がどのマシンで実行されているかに関係なく、このようにファイル指定をコーディングする必要があります。

Tip

LOAD TABLEでファイルを指定するときは、オペレーティング・システムのパーミッションに注意してください。たとえば、データベース・エンジンがWindowsサービスとして実行されている場合は、ファイルを読み取るための十分な権限があるかどうかが問題になります。

LOAD TABLEの入力ファイル指定は、変数ではなく、文字列リテラルでなければなりません。変数を使用する場合は、EXECUTE IMMEDIATEを介してLOAD TABLEを実行します。次の例では、変数@filespecにファイル指定を入れ、別の変数@sqlでLOAD TABLEコマンドを作成してから、EXECUTE IMMEDIATEを介してそのコマンドを実行します。この方法では、残念ながら、¥¥TECRA¥TecraC¥temp¥t1_c.txtにある1つの各円記号を4つの円記号で表さなければなりません。

```
BEGIN
DECLARE @filespec VARCHAR ( 1000 );
DECLARE @sql      VARCHAR ( 1000 );
SET @filespec = '¥¥¥¥¥¥¥¥TECRA¥¥¥¥TecraC¥¥¥¥temp¥¥¥¥t1_c.txt';
SET @sql = STRING ( 'LOAD TABLE t1 FROM ''', @filespec, '''' );
EXECUTE IMMEDIATE @sql;
END;
```

エスケープ文字（¥）は文字列の解析ごとに処理されます。前述の例で、2つ1組の円記号（¥¥）はそれぞれ、SET @filespecが文字列リテラルを解析する場合に1回、EXECUTE IMMEDIATEが@sql内のコマンドを解析する場合に1回の合計2回、その処理の一環として1つの円記号に変換されます。つまり、LOAD TABLEが実際に実行するまで残しておきたい1つの各円記号（¥）は、元の文字列リテラル内では4つの円記号で表し、したがって円記号を2つ残したい場合は、8つの円記号をコーディングします。

LOAD TABLEの動作は、次のようにオプションの拡張リストで制御します。

- **CHECK CONSTRAINTS OFF** —— LOAD TABLEで新しいローを挿入するときに、CHECK制約を無効にする。デフォルトはONで、CHECK制約をチェックする。
- **COMPUTES OFF** ——計算カラムの値の計算を無効にして入力値を受け入れる。デフォルトはONで、入力値を無視して値を計算する。
- **DEFAULTS ON** ——入力ファイルから入力されていないカラムに対してDEFAULTの値の設定を有効にする。デフォルトはOFFで、カラムのDEFAULTの値を無効にする。この設定の影響については、この項の後半で説明する。
- **DELIMITED BY** ——フィールド・デリミタを変更できる。たとえば、「DELIMITED BY '¥x09'」は、入力ファイルをタブで区切ることを指定する。デフォルトはカンマ（,）。
- **ESCAPE CHARACTER** ——入力ファイルの文字列リテラルでエスケープ文字として解釈する単一の文字を指定できる（たとえば、「ESCAPE CHARACTER '!'」）。デフォルトは円記号（¥）。この

オプションが影響を及ぼすのは入力データの処理方法であり、入力ファイル指定内のエスケープ文字の処理方法には関係がない。

- **ESCAPES OFF** ――すべての文字がデータとして扱われるようにエスケープ文字の処理を完全にオフにできる。デフォルトはONで、エスケープ文字を処理する。前述のESCAPE CHARACTERの場合と同様に、このオプションが影響を及ぼすのはファイル内のデータであり、LOAD TABLE文内のファイル指定には関係がない。
- **BCPフォーマット**―― Adaptive Server Enterpriseで使われるバルク・コピー・プログラム（bcp.exe）用の特殊なファイル・フォーマットを、入力ファイルで使用することを指定する。デフォルトは通常のテキスト・ファイル用のASCIIフォーマット。本書では、BCPフォーマットの説明は割愛する。
- **HEXADECIMAL OFF** ―― 0xnnn形式の入力値を引用符で囲まれたバイナリ文字列リテラルとして解釈しないように指示する。入力文字はそのまま格納される。デフォルトはONで、0xnnnスタイルの値は、文字列に変換すべき16進文字列と解釈される。たとえば、0x414243は、HEXADECIMALがOFFの場合は"0x414243"として格納され、HEXADECIMALがONの場合は"ABC"として格納される。これはバイナリ・カラムと文字カラムの両方に影響する。
- **ORDER OFF** ――クラスタード・インデックスを持つテーブルをロードするときに入力ファイルのソートを抑制する。これによって、入力がクラスタード・インデックスのカラム値で既にソートされている場合にLOAD TABLE文の実行が高速化される。デフォルトはONで、クラスタード・インデックスが定義されている場合も入力をソートする。
- **PCTFREE** ―― LOAD TABLEがローを挿入するときに各ページに残す空き領域の量を指定する。空き領域の量は0～100の割合（%）で指定する。これは、CREATE TABLEで指定されたPCTFREE値に優先するが（指定されている場合）、有効なのはLOAD TABLE文の実行中だけである。PCTFREEの詳細については、「1.12　空き領域」を参照。
- **QUOTES OFF** ――引用符で囲まれているかどうかに関係なく、すべての文字列入力値を引用符なしとして扱うことを指定する。つまり、引用符は、どこに現れるかを問わず、データの一部として格納される。また、引用符で囲まれていない入力文字列から先行ブランクを削除しないことも指定する。ただし、空白の入力フィールドは、常に空の文字列として扱われる。デフォルト動作であるQUOTES ONは、入力値を囲む引用符を削除する。詳細は、この項の後半で説明する。
- **STRIP OFF** ――引用符で囲まれていない文字列入力値の後続ブランクを削除しないことを指定する。デフォルトはONで、引用符で囲まれていない文字列値から後続引用符を削除する。引用符で囲まれた文字列値はいずれの設定にも影響されない。先行ブランクは、STRIPオプションではなくQUOTESオプションの影響を受けることに注意。
- **WITH CHECKPOINT ON** ―― LOAD TABLEが終了したときにチェックポイント操作を自動的に実行することを指定する。これによって、新しいローすべてが物理データベース・ファイルに書き込まれることが保証される。LOAD TABLEはトランザクション・ログに新しいローを書き込まないので、この動作は重要。デフォルトはOFFで、チェックポイントは実行されない。

Tip

DELIMITED BY オプションのコーディングには注意が必要です。x の後ろに数字の 0 と 9 が付く 'x09' を含め、ほとんどすべての文字列値が受け入れられます。タブ文字が必要な場合は、エスケープ文字を忘れずに DELIMITED BY '¥x09' と指定してください。

入力値が不足している場合の処理規則は非常に複雑です。言葉では説明しにくいので、簡単な疑似コードによってその規則を示します。次の疑似コードは、1 つのカラムに対する 1 つの値がどのように処理されるかを表しています。

```
IF このカラムが LOAD TABLE に指定された入力名リストから省かれている
OR このカラムに対応するこの入力レコード上のフィールドが空である
THEN
   IF このカラムに対応する DEFAULT 値が CREATE TABLE にない
   OR LOAD TABLE の DEFAULTS オプションが OFF である
   THEN
      IF このカラムは NULL 値を受け入れる
      THEN
         NULL 値が使用される
      ELSE
         空の文字列 '' をカラムのデータ型に変換してその値を使用することが
         試みられる。このプロセスによって、数値カラムと BIT カラムには 0、
         文字カラムとバイナリ・カラムには空の文字列が割り当てられるが、
         DATE カラム、TIME カラム、TIMESTAMP カラムでは失敗してエラー・
         メッセージが出力される
   ELSE
      カラムの DEFAULT 値が使用される
ELSE
   このカラムに対応する入力フィールドの値が使用される
```

引用符で囲まれた入力文字列の処理規則も複雑です。次に、QUOTES ON で使用できるように LOAD TABLE の入力ファイルを準備するときに考慮すべきポイントを示します。以下のコメントは入力の各行に当てはまります。

- デリミタ文字を入力文字列値の一部とする場合は、引用符と QUOTES ON の指定が必要である
- 先行引用符の前の先行ブランクは破棄される。先行引用符がない場合は、最初のブランク以外の文字の前の先行ブランクが破棄される
- 最初のブランク以外の文字が何であるかによって、入力値を引用符付きとして扱うか、引用符なしとして扱うかが決まる
- 最初のブランク以外の文字が引用符でない場合、値は引用符なしとして扱われ、次のデリミタまでの残りの文字はすべて（デリミタ自体は含まれない）そのまま格納される。引用符で囲まれていない入力値は、QUOTES が ON であるかどうかに関係なく使用できる

- 最初のブランク以外の文字が一重引用符または二重引用符の場合、値は引用符付きとして扱われる。これ以降の規則は引用符付きの値にのみ適用される
- 左引用符のタイプ（一重または二重）によって、右引用符のタイプが決まる
- もう一方のタイプの埋め込み引用符はデータの一部として扱われる（たとえば、"Fred's not here" は有効な入力値）
- 同じタイプの2つ1組の埋め込み引用符は1つの引用符に変換され、データの一部として扱われる（たとえば、'Fred''s not here' は Fred's not here として格納される）
- 左引用符の後ろにあるが、2つ1組になっていない同じタイプの引用符は、右引用符として扱われる
- 右引用符の後ろにある文字は、次のデリミタまで破棄される（デリミタは含まれない）。これは、同じタイプの埋め込み引用符を2つ1組で指定するのを忘れたときに発生する（たとえば、'Fred's not here' は、埋め込み引用符が右引用符として扱われるため、'Fred' として格納される）
- 右引用符がない場合、レコードの最後までのデータがすべて、この入力フィールドのデータの一部として扱われる

次に示すのは、自由形式のテキストをテーブルにロードする特殊な目的に LOAD TABLE オプションを使用する例です。raw_text テーブルは、入力ファイルの元の順序を保持するための line_number カラムと、入力の各行をそのまま、再フォーマットも処理も行わずに受け取るための line_text カラムから成り立ちます。

```
CREATE TABLE raw_text (
   line_number          BIGINT NOT NULL DEFAULT AUTOINCREMENT,
   line_text            LONG VARCHAR NOT NULL DEFAULT '',
   PRIMARY KEY ( line_number ) );

LOAD TABLE raw_text ( line_text )
   FROM 'c:¥¥temp¥¥test.txt'
   DEFAULTS ON
   DELIMITED BY ''
   ESCAPES OFF
   QUOTES OFF
   STRIP OFF;
```

この LOAD TABLE 文は、入力値リスト（line_text）を使って、入力ファイルに入っているのが line_text カラムのデータだけで、line_number カラムのデータは入っていないことを指定しています。DEFAULTS ON オプションを使用しているので、DEFAULT AUTOINCREMENT 機能によって line_number 値が生成されます。デフォルトでは、LOAD TABLE コマンドで DEFAULT の値が入力されることはありません。「DELIMITED BY ''」オプションは、フィールド・デリミタとして扱う文字がないことを指定し、ESCAPES OFF は、エスケープ文字を探す必要もないことを意味します。

QUOTES OFFは、先行引用符をデータの一部として扱うことと先行ブランクを保持することをLOAD TABLEに指示します。STRIP OFFは、後続ブランクを保持することをLOAD TABLEに指示します。

次の入力ファイル例は、LOAD TABLEを実行して再フォーマットなしでテキストを格納する際の問題をいくつか示しています。

```
This is a flat text file, containing free-form text with embedded
commas, 'single quotes', and "double quotes". Even lines with
'leading and trailing quotes will be stored as is.'
It will be stored in the line_text column as is, line-by-line, with
one line per row. Empty lines

will be ignored, but blank lines consisting of at least one space

will be stored. Trailing blanks will be stored,
     and so will leading blanks.
Backslash characters ¥, ¥¥, ¥¥¥, ¥¥¥¥, etc., will be stored as is.
```

次に示すのは、ISQLで「SELECT * FROM raw_text ORDER BY line_number」を実行したときのraw_textテーブルの内容です。5行目の次の空の行は無視され、1つのブランクからなる行はロー7として格納されたことがわかります。また、ロー8では後続ブランクが格納され、ロー9では先行ブランクが格納されています。

```
#  line_text
=  =========
1  'This is a flat text file, containing free-form text with embedded'
2  'commas, ''single quotes'', and "double quotes". Even lines with'
3  '''leading and trailing quotes will be stored as is.'''
4  'It will be stored in the line_text column as is, line-by-line, with '
5  'one line per row. Empty lines'
6  'will be ignored, but blank lines consisting of at least one space '
7  ' '
8  'will be stored. Trailing blanks will be stored,                  '
9  '     and so will leading blanks.'
10 'Backslash characters ¥¥, ¥¥¥¥, ¥¥¥¥¥¥, ¥¥¥¥¥¥¥¥, etc., will be stored as is.'
```

LOAD TABLEが高速なのは、3つのショートカットを使用するためです。まず、挿入トリガを起動しません。これは、LOAD TABLEが外部キー検査を省略するという意味ではありません。反対に、参照整合性制約に違反するローをロードしようとすると、LOAD TABLE文は失敗します。しかし、挿入トリガに重要なアプリケーション・ロジックがあっても、トリガは実行されないので、データをロードする別の方法が必要になります。

2番目のショートカットは、個々の挿入されたローをロックする代わりに、テーブル全体に排他ロックをかけることです。このショートカットは同時実行性に影響を与えます。たとえば、他のユーザが頻繁に更新するテーブルでLOAD TABLEを実行した場合、2つの問題が発生する可能性があります。

つまり、LOAD TABLE 文は排他ロックを取得できないので実行が開始されるまで待機し、いったん実行が開始されると、他のユーザはLOAD TABLEの実行が終了するまでブロックされます。LOAD TABLEは途中でコミットせず、実行が成功した場合に1回だけコミットし、失敗した場合はロールバックを実行します。

3番目のショートカットは、トランザクション・ログ・ファイルに個々の挿入されたローを書き込まず、LOAD TABLEコマンド自体のレコードのみを書き込むことです。したがって、挿入されたローをアップロード・ストリームに取り込みたい場合、Mobile Linkを介してアップロードするテーブルにはLOAD TABLEを使用しないでください。Mobile Linkは、トランザクション・ログを調べて、アップロードするローを判別するため、LOAD TABLEを使って挿入されたローはアップロードされません（Mobile Linkの詳細については、「第7章　同期」を参照してください）。

3番目のショートカットは、トランザクション・ログを使ったリカバリにも影響します。この影響は、LOAD TABLEの終了時にチェックポイントを強制的に実行するWITH CHECKPOINTを指定しない場合に現れます。トランザクション・ログに入っているのはLOAD TABLEコマンド自体のみであり、個々のローは入っていないので、リカバリ・プロセスを実行するには元のファイルが必要になります。

LOAD TABLEがトランザクション・ログに実際に何を書き込むかを次の例で示します。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL,
   col_2         INTEGER NOT NULL,
   PRIMARY KEY ( key_1 ) );

LOAD TABLE t1 FROM 't1_d.txt';
```

dbtran.exeユーティリティを使うと、トランザクション・ログ・ファイルに記録されている操作をテキスト・ファイルのSQL文にフォーマットし直すことができます。次の例は、前述のLOAD TABLEによって生成されたものです。個々のローに関する情報はなく、元のLOAD TABLE文自体とCHECKPOINTに関する情報しかないことに注意してください。

```
--CHECKPOINT-0000-0000507397-2003/oct/27 14:57
...
--SQL-1001-0000466662
load into table t1 from 't1_d.txt'
go
```

Note

このdbtranの出力からわかるように、LOAD TABLEはロード操作が開始される前にチェックポイントを自動的に実行します。これは、WITH CHECKPOINTオプションを指定した場合にLOAD TABLEの完了後に実行されるオプションのチェックポイントとは異なります。

LOAD TABLEには、高速に実行できることの他にもう1つメリットがあります。ストアド・プロシージャ内の文が、テキスト・ファイルからテーブルにデータをロードできることです。クライアント・アプリケーションやインタフェースは必要ありません。

ストアド・プロシージャ内でLOAD TABLEを使用できるというメリットはあるが、前述の3つのショートカットによるデメリットがあるため、LOAD TABLEを使って高速に処理することはあきらめざるを得ないと思われるような場合にも、問題を解決するうまいやり方があります。それは、LOAD TABLEでテンポラリ・テーブルを使用し、INSERTを介してデータをコピーする方法です。

前述のLOAD TABLEの例を、テンポラリ・テーブルを使用するように変更する方法を次に示します。

```
BEGIN
   DECLARE LOCAL TEMPORARY TABLE temp_t1 (
      key_1         INTEGER NOT NULL,
      col_2         INTEGER NOT NULL,
      PRIMARY KEY ( key_1 ) )
      NOT TRANSACTIONAL;
   LOAD TABLE temp_t1 FROM 't1_d.txt';
   INSERT t1 SELECT * FROM temp_t1;
END;
```

これで、挿入された個々のローがトランザクション・ログに記録されます。次に、dbtran.exeの出力を示します。

```
--INSERT-1001-0000475402
INSERT INTO DBA.t1(key_1,col_2)
VALUES (1,1)
go
--INSERT-1001-0000475411
INSERT INTO DBA.t1(key_1,col_2)
VALUES (2,2)
go
```

テンポラリ・テーブルが関与する操作はトランザクション・ログに記録されず、NOT TRANSACTIONAL句を指定するとロールバック・ログにも記録されません。つまり、LOAD TABLE文がトランザクション・ログに書き込まれることはなく、この文によって挿入されるローもロールバック・ログに書き込まれず、コミットもチェックポイントも発生しない、ということです。この方法の速度上の不利益はそれほど悪くありません。テンポラリ・テーブルの詳細については、「第1章 作成」の1.15項を参照してください。

Section

2.4 ISQL INPUT

Interactive SQL ユーティリティ（dbisql.exe、つまりISQL）は、LOAD TABLE に似たISQL INPUT 文をサポートしていますが、両者は多くの点で大きく異なります。

ISQL INPUT 文には3つのフォーマットがあります。最初のフォーマットでは、FROM オプションを使ってファイルからデータを読み取ります。2番目のフォーマットでは、PROMPT オプションを使って、個々のカラム値を手動入力するための対話型ダイアログをローごとに開始します。3番目のフォーマットでは、FROM オプションも PROMPT オプションも使用しませんが、ISQL の［SQL 文］ウィンドウまたはINPUT 文を含むコマンド・ファイルのいずれかで、INPUT 文の直後にインラインで指定したテキスト行を読み取ります。インライン・データの処理は、1行にセミコロンなしでそれ自体しか指定されていないキーワードの END、または入力の物理的な終了で終わります。

次に、3つのフォーマットの構文をすべて示します。特に、FROM 句を使う最初のフォーマットについては後で詳しく説明します。他の2つのフォーマットは、データ量が少ない場合は便利ですが、大きなテーブルのロードには向いていません。

```
<isql_input>               ::= <isql_input_from_file>
                             | <isql_input_with_prompt>
                             | <isql_inline_input>
<isql_input_from_file>     ::= INPUT <file_input_option_list>
<isql_input_with_prompt>   ::= INPUT <prompt_input_option_list>
<isql_inline_input>        ::= INPUT <inline_input_option_list>
                               <inline_data>
                               <end_of_input_marker>
<file_input_option_list>   ::= <input_option_list> FROMオプションを含む
<prompt_input_option_list> ::= <input_option_list> PROMPTオプションを含む
<inline_input_option_list> ::= <input_option_list> FROMもPROMPTも含まない
<input_option_list>        ::= { <input_option> }
<input_option>  ::= INTO [ <owner_name> "."] <target_table_name> -- 必須
                  | "(" <column_name_list> ")" -- デフォルトはすべてのカラム
                  | FROM <input_file>
                  | PROMPT
                  | BY NAME        -- 自己記述型のファイル･フォーマット用
                  | BY ORDER       -- デフォルト
                  | COLUMN WIDTHS "(" <column_width_list> ")" -- FIXEDフォーマット用
                  | DELIMITED BY <input_delimiter>            -- デフォルトは ','
                  | ESCAPE CHARACTER <input_escape_character> -- デフォルトは '¥'
                  | FORMAT <input_format>                     -- デフォルトはASCII
                  | NOSTRIP        -- デフォルトは引用符の付いていない後続ブランクの削除
<input_file>    ::= クライアントを基準とする文字列リテラルのファイル指定
                  | クライアントを基準とする二重引用符付きのファイル指定
                  | クライアントを基準とする引用符なしのファイル指定
```

```
<column_width_list>   ::= <column_width> { "," <column_width> }
<column_width>        ::= FIXED フォーマットの場合の整数リテラルのカラム幅
<input_delimiter>     ::= カラム・デリミタ文字列が入っている文字列リテラル
<input_escape_character> ::= 長さが 1 文字のみの文字列リテラル
<input_format>        ::= <input_format_name> が含まれる文字列リテラル
                        | 二重引用符付きの <input_format_name>
                        | 引用符なしの <input_format_name>
<input_format_name>   ::= ASCII     -- デフォルト
                        | DBASE     -- 入力は DBASEII または DBASEIII フォーマット
                        | DBASEII   -- 自己記述型のファイル・フォーマット
                        | DBASEIII  -- 自己記述型のファイル・フォーマット
                        | EXCEL     -- 自己記述型のファイル・フォーマット
                        | FIXED
                        | FOXPRO    -- 自己記述型のファイル・フォーマット
                        | LOTUS     -- 自己記述型のファイル・フォーマット
<inline_data>         ::= INPUT 文の直後に続くデータの行
<end_of_input_marker> ::= END -- 別個の行にこのキーワード自体のみを指定
                        | [SQL 文] ウィンドウ枠の実行された行の終わり
                        | ISQL コマンド・ファイルのファイルの終わり
```

デフォルトの入力ファイル・フォーマットは、各行が個々のローを表し、カンマをデリミタとするASCIIテキストです。次の例では、前項の最初の例のテーブルに、LOAD TABLEではなくINPUT文を使用しています。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL,
   col_2         VARCHAR ( 100 ) NULL,
   col_3         DECIMAL ( 11, 2 ) NULL,
   col_4         TIMESTAMP NULL,
   col_5         INTEGER NOT NULL,
   PRIMARY KEY ( key_1 ) );

INPUT INTO t1 FROM 'c:¥¥temp¥¥t1_a.txt';
```

t1_a.txtの内容は、先ほど使用したものと同じです。

```
1,'Hello, World',67.89,2003-09-30 02:15PM,999
2,   stripped string without quotes    ,  0  ,  ,  0
3,,,,
4,"    double quoted padded string   ",0,2003 9 30,-111
```

この場合、INPUT文は、対応するLOAD TABLEとまったく同じ機能を実行します。t1の内容は同じです。

```
key_1 col_2                                col_3 col_4                   col_5
===== ==================================== ===== ====================== =====
1     'Hello, World'                       67.89 2003-09-30 14:15:00.000 999
2     'stripped string without quotes'     0.00  NULL                    0
3     NULL                                 NULL  NULL                    0
4     '    double quoted padded string   ' 0.00  2003-09-30 00:00:00.000 -111
```

INPUT 文には、複数の異なるソースから入力を読み取れるという、LOAD TABLE にはないメリットがあります。INPUT では、BCP フォーマットがサポートされていませんが、ASCII フォーマット、固定長入力フィールド向けの FIXED フォーマット、自己記述型のファイル・フォーマットの DBASEII、DBASEIII、EXCEL、FOXPRO、LOTUS の他に、DBASE フォーマットもサポートしています。DBASE フォーマットは、「ファイルのフォーマットが dBaseII か dBaseIII かわからないので、INPUT 文に判断を任せる」という意味です。

DBASE、DBASEII、DBASEIII、EXCEL、FOXPRO、LOTUS のいずれかの入力ファイルを使用している場合にターゲット・テーブルがないと、INPUT 文によってターゲット・テーブルが作成されます。データベースにデータを取り込む手段としては高速かつ簡単な方法ですが、デメリットもいくつかあります。まず、サポートされているファイル・フォーマットが限定されています（たとえば、Excel はバージョン 2.1 です）。また、ターゲットのデータ型や制約を制御することはできません。さらに、入力のデータ型のいくつかはまったくサポートされていません。こうした理由から、INPUT 文を実行する場合は事前に明示的にテーブルを作成しておくか、「1.14.4　CREATE EXISTING TABLE」で説明したプロキシ・テーブルを使用することをお勧めします。INPUT 文のテーブル作成機能については、これ以上の説明を割愛します。

ASCII フォーマットと FIXED フォーマットの入力でカラム名リストを使用する目的は 2 つあります。1 つは、入力フィールドをテーブルのカラムに適用する順序を変更するため、もう 1 つは、対応する入力フィールドがないカラムを省略するためです。

- 順序を変更するには、対応する入力フィールドがファイルで実際に出現する順にカラム名をコーディングする
- テーブル内のカラムを省略するには、リストからそのカラム名を削除する

LOAD TABLE の入力名リストと異なり、INPUT のカラム名リストは、カラムにコピーしない入力フィールドの省略には使用できません。LOAD TABLE の "filler()" に相当する機能はありません。

カラム名リストを省略した場合、入力ファイルのレイアウトはテーブルのレイアウトに一致する（つまり、テーブルのカラムと同じ数の入力フィールドがあり、入力フィールドは各入力レコード内で左から右に、CREATE TABLE のカラムと同じ順序で配置されている）と見なされます。

次は、明示的なカラム名リストを使用する INPUT コマンドの例です。2 番目、3 番目、4 番目、5 番目の入力フィールドは、それぞれ col_5、col_4、col_3、col_2 に適用されます。キーワード INPUT は最初に指定しなければなりませんが、その他のオプションの順序は任意です。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL,
   col_2         INTEGER NOT NULL,
   col_3         TIMESTAMP NULL,
   col_4         DECIMAL ( 11, 2 ) NULL,
   col_5         VARCHAR ( 100 ) NULL,
   PRIMARY KEY ( key_1 ) );

INPUT ( key_1, col_5, col_4, col_3, col_2 ) FROM 'c:¥¥temp¥¥t1_e.txt' INTO t1 ;
```

入力ファイルに次のレコードが入っている場合、

```
1, 'Hello, World', 67.89, 2003-09-30 02:15PM, 999
```

t1 に挿入されるローは次のようになります。

```
key_1 col_2 col_3                    col_4 col_5
===== ===== ======================== ===== ==============
1     999   '2003-09-30 14:15:00.000' 67.89 'Hello, World'
```

INPUT FROM のファイル指定は、データベース・サーバが実行されているコンピュータではなく、ISQL が実行されているクライアント・コンピュータを基準とします。UNC ファイル指定が不要なため、LOAD TABLE よりも手軽に利用できます。

ただし、ファイル指定には文字列リテラルを使わなければなりません。EXECUTE IMMEDIATE を使って INPUT 文を実行することはできません。また、INPUT 文をストアド・プロシージャの中や、簡単な BEGIN ブロックの中に埋め込むこともできません。これは、INPUT 文はサーバではなく ISQL で解析／実行されるのに対し、EXECUTE IMMEDIATE ブロックや BEGIN ブロックなどのその他の文はデータベース・エンジンに渡されてから解析／実行されるためです。データベース・エンジンを使って INPUT 文を処理しようとすると、構文エラーが発生します。

INPUT の動作は、次のようなオプションで制御します。

- **BY NAME** ——ファイルの中で定義されているフィールド名をテーブル内のカラム名に対応させて、どのフィールドをどのカラムに使用するかを指定する。このオプションは、自己記述型ファイル・フォーマットの DBASE、DBASEII、DBASEIII、EXCEL、FOXPRO、LOTUS にのみ使用できる。
- **BY ORDER** ——これがデフォルトの動作。入力ファイルのレイアウトがテーブルのレイアウトと一致することを指定する。BY NAME と BY ORDER は、自己記述型のファイル・フォーマット専用のオプションなので、カラム名リストは無視される。ASCII フォーマットと BCP フォーマットでは、デフォルトの BY ORDER か明示的なカラム名リストのいずれかを選択できる。
- **COLUMN WIDTHS** —— FIXED フォーマットの入力用で、入力ファイルのフィールド幅を左か

ら右にリストする。COLUMN WIDTHSは、形式的にはオプションだが、実質的にはFIXEDフォーマットによる入力で必須である。このオプションを省略した場合にINPUT文がカラムの幅を計算する方法については、説明を割愛する。

- **DELIMITED BY** —— ASCIIフォーマットと共に使用し、フィールド・デリミタを変更できる。たとえば、「DELIMITED BY '¥x09'」は、入力ファイルをタブで区切ることを指定する。デフォルトは「DELIMITED BY ','」。
- **ESCAPE CHARACTER** —— ASCIIフォーマットと共に使用し、入力ファイルの文字列リテラルでエスケープ文字として解釈する単一の文字を指定できる（たとえば、「ESCAPE CHARACTER '!'」）。デフォルトは「ESCAPE CHARACTER '¥'」。FIXEDフォーマットで使用するファイルの入力フィールドには、デリミタやエスケープ文字の概念がないので、そのまま処理される。
- **NOSTRIP** —— ASCIIフォーマットと共に使用し、引用符で囲まれていない文字列を処理するときに後続ブランクを保持することを指定する。デフォルトでは、引用符で囲まれていない文字列から後続ブランクが削除される。先行ブランクは、引用符で囲まれていない文字列からは常に削除されるが、引用符で囲まれている文字列からは、NOSTRIPの指定にかかわらず、先行ブランクも後続ブランクも削除されない。

INPUT文は、LOAD TABLEで使用されるいずれのショートカットも使用しないので、お世辞にも速いとは言えません。INPUTが実際に行うのは、データベース・エンジンにローを1つずつ渡して挿入することです。この動作は、データベース・エンジンの処理を調べるための要求レベルのロギング機能をオンにすることによって確認できます。次に示すのは、2行の入力ファイルを2カラムのテーブルにロードするときのLOAD TABLEとINPUTを比較する例です。

```
CREATE TABLE t1 (
   col_1         INTEGER NOT NULL,
   col_2         INTEGER NOT NULL );

CALL sa_server_option ( 'Request_level_log_file', 'r.txt' );
CALL sa_server_option ( 'Request_level_logging', 'SQL+hostvars' );

LOAD TABLE t1 FROM 't1_f.txt';
INPUT INTO t1 FROM 't1_f.txt';

CALL sa_server_option ( 'Request_level_logging', 'NONE' );
```

t1_f.txtの内容は次のとおりです。

```
1, 1
2, 2
```

要求レベルのロギング・ファイルr.txtには、コーディングされたとおりにエンジンがLOAD TABLE

コマンドを受け取ったことが示されています。

```
STMT_PREPARE  "LOAD TABLE t1 FROM 't1_f.txt'"
STMT_EXECUTE  Stmt=66327
```

しかし、INPUT 文は INSERT に変更され、入力ファイルの各レコードにつき 1 回、計 2 回実行されています。このとき、2 つのホスト変数を各フィールド用に使用しています。

```
STMT_PREPARE  "INSERT INTO "t1" ("col_1","col_2") VALUES (?,?)"
STMT_EXECUTE  Stmt=66339
HOSTVAR       0 varchar '1'
HOSTVAR       1 varchar '1'
STMT_EXECUTE  Stmt=66339
HOSTVAR       0 varchar '2'
HOSTVAR       1 varchar '2'
```

この例から、INPUT の実行が LOAD TABLE より遅い理由だけでなく、LOAD TABLE のショートカットのデメリットが回避される理由もわかります。INPUT 文では、トリガが起動され、個々のローがロックされ、挿入がトランザクション・ログに書き込まれます。

要求レベルのロギング機能については、「第 10 章　チューニング」の 10.2 項で詳しく説明します。

Section

2.5 この章のまとめ

この章では、INSERT 文の 5 つの異なるフォーマット（すべてのカラムへの VALUES リストを使った INSERT、指定のカラムへの VALUES リストを使った INSERT、SELECT を使ってすべてのカラムへ値を挿入する INSERT、SELECT を使って指定のカラムへ値を挿入する INSERT、AUTO NAME 機能を使った INSERT）について説明しました。また、外部ファイルのデータを挿入するための LOAD TABLE 文と ISQL INPUT 文についても説明しました。

次の章では、データベースのライフサイクルの 3 番目のステップ、ローの選択について説明します。

Chapter

3

選択

Section

3.1 概要

この章では、データベースからデータを選択する3種類の方法として、SQLのSELECT文、UNLOAD文、ISQL OUTPUT文について説明します。この章を構成する40の項と節のうち、その36個までがSELECT文に関する説明であり、その重要性をうかがわせます。「3.23 CREATE VIEW」さえも、実際はSELECT文に関する説明です。

SELECT文は、簡単に言うと結果セットを返すメカニズムであり、結果セットは、1つ以上のデータ・カラムから成る0個以上のローです。本書では、小文字の「select」は、結果セットを返すこの一般的なメカニズムを表し、大文字の「SELECT」は、各select内に少なくとも1回出現するキーワードを表します

3.2項には、selectを実行するときに使用される架空の（または論理的な）ステップのリストが示されています。このリストは、各句が実行する内容を説明するために示されています。句のコーディング順序は、各句が実行する内容とその影響、とりわけそれぞれの句の相互関係への手がかりとなるため重要です。GROUP BY句で問題が生じたことがある場合、予想していた3つのローではなく10,000ものローがSELECT文で返ってきたことがある場合、またはWHERE句でNUMBER(*)を呼び出せない理由がわからない場合、この章にその答えがあります。例も用意されています。

3.3～3.24.1項では、3.2項に示されている論理ステップとほぼ同じ順序で、さまざまな句／要素の構文とその意味について説明します。たとえば、FROM句がSELECTリストよりかなり前に登場するのは、このためです。

この章では、ブール式を含む式の構文についても説明します。式は他の多くのSQL文でも使用されますが、最も頻出するのはSELECTであるため、ここで説明します。

この章の最後の3.25項と3.26項では、結果セットをファイルに直接書き込むことができる、UNLOADおよびISQL OUTPUTという2つのメソッドについて説明します。

Section

3.2 SELECTの論理的実行

selectは、WITH、ORDER BY、FORなど、いくつかのオプションの句を伴うクエリ式で構成されます。最も単純なクエリ式は、単一のクエリ指定です。最も単純なクエリ指定は、SELECTキーワードの後にselectリストが続く形式で定義されます。また、最も単純なselectリストは、1のような単一のリテラルです。

つまり、最も単純なselectは「SELECT 1」であり、単一のカラムで構成される単一のローを返します。これは1つの極端な形です。対応する両極端のもう1つの形では、複雑な関係の数十のテーブルを対象に数百ものコード行にわたって大規模な選択が行われ、数千ものローが返されます。

```
<select>              ::= [ <with_clause> ]        -- WITH...
                          <query_expression>    -- 少なくとも1つのSELECT...
                          [ <order_by_clause> ] -- ORDER BY...
                          [ <for_clause> ]      -- FOR...
<query_expression>    ::= <query_expression> <query_operator> <query_expression>
                        | <subquery>
                        | <query_specification>
<query_operator>      ::= EXCEPT    [ DISTINCT | ALL ]
                        | INTERSECT [ DISTINCT | ALL ]
                        | UNION     [ DISTINCT | ALL ]
<subquery>            ::= "(" <query_expression>
                              [ <order_by_clause> ]
                              [ <for_xml_clause> ] ")"
<query_specification> ::= SELECT
                          [ DISTINCT ]
                          [ <row_range> ]
                          <select_list>
                          [ <select_into> ]
                          [ <from_clause> ] -- デフォルトはFROM SYS.DUMMY
                          [ <where_clause> ]
                          [ <group_by_clause> ]
                          [ <having_clause> ]
<for_clause>          ::= <for_intent_clause>
                        | <for_xml_clause>
<for_intent_clause>   ::= FOR READ ONLY
                        | FOR UPDATE
                        | FOR UPDATE BY <concurrency_setting>
<concurrency_setting> ::= VALUES
                        | TIMESTAMP
                        | LOCK
<for_xml_clause>      ::= FOR XML RAW [ "," ELEMENTS ] [ "," ROOT ]
                        | FOR XML AUTO [ "," ELEMENTS ] [ "," ROOT ]
                        | FOR XML EXPLICIT [ "," ELEMENTS ] [ "," ROOT ]
```

Note

この項では、まず、上記の<query_specification>について詳しく説明します。他の句は、かなり後で説明するか、またはまったく説明しません。特に、<with_clause>については3.24項で、<order_by_clause>については3.17項で、EXCEPT、INTERSECT、UNIONの各演算子については3.22項でそれぞれ説明します。<for_intent_clause>と<for_xml_clause>の詳細については、SQL Anywhereのオンラインヘルプを参照してください。

selectは、すべてのSQLの中で、最も重要で、最も強力で、最も難解な構成です。難解さの主な理由の1つは、さまざまな句が記述される順序が、句が実際に実行する動作やselectの実行方法とほとんど関係がないことです。この項では、「selectは何を実行するのか」という質問に対する答えを示します。「selectはどのようにして実行されるのか」という質問に対しては、本書では解答していません。クエリ

が実際にどのように実行されるのかという側面については「第10章　チューニング」で説明しますが、本書のほとんどの部分では、あくまでも最終結果を扱います。

selectが実行する内容を説明する1つの方法は、要求された機能を実行するときに使用される、一連の単純なステップについて説明することです。重要なのは、人間が理解できるという意味での「単純さ」であり、コンピュータの実行に関する「速さ」ではありません。これらは、「使用される可能性がある」論理的または架空のステップであり、実際に行われるステップでは決してありません。

次に、論理的な観点で見たselectの処理方法の概要を順を追って示します。

❶ FROM句が評価され、仮想的なカラムで構成される候補の結果セットが生成されます。
❷ 集合関数とNUMBER(*)の呼び出しを除くすべてのselectリスト項目が評価され、各ローに追加されます。
❸ WHERE句が適用され、ローが削除されます。
❹ GROUP BY句が適用され、ローをグループに分割します。
❺ GROUPING以外のすべての集合関数呼び出しがグループごとに評価され、各グループ内の各ローに追加されます。
❻ GROUP BY ROLLUP計算ローが計算され、別個のグループとして候補結果セットに追加されます。
❼ すべてのGROUPING呼び出しがグループごとに評価され、各グループ内の各ローに追加されます。
❽ HAVING句が適用され、グループ全体が削除されます。
❾ ORDER BY句が適用され、グループをソートします。各グループ内の順序は問題ではありません。
❿ 各ローからselectリストにない項目が除外され、各グループは単一のローにまとめられます。
⓫ DISTINCT句が適用され、重複するローが削除されます。
⓬ ローの範囲（FIRST、TOPなど）が適用され、ローが削除されます。
⓭ selectリスト内の各NUMBER(*)呼び出しがローごとに計算され、ローに追加されます。
⓮ FOR XML句が適用され、結果セットが単一カラムに変換されます。
⓯ INTO句が実行され、ホスト変数またはテンポラリ・テーブルが設定されます。

この項の以降の部分では、例を実行しながら、各ステップを展開します。3.3～3.24.1項では、ここに示したステップとほぼ同様の順序で、SQL selectのさまざまな句と要素について詳しく説明します。

Note

ここに示したステップの大部分はオプションであり、対応する要素が実際に存在する場合のみ実行されます。いくつかのステップは相互に排他的です。たとえば、NUMBER(*)呼び出しとDISTINCTを一緒に指定することはできません。したがって、ステップ11と13が同じselectで実行されることはありません。また、UNION、EXCEPT、INTERSECTの各演算子がこのリストには含まれていないことに注意してください。これらの演算子は後で登場するものであり、これらのステップを通過した後の結果セットに対して実行されるものです。

次に、ステップの概要を説明するための例を示します。3つの単純なテーブルにデータを挿入し、主要な句の大部分を含むselectでデータを取得します。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL,
   non_key_1     INTEGER NOT NULL,
   PRIMARY KEY ( key_1 ) );

CREATE TABLE t2 (
   key_1         INTEGER NOT NULL,
   key_2         INTEGER NOT NULL,
   PRIMARY KEY ( key_1, key_2 ),
   FOREIGN KEY fk_t1 ( key_1 ) REFERENCES t1 ( key_1 ) );

CREATE TABLE t3 (
   key_1         INTEGER NOT NULL,
   non_key_1     INTEGER NOT NULL,
   PRIMARY KEY ( key_1 ) );

INSERT t1 VALUES ( 1, 1 );
INSERT t1 VALUES ( 2, 2 );

INSERT t2 VALUES ( 2, 21 );
INSERT t2 VALUES ( 2, 22 );
INSERT t2 VALUES ( 2, 23 );

INSERT t3 VALUES ( 3, 333 );
INSERT t3 VALUES ( 4, 333 );
INSERT t3 VALUES ( 5, 0 );
INSERT t3 VALUES ( 6, 333 );

SELECT DISTINCT
       TOP 4 START AT 2
       t1.key_1 * 100       AS a,
       t3.key_1 * 1000      AS b,
       COUNT(*)             AS c,
       SUM ( t3.non_key_1 ) AS d
  FROM ( t1 LEFT OUTER JOIN t2 ON t1.key_1 = t2.key_1 )
       CROSS JOIN t3
 WHERE b <= 5000
   AND t3.non_key_1 = 333
 GROUP BY ROLLUP ( t1.key_1, t3.key_1 )
HAVING COUNT(*) > 1
 ORDER BY 1, 2;
```

上記のSELECTで生成される結果セットは、次のようになります。

```
a     b     c  d
====  ====  =  ====
100   NULL  2  666
200   NULL  6  1998
200   3000  3  999
200   4000  3  999
```

次に、SELECT の処理方法について、各論理ステップを詳細に示しながら説明します。

ステップ 1：FROM 句が評価され、FROM 句で返されるすべてのカラムから成る候補結果セットが生成されます。これらは、select リスト項目と同じこともあれば同じでないこともあるため、仮想カラムと呼ばれます。これらの仮想カラムは、select リストに指定されない場合であっても、他の句（WHERE、ORDER BY など）が参照する可能性があるため、この時点では必要です。各仮想カラムには、t.c という名前が付けられます。t はテーブル名、ビュー名、または相関名で、c はカラム名またはエイリアス名です。

次に、この時点での等価な SELECT を示します。

```
SELECT t1.key_1,
       t1.non_key_1,
       t2.key_1,
       t2.key_2,
       t3.key_1,
       t3.non_key_1
  FROM ( t1 LEFT OUTER JOIN t2 ON t1.key_1 = t2.key_1 )
       CROSS JOIN t3;
```

LEFT OUTER JOIN、CROSS JOIN、FROM 句の他の要素については、後の項で詳しく説明します。ここでの説明の目的は、FROM 句が他のどの句よりも最初に処理され、残りの select で処理される、単一の結果セットを単に返すことを示すことです。

FROM 句で返される結果セットは次のようになります。説明の際にローを識別できるように、各ローには A、B、C、… の文字を付けています。

```
   t1.    t1.         t2.    t2.    t3.    t3.
   key_1  non_key_1   key_1  key_2  key_1  non_key_1
   =====  =========   =====  =====  =====  =========
A  1      1           NULL   NULL   3      333
B  1      1           NULL   NULL   4      333
C  1      1           NULL   NULL   5      0
D  1      1           NULL   NULL   6      333

E  2      2           2      21     3      333
F  2      2           2      22     3      333
G  2      2           2      23     3      333
```

```
H  2      2          2      21     4      333
I  2      2          2      22     4      333
J  2      2          2      23     4      333

K  2      2          2      21     5      0
L  2      2          2      22     5      0
M  2      2          2      23     5      0

N  2      2          2      21     6      333
O  2      2          2      22     6      333
P  2      2          2      23     6      333
```

Note

SQL Anywhere のクエリ・プロセッサが、select ごとにこのような結果セットを実際に作成するとは思わないでください。このような処理が行われると、クエリによっては実行に相当な時間がかかり、かなりの RAM とディスク容量が消費されてしまうかもしれません。実際には、クエリ・プロセッサはより簡単な方法で処理を行い、処理速度を高めるために可能な限り不要な処理は実行しません。ここで説明する各ステップは、すべての句がどのように組み合わされて最終的な結果に行き着くか説明するための、概念的な処理にすぎません。

ステップ 2：集合関数呼び出しと NUMBER(*) 呼び出しを除く select リスト内の項目が評価され、候補結果リスト内の各ローに追加されます。仮想カラムと select リスト項目の間に重複が生じる可能性がありますが、それはあくまでも一時的です。最終的にはすべての仮想カラムが削除されますが、しばらくの間は必要です。

次に、この時点での等価な SELECT を示します。

```
SELECT t1.key_1,
       t1.non_key_1,
       t2.key_1,
       t2.key_2,
       t3.key_1,
       t3.non_key_1,
       t1.key_1 * 100  AS a,
       t3.key_1 * 1000 AS b
  FROM ( t1 LEFT OUTER JOIN t2 ON t1.key_1 = t2.key_1 )
       CROSS JOIN t3;
```

この時点での候補結果セットは次のようになります。

```
   t1.    t1.         t2.    t2.    t3.    t3.
   key_1  non_key_1   key_1  key_2  key_1  non_key_1  a    b
   =====  =========   =====  =====  =====  =========  ===  ====
A  1      1           NULL   NULL   3      333        100  3000
B  1      1           NULL   NULL   4      333        100  4000
C  1      1           NULL   NULL   5      0          100  5000
D  1      1           NULL   NULL   6      333        100  6000

E  2      2           2      21     3      333        200  3000
F  2      2           2      22     3      333        200  3000
G  2      2           2      23     3      333        200  3000

H  2      2           2      21     4      333        200  4000
I  2      2           2      22     4      333        200  4000
J  2      2           2      23     4      333        200  4000

K  2      2           2      21     5      0          200  5000
L  2      2           2      22     5      0          200  5000
M  2      2           2      23     5      0          200  5000

N  2      2           2      21     6      333        200  6000
O  2      2           2      22     6      333        200  6000
P  2      2           2      23     6      333        200  6000
```

Note

進行状況を理解しやすいように、ローはソートして示しています。実際には、ORDER BY 句が適用されるまで特定の順序が想定されることはなく、それはかなり後になるまで発生しません。

ステップ3：WHERE 句が適用され、ローが削除されます。

WHERE 句は仮想カラムと select リスト項目の両方を参照することがありますが、集合関数呼び出しと NUMBER(*) 呼び出しは参照しません。次に、この時点での等価な SELECT を示します。

```
SELECT t1.key_1,
       t1.non_key_1,
       t2.key_1,
       t2.key_2,
       t3.key_1,
       t3.non_key_1,
       t1.key_1 * 100  AS a,
       t3.key_1 * 1000 AS b
  FROM ( t1 LEFT OUTER JOIN t2 ON t1.key_1 = t2.key_1 )
       CROSS JOIN t3
 WHERE b <= 5000
   AND t3.non_key_1 = 333;
```

候補結果セットはずっと小さくなります。

```
   t1.    t1.        t2.    t2.    t3.    t3.
   key_1  non_key_1  key_1  key_2  key_1  non_key_1  a    b
   =====  =========  =====  =====  =====  =========  ===  ====
A  1      1          NULL   NULL   3      333        100  3000
B  1      1          NULL   NULL   4      333        100  4000
E  2      2          2      21     3      333        200  3000
F  2      2          2      22     3      333        200  3000
G  2      2          2      23     3      333        200  3000
H  2      2          2      21     4      333        200  4000
I  2      2          2      22     4      333        200  4000
J  2      2          2      23     4      333        200  4000
```

ステップ4：GROUP BY句が適用され、ローをグループに分割します。この時点では、各グループは1つ以上のローで構成されています。1グループあたり1ローになるのは、ずっと後です。この時点では、グループ化だけが行われていることにも注意してください。実際の「GROUP BY ROLLUP (t1.key_1, t3.key_1)」句は「GROUP BY t1.key_1, t3.key_1」として解釈され、ROLLUPプロセスは後で発生します。

GROUP BY句は仮想カラムとselectリスト項目の両方を参照することがありますが、集合関数呼び出しとNUMBER(*)呼び出しは参照しません。ただし、GROUP BY句がある場合、selectリストは、集合関数呼び出し、NUMBER(*)呼び出し、GROUP BY句内の項目でのみ構成される必要があります。

Note

selectリストにあるカラムは、GROUP BY句にもあります。ただし、逆は必ずしも成り立ちません。GROUP BY句にあるカラムが、selectリストにある必要はありません。GROUP BY句は、通常、最終的な結果セットに重複するローが含まれていないことを保証しますが、GROUP BY句で指定されるカラムがselectリストにない場合は、必ずしもその限りではありません。

GROUP BY句がない場合、候補結果セット内の各ローは、個別のグループとして扱われます。

最終的に、各グループは単一のローになりますが、これは後で発生します。「GROUP BY t1.key_1, t3.key_1」プロセスが終了した時点でのグループは、次のようになります。

```
   t1.    t1.        t2.    t2.    t3.    t3.
   key_1  non_key_1  key_1  key_2  key_1  non_key_1  a    b
   =====  =========  =====  =====  =====  =========  ===  ====
A  1      1          NULL   NULL   3      333        100  3000  -- グループ1
   ------------------------------------------------------------
B  1      1          NULL   NULL   4      333        100  4000  -- グループ2
   ------------------------------------------------------------
E  2      2          2      21     3      333        200  3000  -- グループ3
F  2      2          2      22     3      333        200  3000
```

```
G  2      2          2      23     3      333        200  3000
   ------------------------------------------------------------
H  2      2          2      21     4      333        200  4000  -- グループ4
I  2      2          2      22     4      333        200  4000
J  2      2          2      23     4      333        200  4000
```

Note

クエリ処理をこのように論理的な視点から見た場合、プロセスのかなり後半になるまで、各グループに複数のローが含まれています。その理由の1つは、この後に行われるAVGなどの統計関数呼び出しのGROUP BY ROLLUP計算が、この基本となるローに基づくためです。GROUP BY ROLLUPプロセスが発生するまでには時間があるため、この基本となるローも長い時間にわたって保持する必要があります。

ステップ5：GROUPINGへの呼び出しを除いて、selectリスト内の各集合関数、GROUP BY句、HAVING句が、グループごとにまとめて計算されます。得られる値は、各グループ内の各ローに追加されます。同じグループ内の異なるローに、重複する集約値が生じますが、この重複は最終的には削除されます。

例の中では、集合関数呼び出しの1つであるSUM (t3.non_key_1)が、selectリスト項目にない仮想カラムを参照しています。これが、この時点まで仮想カラムが引き続き必要であることの理由の1つであり、各グループをまだ単一のローにしない理由です。

COUNT(*)とSUM (t3.non_key_1)が追加された後のグループは次のようになります。

```
   t1.    t1.        t2.    t2.    t3.    t3.                  COUNT  SUM
   key_1  non_key_1  key_1  key_2  key_1  non_key_1  a    b     AS c   AS d
   =====  =========  =====  =====  =====  =========  ===  ====  =====  ====
A  1      1          NULL   NULL   3      333        100  3000  1      333
   ------------------------------------------------------------------------
B  1      1          NULL   NULL   4      333        100  4000  1      333
   ------------------------------------------------------------------------
E  2      2          2      21     3      333        200  3000  3      999
F  2      2          2      22     3      333        200  3000  3      999
G  2      2          2      23     3      333        200  3000  3      999
   ------------------------------------------------------------------------
H  2      2          2      21     4      333        200  4000  3      999
I  2      2          2      22     4      333        200  4000  3      999
J  2      2          2      23     4      333        200  4000  3      999
```

ステップ6：ROLLUP計算ローが計算され、別個のグループとして候補結果セットに追加されます。

この例では、GROUP BY ROLLUP (t1.key_1, t3.key_1)句によって、t1.key_1の値ごとに1つの小計ローが作成されます。このローは、t1.key_1の値の範囲で、異なるt3.key_1値を持つすべてのグループの集約値を計算します。ROLLUPプロセスは、1つの総合計ローも作成します。ここではステップ単

位で説明する都合上、各ROLLUPローを単一ローのグループとして別個に示しています。

3つの新しい単一ローのROLLUPグループが追加された後の結果セットは次のようになります。ROLLUPの総計グループはR1として示し、2つの小計ROLLUPグループはR2およびR3として示しています。COUNTとSUMの値がROLLUPグループ内で合計されていることに注意してください。

```
    t1.    t1.        t2.    t2.    t3.    t3.                    COUNT  SUM
    key_1  non_key_1  key_1  key_2  key_1  non_key_1  a     b     AS c   AS d
    =====  =========  =====  =====  =====  =========  ===   ====  =====  ====
R1  NULL   NULL       NULL   NULL   NULL   NULL       NULL  NULL  8      2664
    ---------------------------------------------------------------------------
R2  NULL   NULL       NULL   NULL   NULL   NULL       100   NULL  2      666
    ---------------------------------------------------------------------------
A   1      1          NULL   NULL   3      333        100   3000  1      333
    ---------------------------------------------------------------------------
B   1      1          NULL   NULL   4      333        100   4000  1      333
    ---------------------------------------------------------------------------
R3  NULL   NULL       NULL   NULL   NULL   NULL       200   NULL  6      1998
    ---------------------------------------------------------------------------
E   2      2          2      21     3      333        200   3000  3      999
F   2      2          2      22     3      333        200   3000  3      999
G   2      2          2      23     3      333        200   3000  3      999
    ---------------------------------------------------------------------------
H   2      2          2      21     4      333        200   4000  3      999
I   2      2          2      22     4      333        200   4000  3      999
J   2      2          2      23     4      333        200   4000  3      999
```

Note

AVGなどの統計関数のROLLUP計算は、グループではなく個別のローに対して適用されます。これは重要な点で、この時点でグループに依然として個別のローが含まれる理由の1つです。たとえば、SELECTリストにAVG(t1.key_1)が指定されている場合、ROLLUP総計ローの値は、8個すべてのローにまたがるt1.key_1の平均値である1.75です。これは正しい答えです。しかし、4つのグループのt1.key_1の平均値である1.5と、2つのROLLUP小計ローのt1.key_1の平均値である4は、答えとして適切ではありません。

ステップ7：すべてのGROUPING呼び出しがグループごとに評価され、各グループ内の各ローに追加されます。この例にはGROUPING呼び出しは含まれていませんが、GROUPING呼び出しが含まれている場合は、各呼び出しによって、1または0を含むさらに別の項目が各ローに追加されます。GOUPING関数については、「3.15 GROUP BY ROLLUP句」を参照してください。

ステップ8：HAVING句が適用され、グループ全体が削除されます。

この例では、HAVING COUNT(*) > 1句によって、a = 100である2つの単一ロー・グループが削除されます。これは、候補結果セットが小計R2または総計R1には「ならない」ことを意味します。

```
    t1.    t1.        t2.    t2.    t3.    t3.                    COUNT  SUM
    key_1  non_key_1  key_1  key_2  key_1  non_key_1  a     b     AS c   AS d
    =====  =========  =====  =====  =====  =========  ===   ====  =====  ====
R1  NULL   NULL       NULL   NULL   NULL   NULL       NULL  NULL  8      2664
    ------------------------------------------------------------------------
R2  NULL   NULL       NULL   NULL   NULL   NULL       100   NULL  2      666
    ------------------------------------------------------------------------
R3  NULL   NULL       NULL   NULL   NULL   NULL       200   NULL  6      1998
    ------------------------------------------------------------------------
E   2      2          2      21     3      333        200   3000  3      999
F   2      2          2      22     3      333        200   3000  3      999
G   2      2          2      23     3      333        200   3000  3      999
    ------------------------------------------------------------------------
H   2      2          2      21     4      333        200   4000  3      999
I   2      2          2      22     4      333        200   4000  3      999
J   2      2          2      23     4      333        200   4000  3      999
```

この例の場合、HAVING COUNT(*) > 1 句は ROLLUP 計算ローに影響しません。ROLLUP 計算ローが単一ロー・グループの場合であっても同様です。これは、COUNT(*) が、R1、R2、R3 という ROLLUP ローに対して 8、2、6 をそれぞれ返すためです。つまり、ROLLUP ローに対する COUNT(*) は、ROLLUP ローに相当するロー数をカウントします。仮に、HAVING 句に COUNT(*) > 2 が指定されている場合は、R2 というラベルの ROLLUP ローは、この時点で削除されます。

ステップ 9：ORDER BY 句が適用され、ローをソートします。残るのは 1 つのローだけなので、グループ内での順序は問題ではありません。この時点で決定されるのは、グループの相対順序だけです。

これまでのすべてのステップでは、便宜上、グループとローがソートされて示されていましたが、実際にはこの時点まで順序の指定は行われないことに注意してください。また、FIRST や NUMBER(*) のような特定の機能も、ローの順序に依存するため、まだ評価されていません。

Note

ORDER BY は、ASC（デフォルト）が使用される場合は、非 NULL 値より先に NULL 値を配置します。

ステップ 10：各ローは、select リストにある項目のみ残され、各グループが単一ローにまとめられます。このステップは、ORDER BY の後に実行されます。これは、ORDER BY 句に仮想カラムを指定できるため、この時点まで仮想カラムが残っている必要があるからです。

5 つのローに減少した後の 5 つのグループは次のようになります。元のロー・ラベルを左側に示したので、最終的なローがどのグループから来たかがわかるはずです。

```
      a     b     c  d
      ====  ====  =  ====
R1    NULL  NULL  8  2664
R2    100   NULL  2  666
```

```
R3     200   NULL  6  1998
E/F/G  200   3000  3  999
H/I/J  200   4000  3  999
```

Note

このステップ単位の説明では、selectリストが可能な限り早く評価されることを想定しています。たとえば、ステップ7でGROUPING呼び出しが処理されるため、GROUPINGへの呼び出しが含まれるselectリスト式がこの時点で評価されていてもかまいません。しかし、selectリスト式にNUMBER(*)が含まれる場合は、少なくともステップ13までは計算できません。

ステップ11：要約（DISTINCTやALL）が適用され、重複するローが削除されます。効果があるのはDISTINCTだけです。この例の場合、GROUP BY句によって、各ローが最終結果内でそれぞれ異なるため、DISTINCTは機能しません。

ただし、GROUP BY句が処理された後に、重複するローが残ることもあります。これは、GROUP BY句に指定された1つ以上のカラムがselectリストでは省略されている場合に起こり得ます。その場合は、DISTINCTによって、残っている重複ローが削除されます。

ステップ12：ロー範囲（FIRST、TOPなど）が適用され、ローが削除されます。

少なくともこの例に関する限り、この時点でSELECTはその最終的な形式に達します。

```
SELECT DISTINCT
       TOP 4 START AT 2
       t1.key_1 * 100       AS a,
       t3.key_1 * 1000      AS b,
       COUNT(*)             AS c,
       SUM ( t3.non_key_1 ) AS d
  FROM ( t1 LEFT OUTER JOIN t2 ON t1.key_1 = t2.key_1 )
       CROSS JOIN t3
 WHERE b <= 5000
   AND t3.non_key_1 = 333
 GROUP BY ROLLUP ( t1.key_1, t3.key_1 )
HAVING COUNT(*) > 1
 ORDER BY 1, 2;
```

START AT 2句によって総計ローが削除され、この例の最終結果が生成されます。

```
a    b     c  d
===  ====  =  ====
100  NULL  2  666
200  NULL  6  1998
200  3000  3  999
200  4000  3  999
```

ステップ 13：select リスト内の各 NUMBER(*) 呼び出しがローごとに計算され、ローに追加されます。NUMBER(*) は、他のすべての句が順に処理されるまで評価されないことに注意してください。すべてのローがソートされ、不要なすべてのローが削除されるまで、ローに番号を付けることはできません。このため、NUMBER(*) の呼び出しには多くの制約があります。たとえば、WHERE 句内で NUMBER(*) を使用することはできません。NUMBER(*) を計算するには早すぎるためです。

次に、NUMBER(*) 呼び出しを追加したサンプルの SELECT を示します。NUMBER(*) と矛盾するため、DISTINCT 句はコメント・アウトしています。

```
SELECT --DISTINCT
       TOP 4 START AT 2
       t1.key_1 * 100       AS a,
       t3.key_1 * 1000      AS b,
       COUNT(*)             AS c,
       SUM ( t3.non_key_1 ) AS d,
       NUMBER(*)            AS e
  FROM ( t1 LEFT OUTER JOIN t2 ON t1.key_1 = t2.key_1 )
       CROSS JOIN t3
 WHERE b <= 5000
   AND t3.non_key_1 = 333
 GROUP BY ROLLUP ( t1.key_1, t3.key_1 )
HAVING COUNT(*) > 1
 ORDER BY 1, 2;
```

出力結果は、ロー番号をカラム e として示しています。

```
a    b     c  d     e
===  ====  =  ====  =
100  NULL  2  666   1
200  NULL  6  1998  2
200  3000  3  999   3
200  4000  3  999   4
```

ステップ 14：FOR XML 句が適用され、結果セットが、単一の XML ドキュメント・カラムで構成される単一のローに変換されます。

ステップ 15：INTO 句が実行され、ホスト変数またはテンポラリ・テーブルが設定されます。テンポラリ・テーブルには複数のローを格納できますが、ホスト変数を使用できるのは結果セットが単一のローの場合だけです。

次に、INTO 句と FOR XML 句を追加したサンプルの SELECT を示します。NUMBER(*) 呼び出しは FOR XML と矛盾するため削除しています。

```
SELECT DISTINCT
       TOP 4 START AT 2
       t1.key_1 * 100       AS a,
       t3.key_1 * 1000      AS b,
       COUNT(*)             AS c,
       SUM ( t3.non_key_1 ) AS d
  INTO #t
  FROM ( t1 LEFT OUTER JOIN t2 ON t1.key_1 = t2.key_1 )
       CROSS JOIN t3
 WHERE b <= 5000
   AND t3.non_key_1 = 333
 GROUP BY ROLLUP ( t1.key_1, t3.key_1 )
HAVING COUNT(*) > 1
 ORDER BY 1, 2
   FOR XML AUTO;
```

本書では、XML処理の詳細については説明しません。次に示すのは、テンポラリ#t内の単一ロー内の単一カラムです。ページに収まるように改行していますが、本来は1行です。

```
<t1 a="100"><t3 c="2" d="666"/></t1><t1 a="200"><t3 c="6" d="1998"/><t3 b="3000"
c="3" d="999"/><t3 b="4000" c="3" d="999"/></t1>'
```

テンポラリ・テーブルを作成するSELECT INTOメソッドの詳細については、「1.15.2.3　SELECT INTO #table_name」を参照してください。

Section

3.3 FROM句

FROM句は、理論上、他のすべての句が操作する「仮想テーブル」または候補結果セットを指定します。かつては、単なるテーブル名のリストで、それ以上のものではありませんでしたが、最近は改良されて、ネストされた操作を伴う複雑なテーブル式や、以前はWHERE句にのみ指定していたブール式を指定できるようになりました。

次に、FROM句の基本構文を示します。

```
<from_clause>           ::= FROM <table_specification>
<table_specification>   ::= <table_expression_list>
<table_expression_list> ::= <table_expression>
                                { "," <table_expression> } -- カンマの回避
```

FROM句は、カンマで区切られたテーブル式のリストで構成され、場合によってテーブル式はさらにネストされたテーブル式、サブクエリ、カッコで囲まれたテーブル式のリストで構成されます。

```
<table_expression>       ::= <table_term>
                           | <table_expression>
                                CROSS JOIN
                                <table_term>
                           | <table_expression>
                                [ <on_condition_shorthand> ] -- 使用しないこと
                                <join_operator>
                                <table_term>
                                [ <on_condition> ]           -- 代わりにこれを使用する
<table_term>             ::= <table_reference>
                           | <view_reference>
                           | <derived_table>
                           | <procedure_reference>
                           | "(" <table_expression_list> ")"
                           | <lateral_derived_table>
<on_condition_shorthand> ::= KEY     -- 外部キー・カラム。使用しない
                           | NATURAL -- 同じ名前のカラム。使用しない
<join_operator>          ::= <inner_join>
                           | <left_outer_join>
                           | <right_outer_join>
                           | <full_outer_join>
<inner_join>             ::= INNER JOIN
                           | JOIN
<left_outer_join>        ::= LEFT OUTER JOIN
                           | LEFT JOIN
<right_outer_join>       ::= RIGHT OUTER JOIN
                           | RIGHT JOIN
<full_outer_join>        ::= FULL OUTER JOIN
                           | FULL JOIN
<on_condition>           ::= ON <boolean_expression> -- 推奨
<table_reference>        ::= [ <owner_name> "."] <table_name>
                                [ [ AS ] <correlation_name> ]
                                [ <hints> ]
<view_reference>         ::= [ <owner_name> "."] <view_name>
                                [ [ AS ] <correlation_name> ]
                                [ <hints> ]
<owner_name>             ::= <identifier>
<table_name>             ::= <identifier>
<correlation_name>       ::= <identifier>
<view_name>              ::= <identifier>
<identifier>             ::=「第1章　作成」の<identifier>を参照
```

本書では、INNER JOINやLEFT OUTER JOINなどの最新のジョイン演算子を使用するFROM句について説明し、カンマで区切られたテーブル式のリストについては説明しません。テーブル式のリストにおけるカンマは、実際には「カンマ・ジョイン」演算子に相当し、常にではありませんが、通常は

CROSS JOIN と同じです。INNER JOIN と同じように動作することもあり、他のジョイン演算子と組み合わせて使用する場合は非常に紛らわしくなります。ただし、例外として「3.8　ラテラル・プロシージャ・コール」では、便利な機能を利用するためにカンマを必要とする例を示しています。それは、別のテーブルのカラムを引数として受け取るストアド・プロシージャ・コールを含むジョインです。しかし、本書全体を通して、FROM 句に多くのカンマが現れることはありません。

Tip

ジョイン演算子を指定する場合は、デフォルトや省略形を使用しないでください。特に、INNER または OUTER 修飾子を指定せずに JOIN 演算子だけを記述したり、省略形キーワードの KEY と NATURAL を使用するのは避けてください。CROSS JOIN を除くすべてのジョインに対して、必ず ON 条件を記述してください。ジョイン演算子を明確に指定することで、コードがわかりやすくなり、デバッグが容易になります。デフォルトや省略形を使用しても、コーディングの量はそれほど減らず、かえってわかりにくくなるだけです。マルチテーブル・ジョインの場合は特にそうです。

FROM 句内のテーブル参照とビュー参照には、SQL Anywhere がそのクエリを実行する際に、テーブルやビューを扱う方法に影響するヒントが含まれる場合があります。

```
<hints>     ::= HOLDLOCK        -- ISOLATION_LEVEL = 3
              | WITH "(" [ <hint_list> ] ")"
<hint_list> ::= <hint> { <hint> }
<hint>      ::= FASTFIRSTROW    -- OPTIMIZATION_GOAL = 'first-row'
              | NOLOCK          -- ISOLATION_LEVEL = 0
              | READUNCOMMITTED -- ISOLATION_LEVEL = 0
              | READCOMMITTED   -- ISOLATION_LEVEL = 1
              | REPEATABLEREAD  -- ISOLATION_LEVEL = 2
              | HOLDLOCK        -- ISOLATION_LEVEL = 3
              | SERIALIZABLE    -- ISOLATION_LEVEL = 3
```

これらのヒントは、OPTIMIZATION_GOAL と ISOLATION_LEVEL の各オプション設定を一時的に上書きします。ISOLATION_LEVEL の詳細については、「第 9 章 保護」の「9.7 ブロックと独立性レベル」を参照してください。

この後の項では、FROM 句を、さまざまなジョイン演算子と、抽出テーブルやストアド・プロシージャ・コールなどの特殊なテーブル形式の面から説明します。<boolean_expression> 構文の詳細については、「3.12　ブール式と WHERE 句」で説明します。

Section

3.4 JOIN

JOIN キーワードに関する演算子は5種類あり、次の5つの項でそれぞれについて説明します。

- CROSS JOIN —— 2つのテーブルの直積を作成する。
- INNER JOIN —— 2つのテーブルに含まれるローの一致する組み合わせを選択する。
- LEFT OUTER JOIN ——両方のテーブルに含まれるローの一致する組み合わせに加えて、左側のテーブルに含まれるすべてのローを選択する。
- RIGHT OUTER JOIN ——右側のテーブルに含まれるすべてのローを選択する。
- FULL OUTER JOIN ——両方のテーブルに含まれるすべてのローを選択する。

3.4.1 CROSS JOIN

最も単純なテーブル式はテーブル形式であり、最も単純なテーブル形式はテーブル参照であり、最も単純なジョイン演算子はCROSS JOINです。次に、2つのテーブル間のCROSS JOINの例を示します。

```
CREATE TABLE t1 (
   c1  INTEGER NOT NULL );

CREATE TABLE t2 (
   c1  INTEGER NOT NULL,
   c2  INTEGER NOT NULL );

INSERT t1 VALUES ( 1 );
INSERT t1 VALUES ( 2 );

INSERT t2 VALUES ( 1, 7 );
INSERT t2 VALUES ( 1, 8 );
INSERT t2 VALUES ( 1, 9 );

SELECT t1.c1,
       t2.c1,
       t2.c2
  FROM t1 CROSS JOIN t2
 ORDER BY t1.c1,
       t2.c1,
       t2.c2;
```

ジョインとは、2つのテーブルに対する操作であり、各テーブルに含まれるローを結合またはジョインして、1つの結果セットとしてのローを作成します。この結果セットは仮想テーブルと呼ばれることも

あり、両方のテーブルに含まれる（論理上は）すべてのカラムで構成されます。

ジョイン演算子が異なると、異なる方法でローが結合されます。特に、CROSS JOIN 演算子は、一方のテーブルのすべてのローと、他方のテーブルのすべてのローを結合します。つまり、2つのテーブルに含まれるローのすべての組み合わせが作成されます。これは、直積とも呼ばれます。先ほどの例の場合、一方のテーブルに2つのローが含まれており、もう一方のテーブルには3つのローが含まれています。したがって、結果セット内のロー数は6です。

```
c1  c2  c3
==  ==  ==
1   1   7
1   1   8
1   1   9
2   1   7
2   1   8
2   1   9
```

CROSS JOIN 演算子は、ON 句で不要なローを削除できない唯一の演算子です。これが、CROSS JOIN演算子が最も単純なジョイン演算子であることの理由であり、最も役に立たない理由でもあります。この演算子は、通常、あまりに多くのローを返します。たとえば、10,000 ローが含まれる2つのテーブルに対してCROSS JOIN を指定すると、100,000,000 ものローが返されます。

3.4.2 INNER JOIN

ON 条件と共に使用する INNER JOIN 演算子は、CROSS JOIN よりはるかに有用で、はるかに一般的です。INNER JOIN は、通常、外部キー関係を使用して2つのテーブルをジョインするときに使用します。次に示すのは、結果セットを限定する ON 条件を使用して、parent テーブルと child テーブルに含まれるローをジョインする例です。各子ローは parent テーブル内の1つのローにのみ結合されます。

```
CREATE TABLE parent (
   parent_key    INTEGER NOT NULL,
   data_1        VARCHAR ( 1 ) NOT NULL,
   PRIMARY KEY ( parent_key ) );

CREATE TABLE child (
   child_key     INTEGER NOT NULL PRIMARY KEY,
   parent_key    INTEGER NULL REFERENCES parent ( parent_key ) );

INSERT parent VALUES ( 1, 'x' ); -- 3つの子を持つ親
INSERT parent VALUES ( 2, 'x' ); -- 子を持たない親
INSERT parent VALUES ( 3, 'y' ); -- 子を持たない親

INSERT child VALUES ( 4, 1 );    -- 親を持つ子
```

```
INSERT child VALUES ( 5, 1 );    -- 親を持つ子
INSERT child VALUES ( 6, 1 );    -- 親を持つ子
INSERT child VALUES ( 7, NULL ); -- 親を持たない子

SELECT parent.parent_key,
       parent.data_1,
       child.child_key,
       child.parent_key
  FROM parent INNER JOIN child ON parent.parent_key = child.parent_key
 ORDER BY parent.parent_key,
       child.child_key;
```

Note

<boolean_expression> 構文の詳細については、「3.12　ブール式と WHERE 句」で説明します。ON と WHERE では同じ <boolean_expression> 構文が使用されますが、ON 条件では、通常、一部の機能しか利用しません。典型的な ON 条件は「ON a = b AND c = d」の形式であり、単純な等価関係と AND 演算子を使用してテーブルをジョインします。これに対して WHERE 句は、より複雑な形式になる傾向があります。このため、<boolean_expression> については 3.12 項で詳しく説明します。

INNER JOIN 演算子は、ON 条件を満たすデータを持つ、一方のテーブル内のすべてのローと、他方のテーブル内のすべてのローを結合します。この例の場合、最初の親ローが最初の 3 つの子ローに結合され、3 つのローから成る結果セットが生成されます。

```
parent.     parent.  child.     child.
parent_key  data_1   child_key  parent_key
==========  =======  =========  ==========
    1          x         4          1       -- 親と子
    1          x         5          1       -- 親と子
    1          x         6          1       -- 親と子
```

2 番目と 3 番目の親ローは子ローと結合できず、ON 条件を満たすことができないため、最終的な結果セットに含まれません。具体的には、parent.parent_key = 2 および parent.parent_key = 3 が、child.parent_key の値に一致しません。4 番目の子ローについても同様です。child.parent_key = NULL はどの parent.parent_key の値にも一致しないため、どの親ローとも一致しません。言い換えると、ON 条件で外部キー関係を使用すると、子のない親ローと親を持たない子ローは INNER JOIN に含まれません。

3.4.3　LEFT OUTER JOIN

一方のテーブル、他方のテーブル、または両方のテーブルに含まれるローを、たとえ ON 条件を満たさない場合でもすべて保持するには、OUTER JOIN 演算子の 1 つを使用します。前の項の例において、

INNER JOINの代わりにLEFT OUTER JOIN演算子を使用すると、parentテーブルの2番目と3番目のローを結果セットに含めることができます。

```
SELECT parent.parent_key,
       parent.data_1,
       child.child_key,
       child.parent_key
  FROM parent LEFT OUTER JOIN child ON parent.parent_key = child.parent_key
 ORDER BY parent.parent_key,
       child.child_key;
```

LEFT OUTER JOINは次のように動作します。まず、前述のようにINNER JOIN演算が実行され、結果セットが作成されます。次に、INNER JOINに加わらなかった左側のテーブル内のローが結果セットに追加されます。このとき、本来であれば右側のテーブルから取得されるカラムに対してNULL値が使用されます。ON条件は最初のステップにのみ適用されることに注意してください。2番目のステップでは無視されます。

この例の場合、LEFT OUTER JOINは、INNER JOINで得られた結果セットに対して2つのローを追加します。この2つのローは、parentテーブルに含まれる2番目と3番目のローに対応します。

```
parent.     parent.  child.     child.
parent_key  data_1   child_key  parent_key
==========  =======  =========  ==========
    1          x         4          1       -- 親と子
    1          x         5          1       -- 親と子
    1          x         6          1       -- 親と子
    2          x       NULL       NULL      -- 子を持たない親
    3          y       NULL       NULL      -- 子を持たない親
```

LEFT OUTER JOIN演算では、左側のテーブルは、すべてのローが結果セット内に1回以上出現するため、「保護テーブル」と呼ばれます。右側のテーブルは、ジョインの結果に含まれるいくつかのローのカラム値としてNULLが使用されることがあるため、「NULL値があり得るテーブル」と呼ばれます。LEFT OUTER JOIN内の「LEFT」という用語は、左側のテーブルが保護テーブルであることを意味しています。この例の場合、対応する子ローを持たない親ローに相当する、新しい2つのローだけが追加されました。LEFT OUTER JOINでは、1つのテーブルだけが保護テーブルであるため、対応する親ローを持たないオーファンの子ローは最終結果に出現しません。

LEFT OUTER JOIN演算子がよく使用されるのは、対応するオプションのデータが存在するかどうかにかかわらず、必須のデータがすべて存在することを確認する場合です。その際には、各テーブルから必須データとオプション・データの両方を収集します。別の言い方をすると、「子を持たない親を含め、すべての親と子のデータを表示する」ということです。

3.4.4 RIGHT OUTER JOIN

RIGHT OUTER JOIN演算子は、LEFT OUTER JOINにおける2つのテーブルの役割を完全に逆にしたものです。たとえば、次の文では、childテーブルがRIGHT OUTER JOINの右側にあるため、保護テーブルとして定義されています。parentテーブルは反対側にあるため、NULL値があり得るテーブルです。

```
SELECT parent.parent_key,
       parent.data_1,
       child.child_key,
       child.parent_key
  FROM parent RIGHT OUTER JOIN child ON parent.parent_key = child.parent_key
 ORDER BY parent.parent_key,
       child.child_key;
```

最終的な結果セットには、parent.parent_keyとparent.data_1がNULLに設定されている親を持たないオーファンの子ローが含まれていますが、対応する子を持たない親ローはありません。

```
parent.     parent.  child.     child.
parent_key  data_1   child_key  parent_key
==========  =======  =========  ==========
   NULL       NULL       7         NULL     -- オーファン
    1          x         4          1       -- 親と子
    1          x         5          1       -- 親と子
    1          x         6          1       -- 親と子
```

すべてのRIGHT OUTER JOINはLEFT OUTER JOINに変換できます。テーブル名を置き換えるだけで、まったく同じように機能します。たとえば、次のLEFT OUTER JOINは、上のRIGHT OUTER JOINと同じローを返します。

```
SELECT parent.parent_key,
       parent.data_1,
       child.child_key,
       child.parent_key
  FROM child LEFT OUTER JOIN parent ON parent.parent_key = child.parent_key
 ORDER BY parent.parent_key,
       child.child_key;
```

Tip

外部ジョインは、最適な状態においてさえも紛らわしいものです。したがって、LEFT OUTER JOIN演算子とRIGHT OUTER JOIN演算子を混用して、コードをさらに紛らわしくするのは避けてください。LEFT OUTER JOINだけを使うようにすれば、保護テーブルが常に同じ側に来るため、コードが理解しやすくなります。

3.4.5 FULL OUTER JOIN

FULL OUTER JOIN演算子は、LEFT OUTER JOINとRIGHT OUTER JOINの両方の機能が組み合わされた拡張機能です。言い換えると、両方のテーブル内のすべてのローが保護され、必要であれば両方のテーブルがNULL値があり得るテーブルになります。これは次のように動作します。まず、INNER JOINがON条件を使用して計算されます。次に、INNER JOINプロセスに含まれなかった左側のテーブル内のローが結果セットに追加されます。このとき、本来であれば右側のテーブルに含まれるカラムに対してNULL値が使用されます。最後に、INNER JOINプロセスに含まれなかった右側のテーブル内のローが結果セットに追加されます。このとき、本来であれば左側のテーブルに含まれるカラムに対してNULL値が使用されます。

次に、parentテーブルとchildテーブルを使用したFULL OUTER JOINの例を示します。

```
SELECT parent.parent_key,
       parent.data_1,
       child.child_key,
       child.parent_key
  FROM parent FULL OUTER JOIN child ON parent.parent_key = child.parent_key
 ORDER BY parent.parent_key,
       child.child_key;
```

結果セットには、両方のテーブルに含まれるすべてのローのすべてのカラムが格納されます。これには、INNER JOINによる親と子の組み合わせに加えて、RIGHT OUTER JOINによる親を持たないオーファンの子ローと、LEFT OUTER JOINによる子を持たない親ローが含まれています。

```
parent.     parent.  child.     child.
parent_key  data_1   child_key  parent_key
==========  =======  =========  ==========
   NULL       NULL       7         NULL    -- オーファン（親を持たない）
    1          x         4          1      -- 親と子
    1          x         5          1      -- 親と子
    1          x         6          1      -- 親と子
    2          x        NULL       NULL    -- 子を持たない親
    3          y        NULL       NULL    -- 子を持たない親
```

ここで重要なのは、ON条件がOUTER JOINプロセスの最初のステップにのみ適用されることです。ON条件がどのようなものであっても、最終的な結果セットには、保護テーブル内のすべてのローが含まれます。次に示すのは、前に示したLEFT OUTER JOINのON条件に「parent.data_1 = 'x'」という制限を追加した例です。

```
SELECT parent.parent_key,
       parent.data_1,
       child.child_key,
       child.parent_key
  FROM parent LEFT OUTER JOIN child ON  parent.parent_key = child.parent_key
                                    AND parent.data_1     = 'x'
 ORDER BY parent.parent_key,
       child.child_key;
```

この場合、結果セットは前とまったく同じです。

```
parent.     parent.  child.     child.
parent_key  data_1   child_key  parent_key
==========  =======  =========  ==========
    1          x         4          1       -- 親と子
    1          x         5          1       -- 親と子
    1          x         6          1       -- 親と子
    2          x        NULL       NULL     -- 子を持たない親
    3          y        NULL       NULL     -- 子を持たない親
```

「parent.data_1 = 'x'」のローだけを選択するようON条件で指定しているにもかかわらず、「parent.data_1 = 'y'」のローが選択されて驚く人がいますが、これがOUTER JOINの働きであり、想定どおりの動作です。ただし、多くの場合、これは本来必要とされている動作でないことも事実です。

Tip

OUTER JOINのON条件の記述には、十分に注意してください。おおざっぱに言うと、一方のテーブルにのみ影響する条件ではなく、両テーブルからのローの結合方法に影響する条件のみを記述します。OUTER JOINを適用する前に一方のテーブルまたは他方のテーブル内のローを削除するには、抽出テーブルまたはビューを使用します。

Section

3.5 抽出テーブル

「抽出テーブル」とは、FROM句内にサブクエリ全体を記述し、そのサブクエリからの結果セットを、FROM句内の他のテーブル形式と同様に扱うメカニズムです。

```
<derived_table>            ::= <subquery>
                                 [ AS ] <correlation_name>
                                 [ <derived_column_name_list> ]
<derived_column_name_list> ::= "(" <alias_name_list> ")"
<alias_name_list>          ::= <alias_name> { "," <alias_name> }
<alias_name>               ::= <identifier>
```

前の例のLEFT OUTER JOINに記述したON条件は、要件（「parent.data_1 = 'x'」の親ローのみを結果セットに含む）を満たしていませんでした。問題は、OUTER JOIN演算子の動作が原因で、「parent.data_1 = 'y'」のローが含まれてしまうことにあります。次に示すのは、抽出テーブルを使用してこの問題を解決する方法です。この方法では、LEFT OUTER JOINが適用される前に、抽出テーブルによって不要なローを削除しています。

```
SELECT parent.parent_key,
       parent.data_1,
       child.child_key,
       child.parent_key
  FROM ( SELECT *
           FROM parent
          WHERE parent.data_1 = 'x' ) AS parent
       LEFT OUTER JOIN child ON parent.parent_key = child.parent_key
 ORDER BY parent.parent_key,
       child.child_key;
```

Tip

抽出テーブルをコーディングするうえでの最低限の要件は、カッコ内にサブクエリを記述し、その後に相関名を指定することです。残りのFROM句では、その相関名によって、サブクエリの結果セットが識別されます。抽出テーブルを使用して行う操作が、テーブルにWHERE句を適用するというだけの場合、サブクエリに「SELECT *」を使用しない理由はありません。必要に応じて相関名としてテーブル名を使用することも可能ですが、カラムのエイリアス名を指定する必要はありません。つまり、テーブル名とカラム名に関する限り、抽出テーブルは元のテーブルとまったく同じように見えます。また、必ずしもパフォーマンスを心配することはありません。クエリ・オプティマイザは、サブクエリをジョインに変更し、実際には不要なカラムを削除するという処理を適切に行います。

上のLEFT OUTER JOINの例では、抽出テーブルは"parent"という名前で、次のように定義されています。

```
     ( SELECT *
         FROM parent
        WHERE parent.data_1 = 'x' ) AS parent
```

LEFT OUTER JOIN と child テーブルに対して、「parent.data_1 = 'x'」のローだけが条件を満たすことになり、最終的な結果セットは次のようになります。

```
parent.     parent.  child.     child.
parent_key  data_1   child_key  parent_key
==========  =======  =========  ==========
    1          x         4          1       -- 親と子
    1          x         5          1       -- 親と子
    1          x         6          1       -- 親と子
    2          x        NULL       NULL     -- 子を持たない親
```

ON 条件が動作せず、かといって抽出テーブルを使用するのも面倒な場合は特に、FROM 句内で ON 条件を使用するのではなく、外側の SELECT で WHERE 句を使用しがちです。しかし、OUTER JOIN では、WHERE 句は ON 条件に似ています。希望どおりの処理が行われることもあれば、行われないこともあります。特に、FROM 句が完全に評価されたかなり後になって WHERE 句が適用されると、「NULL 値があり得るテーブル」で NULL 値に設定されたカラムの含まれるローが、思いもよらず削除されることがあります。

次に、以前にも示した FULL OUTER JOIN を使用する例を示します。ここでは、WHERE 句に制限を追加することで、親ローを「parent.data_1 = 'x'」を満たすローに限定しようとしています。

```
SELECT parent.parent_key,
       parent.data_1,
       child.child_key,
       child.parent_key
  FROM parent FULL OUTER JOIN child ON parent.parent_key = child.parent_key
 WHERE parent.data_1 = 'x'
 ORDER BY parent.parent_key,
       child.child_key;
```

「3.2　SELECT の論理的実行」の説明によると、最初に FROM 句が評価され、WHERE 句は後で適用されます。つまり、WHERE 句はまだ適用されていないため、FROM 句の最初の結果は、以前の「3.4.5 FULL OUTER JOIN」の結果とまったく同じです。

```
parent.     parent.  child.     child.
parent_key  data_1   child_key  parent_key
==========  =======  =========  ==========
   NULL       NULL       7         NULL     -- このローはなくなる：OK でない
    1          x         4          1
    1          x         5          1
    1          x         6          1
    2          x        NULL       NULL
    3          y        NULL       NULL     -- このローはなくなる：OK
```

WHERE 句が適用されて最終結果セットが生成されると、1つのローではなく、2つのローが削除されます。先頭のローは parent.data_1 が NULL であるために削除され、末尾のローは parent.data_1 が 'y' であるために削除されます。どちらも、「parent.data_1 = 'x'」という WHERE 条件を満たしていません。

言い換えると、最終結果セットに親を持たない子ローは存在しないため、FULL OUTER JOIN はもはや FULL OUTER JOIN ではありません。WHERE 句を追加することで、事実上、LEFT OUTER JOIN に変わってしまったのです。

```
parent.     parent.  child.     child.
parent_key  data_1   child_key  parent_key
==========  =======  =========  ==========
    1          x         4          1
    1          x         5          1
    1          x         6          1
    2          x        NULL       NULL
```

実際、parent.data_1 が 'x' でない親ローを1つだけ削除することが目的であっても、この WHERE 句が存在する限り、child テーブルにたとえ 1000 のオーファン・ローがあろうと、それらはすべて取り除かれてしまいます。

ここで再び問題を解決するのは、FULL OUTER JOIN が計算される前に不要な親ローを削除する抽出テーブルです。

```
SELECT parent.parent_key,
       parent.data_1,
       child.child_key,
       child.parent_key
  FROM ( SELECT *
           FROM parent
          WHERE parent.data_1 = 'x' ) AS parent
       FULL OUTER JOIN child ON parent.parent_key = child.parent_key
 ORDER BY parent.parent_key,
       child.child_key;
```

結果セットは、より意味があるものになります。親を持たない子ローは含まれ、不要な親ローは削除されます。

```
parent.     parent.  child.     child.
parent_key  data_1   child_key  parent_key
==========  =======  =========  ==========
   NULL       NULL       7         NULL     -- オーファン（親を持たない）
    1          x         4          1       -- 親と子
    1          x         5          1       -- 親と子
    1          x         6          1       -- 親と子
    2          x        NULL       NULL     -- 子を持たない親
```

Note

WHERE 句によって OUTER JOIN 内のローが予想外に削除されるのはよくあることです。通常、LEFT OUTER JOIN または RIGHT OUTER JOIN は INNER JOIN になり、FULL OUTER JOIN は LEFT OUTER JOIN または RIGHT OUTER JOIN になります。この現象を技術的な観点から説明すると、次のようになります。「NULL 値があり得るテーブル」の属性を参照する「NULL を許容しない述部」によって、結果から NULL 入力ローが削除されます。「NULL を許容しない述部」とは、いずれかの入力が NULL の場合には true と評価されない述部のことです。比較、LIKE、IN など、大部分の SQL 述部は、NULL を許容しません。NULL を許容する述部の例は、IS NULL や、「p IS NOT TRUE」などのように NULL を許容する真偽値テストで修飾される p 述部などです（「Semantics and Compatibility of Transact-SQL Outer Joins」G. N. Paulley 著、2002 年 2 月 15 日、iAnywhere Solutions テクニカル・ホワイト・ペーパー、ドキュメント番号 1017447 より）。

Section

3.6 マルチテーブル・ジョイン

FROM 句の構文では、ジョインできるテーブル数に制限はなく、ネストされたテーブル式を作成するのにカッコを付けても付けなくてもかまいません。また、各ジョインに ON 条件を指定しても、指定しなくてもかまいません。ほとんどの場合、カッコは必要ありませんが、可能な限りすべてのジョイン演算子に ON 条件を指定するのが適当です。

```
<table_expression>       ::= <table_term>
                           | <table_expression>
                                CROSS JOIN
                                <table_term>
                           | <table_expression>
                                [ <on_condition_shorthand> ] -- 使用しないこと
                                <join_operator>
                                <table_term>
                                [ <on_condition> ]           -- 代わりにこれを使用する
<table_term>             ::= <table_reference>
                           | <view_reference>
                           | <derived_table>
                           | <procedure_reference>
                           | "(" <table_expression_list> ")"
                           | <lateral_derived_table>
<on_condition_shorthand> ::= KEY     -- 外部キー・カラム。使用しない
                           | NATURAL -- 同じ名前のカラム。使用しない
<join_operator>          ::= <inner_join>
                           | <left_outer_join>
```

```
                        | <right_outer_join>
                        | <full_outer_join>
```

カッコがない場合、ジョイン演算子は左から右に向かって評価されます。つまり、テーブル形式の最初のペアがジョインして仮想テーブルを作成し、その仮想テーブルが3番目のテーブル形式にジョインして別の仮想テーブルを作成します。以降についても同様です。

次に、SQL Anywhere Studio 9付属のASADEMOデータベースにある4つのテーブルをジョインする例を示します。4つのテーブル（customer、product、sales_order、sales_order_items）と、後の例で使用する2つのテーブル（employee、fin_code）のスキーマは、次のとおりです。

```
CREATE TABLE customer (
   id             INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   fname          CHAR ( 15 ) NOT NULL,
   lname          CHAR ( 20 ) NOT NULL,
   address        CHAR ( 35 ) NOT NULL,
   city           CHAR ( 20 ) NOT NULL,
   state          CHAR ( 16 ) NULL,
   zip            CHAR ( 10 ) NULL,
   phone          CHAR ( 12 ) NOT NULL,
   company_name   CHAR ( 35 ) NULL,
   PRIMARY KEY ( id ) );

CREATE TABLE employee (
   emp_id             INTEGER NOT NULL PRIMARY KEY,
   manager_id         INTEGER NULL,
   emp_fname          CHAR ( 20 ) NOT NULL,
   emp_lname          CHAR ( 20 ) NOT NULL,
   dept_id            INTEGER NOT NULL,
   street             CHAR ( 40 ) NOT NULL,
   city               CHAR ( 20 ) NOT NULL,
   state              CHAR ( 16 ) NULL,
   zip_code           CHAR ( 10 ) NULL,
   phone              CHAR ( 10 ) NULL,
   status             CHAR ( 2 ) NULL,
   ss_number          CHAR ( 11 ) NULL,
   salary             NUMERIC ( 20, 3 ) NOT NULL,
   start_date         DATE NOT NULL,
   termination_date   DATE NULL,
   birth_date         DATE NULL,
   bene_health_ins    CHAR ( 2 ) NULL,
   bene_life_ins      CHAR ( 2 ) NULL,
   bene_day_care      CHAR ( 2 ) NULL,
   sex                CHAR ( 2 ) NULL );

CREATE TABLE fin_code (
   code          CHAR ( 2 ) NOT NULL PRIMARY KEY,
   type          CHAR ( 10 ) NOT NULL,
```

```
   description   CHAR ( 50 ) NULL );

CREATE TABLE product (
   id            INTEGER NOT NULL,
   name          CHAR ( 15 ) NOT NULL,
   description   CHAR ( 30 ) NOT NULL,
   size          CHAR ( 18 ) NOT NULL,
   color         CHAR ( 6 ) NOT NULL,
   quantity      INTEGER NOT NULL,
   unit_price    NUMERIC ( 15, 2 ) NOT NULL,
   PRIMARY KEY ( id ) );

CREATE TABLE sales_order (
   id            INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   cust_id       INTEGER NOT NULL REFERENCES customer ( id ),
   order_date    DATE NOT NULL,
   fin_code_id   CHAR ( 2 ) NULL REFERENCES fin_code ( code ),
   region        CHAR ( 7 ) NULL,
   sales_rep     INTEGER NOT NULL REFERENCES employee ( emp_id ),
   PRIMARY KEY ( id ) );

CREATE TABLE sales_order_items (
   id          INTEGER NOT NULL REFERENCES sales_order ( id ),
   line_id     SMALLINT NOT NULL,
   prod_id     INTEGER NOT NULL REFERENCES product ( id ),
   quantity    INTEGER NOT NULL,
   ship_date   DATE NOT NULL,
   PRIMARY KEY ( id, line_id ) );
```

customer テーブルには、製品を購入する可能性がある会社に関する情報が格納されます。product テーブルは、各販売製品を定義します。sales_order テーブルには、顧客への各販売が記録されます。sales_order_items テーブルは product と sales_order との間の多対多関係であり、どの注文にどの製品が含まれるかが記録されます。これらのテーブル間には関係を定義するための外部キー関係があり、これらの外部キー関係は3つの INNER JOIN 演算の ON 条件で使用されて、どの注文でどの製品がどの顧客に販売されたかに関するすべての情報が収集されます。

```
SELECT customer.company_name,
       sales_order.order_date,
       product.name,
       product.description,
       sales_order_items.quantity,
       product.unit_price * sales_order_items.quantity AS amount
  FROM customer
       INNER JOIN sales_order
               ON sales_order.cust_id = customer.id
       INNER JOIN sales_order_items
               ON sales_order_items.id = sales_order.id
```

```
      INNER JOIN product
            ON product.id = sales_order_items.prod_id
ORDER BY customer.company_name,
      sales_order.order_date,
      product.name;
```

次に、このFROM句の動作方法について論理的な視点から説明します。

- 最初に、顧客IDカラムの値が一致するcustomer内のローとsales_order内のローがジョインされる。最初のINNER JOINで作成される仮想テーブルには、customerテーブルとsales_orderテーブルのすべてのカラムが含まれる
- 2番目のINNER JOINでは、最初の仮想テーブルのローと、注文IDカラムの値が一致するsales_order_itemのローがジョインされる。最初の仮想テーブル内のカラムは、2番目のON条件の中で、そのベース・テーブル名を使用して参照されることがある（sales_order.order_idなど）。2番目のINNER JOINの結果は、customer、sales_order、sales_order_item内のすべてのカラムで構成される新しい仮想テーブルである
- 最後のINNER JOINでは、2番目の仮想テーブルのローと、製品IDカラムの値が一致するproductのローがジョインされる。最後のINNER JOINの結果は、4つのすべてのテーブル内のカラムで構成される仮想テーブルである。これは（概念的に言うと）単一の仮想テーブルであるが、個々のカラムはORDER BY句の中で元のテーブル名を使用して引き続き参照される可能性がある（customer.company_nameなど）

最終結果セットは、1,097ローで構成されます。次に、先頭の6つのローを示します。これは、Able Inc.から受けた最初の3つの注文の詳細です。

```
company_name  order_date  name          description        quantity  amount
============  ==========  ============  =================  ========  ======
Able Inc.     2000-01-16  Sweatshirt    Hooded Sweatshirt  36        864.00
Able Inc.     2000-01-16  Sweatshirt    Zipped Sweatshirt  36        864.00
Able Inc.     2000-03-20  Baseball Cap  Wool cap           24        240.00
Able Inc.     2000-04-08  Baseball Cap  Cotton Cap         24        216.00
Able Inc.     2000.04.08  Baseball Cap  Wool cap           24        240.00
Able Inc.     2000-04-08  Visor         Cloth Visor        24        168.00
```

各ON条件は、直前のジョイン演算子に適用されます。次のFROM句では、前の例の中でカッコを使用して、どのONがどのINNER JOINに対応するかを明示的に示しています。このFROM句は、カッコの有無にかかわらずまったく同じ動作をします。

```
FROM ( ( ( customer
         INNER JOIN sales_order
               ON sales_order.cust_id = customer.id )
         INNER JOIN sales_order_items
               ON sales_order_items.id = sales_order.id )
```

```
        INNER JOIN product
              ON product.id = sales_order_items.prod_id )
```

算術式の中のカッコは、演算子の本来の実行順を変更する必要がある場合に便利です（加算してから乗算する場合など）。また、算術式の中のカッコは、たとえ不要な場合であっても、評価の実行順を明確にするのに役立ちます。しかし、FROM 句内のカッコには、この議論はそのまま当てはまりません。何よりもまず、INNER JOIN や LEFT OUTER JOIN などの異なるジョイン演算子の間には優先順位の差がありません。カッコがないと、単純に左から右に向かって評価されます。また、FROM 句は長くなる傾向があるため、対応するカッコが遠く離れるという問題もあり、カッコを付けてもそれほどわかりやすくなりません。上の単純な例でさえも、カッコが何を意味するかを理解するのは困難であり、カッコがない方がかえってわかりやすいとも言えます。

それでもやはり、FROM 句の中のカッコが必要で役に立つこともあります。次に、前述の ASADEMO データベース内の 4 つのテーブル（customer、product、sales_order、sales_order_items）を使用して、この点を明らかにする例を示します。販売のなかった製品と顧客の組み合わせを含めて、各種のシャツが、ワシントン D.C. の各顧客に対して何枚売れたかを示す必要があるものとします。つまり、実際の販売の有無にかかわらず、ワシントンの顧客とシャツ製品のすべての組み合わせを示します。

一見したところでは、4 つのジョインが必要だと思われます。つまり、customer と product の間で CROSS JOIN を実行して可能なすべての組み合わせを作成し、customer と sales_order の間で LEFT OUTER JOIN を実行して、購入したかどうかにかかわらず顧客を選択し、product と sales_order_items の間で LEFT OUTER JOIN を実行して、売れたかどうかにかかわらず製品を選択し、sales_order と sales_order_items の間で INNER JOIN を実行して、注文と注文項目を対応させます。

この 4 つのジョインをカッコの有無にかかわらず正しい順序で記述することはおそらく可能ですが、分断統治法を使用する方が簡単です。

- 最初に、2 つの異なる仮想テーブルを別々に計算する。つまり、customer と product の間で CROSS JOIN を実行し、sales_orde と sales_order_item の間で INNER JOIN を実行する
- 次に、1 番目と 2 番目の仮想テーブルの間で LEFT OUTER JOIN を実行する。カッコを使用して、1 番目のステップと 2 番目のステップを区別する

次に、この方法を使用した FROM 句の擬似コードを示します。

```
SELECT ...
  FROM ( customer と product のすべての組み合わせ )
       LEFT OUTER JOIN
       ( sales_order と sales_order_items のうち対応するすべての組み合わせ )
 WHERE ...
```

完全な SELECT 文を次に示します。FROM 句には 3 つのジョインだけがあり、その中の 2 つはカッ

コ内でネストされ、2つの単純な仮想テーブルを作成しています。最後のLEFT OUTER JOINによって、2つの仮想テーブル内の4つのベース・テーブルすべてを参照するON句を使用して、この2つの仮想テーブルを結合します。カッコがあることによって、わかりやすくなっています。CROSS JOINは最も単純なジョインで表され、INNERジョインは、sales_orderローと関連するsales_order_itemsローの単純な組み合わせです

```
SELECT customer.company_name              AS company_name,
       product.name                       AS product_name,
       product.description                AS product_description,
       SUM ( sales_order_items.quantity ) AS quantity,
       SUM ( product.unit_price
           * sales_order_items.quantity ) AS amount
  FROM ( customer
         CROSS JOIN product )
       LEFT OUTER JOIN
       ( sales_order
         INNER JOIN sales_order_items
                 ON sales_order_items.id = sales_order.id )
       ON  customer.id = sales_order.cust_id
       AND product.id  = sales_order_items.prod_id
 WHERE customer.state = 'DC'
   AND product.name LIKE '%shirt%'
 GROUP BY customer.company_name,
       product.name,
       product.description
 ORDER BY customer.company_name,
       product.name,
       product.description;
```

最終結果は次のようになります。ワシントンD.C.には2社の顧客があり、5種類のシャツが販売されているため、顧客と製品の組み合わせは10とおりです。5とおりの組み合わせでは、数量（quantity）と総額（amount）にNULL値が示されていて、販売はありませんでした。別の5とおりの組み合わせでは、実際に販売が行われました。

```
company_name             product_name  product_description  quantity  amount
=======================  ============  ==================  ========  =======
Hometown Tee's           Sweatshirt    Hooded Sweatshirt   24        576.00
Hometown Tee's           Sweatshirt    Zipped Sweatshirt   NULL      NULL
Hometown Tee's           Tee Shirt     Crew Neck           NULL      NULL
Hometown Tee's           Tee Shirt     Tank Top            24        216.00
Hometown Tee's           Tee Shirt     V-neck              NULL      NULL
State House Active Wear  Sweatshirt    Hooded Sweatshirt   48        1152.00
State House Active Wear  Sweatshirt    Zipped Sweatshirt   48        1152.00
State House Active Wear  Tee Shirt     Crew Neck           NULL      NULL
State House Active Wear  Tee Shirt     Tank Top            NULL      NULL
State House Active Wear  Tee Shirt     V-neck              60        840.00
```

「スター・ジョイン」は、1つの「ファクト・テーブル」と複数の「ディメンジョン・テーブル」の間のマルチテーブル・ジョインです。絵で説明すると、ファクト・テーブルは星の中心であり、ディメンジョン・テーブルは星の頂点で、中心のファクト・テーブルの回りに配置されます。

ファクト・テーブルには大量のローが格納され、各ローには1つのファクトが格納されます。たとえば、ASADEMO データベースの場合、sales_order テーブルには 600 を超えるローが格納されており、各ローには1つの販売記録が格納されています。ディメンジョン・テーブルには、これらのファクトの属性に関する情報が格納されます。たとえば、customer テーブルには、購入した顧客の名前と住所が格納されています。

各ディメンジョン・テーブルは、ファクト・テーブルが子でディメンジョン・テーブルが親として、外部キー関係でファクト・テーブルに関連付けられます。たとえば、sales_order テーブルは、customer、employee、fin_code という3つのディメンジョン・テーブルと外部キー関係を持ちます。employee テーブルには、注文を受注した営業部員に関する多くの情報が格納されており、fin_code テーブルには、注文の財務会計コードに関する多くの情報が格納されています。

ディメンジョン・テーブルは、通常、ファクト・テーブルよりずっと小さいものです。ASADEMO データベースの場合、sales_order ファクト・テーブルのロー数は、3つのディメンジョン・テーブルをすべて合わせたロー数の3倍です。ディメンジョン・テーブルは、高度に正規化される傾向もあります。たとえば、各顧客の名前と住所は、複数の sales_order ローで繰り返されるのではなく、customer テーブル内の1つのローに格納されます。スター・ジョインは、スター内のテーブル群を非正規化するために使用されます。つまり、スター内のすべてのテーブルからデータを収集し、それを1つの結果セットとして表示します。正規化の詳細については、「1.16　設計の正規化」を参照してください。

スター・ジョインは、FROM 句として表されることもあります。この場合、最初にファクト・テーブルが指定され、その後に、ディメンジョン・テーブルを含む一連の INNER JOIN 演算子が続きます。すべてのジョインの ON 句は、先頭のテーブル、つまりファクト・テーブルを参照します。次に、ある期間内のすべての販売注文と、customer、employee、fin_code の各テーブルの情報を選択する例を示します。sales_order テーブルが、このスター・ジョインの中心となるファクト・テーブルです。

```
SELECT sales_order.order_date        AS order_date,
       sales_order.id                AS order_id,
       customer.company_name         AS customer_name,
       STRING ( employee.emp_fname,
                ' ',
                employee.emp_lname ) AS rep_name,
       fin_code.description          AS fin_code
  FROM sales_order
       INNER JOIN customer
           ON sales_order.cust_id = customer.id
       INNER JOIN employee
           ON sales_order.sales_rep = employee.emp_id
       INNER JOIN fin_code
           ON sales_order.fin_code_id = fin_code.code
```

```
WHERE sales_order.order_date BETWEEN '2000-01-02' AND '2000-01-06'
ORDER BY order_date,
       order_id;
```

スター・ジョインの結果は次のようになります。4つのテーブルが1つの結果セットに事実上「非正規化」されています。

```
order_date  order_id  customer_name          rep_name         fin_code
==========  ========  =====================  ===============  ========
2000-01-02  2131      BoSox Club             Samuel Singer    Fees
2000-01-03  2065      Bloomfields            Samuel Singer    Fees
2000-01-03  2126      Leisure Time           Rollin Overbey   Fees
2000-01-06  2127      Creative Customs Inc.  James Klobucher  Fees
2000-01-06  2135      East Coast Traders     Alison Clark     Fees
```

Section

3.7 SELECT FROMプロシージャ・コール

SQL Anywhereのストアド・プロシージャは結果セットを返すことができ、その結果セットはFROM句の中でテーブルと同じように扱うことができます。

```
<procedure_reference>    ::= [ <owner_name> "."] <procedure_name>
                             "(" [ <argument_list> ] ")"
                             [ WITH "(" <result_definition_list> ")" ]
                             [ [ AS ] <correlation_name> ]
<procedure_name>         ::= <identifier>
<argument_list>          ::= <argument> { "," <argument> }
<argument>               ::= <basic_expression>
                           | <parameter_name> "=" <basic_expression>
<parameter_name>         ::=「第8章　パッケージ」<parameter_name>を参照
<result_definition_list> ::= <result_definition> { "," <result_definition> }
<result_definition>      ::= <alias_name> <data_type>
<data_type>              ::=「第1章　作成」の<data_type>を参照
```

ストアド・プロシージャを使用する利点は、抽出テーブルとビューは単一のクエリとして記述する必要があるのに対して、ストアド・プロシージャには複数の文を指定できることです。難しい問題も、個別のステップに分けることによって簡単になることがあります。たとえば、前の項で示した3つのASADEMOデータベース・テーブル（product、sales_order、sales_order_items）を使用して、売上高が最低だった年で、1日あたりの1色の売上高が2番目と3番目に良かったときに、それに貢献したすべての製品を表示する、という複雑きわまりない要求があるものとします。

分断統治法を使用することで、この問題を解決できます。

- 最初に、総売上高が最低の年を求める
- 次に、その年の中で、1日あたりの1色の売上高が2番目と3番目に良かったものを見つける
- 3番目に、最高の色と受注日の組み合わせに対して、対応する製品を見つける（つまり、その日付で注文された、一致する色を持つ製品を見つける）

各ステップにはそれぞれの問題がありますが、すべての問題を一度に解決する1つのselelctを作成するより、それぞれの問題を個別に解決する方がはるかに簡単です。仮に、すべての操作を実行する1つのクエリを作成できたとしても、他の人がこれを理解するのは難しい可能性があります。場合によっては、正確であるよりも保守しやすいことの方が重要です。

p_best_losers_in_worst_yearという名前のストアド・プロシージャは、最初の2つのステップを実行します。つまり、1つのSELECTで各年の総売上高を計算し、売上高の昇順で結果をソートし、先頭の年を取得して、@worst_yearという名前のローカル変数に格納します。2番目のSELECTでは、@worst_year内の色と日付ごとに総売上高を計算し、総売上高の降順で結果をソートし、結果セットとして2番目と3番目のロー（best losers：準優勝）を返します。

プロシージャは次のようになります。CREATE PROCEDURE文の詳細については、8.9項を参照してください。

```
CREATE PROCEDURE p_best_losers_in_worst_year()
BEGIN
DECLARE @worst_year INTEGER;

-- 総売上高が最低の年を判断する

SELECT FIRST
       YEAR ( sales_order.order_date )
  INTO @worst_year
  FROM product
       INNER JOIN sales_order_items
          ON product.id = sales_order_items.prod_id
       INNER JOIN sales_order
          ON sales_order_items.id = sales_order.id
 GROUP BY YEAR ( sales_order.order_date )
 ORDER BY SUM ( sales_order_items.quantity * product.unit_price ) ASC;

-- 総売上高が最低の年の中で、1日あたりの1色の売上高が
-- 2番目と3番目に良かったものを見つける

SELECT TOP 2 START AT 2
       product.color          AS best_color,
       sales_order.order_date AS best_day,
       SUM ( sales_order_items.quantity * product.unit_price ) AS sales_amount,
```

```
       NUMBER(*) + 1          AS rank
  FROM product
       INNER JOIN sales_order_items
          ON product.id = sales_order_items.prod_id
       INNER JOIN sales_order
          ON sales_order_items.id = sales_order.id
 WHERE YEAR ( sales_order.order_date ) = @worst_year
 GROUP BY product.color,
       sales_order.order_date
 ORDER BY SUM ( sales_order_items.quantity * product.unit_price ) DESC;
END;
```

プロシージャ内の最初のSELECTによって、1つの値が@worst_year変数に設定されます。2番目のクエリにはINTO句がないため、プロシージャが呼び出されると、結果セットは暗黙的に呼び出し元に返されます。

このプロシージャはISQLで次のようにテストできます。

```
CALL p_best_losers_in_worst_year();
```

プロシージャ・コールによって、2番目と3番目の色と日付が売上高と共に次のように返されます。

```
best_color  best_day    sales_amount  rank
==========  ==========  ============  ====
Green       2001-03-24  1728.00       2
Black       2001-03-17  1524.00       3
```

ソリューションの第3ステップでは、プロシージャ・コールをクエリのFROM句内のテーブル形式として使用して、製品の詳細を見つけます。

```
SELECT DISTINCT
       product.id,
       product.name,
       product.description,
       product.color,
       best_loser.rank
  FROM p_best_losers_in_worst_year() AS best_loser
       INNER JOIN product
          ON product.color = best_loser.best_color
       INNER JOIN sales_order_items
          ON product.id = sales_order_items.prod_id
       INNER JOIN sales_order
          ON sales_order_items.id = sales_order.id
         AND sales_order.order_date = best_loser.best_day
 ORDER BY best_loser.rank ASC,
       product.id ASC;
```

このSELECTは次のように動作します。

- p_best_losers_in_worst_year()プロシージャへの参照は、CALLキーワードを指定せずに空の引数リストを指定して記述している。これは、FROM句内のプロシージャ・コールの最低条件である
- "best_loser"という相関名が定義されているが、必須ではない。相関名を明示的に指定しなかった場合、以降のクエリでは、プロシージャ名自体が相関名として使用される
- FROM句はINNER JOIN演算子を使用して、best_loser内のローと、他の3つのテーブル（product、sales_order_items、sales_order）内のローをジョインして、色と販売日が一致する組み合わせを見つける
- 最後に、selectリストは、productに含まれるカラムと、best_loserに含まれるランク（2位または3位）を返す。同じ製品が同じ日の複数の注文に含まれている可能性があるが、それぞれ異なる製品のみを参照する必要があるため、DISTINCTキーワードを使用している

最終結果は次のようになります。1つの緑色の製品が2番目の売上日に貢献し、3つの黒色の製品が3番目の売上日に貢献したことがわかります。

```
id   name          description        color  rank
===  ============  =================  =====  ====
600  Sweatshirt    Hooded Sweatshirt  Green  2
302  Tee Shirt     Crew Neck          Black  3
400  Baseball Cap  Cotton Cap         Black  3
700  Shorts        Cotton Shorts      Black  3
```

ストアド・プロシージャは、2種類の方法で、その結果セットに対してカラム名を指定できます。1つは、selectリスト内の各項目にカラム名またはエイリアス名を指定する方法で、もう1つは、CREATE PROCEDURE文に明示的なRESULT句を指定する方法です。どちらの方法もオプションですが、FROM句内のストアド・プロシージャ参照では問題が生じることがあります。たとえば、「NUMBER(*) + 1」という式に、上記のp_best_losers_in_worst_yearプロシージャで明示的に指定されている"rank"というエイリアス名が含まれない場合、最後のselectリスト内でbest_loser.rankへの参照は使用できません。

もう1つのソリューションは、FROM句内のプロシージャ参照に明示的なWITHリストを追加する方法です。このWITHリストは、このFROM句に関する限り、プロシージャ結果セット内の各カラムで使用されるエイリアス名とデータ型を指定します。ストアド・プロシージャが、その結果セット内のカラムに対して名前を指定する場合でもあっても、WITHリストの名前はそれらを上書きします。次に、上記のSELECTに、プロシージャが返す名前とは異なる2つのエイリアス名を指定する明示的なWITHリストを追加した例を示します。

```
SELECT DISTINCT
       product.id,
       product.name,
       product.description,
       product.color,
       best_loser.ranking
  FROM p_best_losers_in_worst_year()
         WITH ( best_color VARCHAR ( 6 ),
                best_day   DATE,
                best_sales NUMERIC ( 15, 2 ),
                ranking    INTEGER )
           AS best_loser
       INNER JOIN product
          ON product.color = best_loser.best_color
       INNER JOIN sales_order_items
          ON product.id = sales_order_items.prod_id
       INNER JOIN sales_order
          ON sales_order_items.id = sales_order.id
         AND sales_order.order_date = best_loser.best_day
 ORDER BY best_loser.ranking ASC,
       product.id ASC;
```

プロシージャの引数リストが空の場合、またはプロシージャが定数引数のみを受け取る場合、FROM句内のプロシージャ参照は1回だけ実行され、結果セットは1回だけ実体化されます。これは、場合によっては適切であり、場合によっては厄介です。プロシージャが多くの不要なローを返す場合、クエリ・プロセッサは呼び出しを最適化せず、定義可能な等価なビュー参照または抽出テーブルより、プロシージャ参照のパフォーマンスが悪化することがあります。一方、プロシージャが1回だけ確実に呼び出され、結果セットが実体化されることがわかっていると、複雑な問題を解決するうえでの糸口になります。

ここでの「実体化」は、結果セットが完全に評価され、メモリに格納されるか、またはメモリ不足の場合はテンポラリ・ファイルに格納されることを意味します。また、「定数引数」は、FROM句の評価中に値が変化しない引数を意味します。リテラル、プログラム変数、リテラルと変数を含む式がこのカテゴリに分類されますが、FROM句内の他のテーブル内のカラムへの参照は、このカテゴリに分類されません。

次の項では、変数引数、つまりFROM句内の別のテーブルに含まれるカラムを受け取るプロシージャについて説明します。

Section

3.8 ラテラル・プロシージャ・コール

別のテーブルに含まれるカラムが、FROM 句内のプロシージャ参照に引数として渡される場合、そのプロシージャ参照は、ラテラル抽出テーブル定義の一部として指定される必要があります。また、別のテーブルはラテラル抽出テーブル定義より前に指定し、INNER JOIN などのジョイン演算子の 1 つではなく、カンマで区切る必要があります。「カンマ・ジョイン演算子」を使用する状況の 1 つがこれであり、ON 条件は使用できません。

次に、ラテラル抽出テーブルの一般的な構文を示します。

```
<lateral_derived_table> ::= LATERAL
                                <subquery>
                                [ AS ] <correlation_name>
                                [ <derived_column_name_list> ]
                            | LATERAL
                                "(" <table_expression> ")"
                                [ AS ] <correlation_name>
                                [ <derived_column_name_list> ]
```

次に、テーブルとプロシージャ参照との間のジョインの簡略構文を示します。これは、テーブルに含まれるカラムを引数として渡す場合の構文であり、カンマ・ジョインと LATERAL キーワードの使用方法として本書で説明する唯一のものです。

```
<typical_lateral_procedure_call> ::= <table_name> ","
                                        LATERAL "(" <procedure_name>
                                           "(" <table_name>.<column_name> ")" ")"
                                           AS <correlation_name>
```

顧客 ID を引数として受け取り、その顧客のすべての注文情報が含まれる結果セットを返すプロシージャの例を次に示します。

```
CREATE PROCEDURE p_customer_orders ( IN @customer_id INTEGER )
BEGIN
MESSAGE STRING ( 'DIAG ', CURRENT TIMESTAMP, ' ', @customer_id ) TO CONSOLE;
SELECT sales_order.order_date         AS order_date,
       product.name                   AS product_name,
       product.description            AS description,
       sales_order_items.quantity     AS quantity,
       product.unit_price
         * sales_order_items.quantity AS amount
  FROM sales_order
```

```
       INNER JOIN sales_order_items
               ON sales_order_items.id = sales_order.id
       INNER JOIN product
               ON product.id = sales_order_items.prod_id
 WHERE sales_order.cust_id = @customer_id
 ORDER BY order_date,
       product_name,
       description;
END;

CALL p_customer_orders ( 141 );
```

ASADEMOデータベースを使用した場合の、顧客ID 141のCALLの結果は次のようになります。

```
order_date  product_name  description    quantity  amount
==========  ============  =============  ========  ======
2000-11-19  Shorts        Cotton Shorts  36        540.00
2001-02-26  Baseball Cap  Cotton Cap     12        108.00
```

次に、select内のFROM句でこのプロシージャを呼び出す例を示します。このselectでは、顧客ID 141の代わりに、会社名Mall Side Sportsを指定しています。customerテーブルはカンマ・ジョイン演算子でプロシージャ・コールにジョインします。プロシージャ・コールは、customer.idカラムを引数として渡すため、ラテラル抽出テーブル定義の一部として呼び出しています。

```
SELECT customer.company_name,
       customer_orders.*
  FROM customer,
       LATERAL ( p_customer_orders ( customer.id ) ) AS customer_orders
 WHERE customer.company_name = 'Mall Side Sports'
 ORDER BY customer_orders.order_date,
       customer_orders.product_name,
       customer_orders.description;
```

最終結果は次のようになります。前と同じデータに会社名が追加されています。

```
company_name      order_date  product_name  description    quantity  amount
================  ==========  ============  =============  ========  ======
Mall Side Sports  2000-11-19  Shorts        Cotton Shorts  36        540.00
Mall Side Sports  2001-02-26  Baseball Cap  Cotton Cap     12        108.00
```

Note

カンマ・ジョイン演算子の使用は避けるべきです。INNER JOINなどのジョイン演算子やON条件を使用する方が、FROM句がはるかに理解しやすくなります。しかし、この特殊なケースではカンマ・ジョイン演算子を使用する必要があり、INNER JOINの機能として考えることができます。

Tip

FROM 句内のプロシージャ・コールは、記述方法に応じて、1 回でも、何百万回でも呼び出される可能性があります。上の例に示したような MESSAGE 文を追加することによって、プロシージャの呼び出し回数を簡単に確認できます。呼び出しごとに、データベース・エンジン・コンソールに行が表示されます。

Section 3.9

SELECTリスト

select の論理的実行の 2 番目のステップでは、集合関数呼び出しと NUMBER(*) 呼び出しを除く、すべての select リスト項目が評価され、FROM 句で返される仮想テーブル内の各ローに値が追加されます。

```
<select_list>       ::= <select_item> { "," <select_item> }
<select_item>       ::= "*"
                        | [ <owner_name> "."] <table_name> "." "*"
                        | <correlation_name> "." "*"
                        | <expression>
                        | <expression> [ AS ] <select_item_alias>
<select_item_alias> ::= <alias_name>     -- 非常に有用
                        | <string_literal> -- あまり有用でない
<string_literal>    ::= 一重引用符で囲んだ文字列
```

アスタリスク "*" は、FROM 句内のすべてのテーブルに含まれるすべてのカラムを表します。テーブルの順序は、FROM 句に指定されているテーブルの順序と同じであり、各テーブル内のカラムの順序は、CREATE TABLE 文に指定されている順序と同じです。

"*" という表記は、他の select リスト項目と組み合わせて使用することもあります。つまり、「SELECT * FROM …」に限定されません。これは、次の例に示すように、「製品名カラムと、テーブル内の他のすべてのカラムを表示する」というクイック・クエリを実行するうえで役立ちます。

```
SELECT product.name,
       *
  FROM product
       INNER JOIN sales_order_items
               ON sales_order_items.prod_id = product.id
       INNER JOIN sales_order
               ON sales_order.id = sales_order_items.id
 ORDER BY product.name,
       sales_order.order_date DESC;
```

テーブル名に"*"を修飾することで、このテーブル内のすべてのカラムを表すことができます。カラムの順序は、CREATE TABLE文に指定した順序と同じです。selectリスト内での反復に制限はありません。次に、「製品名、sales_order_items内のすべてのカラム、すべてのテーブル内のすべてのカラムを表示する」というクエリの例を示します。

```
SELECT product.name,
       sales_order_items.*,
       *
  FROM product
       INNER JOIN sales_order_items
               ON sales_order_items.prod_id = product.id
       INNER JOIN sales_order
               ON sales_order.id = sales_order_items.id
 ORDER BY product.name,
       sales_order.order_date DESC;
```

Tip

アプリケーション・プログラムでは、通常、アスタリスクの"*"表記を使用するより、selectリストにすべてのカラム名を明示的に指定する方が適当です。

selectリスト内の個々の項目（アスタリスク"*"表記を使用しないもの）には、エイリアス名を割り当てることができます。selectリスト内の別の場所や他の句でこの名前を使用して、このselectリスト項目を参照できます。selectリスト内のカラム名の場合、エイリアス名はオプションです。エイリアス名の有無にかかわらず、カラム名自体を使用して該当の項目を参照できるためです。selectリスト項目が式の場合、そのselectリスト項目を別のロケーションで名前で参照する場合は、エイリアス名は必須です。

Tip

キーワードASはオプションですが、エイリアス名を定義する場合は、どれがエイリアス名でどれがselectリスト項目かを明確にするために、常にキーワードASを使用してください。

Tip

文字列リテラルではなく、識別子をエイリアス名として使用します。selectリストでのみ、文字列リテラルをエイリアスとして使用できますが、そのようなエイリアスの使い方は、他の場面では適用できません。エイリアス名を使用できる他のすべての場面（抽出テーブル定義、CREATE VIEW文、WITH句など）では、識別子のみが許されているので、selectリストでもこれを使用するとよいでしょう。

式やカラム参照など、selectリスト内の個々の項目については、以降の項で詳しく説明します。

Section

3.10 式と演算子

selectリストには、アスタリスクとカラム名以外のものも指定できます。評価されて1つの値を返すのであれば、非常に複雑な式も使用できます。実際、<expression>の構文では、単純な<column_reference>はほとんど使用されません。

```
<expression>          ::= <basic_expression>
                        | <subquery>
<basic_expression>    ::= <simple_expression>
                        | <if_expression>
                        | <case_expression>
<simple_expression>   ::= "(" <basic_expression> ")"       -- 優先度：
                        | "-" <expression>                    -- 1。単項マイナス
                        | "+" <expression>                    -- 1。単項プラス
                        | "~" <expression>                    -- 1。ビット処理NOT
                        | <simple_expression> "&" <expression>  -- 2。ビット処理AND
                        | <simple_expression> "|" <expression>  -- 2。ビット処理OR
                        | <simple_expression> "^" <expression>  -- 2。ビット処理XOR
                        | <simple_expression> "*" <expression>  -- 3。乗算
                        | <simple_expression> "/" <expression>  -- 3。除算
                        | <simple_expression> "+" <expression>  -- 4。加算
                        | <simple_expression> "-" <expression>  -- 4。減算
                        | <simple_expression> "||" <expression> -- 5。連結
                        | <column_reference>
                        | <variable_reference>
                        | <string_literal>
                        | <number_literal>
                        | <special_literal>
                        | NULL
                        | <function_call>
<column_reference>    ::= <column_name>
                        | <alias_name>
                        | [ <owner_name> "."] <table_name> "."<column_name>
                        | <correlation_name> "."<column_name>
<variable_reference> ::= SQL変数への参照
<number_literal>      ::= 整数（正確な数値リテラルまたは浮動小数点数リテラル）
<special_literal>     ::=「第1章　作成」の<special_literal>を参照
```

<expression>の構文は、selectリスト項目の要件を満たすことより複雑です。これは、SQLの他の多くの場所で式が使用され、コンテキストによっては、式を構成する要素や構成しない要素に関して制限を課すためです。特に、上で定義されているように、式には3種類あります。

- 1番目は、フル機能の <expression>。SQL Anywhere が提供するすべての機能を備えている。これは select リストで使用できる種類であり、この項ではこの式について説明する
- 2番目は <basic_expression>。サブクエリを除いて、<expression> と同じ機能を備えている。たとえば、<case_expression> では、CASE キーワードの後にサブクエリを指定できない。こういったコンテキストでは、<basic_expression> の構文を用いる
- 3番目は <simple_expression>。<basic_expression> と似ているが、IF キーワードまたは CASE キーワードで始まることはできない。たとえば、RAISERROR 文のメッセージ・テキスト・パラメータは、<simple_expression> 以上に工夫を凝らすことはできない

実際には、これらの違いは非常に些細なものであり、その違いが問題になることはまずありません。そこで、本書では、式はあくまでも式として扱い、BNF でのみその違いを示します。

Tip

1つの式の中で複数の算術演算子を使用する場合は、カッコを使用して計算の順序を明確にします。カッコがない場合のデフォルトの順序では、乗算と除算が最初に実行され、次に加算と減算が実行されます。すべての人がこの順序を理解し、覚えているとは限らないため、コードを理解しやすいものとするために、カッコを使用するのが適当です。

次に、1つの句のみ（つまり select リストのみ）を含む SELECT の例を示します。1番目と3番目の式では、特殊リテラルの CURRENT DATE から、1日を減算および加算して前日と翌日の日付をそれぞれ計算しています。最後の4つの select リスト項目は、product.unit_price の最大値、product テーブルと sales_order テーブル内のロー数、sales_order_items.quantity 値の合計値をそれぞれ計算するサブクエリです。

```
SELECT CURRENT DATE - 1            AS yesterday,
       CURRENT DATE                AS today,
       CURRENT DATE + 1            AS tomorrow,
       ( SELECT MAX ( unit_price )
           FROM product )          AS max_price,
       ( SELECT COUNT(*)
           FROM product )          AS products,
       ( SELECT COUNT(*)
           FROM sales_order )      AS orders,
       ( SELECT SUM ( quantity )
           FROM sales_order_items ) AS items;
```

結果は次のようになります。

```
yesterday   today       tomorrow    max_price  products  orders  items
==========  ==========  ==========  =========  ========  ======  =====
2003-10-17  2003-10-18  2003-10-19  24.00      10        648     28359
```

Note

デフォルトのFROM句は、実際には"FROM SYS.DUMMY"です。たとえば、"SELECT *"文が実行されると、0値が設定されたdummy_colという1つのカラムから成る1つのローが返されます。これは、組み込みの読み取り専用のSYS.DUMMYテーブルの内容とまったく同じです。上の例に示すように、これが、FROM句が指定されないSELECTが常に1つのローを返す理由です。

次に、いくつかの算術演算子を使用してselectリスト内で計算を実行する例を示します。

```
SELECT product.id,
       product.unit_price * product.quantity   AS stock_value,
       product.unit_price
          * ( SELECT SUM ( quantity )
                FROM sales_order_items
               WHERE sales_order_items.prod_id
                   = product.id )              AS sales_value,
       ( stock_value / sales_value ) * 100.00  AS percent
  FROM product
ORDER BY sales_value DESC;
```

これは次のように動作します。productテーブル内のすべてのローについて、unit_priceとquantityを乗算し、在庫の総価格であるstock_valueを決定します。また、サブクエリは、sales_order_itemsローのうちprod_idがproduct.idと一致するすべてのローを取得し、productテーブル内の各ローについてすべてのsales_order_items.quantityの合計を計算します。この合計値にproduct.unit_priceを乗算して、その製品の総売上高であるsales_valueを計算します。最後に、stock_valueおよびsales_valueというエイリアス名を参照することによって、前の2つの計算結果に対してパーセンテージの計算を実行します。これをASADEMOデータベースに対して実行し、sales_valueで降順にソートした結果は次のようになります。

```
id   stock_value  sales_value  percent
===  ===========  ===========  ========
600   936.00      73440.00     1.274510
700  1200.00      68040.00     1.763668
601   768.00      65376.00     1.174743
301   756.00      33432.00     2.261307
302  1050.00      30072.00     3.491620
400  1008.00      29502.00     3.416718
401   120.00      27010.00      .444280
300   252.00      21276.00     1.184433
500   252.00      18564.00     1.357466
501   196.00      17556.00     1.116427
```

Tip

スプレッドシート内のセル名と同じようにエイリアス名を使用して、式のコードを繰り返すことなく、他の式の結果から新しい式を作成できます。エイリアス名を定義し、それを同じクエリ内の別の場所（たとえば、別の select リスト項目や WHERE 句）で参照できる機能は、SQL Anywhere 独自の機能です。

3.10.1 IF式とCASE式

IF キーワードと CASE キーワードを使用すると、IF-THEN-ELSE 文と CASE 文の作成に加えて、式を作成することもできます。IF-THEN-ELSE 文と CASE 文については「第 8 章　パッケージ」で説明し、式についてはここで説明します。

```
<if_expression> ::= IF <boolean_expression>
                       THEN <expression>
                    [ ELSE <expression> ]
                    ENDIF
```

IF 式は <boolean_expression> を評価して、TRUE、FALSE、または UNKNOWN を判断します。<boolean_expression> の結果が TRUE の場合、IF の結果として THEN <expression> が返されます。<boolean_expression> が FALSE の場合、IF の結果として ELSE <expression> が返されます。ELSE <expression> がない場合、または <boolean_expression> が UNKNOWN の場合、IF の結果として NULL が返されます。

THEN 式と ELSE 式は、さらにネストされた IF 式を含めて、<expression> の構文で使用可能などのような内容でもかまいません。次に、ある製品に対して 'Understocked' および 'Overstocked' を表示し、他の製品に対しては空の文字列を表示する例を示します。

```
SELECT product.id,
       product.quantity,
       IF product.quantity < 20
       THEN 'Understocked'
       ELSE IF product.quantity > 50
            THEN 'Overstocked'
            ELSE ''
            ENDIF
       ENDIF AS level
  FROM product
 ORDER BY product.quantity;
```

この例を ASADEMO データベースに対して実行した場合の結果は次のようになります。

```
id   quantity  level
===  ========  ============
401  12        Understocked
300  28
501  28
601  32
500  36
600  39
301  54        Overstocked
302  75        Overstocked
700  80        Overstocked
400  112       Overstocked
```

TRUE、FALSE、UNKNOWN、およびNULLとの関係の詳細については、「3.12　ブール式とWHERE句」を参照してください。

CASE式には2つの形式があります。

```
<case_expression>       ::= <basic_case_expression>
                          | <searched_case_expression>

<basic_case_expression> ::= CASE <basic_expression>
                               WHEN <expression> THEN <expression>
                             { WHEN <expression> THEN <expression> }
                             [ ELSE <expression> ]
                            END
```

上に示した最初の形式では、CASE <basic_expression>を評価し、それを各WHEN <expression>の値と順番に比較します。この比較では、等号"="演算子を暗黙的に使用します。この比較の結果は、TRUE、FALSE、またはUNKNOWNです。TRUEの結果が生じると、対応するTHEN <expression>が評価され、CASEの結果として返されます。これで、CASEの処理が終了します。すべての比較の結果がFALSEまたはUNKNOWNの場合は、ELSE <expression>が評価され、返されます。ELSE <expression>がない場合は、NULLが返されます。

次に、基本のCASE式を使用して、sales_order.region内の文字列値をソートに適した数値に変換する例を示します。CASE式の結果にはsort_orderというエイリアス名を設定しています。このエイリアス名は、WHERE句とORDER BY句で参照できます。

```
SELECT CASE region
          WHEN 'Western' THEN 1
          WHEN 'Central' THEN 2
          WHEN 'Eastern' THEN 3
          ELSE 0
       END      AS sort_order,
       region,
       COUNT(*) AS orders
```

```
 FROM sales_order
WHERE sort_order > 0
GROUP BY region
ORDER BY sort_order;
```

結果は次のようになります。明示的なソート順が定義されただけでなく、その3つの地域以外のすべての注文が除外されています。

```
sort_order  region   orders
==========  =======  ======
1           Western  61
2           Central  224
3           Eastern  244
```

2番目の形式のCASE式は柔軟性に優れています。暗黙の等号"="演算子に縛られることがなく、WHEN比較の左側に指定できるものも、1つのCASE比較値に限りません。

```
<searched_case_expression> ::= CASE
                                 WHEN <boolean_expression> THEN <expression>
                               { WHEN <boolean_expression> THEN <expression> }
                               [ ELSE <expression> ]
                              END
```

各WHEN <boolean_expression>が順番に評価され、その結果はTRUE、FALSE、またはUNKNOWNです。TRUEの結果が生じると同時に検索が終了し、対応するTHEN <expression>が評価され、CASEの結果として返されます。すべての結果がFALSEまたはUNKNOWNの場合は、ELSE <expression>が評価され、それが結果として返されます。ELSE <expression>がない場合は、NULLが返されます。

次に、検索CASE式を使用して、単純な比較に加えてANDとINを使用する3つのWHEN条件を指定する例を示します。さらに、もう1つ基本CASE式を指定して、最初の式の結果を文字列タイトルに変換します。

```
SELECT CASE
          WHEN sales_rep = 129
           AND region = 'Western'
             THEN 1
          WHEN region = 'Western'
             THEN 2
          WHEN region IN ( 'Eastern', 'Central' )
             THEN 3
          ELSE 0
       END AS sort_order,
       CASE sort_order
```

```
          WHEN 1 THEN 'Western 129'
          WHEN 2 THEN 'Other Western'
          WHEN 3 THEN 'Eastern and Central'
       END      AS breakdown,
       COUNT(*) AS orders
  FROM sales_order
 WHERE sort_order > 0
 GROUP BY sort_order
 ORDER BY sort_order;
```

ASADEMOデータベースを使用した場合の結果は次のようになります。

```
sort_order  breakdown            orders
==========  =============        ======
1           Western 129          6
2           Other Western        55
3           Eastern and Central  468
```

Section

3.11 上位15位のスカラ組み込み関数

関数呼び出しは4つのカテゴリに分かれます。1番目は、CREATE FUNCTION文で作成したユーザ定義関数への参照です。2番目は、他の言語が備えている関数によく似た、ABS()やSUBSTRING()などの通常の組み込み関数です。3番目は、CAST()やNUMBER(*)など、5種類ほどの特殊な組み込み関数です。これらは通常の組み込み関数と同じように動作しますが、引数リストの構文が普通とは異なります。最後は、それ自体で完結する集合組み込み関数です。

```
<function_call> ::= <user_defined_function_call>      -- スカラ関数
                  | <ordinary_builtin_function_call>  -- スカラ関数
                  | <special_builtin_function_call>   -- スカラ関数
                  | <aggregate_builtin_function_call> -- 集合関数
<user_defined_function_call>     ::= <user_defined_function_name>
                                       "(" [ <function_argument_list> ] ")"
<user_defined_function_name>     ::= <identifier>
<function_argument_list>         ::= <expression> { "," <expression> }
<ordinary_builtin_function_call> ::= <ordinary_builtin_function_name>
                                       "(" [ <function_argument_list> ] ")"
<ordinary_builtin_function_name> ::= <identifier>
<special_builtin_function_call>  ::= CAST "(" <expression> AS <data_type> ")"
                                   | NOW "( * )"
                                   | NUMBER "( * )"
                                   | PI "( * )"
                                   | TODAY "( * )"
                                   | TRACEBACK "( * )"
```

最初の3つのカテゴリは、selectリスト内に現れた場合、ローごとに1回実行されるため、スカラ関数と呼ばれます。これに対して集合関数は、複数のローに対して実行されます。

この項では、通常のスカラ組み込み関数と特殊なスカラ組み込み関数の両方について説明します。ただし、NUMBER(*)は例外で、これについては3.20項で説明します。集合関数については3.14項、ユーザ定義関数については8.10項をそれぞれ参照してください。

SQL Anywhere 9には、およそ175種類の組み込み関数が用意されています。REPEAT()やREPLICATE()などを異なる関数として数えるかどうかによって（機能的には同じ）、その個数は変わります。1冊の本でそのすべてをくまなく説明するのは不可能であり、正直なところ、それほどの価値がないものもあります。NOW(*)についてどんなに説明したところで、CURRENT TIMESTAMPを返す、以上のことは説明のしようがありません。

SQL Anywhereの長所の1つは多彩な組み込み関数にあります。SQL Anywhereのオンラインヘルプでは、そのすべてが適切に説明されています。ここでは、（筆者の独断で）最も役に立つ上位15位の関数をアルファベット順に示します。

表3-1　上位15位の組み込みスカラ関数

関数	説明
CAST (p AS q)	pをデータ型qに変換して返す
COALESCE (p, q, ...)	最初の非NULLパラメータを返す
LEFT (p, q)	文字列pの左端のq文字を返す
LENGTH (p)	文字列pの現在の長さを返す
LOCATE (p, q [, r])	rが指定されている場合はrを先頭に検索を開始して、文字列p内での文字列qの先頭の位置を返す
LOWER (p)	文字列pを小文字に変換して返す
LTRIM (p)	文字列pの先行ブランクを削除して返す
REPEAT (p, q)	文字列pのq個のコピーを連結して返す
REPLACE (p, q, r)	文字列p内の文字列qをすべて文字列rに置き換えて返す
RIGHT (p, q)	文字列pの右端のq文字を返す
RTRIM (p)	文字列pの後続ブランクを削除して返す
STRING (p, ...)	文字列に変換され、結合された各パラメータから成る文字列を返す
SUBSTR (p, q [, r])	qを先頭に長さr、またはrが省略されている場合はpの末尾までの部分文字列を返す
TRIM (p)	文字列pの先行ブランクと後続ブランクを削除して返す
UPPER (p)	文字列pを大文字に変換して返す

CAST関数は、あるデータ型から別のデータ型への変換を実行します。たとえば、CAST ('123' AS INTEGER) は、文字列'123'をINTEGERの123に変換します。

CASTは、明らかなデータ変換エラーがあると失敗します。また、いくつかの微妙な制限もあります。たとえば、CAST (123.456 AS INTEGER) は適切に動作して、123.456を切り捨てて123を返し

ますが、CAST ('123.456' AS INTEGER) は失敗します。この変換は、CAST (CAST ('123.456' AS NUMERIC) AS INTEGER) という 2 段階に分けて行う必要があります。

それにもかかわらず、CAST は非常に有用です。次に、その柔軟性を表す別の例を示します。

```
CREATE TABLE t1 (
   key_1         UNSIGNED BIGINT NOT NULL,
   non_key_1     VARCHAR ( 100 ) NOT NULL,
   last_updated  TIMESTAMP NOT NULL,
   PRIMARY KEY ( key_1 ) );

INSERT t1 VALUES ( 1, '123.45', '2003-10-19 15:32.25.123' );

SELECT CAST ( key_1 AS VARCHAR ( 1 ) )         AS a,
       CAST ( key_1 AS VARCHAR )               AS b,
       CAST ( non_key_1 AS NUMERIC ( 10, 2 ) ) AS c,
       CAST ( non_key_1 AS NUMERIC )           AS d,
       CAST ( last_updated AS DATE )           AS e,
       CAST ( last_updated AS TIME )           AS f
  FROM t1;
```

結果は次のようになります。2 番目の CAST では、b を VARCHAR (21) として返します。これは、BIGINT で必要な最大サイズです。また、4 番目の CAST では、d を NUMERIC (30, 6) として返します。これは、NUMERIC データ型のデフォルトの位取りと精度です。通常、CAST は適切な動作を試みます。

```
a    b    c       d           e           f
===  ===  ======  ==========  ==========  ============
'1'  '1'  123.45  123.450000  2003-10-19  15:32:25.123
```

EXPRTYPE 関数を使用すると、CAST が返す内容を確認できます。次に、b が VARCHAR (21) として返されることを証明する例を示します。

```
SELECT EXPRTYPE ( '
SELECT CAST ( key_1 AS VARCHAR ( 1 ) )         AS a,
       CAST ( key_1 AS VARCHAR )               AS b,
       CAST ( non_key_1 AS NUMERIC ( 10, 2 ) ) AS c,
       CAST ( non_key_1 AS NUMERIC )           AS d,
       CAST ( last_updated AS DATE )           AS e,
       CAST ( last_updated AS TIME )           AS f
  FROM t1
', 2 );
```

COALESCE 関数は、名前は変わっていますが、非常に単純で有用です。この関数は、各パラメータを左から右に向かって評価し、NULL でない最初のパラメータを返します。COALESCE には 2 つ以上のパラメータを指定できますが、ほとんどの場合、指定されるパラメータは、カラム名と、カラム値が

NULL の場合に使用される値の 2 つです。次に、テーブル内の NULL 値を非 NULL 値に置換する例を示します。

```
CREATE TABLE t1 (
   key_1         UNSIGNED BIGINT NOT NULL,
   non_key_1     VARCHAR ( 100 ) NULL,
   non_key_2     TIMESTAMP NULL,
   PRIMARY KEY ( key_1 ) );

INSERT t1 VALUES ( 2, NULL, NULL );

SELECT COALESCE ( non_key_1, 'empty' )           AS a,
       COALESCE ( non_key_2, CURRENT TIMESTAMP ) AS b
  FROM t1;
```

SELECT の結果は次のようになります。

```
a        b
=======  ======================
'empty'  2003-10-19 15:58:36.176
```

COALESCE を使用すると、WHERE 句内で IS NOT NULL 比較を使用しなくて済みます。また、クエリから非 NULL 値のみを返すことによって、アプリケーション・プログラム内でインジケータ変数を使用しなくても済みます。すべてのテーブル内のすべての単一カラムが NOT NULL として宣言されている場合でも、結果セットには NULL 値が表示されるため、これは便利です。なぜなら、すべての OUTER JOIN 演算子は、欠如しているローを表すために NULL 値を生成するからです。

たとえば、「3.6　マルチテーブル・ジョイン」の「販売のなかった製品と顧客の組み合わせを含めて、各種類のシャツが、ワシントン D.C. の各顧客に対して何枚売れたかを示す」というクエリがこの状況に当てはまります。結果には、販売のなかった顧客と製品の組み合わせを表す NULL 値が含まれていました。次に、数量と総額の NULL を 0 に変換する COALESCE 呼び出しを指定した同じクエリを示します。

```
SELECT customer.company_name                AS company_name,
       product.name                         AS product_name,
       product.description                  AS product_description,
       COALESCE (
          SUM ( sales_order_items.quantity ),
          0.00 )                            AS quantity,
       COALESCE (
          SUM ( product.unit_price
              * sales_order_items.quantity ),
          0.00 )                            AS amount
  FROM ( customer
         CROSS JOIN product )
       LEFT OUTER JOIN
       ( sales_order
```

```
        INNER JOIN sales_order_items
                ON sales_order_items.id = sales_order.id )
      ON  customer.id = sales_order.cust_id
      AND product.id  = sales_order_items.prod_id
WHERE customer.state = 'DC'
  AND product.name LIKE '%shirt%'
 GROUP BY customer.company_name,
      product.name,
      product.description
ORDER BY customer.company_name,
      product.name,
      product.description;
```

結果セットには0が表示され、どの顧客に対してどの製品の販売が0であったかがわかります。

```
company_name             product_name  product_description  quantity  amount
=======================  ============  ==================  ========  =======
Hometown Tee's           Sweatshirt    Hooded Sweatshirt    24         576.00
Hometown Tee's           Sweatshirt    Zipped Sweatshirt    0            0.00
Hometown Tee's           Tee Shirt     Crew Neck            0            0.00
Hometown Tee's           Tee Shirt     Tank Top             24         216.00
Hometown Tee's           Tee Shirt     V-neck               0            0.00
State House Active Wear  Sweatshirt    Hooded Sweatshirt    48        1152.00
State House Active Wear  Sweatshirt    Zipped Sweatshirt    48        1152.00
State House Active Wear  Tee Shirt     Crew Neck            0            0.00
State House Active Wear  Tee Shirt     Tank Top             0            0.00
State House Active Wear  Tee Shirt     V-neck               60         840.00
```

LEFT、RIGHT、SUBSTRの各関数はすべて、文字列パラメータから部分文字列を返します。LEFT関数は先頭から文字をカウントし、RIGHT関数は右からカウントします。SUBSTR関数は開始位置と長さを使用します。SUBSTR呼び出しで長さパラメータを省略すると、文字列の末尾までのすべての文字が対象です。3つの関数はすべて、文字列操作プロセスの基本構成要素であり、LONG VARCHARパラメータも適切に処理します。次に、いくつかの例を示します。

```
SELECT LEFT   ( '12345', 2 )    AS a,
       RIGHT  ( '12345', 2 )    AS b,
       SUBSTR ( '12345', 2, 3 ) AS c,
       SUBSTR ( '12345', 2 )    AS d;
```

結果は次のようになります。

```
a     b     c      d
====  ====  =====  ======
'12'  '45'  '234'  '2345'
```

Note

SQL Anywhere のすべての文字列関数は、文字列位置のカウントを 0 ではなく 1 で開始します。これは SQL であり、C ではありません。そのため、0 ベースのオフセットや 0 バイトの文字列ターミネータはありません。

LENGTH 関数も文字列操作構成要素であり、文字列パラメータの現在の長さを返します。たとえば、LENGTH (SUBSTR ('12345', 2)) は 4 を返します。

LOCATE 関数は、ある文字列内で最初に出現する別の文字列を検索し、目的の文字列が見つかった場合はその位置を返し、見つからなかった場合は 0 を返します。たとえば、LOCATE ('A=B+C', '=') は 2 を返します。

検索の開始位置を指定するオプションの第 3 パラメータによって、LOCATE 呼び出しを簡単に繰り返すことができます。たとえば、LOCATE ('=A=B+C', '=', 2) は 3 を返します。

LOCATE の戻り値は、常に、文字列全体を基準とする文字位置であり、開始位置が基準ではありません。このため、続けて SUBSTR を呼び出す場合に、この戻り値は便利です。たとえば、次の SELECT は 'B+C' を返します。

```
CREATE TABLE t1 (
   key_1         UNSIGNED BIGINT NOT NULL,
   non_key_1     VARCHAR ( 100 ) NOT NULL,
   PRIMARY KEY ( key_1 ) );

INSERT t1 VALUES ( 1, '=A=B+C' );

SELECT SUBSTR ( non_key_1,
                LOCATE ( non_key_1, '=', 2 ) + 1 )
  FROM t1;
```

LOWER 関数は文字列パラメータをすべて小文字に変換し、UPPER はパラメータを大文字に変換します。たとえば、LOWER ('Hello, World') は 'hello, world' を返し、UPPER ('Hello, World') は 'HELLO, WORLD' を返します。

LTRIM、RTRIM、TRIM の各関数はすべて、文字列パラメータから選択されたブランクを削除し、結果を返します。LTRIM は先行ブランクを削除し、RTRIM は後続ブランクを削除し、TRIM は両方を削除します。たとえば、LTRIM (' AB CD ') は 'AB CD ' を返し、RTRIM (' AB CD ') は ' AB CD' を返し、TRIM (' AB CD ') は 'AB CD' を返します。どの関数の場合も、文字列の内側に埋め込まれたブランクは影響を受けません。

REPLACE 関数は、ある文字列をすべて別の文字列に変更して、結果を返します。たとえば、SELECT REPLACE (' Blah blah blah. ', ' ', '') は、先行ブランク、後続ブランク、埋め込みブランクをすべて削除して、'Blahblahblah.' を返します。

REPEAT 関数は文字列パラメータを受け取り、指定の数のコピーをすべて連結して返します。たと

えば、REPEAT ('AB', 3) は 'ABABAB' を返します。他のいくつかの構成要素関数と同様、REPEAT は単独で使用するより、他の関数と組み合わせた方が効果的です。次の STRING 関数の使用例の中でこのことを示します。

STRING 関数は、最初、それほど役立つようには見えません。可変の数のパラメータを受け取り、それらをすべて文字列に変換し、その文字列をすべて連結して、結果を返すだけです。しかし、他の関数と組み合わせて使用する場合、STRING は驚くほど便利なことがわかります。特に、文字列の長さに制限がないことは重宝します。

次に、REPEAT、STRING、RIGHT を組み合わせて、10 進数を、左側に 0 が埋め込まれる固定長で右揃えの文字列に変換する例を示します。

```
CREATE TABLE t1 (
   key_1         UNSIGNED BIGINT NOT NULL,
   non_key_1     NUMERIC ( 11, 2 ) NOT NULL,
   PRIMARY KEY ( key_1 ) );

INSERT t1 VALUES ( 1, 12345.78 );
INSERT t1 VALUES ( 2, 0.00 );
INSERT t1 VALUES ( 3, 12.34 );

SELECT RIGHT ( STRING ( REPEAT ( '0', 10 ), non_key_1 ), 10 ) AS a
  FROM t1
 ORDER BY key_1;
```

これは次のように動作します。まず、REPEAT 関数によって 10 個の 0 から成る文字列が作成されます。次に、STRING が 12345.78 などの non_key_1 値を文字列 '12345.78' に変換し、10 個の 0 から成る文字列に追加して、'0000000000012345.78' を作成します。最後に、RIGHT 関数が右端の 10 個の文字 '0012345.78' を返します。結果セット全体は次のようになります。

```
a
============
'0012345.78'
'0000000.00'
'0000012.34'
```

次に、サーバ・コンソール・ウィンドウに 'USER DBA at 2003-10-19 18:01:52.151' を表示する別の例を示します。

```
MESSAGE STRING (
   'USER ',
   CURRENT USER,
   ' at ',
   CURRENT TIMESTAMP ) TO CONSOLE;
```

Section

3.12 ブール式とWHERE句

WHERE句の構文はON条件の構文と同じです。両方のキーワードとも、後に<boolean_expression>が続きます。ブール式は、他のロケーションでも使用できます。IF式、IF文、CASE式、CASE文、HAVING句、WHEN句、WHILE文、CHECK制約はすべて、ブール式を使用します。しかし、実際には、WHERE句の方が、他の句や文より複雑な式を使用する傾向があります。そのため、別の場所ではなく、ここで、<boolean_expression>の完全な構文について説明します。

```
<where_clause>       ::= WHERE <boolean_expression> -- TRUE対FALSE/UNKNOWN
<boolean_expression> ::= <predicate>                                -- 優先度:
                       | <boolean_expression> IS <truth_value>          -- 1
                       | <boolean_expression> IS NOT <truth_value>      -- 1
                       | NOT <boolean_expression>                       -- 1
                       | <boolean_expression> AND <boolean_expression>  -- 2
                       | <boolean_expression> OR <boolean_expression>   -- 3
                       | "(" <boolean_expression> ")"
                       | "(" <boolean_expression> "," <user_estimate> ")"
<predicate>          ::= <comparison_predicate>
                       | <exists_predicate>
                       | <in_predicate>
                       | <between_predicate>
                       | <null_test_predicate>
                       | <like_predicate>
                       | <trigger_predicate>
<truth_value>        ::= TRUE
                       | FALSE
                       | UNKNOWN
<user_estimate>      ::= 0.0～100.0の範囲の数値リテラル
```

WHERE句は、FROM句と単純なスカラであるselectリスト項目が評価されて候補の結果セットが作成されてから、selectの論理的実行の3番目のステップとして適用されます。各ローに対して<boolean_expression>が1回評価され、TRUE、FALSE、またはUNKNOWNを返します。ブール式がTRUEを返す場合、そのローはそのままです。FALSEまたはUNKNOWNを返す場合、そのローは結果セットから削除されます。

TRUE、FALSE、UNKNOWNは、すべてのリレーショナル・データベース・システムで使用される特殊な3値的論理システムにおけるブール値です。NULLが含まれる比較の結果は、ほとんどの場合実際に不明であるため、3番目の値のUNKNOWNが必要です。たとえば、XにNULLが格納されている場合、X = 0とX <> 0の両方の比較ともTRUEではありません。FALSEでもなく、両方ともUNKNOWNが返されます。

Note

SQL Anywhere のオンラインヘルプには、「ブール式」はありません。代わりに「探索条件」を参照してください。「探索条件」という用語には反復的な動作を示す意味合いがあり、単純な IF 文よりは WHERE 句によく当てはまります。そのため、本書では代わりに「ブール式」という表現を使用しています。

ブール式の構成は、次のとおりです。

- 1つ以上の述部
- ブール演算子（AND、OR、NOT）
- 真偽値テスト（IS、IS NOT）
- 実行の順序を制御するカッコ
- クエリ・オプティマイザに影響するユーザ推定

述部はブール式の基本構成要素で、結果として TRUE、FALSE、UNKNOWN を生じる最も単純な式です。述部には7種類あり、以降の7つの項でそれぞれについて説明します。この項では、"X=Y" 形式の単純な比較述部を使用して、複数の述部から複雑なブール式を構築する方法を示します。次に、いくつかの単純な述部と、異なる X の値によって得られる真偽値の表を示します。

```
                                  Xの値
                          ====================
述部の例                   NULL      0      1
====================     =======  =====  =====
X = 0                     UNKNOWN  TRUE   FALSE
X = 1                     UNKNOWN  FALSE  TRUE
COALESCE ( X, 0 ) = 0     TRUE     TRUE   FALSE
COALESCE ( X, 0 ) = 1     FALSE    FALSE  TRUE
X <> 0                    UNKNOWN  FALSE  TRUE
X <> 1                    UNKNOWN  TRUE   FALSE
```

この表の先頭行は、次のように解釈します。まず、X に NULL が格納されている場合、述部「X = 0」は UNKNOW を返します。X に 0 が格納されている場合、述部は TRUE を返します。X に 1 が格納されている場合、述部は FALSE を返します。

Note

TRUE、FALSE、UNKNOWNは、ブール値または真偽値を表す実際のSQL Anywhere 9キーワードです。しかし残念ながら、SQL Anywhere 9には明示的なBOOLEANデータ型やTRUTHデータ型はありません。変数やカラムをBOOLEANとして宣言できないため、<boolean_expression>の値を変数やカラムに直接格納することはできず、WHERE句、ON条件などで計算して使用するしかありません。あるいは、他のデータ型のBITを使用して1または0を保持したり、VARCHAR（1）を使用して'Y'または'N'を保持したうえで、その型の値を間接的に計算、格納、使用するコードを記述します。この目的には、IF文とSET文を使用できます。IF文とSET文については、「第8章　パッケージ」を参照してください。

IS演算子とIS NOT演算子を使用すると、述部またはブール式の結果をテストできます。これらの演算子には、常にTRUEまたはFALSEを返すという利点があります。UNKNOWNを返すことはありません。次に、IS演算子とIS NOT演算子の結果の例を示します。

```
                                  Xの値
                          ====================
ブール式                   NULL     0      1
========================  ======  =====  =====
( X = 0 ) IS TRUE         FALSE   TRUE   FALSE
( X = 0 ) IS FALSE        FALSE   FALSE  TRUE
( X = 0 ) IS UNKNOWN      TRUE    FALSE  FALSE
( X = 0 ) IS NOT TRUE     TRUE    FALSE  TRUE
( X = 0 ) IS NOT FALSE    TRUE    TRUE   FALSE
( X = 0 ) IS NOT UNKNOWN  FALSE   TRUE   TRUE
```

ほとんどの場合、UNKNOWNはFALSEと同じであるため、IS UNKNOWN演算子はほとんど使用されません。たとえば、WHERE句では、UNKNOWNまたはFALSEを生じるローが削除されます。

NOT、AND、ORの各演算子を使用すると、次の「真偽値表」に従って、TRUE、FALSE、UNKNOWNの中間結果を変更および結合できます。たとえば、<boolean_expression>がTRUEの場合、NOT <boolean_expression>はFALSEです。SQL Anywhereでは、「NOT TRUE」とそのままコーディングできないことに注意してください。次の表は、「NOT X = 1」のようなコーディングを行うと何が発生するかを手短に説明しているにすぎません。

```
     NOT          結果
===========      ======
NOT TRUE         FALSE
NOT FALSE        TRUE
NOT UNKNOWN      UNKNOWN
```

```
        AND             結果
 ==================   ======
   TRUE AND TRUE      TRUE
   TRUE AND FALSE     FALSE
   TRUE AND UNKNOWN   UNKNOWN
  FALSE AND TRUE      FALSE
  FALSE AND FALSE     FALSE
  FALSE AND UNKNOWN   FALSE
UNKNOWN AND TRUE      UNKNOWN
UNKNOWN AND FALSE     FALSE
UNKNOWN AND UNKNOWN   UNKNOWN

        OR              結果
 ==================   ======
   TRUE OR TRUE       TRUE
   TRUE OR FALSE      TRUE
   TRUE OR UNKNOWN    TRUE
  FALSE OR TRUE       TRUE
  FALSE OR FALSE      FALSE
  FALSE OR UNKNOWN    UNKNOWN
UNKNOWN OR TRUE       TRUE
UNKNOWN OR FALSE      UNKNOWN
UNKNOWN OR UNKNOWN    UNKNOWN
```

3.12.1 比較述部

ブール式を構成できる述部は7種類あります。1種類目は、"="や"<"などの論理演算子が含まれる比較です。

```
<comparison_predicate> ::= <expression>
                               <comparison_operator>
                               <expression>
                         | <expression>
                               <comparison_operator>
                               <quantifier>
                               <subquery> -- 単一カラムの結果セット
<comparison_operator>  ::= "="   -- 等価
                         | ">"   -- より大きい
                         | "<"   -- より小さい
                         | ">="  -- 以上
                         | "<="  -- 以下
                         | "<>"  -- 等価でない
                         | "!="  -- 等価でない
                         | "!<"  -- より小さくない
                         | "!>"  -- より大きくない
<quantifier>           ::= ANY   -- 1つまたは複数
                         | SOME  -- 1つまたは複数
                         | ALL   -- すべて
```

比較述部には、2つの値を指定できます。または、1つの値と修飾子 ANY、SOME、ALL のいずれかに続けて、単一カラムの結果セットを返すサブクエリを指定できます。次に、2つの値と等号演算子を使用する例を示します。

```
CREATE TABLE t1 (
   key_1         UNSIGNED BIGINT NOT NULL,
   non_key_1     INTEGER NOT NULL,
   PRIMARY KEY ( key_1 ) );

INSERT t1 VALUES ( 1, 1 );
INSERT t1 VALUES ( 2, 2 );

SELECT *
  FROM t1
 WHERE t1.non_key_1 = 2;
```

結果セットは次のようになります。

```
key_1  non_key_1
=====  =========
2      2
```

次に、2つのテーブルと ANY 演算子を伴うサブクエリを使用する例を示します。

```
CREATE TABLE t1 (
   key_1         UNSIGNED BIGINT NOT NULL,
   non_key_1     INTEGER NOT NULL,
   PRIMARY KEY ( key_1 ) );

CREATE TABLE t2 (
   key_1         UNSIGNED BIGINT NOT NULL,
   non_key_1     INTEGER NOT NULL,
   PRIMARY KEY ( key_1 ) );

INSERT t1 VALUES ( 3, 5 );
INSERT t1 VALUES ( 4, 6 );
INSERT t2 VALUES ( 11, 999 );
INSERT t2 VALUES ( 22, 50 );
COMMIT;

SELECT *
  FROM t1
 WHERE t1.non_key_1 = ANY ( SELECT t2.non_key_1 / 10
                              FROM t2 );
```

WHERE 句は、少なくとも理論上は次のように動作します。FROM t1 句で返されるローごとに単一カラムのサブクエリ（SELECT t2.non_key_1 / 10 FROM t2）が評価および検索されて、1つ以上の値が t1.non_key_1 に等しいかどうかが判断されます。その比較によって、t1 内のローの1つに対して FALSE が発生し、他のローに対しては TRUE が発生します。最終的な結果セットは次のようになります。

```
key_1  non_key_1
=====  =========
3      5
```

クエリ・オプティマイザは、実はそのように動作しません。実際には、select 全体がジョインに変換され、サブクエリが完全に削除されます。次に示すように、select を文字列リテラルに設定し、それを REWRITE 関数に渡すことによって、このことを自分で確認できます。

```
SELECT REWRITE ( '
SELECT *
  FROM t1
 WHERE t1.non_key_1 = ANY ( SELECT t2.non_key_1 / 10
                              FROM t2 )' );
```

REWRITE 関数によって、クエリ・オプティマイザが select に対して実行する操作が表示されます。この例の場合は、次のようになります。

```
SELECT DISTINCT
       t1_1.key_1,
       t1_1.non_key_1
  FROM t2 as t2_1,
       t1 as t1_1
 WHERE t1_1.non_key_1 = t2_1.non_key_1 / 10;
```

クエリ・オプティマイザによって、パフォーマンスが向上するようにクエリがリライトされます。これは、WHERE 句を自分にとって都合のいいように作成できることを意味するので、歓迎すべきことです。たとえば、ANY 修飾子を伴うサブクエリを使用したいのであれば、そのようにしてかまいません。パフォーマンスについて心配する必要はなく、サブクエリを苦労してジョインに変更する必要もありません。

3.12.2 EXISTS述部

2種類目の述部では、サブクエリにローが存在するかどうかをテストします。

```
<exists_predicate> ::= EXISTS "(" <query_expression> ")"
```

exists述部は、外部キー関係で特に役立ちます。exists述部を使用すると、子テーブル内に対応するローを持つ親テーブル・ロー、または子テーブル内に対応するローを持たない親テーブル・ローを見つけることができます。次に、子テーブルt2に含まれるデータを実際に表示せずに、子テーブルt2内に対応するローを持つ親テーブルt1内のローを表示する例を示します。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL PRIMARY KEY,
   non_key_1     INTEGER NOT NULL );

CREATE TABLE t2 (
   key_1         INTEGER NOT NULL REFERENCES t1 ( key_1 ),
   key_2         INTEGER NOT NULL,
   non_key_1     INTEGER NOT NULL,
   PRIMARY KEY ( key_1, key_2 ) );

INSERT t1 VALUES ( 5, 5 );
INSERT t1 VALUES ( 6, 6 );
INSERT t1 VALUES ( 7, 7 );
INSERT t2 VALUES ( 6, 44, 77 );
INSERT t2 VALUES ( 6, 55, 88 );

SELECT *
  FROM t1
 WHERE EXISTS ( SELECT *
                  FROM t2
                 WHERE t2.key_1 = t1.key_1 );
```

この種のサブクエリは、「相関サブクエリ」または「相関subselect」と呼ばれることがあります。サブクエリ内に外部クエリに存在するテーブルへの参照が含まれており、こうした外部参照が「相関」と呼ばれるためです。論理上、これは次のように動作します。t1内のそれぞれのローごとにサブクエリが評価され、t2.key_1 = t1.key_1であるローを返すかどうかがチェックされます。サブクエリがローを返さない場合、EXISTS述部はFALSEを返し、t1のローは削除されます。それ以外の場合、EXISTSはTRUEを返し、t1のローはそのまま残されます。このプロセスによってt1内の2つのローが削除され、1つが残って、次の最終結果セットが生成されます。

```
key_1  non_key_1
=====  =========
6      6
```

NOT EXISTS述部には、反対の効果があります。

```
SELECT *
  FROM t1
 WHERE NOT EXISTS ( SELECT *
                      FROM t2
                     WHERE t2.key_1 = t1.key_1 );
```

この場合は、子を持たないt1のローだけが返されます。

```
key_1  non_key_1
=====  =========
5      5
7      7
```

EXISTSとNOT EXISTSを使用する場合、サブクエリのselectリストは重要ではありません。クエリ・エンジンが実行する必要があるのは、サブクエリの結果にローがあるかどうかを判断することだけであり、実際にどのようなローがあるかを判断することではありません。つまり、SELECT *でまったく問題はなく、SELECT 1のようにコーディングする意味はありません。

3.12.3 IN述部

3種類目の述部では、リストまたは単一カラムのサブクエリに値が存在するかどうかをテストします。

```
<in_predicate> ::= <expression>
                       [ NOT ] IN
                       "(" <expression> "," <expression> { "," <expression> } ")"
                   | <expression>
                       [ NOT ] IN
                       "(" <basic_expression> ")"
                   | <expression>
                       [ NOT ] IN
                       <subquery> -- 単一カラムの結果セット
```

次に、INとリストを使用して、ASADEMOデータベース内で名前がJohn、Paul、George、またはRingoであるすべての従業員を表示する例を示します。

```
SELECT emp_id,
       emp_fname,
       emp_lname
  FROM employee
 WHERE emp_fname IN ( 'John', 'Paul', 'George', 'Ringo' )
 ORDER BY emp_id;
```

最終結果セットは次のようになります。3人のJohnと1人のPaulがいますが、GeorgeとRingoはいません。

```
emp_id  emp_fname  emp_lname
======  =========  =========
318     John       Crow
862     John       Sheffield
1021    Paul       Sterling
1483    John       Letiecq
```

次に、もう1つ別の例を示します。この例では、2つのIN述部でサブクエリを使用し、名前が別の人の姓に等しい従業員、またはその逆の従業員をすべて見つけます。

```
SELECT emp_id,
       emp_fname,
       emp_lname
  FROM employee
 WHERE emp_fname IN ( SELECT emp_lname
                        FROM employee )
    OR emp_lname IN ( SELECT emp_fname
                        FROM employee )
 ORDER BY emp_id;
```

最終結果セットは2つのローで構成され、"Scott"のみがemployeeテーブル内の名前と姓の両方に出現することがわかります。

```
emp_id  emp_fname  emp_lname
======  =========  =========
501     David      Scott
1576    Scott      Evans
```

3.12.4 BETWEEN述部

4種類目の述部では、ある1つの値が他の2つの値の範囲にあるかどうかをテストします。テストにはすべての範囲が含まれるため、テスト対象の値が両端のいずれかに一致する場合はTRUEが返されます。

```
<between_predicate> ::= <expression>
                          [ NOT ] BETWEEN
                          <expression> AND <expression>
```

BETWEEN述部は、日付範囲に対して特に効果的です。次に、order_dateが2000-01-03 ～ 2000-01-07の範囲に含まれるすべてのsales_orderローを返す例を示します。

```
SELECT id,
       cust_id,
       order_date
  FROM sales_order
 WHERE order_date BETWEEN '2000-01-03' and '2000-01-07'
 ORDER BY order_date;
```

結果は次のようになり、BETWEEN 述部に両端の値が含まれることがわかります。

```
id    cust_id  order_date
====  =======  ==========
2065  164      2000-01-03
2126  136      2000-01-03
2127  142      2000-01-06
2135  205      2000-01-06
2129  166      2000-01-07
```

範囲クエリのパフォーマンスは、比較が行われるカラムのクラスタード・インデックスによって向上することがあります。クラスタード・インデックスの詳細については、「10.7 インデックスの作成」を参照してください。

3.12.5 NULLテスト述部

5種類目の述部では、値にNULLが含まれるかどうかをテストします。

```
<null_test_predicate> ::= <expression> IS [ NOT ] NULL
```

COALESCEを使用してNULL値を非NULL値に変更しない場合は、NULLテスト述部を使用して問題を回避しなければならないかもしれません。次に、termination_dateがNULLのローをカウントすることによって、現役のすべての従業員をカウントする例を示します。

```
SELECT COUNT(*) AS active
  FROM employee
 WHERE termination_date IS NULL;
```

最終結果では、employeeテーブル内のすべてのローはtermination_date値がNULLであるため、ASADEMOデータベースに離職者はいないことがわかります。

```
active
======
75
```

3.12.6 LIKE述部

6種類目の述部では、文字列値がパターンに一致するかどうかをテストします。

```
<like_predicate> ::= <expression>
                        [ NOT ] LIKE
                        <expression>
                        [ ESCAPE <expression> ]
```

LIKE述部内の最初の<expression>は、チェック対象の文字列です。これは、通常、単純なカラム参照です。2番目の<expression>は、通常、目的のテキストと表3-2の特殊文字が含まれる文字列リテラルとしてコーディングされる単純なパターンです。

表3-2　LIKE述部内の特殊文字

文字	説明
%	文字数が0以上の文字列に一致する
_	任意の1文字に一致する
[ ]	カッコで囲まれたセット内の1文字に一致する
[^]	カッコで囲まれたセット以外の1文字に一致する
-	セット内の文字の範囲を指定する

次に、LIKE述部の例と、それに対してTRUEまたはFALSEとなる値を示します。デフォルトでは、文字列比較で大文字と小文字が区別されないことに注意してください。

```
                    TRUEと評価される FALSEと評価される
LIKE述部            Xの値            Xの値
==================  ===============  ==============
X LIKE '%FRED%'     'Alfred Smith'   'Harry Potter'
X LIKE '%FRED'      'Smith, Alfred'  'Alfred Smith'
X LIKE '_a_'        'cat'            'aaaaa'
X LIKE '__[xyz]_'   'CAYA'           'YANK'
X LIKE '[A-Z][0-9]' 'a2'             '4F'
X LIKE '[^r]%'      'Alfred'         'Robert'
```

"%"特殊文字は、非常によく使用されます。次に、住所に'west'という文字列が含まれるすべての従業員を検索する例を示します。

```
SELECT emp_id,
       street
  FROM employee
 WHERE street LIKE '%west%';
```

ASADEMO データベースを使用した場合の結果は次のようになります。

```
emp_id  street
======  ======================
409     190 Westmoreland Street
930     251 Westminster Street
1013    589 West Drive
```

次に、姓が"D"で始まるすべての従業員の電話番号を表示する例を示します。

```
SELECT STRING ( emp_lname, ', ', emp_fname ) AS full_name,
       STRING ( '(', LEFT ( phone, 3 ), ') ',
                SUBSTR ( phone, 4, 3 ), '-',
                RIGHT ( phone, 4 ) ) AS phone
  FROM employee
 WHERE emp_lname LIKE 'd%'
 ORDER BY emp_lname,
       emp_fname;
```

結果は次のようになります。STRING と他の関数を使用して出力がフォーマットされていることに注意してください。

```
full_name         phone
================  ==============
Davidson, Jo Ann  (617) 555-3870
Diaz, Emilio      (617) 555-3567
Dill, Marc        (617) 555-2144
Driscoll, Kurt    (617) 555-1234
```

ESCAPE 式を使用すると、パターンの中でエスケープ文字として解釈される 1 つの文字を指定できます。エスケープ文字の後に特殊文字（パーセント記号やアンダースコアなど）を指定すると、その文字は通常のデータ文字として扱われます。たとえば、LIKE '[a-z]%' は、通常、0 個以上の文字が後に続く 1 文字の英字に一致します。実際のパーセント記号が後に続く 1 文字の英字に一致させるには、エスケープ文字として'?' を使用して、LIKE「'[a-z]?%' ESCAPE '?'」とコーディングします。次の述部は、ESCAPE による相違を示しています。

```
predicate                       result
==============================  =======
'x%' LIKE '[a-z]%'              TRUE    -- %は任意の文字に一致する
'x%' LIKE '[a-z]?%' ESCAPE '?'  TRUE    -- ?%は%に一致する
'x'  LIKE '[a-z]%'              TRUE    -- %は任意の文字に一致する
'x'  LIKE '[a-z]?%' ESCAPE '?'  FALSE   -- ?%は%を期待している
```

3.12.7 トリガ述部

最後の7種類目の述部は、トリガの中でのみ使用し、トリガの起動を引き起こした操作の種類を判断できます。

```
<trigger_predicate>   ::= INSERTING
                        | DELETING
                        | UPDATING [ "(" <column_name_literal> ")" ]
                        | UPDATE "(" <column_name> ")"
<column_name_literal> ::= <column_name>を含む文字列リテラル
<column_name>         ::= <identifier>
```

次に、これらの各トリガ述部の例を示します。これらのトリガ述部は、WHERE句ではなく、トリガ内のCASE文で使用しています。ここでは、MESSAGE文を使用して、診断メッセージをデータベース・コンソールに表示します。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL PRIMARY KEY,
   non_key_1     INTEGER NOT NULL,
   non_key_2     INTEGER NOT NULL,
   non_key_3     INTEGER NOT NULL );

CREATE TRIGGER triud_t1
   BEFORE INSERT, DELETE, UPDATE ON t1
   REFERENCING OLD AS old_t1 NEW AS new_t1
   FOR EACH ROW
BEGIN
   DECLARE @message VARCHAR ( 1000 );
   SET @message =
      CASE
         WHEN INSERTING                THEN 'Inserting t1'
         WHEN DELETING                 THEN 'Deleting t1'
         WHEN UPDATING ( 'non_key_1' ) THEN 'Updating t1.non_key_1'
         WHEN UPDATE ( non_key_2 )     THEN 'Updating t1.non_key_2'
         WHEN UPDATING                 THEN 'Updating other t1 column'
      END;
   MESSAGE @message TO CONSOLE;
END;
```

```
INSERT t1 VALUES ( 1, 1, 1, 1 );
UPDATE t1 SET non_key_1 = 2 WHERE key_1 = 1;
UPDATE t1 SET non_key_2 = 3 WHERE key_1 = 1;
UPDATE t1 SET non_key_3 = 4 WHERE key_1 = 1;
DELETE t1 WHERE key_1 = 1;
```

上のINSERT文、UPDATE文、DELETE文を実行すると、データベース・エンジン・ウィンドウには次のメッセージが表示されます。

```
Inserting t1
Updating t1.non_key_1
Updating t1.non_key_2
Updating other t1 column
Deleting t1
```

CASE文とCREATE TRIGGER文の詳細については、「第8章　パッケージ」を参照してください。

Section

3.13 GROUP BY句

GROUP BY句には、ROLLUPキーワードの有無に応じて2種類の形式があります。第1の形式は、GROUP BYキーワードと1つ以上の式のリストで構成されます。最も一般的な式は、カラム名への単純な参照です。第2の形式は、GROUP BY ROLLUPで始まり、同じように式のリストが続きます。ただし、この形式の場合は、リストをカッコで囲む必要があります。

```
<group_by_clause> ::= GROUP BY <group_by_list>
                    | GROUP BY ROLLUP "(" <group_by_list> ")"
<group_by_list>   ::= <expression> { "," <expression> }
```

GROUP BYもGROUP BY ROLLUPも、最初はまったく同じグループ化プロセスを実行します。ここでは、この最初のグループ化プロセスについて説明するため、これ以降は単にGROUP BYと呼びます。GROUP BY ROLLUP句は、2番目のステップも実行して、候補となる結果セットにロールアップ・ローを追加します。そのロールアップ・ステップについては3.15項で別に説明します。

GROUP BY句は、selectの論理的実行の4番目のステップとして、WHERE句が終了して候補結果セットから不要なローが削除されてから適用されます。GROUP BY句の背後にある一番の目的は、集合関数呼び出しを使用できるようにすることであり、SUMやCOUNTなどの集合関数を使用しない場合には、GROUP BYとDISTINCTを区別するのは困難です。たとえば、次のselectはまったく同じ結果を生成

します。

```
SELECT dept_id
  FROM employee
 GROUP BY dept_id
 ORDER BY dept_id;

SELECT DISTINCT dept_id
  FROM employee
 ORDER BY dept_id;
```

ASADEMO データベースに対して両方の select を実行すると、結果セットは次のようになります。

```
dept_id
=======
100
200
300
400
500
```

理論上、GROUP BY 句は、グループ化式の値に従ってローをグループに分けます。値の組み合わせが同じローはすべて、同じグループに配置されます。次に、select リスト内の集合関数呼び出しがグループごとに 1 回計算されます。この処理が実行されると、各グループは単一ローになります。

Note

グループ化式の値を比較する場合、NULL 値は同じであると見なされます。これは、NULL 値の通常の扱いとは異なります。NULL 値に関する比較の場合、通常、結果は UNKNOWN です。

GROUP BY 句は、select リスト項目と、select リストにない仮想カラムを参照する可能性があります。次に、ニューヨークの従業員をマネージャと都市でグループ化し、それぞれの数を表示する例を示します。

```
SELECT manager_id,
       COUNT(*)
  FROM employee
 WHERE state = 'NY'
 GROUP BY manager_id,
       city
 ORDER BY manager_id;
```

結果は次のようになります。select リストにないカラムでグループ化できることに注意してください。

```
manager_id  COUNT(*)
==========  ========
501         2
501         1
902         1
1293        1
1576        3
```

ほとんどの場合、グループ化式を実際に表示する方が理にかなっています。次に、selectリストにcityを追加した同じ例を示します。

```
SELECT manager_id,
       city,
       COUNT(*)
  FROM employee
 WHERE state = 'NY'
 GROUP BY manager_id,
       city
 ORDER BY manager_id;
```

これで、「manager_id = 501」のローが2つある理由がわかります。

```
manager_id  city        COUNT(*)
==========  ==========  ========
501         Cornwall    2
501         Fort Henry  1
902         Cornwall    1
1293        Cornwall    1
1576        Cornwall    3
```

GROUP BY句を使用する場合は、selectリストに指定できる項目に制限が課せられます。selectリスト内の式がGROUP BYリストにもない場合は、その式は、selectリスト内の集合関数呼び出しに指定する必要があります。たとえば、次のSELECTは無効であり、「'state'に対する関数またはカラムの参照もGROUP BYに記述する必要があります」というエラー・メッセージが出力されます。

```
SELECT manager_id,
       state
  FROM employee
 GROUP BY manager_id
 ORDER BY manager_id;
```

ただし、集合関数呼び出しにstateがある場合は、selectリストにstateがあっても問題ありません。次に、COUNT呼び出しにstateがあるために動作する例を示します。

```
SELECT manager_id,
       COUNT ( DISTINCT state ) AS states
   FROM employee
 GROUP BY manager_id
 ORDER BY manager_id;
```

結果は次のようになります。これは、各マネージャの下で働く従業員の自宅がある州の数を示しています。

```
manager_id  states
==========  ======
501         10
703         5
902         13
1293        7
1576        9
```

COUNT 関数については、次の項で説明します。

Section

3.14 集合関数呼び出し

集合関数呼び出しは、select リスト、HAVING 句、ORDER BY 句に指定される可能性があります。

```
<aggregate_builtin_function_call> ::= AVG "(" [ DISTINCT ] <expression> ")"
                                    | COUNT "( * )"
                                    | COUNT "(" [  DISTINCT ] <expression> ")"
                                    | GROUPING "(" <expression> ")"
                                    | LIST "(" <expression>
                                                [ "," <list_delimiter> ]
                                                [ <order_by_clause> ] ")"
                                    | MAX "(" <expression> ")"
                                    | MIN "(" <expression> ")"
                                    | STDDEV "(" <expression> ")"
                                    | STDDEV_POP "(" <expression> ")"
                                    | STDDEV_SAMP "(" <expression> ")"
                                    | SUM "(" [ DISTINCT ] <expression> ")"
                                    | VAR_POP "(" <expression> ")"
                                    | VAR_SAMP "(" <expression> ")"
                                    | VARIANCE "(" <expression> ")"
<list_delimiter> ::= 各LIST値の間に指定する文字である<expression>
```

すべての集合関数呼び出し（GROUPINGは除く）は、論理上、前の項で説明したグループ化プロセスが終了すると同時に評価される可能性があります。selectリスト内の呼び出しの結果は候補結果セットに設定されるのに対して、HAVING句内の呼び出しの結果は句が終了するまで保存されます（詳細については、「3.16　HAVING句」を参照してください）。ここでは、個々の集合関数が実行する内容について説明します。

AVG関数は、各グループ内の平均値を計算します。次に、ASADEMOデータベース内の各部門の平均給与を計算する例を示します。

```
SELECT dept_id,
       AVG ( salary ) AS average_salary
  FROM employee

 GROUP BY dept_id
 ORDER BY dept_id;
```

結果セットは次のとおりです。

```
dept_id  average_salary
=======  ==============
100      58736.281364
200      48390.947368
300      59500.000000
400      43640.671875
500      33752.200000
```

集合関数の多くでは、DISTINCTキーワードを使用できます。AVG (DISTINCT X)は、すべての異なるXの値について平均値を計算します。グループ内に同じXの値を持つローが複数ある場合、平均値の計算に関する限りその値は1回しかカウントされません。

Note

AVG (ALL X)のように、ALLキーワードも明示的に指定できますが、これはデフォルトであり、各グループ内のすべてのX値の平均値を計算するという、明白なことを指定しているだけであるため、構文には示していません。

DISTINCTは、AVGに関してはあまり意味はありませんが、違いは生じます。次に、上と同じSELECTに、AVG呼び出し内のDISTINCT、各グループ内のロー数をカウントするCOUNT(*)呼び出し、各グループ内の異なるsalary値の数をカウントするCOUNT (DISTINCT salary)呼び出しを追加した例を示します。

```
SELECT dept_id,
       AVG ( DISTINCT salary )   AS average_distinct_salary,
       COUNT (*)                 AS employee_count,
       COUNT ( DISTINCT salary ) AS distinct_salary_count
  FROM employee
 GROUP BY dept_id
 ORDER BY dept_id;
```

新しい結果セットは次のようになります。「dept_id = 100」の場合、従業員数は22人いますが、異なる給与の値は21種類だけであるため、AVG (DISTINCT salary) の値が、前のAVG (salary) とは多少異なることに注意してください。

```
dept_id  average_distinct_salary  employee_count  distinct_salary_count
=======  =======================  ==============  =====================
100      57347.532857             22              21
200      48390.947368             19              19
300      59500.000000             9               9
400      43640.671875             16              16
500      33752.200000             9               9
```

COUNT関数には3種類の便利な形式があります。COUNT(*) は、グループ内のすべてのローをカウントします。COUNT (X) は、XがNULLでないすべてのローをカウントします。COUNT (DISTINCT X) は、すべて異なるNULLでないXの値をカウントします。

COUNT(*) は、カラムや式を操作するのではなく、ローをカウントするため、結果にNULL値を含む唯一の集合関数です。他のすべての集合関数では、計算にNULL値は含まれません。たとえば、次のSELECTを実行すると、NULL、NULL、100、200の平均値は150であると表示されます。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL PRIMARY KEY,
   non_key_1     INTEGER NULL );

INSERT t1 VALUES ( 1, NULL );
INSERT t1 VALUES ( 2, NULL );
INSERT t1 VALUES ( 3, 100 );
INSERT t1 VALUES ( 4, 200 );
COMMIT;

SELECT AVG ( non_key_1 )
  FROM t1;
```

150という値が自分にとって意味のあるものならば、それで問題はありません。これは、すべての非NULL値の平均値、つまり (100 + 200) を2で割った値です。しかし、すべてのローを含む必要がある場合は、COALESCEを使用して、NULLを含むローを非NULL値で置き換えることができます。次の

クエリは、NULL を 0 として処理して、75（300 を 4 で割った値）という平均値を表示します。

```
SELECT AVG ( COALESCE ( non_key_1, 0 ) )
  FROM t1;
```

最後の 2 つの例には、GROUP BY 句がないことに注意してください。通常、GROUP BY 句がない場合は、グループ化は行われません。つまり、各ローがそれぞれ独自のグループであり、グループ化プロセスによるローの削除は行われません。言い換えると、「SELECT * FROM employee」は、employee テーブル内のすべてのローを返します。

しかし、select リスト内で 1 つ以上の集合関数参照を使用し、GROUP BY 句を使用しない場合は、すべてのローが 1 つのグループに属しているかのように扱われます。これは、GROUP BY 1 とコーディングされたのと同じ状態です。リテラル 1 はすべてのローで同じ値なので、すべてが同じグループに設定され、すべてのローではなく 1 つのローだけが求められます。集合計算を実行していることを考えれば、このように動作するのが道理にかなっています。次に示すのは、上の例に GROUP BY 1 句を指定した例です。結果は同じように 75 になります。

```
SELECT AVG ( COALESCE ( non_key_1, 0 ) )
  FROM t1
 GROUP BY 1;
```

select リストに 1 つ以上の集合関数があって GROUP BY がない場合、select リスト内のすべてのカラム参照は、集合関数内になければなりません。次の例はこのルールに違反しているため、「'emp_id' に対する関数またはカラムの参照も GROUP BY に記述する必要があります」というエラー・メッセージが出力されます。

```
SELECT emp_id,
       COUNT(*)
  FROM employee;
```

MAX 関数と MIN 関数はすべてのデータ型に対して動作し、参照されているカラムにインデックスがある場合は非常に効率的に動作します。たとえば、ASADEMO データベースの場合、customer テーブルの lname カラムには ix_cust_name という昇順インデックスがあり、ISQL のグラフィカルなプラン機能によれば、次の SELECT は「ix_cust_name を使用したインデックス・スキャン」を使用します。

```
SELECT MIN ( lname )
  FROM customer;
```

インデックスと ISQL のグラフィカルなプラン機能の詳細については、「第 10 章　チューニング」の 10.5 項を参照してください。

SUM関数は、グループごとに式のすべての値の合計を計算します。次の例では、ASADEMOデータベース内の各部門の支払い給与総額と、2%増加した場合の予想支払い給与総額を計算します。

```
SELECT dept_id,
       SUM ( salary )        AS current_payroll,
       SUM ( salary * 1.02 ) AS projected_payroll
  FROM employee
 GROUP BY dept_id
 ORDER BY dept_id;
```

結果は次のとおりです。

```
dept_id  current_payroll  projected_payroll
=======  ===============  =================
100      1292198.190      1318042.15380
200      919428.000       937816.56000
300      535500.000       546210.00000
400      698250.750       712215.76500
500      303769.800       309845.19600
```

合計はカウントと同じであることを利用して、SUM関数を使用してカウントすることもできます。次に、SUMとIFを使用して、女性、男性、55才未満の従業員、55才以上の従業員の数をカウントする例を示します。

```
SELECT SUM ( IF sex = 'F'
               THEN 1
               ELSE 0
             ENDIF )    AS women,
       SUM ( IF sex = 'M'
               THEN 1
               ELSE 0
             ENDIF )    AS men,
       SUM ( IF ( DATEDIFF ( YEAR, birth_date, CURRENT DATE )
                  - IF DATEFORMAT ( birth_date, 'mmdd' )
                          > DATEFORMAT ( CURRENT DATE, 'mmdd' )
                     THEN 1
                     ELSE 0
                   ENDIF ) < 55
               THEN 1
               ELSE 0
             ENDIF )    AS under_55,
       SUM ( IF ( DATEDIFF ( YEAR, birth_date, CURRENT DATE )
                  - IF DATEFORMAT ( birth_date, 'mmdd' )
                          > DATEFORMAT ( CURRENT DATE, 'mmdd' )
                     THEN 1
```

```
                    ELSE 0
                ENDIF ) >= 55
            THEN 1
            ELSE 0
        ENDIF )    AS "55_and_over"
  FROM employee;
```

ASADEMO データベースを使用した場合の結果は次のようになります。

```
women  men  under_55  55_and_over
=====  ===  ========  ===========
34     41   63        12
```

Tip

SUM でカウントするこの方法を使用すると、サブクエリと、アプリケーション・プログラム・コードさえも削除できます。これは次のように操作します。カウントする場合は 1 を返し、カウントしない場合は 0 を返す式を記述し、その式を SUM 呼び出しの内部に指定します。

AVG に加えて、表 3-3 に示すように、統計計算用のさまざまな集合関数が用意されています。どの関数も DOUBLE 値を返します。

表 3-3　他の統計集合関数【監注 3】

名前	説明
STDDEV_POP	母集団の標準偏差
STDDEV_SAMP	標本の標準偏差
STDDEV	STDDEV_SAMP と同じ
VAR_POP	母集団の平方偏差
VAR_SAMP	サンプルの平方偏差

LIST 集合関数は、1 つの単純な値を返さず、複数の値から成るフォーマットされたリストを含む文字列を返す点で、他の関数とは異なります。LIST の最も単純な形式では、グループ内のすべてのローの式を評価し、その値を文字列に変換し、各値をカンマ（,）で区切って結合して 1 つの長い文字列を生成します。次に、「dept_id = 100」のすべての employee.state 値から成るリストを生成する例を示します。

```
SELECT LIST ( state ) AS states
  FROM employee
 WHERE dept_id = 100;
```

【監注 3】SQL Anywhere Studio 9.0.2 では、この他に、VARIANCE（VAR_SMAP と同じ）や共分散のためのもの（COVAR_POP、COVAR_SAMP）、回帰分析のためのもの（REGR_AVRX、REGR_AVRY、REGR_COUNT、REGR_INTERCEPT、REGR_R2、REGR_SLOPE、REGR_SXX、REGR_SYY、REGR_SXY）がある。

結果は次のようになります。リストはソートされていないため、重複があることに注意してください。

```
states
======
NY,UT,PA,UT,UT,NY,FL,CO,MI,FL,MI,WY,RI,IL,WY,UT,RI,TX,TX,TX,NY,UT
```

LIST 呼び出し内で ORDER BY 句を使用してリストをソートできます。

```
SELECT LIST ( state ORDER BY state ) AS states
  FROM employee
 WHERE dept_id = 100;

states
======
CO,FL,FL,IL,MI,MI,NY,NY,NY,PA,RI,RI,TX,TX,TX,UT,UT,UT,UT,UT,WY,WY
```

リスト式以外のものでリストを並べ替えることができます。

```
SELECT LIST ( state ORDER BY salary DESC ) AS different_states
  FROM employee
 WHERE dept_id = 100;

different_states
================
RI,TX,FL,MI,UT,UT,IL,MI,UT,FL,PA,UT,TX,CO,UT,NY,NY,RI,TX,WY,NY,WY
```

DISTINCT キーワードで重複する値を削除できます。

```
SELECT LIST ( DISTINCT state ORDER BY state ) AS different_states
  FROM employee
 WHERE dept_id = 100;

different_states
================
CO,FL,IL,MI,NY,PA,RI,TX,UT,WY
```

LIST のデリミタを変更することもできます。次の例では、カンマの後に 1 つのブランクが含まれています。

```
SELECT LIST ( DISTINCT state, ', ' ORDER BY state ) AS different_states
  FROM employee
 WHERE dept_id = 100;

different_states
```

```
================
CO, FL, IL, MI, NY, PA, RI, TX, UT, WY
```

カンマは先頭の2つのLISTパラメータを区切っており、ORDER BY句を区切っているわけではないことに注意してください。

次に、LISTデリミタを利用して、HTMLドキュメント全体が含まれる1つの文字列を生成する例を示します。

```
SELECT STRING (
          '<HTML><BODY><OL>¥x0d¥x0a',
          '   <LI>',
          LIST ( DISTINCT state,
                 '</LI>¥x0d¥x0a   <LI>'
                 ORDER BY state ),
          '</LI>¥x0d¥x0a',
          '</OL></BODY></HTML>' ) AS states_page
  FROM employee
 WHERE dept_id = 100;
```

文字列は次のようになります。文字列には、¥x0d¥x0aという復帰（carriage return）と改行（linefeed）のペアが埋め込まれていることに注意してください。

```
<HTML><BODY><OL>
   <LI>CO</LI>
   <LI>FL</LI>
   <LI>IL</LI>
   <LI>MI</LI>
   <LI>NY</LI>
   <LI>PA</LI>
   <LI>RI</LI>
   <LI>TX</LI>
   <LI>UT</LI>
   <LI>WY</LI>
</OL></BODY></HTML>
```

Tip

LIST項目の順序が重要な場合は、DISTINCTを使用する場合でも、常にORDER BY句を使用してください。LIST項目の確実で固定的な順序はありません。将来的にソートを使ったDISTINCTが実装される可能性もありますが、そうでない可能性もあります。

候補結果セットにローがない場合は、selectリスト内の集合関数呼び出しによって、最終結果セットに1つのローが含まれます。次の例では、LIST関数は空の文字列を返し、COUNT(*)は0を返し、MINなどの他の関数はNULL値を返します。

```
SELECT LIST ( dummy_col ), MIN ( dummy_col ), COUNT(*)
  FROM dummy
 WHERE dummy_col <> 0;
```

GROUPING 集合関数については、次の項で説明します。

Section

3.15 GROUP BY ROLLUP句

GROUP BY ROLLUP 句は、基本の GROUP BY プロセスが終了した後、候補結果セットに1つ以上のローを追加します。

```
<group_by_clause> ::= GROUP BY <group_by_list>
                    | GROUP BY ROLLUP "(" <group_by_list> ")"
<group_by_list>   ::= <expression> { "," <expression> }
```

ROLLUP キーワードを使用すると、グループ化式ごとに1レベルの計算ローが挿入されます。1番左のグループ化式は1つの総計ローを生成し、2番目のグループ化式は1つ以上の小計ローを生成します。3番目のグループ化式はサブ小計ローを生成し、以降についても同様です。

ROLLUP 計算ローは次のようになります。select リスト内の1つ以上のグループ化式が NULL に置換されます。NULL 値の個数は小計のレベルによって決まります。たとえば、総計 ROLLUP ローの場合は、select リスト内のすべてのグループ化式が NULL に置換されます。一方、最下位の各小計ローの場合、NULL の個数は1つだけです。COUNT、SUM、LIST などの集合関数を使用している select リスト内の他のカラムは、グループ内のすべての基本ローに対して再計算されます。

次に、ASADEMO データベースを使用して、オハイオの各顧客に対する 2000 年3月、4月、5月の営業部員ごとの販売件数を計算する例を示します。

```
SELECT customer.id,
       sales_order.sales_rep,
       COUNT ( sales_order.sales_rep ) AS sales
  FROM ( SELECT *
           FROM customer
          WHERE customer.state IN ( 'OH' ) ) AS customer
       INNER JOIN
       ( SELECT *
           FROM sales_order
          WHERE sales_order.order_date
          BETWEEN '2000-03-01' AND '2000-05-31'  ) AS sales_order
       ON sales_order.cust_id = customer.id
```

```
GROUP BY ROLLUP ( customer.id,
       sales_order.sales_rep )
ORDER BY COALESCE ( customer.id, 99999999 ),
      COALESCE ( sales_order.sales_rep, 99999999 );
```

ROLLUPキーワードを使用しない場合は、GROUP BY句によって、顧客と営業部員の5種類の組み合わせに対応する5つのローが生成されます。ROOLUPを指定すると、結果セットにさらに5つのローが追加されます。その内訳は、4人の顧客ごとに、その顧客に対して売り上げを上げたすべての営業部員の小計を示す1つの小計ローと、すべての顧客の合計を示す1つの総計ローです。小計ローではsales_repカラムはNULLで、総計ローではcustomer.idとsales_repがNULLです。

```
id    sales_rep  sales
===   =========  =====
110   667        1
110   NULL       1      -- 顧客 110 の小計
117   467        1
117   NULL       1      -- 顧客 117 の小計
134   299        1
134   856        1
134   NULL       2      -- 顧客 134 の小計
153   1142       1
153   NULL       1      -- 顧客 153 の小計
NULL  NULL       5      -- 総計
```

Note

ROLLUPは、SUMやCOUNTに加えて、MINやLISTなどの集合関数でも動作するため、「小計」という用語は必ずしも正確ではありません。また、合計の計算は、下位レベルの計算ローではなく、基本ローに基づくという事実は、LISTやCOUNTなど、ほとんどの集合関数にとっては違いがありませんが、AVGやVARIANCEなどの統計関数には影響があります。

INNER JOIN演算が含まれるほとんどのGROUP BY ROLLUPクエリでは、小計ローおよび総計ローと、基本のGROUP BYプロセスによって生じるローを区別するのは簡単です。しかし、OUTER JOIN演算はグループ化カラム内にNULL値を生成することがあるため、GROUP BYローと、ROLLUP演算によって追加されたローを区別するのは困難です。たとえば、上のSELECT内のINNER JOINをLEFT OUTER JOINに変更すると、何も購入していなくてもすべての顧客が含まれるため、結果セットが紛らわしくなります。

```
id    sales_rep  sales
====  =========  =====
110   667        1
110   NULL       1
```

```
117   467        1
117   NULL       1
134   299        1
134   856        1
134   NULL       2
153   1142       1
153   NULL       1
155   NULL       0     -- これは小計ロー？
155   NULL       0     -- ...それともこれが小計ロー？
162   NULL       0
162   NULL       0
167   NULL       0
167   NULL       0
NULL  NULL       5
```

GROUPING集合関数をselectリストの中で使用すると、ローがROLLUP演算の結果であるかどうかを判断できます。また、ローがROLLUP演算の結果である場合は、それが、総計ローであるか、または各種レベルの小計ローの1つであるかを判断できます。たとえば､SELECTにGROUP BY ROLLUP (A, B, C)が指定されている場合、GROUPING (A)は総計ローに対して1を返し、他のすべてに対して0を返します。GROUPING (B)は総計ローと小計ローに対して1を返し、他のすべてに対して0を返します。GROUPING (C)は総計ロー、小計ロー、サブ小計ローに対して1を返し、他のすべてに対して0を返します。

次に、前のSELECTで、ROLLUPローにNULLではなくフォーマットされたテキストを生成するGROUPING呼び出しを使用する例を示します。

```
SELECT IF GROUPING ( customer.id ) = 1
          THEN 'Grand Total'
          ELSE STRING ( customer.id )
       ENDIF                                  AS customer_id,
       CASE
          WHEN GROUPING ( customer.id ) = 1
             THEN ''
          WHEN GROUPING ( sales_order.sales_rep ) = 1
             THEN 'Subtotal'
          ELSE STRING ( '', sales_order.sales_rep )
       END                                    AS sales_rep,
       COUNT ( sales_order.sales_rep )        AS sales,
       LIST ( sales_order.id, ', ' )          AS order_ids
  FROM ( SELECT *
           FROM customer
          WHERE customer.state IN ( 'OH' ) ) AS customer
       LEFT OUTER JOIN
       ( SELECT *
           FROM sales_order
          WHERE sales_order.order_date
          BETWEEN '2000-03-01' AND '2000-05-31'  ) AS sales_order
```

```
       ON sales_order.cust_id = customer.id
 GROUP BY ROLLUP ( customer.id,
       sales_order.sales_rep )
ORDER BY customer_id,
      sales_rep;
```

結果は次のようになり、GROUPING呼び出しによって紛らわしさが解消されています。また、LIST集合関数が、基本のGROUP BYローと同様に、ROLLUP計算ローで機能していることに注意してください。

```
customer_id  sales_rep  sales  order_ids
===========  =========  =====  ======================
110          667        1      2231
110          Subtotal   1      2231
117          467        1      2238
117          Subtotal   1      2238
134          299        1      2177
134          856        1      2124
134          Subtotal   2      2124, 2177
153          1142       1      2171
153          Subtotal   1      2171
155                     0
155          Subtotal   0
162                     0
162          Subtotal   0
167                     0
167          Subtotal   0
Grand Total             5      2231, 2238, 2124, 2177, 2171
```

Tip

NULL値が含まれるローをカウントしない場合は、COUNT(*)ではなくCOUNT（ <expression> ）を使用します。たとえば、上のSELECTでは、COUNT（ sales_order.sales_rep ）と指定します。COUNT(*)を使用すると、salesカラムは、売り上げのないローに対して間違った値を表示します。COUNT(*)はNULL値を考慮せずにローをカウントするため、これらのローには0ではなく1が表示され、このエラーはROLLUP計算ローにも影響します。

Tip

COALESCEを呼び出さずに、式を文字列に変換し、同時にNULL値を削除するには、STRING（ ", <expression> ）関数呼び出しを使用します。たとえば、上のSELECTでは、STRING（ ", sales_order.sales_rep ）を使用します。この方法は、NULL文字列と非NULL文字列を"¦¦"演算子またはSTRING関数で連結すると、NULL文字列はNULLでない空の文字列として解釈されることを利用しています。NULLを伴うほとんどの操作はNULLを返しますが、文字列連結はこれに該当しません。

Section

3.16 HAVING句

HAVING 句は WHERE 句とまったく同じように動作し、候補結果セットからローを削除します。理論上の相違は、HAVING 句は GROUP BY と ROLLUP のすべての処理が終了してから適用され、その時点で存在する候補結果セットからグループ全体を削除することによって動作することです。このことは、HAVING 句の中のブール式には、集合関数への参照が含まれる可能性がある（通常は含まれる）ことを意味しています。一方、WHERE 句には集合関数への参照は含まれません。

```
<having_clause> ::= HAVING <boolean_expression>
```

HAVING 句には、グループ化式への参照、つまり、GROUP BY 句に指定される式への参照が含まれる可能性もあります。次に、カウントが 10 以上で customer id が 200 以上という条件で、顧客によるすべての販売をカウントする SELECT の例を示します。HAVING 句内の述部「sales >= 10」は、実際には、エイリアス名 salse を通じての集合関数呼び出し COUNT(*) の結果への参照であることに注意してください。

```
SELECT customer.id  AS customer_id,
       COUNT(*)     AS sales
  FROM customer
       INNER JOIN sales_order
               ON sales_order.cust_id = customer.id
 GROUP BY customer_id
HAVING customer_id >= 200
   AND sales >= 10
 ORDER BY customer_id;
```

ASADEMO データベースを使用した結果は次のようになります。この select リストでは、COUNT(*) は問題ないことに注意してください。INNER JOIN では NULL 値のカウントを考慮する必要がないためです。

```
customer_id  sales
===========  =====
201          11
204          11
209          10
```

計算ロー内の NULL 値はブール式で問題を引き起こすことがあるため、GROUP BY ROLLUP を使用する場合は、HAVING 句の指定の仕方に注意してください。たとえば、上の SELECT を変更して

GROUP BY ROLLUP (customer_id) を使用すると、総計の計算ローは表示されず、結果セットは上の場合とまったく同じになります。これは、総計 ROLLUP ローでは customer_id が NULL であるため、述部「customer_id >= 200」が UNKNOWN を生じ、次にこれが原因で HAVING 句が UNKNOWN を生じ、TRUE を生じるローだけが残るためです。

この問題を解決する方法の 1 つは COALESCE を呼び出すことであり、もう 1 つの方法は「customer_id >= 200」を WHERE 句内に移動して、GROUP BY 句と HAVING 句の前に処理されるようにすることです。

```
SELECT customer.id  AS customer_id,
       COUNT(*)     AS sales
  FROM customer
       INNER JOIN sales_order
               ON sales_order.cust_id = customer.id
 WHERE customer_id >= 200
 GROUP BY ROLLUP ( customer_id )
HAVING sales >= 10
 ORDER BY customer_id;
```

ROLLUP 計算ローが残り、最終結果セットに含まれます。

```
customer_id  sales
===========  =====
NULL         87
201          11
204          11
209          10
```

Section

3.17 ORDER BY句

ORDER BY 句は、GROUP BY 句と HAVING 句が処理された後で適用され、候補結果グループを 1 つ以上の明示的な式または番号で指定された select リスト項目でソートします。キーワード ASC と DESC は昇順と降順をそれぞれ表し、ASC がデフォルトです。

```
<order_by_clause>    ::= ORDER BY <order_by_item> { "," <order_by_item> }
<order_by_item>      ::= <select_item_number> [ ASC | DESC ]
                       | <expression> [ ASC | DESC ]
<select_item_number> ::= 1 ～ select リストの項目数の範囲を表す整数リテラル
```

selectリスト項目の番号は、selectリスト項目の順番を表します。たとえば、「ORDER BY 1, 2, 3」はselectリスト項目の最初の3つを表します。この番号付けは、「*」がselect項目の明示的なリストとしてコーディングされているかのように、「SELECT *」でも有効です。たとえば、次のSELECTは、ASADEMOデータベース内のemployeeテーブルの4番目と3番目のカラムであるemp_lnameとemp_fnameで結果セットをソートします。

```
SELECT *
  FROM employee
 ORDER BY 4, 3;
```

<expression>の構文では、これが整数リテラルでも許可されていますが、ORDER BYの場合、整数リテラルは評価され、ソートに使用される式としては解釈されず、selectリスト項目を検索するための序数として解釈されます。固定の式によるソートは順序に影響しないため、これによって利便性が損なわれることはありません。

明示的なORDER BY式は、selectリスト項目と同じにすることができ、またしばしばそのようにしますが、必ずしもそうである必要はありません。たとえば、selectリストにないカラムで結果セットをソートできます。また、selectのどこにも指定されていない集合関数参照をORDER BYに指定することもできます。

ただし、制限もあります。たとえば、カラムXをGROUP BYに指定する場合は、カラムYをORDER BYに指定することはできません。ORDER BYとGROUP BYを一緒に使用すると、ORDER BYは実際にグループを並べ替えようとするわけですが、各グループには異なるYの値が複数含まれるので、Yでソートすることができないからです。

おおざっぱに言うと、selectリストに指定できない項目は、ORDER BYには指定できません。GROUP BYにカラムXを指定すると、selectリストにYを指定できません。したがって、ORDER BY Yは実行できません。ただし、selectリストにSUM (Y)は指定できるため、ORDER BYにもSUM (Y)を指定できます。

次に、結果セットに含まれない式でソートし、結果セットを生成するORDER BYの例を示します。

```
SELECT sales_order.order_date,
       COUNT(*) AS sales
  FROM sales_order
       INNER JOIN sales_order_items
               ON sales_order_items.id = sales_order.id
 WHERE sales_order.order_date BETWEEN '2000-04-01' AND '2000-11-30'
 GROUP BY sales_order.order_date
HAVING COUNT(*) >= 5
 ORDER BY SUM ( sales_order_items.quantity ) DESC;
```

最終結果セットはソートされているように見えませんが、実際にはソートされています。先頭ローは、販売された製品の数量が最も多い注文日を示しています。2番目のローは2番目に数量が多い日を示し

ており、以降についても同様です。数量は表示されず、注文日と注文数のみが表示されているため、ソート順が明らかではありません。

```
order_date  sales
==========  =====
2000-05-29  5      -- SUM ( sales_order_items.quantity ) の最大値
2000-10-30  5
2000-04-02  7
2000-11-25  6
2000-11-19  5      -- SUM ( sales_order_items.quantity ) の最小値
```

Tip

結果セットに表示されないカラムでのソートは、それほど意味がないわけではありません。たとえば、FIRST キーワードを使用して ORDER BY リストの先頭のローを取得することが可能であり、それだけが必要な場合もあります。

論理上は、ORDER BY 句が処理されると、グループ内に複数のローや余分なカラムを保持する必要はもはやなく、各グループを、select リスト項目のみから成る単一のローにすることができます。ORDER BY がない場合でも、この時点でグループはローになります。たとえば、UNION の一部である SELECT は、独自に ORDER BY 句を持つことができませんが、UNION はグループではなくローに対して動作します。

Section

3.18 SELECT DISTINCT

SELECT DISTINCT キーワードは、候補結果セットからすべての重複（ある場合）を削除します。

```
<query_specification> ::= SELECT
                          [ DISTINCT ]
                          [ <row_range> ]
                          <select_list>
                          [ <select_into> ]
                          [ <from_clause> ]
                          [ <where_clause> ]
                          [ <group_by_clause> ]
                          [ <having_clause> ]
```

Note

「SELECT ALL * FROM employee」のように、ALLキーワードを明示的に指定できますが、これはデフォルトであり、すべてのローを選択するという、明白なことを指定しているだけであるため、構文には示していません。

重複ローとは、すべてのselectリスト項目が、別のローの対応する項目と同じ値であるローのことです。DISTINCTキーワードは、その後に続く最初のselectリスト項目だけでなく、selectリスト全体に適用されます。重複ローのセットごとに、すべてのローが1つのローを除いて削除されます。この処理は、GROUP BYで使用される処理に似ています。

たとえば、次のSELECTをASADEMOデータベースに対して実行すると13ローが返されますが、DISTINCTキーワードがない場合は91ローが返されます。

```
SELECT DISTINCT
       prod_id,
       line_id
  FROM sales_order_items
 WHERE line_id >= 3
 ORDER BY prod_id,
       line_id;
```

Note

DISTINCTキーワードを処理して値を比較する場合、NULL値は同じであると見なされます。これは、NULL値の通常の扱いとは異なります。NULL値に関する比較の場合、通常、結果はUNKNOWNです。

Section

3.19 FIRSTとTOP

FIRSTキーワードまたはTOP句を使用すると、候補結果セット内のロー数を制限できます。論理上、これはDISTINCTキーワードが適用されてから発生します。

```
<row_range>           ::= FIRST                    -- TOP 1と同じ
                        | TOP <maximum_row_count>
                              [ START AT <start_at_row_number> ]
<maximum_row_count>   ::= 返される最大ロー数を表す整数リテラル
<start_at_row_number> ::= 返される最初のロー数を表す整数リテラル
```

FIRST は、先頭のロー以外のすべてのローを単に破棄します。

TOP 句には、最大ロー・カウントとオプションの START AT 句を指定します。たとえば、TOP 4 を指定すると、先頭の 4 つのローだけが残り、他のすべてのローは破棄されます。TOP 4 START AT 3 を指定すると、3、4、5、6 番目のローだけが残ります。

FIRST は、複数のローを処理できないコンテキストで使用されることがあります。たとえば、プログラム変数を指定する INTO 句を伴う SELECT や、select リスト内のサブクエリなどです。select によって複数のローが返されるが、どれを使用するかは問題でない場合は、FIRST を使用すると、確実に 1 つだけが返されます。どのローを取得するかが問題の場合は、ORDER BY 句を使用して、目的のローを先頭にソートします。

TOP と START では、整数リテラルのみを使用できます。変数を使用する場合は、EXECUTE IMMEDIATE を使用します。次に、ストアド・プロシージャを呼び出して注文項目の 15 ページ目を表示する例を示します。ここでは、「1 ページ」を 10 ローとして定義しています。

```
CREATE PROCEDURE p_pagefull (
   @page_number INTEGER )
BEGIN
   DECLARE @page_size   INTEGER;
   DECLARE @start       INTEGER;
   DECLARE @sql         LONG VARCHAR;
   SET @page_size = 10;
   SET @start = 1;
   SET @start = @start + ( ( @page_number - 1 ) * @page_size );
   SET @sql = STRING (
      'SELECT TOP ',
      @page_size,
      ' START AT ',
      @start,
      ' * FROM sales_order ORDER BY order_date' );
   EXECUTE IMMEDIATE @sql;
END;

CALL p_pagefull ( 15 );
```

ASADEMO データベースに対してプロシージャ・コールを実行して返される結果セットは次のようになります。CREATE PROCEDURE 文と EXECUTE IMMEDIATE 文の詳細については、「第 8 章 パッケージ」を参照してください。

```
id    cust_id  order_date  fin_code_id  region   sales_rep
====  =======  ==========  ===========  =======  =========
2081  180      2000-06-03  r1           Eastern  129
2241  123      2000-06-03  r1           Canada   856
2242  124      2000-06-04  r1           Eastern  299
2243  125      2000-06-05  r1           Central  667
```

```
2244  126      2000-06-08  r1          Western  129
2245  127      2000-06-09  r1          South    1142
2246  128      2000-06-10  r1          Eastern  195
2247  129      2000-06-11  r1          Eastern  690
2248  130      2000-06-12  r1          Central  1596
2029  128      2000-06-12  r1          Eastern  856
```

このようなページ単位でのデータの取得は、状況によっては効果的です。たとえば、Web アプリケーションでは、クライアントとの対話の間、膨大な結果セットをそのままの状態にしておいたり、カーソルをオープンしたままにしておくことは望ましくありません。

Tip

最大値や最小値に関する要求を耳にした場合は、FIRST と TOP を ORDER BY と一緒に使用することを検討してください。この組み合わせは、MAX や MIN より簡単に問題を解決できます。

Section

3.20 NUMBER(*)

NUMBER(*) 関数は、select によって返された最終結果セット内のロー番号を返します。この関数は、FIRST、TOP、DISTINCT、ORDER BY など、他のすべての句が結果セットに対する処理を終了してから評価されます。そのため、事前に処理される WHERE 句や select の他の部分ではなく、select リスト自体の中でのみ NUMBER(*) を参照できます。

次に、姓が "D" で始まるすべての従業員の電話番号と順番を表示する例を示します。

```
SELECT NUMBER(*) AS "#",
       STRING ( emp_lname, ', ', emp_fname ) AS full_name,
       STRING ( '(', LEFT ( phone, 3 ), ') ',
                SUBSTR ( phone, 4, 3 ), '-',
                RIGHT ( phone, 4 ) ) AS phone
  FROM employee
 WHERE emp_lname LIKE 'd%'
 ORDER BY emp_lname,
       emp_fname;
```

結果セットは次のようになります。WHERE と ORDER BY が終了してから番号が付けられることに注意してください。

```
#  full_name         phone
=  ===============  ==============
1  Davidson, Jo Ann  (617) 555-3870
2  Diaz, Emilio      (617) 555-3567
3  Dill, Marc        (617) 555-2144
4  Driscoll, Kurt    (617) 555-1234
```

NUMBER(*) と ORDER BY を一緒に使用すると、SELECT INTO、および INSERT 文と SELECT 文でシーケンス番号を生成できます。この方法は、DEFAULT AUTOINCREMENT 機能の代替として役立つことがあります。次の例では、SELECT INTO #t によって最初にテンポラリ・テーブルを作成し、"D" で始まる番号付きの名前をすべて挿入します。次に、INSERT と SELECT を使用して、"E" で始まる番号付きの名前をすべて、そのテンポラリ・テーブルに追加します。最後に、文字と番号でソートした結果を表示します。

```
SELECT NUMBER(*)             AS "#",
       LEFT ( emp_lname, 1 ) AS letter,
       STRING ( emp_fname, ' ', emp_lname ) AS full_name
  INTO #t
  FROM employee
 WHERE emp_lname LIKE 'D%'
 ORDER BY emp_lname,
       emp_fname;

INSERT #t
SELECT NUMBER(*)             AS "#",
       LEFT ( emp_lname, 1 ) AS letter,
       STRING ( emp_fname, ' ', emp_lname ) AS full_name
  FROM employee
 WHERE emp_lname LIKE 'E%'
 ORDER BY emp_lname,
       emp_fname;

SELECT "#",
       full_name
  FROM #t
 ORDER BY letter,
       "#";
```

SELECT が最終的に生成する結果は次のようになります。他にも、これと同じことをより適切に実現する方法がありますが、この NUMBER(*) を使用した例では、ソートに使用された元のデータが破棄された後でも順番を保持できます。

```
#  full_name
=  ================
1  Jo Ann Davidson
```

```
2  Emilio Diaz
3  Marc Dill
4  Kurt Driscoll
1  Melissa Espinoza
2  Scott Evans
```

DEFAULT AUTOINCREMENT と SELECT INTO テンポラリ・テーブルの詳細については、「第1章　作成」を参照してください。INSERT 文の詳細については、「第2章　挿入」を参照してください。

NUMBER(*) は、UPDATE 文の SET 句内の新しい値としても使用できます。詳細については、「4.4 集合型 UPDATE の実行ロジック」を参照してください。

Section

3.21 INTO句

select の INTO 句は、まったく異なる2つの目的で使用できます。1つは、ローを作成し、シャープ記号（#）で始まる名前を持つテンポラリ・テーブルにそのローを挿入することです。もう1つは、単一ローの結果セットの select リストに含まれる値をプログラム変数に格納することです。ここでは、プログラム変数について説明します。テンポラリ・テーブルの作成の詳細については、「1.15.2.3　SELECT INTO #table_name」を参照してください。

```
<select_into>               ::= INTO <temporary_table_name>
                              | INTO <select_into_variable_list>
<temporary_table_name>      ::=「第1章　作成」の<temporary_table_name>を参照
<select_into_variable_list> ::= <non_temporary_identifier>
                                    { "," <non_temporary_identifier> }
<non_temporary_identifier>  ::=「第1章　作成」の<non_temporary_identifier>を参照
```

次に示す例では、2つのプログラム変数を使用して、最も多くのローを持つテーブルの名前とそのロー数を記録します。ASADEMO データベースに対して実行すると、サーバ・コンソール・ウィンドウに「SYSPROCPARM has the most rows: 1632」（最も多くのローを持つのは SYSPROCPARM：1632）というメッセージが表示されます。

```
BEGIN
   DECLARE @table_name VARCHAR ( 128 );
   DECLARE @row_count  BIGINT;

   CHECKPOINT;

   SELECT FIRST
```

```
          table_name,
          count
     INTO @table_name,
          @row_count
     FROM SYSTABLE
    ORDER BY count DESC;

   MESSAGE STRING (
      @table_name,
      ' has the most rows: ',
      @row_count ) TO CONSOLE;
END;
```

Note

SYSTABLE.count カラムには、直前のチェックポイントにおけるテーブル内のロー数が格納されます。上の例では明示的な CHECKPOINT コマンドを使用して、SYSTABLE.count を最新の状態にしています。ローの最大数を見つけるためにすべてのテーブルに対して「SELECT COUNT(*)」を計算する方法もありますが、テーブルが大きい場合、実行に時間がかかり、コーディングも難しくなります。

BEGIN ブロックと DECLARE 文の詳細については、「第 8 章　パッケージ」を参照してください。

Section

3.22 UNION、EXCEPT、INTERSECT

複数の結果セットを UNION、EXCEPT、INTERSECT の各演算子で比較および結合して、元の結果セットのユニオン、差、共通の部分である結果セットをそれぞれ生成できます。

```
<select>              ::= [ <with_clause> ]        -- WITH...
                          <query_expression>    -- 少なくとも 1 つの SELECT...
                          [ <order_by_clause> ] -- ORDER BY...
                          [ <for_clause> ]      -- FOR...
<query_expression>    ::= <query_expression> <query_operator> <query_expression>
                        | <subquery>
                        | <query_specification>
<query_operator>      ::= EXCEPT    [ DISTINCT | ALL ]
                        | INTERSECT [ DISTINCT | ALL ]
                        | UNION     [ DISTINCT | ALL ]
```

比較には、結果セット内のすべてのカラムが関係します。1 番目の結果セット内のあるローのすべてのカラム値が、2 番目の結果セット内のローの対応する値とまったく同じ場合、2 つのローは同じであり、

そうでない場合は、2つのローは異なります。このことは、両方の結果セット内のローが、同じ数のカラムを持つ必要があることを意味します。

Note

EXCEPT、INTERSECT、UNIONの各演算子を評価するときにローを比較する場合、NULL値は同じものとして扱われます。

「A EXCEPT B」の演算は、結果セットAに存在し、Bには存在しないすべてのローを返します。これは、「A-B」（Aマイナス B）とも呼ばれます。「A EXCEPT B」と「B EXCEPT A」は異なることに注意してください。

「A INTERSECT Bは」、Aにのみ存在するローやBにのみ存在するローではなく、AとBの両方に存在するすべてのローを返します。

「A UNION B」は、AとBの両方に含まれるすべてのローを返します。これは、「A+B」（Aプラス B）と呼ばれます。

DISTINCTキーワードを指定すると、最終結果セットに重複ローが含まれません。一方、ALLは重複を許可します。デフォルトはDISTINCTです。「A EXCEPT ALL B」が重複を返すのは、Aに重複ローが既に存在する場合だけです。「A INTERSECT ALL B」が重複を返すのは、対応するローがAとBで重複する場合だけです。「A UNION ALL B」には、重複が含まれることもあれば含まれないこともあり、重複は一方、他方、またはAとBの両方に含まれます。

次の例では、ASADEMOデータベース内のcustomer.stateとemployee.stateのDISTINCT値を使用して、EXCEPT、INTERSECT、UNIONを示します。次に示す7種類のselectを使用します。

- customer.stateのDISTINCT値
- employee.stateのDISTINCT値
- customer_statesからemployee_statesを除く（EXCEPT）
- employee_statesからcustomer_statesを除く（EXCEPT）
- customer_statesとemployee_statesの「排他的OR」（XOR）。両方のテーブルではなく、いずれか一方のテーブルにのみ存在するstates
- customer_statesとemployee_statesの共通部分（INTERSECT）
- customer_statesとemployee_statesのユニオン（UNION）

これらのselectでは、EXCEPT、INTERSECT、UNIONの各演算にの他に抽出テーブルを使用し、stateのDISTINCT値の結果セットを計算します。LIST関数は簡略な出力を生成し、COUNT関数は各リスト内のエントリ数を計算します。

```
SELECT COUNT(*) AS count,
       LIST ( state ORDER BY state ) AS customer_states
  FROM ( SELECT DISTINCT state
           FROM customer )
       AS customer;

SELECT COUNT(*) AS count,
       LIST ( state ORDER BY state ) AS employee_states
  FROM ( SELECT DISTINCT state
           FROM employee )
       AS employee;

SELECT COUNT(*) AS count,
       LIST ( state ORDER BY state ) AS customer_except_employee
  FROM ( SELECT state
           FROM customer
         EXCEPT
         SELECT state
           FROM employee )
       AS customer_except_employee;

SELECT COUNT(*) AS count,
       LIST ( state ORDER BY state ) AS employee_except_customer
  FROM ( SELECT state
           FROM employee
         EXCEPT
         SELECT state
           FROM customer )
       AS employee_except_customer;

SELECT COUNT(*) AS count,
       LIST ( state ORDER BY state ) AS customer_xor_employee
  FROM ( ( SELECT state
             FROM customer
           EXCEPT
           SELECT state
             FROM employee )
       UNION ALL
         ( SELECT state
             FROM employee
           EXCEPT
           SELECT state
             FROM customer ) )
       AS customer_xor_employee;

SELECT COUNT(*) AS count,
       LIST ( state ORDER BY state ) AS customer_intersect_employee
  FROM ( SELECT state
           FROM customer
```

```
        INTERSECT
        SELECT state
          FROM employee )
      AS customer_intersect_employee;

SELECT COUNT(*) AS count,
       LIST ( state ORDER BY state ) AS customer_union_employee
  FROM ( SELECT state
           FROM customer
         UNION
         SELECT state
           FROM employee )
       AS customer_intersect_employee;
```

結果は次のようになります。各SELECTは異なるカウントをそれぞれ生成し、2つのEXCEPTの結果が異なることに注意してください。特に、各リストにおけるCA、AZ、ABの有無は、EXCEPT、INTERSECT、UNIONの違いをよく示しています。

```
count  LIST of states
=====  ==============
36     AB,BC,CA,CO,CT,DC,FL,GA,IA,IL,IN,KS,LA,MA,  -- customer_states
       MB,MD,MI,MN,MO,NC,ND,NJ,NM,NY,OH,ON,OR,PA,
       PQ,TN,TX,UT,VA,WA,WI,WY

16     AZ,CA,CO,FL,GA,IL,KS,ME,MI,NY,OR,PA,RI,TX,  -- employee_states
       UT,WY

23     AB,BC,CT,DC,IA,IN,LA,MA,MB,MD,MN,MO,NC,ND,  -- customer_except_employee
       NJ,NM,OH,ON,PQ,TN,VA,WA,WI

3      AZ,ME,RI                                    -- employee_except_customer

26     AB,AZ,BC,CT,DC,IA,IN,LA,MA,MB,MD,ME,MN,MO,  -- customer_xor_employee
       NC,ND,NJ,NM,OH,ON,PQ,RI,TN,VA,WA,WI

13     CA,CO,FL,GA,IL,KS,MI,NY,OR,PA,TX,UT,WY      -- customer_intersect_employee

39     AB,AZ,BC,CA,CO,CT,DC,FL,GA,IA,IL,IN,KS,LA,  -- customer_union_employee
       MA,MB,MD,ME,MI,MN,MO,NC,ND,NJ,NM,NY,OH,ON,
       OR,PA,PQ,RI,TN,TX,UT,VA,WA,WI,WY
```

EXCEPT、INTERSECT、UNIONの3つの演算子の中では、UNIONが最も役立ちます。UNIONは、分断統治法による問題の解決を促進します。1つの複雑なselectを記述するより、複数の単純なselectを記述する方が通常は簡単です。複数のselectのUNIONは、1つのSELECTより高速な場合があります。特に、UNIONを使用してブール式からOR演算子を削除すると、それが顕著になります。これは、ORの最適化は困難だがUNIONの計算は簡単なためで、特にUNION ALLの計算は簡単です。

Tip

UNION ALLは高速であるため、使用できない場合を除いて、常にUNION ALLを使用してください。個々の結果セットに重複がないことがわかっている場合、または重複があっても問題ない場合は、UNION ALLを使用します。アプリケーションの中で重複を削除する方が、サーバにその処理を実行させるより高速な場合もあります。

次に、姓が"K"で始まるすべての顧客と従業員の電話番号を表示する例を示します。結果セットには文字列リテラルの'Customer'と'Employee'が含まれており、最終的なUNION ALLの中でデータの出所が保持されます。

```
SELECT STRING ( customer.lname, ', ', customer.fname ) AS full_name,
       STRING ( '(', LEFT ( customer.phone, 3 ), ') ',
                SUBSTR ( customer.phone, 4, 3 ), '-',
                RIGHT ( customer.phone, 4 ) )          AS phone,
       'Customer'                                      AS relationship
  FROM customer
 WHERE customer.lname LIKE 'k%'
UNION ALL
SELECT STRING ( employee.emp_lname, ', ', employee.emp_fname ),
       STRING ( '(', LEFT ( employee.phone, 3 ), ') ',
                SUBSTR ( employee.phone, 4, 3 ), '-',
                RIGHT ( employee.phone, 4 ) ),
       'Employee'
  FROM employee
 WHERE employee.emp_lname LIKE 'k%'
 ORDER BY 1;
```

最終結果は次のとおりです。

```
full_name         phone           relationship
================  ==============  ============
Kaiser, Samuel    (612) 555-3409  Customer
Kelly, Moira      (508) 555-3769  Employee
King, Marilyn     (219) 555-4551  Customer
Klobucher, James  (713) 555-8627  Employee
Kuo, Felicia      (617) 555-2385  Employee
```

INTO句が最初のSELECTにのみある場合、INTO #table_name句とUNIONを一緒に使用できます。次に示す例では、テンポラリ・テーブルを作成し、customerテーブルとemployeeテーブルに含まれる"K"で始まるすべての名前をそのテンポラリ・テーブルに入れます。

```
SELECT customer.lname AS last_name
  INTO #last_name
```

```
  FROM customer
 WHERE customer.lname LIKE 'k%'
UNION ALL
SELECT employee.emp_lname
  FROM employee
 WHERE employee.emp_lname LIKE 'k%';

SELECT *
  FROM #last_name
 ORDER BY 1;
```

#last_name テーブルの内容は次のようになります。

```
last_name
=========
Kaiser
Kelly
King
Klobucher
Kuo
```

テンポラリ・テーブルを作成するこの方法の詳細については、「1.15.2.3　SELECT INTO #table_name」を参照してください。

一連の EXCEPT、INTERSECT、UNION 演算の先頭のクエリでは、最終結果セット内のカラムのエイリアス名を確立します。ただし、これはデータ型には当てはまりません。SQL Anywhere では、すべてのクエリ内の対応する select リスト項目を調べて、最終結果セットのデータ型を決定します。

Tip

UNION 内のデータ型には注意してください。具体的に言うと、一連の EXCEPT、INTERSECT、UNION 演算内の各クエリ内の各 select リスト項目は、一連の演算内の他のすべてのクエリ内の対応する項目のデータ型と完全に一致している必要があります。データ型が同じでない場合、または同じであることが確実でない場合は、CAST を使用してデータ型を強制的に一致させてください。この操作を行わないと、目的の結果が得られない可能性があります。たとえば、VARCHAR (100) と VARCHAR (10) の UNION を実行すると、結果は VARCHAR (100) になります（これは問題ありません）。しかし、VARCHAR と BINARY の UNION を実行すると、結果は LONG BINARY になります。特に、大文字と小文字を区別しない文字列比較を行う場合、この結果は望ましくない可能性があります。

Section

3.23 CREATE VIEW

CREATE VIEW 文を使用すると、select を永続的に記録できます。この select は、他の select の FROM 句の中で、テーブルと同じように参照できます。

```
<create_view>           ::= CREATE VIEW [ <owner_name> "."] <view_name>
                            [ <view_column_name_list> ]
                            AS
                            [ <with_clause> ]     -- WITH...
                            <query_expression>    -- 少なくとも1つのSELECT...
                            [ <order_by_clause> ] -- ORDER BY...
                            [ <for_xml_clause> ]
                            [ WITH CHECK OPTION ]
<view_column_name_list> ::= "(" [ <alias_name_list> ] ")"
```

ビューは、複雑さを隠すのに便利です。たとえば、次の CREATE VIEW は、SQL Anywhere システム・テーブルに関するかなり複雑な SELECT を含んでいます。

```
CREATE VIEW v_parent_child AS
SELECT USER_NAME ( parent_table.creator ) AS parent_owner,
       parent_table.table_name            AS parent_table,
       USER_NAME ( child_table.creator )  AS child_owner,
       child_table.table_name             AS child_table
  FROM SYS.SYSFOREIGNKEY AS foreign_key
       INNER JOIN
          ( SELECT table_id,
                   creator,
                   table_name
              FROM SYS.SYSTABLE
             WHERE table_type = 'BASE' ) -- VIEW類以外
       AS parent_table
       ON parent_table.table_id = foreign_key.primary_table_id
       INNER JOIN
          ( SELECT table_id,
                   creator,
                   table_name
              FROM SYS.SYSTABLE
             WHERE table_type = 'BASE' ) -- VIEW類以外
       AS child_table
       ON  child_table.table_id =  foreign_key.foreign_table_id;
```

SYSTABLE テーブルには、データベース内の各テーブルに関する情報が含まれています。SYSFOREIGNKEY は、SYSTABLE 内の親ローと子ローを結ぶ多対多関係テーブルです。

USER_NAME は、1 などの数字のユーザ番号を、対応するユーザ ID である 'DBA' に変換する組み込み関数です。v_parent_child ビューは、データベース内の各外部キー定義の親テーブルと子テーブルについて、その所有者とテーブル名を結果セットとして生成します。SYSFOREIGNKEY にはテーブル名が含まれず、数値の table_id 値だけが含まれているため、INNER JOIN 演算が必要です。目的の名前が含まれているのは SYSTABLE です。

Note

すべての SQL Anywhere データベースには、たとえば SYSFOREIGNKEYS など、これに似た定義済みのビューが用意されています。

次の SELECT では、v_parent_child を使用して、'DBA' が所有するテーブルに関するすべての外部キー関係を表示します。この SELECT は、単純で理解しやすく、基本のビュー定義より簡単です。

```
SELECT parent_owner,
       parent_table,
       child_owner,
       child_table
  FROM v_parent_child
 WHERE parent_owner = 'DBA'
   AND child_owner = 'DBA'
 ORDER BY 1, 2, 3, 4;
```

ASADEMO データベースに対してこの SELECT を実行すると、結果セットは次のようになります。

```
parent_owner  parent_table  child_owner  child_table
============  ============  ===========  ================
DBA           customer      DBA          sales_order
DBA           department    DBA          employee
DBA           employee      DBA          department
DBA           employee      DBA          sales_order
DBA           fin_code      DBA          fin_data
DBA           fin_code      DBA          sales_order
DBA           product       DBA          sales_order_items
DBA           sales_order   DBA          sales_order_items
```

Tip

安易にビューを作成しないようにしてください。特に、アプリケーションとスキーマの変更を切り離すことを目的として、単にすべてのカラムを選択するビューをすべてのテーブルに作成するのは避けてください。この方法の場合、管理するスキーマ・オブジェクトの数が倍になり、実際にはメリットはありません。スキーマの変更は、アプリケーションに影響しないか、アプリケーションの保守を必要とするかどちらかですが、不明瞭な部分が増えても得にはなりません。また、カラム名を理解しやすいものにするためだけにビューを作成しないください。ベース・テーブル自体に、理解しやすいカラム名を付けて

ください。変に凝った名前付け規則は既に過去のものであり、現在では通用しません。

Tip

ビューが過度に複雑なために生じるパフォーマンスの問題に注意してください。ビューは、ビューを使用するクエリが実行されるたびに、最初から評価および実行されます。たとえば、すべてのテーブルとすべてのクエリに影響する複雑なセキュリティ認証スキーマを、マルチテーブル・ジョインを含むビューを使用して実装したとしたら、パフォーマンスが犠牲になることでしょう。ビューは、開発者に対しては複雑さを隠せますが、クエリ・オプティマイザに対しては複雑さを隠せません。さまざまなFROM句の中で実質的に数十または数百のテーブル参照が行われるマルチビュー・ジョインは、適切に処理できないことがあります。

更新、挿入、削除可能なビューの場合、ビューを使用してローのUPDATE、INSERT、DELETEをそれぞれ実行できます。更新する必要があるベース・テーブル内のローを特定できる場合は、ビューは更新可能です。このことは、DISTINCT、GROUP BY、UNION、EXCEPT、INTERSECT、または集合関数参照を使用したビューは更新可能でないことを意味します。ビューが更新可能で、1つのテーブルだけが含まれる場合、ビューは挿入可能です。削除可能なビューにも、同じルールが当てはまります（つまり、1つのテーブルだけが含まれ、更新可能であることが必要です）。

オプションのWITH CHECK OPTION句は、ビューを伴うINSERT演算とUPDATE演算に適用されます。この句を指定した場合、これらの演算時にビュー定義がチェックされ、影響を受けるすべてのローがビュー自体によって選択される場合にのみ、演算が許可されます。詳細については、SQL Anywhereのオンラインヘルプを参照してください。本書では、次の例に示す以外には、更新可能ビューについては説明しません。

```
CREATE TABLE parent (
   key_1         INTEGER NOT NULL PRIMARY KEY,
   non_key_1     INTEGER NOT NULL );

CREATE VIEW v_parent AS
SELECT *
  FROM parent;

CREATE TABLE child (
   key_1         INTEGER NOT NULL REFERENCES parent ( key_1 ),
   key_2         INTEGER NOT NULL,
   non_key_1     INTEGER NOT NULL,
   PRIMARY KEY ( key_1, key_2 ) );

CREATE VIEW v_child AS
SELECT *
  FROM child;
```

```
CREATE VIEW v_family (
   parent_key_1,
   parent_non_key_1,
   child_key_1,
   child_key_2,
   child_non_key_1 ) AS
SELECT parent.key_1,
       parent.non_key_1,
       child.key_1,
       child.key_2,
       child.non_key_1
  FROM parent
       INNER JOIN child
               ON child.key_1 = parent.key_1;

INSERT v_parent VALUES ( 1, 444 );
INSERT v_parent VALUES ( 2, 555 );
INSERT v_parent VALUES ( 3, 666 );

INSERT v_child VALUES ( 1, 77, 777 );
INSERT v_child VALUES ( 1, 88, 888 );
INSERT v_child VALUES ( 2, 99, 999 );
INSERT v_child VALUES ( 3, 11, 111 );

UPDATE v_family
   SET parent_non_key_1 = 1111,
       child_non_key_1  = 2222
 WHERE parent_key_1 = 1
   AND child_key_2  = 88;

DELETE v_child
 WHERE key_1 = 3
   AND key_2 = 11;

SELECT * FROM v_family
 ORDER BY parent_key_1,
       child_key_2;
```

上のINSERT文とDELETE文は、v_parentとv_childが挿入可能、削除可能、更新可能であるために動作します。ただし、v_familyビューには2つのテーブルが含まれるために、更新のみ可能で、挿入と削除は可能ではありません。1つのUPDATE文が、2つの異なるテーブルでそれぞれ1つのローを変更していることに注意してください。最後のSELECTの結果セットは次のようになります。

```
parent_key_1  parent_non_key_1  child_key_1  child_key_2  child_non_key_1
============  ================  ===========  ===========  ===============
1             1111              1            77           777
```

```
1          1111          1          88          2222
2          555           2          99          999
```

Section

3.24 WITH句

WITH句を使用すると、1つ以上のローカル・ビューを定義できます。WITH句は、1つ以上のselectが含まれるクエリ式の前に追加します。WITH句で定義されるローカル・ビューは、そのselect内で使用できます。RECURSIVEキーワードは、1つ以上のローカル・ビューを再帰ユニオン演算で使用できることを指定します。再帰ユニオンの詳細については、次の項で説明します。

```
<select>                      ::= [ <with_clause> ]        -- WITH...
                                  <query_expression>    -- 少なくとも1つのSELECT
                                  [ <order_by_clause> ] -- ORDER BY...
                                  [ <for_clause> ]      -- FOR...
<with_clause>                 ::= WITH [ RECURSIVE ] <local_view_list>
<local_view_list>             ::= <local_view> { "," <local_view> }
<local_view>                  ::= <local_view_name>
                                  [ <local_view_column_name_list> ]
                                  AS <subquery>
<local_view_name>             ::= <identifier>
<local_view_column_name_list> ::= "(" [ <alias_name_list> ] ")"
```

Note

SQL Anywhereのオンラインヘルプでは、「ローカル・ビュー」ではなく、「テンポラリ・ビュー」という用語を使用しています。しかし、テンポラリ・テーブルとは異なり、このビューは、WITH句が追加されたselect内でローカルに参照されるだけです。「テンポラリ」という用語には、接続が切断されるまでビュー定義が保持されるという意味合いがあります。CREATE TEMPORARY VIEWというものはないので、本書では、代わりに、「ローカル・ビュー」という表現を使用しています。

WITH句を使用すると、コード内の重複を避けるのに使用できます。WITH句で定義した1つのローカル・ビューは、以降のselectのFROM句で、名前で何度も参照できます。たとえば、前の項のv_parent_childの例は、base_tableという名前の1つのローカル・ビューを使用して、2つの同一の抽出テーブル定義を置き換えることで簡略化できます。CREATE VIEWの内部でWITH句を使用する、つまり永続ビューの中でローカル・ビューを定義しても問題はありません。

```
CREATE VIEW v_parent_child AS
WITH base_table AS
```

```
        ( SELECT table_id,
                 creator,
                 table_name
            FROM SYS.SYSTABLE
           WHERE table_type = 'BASE' )
 SELECT USER_NAME ( parent_table.creator ) AS parent_owner,
        parent_table.table_name            AS parent_table,
        USER_NAME ( child_table.creator )  AS child_owner,
        child_table.table_name             AS child_table
   FROM SYS.SYSFOREIGNKEY AS foreign_key
        INNER JOIN base_table
                AS parent_table
                ON parent_table.table_id = foreign_key.primary_table_id
        INNER JOIN base_table
                AS child_table
                ON child_table.table_id =  foreign_key.foreign_table_id;
```

SELECT、CREATE VIEW、INSERT の各文の、最も外側の SELECT の前にのみ WITH 句を指定できます。ネストされたクエリ式の内側の任意の位置でローカル・ビュー名を参照できるため、これはそれほど大きな制限ではありません。サブクエリの内側に、もっと多くの WITH 句を記述できないというだけのことです。

3.24.1 再帰ユニオン

再帰ユニオンは、WITH 句を使用して、2つのクエリの UNION ALL に基づくローカル・ビューを定義する、特別な方法です。

- ローカル・ビュー内の1番目のクエリは「初期シード・クエリ」であり、1つまたは複数のローを提供してプロセスを開始する
- 2番目のクエリには、ローカル・ビュー名自体への再帰参照が含まれ、1番目のクエリで生成された初期結果セットに複数のローを追加する。再帰が動作するためには、WITH 句に RECURSIVE キーワードを指定する必要がある

WITH 句全体は、同じくローカル・ビューを参照する3番目の外部クエリの前に指定します。プロセス全体を駆動し、実際の結果セットを生成するのはこの外部クエリです。

次に、典型的な再帰ユニオンの構文を示します。

```
<typical_recursive_union>       ::= WITH RECURSIVE <local_view_name>
                                        "(" <alias_name_list> ")"
                                    AS "(" <initial_query_specification>
                                           UNION ALL
                                           <recursive_query_specification> ")"
```

```
                                    <outer_query_specification>
                                    [ <order_by_clause> ]
                                    [ <for_clause> ]
<initial_query_specification>   ::= シード・ローを提供する <query_specification>
<recursive_query_specification> ::= <local_view_name> を再帰的に参照する <query_specification>
<outer_query_specification>     ::= <local_view_name> を参照する <query_specification>
```

Note

再帰プロセスは、自身に関して定義されるプロセスです。値の階乗を考えてみましょう。たとえば、6の階乗は「6 * 5 * 4 * 3 * 2 * 1」として、つまり 720 として定義されます。そこで、階乗の公式は、「factorial (n) = n * factorial (n - 1)」という再帰定義を使用して記述できます。これは、複雑なプロセスを考えるうえで好都合であり、考えたようにコーディングできるのは便利です。SQL Anywhere では、階乗のような再帰関数をコーディングできます。CREATE FUNCTION 文の詳細については、「第8章　パッケージ」の 8.10 項を参照してください。ここでは、それとは別の再帰プロセス、つまり再帰ユニオンについて説明します。

再帰ユニオンを使用すると、データの階層関係を処理できます。データの階層は、同じテーブル内の異なるローが互いの子や親として動作する、自己参照型の外部キー関係を伴うことがあります。階層のレベル数が多くなる場合は特に、通常の SQL でこれらの関係を処理することは非常に困難です。

図 3-1 は、まさにこのような関係を示しています。これは 14 名の従業員で構成される会社の組織図で、矢印はレポートの構造を表します（Briana、Calista、Delmar はすべて Ainslie にレポートし、Electra は Briana にレポートします。他についても同様です）。

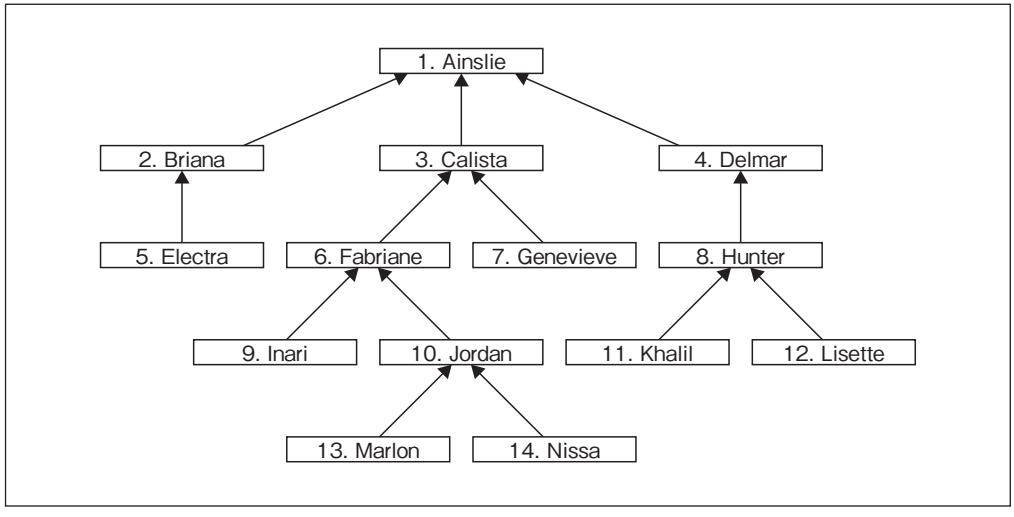


図 3-1　組織図

次に、図3-1の組織図を表すテーブル定義とデータを示します。employee_idは、各従業員を識別するプライマリ・キーで、manager_idカラムは、図3-1の矢印に相当する、従業員の上司を指します。nameカラムとsalaryカラムには、従業員に関するデータが格納されます。「employee_id = 1」のmanager_idを1に設定することは、Ainslieが図の最上位に位置し、社内のだれにもレポートしないことを意味します。

```
CREATE TABLE employee (
   employee_id  INTEGER NOT NULL,
   manager_id   INTEGER NOT NULL REFERENCES employee ( employee_id ),
   name         VARCHAR ( 20 ) NOT NULL,
   salary       NUMERIC ( 20, 2 ) NOT NULL,
   PRIMARY KEY ( employee_id ) );

INSERT INTO employee VALUES ( 1,  1,  'Ainslie',  1000000.00 );
INSERT INTO employee VALUES ( 2,  1,  'Briana',    900000.00 );
INSERT INTO employee VALUES ( 3,  1,  'Calista',   900000.00 );
INSERT INTO employee VALUES ( 4,  1,  'Delmar',    900000.00 );
INSERT INTO employee VALUES ( 5,  2,  'Electra',   750000.00 );
INSERT INTO employee VALUES ( 6,  3,  'Fabriane',  800000.00 );
INSERT INTO employee VALUES ( 7,  3,  'Genevieve', 750000.00 );
INSERT INTO employee VALUES ( 8,  4,  'Hunter',    800000.00 );
INSERT INTO employee VALUES ( 9,  6,  'Inari',     500000.00 );
INSERT INTO employee VALUES ( 10, 6,  'Jordan',    100000.00 );
INSERT INTO employee VALUES ( 11, 8,  'Khalil',    100000.00 );
INSERT INTO employee VALUES ( 12, 8,  'Lisette',   100000.00 );
INSERT INTO employee VALUES ( 13, 10, 'Marlon',    100000.00 );
INSERT INTO employee VALUES ( 14, 10, 'Nissa',     100000.00 );
```

Note

ここに示したemployeeテーブルは、ASADEMOデータベース内のemployeeテーブルとは異なります。

次に、「Ainslieに至るまでに、Marlonの上位にはだれがいるか?」という質問に答えるSELECTを示します。

```
WITH RECURSIVE superior_list
   ( level,
     chosen_employee_id,
     manager_id,
     employee_id,
     name )
AS ( SELECT CAST ( 1 AS INTEGER ) AS level,
            employee.employee_id    AS chosen_employee_id,
            employee.manager_id     AS manager_id,
            employee.employee_id    AS employee_id,
```

```
            employee.name           AS name
       FROM employee
     UNION ALL
     SELECT superior_list.level + 1,
            superior_list.chosen_employee_id,
            employee.manager_id,
            employee.employee_id,
            employee.name
       FROM superior_list
            INNER JOIN employee
                    ON employee.employee_id = superior_list.manager_id
      WHERE superior_list.level <= 99
        AND superior_list.manager_id <> superior_list.employee_id )
SELECT superior_list.level,
       superior_list.name
  FROM superior_list
 WHERE superior_list.chosen_employee_id = 13
 ORDER BY superior_list.level DESC;
```

最終結果セットは、Marlon と Ainslie の間に Jordan、Fabriane、Calista がいる 5 レベルの階層を示しています。

```
level  name
=====  ========
5      Ainslie
4      Calista
3      Fabriane
2      Jordan
1      Marlon
```

上の SELECT は次のように動作します。

❶ WITH RECURSIVE 句は、ローカル・ビューに superior_list という名前を設定し、そのローカル・ビュー内の 5 つのカラムのエイリアス名のリストを設定することによって開始します。

❷ ビューの結果セット内の各ローには、Marlon と Ainslie の間にいる、Marlon の上司の 1 人に関する情報が格納されます。終端も含まれるため、Marlon 自身のローもあります。

❸ ビューの level カラムには階層レベルが格納されます。一番下の Marlon の番号は 1 で、次のレベルは 2 です。以降についても同様です。

❹ chosen_employee_id カラムは、目的の従業員を識別します。この例の場合は、Marlon に関する質問であるため、Marlon に相当する 13 という固定値になります。言い換えると、すべてのローに 13 が格納されます。このしくみについては、以下のステップ 10 で説明します。

❺ employee_id と name がこのレベルの従業員を識別するのに対して、manager_id カラムは、この従業員の 1 つ上のレベルにいる従業員を識別します。

❻ UNION ALL内の最初のクエリは、employeeテーブルに含まれるすべてのローを選択し、そのすべてにレベル番号1を割り当てます。このローは、「この従業員の上司はだれか?」に関するあらゆるクエリの、一番下の開始ポイントです。これは再帰的でない「シード・クエリ」であり、ここから処理が始まります。実際、このクエリで生成されるローは1つだけです。そのしくみについては、以下のステップ10で説明します。

❼ UNION ALLの2番目のクエリは、employeeテーブル内のローと、superior_list結果セットに既に存在するローの間でINNER JOINを実行します。後者のローの始まりは、シード・クエリに含まれるローです。superior_list内の既存のローごとに、INNER JOINは、「ON employee.employee_id = superior_list.manager_id」を通じて、階層内の1つ上のレベルにあるemployeeローを見つけます。ローカル・ビューへのこの再帰参照が、WITH句にRECURSIVEキーワードを指定する理由です。

❽ UNION ALL内の2番目のクエリで結果セットに追加された新しい各ローに対して、superior_list内の既存のローより1つ上のレベル値が設定されます。chosen_employee_idには、superior_list内の既存のローのchosen_employee_idと同じ値が設定されます。他の3つのカラム（manager_id、employee_id、name）は、階層内の1つ上のレベルに位置する人物に相当する、employee内のローから取得されます。

❾ WHERE句は、再帰が暴走しないようにします。まず、levelに対する正常性検査を行い、99というあり得ない値が発生した場合にクエリを停止します。WHERE句内の2番目の述部である「superior_list.manager_id <> superior_list.employee_id」では、Ainslieのローに達すると再帰を停止します。ローの1つがsuperior_listに既に存在する状態であれば、Ainslieのローより上のレベルが検索されることはありません。

❿ 外側のSELECTによって、chosen_employee_idが13（Marlon）である、superior_list内のすべてのローが表示されます。外側のWHERE句は、Marlonのローを除いて、UNION ALL内の最初のクエリによるすべてのローを実質的に破棄します。また、Marlonの上のパス上のローを除いて、UNION ALL内の2番目のクエリで追加されたすべてのローも除外します。ORDER BYは、結果をレベルの降順でソートするため、Ainslieが一番上になり、Marlonは一番下になります。

Tip

再帰ユニオンの結果セット内のレベル番号と、値の妥当性検査を実行するWHERE句を、常に指定してください。データのループやクエリ内のバグによってクエリが暴走することがありますが、SQL Anywhereがエラーを発生する前に暴走を阻止するのが適当です。

CREATE VIEW文を使用すると、複雑な再帰ユニオンを格納して、複数の異なるクエリで使用できます。次に示すように、前のクエリはCREATE VIEW文によって永続ビューに変換できます。外側のSELECTを単純な「SELECT *」に変更し、クエリに名前を設定しています。

```
CREATE VIEW v_superior_list AS
WITH RECURSIVE superior_list
   ( level,
     chosen_employee_id,
     manager_id,
     employee_id,
     name )
AS ( SELECT CAST ( 1 AS INTEGER ) AS level,
            employee.employee_id    AS chosen_employee_id,
            employee.manager_id     AS manager_id,
            employee.employee_id    AS employee_id,
            employee.name           AS name
       FROM employee
     UNION ALL
     SELECT superior_list.level + 1,
            superior_list.chosen_employee_id,
            employee.manager_id,
            employee.employee_id,
            employee.name
       FROM superior_list
            INNER JOIN employee
                    ON employee.employee_id = superior_list.manager_id
      WHERE superior_list.level <= 99
        AND superior_list.manager_id <> superior_list.employee_id )
SELECT *
  FROM superior_list;
```

ビュー v_superior_list を使用することで、前の例の外部クエリは非常に単純なスタンドアロン・クエリになります。

```
SELECT v_superior_list.level,
       v_superior_list.name
  FROM v_superior_list
 WHERE v_superior_list.chosen_employee_id = 13
 ORDER BY v_superior_list.level DESC;
```

このクエリは、前とまったく同じ結果セットを生成します。

```
level  name
=====  ========
5      Ainslie
4      Calista
3      Fabriane
2      Jordan
1      Marlon
```

次に、同じビューを異なる方法で使用する別のクエリを示します。LIST 関数によってすべての上司が 1 行に表示され、WHERE 句によってリストから Khalil 自身の名前が削除されます。

```
SELECT LIST ( v_superior_list.name,
              ', then '
              ORDER BY v_superior_list.level ASC ) AS "Khalil's Superiors"
  FROM v_superior_list
 WHERE v_superior_list.chosen_employee_id = 11
   AND v_superior_list.level > 1;
```

上のクエリによる 1 行の結果は次のようになります。

```
Khalil's Superiors
==================
Hunter, then Delmar, then Ainslie
```

次に、「各従業員とその従業員の部下全員の総給与はいくらか?」など、トップダウンの質問に答えることができる再帰ユニオンの例を示します。

```
CREATE VIEW v_salary_list AS
WITH RECURSIVE salary_list
   ( level,
     chosen_employee_id,
     manager_id,
     employee_id,
     name,
     salary )
AS ( SELECT CAST ( 1 AS INTEGER ) AS level,
            employee.employee_id  AS chosen_employee_id,
            employee.manager_id   AS manager_id,
            employee.employee_id  AS employee_id,
            employee.name         AS name,
            employee.salary       AS salary
       FROM employee
     UNION ALL
     SELECT salary_list.level + 1,
            salary_list.chosen_employee_id,
            employee.manager_id,
            employee.employee_id,
            employee.name,
            employee.salary
       FROM salary_list
            INNER JOIN employee
                  ON employee.manager_id = salary_list.employee_id
      WHERE salary_list.level <= 99
        AND employee.manager_id <> employee.employee_id )
```

```
SELECT *
  FROM salary_list;
```

このビューは、前の例とは異なる動作をします。v_superior_list とは違って、v_salary_list は階層内をトップダウンで進みます。UNION ALL 内の最初のクエリは、前と同じようにすべての従業員からなる結果セットを生成しますが、2 番目のクエリは、v_superior_list 内の条件「ON employee.employee_id = superior_list.manager_id」に対して、条件「ON employee.manager_id = salary_list.employee_id」を使用することで、階層内で下にある employee ローを検索します。

v_salary_list を使用して、社内の従業員ごとに支払い給与総額を計算する方法を次に示します。employee テーブル内のローごとに、chosen_employee_id が employee.employee_id に一致するすべての v_salary_list.salary 値の SUM が、サブクエリによって計算されます。

```
SELECT employee.name,
       ( SELECT SUM ( v_salary_list.salary )
           FROM v_salary_list
          WHERE v_salary_list.chosen_employee_id
              = employee.employee_id )           AS payroll
  FROM employee
 ORDER BY 1;
```

最終結果セットは次のようになります。Ainslie の支払い給与総額は、全員の給与の合計です。一番下の Nissa の値には、Nissa 自身の給与が含まれ、他の人の給与は含まれません。

```
name       payroll
=========  ==========
Ainslie    7800000.00
Briana     1650000.00
Calista    3250000.00
Delmar     1900000.00
Electra     750000.00
Fabriane   1600000.00
Genevieve   750000.00
Hunter     1000000.00
Inari       500000.00
Jordan      300000.00
Khalil      100000.00
Lisette     100000.00
Marlon      100000.00
Nissa       100000.00
```

Section

3.25 UNLOAD TABLEとUNLOAD SELECT

UNLOAD TABLE 文と UNLOAD SELECT 文は、データベースからデータを選択し、それをフラット・ファイルに書き出す非常に効率的な手段です。

```
<unload>            ::= <unload_table>
                      | <unload_select>
<unload_table>      ::= UNLOAD [ FROM ] TABLE [ <owner_name> "."] <table_name>
                            TO <unload_filespec>
                            { <unload_table_option> }
<unload_select>     ::= UNLOAD <select_for_unload>
                            TO <unload_filespec>
                            { <unload_select_option> }
<select_for_unload> ::= [ <with_clause> ]
                            <query_expression>
                            [ <order_by_clause> ]
                            [ <for_xml_clause> ]
<unload_filespec>   ::= サーバを基準とする文字列リテラルのファイル指定
<unload_table_option>  ::= <unload_select_option>
                         | ORDER ( ON | OFF )              -- デフォルトは ON
<unload_select_option> ::= APPEND ( ON | OFF )            -- デフォルトは OFF
                         | DELIMITED BY <unload_delimiter> -- デフォルトは ','
                         | ESCAPE <escape_character>       -- デフォルトは '¥'
                         | ESCAPES ( ON | OFF )            -- デフォルトは ON
                         | FORMAT ( ASCII | BCP )          -- デフォルトは ASCII
                         | HEXADECIMAL ( ON | OFF )        -- デフォルトは ON
                         | QUOTES ( ON | OFF )             -- デフォルトは ON
<unload_delimiter>     ::= 長さが 1 ～ 255 文字の文字列リテラル
<escape_character>     ::= 長さが 1 文字のみの文字列リテラル
```

最初のフォーマットの UNLOAD TABLE は、2 番目のフォーマットの UNLOAD SELECT の限定版のようなものです。たとえば、次の 2 つの文では同じファイルが作成されます。

```
UNLOAD TABLE t1 TO 't1_a1.txt';
UNLOAD SELECT * FROM t1 TO 't1_a2.txt';
```

UNLOAD TABLE 文には ORDER というオプションがありますが、これは UNLOAD SELECT には適用されません。他のオプションは両方の文に適用されますが、UNLOAD SELECT の方が柔軟性に優れています。そのため、ここでは UNLOAD SELECT を中心に、2 つの文を一緒に説明します。

UNLOAD 文におけるファイル仕様のコーディング・ルールは、LOAD TABLE 文におけるファイル仕様のルールと同じです。詳細については、「2.3　LOAD TABLE」を参照してください。

UNLOAD 文は、テーブルまたは結果セット内のローごとに、1つのレコードを出力ファイルに書き込みます。各レコードは、最後のレコードを含めて、ASCIIの復帰と改行のペア"¥x0D¥x0A"で終了します。結果セット内の各カラムは文字列フィールド値に変換され、テーブルまたは結果セット内のカラムの順序で、出力レコードに追加されます。各出力フィールドのフォーマットは、元のカラム・データ型と、さまざまなUNLOADオプション設定に依存します。

出力ファイルのレイアウトは、次のUNLOADオプションで制御します。

- **APPEND ON** ——ファイルが既に存在する場合、その末尾に出力レコードを追加することを指定する。ファイルがない場合は新しいファイルが作成される。デフォルトはAPPEND OFFで、既存のファイルは上書きされる。
- **DELIMITED BY** ——出力フィールドのデリミタを変更できる。たとえば、「DELIMITED BY '¥x09'」は、出力ファイルがタブで区切られることを指定する。「DELIMITED BY ''」と指定すると、フィールド・デリミタをまとめて削除できる。デフォルトは「DELIMITED BY ','」。
- **ESCAPE CHARACTER** ——出力ファイル内の文字列リテラルの中でエスケープ文字として使用する単一の文字を指定できる（「ESCAPE CHARACTER '!'」など）。デフォルトは「ESCAPE CHARACTER '¥'」。このオプションは、出力データの生成方法に影響するもので、出力ファイル仕様内でのエスケープ文字の扱いには関係しない。
- **ESCAPES OFF** ——出力文字列リテラル内でのエスケープ文字の生成をオフにできる。デフォルトはESCAPES ONで、エスケープ文字が生成される。このオプションも、ファイル内のデータに関するものであり、UNLOAD文内のファイル仕様には関係しない。
- **FORMAT BCP** ——特別なAdaptive Server Enterpriseバルク・コピー・プログラム（bcp.exe）のファイル・フォーマットを出力ファイルで使用することを指定する。デフォルトはFORMAT ASCIIで、通常のテキスト・ファイルを表す。本書では、FORMAT BCPの詳細については説明しない。
- **HEXADECIMAL OFF** ——バイナリ文字列データに対して、「0xnn」というスタイルの引用符の付いていないバイナリ文字列リテラルの生成をオフにする。デフォルトはHEXADECIMAL ONであり、0xnnスタイルの出力値を生成する。
- **ORDER OFF** ——UNLOAD TABLEと共に使用すると、出力データのソートを抑制できる。デフォルトはORDER ONであり、クラスタード・インデックスがある場合はクラスタード・インデックスによって、または、クラスタード・インデックスがなくてプライマリ・キーがある場合はプライマリ・キーによって、出力データがソートされる。クラスタード・インデックスもプライマリ・キーもない場合は、ORDER ONは機能しない。このソートは本来、LOAD TABLEによるファイルの再ロードの処理を高速化することを目的とする。ORDERオプションはUNLOAD SELECT文に適用されないが、UNLOAD SELECT文では代わりにORDER BY句を使用できる。
- **QUOTES OFF** ——すべての文字列データを一重引用符で囲まずに書き込むことを指定する。埋め込まれている一重引用符を2つ重ねることもない。デフォルトの動作はQUOTES ONであり、文字列データを引用符で囲まれた文字列リテラルとして書き込む。

Tip

独自のUNLOAD文を記述する場合は、UNLOAD TABLE文にこだわらずに、UNLOAD SELECTとORDER BYを使用してください。UNLOAD SELECTの方が柔軟性に優れており、UNLOAD TABLEと同じ操作を実行するのにコーディングが難しいということはないため、UNLOAD SELECTを覚えて損はありません。唯一の例外は、インデックスのソート順で大量のテーブルをファイルにダンプする場合で、これはORDER BY句を記述せずに行うことはできません。この場合は、ORDER ONデフォルトによって、UNLOAD TABLEを使用する方が簡単です。

次の例は、異なるデータ型の値に対するさまざまなUNLOADオプションの効果を示しています。5つの異なるオプション・セットを使用して、5つの異なるテキスト・ファイルに同じデータが書き込んでいます。テーブル内の各ローでは、col_2値とcol_3値が実際には同じであることに注意してください。INSERT文の入力フォーマットがUNLOAD文の出力フォーマットに関係しないことを示すため、INSERT文のVALUE句内では、異なるフォーマットを使用しています。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL,
   col_2         VARCHAR ( 100 ) NULL,
   col_3         BINARY ( 100 ) NULL,
   col_4         DECIMAL ( 11, 2 ) NULL,
   col_5         DATE NULL,
   col_6         INTEGER NOT NULL,
   PRIMARY KEY ( key_1 ) );
INSERT t1 VALUES (
   1, 'Fred''s Here',         'Fred''s Here',     12.34, '2003-09-30', 888 );
INSERT t1 VALUES (
   2, 0x74776f0d0a6c696e6573, 'two¥x0d¥x0alines', 67.89, '2003-09-30', 999 );
COMMIT;
UNLOAD SELECT * FROM t1 ORDER BY key_1
   TO 't1_b1.txt';
UNLOAD SELECT * FROM t1 ORDER BY key_1
   TO 't1_b2.txt' ESCAPES OFF;
UNLOAD SELECT * FROM t1 ORDER BY key_1
   TO 't1_b3.txt' ESCAPES OFF QUOTES OFF;
UNLOAD SELECT * FROM t1 ORDER BY key_1
   TO 't1_b4.txt' HEXADECIMAL OFF ESCAPES OFF QUOTES OFF;
UNLOAD SELECT * FROM t1 ORDER BY key_1
   TO 't1_b5.txt' DELIMITED BY '' HEXADECIMAL OFF ESCAPES OFF QUOTES OFF;
```

Tip

出力の順序が重要な場合は、ORDER BYとUNLOAD SELECTを使用します。SQL Anywhereのテーブルには、クラスタード・インデックスがある場合でさえも、確実で固定的なローの順序はありません。

上の t1_b1.txt ファイルへの書き込みの例では、オプション設定をすべてデフォルトのままにしています。これは、LOAD TABLE を使用してデータベースにロードし直すことができるファイルを作成する場合に最も適しています。ファイルは次のようになります。VARCHAR 値を囲む引用符、一重引用符の二重化、エスケープ文字、0xnn スタイルの BINARY 値、カンマのフィールド・デリミタに注意してください。

```
1,'Fred''s Here',0x4672656427732048657265,12.34,2003-09-30,888
2,'two¥x0d¥x0alines',0x74776f0d0a6c696e6573,67.89,2003-09-30,999
```

t1_b2.txt ファイルは、「ESCAPES OFF」で書き込んでいます。メモ帳やワードパッドでファイルを表示すると次のようになります。VARCHAR カラムに埋め込まれている復帰と改行のペア '¥x0d¥x0a' がエスケープ文字シーケンスに変換されず、出力ファイルにそのまま書き込まれて、本当に改行されていることに注意してください。

```
1,'Fred''s Here',0x4672656427732048657265,12.34,2003-09-30,888
2,'two
lines',0x74776f0d0a6c696e6573,67.89,2003-09-30,999
```

t1_b3.txt ファイルは、「ESCAPES OFF QUOTES OFF」で書き込んでいます。ファイルは次のようになります。先行と後続の一重引用符はなく、埋め込まれている引用符も二重化されていません。

```
1,Fred's Here,0x4672656427732048657265,12.34,2003-09-30,888
2,two
lines,0x74776f0d0a6c696e6573,67.89,2003-09-30,999
```

t1_b4.txt ファイルは、「HEXADECIMAL OFF ESCAPES OFF QUOTES OFF」で書き込んでいます。大きな違いは、HEXADECIMAL OFF 設定によって、BINARY 値が 0xnn スタイルで出力されないことです。BINARY 値は VARCHAR 値のように表示され、埋め込まれている復帰と改行のペアは出力ファイルにそのまま送られます。

```
1,Fred's Here,Fred's Here,12.34,2003-09-30,888
2,two
lines,two
lines,67.89,2003-09-30,999
```

t1_b5.txt ファイルは、「DELIMITED BY " HEXADECIMAL OFF ESCAPES OFF QUOTES OFF」で書き込んでいます。これは、HTML や XML を含むテキストを書き込む場合など、テキストを「そのまま」書き込む場合に最も適しています。カラム値を文字列に変換した後で、余分なフォーマットを行いません。「DELIMITED BY "」によって、フィールド・デリミタが効率的に削除されます。

```
1Fred's HereFred's Here12.342003-09-30888
2two
linestwo
lines67.892003-09-30999
```

UNLOAD 文は、各値が文字列に変換されて出力される、という点に関して、STRING 関数と同じ働きをします。HEXADECIMAL ON や ESCAPES ON など、さまざまなオプションを指定することで、変換が完了してからさらにフォーマットが行われますが、すべてのオプションがオフの場合、UNLOAD の結果と STRING の結果は同じです。たとえば、次の SELECT を実行すると、上の t1_b5.txt ファイルに書き込まれたデータとまったく同じ文字列値が返されます。

```
SELECT STRING (
          key_1,
          col_2,
          col_3,
          col_4,
          col_5,
          col_6 )
  FROM t1
 ORDER BY key_1;
```

「3.14　集合関数呼び出し」では、STRING 関数と LIST 関数を使用して、HTML ドキュメント全体の文字列を生成する例を示しました。次に、同じ例を再び示します。ここでは、UNLOAD SELECT を使用してドキュメントをファイルに書き込みます。

```
UNLOAD
SELECT STRING (
          '<HTML><BODY><OL>¥x0d¥x0a',
          '   <LI>',
          LIST ( DISTINCT state,
                 '</LI>¥x0d¥x0a   <LI>'
                 ORDER BY state ),
          '</LI>¥x0d¥x0a',
          '</OL></BODY></HTML>' ) AS states_page
  FROM employee
 WHERE dept_id = 100
TO 'c:¥¥temp¥¥states_page.html' ESCAPES OFF QUOTES OFF;
```

図 3-2 は、Internet Explorer に表示される c:¥temp¥states_page.html ファイルを示しています。書き込まれる BINARY 値がないので、HEXADECIMAL OFF オプションは不要です。また、出力レコード内のフィールドは 1 つだけなので、「DELIMITED BY "」も不要です。

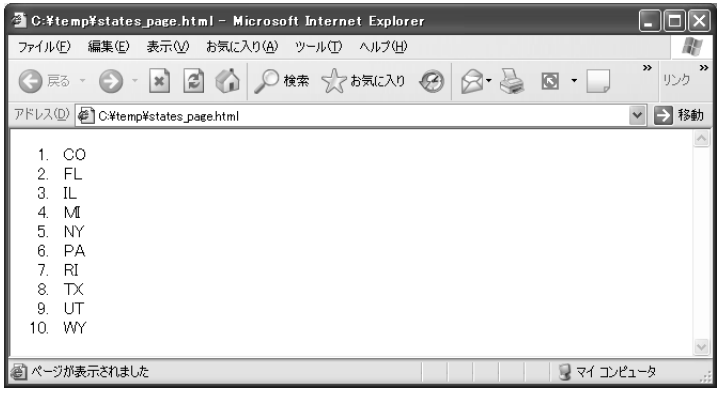


図3-2　UNLOAD SELECTで書き込まれたHTML

Section

3.26 ISQL OUTPUT

Interactive SQLユーティリティ（dbisql.exeまたはISQL）は、UNLOAD SELECTと類似の機能を実行するISQL OUTPUT文をサポートしますが、両者は多くの点で大きく異なります。

```
<isql_output>   ::= OUTPUT TO <output_file> { <output_option> }
<output_file>   ::= クライアントを基準とする文字列リテラルのファイル指定
                  | クライアントを基準とする二重引用符によるファイル指定
                  | クライアントを基準とする引用符なしのファイル指定
<output_option> ::= APPEND ( ON | OFF )              -- デフォルトは上書き
                  | COLUMN WIDTHS "(" <output_column_width_list> ")"
                  | DELIMITED BY <output_delimiter>            -- デフォルトは ','
                  | ESCAPE CHARACTER <output_escape_character> -- デフォルトは '¥'
                  | FORMAT <output_format>           -- デフォルトは ASCII
                  | HEXADECIMAL <hexadecimal_option> -- デフォルトは ON
                  | QUOTE <output_quote> [ ALL ]     -- デフォルトは文字列を ' 引用符 ' で囲む
                                                     -- QUOTE '' は引用符なし
                  | VERBOSE                          -- デフォルト・データのみ
<output_column_width_list> ::= <output_column_width> { "," <output_column_width> }
<output_column_width>      ::= カラム幅を指定する整数リテラル（FIXEDフォーマット用）
<output_delimiter>         ::= カラム・デリミタ文字を含む文字列リテラル
<output_escape_character>  ::= 長さが1文字のみの文字列リテラル
<output_format>            ::= <output_format_name>を含む文字列リテラル
                             | 二重引用符で囲んだ<output_format_name>
                             | 引用符で囲まない<output_format_name>
<output_format_name>       ::= ASCII    -- デフォルト
                             | DBASEII
```

```
                              | DBASEIII
                              | EXCEL
                              | FIXED
                              | FOXPRO
                              | HTML
                              | LOTUS
                              | SQL
                              | XML
<hexadecimal_option> ::= ON   -- デフォルト（バイナリ文字列は「0xnn..」）
                       | OFF  -- バイナリを文字として扱う。エスケープ文字あり
                       | ASIS -- バイナリを文字として扱う。エスケープ文字なし
<output_quote>       ::= 文字列リテラルを引用符で囲んだ文字列リテラル
```

OUTPUT コマンドは、ISQL で結果セットを使用できる場合のみ、ISQL コマンドとして動作します。つまり、次の例に示すように、OUTPUT は通常、SELECT と一緒に実行します。

```
SELECT *
  FROM product
 WHERE name = 'Sweatshirt'
 ORDER BY id;
OUTPUT TO 'product.txt';
```

これらの文を ASADEMO データベースに対して実行した場合の product.txt ファイルは次のようになります。

```
600,'Sweatshirt','Hooded Sweatshirt','Large','Green',39,24.00
601,'Sweatshirt','Zipped Sweatshirt','Large','Blue',32,24.00
```

Note

OUTPUT や INPUT などの ISQL コマンドは、BEGIN ブロックの内部でネストすることはできません。しかし、SQL コマンド・ファイルの内部に配置して、他のコマンドと一緒に実行できます。言い換えると、バッチ・モードでは実行できますが、BEGIN ブロックの内部に配置することはできません。BEGIN ブロックは、データベース・エンジンに送られて実行されるからです。データベース・エンジンは SQL コマンドのみを理解し、ISQL コマンドは理解しません。

OUTPUT 文におけるファイル仕様のコーディング・ルールは、INPUT 文におけるファイル仕様のルールと同じです。詳細については、「2.4　ISQL INPUT」を参照してください。

OUTPUT が UNLOAD SELECT より優れているのは、その多様な出力フォーマットです。DBASEII、DBASEIII、EXCEL、FOXPRO、LOTUS、SQL などの従来のフォーマットだけでなく、デフォルトの ASCII に加えて、FIXED、HTML、XML などの興味深いフォーマットも備えています。

次に、ASADEMO データベースから 2 つのローを取得する SELECT を示します。その後には、3 つ

の異なるファイルに書き込みを行うOUTPUT文が続きます。

```
SELECT id,
       name,
       quantity,
       unit_price
  FROM product
 WHERE name = 'Sweatshirt'
 ORDER BY id;

OUTPUT TO 'c:¥¥temp¥¥product.dta' COLUMN WIDTHS ( 5, 15, 5, 7 ) FORMAT FIXED;
OUTPUT TO 'c:¥¥temp¥¥product.html' FORMAT HTML;
OUTPUT TO 'c:¥¥temp¥¥product.xml' FORMAT XML;
```

FIXEDフォーマットで出力したproduct.dtaファイルは次のようになります。各フィールドは固定長で、引用符とデリミタはなく、文字列への変換以外のフォーマットは行われていません。INTEGER値のidとquantity（600、39など）は右揃えされていますが、NUMERIC値のunit_price（24.00など）は左揃えされていることに注意してください。

```
600Sweatshirt        3924.00
601Sweatshirt        3224.00
```

図3-3は、FORMAT HTMLで出力したproduct.htmlファイルをInternet Explorerで表示したものです。データは、見出しと枠線を使ったHTMLの<table>を使用して表示しています。

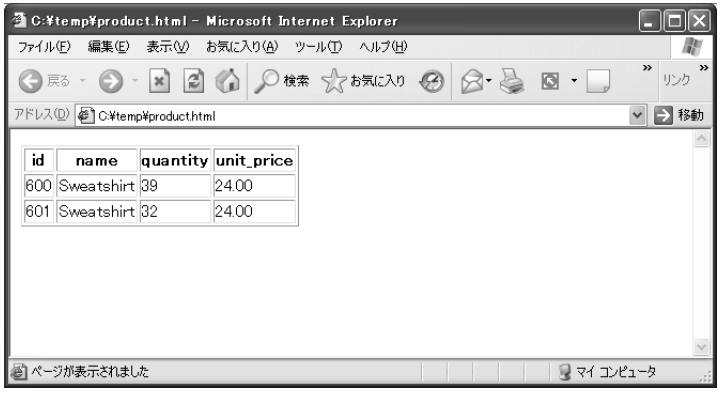


図3-3　OUTPUT FORMAT HTMLで書き込んだファイル

図3-4は、FORMAT XMLで出力したproduct.xmlファイルをInternet Explorerで表示したものです。データは<resultset>、<row>、<column>の各要素を使用して構造化され、カラム名は<column>タグの属性として記述され、ファイルにはDTDが埋め込まれています。

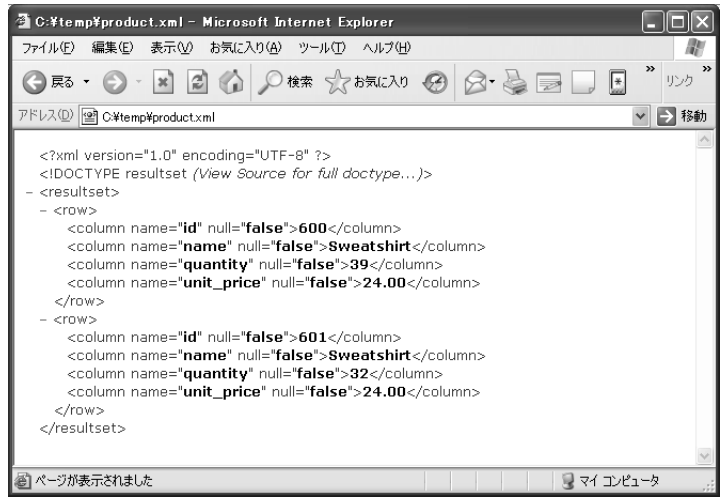


図3-4　OUTPUT FORMAT XMLで書き込んだファイル

OUTPUTの動作は、各種のオプションで制御します。

- **APPEND** ――ファイルが既に存在する場合、その末尾に出力レコードを追加することを指定する。ファイルがない場合は新しいファイルが作成される。このオプションは、ASCII、FIXED、SQLの各フォーマットにのみ適用される。デフォルトでは、既存のファイルは上書きされる。
- **COLUMN WIDTHS** ―― FIXEDフォーマット出力のカラム幅を表す数値リストを指定する。FIXEDフォーマットでは、COLUMN WIDTHSの明示的なリストを常に使用する必要がある。本書では、COLUMN WIDTHSが指定されない場合、またはその値が少なすぎる場合に、デフォルトの幅がどのように計算されるかについて説明しない。
- **DELIMITED BY** ―― ASCIIフォーマット出力の出力フィールド・デリミタを変更できる。たとえば、「DELIMITED BY '¥x09'」は、出力ファイルがタブで区切られることを指定する。「DELIMITED BY ''」を使用すると、フィールド・デリミタをまとめて削除できる。デフォルトは「DELIMITED BY ','」。
- **ESCAPE CHARACTER** ―― ASCIIフォーマット出力内の文字列リテラルの中でエスケープ文字として使用する単一の文字を指定できる（「ESCAPE CHARACTER '!'」など）。デフォルトは「ESCAPE CHARACTER '¥'」。ESCAPES OFFオプションはないが、HEXADECIMAL ASISで同じ処理が行われる。
- **FORMAT FIXED** ――各出力フィールドが固定長で、文字列への変換以外のフォーマットが行われないことを指定できる。つまり、フィールド・デリミタは挿入されず、エスケープ文字は生成されず、0xnnスタイルのフォーマットは行われない。デフォルトはFORMAT ASCII。上の例にFORMAT HTMLとXMLが示されているが、ここでは詳しく説明しない。FORMAT

DBASEII、DBASEIII、EXCEL、FOXPRO、LOTUS、SQLも使用できる。

- **HEXADECIMAL OFF** ——、ASCIIフォーマット出力の中でエスケープ文字を使用してBINARY文字列をフォーマットできる。たとえば、「SELECT CAST ('1¥¥2¥x0d¥x0a3' AS BINARY)」は、'1¥¥2¥x0d¥x0a3'として出力される。デフォルトはHEXADECIMAL ONであり、0xnnスタイルを使用する。たとえば、「SELECT CAST ('1¥¥2¥x0d¥x0a3' AS BINARY)」は、0x315c320d0a33として出力される。

  HEXADECIMAL ASISを使用すると、ASCIIフォーマット出力におけるBINARY文字列のすべての再フォーマットをオフにできる。たとえば、「SELECT CAST ('1¥¥2' AS BINARY)」は、'1¥2'として出力される。SELECT内の文字列リテラル'1¥¥2'には、1つの円記号として格納されるエスケープ文字列シーケンス¥¥が含まれている。CAST関数はこの文字列をBINARYに変換し、HEXADECIMAL ASISはOUTPUTコマンドに対してエスケープ文字を元に戻さないことを指定する。

  HEXADECIMAL ASISは、ASCIIフォーマット出力内のVARCHAR文字列にも影響する。これは、UNLOAD文におけるESCAPES OFFのように、文字列に対してエスケープ文字の生成をオフにする。たとえば、「SELECT '1¥¥2'」は'1¥2'として出力され、「SELECT '1¥x0d¥x0a2'」は1と2の間が実際に改行されて'12'として出力される。
- **QUOTE** —— ASCIIフォーマット出力の中で出力フィールドが「引用符」で囲まれる方法を変更できる。引用符として使用する文字が変更可能であり、ALLキーワードを使用すると、すべての出力フィールドが引用符で囲まれる。たとえば、「QUOTE ""」と指定すると、引用符が二重引用符に変更され、「QUOTE "" ALL」と指定すると、すべての出力フィールドが一重引用符で囲まれる。文字列リテラル「""」は一重引用符を表す。デフォルトは「QUOTE ""」であり、文字列フィールドが一重引用符で囲まれる。HEXADECIMALオプションも、引用符で囲むものと囲まないものに影響する。

  「QUOTE "」を使用すると、引用符をすべてオフにできる。ここで、「"」は空の文字列を表す。この指定によって、文字列値に埋め込まれている引用符の二重化もオフになるため、「'Fred"s Here'」はファイル内では「Fred's Here」として表示される。
- **VERBOSE** —— ASCII、FIXED、SQLの各フォーマットが使用される場合に、実際のデータと共に追加の情報が出力に含まれる。デフォルトでは、データのみが含まれる。

Tip

BINARYデータには注意してください。出力が本当に求める出力であるかどうか確認してください。FIXEDフォーマットの場合は特に、CASTを呼び出してBINARYデータをVARCHARに変換した方がよいでしょう。

Tip

COLUMN WIDTHSオプションに含まれる幅の個数と、OUTPUTされる結果セット内のカラム数を必

ず同じにしてください。多すぎたり少なすぎたりする場合、警告やエラー・メッセージは表示されず、OUTPUT は左から右に向かって幅を適用したり、余分な幅を無視したり、不足しているエントリのデフォルト幅を計算したりします。

次の例では、異なるオプション値が指定されている場合に、FORMAT ASCII 出力の中で各種のデータ型がどのように表示されるかを示します。最初の OUTPUT 文ではすべてのデフォルトを使用するのに対して、次の OUTPUT 文では、文字列への変換以外のすべてのフォーマットをオフにしています。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL,
   col_2         VARCHAR ( 100 ) NULL,
   col_3         BINARY ( 100 ) NULL,
   col_4         DECIMAL ( 11, 2 ) NULL,
   col_5         DATE NULL,
   col_6         INTEGER NOT NULL,
   PRIMARY KEY ( key_1 ) );

INSERT t1 VALUES (
   1, 'Fred''s Here',         'Fred''s Here',     12.34, '2003-09-30', 888 );
INSERT t1 VALUES (
   2, 0x74776f0d0a6c696e6573, 'two¥x0d¥x0alines', 67.89, '2003-09-30', 999 );
COMMIT;

SELECT * FROM t1 ORDER BY key_1;
OUTPUT TO 't1_c1.txt';
OUTPUT TO 't1_c2.txt' DELIMITED BY '' HEXADECIMAL ASIS QUOTE '';
```

図 3-5 は、ISQL に表示される結果セットを示しています。

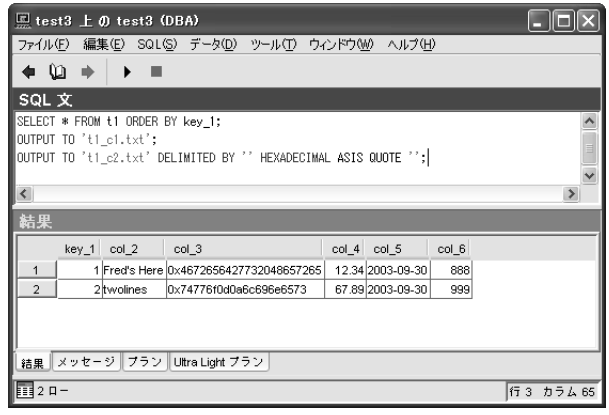


図 3-5　ISQL における SELECT と OUTPUT

最初の t1_c1.txt ファイルは次のようになります。フィールド・デリミタが含まれ、文字列は引用符で囲まれ、エスケープ文字が使用され、埋め込みの引用符は二重化されます。さらに、BINARY データは 0xnn スタイルでフォーマットされます。

```
1,'Fred''s Here',0x467265642773204865726573,12.34,'2003-09-30',888
2,'two¥x0d¥x0alines',0x74776f0d0a6c696e6573,67.89,'2003-09-30',999
```

2 番目の t1_c2.txt ファイルは次のようになります。デリミタも引用符もエスケープ文字もなく、実際の改行を含めて、データがそのまま表示されます。

```
1Fred's HereFred's Here12.342003-09-30888
2two
linestwo
lines67.892003-09-30999
```

Tip

フォーマットを行わずにあるがままに出力する場合、文字列に変換された後の実際のデータに応じてフィールド幅が変わるので、「FORMAT ASCII DELIMITED BY " HEXADECIMAL ASISQUOTE "」の方が FIXED フォーマットより適しています。言い換えると、ASCII フォーマットは FIXED フォーマットより STRING 関数に似た動作をします。

Section

3.27 この章のまとめ

この章では、select のさまざまな句が連携してデータを取得し、最終的な結果セットを生成する方法について説明しました。論理的または架空のステップのリストを示すことで、各句の関係を明確にし、「WHERE 句内で NUMBER(*) を呼び出せないのはなぜか?」などの質問に対する答えを示しました（答えは、「WHERE 句は ORDER BY の前に実行されるが、NUMBER(*) を呼び出すには、その前にローをソートする必要があるため」です）。

この章では、式、ブール式、UNION 演算子、EXCEPT 演算子、INTERSECT 演算子、CREATE VIEW 文、WITH 句、再帰ユニオン、UNLOAD 文、ISQL OUTPUT 文についても説明しました。

次の章では、データベースのライフサイクルの 4 番目のステップであるローの更新について説明します。

Chapter

4

更新

Section

4.1 概要

この章では、はじめに単一のテーブルに含まれる単一のローのデータを変更する単純なUPDATE文について説明し、続いて単一のテーブルに含まれる複数のローのUPDATEについて説明します。その後、4.4項で複数テーブルの複数のローが、より複雑なUPDATE文によってどのように処理されるかを、理論的な観点から解説します。この解説は「1つのUPDATEが複数のテーブルをどのようにして変更するのか」、また「1つのUPDATEがどのようにして同じローを2回以上変更するのか」といった疑問に答えるものです。

4.4.1項では、集合指向型のUPDATE文の構文全体について、例を用いながら説明します。4.5項ではカーソル指向型のUPDATEを説明してこの章を締めくくります。カーソル指向型のUPDATEは単一のローを処理するのが一般的ですが、実際には1回の実行で複数のローを変更することもできます。

Section

4.2 単一のローの更新

最も単純で、最も一般的なUPDATE式は、1つのテーブルに含まれる1つのローのカラム値を変更する使い方のものです。

```
<typical_single_row_update> ::= UPDATE [ <owner_name> "." ] <table_name>
                                    <simple_set_clause>
                                    <single_row_where_clause>
<simple_set_clause>         ::= SET { <simple_set_item> "," }
                                       <simple_set_item>
<simple_set_item>           ::= <column_name> "=" <expression>
<single_row_where_clause>   ::= 1つのローにだけ一致する <where_clause>
<owner_name>                ::= <identifier>
<table_name>                ::= <identifier>
<column_name>               ::= <identifier>
<identifier>                ::=「第1章　作成」の <identifier> を参照
<expression>                ::=「第3章　選択」の <expression> を参照
<where_clause>              ::=「第3章　選択」の <where_clause> を参照
```

通常、単一ローのUPDATEには、1つのテーブル名と、SETキーワードに続けて1つ以上のカラムの代入を指定し、さらに、テーブル内の単一のローに一致するWHERE句を指定します。次に示すのは、ASADEMOデータベースを使った更新の例です。プライマリ・キーIDが101である企業のローの

address カラムの値を、'114 Pioneer Avenue' から '114 PIONEER AV' に変更します。

```
UPDATE customer
   SET address = '114 PIONEER AV'
 WHERE id = 101;
```

SQL Anywhere Studio 9 に付属の ASADEMO データベースに含まれる、customer テーブルの詳細については、「3.6　マルチテーブル・ジョイン」を参照してください。

WHERE 句で 1 つのローが一致し、SET 句に指定された新しいカラム値のうち少なくとも 1 つが現在の値と異なる場合、UPDATE は次のように処理されます。このテーブルに関連付けられたすべての BEFORE UPDATE トリガが起動し、ローのロックが取得されます。ローが更新されて、UPDATE がトランザクション・ログに記録されます。すべての AFTER UPDATE トリガが起動し、完全に正常な状態であることを示すために、SQLSTATE という特殊なリテラルが '00000' にセットされます。トリガの詳細については、「8.11　CREATE TRIGGER」を参照してください。また、ロックの詳細については、「9.6　ロック」を参照してください。

WHERE 句に 1 つのローが一致したが、新しいカラム値がすべて現在の値と同じである場合、UPDATE は次のように処理されます。このテーブルに関連付けられたすべての BEFORE UPDATE トリガが起動し、ローのロックが取得されます。ただし、何も更新されず、トランザクション・ログに何も記録されません。AFTER UPDATE トリガはすべて起動せず、特殊なリテラル SQLSTATE が '00000' にセットされます。

WHERE 句がどのローとも一致しない場合もエラーではありませんが、SQLSTATE は「ローが見つかりません」を示す値である '02000' にセットされます。この場合、トリガは何も起動せず、ローのロックも取得されません。更新は実行されず、トランザクション・ログには何も記録されません。

Note

複数ローの更新の場合も動作は同じです。これらの動作が 1 つ 1 つのローに対して実行されます。ただし、UPDATE 文が実行されるたびに SQLSTATE が 1 度設定される点と、UPDATE 文が実行されるたびに AFTER STATEMENT トリガが 1 度起動される点の 2 つが異なります。

Note

本書では、ISOLATION_LEVEL オプションの値がデフォルトの 0 に設定されていることを前提としています。これは、パフォーマンスと同時実行性を最高レベルに設定する値です。0 よりも大きい値に設定された場合、ロックの動作が異なる可能性があります。たとえば、ISOLATION_LEVEL が 3 に設定されていると、先ほど述べたような「WHERE 句がどのローとも一致しない場合、ローのロックが取得されない」という保証はありません。独立性レベルの詳細については、「第 9 章　保護」の「9.7　ブロックと独立性レベル」を参照してください。

先ほどのUPDATEを修正し、現在のaddressの値が'114 Pioneer Avenue'から変わっていない場合にだけローを変更する例を次に示します。現在の値が他の処理によって変更されている場合、WHERE句はどのローとも一致せず、SQLSTATEは'02000'にセットされます。

```
UPDATE customer
   SET address = '114 PIONEER AV'
 WHERE id = 101
   AND address = '114 Pioneer Avenue';
```

これは、楽観的同時実行制御の簡単な例です。アプリケーションは、selectからupdateまでの間に不要なロー・ロックを獲得しないようにする代わりに、ローを更新する前に、そのローが他のプロセスによって変更されていないか確認します。

companyテーブルの同じローを更新する例をもう1つ紹介します。この例では、顧客の連絡先のファースト・ネームとラスト・ネームの、2つのカラムが変更されます。また、WHERE句にはプライマリ・キーではなく、企業名を指定しています。

```
UPDATE customer
   SET fname = 'Fred',
       lname = 'Jones'
WHERE company_name = 'The Power Group';
```

Tip

単一のローのUPDATE文では、誤って複数のロー（もしくはテーブル全体）を誤って更新しないように注意してください。WHERE句が必ずプライマリ・キーまたはユニークなインデックスを参照していること、また、プライマリ・キーやインデックスが２つ以上のカラムから成る場合は、WHERE句にそれらすべてのカラムの値を指定するようにしてください。

Section

4.3 複数のローの更新

UPDATEは、構文に関して言えば、単一のローを対象とした場合も複数のローを対象とした場合もまったく同じです。実際、単一ローのUPDATEは、通常、WHERE句の制限を緩めるか、WHERE句の指定を削除するだけで複数ローのUPDATEになります。

```
<typical_multi_row_update> ::= UPDATE [ <owner_name> "." ] <table_name>
                                  <simple_set_clause>
                                  [ <multi_row_where_clause> ]
<multi_row_where_clause>   ::= 0または1つ以上のローに一致する<where_clause>
```

WHERE 句を指定しない場合は、テーブル内のすべてのローが更新されます。次に示す例は、すべての従業員の給与を 5% 昇給します。SET の代入文の右側にあるように、更新されるローの現在のカラム値を式の中で参照できます。この例の「SET salary = salary * 1.05」は、「各ローの salary カラムの新しい値として、元の値に 1.05 をかけた値を設定する」という意味になります。

```
UPDATE employee
   SET salary = salary * 1.05;
```

WHERE 句を指定し、それが 2 つ以上のローに一致する場合は、一致するすべてのローに対して UPDATE が実行されます。次の例は、部署が 100 であるすべての従業員に退社日を設定します。要するに、R&D 部門全体が解雇されてしまった場合の例です。

```
UPDATE employee
   SET termination_date = CURRENT DATE
 WHERE dept_id = 100;
```

マルチユーザ環境の大規模なテーブルで複数のローの UPDATE 文を実行する場合は注意が必要です。複数のローの集合指向型 UPDATE は非常に強力ですが、処理の対象となるすべてのローの排他ロックを取得します。また、それらのロックは、少なくとも UPDATE の実行中は解放されません。たとえば、100 万ローを変更する UPDATE の処理には何分間もかかることがあるため、この処理が取得するロックによって、他のユーザが作業を行えなくなる可能性があります。また、処理の途中で別のユーザの保持するロックによって自分がブロックされる場合もあります。これが起きると状況はさらに悪くなります。このようなケースでは、FETCH ループ内でカーソル指向型の UPDATE を使用し、ときどき COMMIT を実行してロックを解放するという方法が適切です。カーソル FETCH ループの詳細については、「第 6 章　フェッチ」を参照してください。

Section

4.4 集合型UPDATEの実行ロジック

集合指向型UPDATEの構文は、ローの範囲を指定する句（FIRSTまたはTOP）、「3.3　FROM句」で説明したようなテーブル仕様、およびWHERE句とORDER BY句で構成されます。つまり、UPDATE文には複数のテーブルのジョインを指定できます。さらに言えば、複数のテーブルに含まれるカラムをSET句に指定することも可能です。

```
<set_update> ::= UPDATE
                    [ <row_range> ]
                    <table_specification>
                    <set_clause>
                    [ <where_clause> ]
                    [ <order_by_clause> ]
```

以降では、UPDATE句について、例を用いて詳しく説明します。この項では、「複数のテーブルのUPDATEでは実際に何が行われているのだろう」という疑問について取り上げます。この疑問は重要です。なぜなら、実際には単一のUPDATEが同じローを2度以上更新することもあれば、複数のテーブルに含まれる複数のローを変更することもあるからです。

UPDATEの動作について説明するために、要求された機能を実行するために行われる一連の単純なステップを紹介します。「3.2　SELECTの論理的実行」で説明したステップと同様、これらも理論上の（つまり、仮想的な）ステップです。実際に行われるステップではなく、「行われ得る」ステップであることに注意してください。

以下に、UPDATEが処理される際の各ステップを、理論的な観点から簡単に説明します。

❶ UPDATEに対応するSELECT文の作成が開始されます。TOP句またはFIRST句をselectリストにコピーし、UPDATEのテーブル式をSELECT FROM句にコピーし、UPDATEのWHERE句とORDER BY句もSELECTにコピーします。
❷ SET句で更新するベース・テーブルを決定します。
❸ ベース・テーブルのすべてのカラムとNUMBER(*)の呼び出しをselectリストに入れます。NUMBER(*)の呼び出しは、いずれかのSET式でNUMBER(*)が使用されている場合に必要です。
❹ SELECTを実行して、候補結果セットを生成します。
❺ ベース・テーブルの実際のカラムのうち、候補結果セットに存在するカラムに対してSET句を適用します。

この項の残りの部分では、2つの簡単なテーブルに含まれる4つのローの例を使いながら、これらのステップをさらに詳しく説明していきます。

```
CREATE TABLE t1 (
   key_1         UNSIGNED INTEGER NOT NULL PRIMARY KEY,
   non_key_1     INTEGER NOT NULL );

CREATE TABLE t2 (
   key_1         UNSIGNED INTEGER NOT NULL PRIMARY KEY,
   non_key_1     INTEGER NOT NULL );

INSERT t1 VALUES ( 1,   0 );
INSERT t1 VALUES ( 2,   0 );
INSERT t2 VALUES ( 1, 100 );
INSERT t2 VALUES ( 2, 100 );
```

UPDATEを実行する前のテーブルの内容は以下のとおりです。

```
t1.key_1  t1.non_key_1
========  ============
1         0
2         0

t2.key_1  t2.non_key_1
========  ============
1         100
2         100
```

以下に示すUPDATEの例は、各ステップが実行される様子を示すためだけに例として作成したものです。この例では、テーブルt1とそれ自身とのCROSS JOINを実行し、その後にテーブルt2とのCROSS JOINを実行します。SET句は、3.20項で説明したNUMBER(*)関数を参照します。WHERE句は、更新対象のローを、t1.non_key_1が0であるローに制限します。また、ORDER句はNUMBER(*)が返す値を順序どおりに並べます。

```
UPDATE t1
       CROSS JOIN t1 AS x
       CROSS JOIN t2
   SET t1.non_key_1 = NUMBER(*),
       t2.non_key_1 = NUMBER(*)
 WHERE t1.non_key_1 = 0
 ORDER BY t1.key_1 DESC,
       t2.key_1 DESC;
```

UPDATE実行後のテーブルの内容は以下のとおりです。2つのテーブルに含まれるすべてのローが更新されています。また、各テーブルにはローが2つしかないにもかかわらず、NUMBER(*)関数は、4、6、8などの明らかに大きい値を返しています。

```
t1.key_1  t1.non_key_1
========  ============
1         8
2         4

t2.key_1  t2.non_key_1
========  ============
1         8
2         6
```

ここからはUPDATEの処理方法について、理論上の各ステップをより詳しく説明します。

ステップ1：SELECT文の作成が開始され、TOPまたはFIRST、WHERE、ORDER BYの各句がSELECTにコピーされます。また、UPDATEのテーブル式もSELECT FROM句にコピーされます。このステップを実行した後の、未完成のSELECTは次のようになります。

```
SELECT ...
  FROM t1
       CROSS JOIN t1 AS x
       CROSS JOIN t2
 WHERE t1.non_key_1 = 0
 ORDER BY t1.key_1 DESC,
       t2.key_1 DESC;
```

ステップ2：SET句を調べることによって、更新するベース・テーブルを決定します。ここで言う「ベース・テーブル」とは、データベースの実際のテーブルを指します。これはグローバルな永続テーブルであったり、ローカル・テンポラリ・テーブル、グローバル・テンポラリ・テーブル、またはプロキシ・テーブルであったりします。ただし、ビュー、抽出テーブル、プロシージャ・コールではありません。ビューや抽出テーブルに含まれるカラムを指定するようにSET句を定義することもできますが、その場合は、実際に更新するベース・テーブルを、このステップの中で決定する必要があります。すなわち、ビューや抽出テーブルが実際に更新されることはなく、実テーブルだけが更新されます。

Note

ビューをとおしてローを更新できるのは、そのビューが更新可能な場合だけです。つまり、ビューがDISTINCT、GROUP BY、UNION、EXCEPT、INTERSECT、または集合関数の参照を使用している場合は更新できません。ビューの詳細については、「3.23　CREATE VIEW」を参照してください。

この例では、t1とt2の2つのベース・テーブルがあります。この情報は次のステップで必要になります。また、t1はJOINで2度指定されていますが、これは問題ありません。

ステップ3：ベース・テーブルのすべてのカラムとNUMBER(*)の呼び出しを、selectリストに入れます。完成したSELECTは次のようになります。各ベース・テーブルに「.*」の表記を使用している点に注目してください。「SELECT *」と記述しただけでは、これと同じ結果になりません。

```
SELECT t1.*,
       t2.*,
       NUMBER(*)
  FROM t1
       CROSS JOIN t1 AS x
       CROSS JOIN t2
 WHERE t1.non_key_1 = 0
 ORDER BY t1.key_1 DESC,
       t2.key_1 DESC;
```

また、JOINの部分には変化がありません。ベース・テーブルのカラムに制限するのはselectリストだけです。その理由は、UPDATEが実テーブルに含まれる実際のカラムしか更新できないためです。したがって、このUPDATEで重要なのはそれらのカラムと、SET句で使用されるNUMBER(*)関数だけです。

ステップ4：SELECTを実行して、候補結果セットを生成します。WHERE句とORDER BY句はこの時点で適用されます。また、NUMBER(*)関数の参照も、UPDATE SET句が実行される前に評価されます。SELECTの結果セットは次のようになります。各ローにあるA、B、C…の文字は、この先の説明でローを識別するために付けたものです。

```
   t1.key_1  t1.non_key_1  t2.key_1  t2.non_key_1  NUMBER(*)
   ========  ============  ========  ============  =========
A  2         0             2         100           1
B  2         0             2         100           2
C  2         0             1         100           3
D  2         0             1         100           4
E  1         0             2         100           5
F  1         0             2         100           6
G  1         0             1         100           7
H  1         0             1         100           8
```

候補結果セットの各ローには、t1とt2の両方のカラムが含まれています。また、CROSS JOINが行われているため、t1の各ローが候補結果セットの4つのローに現れます。t2についても同じです。t2の各ローが結果セットに4回現れます。

ステップ5：実ベース・テーブルのカラムのうち、候補結果セットに現れるカラムにSET句を適用します。SET句は次のようになります。

```
   SET t1.non_key_1 = NUMBER(*),
       t2.non_key_1 = NUMBER(*)
```

この処理は、候補結果セット内の各ローごとに実行されます。つまり、SET句が8回適用されます。また、2つのテーブルが処理されるため、ローの更新処理は合計16回行われることになります。

Aのローでは、NUMBER(*)は1を返します。このローは次のように更新されます。「t1.key_1 = 2」であるローのt1.non_key_1に1がセットされ、「t2.key_1 = 2」であるローのt2.non_key_1に1がセットされます。これは以下に示す2つの単一ローのUPDATE文を実行した場合と同じです。

```
UPDATE t1 SET non_key_1 = 1 WHERE key_1 = 2;
UPDATE t2 SET non_key_1 = 1 WHERE key_1 = 2;
```

以下は、更新される16のローすべてを、SQL Anywhereが実際に操作する順に並べたものです。最終的な答えが正しければ、順序は重要ではありません。t2の更新処理はすべて、t1よりも先に実行されていることがわかります。t1の各ローとt2の各ローがそれぞれ4回ずつ更新されますが、各ローの最後のUPDATE（finalと記入されたもの）だけが結果として残ります。

```
UPDATE t2 SET non_key_1 = 1 WHERE key_1 = 2; -- □ー A
UPDATE t2 SET non_key_1 = 2 WHERE key_1 = 2; -- □ー B
UPDATE t2 SET non_key_1 = 3 WHERE key_1 = 1; -- □ー C
UPDATE t2 SET non_key_1 = 4 WHERE key_1 = 1; -- □ー D
UPDATE t2 SET non_key_1 = 5 WHERE key_1 = 2; -- □ー E
UPDATE t2 SET non_key_1 = 6 WHERE key_1 = 2; -- □ー F - final
UPDATE t2 SET non_key_1 = 7 WHERE key_1 = 1; -- □ー G
UPDATE t2 SET non_key_1 = 8 WHERE key_1 = 1; -- □ー H - final
UPDATE t1 SET non_key_1 = 1 WHERE key_1 = 2; -- □ー A
UPDATE t1 SET non_key_1 = 2 WHERE key_1 = 2; -- □ー B
UPDATE t1 SET non_key_1 = 3 WHERE key_1 = 2; -- □ー C
UPDATE t1 SET non_key_1 = 4 WHERE key_1 = 2; -- □ー D - final
UPDATE t1 SET non_key_1 = 5 WHERE key_1 = 1; -- □ー E
UPDATE t1 SET non_key_1 = 6 WHERE key_1 = 1; -- □ー F
UPDATE t1 SET non_key_1 = 7 WHERE key_1 = 1; -- □ー G
UPDATE t1 SET non_key_1 = 8 WHERE key_1 = 1; -- □ー H - final
```

すべての更新処理が終了すると、t1とt2は次のようになります。

```
t1.key_1  t1.non_key_1
========  ============
1         8
2         4

t2.key_1  t2.non_key_1
========  ============
1         8
2         6
```

重要なのは、元のWHERE句である「WHERE t1.non_key_1 = 0」によって、16回のローの更新が（t1.non_key_1がすぐに0以外になるにもかかわらず）制限されなかったことです。その理由は、元のWHERE句は候補結果セットの決定に使用され、「SET t1.non_key_1 = NUMBER(*)」のSET句によって実行される実際の更新は、その後に行われるからです。

元のORDER BY句についても同じことが言えます。実際の更新処理は候補結果セットの順番に影響しません。この順番はその前のステップで決定するからです。

4.4.1 集合型UPDATE

集合指向型UPDATE文の構文全体を以下に示します。

```
<set_update>          ::= UPDATE
                            [ <row_range> ]
                            <table_specification>
                            <set_clause>
                            [ <where_clause> ]
                            [ <order_by_clause> ]
<row_range>           ::= FIRST                    -- TOP 1と同じ
                        | TOP <maximum_row_count>
                            [ START AT <start_at_row_number> ]
<maximum_row_count>   ::= 返すローの最大数を表す整数リテラル
<start_at_row_number> ::= 返すローの最初のロー番号を表す整数リテラル
<table_specification> ::=「第3章　選択」の<table_specification>を参照
<set_clause>          ::= SET { <set_item> "," }
                                <set_item>
<set_item>            ::= <column_reference> "=" <expression>
<column_reference>    ::= <column_name>
                        | <alias_name>
                        | [ <owner_name> "." ] <table_name> "." <column_name>
                        | <correlation_name> "." <column_name>
<order_by_clause>     ::=「第3章　選択」の<order_by_clause>を参照
<alias_name>          ::= <identifier>
<correlation_name>    ::= <identifier>
```

集合指向型UPDATEは、単一のテーブルの単一のローでも、複数のテーブルの複数のローでも自由に更新できます。更新するテーブル（更新しないテーブルも含む）のジョインの指定も可能です。また、ビュー、抽出テーブル、プロシージャ・コールも使用できます。これらの使用が実際に制限されるのは、更新されるテーブルにおいて、どのローを更新するかを確実に判断できない場合だけです。たとえば、UPDATE文にGROUP BY句を記述したり、UNIONをベースとするビューを更新したりできないのはこのためです。

先の項で説明したUPDATEは、t1とt2のテーブルがビュー、プロシージャ・コール、または抽出テーブルを通して参照されている場合でも、まったく同じように機能します。以下にその例を示します。

```
CREATE VIEW v1 AS
SELECT * FROM t1;

CREATE PROCEDURE p1()
BEGIN
   SELECT * FROM t1;
END;

UPDATE v1
       CROSS JOIN p1() AS x
       CROSS JOIN ( SELECT * FROM t2 ) AS d2
   SET v1.non_key_1 = NUMBER(*),
       d2.non_key_1 = NUMBER(*)
 WHERE v1.non_key_1 = 0
 ORDER BY v1.key_1 DESC,
       d2.key_1 DESC;
```

Tip

集合指向型UPDATE文の中で、テーブル名をカンマで区切られたリストで指定するのは、CROSS JOINを実際に処理するのでなければ避けてください。たとえば、先の例は、「UPDATE v1, p1() AS x, (SELECT * FROM t2) AS d2...」のようにカンマを使用して記述しても、まったく同じように処理されます。しかし、これは不自然な例であり、実際の世界ではCROSS JOINはあまり好まれません。誤ってCROSS JOINになった場合は特にそうです。

次に、ASADEMOデータベースを使用して、もっと現実的な例を示します。次のUPDATEは、営業成績が最も良かった従業員を10%昇給します。これは複数テーブルのジョインを使用した、単一ロー、単一テーブルのUPDATEです。つまり、employeeテーブルの単一のローしか更新されませんが、そのローはきわめて複雑なジョインによって決定されます。どの営業担当者の売り上げ(ドル)が最も多かったかは、抽出テーブルを使用して判断します。このUPDATEにはGROUP BYが使用されていますが、これは問題ありません。なぜなら、SET句はemployeeテーブルだけを参照し、GROUP BYに含まれるどのテーブルも参照していないからです。

```
UPDATE employee
       INNER JOIN ( SELECT FIRST sales_order.sales_rep AS top_rep_id
                    FROM sales_order
                         INNER JOIN sales_order_items
                                 ON sales_order_items.id = sales_order.id
                         INNER JOIN product
                                 ON product.id = sales_order_items.prod_id
                    GROUP BY sales_order.sales_rep
                    ORDER BY SUM ( sales_order_items.quantity
                                   * product.unit_price ) DESC )
                  AS top_rep
```

```
              ON top_rep.top_rep_id = employee.emp_id
   SET employee.salary = employee.salary * 1.1;
```

このUPDATEを実行して得られる結果は、次の文とまったく同じです。

```
UPDATE employee SET salary = 43230.00 WHERE emp_id = 299; -- 元の給与は 39300.00
```

プロシージャ・コールを通してて参照されるテーブルにSET句を実行することはできませんが、だからといってUPDATEの中でプロシージャ・コールを使用できないわけではありません。次に、プロシージャを呼び出して、成績が最も良かった1人以上の営業担当者（この場合は上位3名）を検索し、それぞれを10%昇給する例を示します。この例でも先のUPDATEと同じGROUP BYを使用しますが、ストアド・プロシージャを使用することで2つの利点を得ています。1つは、複雑なジョインをUPDATEから切り離しているため、コードが理解しやすいこと。もう1つは不変のFIRSTキーワードではなく変数TOPのカウントを使用するため、プロシージャが非常に柔軟になることです。

```
CREATE PROCEDURE p_top_salespeople
   ( IN @top_count INTEGER )
   RESULT ( top_rep_id INTEGER )
BEGIN
   DECLARE @select LONG VARCHAR;

   SET @select = STRING (
      'SELECT TOP ', @top_count, ' sales_order.sales_rep
        FROM sales_order
             INNER JOIN sales_order_items
                     ON sales_order_items.id = sales_order.id
             INNER JOIN product
                     ON product.id = sales_order_items.prod_id
       GROUP BY sales_order.sales_rep
       ORDER BY SUM ( sales_order_items.quantity
                           * product.unit_price ) DESC' );

   EXECUTE IMMEDIATE @select;
END;

UPDATE employee
       INNER JOIN p_top_salespeople ( 3 )
               ON p_top_salespeople.top_rep_id = employee.emp_id
   SET employee.salary = employee.salary * 1.1;
```

このUPDATEを実行して得られる結果は、次の3つの文とまったく同じです。

```
UPDATE employee SET salary = 43230.00 WHERE emp_id = 299;  -- 元の給与は 39300.00
UPDATE employee SET salary = 38381.20 WHERE emp_id = 856;  -- 元の給与は 34892.00
UPDATE employee SET salary = 49500.00 WHERE emp_id = 1142; -- 元の給与は 45000.00
```

ストアド・プロシージャとEXECUTE IMMEDIATE文の詳細については、「第8章　パッケージ」を参照してください。

FIRSTやORDER BYのような機能を利用するために、ビュー、抽出テーブル、またはプロシージャ・コールに頼る必要はありません。これらはUPDATE文そのものに使用できます。Finance部で勤続期間の最も短い従業員2人を5%減給する例は次のようになります。

```
UPDATE TOP 2
       employee
       INNER JOIN department
               ON department.dept_id = employee.dept_id
   SET employee.salary = employee.salary * 0.95
 WHERE department.dept_name = 'Finance'
 ORDER BY employee.start_date DESC;
```

このUPDATEを実行して得られる結果は、次の2つの文とまったく同じです。

```
UPDATE employee SET salary = 71630.00 WHERE emp_id = 1483; -- 元の給与は 75400.00
UPDATE employee SET salary = 55983.50 WHERE emp_id = 1390; -- 元の給与は 58930.00
```

Note

SQL Anywhere Studioでは、別のFROM句を使用してジョイン条件を指定する、もう1つの形式の集合指向型UPDATEを使用できます。この形式のUPDATEは紛らわしく、相関名を間違える危険さえあります。また、1つのテーブルしか更新できないという制限もあるので、本書では取り上げません。この項で説明したUPDATE構文は、わかりやすく簡単です。ジョイン条件をUPDATEキーワードに続けて指定し、更新するテーブルとカラムをSET句に指定します。

Section 4.5

UPDATE WHERE CURRENT OF カーソル

この項では、カーソル指向型のUPDATE文の動作について簡単に説明します。

```
<update_where_current_of_cursor> ::= UPDATE
                                       <table_or_view_list>
                                       <set_clause>
                                       <where_current_of_clause>
<table_or_view_list>             ::= { <table_or_view_reference> "," }
                                     <table_or_view_reference>
```

```
<table_or_view_reference>        ::= [ <owner_name> "." ] <table_name>
                                   | [ <owner_name> "." ] <view_name>
<where_current_of_clause>        ::= WHERE CURRENT OF <cursor_name>
<cursor_name> ::= カーソルの DECLARE または FOR 文に定義された <identifier>
```

WHERE CURRENT OF 句を使用して、カーソル FETCH ループによる UPDATE を実行する場合の動作は、「4.4　集合型 UPDATE の実行ロジック」での説明と同様、5 つのステップで説明できます。ステップ 1 ～ 4 までは異なる部分があり、第 4 章では候補結果セットを作成していましたが、カーソル指向型の場合は、カーソル宣言に明示的に定義する SELECT 文がその役割を果たします。最後のステップである SET 句だけが、実際に UPDATE 文によって実行されます。

この形式の UPDATE では、FROM 句やジョイン処理を使用しません。それらはカーソルの SELECT 内で行われます。UPDATE では更新の対象となるテーブルやビューを指定する必要があります。複数を更新する場合はカンマ区切りのリストを使用でき、CROSS JOIN にしてしまう危険はありません。このリストにはテーブル名とビュー名を単純に並べて指定します。

カーソル指向型の UPDATE を 1 度実行するたびに、カーソル結果セットの 1 つのローだけが処理されます。ただし、複数のベース・テーブルを更新するように SET 句が指定されている場合は、複数のローが処理される場合もあります。

次に示す例は、4.4 項で使用した例とまったく同じ更新を実行します。カーソルの DECLARE には、まったく同じテーブル式と WHERE 句、ORDER BY 句を使用した SELECT を定義しています。また、UPDATE WHERE CURRENT OF 文に使用している SET 句もまったく同じです。

```
BEGIN
DECLARE @t1_key_1          INTEGER;
DECLARE @t1_non_key_1      INTEGER;
DECLARE @t2_key_1          INTEGER;
DECLARE @t2_non_key_1      INTEGER;
DECLARE @number            INTEGER;
DECLARE @SQLSTATE          VARCHAR ( 5 );

DECLARE cloop1 CURSOR FOR
SELECT t1.key_1,
       t1.non_key_1,
       t2.key_1,
       t2.non_key_1
  FROM t1
       CROSS JOIN t1 AS x
       CROSS JOIN t2
 WHERE t1.non_key_1 = 0
 ORDER BY t1.key_1 DESC,
       t2.key_1 DESC;

OPEN cloop1;
```

```
FETCH cloop1 INTO
   @t1_key_1,
   @t1_non_key_1,
   @t2_key_1,
   @t2_non_key_1;
SET @SQLSTATE = SQLSTATE;

SET @number = 0;

WHILE ( @SQLSTATE IN ( '00000', '01W04' ) ) LOOP

   SET @number = @number + 1;

   UPDATE t1, t2
      SET t1.non_key_1 = @number,
          t2.non_key_1 = @number
    WHERE CURRENT OF cloop1;

   FETCH cloop1 INTO
      @t1_key_1,
      @t1_non_key_1,
      @t2_key_1,
      @t2_non_key_1;
   SET @SQLSTATE = SQLSTATE;

END LOOP;
CLOSE cloop1;
END;
```

可能であれば、これよりもずっと短いFORループ構文を使用してください。上記のカーソル・ループには多数のローカル変数に加え、DECLARE文、OPEN文、そしてFETCH文が別々に使用されています。Embedded SQLなどのアプリケーション・プログラムでカーソル・ループを記述する場合は、「6.2 カーソルFETCHループ」で示す例のようなコードを使用する必要があります。しかし、SQLでカーソル・ループを記述する場合は、上記に示すようなコードであれば、「6.3　カーソルのFORループ」で説明する簡単なFORループを使用できます。

このループを実行して得られる結果は、次の一連の文とまったく同じです。

```
UPDATE t2 SET non_key_1 = 1 WHERE key_1 = 2; -- ループ・パス #1
UPDATE t1 SET non_key_1 = 1 WHERE key_1 = 2;
UPDATE t2 SET non_key_1 = 2 WHERE key_1 = 2; -- ループ・パス #2
UPDATE t1 SET non_key_1 = 2 WHERE key_1 = 2;
UPDATE t2 SET non_key_1 = 3 WHERE key_1 = 1; -- ループ・パス #3
UPDATE t1 SET non_key_1 = 3 WHERE key_1 = 2;
```

```
UPDATE t2 SET non_key_1 = 4 WHERE key_1 = 1; -- ループ・パス #4
UPDATE t1 SET non_key_1 = 4 WHERE key_1 = 2;
UPDATE t2 SET non_key_1 = 5 WHERE key_1 = 2; -- ループ・パス #5
UPDATE t1 SET non_key_1 = 5 WHERE key_1 = 1;
UPDATE t2 SET non_key_1 = 6 WHERE key_1 = 2; -- ループ・パス #6
UPDATE t1 SET non_key_1 = 6 WHERE key_1 = 1;
UPDATE t2 SET non_key_1 = 7 WHERE key_1 = 1; -- ループ・パス #7
UPDATE t1 SET non_key_1 = 7 WHERE key_1 = 1;
UPDATE t2 SET non_key_1 = 8 WHERE key_1 = 1; -- ループ・パス #8
UPDATE t1 SET non_key_1 = 8 WHERE key_1 = 1;
```

カーソル・ループの場合、ローが更新される順番は異なりますが、t1 と t2 のテーブルの最終的な内容は 4.4 の項の例と同じです。

```
t1.key_1  t1.non_key_1
========  ============
1         8
2         4

t2.key_1  t2.non_key_1
========  ============
1         8
2         6
```

前述の WHILE は、SQLSTATE の２つの値（00000 と 01W04）をテストします。00000 はすべて正常であることを示し、01W04 はフェッチしようとしているベース・テーブル・ローが前回のフェッチ以降に変更されているという警告を示します。この例では警告を無視していますが、アプリケーションによってはビジネスの観点においてこれが重要な問題になることもあります。

カーソル・ループについては、「第 6 章　フェッチ」でさらに詳しく説明します。

Section

4.6 この章のまとめ

この章では単一のテーブルに含まれる単一のロー、および複数のローの更新について説明し、複数のロー、複数のテーブルに対するUPDATEの動作について説明しました。集合指向型UPDATE文の構文全体を紹介した後、カーソル指向型のUPDATE WHERE CURRENT OF文について簡単に説明しました。

次の章ではデータベースのライフ・サイクルにおける5番目のステップ、すなわちデータの削除について説明します。

Chapter

5

削除

Section

5.1 概要

この章では、最初に一般的な DELETE 文、つまり、単一のテーブルに含まれる単一のローと複数のローの DELETE 文について説明します。次に、5.4 項で、複数テーブルのジョインを含む複雑な DELETE 文の動作について、理論的な観点から解説します。ここでは、DELETE が単一のテーブルのローしか処理しないこと、そして DELETE が単一のローを 1 度しか処理できないことという、UPDATE と DELETE における 2 つの重要な違いについて説明します。

5.5.1 項では、集合指向型の DELETE の構文全体を現実的な例を用いて解説します。5.5 項では、カーソル指向型の DELETE WHERE CURRENT OF 文と、この文がカーソル FETCH ループの実行に及ぼす影響について説明します。5.6 項では、高速な TRUNCATE TABLE と、その注意すべき副作用について説明します。また、一部のローを残して削除を実行する場合の TRUNCATE TABLE の使用方法についても紹介します。

Section

5.2 単一のローの削除

最も単純な形式の DELETE 文は、単一のテーブルに含まれる単一のローを削除するものです。

```
<typical_single_row_delete> ::= DELETE [ <owner_name> "." ] <table_name>
                                    <single_row_where_clause>
<owner_name>                ::= <identifier>
<table_name>                ::= <identifier>
<single_row_where_clause>   ::= 1 つのローにだけ一致する <where_clause>
<identifier>                ::= 「第 1 章　作成」の <identifier> を参照
<where_clause>              ::= 「第 3 章　選択」の <where_clause> を参照
```

通常、単一のローの DELETE には、テーブル名とそのテーブルの単一のローに一致する WHERE 句を指定します。次の例は、ASADEMO データベース内で、プライマリ・キーが「id = 2015」と「line_id = 4」で構成される単一の受注項目を削除します。

```
DELETE sales_order_items
 WHERE id = 2015
   AND line_id = 4;
```

SQL Anywhere Studio 9 に付属の ASADEMO データベースに含まれる、sales_order_items テーブ

ルの詳細については、「3.6　マルチテーブル・ジョイン」を参照してください。

WHERE 句がローに一致すると、DELETE は次のように処理されます。このテーブルに関連付けられたすべての BEFORE DELETE トリガが起動し、ローのロックが取得されます。ローが削除されて、削除がトランザクション・ログに記録されます。すべての AFTER DELETE トリガが起動し、完全に正常な状態であることを示すために、SQLSTATE という特殊なリテラルが '00000' にセットされます。トリガの詳細については、「8.11　CREATE TRIGGER」を参照してください。また、ロックの詳細については、「9.6　ロック」を参照してください。

WHERE 句がどのローとも一致しない場合もエラーではありませんが、SQLSTATE は「ローが見つかりません」を示す値である '02000' にセットされます。この場合、トリガは何も起動せず、ローのロックも取得されません。削除は実行されず、トランザクション・ログには何も記録されません。

Note

これらの動作は、複数のローの削除の場合も同じです。これらの動作が１つ１つのローに対して実行されます。ただし、DELETE 文が実行されるたびに SQLSTATE が１度設定される点と、DELETE 文が実行されるたびに AFTER STATEMENT トリガが１度起動される点の２つが異なります。

Note

本書では、ISOLATION_LEVEL オプションの値が、デフォルトの０に設定されていることを前提としています。これは、パフォーマンスと同時実行性を最高レベルにする値です。０よりも大きい値に設定した場合、ロックの動作が異なる可能性があります。たとえば、ISOLATION_LEVEL が３に設定されていると、先ほど述べたような「WHERE 句がどのローとも一致しない場合、ローのロックは取得されない」という保証はありません。独立性レベルの詳細については、「9.7　ブロックと独立性レベル」を参照してください。

Tip

単一のローの DELETE 文では、誤って複数のロー（もしくはテーブル全体）を削除しないように注意してください。WHERE 句が必ずプライマリ・キーまたはユニークなインデックスを参照していること、また、プライマリ・キーやインデックスが２つ以上のカラムから成る場合は、WHERE 句にそれらすべてのカラムの値を指定していることを確認してください。

Section

5.3 複数のローの削除

単一のローのDELETEと複数のローのDELETEには、それほど大きな違いはありません。実際、単一のローのDELETEは、通常はWHERE句の制限を緩めるか、WHERE句の指定を削除するだけで複数のローのDELETEになります。

```
<typical_multi_row_delete> ::= DELETE [ <owner_name> "." ] <table_name>
                                  [ <multi_row_where_clause> ]
<multi_row_where_clause>   ::= 0または1つ以上のローに一致する<where_clause>
```

WHERE句を省略すると、テーブル内のすべてのローが削除されます。sales_order_itemsテーブルに含まれる1,097ローをすべて削除する例を次に示します。

```
DELETE sales_order_items;
```

Tip

すべてのローを削除する場合は、TRUNCATE TABLEの方がDELETEよりもかなり高速に処理できます。詳細については、この章の5.6項を参照してください。

1つの受注（sales_order）に対するすべての受注品目（sales_order_items）を削除する例を以下に示します。idカラムは受注を識別します。このカラムは、sales_order_itemsテーブルのプライマリ・キーを構成する2つのカラムのうちの1つです。

```
DELETE sales_order_items
 WHERE id = 2015;
```

この文を実行すると4つのローが削除されます。テーブルに関する限り、この結果は単一のローを削除する以下の4つの文と同じになります。

```
DELETE sales_order_items WHERE id = 2015 AND line_id = 1;
DELETE sales_order_items WHERE id = 2015 AND line_id = 2;
DELETE sales_order_items WHERE id = 2015 AND line_id = 3;
DELETE sales_order_items WHERE id = 2015 AND line_id = 4;
```

Tip

大きなテーブルに含まれるローの大半を削除する場合は、削除しないローをテンポラリ・テーブルにコピーし、TRUNCATE TABLE を使用してすべて削除した後、コピーしておいたローを元に戻すという方法の方が高速に処理できます。詳細については、5.6 項を参照してください。

Section

5.4 集合型DELETEの実行ロジック

集合指向型 DELETE の構文は、ローの範囲を指定する句（FIRST または TOP）、「3.3　FROM 句」で説明したテーブル仕様、および WHERE 句で構成されます。削除できるのは単一のテーブルのローだけですが、DELETE 文には複数のテーブルのジョインを指定できます。

```
<set_delete> ::= DELETE [ <row_range> ] [ FROM ]
                    <table_or_view_reference> [ AS <correlation_name> ]
                    <from_clause>
                    [ <where_clause> ]
```

DELETE の句については、次の項で詳しく説明します。この項では「複数テーブルの DELETE では実際に何が行われるのか」という疑問について考えます。

DELETE の動作について説明するために、要求された機能を実行するときに行われる一連の単純なステップを紹介します。「3.2　SELECT の論理的実行」で説明したステップと同様、これらも理論上の（つまり、仮想的な）ステップです。実際に行われるステップではなく、「行われ得る」ステップであることに注意してください。

以下に、DELETE が処理される際の各ステップを、理論的な観点から簡単に説明します。

❶ DELETE に対応する SELECT 文の作成が開始されます。DISTINCT キーワードを追加して、TOP 句または FIRST 句、FROM 句、そして WHERE 句を SELECT にコピーします。
❷ どのベース・テーブルで削除を行うかを決定します。
❸ 削除を行うベース・テーブルのすべてのカラムを select リストに入れます。
❹ SELECT を実行して、候補結果セットを生成します。
❺ 候補結果セットに出現するローをベース・テーブルから削除します。

この項の残りの部分では、1 つの単純なテーブルに含まれる 5 つのローを例に、これらのステップをさらに詳しく説明します。

```
CREATE TABLE t1 (
   key_1         UNSIGNED INTEGER NOT NULL PRIMARY KEY,
   non_key_1     INTEGER NOT NULL );

INSERT t1 VALUES ( 1, 1 );
INSERT t1 VALUES ( 2, 2 );
INSERT t1 VALUES ( 3, 3 );
INSERT t1 VALUES ( 4, 4 );
INSERT t1 VALUES ( 5, 5 );
```

以下に示すDELETEの例は、各ステップが実行される様子を示すためだけに作成したものです。FROM句には、t1とそれ自身とのCROSS JOINを定義しています。また、WHERE句は、候補結果セットに入れるローを「t1.key_1 = 2」に制限します。

```
DELETE t1
  FROM t1
       CROSS JOIN t1 AS x
 WHERE t1.key_1 = 2;
```

このDELETEを実行すると、t1からローが1つだけ削除されます。この結果は次の文を実行した場合と同じです。

```
DELETE t1 WHERE key_1 = 2;
```

ここからはDELETEの処理方法について、理論上の各ステップをより詳しく説明します。

ステップ1：SELECT文の作成が開始されます。DISTINCTキーワードを追加して、TOP句またはFIRST句、FROM句、そしてWHERE句をSELECTにコピーします。この例ではTOP句もFIRST句も使用していません。したがって、このステップの後の未完成のSELECTは次のようになります。

```
SELECT DISTINCT ...
  FROM t1
       CROSS JOIN t1 AS x
 WHERE t1.key_1 = 2;
```

ステップ2：DELETEキーワードに続けて記述した名前を調べることによって、削除の対象となるベース・テーブルが判断されます。ここで言う「ベース・テーブル」はビューではなく、データベースに含まれる実際のテーブルを指します。DELETEキーワードの後にビュー名を記述することも可能です。その場合は、実際に削除されるベース・テーブルを、このステップの中で判断する必要があります。

Note

ビューを通してローを削除できるのは、そのビューが更新可能ビューとして定義され、かつ、テーブルを1つだけ含んでいる場合に限られます。ビューの詳細については、「3.23　CREATE VIEW」を参照してください。

この例のベース・テーブルはt1です。この情報は次のステップで必要になります。また、t1はFROM句で2度指定されていますが、これは問題ありません。

ステップ3：削除が行われるベース・テーブルのすべてのカラムをselectリストに入れます。このステップの実行後、SELECTは次のようになります。「t1.*」と指定している点に注目してください。「SELECT *」としただけでは、これと同じ結果になりません。

```
SELECT DISTINCT t1.*
  FROM t1
       CROSS JOIN t1 AS x
 WHERE t1.key_1 = 2;
```

Note

この例のように、テーブルにプライマリ・キー制約が指定されている場合、SELECT DISTINCTリストにはプライマリ・キー・カラム以外は記述する必要がありません。プライマリ・キー・カラムさえあれば、削除するローを検索できます。「t1.*」と記述してすべてを選択しているのは、例をわかりやすく簡単にするためです。

ステップ4：SELECTが実行されて、候補結果セットが生成されます。このSELECTが「SELECT *」のように記述されていれば、CROSS JOINは最初に25ローを生成し、WHEREによって5ローまで減らされます。しかし、「SELECT DISTINCT t1.*」とあるため、1つのローだけが生成されます。

```
key_1  non_key_1
=====  =========
2      2
```

ステップ5：候補結果セットに出現するローがベース・テーブルから削除されます。この例の実行結果は、次の単一の文を実行した場合と同じです。

```
DELETE t1 WHERE key_1 = 2;
```

これらのステップから、UPDATEとDELETEの重要な違いが2つわかります。DELETEは単一のテーブルに対してのみ動作し、各ローを1度しか削除できません。一方、UPDATEは複数のテーブルに対

して実行でき、各ローを複数回変更できます。

5.4.1 集合型DELETE

集合指向型 DELETE 文には、FROM 句を付ける形式と付けない形式の 2 とおりがあります。どちらの形式を使用しても複数のローを削除できますが、削除できるのは単一のテーブルのローだけです。FROM 句を使用すると、削除するローを選択するプロセスに、他のテーブルを参加させることができます。

```
<set_delete>           ::= DELETE [ <row_range> ] [ FROM ]
                              <table_or_view_reference> [ AS <correlation_name> ]
                              <from_clause>
                              [ <where_clause> ]
<simple_set_delete>    ::= DELETE [ <row_range> ] [ FROM ]
                              <table_or_view_reference>
                              [ <where_clause> ]
<row_range>            ::= FIRST
                         | TOP <row_count> [ START AT <row_number> ]
<row_count>            ::= 返すローの最大数を表す整数リテラル
<row_number>           ::= 返すローの最初のロー番号を表す整数リテラル
<table_or_view_reference> ::= [ <owner_name> "." ] <table_name>
                            | [ <owner_name> "." ] <view_name>
<view_name>            ::= <identifier>
<correlation_name>     ::= <identifier>
<from_clause>          ::= FROM <table_specification>
<table_specification>  ::=「第 3 章　選択」の <table_specification> を参照
```

Tip

DELETE キーワードの直後に続くオプションの FROM キーワードを使ってはいけません。このキーワードを FROM 句と混同しないよう注意してください。"FROM" のキーワードを省略すると、「ジョインを行うテーブルがある」という意味になります。これは、SELECT や UPDATE などの文と同じです。

次に、「3.6　マルチテーブル・ジョイン」で解説した、ASADEMO データベースの 4 つのテーブル（sales_order、sales_order_items、employee、customer）を使用した例を示します。この例では、既に退社した 2 人の営業担当者による、現在は営業活動をしていない 2 つの顧客からの古い受注をすべて削除します。具体的には、2000 年 12 月 31 までの、Rollin Overbey と Philip Chin という従業員による、The Power Group と Darling Associates という顧客からの受注に関するすべてのローを、sales_order と sales_order_items から削除します。

Tip

複雑な集合指向型 DELETE をコーディングするときには、まずプロトタイプの SELECT 文を記述し、対象のデータを表示して正しいローを取得していることを確認してください。ISQL で FROM 句をテストした後、SELECT を DELETE に変更すると非常に簡単です。

次に示す SELECT 文は、削除されるデータを表示します。結果セットには、sales_order のプライマリ・キー・カラム（order_id）と sales_order_items のプライマリ・キー・カラム（order_id と line_id）、従業員名、および顧客名が含まれます。

```
SELECT customer.id AS cust_id,
       customer.company_name,
       sales_order.id AS order_id,
       sales_order.order_date,
       employee.emp_id,
       STRING ( employee.emp_fname, ' ', employee.emp_lname ) AS emp_name,
       sales_order_items.line_id
  FROM customer
       INNER JOIN sales_order
               ON sales_order.cust_id = customer.id
       INNER JOIN employee
               ON employee.emp_id = sales_order.sales_rep
       INNER JOIN sales_order_items
               ON sales_order_items.id = sales_order.id
 WHERE STRING ( employee.emp_fname, ' ', employee.emp_lname )
          IN ( 'Rollin Overbey', 'Philip Chin' )
   AND customer.company_name
          IN ( 'The Power Group', 'Darling Associates' )
   AND sales_order.order_date <= '2000-12-31'
 ORDER BY 1, 2, 3, 4, 5, 6, 7;
```

SELECT から返されるデータは次のようになります。5 つの受注（order_id の値が 5 種類）からの計 11 ローが、sales_order_item から返されます。また、会社名、受注日、従業員名が正しく選択されていることもわかります。

```
cust_id  company_name        order_id  order_date  emp_id  emp_name        line_id
=======  ==================  ========  ==========  ======  ==============  =======
101      The Power Group     2001      2000-03-16  299     Rollin Overbey  1
101      The Power Group     2001      2000-03-16  299     Rollin Overbey  2
101      The Power Group     2001      2000-03-16  299     Rollin Overbey  3
101      The Power Group     2206      2000-04-16  299     Rollin Overbey  1
101      The Power Group     2206      2000-04-16  299     Rollin Overbey  2
101      The Power Group     2206      2000-04-16  299     Rollin Overbey  3
101      The Power Group     2206      2000-04-16  299     Rollin Overbey  4
101      The Power Group     2279      2000-07-23  299     Rollin Overbey  1
```

```
103      Darling Associates  2340      2000-09-25  299     Rollin Overbey  1
103      Darling Associates  2451      2000-12-15  129     Philip Chin     1
103      Darling Associates  2451      2000-12-15  129     Philip Chin     2
```

1つのDELETEで処理できるテーブルは1つだけなので、sales_order用とsales_order_items用の2つのDELETE文が必要です。sales_order_itemsはsales_orderと外部キー関係を持つ子テーブルなので、先に削除する必要があります。最初のDELETEは次のとおりです。FROM句とWHERE句は、先ほどのSELECTとまったく同じです。

```
DELETE sales_order_items
  FROM customer
       INNER JOIN sales_order
               ON sales_order.cust_id = customer.id
       INNER JOIN employee
               ON employee.emp_id = sales_order.sales_rep
       INNER JOIN sales_order_items
               ON sales_order_items.id = sales_order.id
 WHERE STRING ( employee.emp_fname, ' ', employee.emp_lname )
          IN ( 'Rollin Overbey', 'Philip Chin' )
   AND customer.company_name
          IN ( 'The Power Group', 'Darling Associates' )
   AND sales_order.order_date <= '2000-12-31';
```

このDELETEを実行すると、次に示す単一のローのDELETE文を実行した場合とまったく同じ処理が行われます。

```
DELETE sales_order_items WHERE id = 2001 AND line_id = 1;
DELETE sales_order_items WHERE id = 2001 AND line_id = 2;
DELETE sales_order_items WHERE id = 2001 AND line_id = 3;
DELETE sales_order_items WHERE id = 2206 AND line_id = 1;
DELETE sales_order_items WHERE id = 2206 AND line_id = 2;
DELETE sales_order_items WHERE id = 2206 AND line_id = 3;
DELETE sales_order_items WHERE id = 2206 AND line_id = 4;
DELETE sales_order_items WHERE id = 2279 AND line_id = 1;
DELETE sales_order_items WHERE id = 2340 AND line_id = 1;
DELETE sales_order_items WHERE id = 2451 AND line_id = 1;
DELETE sales_order_items WHERE id = 2451 AND line_id = 2;
```

sales_orderのDELETEもほとんど同じですが、sales_order_itemsとのINNER JOINは、削除するかLEFT OUTER JOINに変更しなければならない点が異なります。その理由は、sales_order_items内の一致するすべてのローが既に削除されているため、INNER JOINを行っても空の結果セットが作成されて、DELETEが何も処理しないからです。sales_order用のDELETEは次のとおりです。sales_order_itemsとのINNER JOINは削除しています（この例では、OUTER JOINを使用しても特に意味はありません）。

```
DELETE sales_order
  FROM customer
       INNER JOIN sales_order
               ON sales_order.cust_id = customer.id
       INNER JOIN employee
               ON employee.emp_id = sales_order.sales_rep
 WHERE STRING ( employee.emp_fname, ' ', employee.emp_lname )
         IN ( 'Rollin Overbey', 'Philip Chin' )
   AND customer.company_name
         IN ( 'The Power Group', 'Darling Associates' )
   AND sales_order.order_date <= '2000-12-31';
```

新しいFROM句には5つのローが一致します。このDELETEを実行した結果は、次の5つの文を実行した場合とまったく同じです。

```
DELETE sales_order WHERE id = 2001;
DELETE sales_order WHERE id = 2206;
DELETE sales_order WHERE id = 2279;
DELETE sales_order WHERE id = 2340;
DELETE sales_order WHERE id = 2451;
```

5.4項の冒頭で説明した4つ目までのステップが、上記の集合指向型の2つのDELETE文で同様に実行され、削除されるローを示すSELECT文が作成されます。作成される2つのSELECT文を次に示します。

```
SELECT DISTINCT sales_order_items.*
  FROM customer
       INNER JOIN sales_order
               ON sales_order.cust_id = customer.id
       INNER JOIN employee
               ON employee.emp_id = sales_order.sales_rep
       INNER JOIN sales_order_items
               ON sales_order_items.id = sales_order.id
 WHERE STRING ( employee.emp_fname, ' ', employee.emp_lname )
         IN ( 'Rollin Overbey', 'Philip Chin' )
   AND customer.company_name
         IN ( 'The Power Group', 'Darling Associates' )
   AND sales_order.order_date <= '2000-12-31';

SELECT DISTINCT sales_order.*
  FROM customer
       INNER JOIN sales_order
               ON sales_order.cust_id = customer.id
       INNER JOIN employee
               ON employee.emp_id = sales_order.sales_rep
       INNER JOIN sales_order_items
```

```
              ON sales_order_items.id = sales_order.id
 WHERE STRING ( employee.emp_fname, ' ', employee.emp_lname )
         IN ( 'Rollin Overbey', 'Philip Chin' )
   AND customer.company_name
         IN ( 'The Power Group', 'Darling Associates' )
   AND sales_order.order_date <= '2000-12-31';
```

次に示す例では、少なくとも3年以上が経過したsales_order_itemsのローをビューによって選択し、その後、簡単なDELETEを使用して、出荷数が12以下の古いローを削除します。このDELETEではジョインを使用しないため、FROM句を指定する必要はありません。また、ビューを使用しますが、このビューはテーブルを1つだけ含み、GROUP BYやUNIONなどの機能を何も使用しないので問題はありません。

```
CREATE VIEW v_old_items AS
SELECT *
  FROM sales_order_items
 WHERE ship_date < DATEADD ( YEAR, -3, CURRENT DATE );

DELETE v_old_items
 WHERE quantity <= 12;
```

DELETEをこのように使用すると、データベースから古いローを消去できるので便利です。毎日でも繰り返し実行でき、時間の経過と共に不要になったローを削除できます。

Section

5.5 DELETE WHERE CURRENT OF カーソル

この項では、カーソル指向型のDELETE文の動作について簡単に説明します。

```
<delete_where_current_of_cursor> ::= DELETE <table_or_view_reference>
                                        <where_current_of_clause>
<where_current_of_clause>        ::= WHERE CURRENT OF <cursor_name>
<cursor_name> ::= カーソルのDECLAREまたはFOR文に定義した<identifier>
```

WHERE CURRENT OF句を使用してカーソルFETCHループによるDELETEを実行する場合の動作は、「5.4　集合型DELETEの実行ロジック」で説明したのと同じ5つのステップを用いて説明できます。ステップ1～4までは同じではありません。これらの処理では候補結果セットを作成していましたが、カーソル指向型の場合、カーソル宣言に明示的に定義するSELECT文がその役割を果たします。ローを削除する最後のステップだけが、実際にDELETE文によって実行されます。

この形式のDELETEでは、FROM句やジョイン処理を使用しません。それらはカーソルのSELECT

内で行います。DELETEには、削除するテーブルまたはビューの他に、使用するカーソルを指定する必要があります。

カーソル指向型のDELETE文が実行されるたびに、1つのテーブルに含まれる1つのローが削除されます。次に、5.4項の例とまったく同じ削除を実行する例を示します。カーソルのDECLAREには、まったく同じFROM句とWHERE句を指定したSELECTを定義しています。

```
CREATE TABLE t1 (
   key_1         UNSIGNED INTEGER NOT NULL PRIMARY KEY,
   non_key_1     INTEGER NOT NULL );

INSERT t1 VALUES ( 1, 1 );
INSERT t1 VALUES ( 2, 2 );
INSERT t1 VALUES ( 3, 3 );
INSERT t1 VALUES ( 4, 4 );
INSERT t1 VALUES ( 5, 5 );

BEGIN
DECLARE @t1_key_1          INTEGER;
DECLARE @t1_non_key_1      INTEGER;
DECLARE @SQLSTATE          VARCHAR ( 5 );

DECLARE cloop1 CURSOR FOR
SELECT t1.key_1,
       t1.non_key_1
  FROM t1
       CROSS JOIN t1 AS x
 WHERE t1.key_1 = 2;

OPEN cloop1;

FETCH cloop1 INTO
   @t1_key_1,
   @t1_non_key_1;
SET @SQLSTATE = SQLSTATE;

WHILE ( @SQLSTATE = '00000' ) LOOP

   DELETE t1
    WHERE CURRENT OF cloop1;

   FETCH cloop1 INTO
      @t1_key_1,
      @t1_non_key_1;
   SET @SQLSTATE = SQLSTATE;

END LOOP;
CLOSE cloop1;
END;
```

このループを実行して得られる結果は、次の1つの文とまったく同じです。

```
DELETE t1 WHERE key_1 = 2;
```

SELECT に指定された CROSS JOIN によって、5つのローを含む候補結果セットが生成されますが、実際には WHILE ループが1度だけ実行された後、FETCH によって SQLSTATE に（ローが見つからないことを示す）'02000' がセットされます。ループが途中で終了するのは、候補結果セットのすべてのローに出現するローを DELETE がベース・テーブルから削除し、実質的に結果セットを消去することになるためです。カーソル・ループの詳細については、「第6章　フェッチ」を参照してください。

Section

5.6 TRUNCATE TABLE

TRUNCATE TABLE 文は、テーブルに含まれるすべてのローを削除します。多くの場合、集合指向型 DELETE 文を使うよりもはるかに高速に処理できます。

```
<truncate_table> ::= TRUNCATE TABLE [ <owner_name> "." ] <table_name>
```

TRUNCATE TABLE には、高速と低速の2つのバージョンがあります。高速バージョンは2つの要件を満たす場合に使用されます。1つは、空でない子テーブルが存在しないこと。もう1つは TRUNCATE_WITH_AUTO_COMMIT データベース・オプションが 'ON'（デフォルト）になっていることです。

最初の要件は、トランケートされるテーブルが、ローを持つ子テーブルと外部キー関係にある親であってはならないという意味です。つまり、子テーブルが存在すること自体は問題ではありませんが、それが空でない場合は高速バージョンの TRUNCATE TABLE は使用されません。

2つ目の、TRUNCATE_WITH_AUTO_COMMIT が 'ON' という要件は少し複雑です。最初の要件が満たされている場合、TRUNCATE TABLE はローの削除の終了時に COMMIT を実行します。また、同じく最初の要件が満たされており、かつ、TRUNCATE TABLE を実行する前にトランザクションが既に進行中である場合は、ローの削除を開始する前にも COMMIT が実行されます。最初の要件が満たされていない場合は、TRUNCATE_WITH_AUTO_COMMIT が 'ON' であっても、TRUNCATE TABLE によって COMMIT が実行されることはありません。

高速な TRUNCATE TABLE と低速な TRUNCATE TABLE には、かなりの違いがあります。あるテストでは、高速バージョンの TRUNCATE TABLE を使用したとき、サイズが 50M ある 30,000 のローが 10 秒で削除されました。低速バージョンの TRUNCATE TABLE と DELETE 文では、同じ処理にどちらも 4 分半かかりました。

高速バージョンのTRUNCATE TABLEは、いくつかショートカットを行うことによって処理時間を短縮しています。1つ目のショートカットは、低速バージョンと同様、TRUNCATE TABLEが削除トリガを何も起動しないことです。削除トリガに重要なアプリケーション・ロジックが含まれる場合、そのロジックは実行されないので、別の方法でデータを削除しなければならない場合もあります。

ただし、TRUNCATE TABLEで外部キーのチェックが行われないという意味ではありません。逆に、削除を試みたローが外部キー関係の親であった場合は、TRUNCATE TABLE文が失敗します。これはON DELETE CASCADEを指定している場合でも同じです。TRUNCATE TABLEはON DELETE RESTRICTが指定されているかのように動作します。親テーブルから子テーブルに削除をカスケードするためにON DELETE CASCADEを使用することはできません。当然ながら、高速バージョンのTRUNCATE TABLEが参照整合性に違反することはありません。子テーブルが存在する場合、そのテーブルが空でなければ高速バージョンを使用できないからです。

Note

子テーブルが空でない場合でも、外部キー・カラムに含まれる値がNULLだけであれば、参照整合性違反が発生しないので、TRUNCATE TABLEは正常に実行されます。ただし、子テーブルが空ではないので、高速バージョンのTRUNCATE TABLEは使用されません。この条件下では、TRUNCATE_WITH_AUTO_COMMITの'ON'は無視され、TRUNCATE TABLEによってコミットが実行されることはありません。

2つ目のショートカットも高速と低速の両方のTRUNCATE TABLEで行われます。1つ1つのローの削除はトランザクション・ログ・ファイルに記録されず、TRUNCATE TABLEコマンドそのものだけが記録されます。したがって、削除したローをMobile Linkのアップロード・ストリームに含めるのであれば、Mobile Linkを介してアップロードするテーブルにはTRUNCATE TABLEを使用しないでください。Mobile Linkはトランザクション・ログを調べることによってアップロードするローを判断しますが、TRUNCATE TABLEを使用して削除したローはログに記録されません。Mobile Linkの詳細については、「第7章　同期」を参照してください。

3つ目は、高速バージョンのTRUNCATE TABLEだけで行われるショートカットです。削除する1つ1つのローに対してロックを取得することはせず、テーブル全体に対して排他ロックを取得します。この方法では、ほとんどの場合、同時実行性の問題が起こる可能性が相対的に低くなります。他の方法（つまり、DELETEや低速のTRUNCATE TABLE）は、実行に時間がかかるうえ、すべてのローに対してロックを取得するからです。

4つ目のショートカットも高速バージョンのTRUNCATE TABLEだけで行われます。ロールバック・ログとチェックポイント・ログのための余分な領域が、データベース・ファイルに割り当てられることがありません。

Note

DELETE または低速バージョンの TRUNCATE TABLE を使用して、大きなテーブルからすべてのローを削除し、それらを再び挿入すると、ロールバック・ログとチェックポイント・ログを保持するのに必要となる領域のせいで、データベース・ファイルのサイズが２倍にも３倍にもなる可能性があります。これらのログの詳細については、「9.11　ロギングとリカバリ」を参照してください。

Tip

大きなテーブルからすべてのローを削除した後に変更をコミットする際、データベース・ファイルのサイズが増大するのを避けたい場合は、明示的な COMMIT 文と CHECKPOINT 文を、DELETE または TRUNCATE TABLE のすぐ後に実行してください。これらの文を実行すると、削除処理中にロールバック・ログとチェックポイント・ログのために割り当てられた、データベース・ファイルの余分な領域を、データベース・エンジンが再利用または解放できる可能性が高くなります。高速バージョンの TRUNCATE TABLE の場合、明示的な COMMIT は不要で、もし実行したとしても問題はありません。また、高速と低速のどちらのバージョンになるのか予測が難しい場合もあります。明示的な CHECKPOINT についても同じことが言えます。CHECKPOINT を実行するタイミングはデータベース・エンジンが自分で判断できるため、明示的な CHECKPOINT は不要です。ただし、CHECKPOINT を余計に実行しても問題が起きることはありません。

Note

CHECKPOINT 文によって負荷が高くなる可能性があります。通常、アプリケーション・プログラムに明示的な CHECKPOINT 文を追加する必要はありません。なぜなら、パフォーマンスへの影響を最小限に抑えるように、サーバがチェックポイントを効率的にスケジュールするからです。明示的な CHECKPOINT は、特に負荷の高いマルチユーザ環境では、十分に検討したうえで使用してください。

次の表は、子テーブルにローが存在するかどうか、TRUNCATE_WITH_AUTO_COMMIT が 'ON' か 'OFF' か、データベース・トランザクションが既に進行中かどうかによって、TRUNCATE TABLE の動作がどのように変わるかを示しています。8 とおりの組み合わせの中で、TRUNCATE TABLE が高速バージョンで使用されるのは 2 つだけです。また、TRUNCATE_WITH_AUTO_COMMIT が 'ON' であっても、コミットが実行されない組み合わせが 2 とおりあります

```
子テーブ
ルにロー   TRUNCATE_WITH  トランザクシ
が存在      _AUTO_COMMIT  ョンが進行中          TRUNCATE TABLE の動作
=========  =============  ============  ========================================
   Yes         'ON'           Yes                          slow TRUNCATE
   Yes         'ON'           No              BEGIN TRAN, slow TRUNCATE
   Yes         'OFF'          Yes                          slow TRUNCATE
   Yes         'OFF'          No              BEGIN TRAN, slow TRUNCATE
   No          'ON'           Yes      COMMIT, BEGIN TRAN, fast TRUNCATE, COMMIT
```

```
   No          'ON'          No              BEGIN TRAN, fast TRUNCATE, COMMIT
   No          'OFF'         Yes                         slow TRUNCATE
   No          'OFF'         No              BEGIN TRAN, slow TRUNCATE
```

Note

本書では、CHAINED データベース・オプションが 'ON' にセットされていることを前提としています。BEGIN TRAN（BEGIN TRANsaction の省略形）の動作が表のようになるのはこのためです。連鎖モードの動作では、INSERT、UPDATE、DELETE、および TRUNCATE TABLE などのデータ操作によって、データベース・トランザクションが（開始されていなければ）自動的に開始されます。通常、そのトランザクションは COMMIT または ROLLBACK が明示的に実行されるまで終了しません。CREATE TABLE や高速バージョンの TRUNCATE TABLE など一部のコマンドでは、COMMIT が副次的に実行されます。トランザクションの詳細については、9.3 項を参照してください。

TRUNCATE TABLE の動作を示す例を次にいくつか紹介します。最初にテーブルを 2 つ作成し、それぞれにローを 1 つずつ挿入します。

```
CREATE TABLE t1 (
   key_1         UNSIGNED INTEGER NOT NULL PRIMARY KEY,
   non_key_1     INTEGER NOT NULL );

CREATE TABLE t2 (
   key_1         UNSIGNED INTEGER NOT NULL PRIMARY KEY,
   non_key_1     INTEGER NOT NULL );

INSERT t1 VALUES ( 1, 1 );
INSERT t2 VALUES ( 22, 22 );
COMMIT;
```

最初のテストでは、TRUNCATE_WITH_AUTO_COMMIT を明示的に 'ON' にセットします。テーブル t2 のローを更新し、テーブル t1 に対して TRUNCATE TABLE を実行してから、ROLLBACK 文を実行します。

```
SET EXISTING OPTION PUBLIC.TRUNCATE_WITH_AUTO_COMMIT = 'ON';
UPDATE t2 SET non_key_1 = 999;
TRUNCATE TABLE t1;
ROLLBACK;
```

これらの文の実行後、t1 は空になり、t2.non_key_1 の値は 999 になります。TRUNCATE TABLE の実行前と実行後に COMMIT が処理され、ROLLBACK 文は完全に無視されます。トランザクション・ログに記録された、これらの処理のエントリを次に示します。

```
BEGIN TRANSACTION
UPDATE DBA.t2
   SET non_key_1=999
 WHERE key_1=22
COMMIT WORK
BEGIN TRANSACTION

truncate table t1
COMMIT WORK
```

TRUNCATE_WITH_AUTO_COMMIT が'OFF'の場合、結果はまったく異なります。UPDATE 文と TRUNCATE TABLE 文によって加えられた変更が ROLLBACK によって元に戻され、2つのテーブルには元のローが含まれます。

```
SET EXISTING OPTION PUBLIC.TRUNCATE_WITH_AUTO_COMMIT = 'OFF';
UPDATE t2 SET non_key_1 = 999;
TRUNCATE TABLE t1;
ROLLBACK;
```

TRUNCATE_WITH_AUTO_COMMIT を'OFF'にセットした場合のトランザクション・ログは、次のようになります。

```
BEGIN TRANSACTION
UPDATE DBA.t2
   SET non_key_1=999
 WHERE key_1=22
truncate table t1
ROLLBACK WORK
```

TRUNCATE TABLE が DELETE より高速なのは、すべてのローを削除するときだけではありません。多数のローを削除し、一部を残したい場合にも短時間で処理できます。それには、3つのステップから成る方法を用います。まず、残したいローをテンポラリ・テーブルにコピーします。次に、元のテーブルをトランケートし、最後にコピーしておいたローを元に戻します。

サイズが160M あり、100,000 ローを含むテーブルを使用して、TRUNCATE TABLE と DELETE を比較する例を紹介します。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL PRIMARY KEY,
   inserted_date DATE NOT NULL DEFAULT CURRENT DATE,
   blob          LONG VARCHAR );
```

次の集合指向型 DELETE では、99.9% のローを削除するのにおよそ1分かかりました。

```
DELETE t1
 WHERE inserted_date < DATEADD ( DAY, -7, CURRENT DATE );
```

次の3つの文を使うと、まったく同じ処理が半分以下の時間（27秒）で実行されました。

```
SELECT *
  INTO #t1
  FROM t1
 WHERE inserted_date >= DATEADD ( DAY, -7, CURRENT DATE );

TRUNCATE TABLE t1;

INSERT t1
SELECT *
  FROM #t1;
```

Note

先の例で、TRUNCATE TABLEの直後（ただし、最後の「INSERT t1」が終了し、COMMITが実行される前）にサーバがクラッシュした場合、残したいローをリカバリするためには、バックアップからデータベースをリストアする必要があります。なぜなら、残したいローはテンポラリ・テーブルにしか存在しないため、リカバリしても戻らないからです。

テンポラリ・テーブルの作成とデータ入力に使用するSELECT INTOの詳細については、「1.15.2.3 SELECT INTO #table_name」を参照してください。INSERTを使用して、データをあるテーブルから別のテーブルにコピーする方法については、「2.2.3　selectを使用したすべてのカラムへのINSERT」を参照してください。

Note

本書で紹介するパフォーマンス・テストは、ベンチマークに相当するほど厳密なものではなく、異なる手法をそれなりに公平に比較することを目的としています。たとえば前述のテストは、Windows 2000が動作する933MHzのIntel CPUで512Mのキャッシュを使って実行しています。また、公平さを保つため、各テストの実行前にsa_flush_cacheプロシージャを呼び出しました【監注4】。

【監注4】このシステム・プロシージャにより、サーバ上のキャッシュをクリアできる。ベンチマーク結果がキャッシュの状態に依存しないようにしている。

Section

5.7 この章のまとめ

この章では、1つまたは複数のローを単一のテーブルから削除する、単純なDELETE文の記述方法を説明しました。また、複数のテーブルのジョインを行うDELETEの動作についても解説しました。集合指向型のDELETEの構文全体を説明して、カーソル指向型のDELETE WHERE CURRENT OFについて取り上げた後、TRUNCATE TABLE文について説明しました。

次の章では、SQLで作成するアプリケーション・ロジックへと話題を移し、カーソルFETCHループについて説明します。

Chapter

6

フェッチ

Section

6.1 概要

この章では、最初に、カーソルのDECLARE文、OPEN文、FETCH文、およびCLOSE文と、DELETE WHERE CURRENT OFから成るカーソル・ループの例を示します。これには、SQLを使った例と、C言語でEmbedded SQLを使った例の両方を示し、処理の内容を順を追って説明します。

続く5つの項では、カーソルのDECLARE文の3つの形式とその構文について説明し、その後、OPEN、CLOSE、FETCHの各文について説明します。最後の項では、カーソルのFORループについて説明します。FORループを使用すると、プログラミングを簡略化することができます。

Section

6.2 カーソルのFETCHループ

「カーソル・ループ」は、複数のローから成る結果セットを、一度に1つのローずつ処理するためのしくみです。カーソルのタイプに応じて、前方や後方の1つ以上のローに移動すること、特定の位置のローに移動すること、現在のローを更新または削除することができます。カーソル・ループは、アプリケーション・プログラムでよく使われます。コードで明示的に使われることもあれば、プログラミング環境で暗黙的に使われることもあります。たとえば、PowerBuilder DataWindowのRetrieve関数への呼び出しは1つの操作のように見えますが、背後でカーソル・ループを使用してDataWindowバッファにデータを取り込んでいます。

カーソル・ループは、SQLのストアド・プロシージャやその他のSQLプログラミング・ブロック内にコーディングすることもできます。変数のDECLARE文、WHILEループ、カーソルのDECLARE、OPEN、FETCH、CLOSEの各文など、さまざまな種類のSQL文を使って作成します。次に示すのは、標準的なSQLカーソル・ループの例です。この例は短くて簡単ではあるものの、実用性にも優れています。具体的な処理内容としては、古いローをテーブルから削除します。その際、1回の実行で削除されるローの数を1,000に制限し、100ローを削除するたびにCOMMITを実行します。

```
BEGIN
DECLARE @key_1          INTEGER;
DECLARE @non_key_1      VARCHAR ( 100 );
DECLARE @last_updated   TIMESTAMP;
DECLARE @SQLSTATE       VARCHAR ( 5 );
DECLARE @loop_counter   INTEGER;

DECLARE c_fetch NO SCROLL CURSOR FOR
```

```
SELECT TOP 1000
       t1.key_1,
       t1.non_key_1,
       t1.last_updated
  FROM t1
 WHERE t1.last_updated < DATEADD ( MONTH, -6, CURRENT DATE )
 ORDER BY t1.last_updated
FOR UPDATE;

OPEN c_fetch WITH HOLD;

FETCH c_fetch INTO
   @key_1,
   @non_key_1,
   @last_updated;
SET @SQLSTATE = SQLSTATE;

SET @loop_counter = 0;

WHILE @SQLSTATE = '00000' LOOP

   SET @loop_counter = @loop_counter + 1;
   MESSAGE STRING ( 'Deleting ',
                    @loop_counter, ', ',
                    @key_1, ', "',
                    @non_key_1, '", ',
                    @last_updated ) TO CONSOLE;

   DELETE t1 WHERE CURRENT OF c_fetch;
   IF MOD ( @loop_counter, 100 ) = 0 THEN
      COMMIT;
      MESSAGE STRING ( 'COMMIT after ', @loop_counter, ' rows.') TO CONSOLE;
   END IF;

   FETCH c_fetch INTO
      @key_1,
      @non_key_1,
      @last_updated;
   SET @SQLSTATE = SQLSTATE;

END LOOP;
CLOSE c_fetch;
COMMIT;
MESSAGE STRING ( 'Final COMMIT after ', @loop_counter, ' rows.') TO CONSOLE;
END;
```

この例で、最初の3つのローカル変数（@key_1、@non_key_1、@last_updated）は、FETCH文を介してカーソルのSELECTによって返されるカラム値を受け取るのに必要です。@SQLSTATE変数は、

現在の実行状況をチェックするために使用し、@loop_counter は、COMMIT を実行するタイミングを判断するために使用します。

カーソルの DECLARE 文では、カーソルに c_fetch という名前を付け、NO SCROLL キーワードを使用して、コードが結果セットの後方に移動しないことを指定しています。これにより、ある種の最適化を SQL Anywhere が自由に行えるようになります。SELECT は、6 か月以上経過しているローを取得し、最も古いローが先頭になるようにソートし、返されるローの数を 1,000 に制限しています。FOR UPDATE キーワードは、取得されるローが変更可能であることを指定します（この例では、ローは削除されます）。

OPEN 文は、カーソルの DECLARE で定義された SELECT を実際に実行し、処理を開始します。WITH HOLD キーワードは、COMMIT の実行時にカーソルを暗黙的に閉じずに、開いたままにすることを指定します。

最初の FETCH 文は、結果セットの最初のローを取得し、カラム値を 3 つのローカル変数に代入します。それに続く SET 文で、SQLSTATE の値をローカル変数 @SQLSTATE に代入しています。SQLSTATE は多くの SQL 文によって変更されますが、このコードでは FETCH によって設定される値だけが重要なので、このような代入を行うのは良い方法です。

WHILE 文はループを開始し、ローがなくなるまでループを実行します。ローがなくなると、@SQLSTATE に '02000' が格納されます。ループ内の最初の MESSAGE 文は、現在のローを表示します。

DELETE 文は、現在のローを削除します。DELETE WHERE CURRENT OF カーソル文の詳細については、5.5 項を参照してください。

DELETE の後ろの IF 文は、MOD 関数を使用して、100 ローの倍数に達したタイミングを判断します。MOD は最初のパラメータを 2 番目のパラメータで除算し、その余りを返します。最初のパラメータが 2 番目のパラメータで割り切れると余りは 0 になるので、@loop_counter が 100、200、300 などのときに「MOD (@loop_counter, 100) = 0」になります。

次の FETCH 文は、2 番目以降のローを返し、3 つのローカル変数に新しいカラム値を代入します。最終的に、この FETCH は SQLSTATE を、「ローが見つかりません」を意味する '02000' に設定します。ループが終了すると、カーソルが閉じ、最後の COMMIT 文と MESSAGE 文が実行されます。

このカーソル・ループによる MESSAGE 出力の最後の数行は次のようになります。

```
Deleting 998, 9003, "", 1979-05-11 10:04:07.389
Deleting 999, 9002, "", 1979-05-12 10:04:07.389
Deleting 1000, 9001, "", 1979-05-13 10:04:07.389
COMMIT after 1000 rows.
Final COMMIT after 1000 rows.
```

次も同じループですが、こちらは Embedded SQL を使用したスタンドアロンの C プログラムとして作成しています。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sqldef.h"
EXEC SQL INCLUDE SQLCA;
int main() {

EXEC SQL BEGIN DECLARE SECTION;
long   key_1;
char   non_key_1 [ 101 ];
char   last_updated [ 24 ];
EXEC SQL END DECLARE SECTION;

char   copy_SQLSTATE [ 6 ];
long   loop_counter;
ldiv_t loop_counter_ldiv;

db_init( &sqlca );
EXEC SQL CONNECT USING 'ENG=test6;DBN=test6;UID=DBA;PWD=SQL';

EXEC SQL DECLARE c_fetch NO SCROLL CURSOR FOR
SELECT TOP 1000
       t1.key_1,
       t1.non_key_1,
       DATEFORMAT ( t1.last_updated, 'yyyy-mm-dd hh:nn:ss.sss' )
  FROM t1
 WHERE t1.last_updated < DATEADD ( MONTH, -6, CURRENT DATE )
 ORDER BY t1.last_updated
FOR UPDATE;

EXEC SQL OPEN c_fetch WITH HOLD;

EXEC SQL FETCH c_fetch INTO
   :key_1,
   :non_key_1,
   :last_updated;
strcpy ( copy_SQLSTATE, SQLSTATE );

loop_counter = 0;

while ( strcmp ( copy_SQLSTATE, "00000" ) == 0 ) {

   loop_counter = loop_counter + 1;
   printf ( "Deleting %d, %d, '%s', %s¥n",
      loop_counter,
      key_1,
      non_key_1,
      last_updated );
```

```
   EXEC SQL DELETE t1 WHERE CURRENT OF c_fetch;
   loop_counter_ldiv = ldiv ( loop_counter, 100L );
   if ( loop_counter_ldiv.rem == 0 ) {
      EXEC SQL COMMIT;
      printf ( "COMMIT after %d rows.¥n", loop_counter );
   }

   EXEC SQL FETCH c_fetch INTO
      :key_1,
      :non_key_1,
      :last_updated;
   strcpy ( copy_SQLSTATE, SQLSTATE );

} // while...
EXEC SQL CLOSE c_fetch;
EXEC SQL COMMIT;
EXEC SQL DISCONNECT;
db_fini ( &sqlca );
printf ( "Done after %d rows.¥n", loop_counter );
return ( 0 );
} // main
```

Note

本書では、Embedded SQL について特に詳しい説明をしません。上記の例を示したのは、カーソルの FETCH ループが、さまざまな形式の Embedded SQL 文を使用するアプリケーションで非常によく使われており、C のコードは PowerBuilder を含む他の開発環境で使われる Embedded SQL 構文の代表例だからです。

次の項では、カーソルの FETCH ループに指定する各コンポーネントの構文について詳しく説明します。

6.2.1 DECLARE CURSOR FOR select

カーソルは、select として、select を含む文字列変数を参照する USING 句として、またはプロシージャ CALL（コール）として定義できます。

```
<declare_cursor> ::= <declare_cursor_for_select>
                   | <declare_cursor_using_select>
                   | <declare_cursor_for_call>
```

次に示すのは、最初の形式の構文です。

```
<declare_cursor_for_select> ::= DECLARE <cursor_for_select>
<cursor_for_select> ::= <cursor_name>
                          [ <cursor_type> ]
                          CURSOR FOR
                          <select>
<cursor_name>       ::= カーソルのDECLAREコマンドまたはFORコマンドで定義した<identifier>
<identifier>        ::=「第1章　作成」の<identifier>を参照
<cursor_type>       ::= NO SCROLL      -- asensitive
                      | DYNAMIC SCROLL -- asensitive; デフォルト
                      | SCROLL         -- value-sensitive（キーセット駆動型）
                      | INSENSITIVE    -- insensitive
                      | SENSITIVE      -- sensitive
<select>            ::= [ <with_clause> ]
                          <query_expression>
                          [ <order_by_clause> ]
                          [ <for_intent_clause> ]
<for_intent_clause> ::= FOR READ ONLY
                      | FOR UPDATE
<with_clause>       ::=「第3章　選択」の<with_clause>を参照
<query_expression>  ::=「第3章　選択」の<query_expression>を参照
<order_by_clause>   ::=「第3章　選択」の<order_by_clause>を参照
```

カーソルのDECLAREに指定する各句は、カーソルのライフサイクルのうち、主に2つの段階を制御します。WITH句、クエリ式、およびORDER BY句は、OPEN文が実行されたときに、カーソル結果セットをどのように表示するかを指定します。<cursor_type>と<for_intent_clause>は、その後のカーソル・ループでフェッチおよび処理されたときに、結果セットがどのように動作するかを指定します。

カーソルのDECLARE文には、実行可能な動作を指定する多くの要素がありますが、DECLARE文自体は実行可能な文ではありません。カーソルの各DECLARE文は、BEGINブロックのはじめに記述し、かつすべての実行可能な文の前に記述する必要があります。1つのブロック内で複数のカーソルを宣言して使用することができますが、個々のカーソル名はそのブロック内でユニークにしなければなりません。

WITH句、クエリ式、およびORDER BY句については、「第3章　選択」を参照してください。

<cursor_type>は、次に示す3つのカーソル属性のデフォルトを間接的に指定します。

- スクロール動作は、ローをフェッチする順序を制御する。具体的には、後のローをフェッチした後で、それより前のローを再びフェッチできるかどうかを制御する
- 更新可能性は、現在のカーソルでUPDATE WHERE CURRENT OF文とDELETE WHERE CURRENT OF文を使用できるかどうか、およびEmbedded SQLでPUT文を使用できるかどうかを制御する。カーソルの名前を参照せずに、基本となるテーブルを直接操作するUPDATE文、DELETE文、およびINSERT文は、カーソルが更新可能かどうかにかかわらず常に使用できる
- 感知性は、カーソルの結果セットをフェッチ中、基となるテーブルに変更が加えられた場合に、それをカーソルの結果セット自体に反映させるかどうかを制御する。カーソルの感知性は、他の

接続によって加えられた変更だけでなく、現在のカーソル自体に適用したUPDATE WHERE CURRENT OF文、DELETE WHERE CURRENT OF文、およびPUT文による変更にも適用される

カーソルの感知性は、カーソル・タイプの最も複雑な属性ですが、次のような定義によって表すことができます。

- カーソルは、ある種類の変更に関してsensitive（感知）になり、別の種類の変更に関してinsensitive（非感知）になることがある
- 「メンバシップの感知性」は、カーソルのWHERE句で指定されたカラムの値に変更があった場合に、結果セットにローを表示したり、結果セットからローを消去したりするかどうかを制御する
- 「順序の感知性」は、ORDER BY句のカラムに変更があった場合に、ローを結果セットの別の位置に移動し、元の位置に空きを残すかどうかを制御する
- 「値の感知性」は、カラム値自体に変更があった場合に、カーソル結果セットのローに変更を反映するかどうかを制御する
- 「削除の感知性」は、基のローが削除された場合に、結果セットで削除を反映するかどうか、およびどのように反映するかを制御する
- 「sensitiveカーソル」の結果セットでは、フェッチされたすべてのローがカーソルのWHERE句およびORDER BY句に適合し、カラム値が常に基のテーブルと一致する。UPDATEがWHERE句およびORDER BY句で指定されたカラムに影響を与える場合は、UPDATEによって結果セットのローが表示されたり消えたり、位置が変わったりすることがある。DELETEでは、ローがまったく存在しなかったかのように結果セットから消去される
- ISOLATION_LEVEL接続オプションの設定値が高いほど、感知性が高まる。たとえば、独立性レベル3で動作しているsensitiveカーソルでは、他の接続による変更ができないようにするロックを取得し、変更がカーソルの結果セットに反映されないようにすることができる。これについては、「9.7　ブロックと独立性レベル」で詳しく説明する
- 「value-sensitive」（キーセット駆動型）カーソルは、メンバシップと順序に関してはinsensitiveで、値と削除に関してはsensitiveである。WHERE句のカラムに影響を与えるUPDATEは、たとえWHERE句が既にそのローについてTRUEと評価されなくなっても、既にフェッチされているローのメンバシップには影響を与えない。また、ORDER BY句のカラムに影響を与えるUPDATEによって、ローが別の位置に移動することもない。ただし、どちらの場合もローを再びフェッチすると、変更されたカラム値が反映される。

  value-sensitiveカーソルの動作に関しては、2つの側面がある。まず、DELETEによって結果セットに空きができ、そのローを再びフェッチしようとすると、エラーSQLSTATE 24503「カーソルの現在のローがありません」が発生する。ただし、カーソルは開いたままで、それ以降のフェッチは処理される。したがって、これはカーソルの処理を停止させない唯一のエラー条件である。また、value-sensitiveカーソルは、UPDATEがプライマリ・キー・カラムを変更する場合、ローのメン

バシップに関してsensitiveである。理由は、この操作がDELETEとそれに続くINSERTとして扱われるからである

- 「insensitiveカーソル」は、メンバシップ、順序、値、および削除に関してinsensitiveである。実際には、カーソルを開いたときに、結果セット全体のコピーが一時的に作成される。それ以降、基となるテーブルに変更があっても、カーソルの結果セットには反映されない
- 「asensitiveカーソル」には、メンバシップ、順序、値、および削除の感知性に関して未定義の動作がある。SQL Anywhereは、感知性にかかわらず、最も効率的なカーソルの実行方法を自由に選択できる

次に、5つのカーソル・タイプによって、スクロール動作、更新可能性、感知性の3つのカーソル属性のデフォルトがどのように指定されるかを示します。

- **NO SCROLLカーソル**――後方へのスクロールができない。FETCH NEXT、FETCH RELATIVE 0、およびFETCH RELATIVE 1の操作のみが可能。このカーソルは、デフォルトで更新可能で、asensitiveである。
- **DYNAMIC SCROLLカーソル**――すべての形式のスクロールが可能。デフォルトで更新可能で、asensitive。このカーソルは、デフォルトのカーソル・タイプである。
- **SCROLLカーソル**――すべての形式のスクロールが可能。デフォルトで更新可能で、value-sensitiveである。
- **INSENSITIVEカーソル**――すべての形式のスクロールが可能。デフォルトで読み込み専用で、insensitiveである。
- **SENSITIVEカーソル**――すべての形式のスクロールが可能。デフォルトで更新可能で、sensitiveである。

<for_intent_clause>は、現在のカーソルで、UPDATE WHERE CURRENT OF、DELETE WHERE CURRENT OF、およびPUT文による変更を可能にするかどうかを制御します。FOR READ ONLYの場合、たとえ宣言されたカーソル・タイプが更新可能を意味するものでも（DYNAMIC SCROLLなど）、カーソルは更新不可です。同様に、FOR UPDATEの場合、たとえ宣言されたカーソル・タイプが読み込み専用を意味するものでも（INSENSITIVEなど）、カーソルは更新可能です。<for_intent_clause>はカーソル・タイプに優先します。たとえば、INSENSITIVEとして宣言されたカーソルでも、FOR UPDATE句が指定されていれば、実際にはvalue-sensitiveカーソルとして実装されます。

Tip

最も効率的なカーソルの種類は、FOR READ ONLYを指定したNO SCROLLおよびDYNAMIC SCROLLです。

カーソルのDECLARE文で行うホスト変数の代入は、カーソルのDECLAREが含まれるブロックに入るときに変数が存在し、その変数に値が格納されていれば可能です。これは、外部ブロックで変数を宣言および初期化し、内部ブロック内にカーソルのDECLAREを記述したネストされたBEGINブロックを使って行うことができます。また、ストアド・プロシージャを使って行うこともできます。次に示すのは、パラメータ値への参照を含む、カーソルのDECLAREを定義したプロシージャの例です。次のプロシージャとCALL文は、前述の「6.2　カーソルのFETCHループ」で示した例と同じ処理を実行します。

```
CREATE PROCEDURE p_delete_oldest ( IN @age_in_months INTEGER )
BEGIN
DECLARE @key_1          INTEGER;
DECLARE @non_key_1      VARCHAR ( 100 );
DECLARE @last_updated   TIMESTAMP;
DECLARE @SQLSTATE       VARCHAR ( 5 );
DECLARE @loop_counter   INTEGER;

DECLARE c_fetch NO SCROLL CURSOR FOR
SELECT TOP 1000
       t1.key_1,
       t1.non_key_1,
       t1.last_updated
  FROM t1
 WHERE t1.last_updated < DATEADD ( MONTH, -@age_in_months, CURRENT DATE )
 ORDER BY t1.last_updated
FOR UPDATE;

OPEN c_fetch WITH HOLD;

FETCH c_fetch INTO
   @key_1,
   @non_key_1,
   @last_updated;
SET @SQLSTATE = SQLSTATE;

SET @loop_counter = 0;

WHILE @SQLSTATE = '00000' LOOP
   SET @loop_counter = @loop_counter + 1;
   DELETE t1 WHERE CURRENT OF c_fetch;
   IF MOD ( @loop_counter, 100 ) = 0 THEN
      COMMIT;
   END IF;
   FETCH c_fetch INTO
      @key_1,
      @non_key_1,
      @last_updated;
   SET @SQLSTATE = SQLSTATE;
```

```
END LOOP;
CLOSE c_fetch;
COMMIT;
END; -- p_delete_oldest

CALL p_delete_oldest ( 6 );
```

カーソルのDECLAREでは、WITH句、複数のselect、UNIONなどの演算子をはじめ、「第3章　選択」で説明した機能を使用するクエリを指定できます。次に示すのは、「3.24.1　再帰ユニオン」の最初の例を基にしたカーソルのFETCHループです。このクエリは、「組織図上でAinslieに至るまで、Marlonの上司にはだれがいるか」を調べます。出力は3.24.1項で示したものと同じです。

```
BEGIN
DECLARE @level         INTEGER;
DECLARE @name          VARCHAR ( 20 );
DECLARE @SQLSTATE      VARCHAR ( 5 );
DECLARE @loop_counter  INTEGER;

DECLARE c_fetch NO SCROLL CURSOR FOR
WITH RECURSIVE superior_list
   ( level,
     chosen_employee_id,
     manager_id,
     employee_id,
     name )
AS ( SELECT CAST ( 1 AS INTEGER )   AS level,
            employee.employee_id    AS chosen_employee_id,
            employee.manager_id     AS manager_id,
            employee.employee_id    AS employee_id,
            employee.name           AS name
       FROM employee
     UNION ALL
     SELECT superior_list.level + 1,
            superior_list.chosen_employee_id,
            employee.manager_id,
            employee.employee_id,
            employee.name
       FROM superior_list
            INNER JOIN employee
                    ON employee.employee_id = superior_list.manager_id
      WHERE superior_list.level <= 99
        AND superior_list.manager_id <> superior_list.employee_id )
SELECT superior_list.level,
       superior_list.name
  FROM superior_list
 WHERE superior_list.chosen_employee_id = 13
 ORDER BY superior_list.level DESC
```

```
FOR READ ONLY;

OPEN c_fetch WITH HOLD;
FETCH c_fetch INTO
   @level,
   @name;
SET @SQLSTATE = SQLSTATE;
SET @loop_counter = 0;
WHILE @SQLSTATE = '00000' LOOP
   SET @loop_counter = @loop_counter + 1;
   MESSAGE STRING ( @level, ' ', @name ) TO CONSOLE;
   FETCH c_fetch INTO
      @level,
      @name;
   SET @SQLSTATE = SQLSTATE;
END LOOP;
CLOSE c_fetch;
END;
```

6.2.2 DECLARE CURSOR USING select

カーソルに使用するクエリを文字列変数に格納し、その変数をカーソルのDECLAREのUSINGキーワードの後ろに指定することができます。

```
<declare_cursor_using_select> ::= DECLARE <cursor_using_select>
<cursor_using_select>         ::= <cursor_name>
                                      [ <cursor_type> ]
                                      CURSOR USING
                                      <cursor_select_variable>
<cursor_select_variable>      ::= <select>を含む文字列<identifier>
```

次に示すのは、前述の「6.2　カーソルのFETCHループ」で示した例を、SELECTを含む変数を使うように変更したものです。文字列変数@selectを宣言して初期化するために、外部のBEGINブロックを追加しています。FOR UPDATE句は、外部のカーソルDECLARE文ではなく、カーソルのselectの一部なので、文字列値に含まれています。

```
BEGIN -- 外部ブロック
DECLARE @select LONG VARCHAR;
SET @select = '
SELECT TOP 1000
       t1.key_1,
       t1.non_key_1,
       t1.last_updated
  FROM t1
 WHERE t1.last_updated < DATEADD ( MONTH, -6, CURRENT DATE )
```

```
 ORDER BY t1.last_updated
FOR UPDATE';

BEGIN -- 内部ブロック
DECLARE @key_1          INTEGER;
DECLARE @non_key_1      VARCHAR ( 100 );
DECLARE @last_updated   TIMESTAMP;
DECLARE @SQLSTATE       VARCHAR ( 5 );
DECLARE @loop_counter   INTEGER;

DECLARE c_fetch NO SCROLL CURSOR USING @select;

OPEN c_fetch WITH HOLD;
FETCH c_fetch INTO
   @key_1,
   @non_key_1,
   @last_updated;
SET @SQLSTATE = SQLSTATE;
SET @loop_counter = 0;
WHILE @SQLSTATE = '00000' LOOP
   SET @loop_counter = @loop_counter + 1;
   DELETE t1 WHERE CURRENT OF c_fetch;
   IF MOD ( @loop_counter, 100 ) = 0 THEN
      COMMIT;
      MESSAGE STRING ( 'COMMIT after ', @loop_counter, ' rows.') TO CONSOLE;
   END IF;
   FETCH c_fetch INTO
      @key_1,
      @non_key_1,
      @last_updated;
   SET @SQLSTATE = SQLSTATE;
END LOOP;
CLOSE c_fetch;
COMMIT;
END; -- 内部ブロック
END; -- 外部ブロック
```

USING 句を使うと、カーソルの select 全体を動的に作成できます。この方法はストアド・プロシージャ内で特に有用で、テーブル名、カラム名、WHERE 句などのさまざまなコンポーネントをパラメータとして渡すことができます。

6.2.3 DECLARE CURSOR FOR CALL

カーソルの DECLARE で、SELECT の代わりにプロシージャの CALL を指定することができます。この形式のカーソルは、暗黙的に読み込み専用です。FOR UPDATE 句は使用できません。

```
<declare_cursor_for_call> ::= DECLARE <cursor_for_call>
<cursor_for_call>  ::= <cursor_name>
                         [ <cursor_type> ]
                         CURSOR FOR CALL [ <owner_name> "."] <procedure_name>
                         [ <argument_list> ]
<owner_name>       ::= <identifier>
<procedure_name>   ::= <identifier>
<argument_list>    ::= { <argument> "," } <argument>
<argument>         ::= <basic_expression>
                     | <parameter_name> "=" <basic_expression>
<basic_expression> ::=「第3章　選択」の<basic_expression>を参照
                         -- サブクエリではない式
<subquery>         ::=「第3章　選択」の<subquery>を参照
<parameter_name>   ::= プロシージャ内でパラメータとして定義した<identifier>
```

ここでも「6.2　カーソルのFETCHループ」の例を使用します。今回はプロシージャのCALLを使用します。DELETE WHERE CURRENT OFを通常のDELETE文に変更し、WHERE句でプライマリ・キーの値を明示します。カーソルが更新不可だからといって、更新が不可能というわけではないのです。

```
CREATE PROCEDURE p_oldest ( IN @age_in_months INTEGER )
BEGIN
SELECT TOP 1000
       t1.key_1,
       t1.non_key_1,
       t1.last_updated
  FROM t1
 WHERE t1.last_updated < DATEADD ( MONTH, -@age_in_months, CURRENT DATE )
 ORDER BY t1.last_updated;
END;

BEGIN
DECLARE @key_1         INTEGER;
DECLARE @non_key_1     VARCHAR ( 100 );
DECLARE @last_updated  TIMESTAMP;
DECLARE @SQLSTATE      VARCHAR ( 5 );
DECLARE @loop_counter  INTEGER;

DECLARE c_fetch NO SCROLL CURSOR FOR CALL p_oldest ( 6 );

OPEN c_fetch WITH HOLD;

FETCH c_fetch INTO
   @key_1,
   @non_key_1,
   @last_updated;
SET @SQLSTATE = SQLSTATE;
```

```
SET @loop_counter = 0;

WHILE @SQLSTATE = '00000' LOOP
   SET @loop_counter = @loop_counter + 1;
   DELETE t1
    WHERE t1.key_1 = @key_1;
   IF MOD ( @loop_counter, 100 ) = 0 THEN
      COMMIT;
   END IF;
   FETCH c_fetch INTO
      @key_1,
      @non_key_1,
      @last_updated;
   SET @SQLSTATE = SQLSTATE;
END LOOP;
CLOSE c_fetch;
COMMIT;
END;
```

6.2.4 OPENとCLOSEカーソル

OPEN 文は、カーソルの DECLARE で定義したクエリを実際に実行します。CLOSE 文を使用すると、処理の完了後にカーソルを閉じることができます。

```
<open_cursor>     ::= OPEN <cursor_name>
                        [ WITH HOLD ]
                        [ <isolation_level> ]
<isolation_level> ::= ISOLATION LEVEL 0
                    | ISOLATION LEVEL 1 -- ダーティ・リードを防ぐ
                    | ISOLATION LEVEL 2 -- さらに繰り返し不可能読み出しも防ぐ
                    | ISOLATION LEVEL 3 -- さらに幻ロー も防ぐ
<close_cursor>    ::= CLOSE <cursor_name>
```

WITH HOLD 句を使用すると、COMMIT を発行してもカーソルが暗黙的に閉じられません。デフォルトでは、COMMIT 文は開いているカーソルをすべて閉じるため、それ以降の FETCH が失敗します。

ISOLATION LEVEL 句は、このカーソルにかかわるすべての操作の独立性レベルを設定します。これにより、ISOLATION_LEVEL 接続オプションの現在の設定が上書きされます。独立性レベルの詳細については、「9.7　ブロックと独立性レベル」を参照してください。

カーソルの OPEN 文は、エラーではなく警告として処理されるさまざまな例外条件を検出できます。そうした警告条件の 1 つは、SQLSTATE を '01S02' に設定します。これは、カーソルの 1 つ以上の属性が変更されており、カーソルの DECLARE で指定または暗示された属性との間に違いがあることを意味します。たとえば、DECLARE で FOR UPDATE と共に INSENSITIVE カーソル・タイプが使われているような場合です。デフォルトでは、OPEN 文がプロシージャまたは BEGIN ブロック内で実行されている場合、SQL Anywhere はこれらの警告条件を無視します。警告条件を検出するか、またはエラー

として処理するには、そのためのコードを追加する必要があります。次に、'00000' 以外の SQLSTATE をエラーにする OPEN 文と IF 文の例を示します。

```
OPEN c_fetch WITH HOLD;
IF SQLSTATE <> '00000' THEN
   RAISERROR 20000 STRING ( 'Cursor OPEN SQLSTATE = ', SQLSTATE )
END IF;
```

RAISERROR 文の詳細については、「9.5.2　RAISERROR と CREATE MESSAGE」を参照してください。

6.2.5　FETCHカーソル

FETCH 文は、カーソルの結果セットの特定位置に移動し、その位置にローが存在する場合はローからカラム値を取得し、その値をホスト変数に代入します。

```
<fetch_cursor>       ::= FETCH [ <cursor_positioning> ] <cursor_name>
                           INTO <fetch_into_list>
                           [ FOR UPDATE]
<cursor_positioning> ::= NEXT     -- デフォルト、RELATIVE 1と同じ
                       | FIRST    -- ABSOLUTE 1と同じ
                       | LAST     -- ABSOLUTE -1と同じ
                       | PRIOR    -- RELATIVE -1と同じ
                       | ABSOLUTE <move_to_row_number>
                       | RELATIVE <move_to_row_offset>
<move_to_row_number> ::= 正または負の数値の<simple_expression>
<move_to_row_offset> ::= 正または負の数値の<simple_expression>
<simple_expression>  ::=「第3章　選択」の<simple_expression>を参照
                        -- サブクエリではなく、かつIFまたはCASEで始まらないこと
<fetch_into_list> ::= { <non_temporary_identifier> "," } <non_temporary_identifier>
<non_temporary_identifier> ::=「第1章　作成」の<non_temporary_identifier>を参照
```

カーソルの位置は、結果セットの実際のローに対応する場合もしない場合もあります。カーソルを最初に開いた時点では、カーソルは最初のローの前の位置にあり、後で位置をそのポイントに戻すことができます。また、カーソルを最後のローの後の位置に移動することや、既に存在しないローが以前に占めていた位置に移動することもできます。FETCH が実際のローに対応しない位置に移動した場合、INTO 句は無視され、SQLSTATE は「ローが見つかりません」の警告値 '02000' に設定されます。

カーソル位置の各キーワードは、次のような働きをします。

- **NEXT** ――デフォルト値。カーソルの結果セットの次の位置に移動する。カーソルを最初に開いた時点では、最初のローの前の位置にカーソルがあるので、最初の FETCH NEXT 操作は最初のローに移動する。

- **FIRST** ——最初の位置に移動する。
- **LAST** ——最後の位置に移動する。
- **PRIOR** ——前の位置に移動する。
- **ABSOLUTE** ——指定した位置に移動する。最初のローは1、2番目のローは2のように指定する。FETCH ABSOLUTE 1はFETCH FIRSTと同じである。
- **RELATIVE** ——正の数であれば、その数だけ前方に移動し、負の数であれば、その数だけ後方に移動する。FETCH RELATIVE 1はFETCH NEXTと、FETCH RELATIVE -1はFETCH PRIORと同じである。

「最初のローの前」として扱われる位置は1つしかなく、「最後のローの後」として扱われる位置も1つしかありません。たとえば、カーソルに5つのローがある場合、「FETCH ABSOLUTE -999」は最初のローの前に移動し、それに続くFETCH NEXTは最初のローに移動します。同様に、「FETCH ABSOLUTE +999」に続くFETCH PRIORは最後のローに移動します。

カーソル・タイプによっては、FETCHを繰り返し、最後にフェッチした後で1つ以上のカラムが変更されたローを検出できるものがあります。これは警告として処理され、SQLSTATEが'01W04'に設定されます。

INTO句は、フェッチしたローからカラム値を受け取る1つ以上の変数を指定します。INTO句の変数のリストは、カーソルのDECLARE selectリストと数と順序が一致している必要があります。フェッチした位置にローが存在しない場合、変数は変更されません。

Section

6.3 カーソルのFORループ

FORループを使用すると、カーソル・ループのコーディングを簡略化することができます。具体的には、カーソルのDECLAREとWHILEループが1つのFOR文にまとまり、OPEN、CLOSE、FETCHの各文が不要になり、各ローからフェッチしたカラム値を受け取るローカル変数が暗黙的に定義されます。

FORループには2つの形式があります。LEAVE文のターゲットになるラベル付きの形式と、ラベルなしの形式です。

```
<for_cursor_loop>          ::= FOR <for_name>
                               AS <cursor_declaration>
                               DO
                                  <for_loop_body_statements>
                               END FOR
                             | <for_label> ":"
```

```
                              FOR <for_name>
                              AS <cursor_declaration>
                              DO
                                 <for_loop_body_statements>
                              END FOR [ <for_label> ]
<for_name>                 ::= <identifier>
<cursor_declaration>       ::= <cursor_for_select>
                             | <cursor_using_select>
                             | <cursor_for_call>
<for_loop_body_statements> ::= selectリストの各項目を名前で参照する文
<for_label>        ::= <leave_statement>で使用される<identifier>
<leave_statement>  ::=「第8章　パッケージ」の<leave_statement>を参照
```

次に示すのは、「6.2　カーソルのFETCHループ」の例を、DECLARE、OPEN、FETCH、およびWHILE文の代わりにFORループを使ってコーディングしたものです。

```
BEGIN
DECLARE @loop_counter  INTEGER;
SET @loop_counter = 0;
FOR f_fetch
AS c_fetch NO SCROLL CURSOR FOR
SELECT TOP 1000
       t1.key_1        AS @key_1,
       t1.non_key_1    AS @non_key_1,
       t1.last_updated AS @last_updated
  FROM t1
 WHERE t1.last_updated < DATEADD ( MONTH, -6, CURRENT DATE )
 ORDER BY t1.last_updated
FOR UPDATE
DO
   SET @loop_counter = @loop_counter + 1;
   MESSAGE STRING ( 'Deleting ',
                    @loop_counter, ', ',
                    @key_1, ', "',
                    @non_key_1, '", ',
                    @last_updated ) TO CONSOLE;
   DELETE t1 WHERE CURRENT OF c_fetch;
   IF MOD ( @loop_counter, 100 ) = 0 THEN
      COMMIT;
   END IF;
END FOR;
COMMIT;
END;
```

このコードでは、1つの変数だけが明示的に宣言されています。@loop_counterは、単にCOMMITを実行するタイミングを判断するために使用しており、カーソル処理の一部ではありません。他の3つの変数は、FOR文のカーソル定義によって暗黙的に作成されます。@key_1、@non_key_1、および

@last_updated は、SELECT リストのカラムから名前とデータ型を取得します。

Tip

FOR ループの SELECT リストでは、カラムに常にエイリアス名を指定し、それらのエイリアス名がカラム名自体と異なるようにしてください。SELECT リストの項目は、カーソルのローからフェッチされた値を格納するローカル変数を暗黙的に作成するために使われ、デフォルトでは、カラム名が変数名として使用されます。この場合、カーソル・ループ内で同じテーブルを参照する SQL 文を追加しようとすると問題が生じます。これらの文では、変数名への参照が、すべてカラム名への参照として解釈されるからです。前の例に示すように、変数とカラムの区別を明確にするには、"@" というプレフィクスを使うと便利です。

次に示すのは、「6.2.3　DECLARE CURSOR FOR CALL」のコードを FOR ループを使って簡略化したものです。この場合、@key_1 などのエイリアス名が絶対に必要です。エイリアス名 @key_1 を使わない場合、DELETE 文は「DELETE t1 WHERE t1.key_1 = key_1」のように記述しますが、DELETE のコンテキスト内では key_1 がカラム名として解釈されるので、t1 のすべてのローが削除されてしまいます。

```
CREATE PROCEDURE p_oldest ( IN @age_in_months INTEGER )
BEGIN
SELECT TOP 1000
       t1.key_1        AS @key_1,
       t1.non_key_1    AS @non_key_1,
       t1.last_updated AS @last_updated
  FROM t1
 WHERE t1.last_updated < DATEADD ( MONTH, -@age_in_months, CURRENT DATE )
 ORDER BY t1.last_updated;
END;

BEGIN
DECLARE @loop_counter  INTEGER;
SET @loop_counter = 0;
FOR f_fetch
AS c_fetch NO SCROLL CURSOR FOR CALL p_oldest ( 6 )
DO
   SET @loop_counter = @loop_counter + 1;
   MESSAGE STRING ( 'Deleting ',
                    @loop_counter, ', ',
                    @key_1, ', "',
                    @non_key_1, '", ',
                    @last_updated ) TO CONSOLE;
   DELETE t1
    WHERE t1.key_1 = @key_1;
   IF MOD ( @loop_counter, 100 ) = 0 THEN
```

```
      COMMIT;
   END IF;
END FOR;
COMMIT;
END;
```

FOR ループでは、文字列変数と USING 句も使用できます。たとえば、「6.2.2　DECLARE CURSOR USING select」で示した例を、FOR ループとして書き換えることができます。

Section

6.4 この章のまとめ

この章では、DECLARE、OPEN、FETCH、CLOSE の各文を使用して、カーソル・ループをコーディングする方法を説明しました。その中で、インライン・クエリ、クエリを含む文字列変数、およびプロシージャ・コールを使用する、DECLARE の複数の形式を具体例と共に示しました。また、それらの例を基にして、カーソルの FOR ループを使用すると SQL コードが簡略化されることも示しました。

次の章では話題を変えて、複数のデータベースへのデータの分散と、Mobile Link によるデータベースの同期について説明します。

Chapter

7

同期

Section

7.1 概要

データ分散とは、データの物理的な記憶領域を複数の異なるロケーションに用意し、そのロケーションに関係なく必要なデータへアクセスできるようにすることを言います。データ分散の中には、過去の経緯によって行われるものもあります。たとえば、複数のアプリケーションが、それぞれ異なるソフトウェアとハードウェアを使用する場合などです。あるいは、特定の目的のためにデータ分散を行うこともあります。たとえば、エンド・ユーザに近い場所にデータを移動してアクセスを高速化したり、ネットワーク障害が発生した場合の信頼性を向上したり、ネットワーク接続を利用できる時間が限られていることへ対処したりといった目的です。

SQL Anywhere Studio 9には、分散データを実装する方法として、プロキシ・テーブル、Replication Server、SQL Remote、Mobile Linkの4つが用意されています。

プロキシ・テーブルとは、異なるロケーションのデータに対し、あたかも単一のデータベースにすべて格納されているかのようにリアルタイムでアクセスできる機能です。この機能については、「1.14　リモート・データ・アクセス」で説明します。

Replication Server【監注5】とは、少数のデータベース間でリアルタイムに近いデータのコピーを実現する機能です。SQL Anywhere Studio 9には、Replication Serverを利用するために必要なコンポーネントは付属していますが、Replication Server自体は付属していませんので、本書では取り上げません。

SQL RemoteとMobile Linkはいずれも、単一の統合データベースと、事実上無制限の数のリモート・データベースとの間でデータを随時コピーするというしくみで動作します。いずれの製品も、SQL Anywhere Studio 9に付属しており、iAnywhere Solutionsによって完全にサポートされています。この章では、Mobile Linkについてのみ説明します。ただし、Mobile LinkとASAリモート・データベースという組み合わせでの使用についてのみです。Mobile Linkは、Ultra Lightデータベースと組み合わせて使用することもできますが、Ultra Lightに関する話題は本書では取り上げません。

SQL Remoteの機能について述べるときには「レプリケーション」という言葉が使用されるのに対し、Mobile Linkでは「同期（シンクロナイゼーション）」という言葉が使用されます。両者の違いは、Mobile Linkでは、統合データベースとリモート・データベースがデータ転送の時点で「同期のとれた状態」であることが保証されるのに対し、SQL Remoteでは、通信の遅延があるため、そうした保証がないという点です。

この章のタイトルが「レプリケート」や「分散」ではなく「同期」となっているのはそのためです。

【監注5】Replication Serverについては、サイベース社のWebサイト（http://www.sybase.co.jp/product/rep/）を参照。

Section

7.2 Mobile Linkのしくみ

Mobile Link ソフトウェアは、クライアントとサーバという2つの主要なコンポーネントで構成されます。同期セッションはクライアントから開始されます。クライアントはまず、リモート・データベースに接続し、前回の同期以降に変更されたすべてのローを取りまとめて、サーバにアップロードします。これを受けてサーバは、アップロードされた変更を統合データベースに反映し、ダウンロードするローを選択して、クライアントに送信します。これを受けてクライアントは、ダウンロードされた変更をリモート・データベースに反映し、受信確認をサーバに返送します。以上で完了です。

Mobile Link では、Mobile Link クライアント・コンポーネントがインストールされた数千のリモート・データベースが、1つの統合データベースと同期できます。

Mobile Link は、非常に強力かつ柔軟でありつつも、設定は特に難しいわけではありません。Mobile Link を有効に活用するためには、そのアーキテクチャを理解することが必要です。ここでは、Mobile Link アーキテクチャの主な特徴について説明することにします。

Mobile Link は、管理の一元化を実現できるように設計されています。Mobile Link のそもそもの設計は、リモート・データベースの実地メンテナンスを実施できる「リモート DBA」がいないモバイル環境向けを意図したものでした。その後、Mobile Link は、据え置き型の大規模なリモート・データベースでも使用されるようになっていますが、目指すところは変わっていません。つまり、管理機能の大半を統合データベースで行うということです。

Mobile Link がサポートするのは、連続的な操作ではなく、断続的な接続です。Mobile Link の同期は、データが入力されてから他のデータベースへ転送するまでの遅延時間や間隔が長いことが特徴です。同期の実行は1日に1～2回というのが一般的ですが、数分間隔ということも可能です。したがって、自動化されたスケジュールに組み込むことも可能です。ただし、Mobile Link は、連続的なリアルタイム・レプリケーションを行うことを意図したものではありません。

Mobile Link では、差分や変更箇所だけではなく、ロー全体が転送されます。同じローに対する複数の変更は、個々の更新としてではなく、そのローの最終的な単一のコピーとして転送されます。これは、アップロードとダウンロードの両方に当てはまりますが、その詳細は大きく異なります。

Mobile Link は、ボリュームの小さな同期が多数実行されるという状況に合わせて最適化されています。これも、モバイル環境向けを意図した結果によるものです。Mobile Link では、数百万件のローを1回の同期で転送することも可能であり、実際にそうした使い方もされていますが、それには代償が伴います。具体的には、アップロード・ストリーム全体とダウンロード・ストリーム全体が、それぞれ単一のトランザクションとして扱われ、単一のコミット操作で処理されるということです。大量の同期の場合、同時実行性の問題が両方の側で生じる可能性があり、たとえば、巨大なダウンロードがスタンドアロン・プロセスで実行されてしまうことがあります。

同期は、ファイル・ベースやメッセージ・ベースではなく、セッション・ベースの処理です。Mobile

Link ソフトウェアは、同期処理の際に、統合データベースとリモート・データベースの両方に接続し、すべての変更はセッション中に反映されます。いったん格納されたうえで後から反映されるのではありません。同期処理が完了する時点で、2つのデータベースは「同期のとれた状態」になります。

同期は、ピア・ツー・ピアではなく、階層型の処理です。あるリモート・データベースから他のリモート・データベースに変更を引き渡す場合には、まず1つのセッションで統合データベースにアップロードした後で、別のセッションで相手のリモート・データベースにダウンロードする必要があります。直接引き渡す形ではありません。

Mobile Link では、さまざまな種類のデータベースを統合データベースとして使用できます。ASA、Sybase ASE、Oracle、Microsoft SQL Server、IBM DB2 UDB はすべて正式にサポートされており、それ以外にも正常に使用できているソフトウェアがあります【監注6】。制約要因となるのは、ODBC ドライバの機能と、データベースのストアド・プロシージャ言語です。

Tip

プロキシ・テーブル機能では、Mobile Link に加えて、他のデータベース・アプリケーションもサポートされています。統合データベース・ソフトウェアで Mobile Link を直接使用できない場合は、ASA 統合データベースをセット・アップし、プロキシ・テーブルを使用して、他のデータベースとの間でデータを転送してください。この方法には制約があり、追加的なコーディングの手間も当然ながらかかります。本書ではその詳細については説明しません。それでも、他の種類のデータベースに対して同期の範囲を広げるためには、その方法を使用できます。

Mobile Link でリモート・データベースとして使用できるのは、ASA と Ultra Light のみです。本書では Ultra Light については取り上げませんので、ASA を使用するリモート・データベースについてのみ説明します。

Mobile Link は、異種データベース間でもきちんと動作します。リモート・データベースと統合データベースでそれぞれ別個のソフトウェアを使用できるばかりか、スキーマが異なっていても問題ありません。テーブルやカラム名が異なっていたり、カラムの数が異なっていたり、正規化のレベルが異なっていても対応できます。たとえば、統合データベースにある2つのテーブルのデータを単一のリモート・テーブルにダウンロードしたり、その逆を行ったりすることも可能です。また、スクリプトを注意深く使用すれば、アップロード側でも同じことが可能です。

Mobile Link の同期処理は非対称型です。Mobile Link のクライアント・コンポーネントとサーバ・コンポーネントは、アップロード・ストリームとダウンロード・ストリームを、まったく異なる手法で処理します。具体的には、クライアント・コンポーネントがアップロード・ストリームを作成する、サーバがアップロード・ストリームを反映する、サーバがダウンロード・ストリームを作成する、クライアントがダウンロード・ストリームを反映するという4種類の処理があり、すべて異なる手法が使用されます。

【監注6】統合データベースには、Mobile Link 用のシステム・テーブルやストアド・プロシージャが必要。SQL Anywhere には標準装備されている。Sybase ASE、Oracle、Microsoft SQL Server、IBM DB2 UDB については、SQL Anywhere 付属のスクリプトで作成する。それ以外のデータベースはサポートされていない。

アップロード・ストリームは、トランザクション・ログから自動的に構築されます。Mobile Link クライアントは、トランザクション・ログを読み込んで、前回成功した同期以降に挿入、更新、または削除されたローを検出します。次に、それらのローの最終バージョンを使用してアップロード・ストリームを構築し、「挿入」、「更新」、または「削除」のマークを付けて送信します。Mobile Link ではリモート・データベースにASAとUltra Lightしか使用できないという制限は、この特性によるものです。トランザクション・ログの仕様は製品独自のものなので、Oracle など他社製データベースのトランザクション・ログの情報は利用できません。

アップロード・ストリームは、自作のスクリプトで統合データベースに反映します。挿入、更新、削除の各操作をそれぞれのテーブルに適用することが予期される場合、操作の各種類ごとにそれぞれ1つのスクリプトを作成する必要があります。また、更新の競合が予期される場合、それを処理するためのスクリプトも作成します。アップロード・スクリプトについては、「7.6.4.1　アップロードされる挿入の処理」以降で説明します。

ダウンロード・ストリームは、自作のスクリプトで統合データベースから構築します。リモート・データベースに挿入および更新するすべてのローを選択するスクリプトを各テーブルごとに1つずつ作成する必要があります。また、リモート・テーブルからローを削除するためのスクリプトも必要に応じて別途作成します。ダウンロード・スクリプトについては、「7.6.4.6　挿入と更新のダウンロード」以降で説明します。

ダウンロード・ストリームは自動的にフィルタされ、アップロードされたばかりのローは除外されます。これにより、ダウンロード・スクリプトの作成が楽になります。無駄なネットワーク・トラフィックを減らすためのローの除外について心配しなくても、自動的に行われるからです。

ダウンロード・ストリームはリモート・データベースに自動的に反映されます。挿入と更新のどちらであるかは、ダウンロード・ストリーム内には指定されておらず、プライマリ・キーをリモート・データベースに照らして判断されます。つまり、同じキーのローが既に存在する場合には、ダウンロードされたローは更新として反映され、その他の場合には、挿入として反映されます。削除されたローは別途ダウンロードされます。それらは、削除である旨が指定されており、そのように処理されます。以上のことからわかるのは、Mobile Link ではリモート・データベースのすべてのテーブルにプライマリ・キーが必要だということです。これは、第1章で示したリレーショナル・データベースの「ルール2」に合致するものであり、Mobile Link では絶対的な必要条件です。

デフォルトで、Mobile Link クライアントは、リモート・データベースにダウンロードされた変更によって生じた参照整合性違反を自動的に解決します。この自動解決の結果、ダウンロードされた変更が通知なく無視されることや、ON CASCADE DELETE が指定された場合のように既存のローが削除されることがあります。この動作の最大の目的は、ミスがあった場合の管理の手間を軽減することにあります。このトピックの詳細については、「7.6.4.8　ダウンロード・エラーの処理」で説明します。

Mobile Link では、多数のスクリプトを作成することが必要です。同期に影響する複雑なビジネス・ルールがあまりない単純な実装で、統合データベースとリモート・データベースでスキーマに大きな違いがない場合であっても、すべての Mobile Link スクリプトを作成およびテストするのには、多大な労力を要します。もっとも、Mobile Link で強力な機能と柔軟性が実現されるのは、これらのスクリプト

があるからこそです。スクリプトを使用することで、複雑な問題を解決したり、データベース設計の大きな違いに対応したりできます。

Mobile Linkの同期は、リモート・データベースのスキーマによって制御されます。Mobile Linkのアップロード・スクリプトとダウンロード・スクリプトでは、リモート・データベースのテーブルの名前を使用します。スクリプトは統合データベースで実行されるにもかかわらずです。アップロードされたローが統合データベースに反映されるときには、統合データベースではなくリモート・データベースで定義されている参照整合性が維持される順序で行われます。言い換えると、アップロード・ストリームは、リモート・データベースの外部キーの順序に応じてソートされます。親ローは最初に挿入されて最後に削除される、といった形です。

Mobile Linkスクリプトは、統合データベース上で実行され、リモート・データベースにはアクセスできません。Mobile Linkの同期がセッション・ベースだからと言って、両方のデータベースに対してコードで同時にアクセスできるということにはなりません。実際、それは不可能です。Mobile Linkスクリプトでは、リモート・データベースのテーブルの名前を使用していますが、スクリプト自体は統合データベースのテーブルにしかアクセスできません。統合データベースとリモート・データベースとでスキーマが異なる場合には、最初はこれが大きな混乱の原因となることがありますが、いったん理解してしまえば、対応するのは簡単です。

Mobile Linkスクリプトは、Javaや.NET言語（C#など）の他、統合データベース・ソフトウェアがサポートするSQLでも作成できます。本書の例はすべて、SQL Anywhere 9用のSQLで作成しました。

Mobile Linkには、システム全体に対するロックのしくみは用意されていません。2つの異なるリモート・データベースに対して、同じプライマリ・キーでそれぞれ異なるローが挿入されたり、同一のローにそれぞれ異なる更新が適用されたりという事態を防ぐ手立てはありません。こうしたローが統合データベースにアップロードされると、問題が生じます。こうしたエラーや競合が生じないようにアプリケーションを設計するか、または、これらを処理するためのスクリプトを作成する必要があります。特に、プライマリ・キーの衝突は防ぐ必要があります。

失敗に対する処理は、アップロード・ストリームとダウンロード・ストリームのレベルで行われます。アップロード・ストリームの処理が失敗した場合、統合データベースにアップロードされた変更はすべてロールバックされ、ダウンロード処理の開始前に同期が停止します。アップロード中に失敗があった後で同期が再試行された場合、前回の処理はなかったかのように、アップロード・ストリーム全体が再構築され、処理全体が繰り返されます。アップロード・ストリームの再構築を処理するための特別なコードを作成する必要はありません。Mobile Linkクライアント・プログラムが自動的に行ってくれます。

アップロード・ストリームが成功した場合、アップロードされた変更がすべてコミットされ、ダウンロード処理が開始されます。ダウンロード・ストリームの処理が失敗した場合、リモート・データベースにダウンロードされた変更はすべてロールバックされます。ダウンロード中に失敗があった後で同期が再試行された場合、元のアップロード・ストリームは再構築されません。前回の同期で正常に反映されているからです。一方、元のダウンロード・ストリームは再構築されます。前回の処理で正常に反映されなかったからです。ダウンロード・ストリームのこの再構築は、自分で行う必要があります。たいていのアプリケーションでは、これは難しくありませんが、状況によっては、再構築を正しく行うために、

追加的なコーディングが必要になる場合があります。このトピックの詳細については、「7.6.4.8　ダウンロード・エラーの処理」で説明します。

通信エラーは同期の失敗として確実に検出され、上で説明した失敗時の処理が適用されます。データベース・エラーは、デフォルトでは同期の失敗として処理されますが、コードを作成することによって、エラーが起きているローを省略したり、診断情報をテーブルに書き込んでエラーをレポートしたりできます。

更新の競合は、異なるデータベースにおいて、同じローがそれぞれ異なる形で更新され、それらのローが統合データベースにアップロードされたときに生じます。競合はエラーではなく、デフォルトでは通知なしで処理されます。具体的には、最後にアップロードされたローが優先され、そのローのそれ以外のバージョンはすべて置換されます。当然ながら、更新の競合は統合データベースでのみ生じ、検出も統合データベースでしかできません。アプリケーションに適した形で競合を処理または解析するスクリプトを用意すると、デフォルトの動作を無効にできます。詳細については、「7.6.4.4　アップロード競合の処理」を参照してください。

Mobile Link スクリプトはイベント駆動型です。各スクリプトは、単一のイベントに対する応答として呼び出されます。たとえば、テーブル t に対する upload_insert スクリプトは、「挿入」用というマーク付きでアップロードされたローをリモート・テーブル t から受信するたびに呼び出されます。upload_insert スクリプトで必要なのは、そのローを処理することだけです。通常は INSERT を使用して、統合データベース上の対応するテーブルに挿入します。

Mobile Link イベントには、ロー、テーブル、セッション、接続という 4 つのレベルがあります。オンラインヘルプでは、テーブルと接続という 2 つのレベルとして説明されていますが、オンラインヘルプで言う「テーブル・イベント」は、本書で言うロー・レベル・イベントとテーブル・レベル・イベントをひとくくりにしたもので、「接続イベント」は、セッション・レベル・イベントと接続レベル・イベントをひとくくりにしたものです。

ロー・レベル・イベントは、アップロード・ストリームまたはダウンロード・ストリームの各ローまたはロー・セットに対して 1 回発生します。たとえば、テーブル t の upload_insert イベントでは、テーブル t からアップロードされた単一の挿入を処理し、テーブル t の download_cursor イベントでは、リモート・テーブル t にダウンロードするための挿入と更新の結果セットを生成します。ロー・レベルのスクリプトは最も一般的なもので、少なくとも 1 つは作成する必要があります。そうしないと、同期のときに何も処理が行われません。

テーブル・レベル・イベントは、アップロード・ストリームまたはダウンロード・ストリームの各テーブルに対して 1 回発生しますが、データの実際のローを明示的に扱うものではありません。たとえば、テーブル t の begin_upload イベントは、テーブル t に対応するローで最初にアップロードされたものが処理される直前に呼び出されます。これらのイベントの大半はめったに使用されません。

セッション・レベル・イベントは、単一のリモート・データベースからの各同期セッションに対して 1 回発生します。たとえば、begin_download イベントは、アップロード・ストリームが処理された後、ダウンロード・ストリームが生成される直前に呼び出されます。これらのイベントの中には、管理タスクを処理する目的でよく使用されるものもいくつかあります。

接続レベル・イベントは、Mobile Link サーバが統合データベースとの間で確立した各接続に対して1回発生します。Mobile Link サーバは、複数のセッションに対して、同じ接続を次々と再利用できます。したがって、接続レベル・イベントは、単一のセッションに固有の処理には使用できません。これらのイベントは、他のイベントで使用するテンポラリ・テーブルとグローバル変数を作成するためによく使用されます。

特定のイベントに対してスクリプトを用意しなかった場合、デフォルトの動作が行われます。ほとんどの場合は、「何もしない」というのがデフォルトの動作です。たとえば、テーブル t に upload_insert スクリプトがない場合、テーブル t に対する挿入がアップロードされると、統合データベースに到達したときにすべてスキップされます。スクリプトがない場合になんらかの処理が行われるのはエラー処理イベントだけです。具体的には、現在のトランザクションがロールバックされ、セッションが終了されます。

Section

7.3 Mobile Link ASAクライアント

ASA リモート・データベースの Mobile Link クライアントは dbmlsync.exe という名前です。通常は、リモート・データベース・エンジンと同じマシンで実行され、リモート・データベースと統合データベースの間の同期処理を制御します。これは、単一の同期を実行して停止するという形で単発で実行することもできるし、定常的に動作したままにして、あらかじめ決められたスケジュールに基づいて複数の同期を実行することもできます。

Mobile Link クライアントは、新機能であるサーバ駆動同期機能で起動することもできますが、たとえその場合でも、制御を行うのは dbmlsync.exe であり、実際の同期処理は dbmlsync.exe から起動されます。

dbmlsync.exe は、実行すると直ちにリモート・データベースに接続し、同期すべきテーブルとカラムを判断します。また、さまざまなコマンド・ライン・パラメータと、リモート・データベースに格納されている値とを組み合わせて、同期を制御する一連のオプションを判断します。同期を実行するときになると、リモート・データベースの1つまたは複数のトランザクション・ログ・ファイルをスキャンし、アップロード・ストリームを構築して、Mobile Link サーバに接続します。

この時点で、dbmlsync は2つのサーバに対するクライアントの役割を果たしています。つまり、リモート・データベースとの間にはデータベース接続が確立され、Mobile Link サーバとの間にはネットワーク接続が確立されています。

dbmlsync は、Mobile Link サーバに接続した後で、アップロード・ストリームを送信します。そして、統合データベースに対するアップロードの反映とコミット、およびダウンロード・ストリームの構築を Mobile Link サーバが行うのを待ちます。

これ以降は、SendDownloadACK 拡張オプションの設定によってイベントの順序が変わります。

SendDownloadACK が 'OFF'（デフォルト）の場合、Mobile Link サーバは、ダウンロード・ストリームを構築し、統合データベースに対し他のコミットを直ちに実行して、ダウンロード・ストリームを送信します。Mobile Link クライアントは、ダウンロード・ストリームをリモート・データベースに反映し、リモート・データベースに対しコミットを実行して、Mobile Link サーバとリモート・データベースから切断します。

SendDownloadACK が 'ON' の場合、Mobile Link サーバは、ダウンロード・ストリームを構築して送信したら、受信確認を待ったうえで、次のコミットを実行します。Mobile Link クライアントは、リモート・データベースにダウンロード・ストリームを反映し、リモート・データベースに対しコミットを実行して、Mobile Link サーバに受信確認を送信します。これを受けて、Mobile Link サーバは、統合データベースに対して次のコミットを実行し、Mobile Link クライアントは、Mobile Link サーバおよびリモート・データベースから切断します。SendDownloadACK 拡張オプションの詳細については、「7.4.4 DBMLSYNC コマンド・ライン」および「7.6.4.8　ダウンロード・エラーの処理」で説明します。

Section

7.4 Mobile Linkクライアントの設定

Mobile Link クライアントを動作させるために必要なコマンドは 4 つあります。1 つ目は CREATE PUBLICATION です。同期する対象のテーブルとカラムを定義します。2 つ目は CREATE SYNCHRONIZATION USER です。リモート・データベースをユニークに識別します。3 つ目は CREATE SYNCHRONIZATION SUBSCRIPTION です。パブリケーションをリモート・データベースにリンクします。4 つ目は、dbmlsync のコマンド・ライン自体です。

```
<MobiLink_client_setup>          ::= <create_publication>
                                   | <create_synch_user>
                                   | <create_synch_subscription>
                                   | <dbmlsync_command>
                                   | <store_publication_defaults>
```

さらに、任意で使用可能な 5 つ目のコマンドがあります。特別バージョンの CREATE SYNCHRONIZATION SUBSCRIPTION です。これを使用すると、さまざまなオプションのデフォルト値をパブリケーション・レベルで格納できます。

Note

以前のバージョンの Mobile Link では、同期に使用するリモート・データベースの設定方法が異なっていました。本書では、そちらの方法は取り上げません。つまり、CREATE SYNCHRONIZATION DEFINITION、CREATE SYNCHRONIZATION SITE、CREATE SYNCHRONIZATION TEMPLATE の各文については説明しません。

7.4.1 CREATE PUBLICATION

CREATE PUBLICATION 文では、同期の対象とするアーティクル（テーブルとカラム）のリストに名前を割り当てます。この文は、統合データベースではなくリモート・データベース上のみで実行され、リモート・データベースのテーブルとカラムを参照します。

```
<create_publication> ::= CREATE PUBLICATION <publication>
                         "(" <article_list> ")"
<publication>        ::= [ <owner_name> "." ] <publication_name>
<publication_name>   ::= <identifier>
<article_list>       ::= { <article> "," } <article>
<article>            ::= TABLE [ <owner_name> "." ] <table_name>
                         [ "(" [ <column_list> ] ")" ]
                         [ <where_clause> ]
<column_list>        ::= { <column_name> "," } <column_name>
```

パブリケーションに含めるすべてのテーブルをアーティクルで明示的に指定する必要があります。一方、カラム・リストは省略可能です。カラム・リストが省略された場合、そのテーブルのすべてのカラムが対象と見なされます。

カラム・リスト内でのカラムの順序は重要ではありません。アップロードおよびダウンロードされるローの構造は、リモート・データベースでの CREATE TABLE 文のカラムの順序に応じて決まります。

また、アーティクル・リストでのテーブルの順序も重要ではありません。アップロード・ストリームが統合データベースに反映されるときには、リモート・データベースで定義されている参照整合性を維持する順序で行われ、ダウンロード・ストリームがリモート・データベースに反映されるときには、参照整合性の検査は最後のコミットまで先延ばしされます。

Tip

TableOrder 拡張オプションを使用すると、アップロード・ストリームの順序を変更できます。統合データベースで外部キー構造が異なる場合や、外部キー循環に問題がある場合には、これが役立つことがあります。

同期で実際に使用されるどのパブリケーションでも指定されていないテーブルまたはカラムは、同期の対象になりません。これは、カラムの分割を実装する方法の1つです。つまり、一部のテーブルとカラムのみを指定し、それ以外は省略すると、リモート・データベース上でのみ値が存在するテーブルやカラムが、アップロードやダウンロードの対象外となるように定義できます。

複数のパブリケーションに対して、それぞれ異なる組み合わせのテーブルを指定できます。時と場合に応じて、それらのパブリケーションを使用して同期することで、それぞれ異なる目的を達成できます。ただし、複数のパブリケーションで同じテーブルを指定する場合は、そのテーブルに対しては、同じ組

み合わせのカラムを指定する必要があります。時と場合に応じて異なるカラムを同期したい場合は、それらのカラムを別々のテーブルに置いておく必要があります。たとえば、Mobile Link クライアントが、高速で低料金のローカル接続で Mobile Link サーバとつながっているときだけは、ある大きな blob カラムを同期したいが、低速または高料金の長距離接続を使用しているときには同期したくない、という場合には、その blob カラムを単独で別のテーブルに配置します。

パブリケーション内では、各テーブルに対して WHERE 句を使用できます。この機能は、Mobile Link ではあまり使用しませんが、アップロードするローを決定するために使用できます。これは、ローの分割を実装する方法の 1 つです。つまり、どのローをアップロードし、どのローをリモート・データベースにのみ存在させるかを指定できます。この WHERE 句はダウンロード・ストリームには影響しません。つまり、ダウンロードされたローは、パブリケーションの WHERE 句に一致するかどうかにかかわらず、すべて受け入れられます。

次の例は、2 つのテーブルのカラム名を明示的に指定する、p1 というパブリケーションを示します。

```
CREATE PUBLICATION p1 (
   TABLE t1 ( key_1,
              key_2,
              non_key_1,
              non_key_2 ),

   TABLE t2 ( key_1,
              non_key_1 ) );
```

Tip

すべてのカラムを同期するテーブルについても、カラム・リストを明示的に指定する方がよいかどうか検討してみましょう。そのようにすると、メンテナンス作業の量は増えてしまうものの、スキーマが変更になったそのつど、嫌でもパブリケーションを確認することになります。スキーマの変更は、同期がうまくいかなくなる原因となることがあり、Mobile Link 設定のすべての面をチェックしておくことは、いつでも重要です。

7.4.2 CREATE SYNCHRONIZATION USER

CREATE SYNCHRONIZATION USER 文では、このリモート・データベースに対し、グローバルにユニークな Mobile Link ユーザ名を定義します。1 つの統合データベースに同期される各リモート・データベースに対し、それぞれ異なる Mobile Link ユーザ名が必要です。この名前は「データベース識別子」であり、データベースのユーザ ID、パーミッション、所有権とは一切関係ありません。

```
<create_synch_user>     ::= CREATE SYNCHRONIZATION USER
                                <MobiLink_user_list>
```

```
                              [ TYPE <MobiLink_protocol> ]
                              [ ADDRESS <MobiLink_protocol_parameters> ]
                              [ OPTION <extended_option_list> ]
<MobiLink_user_list>    ::= { <MobiLink_user> "," } <MobiLink_user>
<MobiLink_user>         ::= <identifier>
<MobiLink_protocol>     ::= http
                          | https
                          | tcpip
                          | ActiveSync
<MobiLink_protocol_parameters> ::= 文字列リテラル 'keyword=value;...'
<extended_option_list>  ::= { <extended_option> "," } <extended_option>
<extended_option>       ::= <extended_option_name> "=" <extended_option_value>
<extended_option_name>  ::= dbmlsync 拡張オプションのキーワード（表 7-3 を参照）
<extended_option_value> ::= dbmlsync 拡張オプションの値（表 7-3 を参照）
```

CREATE SYNCHRONIZATION USER 文では、dbmlsync が使用するさまざまなパラメータも指定できます。実は、dbmlsync のパラメータを指定できる場所は、ここを含めて 4 つあります。残りの 3 つについては、次からの 3 つの項、すなわち「7.4.3　CREATE SYNCHRONIZATION SUBSCRIPTION」、「7.4.4　DBMLSYNC コマンド・ライン」、「7.4.5　SYSSYNC とパブリケーション・デフォルト」で説明します。

次の例は、典型的な CREATE SYNCHRONIZATION USER 文です。

```
CREATE SYNCHRONIZATION USER "1"
   TYPE tcpip
   ADDRESS 'host=mobilink.risingroad.com'
   OPTION ConflictRetries='2';
```

上の例の TYPE 句では、Mobile Link クライアントと Mobile Link サーバの間で使用する通信プロトコルとして、TCP/IP を指定しています。ADDRESS 句では、Mobile Link サーバのネットワーク・ロケーションを指定しています。ホスト・アドレスは、たとえば同じコンピュータの場合は localhost、LAN 上で隣接するサーバの場合は 192.1.1.50、インターネット上のどこかにあるサーバの場合は 64.7.134.118（または mobilink.risingroad.com）といった形で指定します。OPTION 句では、同期の拡張オプションを定義します。ConflictRetries は、他の接続から行われた変更と競合した場合に行うダウンロードの再試行の回数のリミットを設定します。これらのパラメータや他のパラメータの詳細については、7.4.4 項で説明します。

従来、同期ユーザと、リモート・データベースの実際のユーザとが混同されることがありました。以前の実装では、両者の間に 1 対 1 の関係があったからです。この混乱は現在でも続いています。本書を含むすべての資料で、フィールドが「Mobile Link ユーザ名」という名前で呼ばれていることや、同期の際のセキュリティ・チェックの実装に使用できる Mobile Link スクリプトで authenticate_user というものがあることが原因です。

Tip

柔軟性を高め、管理の手間を軽減するためには、Mobile Link ユーザ名には、リモート・データベース自体をユニークに識別する名前を使用します。リモート・データベースを実際に使用する人ではありません。同期のときに実際のユーザを認証する必要がある場合には、authenticate_user ではなく authenticate_parameters スクリプトを使用します。こうすると、複数の人が同じリモート・データベースを使用したり、1 人が複数のリモート・データベースを使用したりするのが簡単になり、同期が複雑にならずに済みます。

Tip

管理の手間をさらに軽減するには、Mobile Link ユーザ名を、GLOBAL_DATABASE_ID オプション設定と同じにします。CREATE SYNCHRONIZATION USER で "1" や "2" などの数値文字列を使用しても問題ありません。

1 つのリモート・データベースに対して複数の Mobile Link ユーザを定義することも可能です。すべてのリモート・データベースにわたってすべての値がユニークになるようにします。複数の Mobile Link ユーザを定義するのは、同期するデータの組み合わせを時と場合に応じて変えるための 1 つの方法です。ただし、この方法はあまり使用しません。同じ柔軟性を、もっと簡単な方法で実現できるからです。本書の例を含め、大半のアプリケーションでは、各リモート・データベースをそれぞれ単一の Mobile Link ユーザ名で識別します。

Tip

各リモート・データベースの設定に必要な手作業の労力を軽減するには、最初の同期の前に、EXECUTE IMMEDIATE と UUIDTOSTR (NEWID()) を使用して、ユニバーサル・ユニーク識別子を Mobile Link ユーザ名として定義する CREATE SYNCHRONIZATION USER コマンドを実行します。生成される値が見苦しすぎて好みに合わない場合は、統合データベースで DEFAULT AUTOINCREMENT カラムを使用して、代わりとなる数値を生成し、その新しい値を最初の同期の中で返送した後で、リモート・データベースで DROP SYNCHRONIZATION USER と CREATE SYNCHRONIZATION USER を実行して、その値を有効にします。トリッキーなコードですが、設定時の管理の手間をなくすうえでは役立ちます。

7.4.3 CREATE SYNCHRONIZATION SUBSCRIPTION

Mobile Link サブスクリプションとは、1 つのパブリケーションと 1 つの Mobile Link ユーザ名とを結び付けるリンクです。CREATE SYNCHRONIZATION SUBSCRIPTION 文を使用すると、リモート・データベースにこれらのリンクを一度に 1 つずつ作成したり、パブリケーションと Mobile Link ユーザ

名を複数指定して複数のリンクをまとめて作成することもできます。

```
<create_synch_subscription> ::= CREATE SYNCHRONIZATION SUBSCRIPTION
                                    TO <publication_list>
                                    FOR <MobiLink_user_list>
                                    [ TYPE <MobiLink_protocol> ]
                                    [ ADDRESS <MobiLink_protocol_parameters> ]
                                    [ OPTION <extended_option_list> ]
<publication_list>          ::= { <publication> "," } <publication>
```

次は、単一のサブスクリプションを作成する例です。"1" という Mobile Link ユーザ名で識別されるリモート・データベースが、パブリケーション p1 で指定されているすべてのテーブルとカラムを処理対象にします。

```
CREATE SYNCHRONIZATION SUBSCRIPTION TO p1 FOR "1"
   TYPE tcpip
   ADDRESS 'port=2439'
   OPTION LockTables='OFF',
          FireTriggers='ON',
          SendTriggers='ON',
          ScriptVersion='1',
          SendDownloadACK='ON';
```

上の例で、仮に 3 つのパブリケーションと 4 つの Mobile Link ユーザ名を指定した場合には、それぞれの組み合わせごとに 1 つずつ、つまり合計 12 個の異なるサブスクリプションが作成されます。しかし実際には、サブスクリプションは、個別の CREATE SYNCHRONIZATION SUBSCRIPTION 文で 1 つずつ作成するのが現実的です。大半のリモート・データベースでは、Mobile Link ユーザ名は 1 つしかなく、パブリケーションも 1 つか 2 つであることが多いので、12 個のサブスクリプションをまとめて作成する必要性は現実的にはありません。

TYPE 句、ADDRESS 句、OPTION 句を使って、dbmlsync が使用するさまざまなパラメータを指定できます。これらのパラメータについては、次の項で詳しく説明します。

7.4.4 DBMLSYNCコマンド・ライン

Mobile Link クライアント・プログラムである dbmlsync.exe は、［スタート］－［ファイル名を指定して実行］から起動するか、オペレーティング・システムのコマンド・ファイルから起動できます。また、アプリケーション・プログラム内からも起動でき、さらには、データベース内のストアド・プロシージャやイベントからでも起動できます。また、新機能であるサーバ駆動同期機能を使用して、dbmlsync をリスナ・プログラム経由で起動することもできます。

次の例は、dbmlsync を実行する Windows バッチ・ファイルを示します。

```
"%ASANY9%¥win32¥dbmlsync.exe" -c "DSN=remo" -o c:¥temp¥dbmlsync.txt -vnosu -x
```

Tip

バッチ・ファイルでSQL Anywhere 9プログラムのロケーションを明示的に指定するには、%ASANY9% を使用してください。この方が、フル・パスをハード・コーディングするより簡単です。また、PATHを使用するよりも安全です。PATHには、別バージョンのSQL Anywhereに対応するエントリが入っている可能性があるからです。ASANY9環境変数は、SQL Anywhere Studio 9のセット・アップ・プロセスで作成され、ソフトウェアのインストール・パス（たとえばC:¥Program Files¥Sybase¥SQL Anywhere 9）が代入されています。

次は、Windowsで動作するPowerBuilderアプリケーションの例です。ls_addressのMobile Linkサーバ・アドレス以外は、すべてのオプションがハード・コーディングされています。

```
ls_address = 'host=localhost'
li_RC = Run ( '"C:¥Program Files¥Sybase¥SQL Anywhere 9¥win32¥dbmlsync.exe" ' &
   + '-ap "test1,test2" -c "DSN=remo" -e adr=~'' &
   + ls_address &
   + '~' -k -o c:¥temp¥dbmlsync.txt -vnorsu -x' )
```

次は、PocketPCで動作する組み込みVisual Basicアプリケーションでの同様の例です。

```
intRC = CreateProcess("¥windows¥dbmlsync.exe", "-c ""dsn=handheld"" -e adr='" _
    & strHost _
    & "';sa='OFF' -d -k -o ¥Temp¥dbmlsync.txt -pd dbsock9.dll -vnosu -x", _
    0, 0, 0, 0, 0, 0, 0, 0)
```

次は、リモート・データベース自体の中で動作するストアド・プロシージャからdbmlsyncを実行する例です。

```
SET @address = 'host=localhost';
CALL xp_cmdshell ( STRING (
   '"%ASANY9%¥win32¥dbmlsync.exe" -ap "test1,test2" -c "DSN=remo"  -e adr=''',
   @address,
   ''' -k -o c:¥temp¥dbmlsync.txt -vnosu -x' ) );
```

次は、Windowsバッチ・ファイルにコーディングするときのdbmlsyncコマンド・ラインのフォーマットを示します。他の環境でのフォーマットは、コーディング文字列と行セパレータの規則に応じて変わります。

```
<dbmlsync_command> ::= dbmlsync <dbmlsync_option_list> [ <transaction_log_folder> ]
<dbmlsync_option_list>       ::= <dbmlsync_option> { <dbmlsync_option> }
<dbmlsync_option>            ::= <dbmlsync_connection_option>
                               | <dbmlsync_interface_option>
                               | <dbmlsync_session_option>
                               | <dbmlsync_extended_options>
<dbmlsync_connection_option> ::= -c リモート・データベースへの接続文字列
<dbmlsync_interface_option> ::= インタフェースの動作に影響するオプション（表 7-1 を参照）
<dbmlsync_session_option>   ::= 同期に影響するオプション（表 7-2 を参照）
<dbmlsync_extended_options> ::= -e すべての同期に対するオプション（表 7-3 を参照）
                              | -eu 単一の -n パブリケーションに対するオプション（表 7-3 を参照）
<transaction_log_folder>    ::=  古いログ・ファイルか現在のログ・ファイルまたはその両方が格納
                                 されたフォルダ
```

dbmlsync コマンド・ライン自体で必ず指定する必要があるのは、リモート・データベースへの接続文字列のみです。それ以外はすべて省略可能であるか、または、CREATE SYNCHRONIZATION USER 文と CREATE SYNCHRONIZATION SUBSCRIPTION 文を使用して、リモート・データベース自体の中で指定できます。

Note

同じオプションに対して、別々の場所でそれぞれ異なる値を指定した場合、dbmlsync コマンド・ラインの値が優先されるという単純な規則が適用されます。優先度の詳細については、「7.4.5　SYSSYNC とパブリケーション・デフォルト」で説明します。

各オプションの役割と使用方法の完全な説明は、SQL Anywhere 9 のオンラインヘルプに掲載されています。ここでは、参考までに、オプションの概要を示します。表 7-1 は、dbmlsync の内部動作よりも外部インタフェースに関連するオプションを示します。内部の同期処理を制御するオプションは表 7-2 と表 7-3 に示します。表 7-3 は、表 7-2 の -e オプションと -eu オプションの詳細です。

Tip

これらのオプションを見て圧倒される必要はありません。dbmlsync のほぼすべてのオプションは、デフォルト値をそのまま受け入れて問題ありません。別の言い方をすれば、理解できないオプションは触るな、ということです。手始めにどのオプションから理解すればよいのかわからない場合には、この項で先ほど示した例に登場したオプションから始めるとよいでしょう。

表 7-1　動作に影響する DBMLSYNC オプション

オプション	説明
-a	エラー時に再入力のプロンプトを表示しない
-dl	コンソールにログ・メッセージを表示する

オプション	説明
-is	スケジュールを無視する
-k	完了時にウィンドウを閉じる
-l	使用可能な拡張オプションをリストする
-o logfile	診断トレースをテキスト・ファイルに書き込む
-os size	トレース・ファイルの最大サイズ
-ot logfile	診断トレースをテキスト・ファイルにトランケートして書き込む
-p	ログスキャンのポーリングを無効にする
-pd dllname;...	PocketPC でのバグを回避するため（たとえば、-pd dbsock8.dll）
-pi	Mobile Link サーバに ping する
-pp number	ログスキャンのポーリング間隔
-q	最小化ウィンドウで実行する
-urc row-estimate	アップロードされるロー数の推定値
-v［levels］	冗長オペレーション：-v+cpnorsu
-wc classname	ActiveSync 同期の Windows CE クラス名
-x	トランザクション・ログの名前を変更して再起動する

表 7-2　同期に影響する DBMLSYNC オプション

オプション	説明
-ap "value,..."	アプリケーション定義のパラメータをアップロードする
-ba filename	ダウンロード・ファイルを適用する
-bc filename	ダウンロード・ファイルを作成する
-be string	ダウンロード・ファイルに文字列を追加する
-bg	ダウンロード・ファイルを新しいリモート・データベースに適合させる
-d	競合する接続を削除する
-ds	ダウンロードのみの同期処理
-e "opt=value;..."	拡張オプション（OPTION 句と同一、表 7-3 を参照）
-eu "opt=value;..."	単一のパブリケーションに対する拡張オプション（表 7-3 を参照）
-eh	フック関数で発生したエラーを無視する
-ek key	暗号化キー
-ep	暗号化キーを入力するよう要求する
-i filename	同期の後で実行する SQL コマンド・ファイル
-mn password	新しい Mobile Link パスワード
-mp password	Mobile Link パスワード
-n name,...	同期するパブリケーション名
-ra	リモート・データベースの進行状況が統合データベースより後の場合に、リモートの進行状況からアップロードをリトライする
-rb	リモート・データベースの進行状況が統合データベースより前の場合に、リモートの進行状況からアップロードをリトライする
-sc	各同期の前にスキーマ情報を再ロードする
-u ml_username	同期する Mobile Link ユーザ名
-uo	アップロードのみ

表 7-3　DBMLSYNC 拡張オプション -e および -eu

省略形のキーワード	デフォルト	長いキーワードと説明
adr='network-parms;...'		CommunicationAddress。ADDRESS 句と同じ
cr=number	-1	ConflictRetries。無限の場合は -1
ctp=sync-type	tcpip	CommunicationType。OPTION 句と同じ
dbs=number［K|M］	1M（CE 32K）	DownloadBufferSize
dir=path		OfflineDirectory。<transaction_log_folder> と同じ
ds={ON|OFF}	OFF	DownloadOnly
eh={ON|OFF}	OFF	IgnoreHookErrors
el=number［K|M］	32K	ErrorLogSendLimit。アップロードのトレース用
ft={ON|OFF}	ON	FireTriggers。ダウンロードでトリガが起動されるかどうか
hrt=number［K|M］	1M	HoverRescanThreshold。スケジュールされたセッション用
inc=number［K|M］		Increment。インクリメンタル・アップロード時に分割するサイズ。デフォルトは分割せずにひとかたまりでアップロード。
isc={ON|OFF}	OFF	IgnoreScheduling
lt={ON|OFF}	ON	LockTables
mem=number［K|M］	1M	Memory。アップロード・ストリームの構築用
mn=new-password		NewMobiLinkPwd
mp=password		MobiLinkPwd
p={ON|OFF}	OFF	DisablePolling。ログ・スキャン用
pp=number［S|M|H|D］	1M	PollingPeriod。ログ・スキャン用
sa={ON|OFF}	OFF	SendDownloadACK。バージョン 9 での新しいデフォルト
sch=schedule		Schedule。同期用
scn={ON|OFF}	OFF	SendColumnNames。dbmlsrv9 -za、-ze 用
st={ON|OFF}	OFF	SendTriggers。トリガ・アクションのアップロード用
sv=version-name	デフォルト	ScriptVersion。Mobile Link スクリプトを識別
tor=table,...		TableOrder。アップロード・ストリーム内での順序
uo={ON|OFF}	OFF	UploadOnly
v={ON|OFF}	OFF	Verbose。最大限の情報を得る -v+ と同様
vm={ON|OFF}	OFF	VerboseMin。最小限のトレースとする -vm と同様
vn={ON|OFF}	OFF	VerboseRowCounts。-vn と同様
vo={ON|OFF}	OFF	VerboseOptions。-vo と同様
vr={ON|OFF}	OFF	VerboseRowValues。-vr と同様
vs={ON|OFF}	OFF	VerboseHooks。フック・トレース用の -vs と同様
vu={ON|OFF}	OFF	VerboseUpload。アップロード・トレース用の -vu と同様

dbmlsync オプションのリスト自体の中には重複があります。また、dbmlsync コマンド・ラインと、CREATE SYNCHRONIZATION USER 文および CREATE SYNCHRONIZATION SUBSCRIPTION 文との間にも重複があります。たとえば、<transaction_log_folder> は、dbmlsync コマンド・ラインの

末尾に単独で指定することも、-e オプションの中に dir=path または OffLineDirectory=path として指定することもできます。

dbmlsync は、実行時に、どの Mobile Link ユーザに対するどのサブスクリプションを処理するかを判断する必要があります。これは、ユーザ、パブリケーション、サブスクリプションがそれぞれ 1 つずつしかない場合には簡単ですが、ユーザとサブスクリプションが複数あると、話が複雑になってきます。

Note

dbmlsync の単一の実行においては、すべてのサブスクリプションが、同じ同期ユーザに対するものであることが必要です。

dbmlsync では、-n パラメータを使用して 1 つまたは複数のパブリケーションを指定でき、-u パラメータを使用して同期ユーザを指定できます。サブスクリプションは次のように決まります。

- リモート・データベースに同期ユーザが 1 つしかなく、dbmlsync に -n パラメータと -u パラメータを指定しなかった場合、そのユーザのすべてのサブスクリプションが処理される。これは最も一般的な使用法である。この項で先ほど示した例はいずれも、-n オプションや -u オプションは不要である
- リモート・データベースに同期ユーザが 1 つしかない場合、dbmlsync の -n パラメータを使用して、1 つまたは複数のパブリケーションを指定できる。当該の単一ユーザと、指定されたパブリケーション名を使用して、どのサブスクリプションを処理するかが判断される
- リモート・データベースに同期ユーザが 1 つしかない場合、dbmlsync の -u パラメータは余分である。このオプションを指定する場合には、当該の単一ユーザを指定する必要がある
- リモート・データベースに複数の同期ユーザがある場合、dbmlsync の -u パラメータを必ず指定して、このセッションの同期ユーザを示す必要がある
- リモート・データベースに複数の同期ユーザがあり、dbmlsync の -n パラメータを指定しなかった場合には、指定のユーザに対応するすべてのサブスクリプションが処理される
- リモート・データベースに複数の同期ユーザがある場合、dbmlsync の -n パラメータを使用して、1 つまたは複数のパブリケーションを指定できる。指定したユーザとパブリケーション名を使用して、どのサブスクリプションを処理するかが判断される

7.4.5 SYSSYNCとパブリケーション・デフォルト

dbmlsync のオプションが CREATE SYNCHRONIZATION USER 文と CREATE SYNCHRONIZATION SUBSCRIPTION 文を通じてデータベースに格納されるときには、SYSSYNC システム・テーブルに配置されます。具体的には、OPTION 句の値は option カラムに、ADDRESS 句の値は server_connect カラムに、TYPE 句の値は server_conn_type カラムに、それぞれ格納されます。

```
CREATE TABLE SYS.SYSSYNC (
  sync_id            UNSIGNED INT NOT NULL,
  type               CHAR ( 1 ) NOT NULL, -- Mobile Linkでは'D'
  publication_id     UNSIGNED INT NULL,   -- SYSPUBLICATONを指す
  progress           NUMERIC ( 20 ) NULL,
  site_name          CHAR ( 128 ) NULL,   -- Mobile Linkユーザ名
  "option"           LONG VARCHAR NULL,   -- OPTION句
  server_connect     LONG VARCHAR NULL,   -- ADDRESS句
  server_conn_type   LONG VARCHAR NULL,   -- TYPE句
  last_download_time TIMESTAMP NULL,
  last_upload_time   TIMESTAMP NOT NULL DEFAULT 'jan-1-1900',
  created            NUMERIC ( 20 ) NULL,
  log_sent           NUMERIC ( 20 ) NULL,
  generation_number  INTEGER NOT NULL DEFAULT 0,
  extended_state     VARCHAR ( 1024 ) NOT NULL DEFAULT '',
  PRIMARY KEY ( sync_id ) );
```

SYSSYNCに格納されるローは3種類に分かれます。

- 「サブスクリプション」ロー。site_nameカラムにはMobile Linkユーザ名が格納され、publication_idはSYSPUBLICATIONテーブルのローを指す。CREATE SYNCHRONIZATION SUBSCRIPTIONで作成される各サブスクリプションについて、この種類のローが1つ挿入される
- 「ユーザ」ロー。site_nameカラムにはMobile Linkユーザ名が格納されるが、publication_idは空である。CREATE SYNCHRONIZATION USERで作成される各Mobile Linkユーザについて、この種類のローが1つ挿入される
- 「パブリケーション・デフォルト」ロー。site_nameは空だが、publication_idには値が格納される。このローは省略可能である。CREATE PUBLICATIONでは作成されないが、この項で後述する特別な形式のCREATE SYNCHRONIZATION SUBSCRIPTIONでは作成される

dbmlsyncを実行するときには、6か所の異なるロケーションからオプション値が収集されます。そのうちの3つはdbmlsyncコマンド・ライン自体で、残りの3つはすべてSYSSYNC内にあります。複数の場所で、同じオプションに対して異なる値が指定された場合、次のリストに応じて、優先度が最も高い値が使用されます。

- -euコマンド・ライン・オプションの値は、他のすべてのロケーションの値より優先される
- -eコマンド・ライン・オプションの値は2番目に優先される
- 他のコマンド・ライン・オプションの値は3番目に優先される
- CREATE SYNCHRONIZATION SUBSCRIPTIONによってSYSSYNCの「サブスクリプション」ローに格納されたオプションは4番目に優先される
- CREATE SYNCHRONIZATION USERによってSYSSYNCの「ユーザ」ローに格納されたオプションは5番目に優先される

- SYSSYNC の「パブリケーション・デフォルト」ローに格納されたオプションは優先度が最も低い

SYSSYNC の「パブリケーション・デフォルト」ローは省略可能です。パブリケーションは指定するが Mobile Link ユーザ名は割愛する、次のような特別な形式の CREATE SYNCHRONIZATION SUBSCRIPTION 文で作成されます。

```
<store_publication_defaults> ::= CREATE SYNCHRONIZATION SUBSCRIPTION
                                 TO <publication_list>
                                 [ TYPE <MobiLink_protocol> ]
                                 [ ADDRESS <MobiLink_protocol_parameters> ]
                                 [ OPTION <extended_option_list> ]
```

次は、パブリケーション p1 が同期にかかわる場合に使用されるオプション値を指定する例です。ただし、これは他の場所で同じオプションに対して別の値が指定されなかった場合にのみ使用されます。

```
CREATE SYNCHRONIZATION SUBSCRIPTION TO p1
   TYPE tcpip
   ADDRESS 'host=localhost'
   OPTION ConflictRetries='2';
```

dbmlsync に対し、複数の -n オプションでそれぞれ異なるパブリケーションを指定した場合、dbmlsync の 1 回の起動で複数の同期セッションが実行される可能性があります。すべてのセッションは同じ同期ユーザを持つ必要がありますが、各セッションでそれぞれ異なる拡張オプションを使用できます。それには、各セッションに対し、サブスクリプションまたはパブリケーション・レベルで異なる値を指定します。具体的には次のような動作になります。「dbmlsync -n p1 -n p2」と指定した場合、2 つのセッションが別個に実行され、拡張オプションは各セッションに対してそれぞれ別個に決定されます。「dbmlsync -n p1,p2」と指定した場合、1 つのセッションが実行され、拡張オプションは、パブリケーション p1 に対するサブスクリプションについて決定されたものが p2 に対しても使用されます。サブスクリプション p2 に対して指定したオプションを使用するには、「dbmlsync -n p2,p1」と指定します。

Section

7.5 Mobile Linkサーバ

Mobile Link サーバは dbmlsrv9.exe という名前です。統合データベース・サーバと同じマシンで実行することもできるし、隣接する別のマシンで実行して負荷を分散させることもできます。Windows のサービスまたは UNIX のデーモンとして起動することが多く、継続的に動作したままにして、Mobile Link クライアントからの同期要求に対応します。

dbmlsrv9.exe は、起動するとすぐに、統合データベースに対して接続を 1 つ確立します。次に、Mobile Link クライアントから受け取る同期要求に対応できるよう、いくつかのワーカ・スレッドを用意します。これらのスレッドのいずれかで最初の同期セッションが開始されると、統合データベースに対して別の接続が確立されます。この接続は、セッションが完了してもオープンしたままとなり、同じスレッドの次のセッションでは、同じ接続が再利用されます。この一連の処理は接続プーリングと呼ばれます。同じ接続が何度も繰り返して使用され、データベース接続をオープンおよびクローズするオーバヘッドが軽減されます。

2つの Mobile Link クライアントが1つずつ順番に同期を実行すると、Mobile Link サーバでは、ワーカ・スレッドや、データベースに対する接続は、同じものが使用されます。ある Mobile Link クライアントが同期を完了していない時点で、他の複数の Mobile Link クライアントが同期を開始すると、それぞれに対してワーカ・スレッドが割り当てられ、サーバは、使用している各ワーカ・スレッドごとに、統合データベースに対する接続を別個に確立します。すべてのスレッドがビジーのときには、空きスレッドができるまで、新しい同期要求は待機する必要があります。

Mobile Link サーバから統合データベースへの接続では、ODBC が使用されます。Mobile Link の同期を首尾よく行うためには、ODBC ドライバの選択が肝心です。SQL Anywhere 9 のマニュアルで推奨されているドライバを使用することがきわめて重要です。そのドライバは、Oracle などのデータベース・ベンダが提供しているドライバとは異なる場合があります。

Tip

Mobile Link で推奨される ODBC ドライバを見つけるには、http://www.ianywhere.com/developer/technotes/odbc_mobilink.html を参照するか、または Google で「mobilink recommended odbc driver site:ianywhere.com」と検索してください。

同期セッションが開始されると、dbmlsrv9.exe はサーバとクライアントの両方の役割を果たします。Mobile Link クライアントである dbmlsync.exe から見るとサーバであり、統合データベース・サーバから見るとクライアントです。アップロード処理とダウンロード処理の流れについては、「7.3　Mobile Link ASA クライアント」で既に説明しました。

Section

7.6 Mobile Linkサーバの設定

Mobile Link サーバの制御は、自分で作成して統合データベースに格納したスクリプトで行います。これらのスクリプトは ml_script テーブル内に文字列として格納され、同期処理の際に Mobile Link サーバが実行するまで、コンパイルや構文検査は行われません。

```
<store_MobiLink_script> ::= <store_MobiLink_connection_script>
                          | <store_MobiLink_session_script>
                          | <store_MobiLink_table_script>
                          | <store_MobiLink_row_script>
```

スクリプトの保存を簡単にするためのストアド・プロシージャが2つ用意されています。1つは、特定のテーブルに関連付けられていない、接続レベルおよびセッション・レベルのスクリプト用、もう1つは、テーブル名を必要とする、テーブル・レベルとロー・レベルのスクリプト用です。

Note

本書では、Mobile Link イベントを、接続、セッション、テーブル、ローという４種類に分類しています。一方、SQL Anywhere ９のマニュアルでは、接続とテーブルの２種類に分類しています。

7.6.1 接続レベルのスクリプト

接続レベル・イベントは各接続につき1回のみ実行されます。また、接続プーリングがあるため、同じ接続で複数のセッションが次々と実行される場合があります。したがって、接続レベル・イベントは、他のスクリプトで使用するテンポラリ変数やテンポラリ・テーブルの作成以外には、あまり使い道がありません。

次に示すのは、接続レベルのスクリプトを格納するための構文です。

```
<store_MobiLink_connection_script> ::= CALL ml_add_connection_script
                                       ( <script_version>,
                                         <connection_script_name>,
                                         <MobiLink_script> )
<script_version>          ::= 1～128文字以内の文字列値
<connection_script_name> ::= 'begin_connection'
                           | 'begin_connection_autocommit'
                           | 'end_connection'
<MobiLink_script>         ::= 統合データベースのSQLコードが格納された文字列
```

Note

本書で示す Mobile Link スクリプトは、SQL Anywhere ９を使用した統合データベース用の SQL で記述したものです。他の言語（Java や C# など）で記述されたスクリプトや、他の統合データベース（Oracle や Microsoft SQL Server など）用のスクリプトは取り上げません。

次は、@g_session_started というグローバル変数を作成する begin_connection というスクリプトの例です。

```
CALL ml_add_connection_script  ( '1', 'begin_connection',
'CREATE VARIABLE @g_session_started TIMESTAMP;' );
```

「スクリプト・バージョン」は任意に指定できる文字列値で、これを使用して、統合データベースのMobile Linkスクリプトの複数のバージョンを識別できます。各スクリプトのグループ内で、このスクリプト・バージョン値が異なっていれば、同じスクリプトの複数バージョンが共存できます。このスクリプト・バージョンには、自分が必要とするどんな値でも格納できます。たとえば、'1'、'Version 1'、'Intermec Beta Version 0.105.B'はいずれも有効です。大半のプロジェクトでは、スクリプト・バージョンを表す単一の値で始めます。

Note

スクリプト・バージョンは、dbmlsyncの実行時に判断されるオプションの1つです。通常は1回の実行に含まれるすべてのスクリプトで同じスクリプト・バージョンを使用しますが、サブスクリプションの違いに応じて異なるスクリプト・バージョンを使用することもできます。それには、別個のセッションで実行するようdbmlsyncに指示しておきます。具体的には、dbmlsync -n p1 -n p2のように、別個の-nパラメータでパブリケーションを指定します。

7.6.2 セッション・レベルのスクリプト

Mobile Linkのセッション・レベルのスクリプトは、定義に使用するストアド・プロシージャは同じですが、考慮すべきイベントの数はずっと多く、接続レベル・イベントよりもはるかに有用です。

```
<store_MobiLink_session_script> ::= CALL ml_add_connection_script
                                        ( <script_version>,
                                          <session_script_name>,
                                          <MobiLink_script> )
<session_script_name>           ::= 'authenticate_parameters'
                                  | 'authenticate_user'
                                  | 'authenticate_user_hashed'
                                  | 'begin_download'
                                  | 'begin_publication'
                                  | 'begin_synchronization'
                                  | 'begin_upload'
                                  | 'download_statistics'
                                  | 'end_download'
                                  | 'end_publication'
                                  | 'end_synchronization'
                                  | 'end_upload'
                                  | 'handle_error'
                                  | 'handle_odbc_error'
```

```
                                    | 'modify_last_download_timestamp'
                                    | 'modify_next_last_download_timestamp'
                                    | 'modify_user'
                                    | 'prepare_for_download'
                                    | 'report_error'
                                    | 'report_odbc_error'
                                    | 'synchronization_statistics'
                                    | 'time_statistics'
                                    | 'upload_statistics'
```

セッション・レベルのスクリプトの大半は、各同期セッションにつき1回実行されます。たとえば、modify_user、begin_upload、begin_download などがそうです。一方、複数回呼び出されるものもいくつかあります。たとえば、time_statistics や、リターン・コードを「現在のローをスキップして、処理を継続」に設定した場合の handle_error です。handle_error スクリプトの詳細については、「7.6.4.5　アップロード・エラーの処理」を参照してください。

次は、セッション・レベルの begin_synchronization というスクリプトの例です。ストアド・プロシージャの呼び出しとしてコーディングされています。

```
CALL ml_add_connection_script ( '1', 'begin_synchronization',
'CALL ml_begin_synchronization ( ? )' );

CREATE PROCEDURE ml_begin_synchronization (
  IN @ml_username    VARCHAR ( 128 ) )
BEGIN
   INSERT session_history ( ml_username, details )
   VALUES ( @ml_username, 'begin_synchronization' );
END;
```

大半のイベントには、Mobile Link ユーザ名や、前回ダウンロードに成功した日付／時刻などのパラメータが渡されます。これらのパラメータは位置パラメータで、スクリプト内では疑問符としてコーディングする必要があります。上の例では、@ml_username を表す "?" パラメータを1つ使用しています。SQL Anywhere 9 のオンラインヘルプには、すべてのパラメータが詳細に説明されています。

Tip

セッション・レベルとテーブル・レベルのスクリプトは、ストアド・プロシージャ・コールとして記述するとよいでしょう。そうすると、位置パラメータ "?" のプレースホルダを、上の例の @ml_username のような名前付きパラメータに変えることができます。また、スクリプト自体が小さくなるし、スクリプト・バージョンが異なる場合でも、スクリプトの内容が同じなら、同一のプロシージャを使用できます。さらに、もう1つメリットがあります。CREATE PROCEDURE では、構文エラーをコンパイル時に検出できるということです。すべてのコードを Mobile Link スクリプトに記述した場合、エラーは実行時まで検出されません。

Tip

ストアド・プロシージャには、呼び出し元のスクリプトにちなんだ名前を付け、かつ "ml_" などの共通のプレフィクスを付けるとよいでしょう。そうすれば、プロシージャの数が多い場合に目的のプロシージャを探しやすくなり、Mobile Link 関連のプロシージャと、アプリケーションおよびシステム・プロシージャとを切り分けることができます。たとえば、begin_synchronization スクリプトから呼び出されるプロシージャは、ml_begin_synchronization という名前の方が探しやすくなります。

Tip

Sybase Central の「スクリプトのテスト」機能を使用すると、いくつかの基本的なミスを検出できます（図 7-1 を参照）。[Mobile Link 同期 9] プラグインを使用して統合データベースに接続し、左ウィンドウ枠で [接続スクリプト] フォルダを右クリックして、[スクリプトのテスト] を選択します。この機能は、実際のデータを使用した完全なテストの代わりとなるものではありませんが、構文エラーを早い段階で検出できるのは有効です。

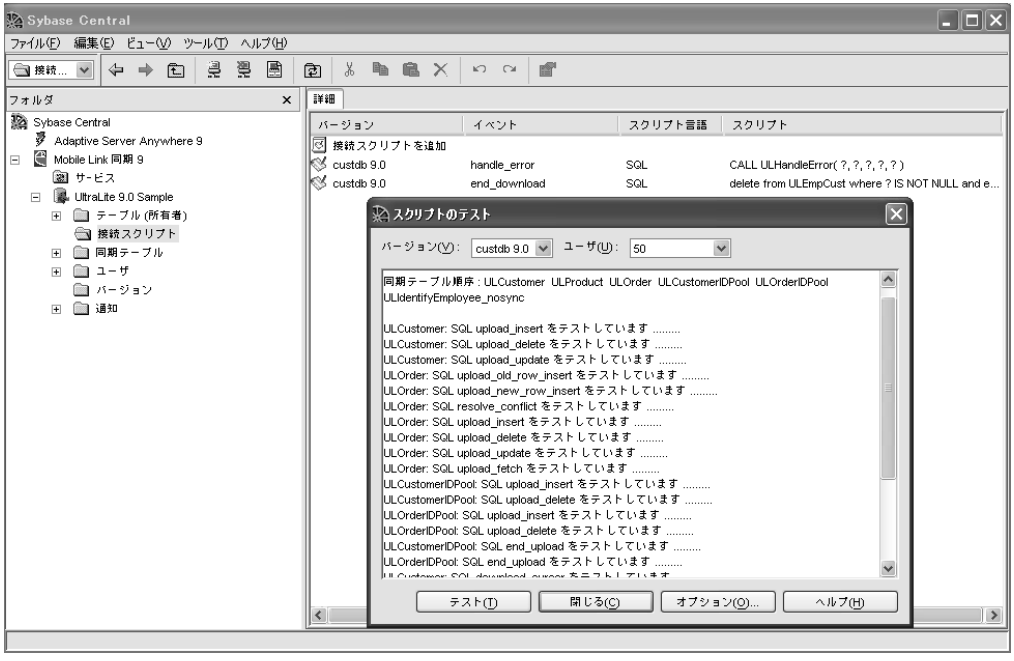


図 7-1　Sybase Central での、Mobile Link の [スクリプトのテスト] ウィンドウ

7.6.3　テーブル・レベルのスクリプト

Mobile Link のテーブル・レベル・イベントは、それぞれ特定のテーブルに対応付けられており、スクリプトの格納には、別のストアド・プロシージャが用意されています。テーブル・レベル・イベント

は18種類あります。それらすべてに対してスクリプトをコーディングすると、仮にデータベースのテーブル数が100だった場合、スクリプトを1,800も作成することになってしまいます。しかし、必須のスクリプトは1つもありませんし、大半のアプリケーションでは、これらのスクリプトは1つも使用しません。

```
<store_MobiLink_table_script> ::= CALL ml_add_table_script
                                    ( <script_version>,
                                      <remote_table_name>,
                                      <table_script_name>,
                                      <MobiLink_script> )
<remote_table_name>           ::= リモート・データベースのテーブル名を表す文字列
<table_script_name>           ::= 'begin_download'
                                | 'begin_download_deletes'
                                | 'begin_download_rows'
                                | 'begin_synchronization'
                                | 'begin_upload'
                                | 'begin_upload_deletes'
                                | 'begin_upload_rows'
                                | 'download_statistics'
                                | 'end_download'
                                | 'end_download_deletes'
                                | 'end_download_rows'
                                | 'end_synchronization'
                                | 'end_upload'
                                | 'end_upload_deletes'
                                | 'end_upload_rows'
                                | 'synchronization_statistics'
                                | 'time_statistics'
                                | 'upload_statistics'
```

次は、テーブル・レベルのスクリプトの例です。テーブルt1に対してテーブル・レベルとロー・レベルの各イベントが発生した回数を記録します。

```
CALL ml_add_table_script  ( '1', 't1', 'time_statistics',
'CALL ml_time_statistics_by_table ( ?, ?, ?, ?, ?, ?, ? )' );

CREATE PROCEDURE ml_time_statistics_by_table (
  IN @ml_username  VARCHAR ( 128 ),
  IN @table        VARCHAR ( 128 ),
  IN @event_name   VARCHAR ( 128 ),
  IN @num_calls    INTEGER,
  IN @min_time     INTEGER,
  IN @max_time     INTEGER,
  IN @total_time   INTEGER )
BEGIN
   INSERT session_history ( ml_username, details )
   VALUES ( @ml_username,
      STRING ( @table, ' - ', @event_name, ' ', @num_calls ) );
END; -- ml_time_statistics_by_table
```

time_statistics はテーブル・レベル・イベントなので、処理する対象の各テーブルに対してそれぞれ1つのスクリプトを作成する必要があります。しかし、このケースでは、実行時にテーブル名もスクリプトに渡されます。この冗長な情報は、実は役に立ちます。1つだけ作成したストアド・プロシージャを、すべてのスクリプトから呼び出すことができるからです。次は、テーブル t2 に対するスクリプトの例です。

```
CALL ml_add_table_script ( '1', 't2', 'time_statistics',
'CALL ml_time_statistics_by_table ( ?, ?, ?, ?, ?, ?, ? )' );
```

7.6.4 ロー・レベルのスクリプト

Mobile Link のロー・レベル・イベントは、肝心な動作がすべて行われる部分です。アップロードされる挿入、更新、削除の処理には、それぞれ別個のスクリプトが必要です。具体的には、upload_insert、upload_update、upload_delete の各スクリプトです。また、ダウンロードされる挿入と更新の生成は1つのスクリプト（download_cursor）で行われ、ダウンロードされる削除の生成はそれとは別のスクリプト（download_delete_cursor）で行われます。その他、別個のリモート・データベースからアップロードされたローどうしの競合の処理に使用するスクリプトもあります。

ロー・レベルのスクリプトの格納には、テーブル・レベルのスクリプトで先ほど使用したのと同じ、ml_add_table_script というストアド・プロシージャを使用します。

```
<store_MobiLink_row_script> ::= CALL ml_add_table_script
                                  ( <script_version>,
                                    <remote_table_name>,
                                    <row_script_name>,
                                    <MobiLink_script> )
<row_script_name>           ::= 'download_cursor'
                              | 'download_delete_cursor'
                              | 'resolve_conflict'
                              | 'upload_delete'
                              | 'upload_fetch'
                              | 'upload_insert'
                              | 'upload_new_row_insert'
                              | 'upload_old_row_insert'
                              | 'upload_update'
```

スクリプトは、実行が予期される操作に対応するもののみ作成すればかまいません。たとえば、削除されるローがない場合には、upload_delete スクリプトや download_delete_cursor スクリプトを作成する必要はありません。

Note

以前のバージョンの Mobile Link では、アップロード・ストリームの処理に別の方法を使用してい

ました。その方法は現在でも使用できますが、本書ではそちらについては取り上げません。つまり、upload_cursor、new_row_cursor、old_row_cursor の各イベントについては説明しません。

7.6.4.1 アップロードされる挿入の処理

ロー・レベル・イベント upload_insert は、リモート・データベースに挿入されて dbmlsync によりアップロードされる各ローに対して、統合データベース上で 1 回呼び出されます。

Note

リモート・データベースで、前回の同期の後で挿入されたローが、さらにその後で更新された場合、そのローは、最終バージョンの単一の挿入としてアップロードされます。

upload_insert スクリプトでは、統合データベースの対応するテーブルに対してローを単純に挿入するのが一般的です。

```
<typical_upload_insert>   ::= INSERT <consolidated_table_name>
                                  "(" { <column_name> "," }
                                        <column_name> ")"
                                  VALUES "(" { "?," } "?" ")"
<consolidated_table_name> ::= 統合データベース上に存在するテーブル
```

次の例が示すように、スクリプト内では、アップロードされる各カラムをそれぞれ単一の "?" パラメータ・プレースホルダで表します。これらのカラム・プレースホルダは、リモート・データベースの CREATE TABLE のカラム順序で指定します。

```
CALL ml_add_table_script ( '1', 't1', 'upload_insert', '
INSERT t1
       ( key_1,
         key_2,
         non_key_1,
         non_key_2 )
VALUES ( ?,
         ?,
         ?,
         ? )' );
```

7.6.4.2 アップロードされる更新の処理

ロー・レベル・イベント upload_update は、リモート・データベースで更新されて dbmlsync によりアップロードされる各ローに対して、統合データベース上で 1 回呼び出されます。

Note

リモート・データベースで、前回の同期の後で削除されたローが、さらにその後で再度挿入された場合、そのローは、挿入や削除ではなく更新としてアップロードされます。

upload_update スクリプトでは、統合データベースの対応するテーブルのローを単純に更新するのが一般的です。

```
<typical_upload_update>       ::= UPDATE <consolidated_table_name>
                                    SET { <non_primary_key_column_name> "= ?," }
                                          <non_primary_key_column_name> "= ?"
                                  WHERE <primary_key_column_name> "= ?"
                                  { AND <primary_key_column_name> "= ?" }
<non_primary_key_column_name> ::= プライマリ・キーでないカラム
<primary_key_column_name>     ::= プライマリ・キーのカラム
```

次の例が示すように、まず非プライマリ・キー・カラムに"?"プレースホルダを指定し、その後でプライマリ・キー・カラムに"?"プレースホルダを指定して、SET 句と WHERE 句に正しく配置します。

```
CALL ml_add_table_script ( '1', 't1', 'upload_update', '
UPDATE t1
   SET non_key_1 = ?,
       non_key_2 = ?
 WHERE key_1 = ?
   AND key_2 = ?' );
```

処理ロジックを追加したい場合には、upload_update スクリプトで、UPDATE の代わりにストアド・プロシージャ・コールを指定することができます。要注意なのは、パラメータを正しい順序で指定する部分です。まず、非プライマリ・キー・カラムを、リモートの CREATE TABLE の順序で指定し、その後でプライマリ・キー・カラムを、こちらもリモートの CREATE TABLE の順序で指定します。

たとえば、もともとは次のようなスクリプトだとします。

```
CALL ml_add_table_script ( '1', 't3', 'upload_update', '
UPDATE t3
   SET non_key_1 = ?
 WHERE key_1 = ?
' );
```

これをストアド・プロシージャ・コールに書き換えると、次のようになります。

```
CALL ml_add_table_script  ( '1', 't3', 'upload_update', '
CALL ml_upload_update_t3 ( ?, ? )
```

```
' );

CREATE PROCEDURE ml_upload_update_t3 (
  IN @non_key_1    INTEGER,
  IN @key_1        UNSIGNED BIGINT )
BEGIN
-- このコードは、処理の内容は同じだが、
-- こちらの方が複雑なロジックを使用しやすい
   UPDATE t3
      SET non_key_1 = @non_key_1
    WHERE key_1 = @key_1;
END;
```

Tip

作成する Mobile Link スクリプトで、COMMIT コマンドや ROLLBACK コマンドを使用したり、これらのコマンドを使用するストアド・プロシージャを呼び出したり、暗黙的なコミットを行う文（TRUNCATE TABLE や DROP INDEX など）を実行したりすることは避けてください。同期の際のトランザクション処理は Mobile Link サーバの役割です。特に、エラーが発生した場合には、Mobile Link サーバは、次回の同期で処理を安全に繰り返すことができるよう、アップロード・ストリームまたはダウンロード・ストリームの全体をロールバックできる必要があります。

Tip

ml_add_connection_script または ml_add_table_script を呼び出した後は、COMMIT を忘れないでください。これを忘れた場合、Mobile Link のシステム・テーブルでロックが保持される可能性があり、次の同期セッションが実行できなくなる恐れがあります。これは、「スクリプト内で COMMIT を使用してはいけない」というヒントとは別の話です。「スクリプトを追加した後で COMMIT を実行せよ」というアドバイスです。

7.6.4.3　アップロードされる削除の処理

ロー・レベル・イベント upload_delete は、リモート・データベースで削除されて dbmlsync によりアップロードされる各ローに対して、統合データベース上で 1 回呼び出されます。

Note

リモート・データベースで、前回の同期の後で更新されたローが、さらにその後で削除された場合、そのローは単一の削除としてアップロードされます。また、挿入された後で削除されたローはアップロードされません。そのローは統合データベースにはこれまで存在せず、現時点でリモート・データベースにも存在しないからです。

upload_delete スクリプトでは、統合データベースの対応するテーブルからローを単純に削除するのが一般的です。

```
<typical_upload_delete> ::= DELETE <consolidated_table_name>
                              WHERE <primary_key_column_name> "= ?"
                              { AND <primary_key_column_name> "= ?" }
```

次の例が示すように、upload_delete スクリプトでは、プライマリ・キー・カラムに対する "?" プレースホルダのみが必要です。

```
CALL ml_add_table_script ( '1', 't1', 'upload_delete', '
DELETE t1
 WHERE key_1 = ?
   AND key_2 = ?' );
```

Tip

統合データベースに特定の種類の操作を反映したくない場合には、そのスクリプトを割愛します。たとえば、挿入と更新は反映するが削除は反映したくない場合は、upload_delete スクリプトを割愛します。その場合でも、削除はアップロードされますが、Mobile Link サーバはこれを無視します。

7.6.4.4 アップロード競合の処理

大まかに言うと、アップロード競合とは、アップロードされた単一のローに対して resolve_conflict イベントを発生させるものです。このように大雑把な定義で述べたのには理由があります。実は、アップロード競合とは、対処が必要な単なる問題ではなく、強力なプログラミング・ツールでもあるのです。アップロード競合には、通常の競合と強制的競合の 2 種類があります。そして、強制的競合の方は、自分が望むとおりに扱うことができます。この項では、まず通常の競合について説明し、続いて強制的競合について説明します。

「通常の競合」とは、複数のリモート・データベースで同じローが更新され、それらが統合データベースにアップロードされたときに生じるものです。また、統合データベースとリモート・データベースの両方で同じローが更新され、そのローがリモート・データベースから統合データベースにアップロードされたときにも生じます。

アプリケーションによっては、競合が起きないものもあります。同じローを複数のデータベースで更新できないようにデータベースが設定されている場合です。また、アプリケーションによっては、競合を特に問題としない場合もあります。つまり、「最後にアップロードされた更新が優先される」というデフォルトの動作でかまわないというケースです。しかし、多くのアプリケーションでは、競合が発生した場合に従うべき、特別なビジネス・ルールの類があります。こうしたアプリケーションでは、競合をまず検出したうえで、その競合に対処する必要があります。そして、検出と対処というそれぞれのアク

ションのために、追加的なMobile Linkスクリプトを作成することが必要です。

アップロードされる各更新は、ローの2つのコピーで構成されます。すなわち、ローの更新前にリモート・データベースに入っていた古いカラム値と、upload_updateスクリプトで統合データベースに通常適用される新しいカラム値の両方が含まれています。通常の競合は、アップロードされた古い値（アップロードされた新しい値ではない）を、統合データベースに現在格納されている値と比較することで検出されます。両者が同じ場合は、競合は生じておらず、upload_updateスクリプトの処理が進められ、新しい値が反映されます。

リモート・データベースからアップロードされた古い値が、統合データベースの現在の値と異なる場合は、通常の競合が生じています。これは、次の2つの方法のいずれかで検出できます。

1つ目は、upload_fetchスクリプトを使用する方法です。テーブルにupload_fetchスクリプトが作成されている場合、Mobile Linkは、そのスクリプトを使用して、アップロードされた各更新に対する競合検査を実行します。競合が検出されなかった場合は、upload_updateスクリプトにローが引き渡され、処理が進められます。競合が検出された場合は、upload_updateイベントは呼び出されません。代わりに行われる処理については少し後で説明することにして、ここでは、競合の検出に絞って話を進めていきます。

upload_fetchスクリプトでは、SELECTを使用する必要があります。そのSELECTでは、selectリストにすべてのカラムを指定し、WHERE句にすべてのプライマリ・キー・カラムを指定します。他のMobile Linkスクリプトと同様に、テーブルとカラムは統合データベースに存在するものを指定しますが、カラムの順序はリモート・データベースのCREATE TABLEに指定したカラムの順序に一致させる必要があります。

```
<typical_upload_fetch>      ::= SELECT { <column_name> "," }
                                      <column_name>
                                FROM <current_values_table_name>
                                WHERE <primary_key_column_name> "= ?"
                                { AND <primary_key_column_name> "= ?" }
<current_values_table_name> ::= 統合データベースのターゲット・テーブル
```

upload_fetchスクリプトの例を次に示します。統合データベースから現在のカラム値を取り出す方法をMobile Linkに対して指定するSELECTは、自分で記述する必要があります。そうすると、Mobile Linkサーバがその SELECTを実際に実行して、取り出した値と、リモート・データベースからアップロードされた古いカラム値とを比較してくれます。

```
CALL ml_add_table_script ( '1', 't2', 'upload_fetch', '
SELECT key_1,
       key_2,
       non_key_1,
       non_key_2
  FROM t2
 WHERE key_1 = ?
   AND key_2 = ?' );
```

競合を検出する方法の2つ目は、upload_fetch スクリプトとは別の方法です。upload_update スクリプトの WHERE 句で、プライマリ・キー・カラムに加えて非キー・カラムがすべて指定されている場合、Mobile Link はこの拡張版 upload_update スクリプトを使用して競合を検出します。競合が検出された場合は、拡張版 upload_update スクリプトは更新の適用を行いません。競合が検出されなかった場合は、拡張版 upload_update は通常どおり処理を進めます。

```
<typical_extended_upload_update> ::= UPDATE <consolidated_table_name>
                                      SET { <non_primary_key_column_name> "= ?," }
                                            <non_primary_key_column_name> "= ?"
                                     WHERE <primary_key_column_name> "= ?"
                                     { AND <primary_key_column_name> "= ?" }
                                       AND <non_primary_key_column_name> "= ?"
                                     { AND <non_primary_key_column_name> "= ?" }
```

先ほどの upload_fetch と同様に通常の競合を検出できる拡張版 upload_update の例を次に示します。WHERE 句では、プライマリ・キー・カラムをまず指定し、その後で非キー・カラムを指定しています。

```
CALL ml_add_table_script ( '1', 't2', 'upload_update', '
UPDATE t2
   SET non_key_1 = ?,
       non_key_2 = ?
 WHERE key_1 = ?
   AND key_2 = ?
   AND non_key_1 = ?
   AND non_key_2 = ?' );
```

upload_fetch スクリプトと拡張版 upload_update スクリプトの両方を作成した場合でも、害はありませんが、upload_update の長い WHERE 句をコーディングする苦労が無駄骨になってしまいます。競合の検出は upload_fetch で行われるからです。

Note

upload_delete スクリプトでも、同じ拡張 WHERE 句を使用でき、非キー・カラムをすべて含む述部を追加できます。

競合の検出は処理全体の第1歩にすぎません。競合へ実際に対処するためには、upload_new_row_insert、upload_old_row_insert、resolve_conflict という3つのスクリプトがさらに必要です。upload_new_row_insert スクリプトと upload_old_row_insert スクリプトでは、アップロードされた古い値と新しい値を格納できます。通常はテンポラリ・テーブルに格納します。resolve_conflict スクリプトには、競合に対処するコードを記述します。

```
<typical_upload_old_row_insert> ::= INSERT <old_values_table_name>
                                      "(" { <column_name> "," }
                                            <column_name> ")"
                                     VALUES "(" { "?," } "?" ")"
<old_values_table_name> ::= アップロードされた更新前イメージを保持するテンポラリ・テーブル
<typical_upload_new_row_insert> ::= INSERT <new_values_table_name>
                                      "(" { <column_name> "," }
                                            <column_name> ")"
                                     VALUES "(" { "?," } "?" ")"
<old_values_table_name> ::= アップロードされた更新後イメージを保持するテンポラリ・テーブル
```

upload_old_row_insert イベントは、各競合に対して 1 回呼び出され、アップロードされた更新ローの各カラムの古い値が渡されます。同様に、upload_new_row_insert には、新しいカラム値が渡されます。次に、resolve_conflict スクリプトが呼び出されます。古い値と新しい値を保存した場合には、ローの 3 つのバージョン（古いロー、新しいロー、現在のロー）のすべてにここでアクセスできることになります。

複数の更新の競合をマージすることが必要なビジネス・ルールの実装例を次に示します。ここでは、競合する変更を取りまとめて、その結果を統合データベースに適用します。upload_old_row_insert スクリプトは t2_old テンポラリ・テーブルにローを挿入し、upload_new_row_insert スクリプトは t2_new にローを挿入します。resolve_conflict スクリプトは、3 つのテーブルをすべてジョインして、non_key_1 カラムと non_key_2 カラムの最終的な値を計算します。ここでは、スクリプトが短くなるよう、ストアド・プロシージャを使用しています。

```
CALL ml_add_table_script ( '1', 't2', 'upload_old_row_insert', '
INSERT t2_old
       ( key_1,
         key_2,
         non_key_1,
         non_key_2 )
VALUES ( ?, ?, ?, ? )' );

CALL ml_add_table_script ( '1', 't2', 'upload_new_row_insert', '
INSERT t2_new
       ( key_1,
         key_2,
         non_key_1,
         non_key_2 )
VALUES ( ?, ?, ?, ? )' );

CALL ml_add_table_script ( '1', 't2', 'resolve_conflict',
'CALL ml_resolve_conflict_t2 ( ?, ? )' );

CREATE PROCEDURE ml_resolve_conflict_t2 (
  IN @ml_username    VARCHAR ( 128 ),
  IN @table_name     VARCHAR ( 128 ) )
```

```
BEGIN
   UPDATE t2
      SET t2.non_key_1 = t2.non_key_1 - t2_old.non_key_1 + t2_new.non_key_1,
          t2.non_key_2 = t2.non_key_2 - t2_old.non_key_2 + t2_new.non_key_2
     FROM t2
     JOIN t2_old
       ON t2.key_1 = t2_old.key_1
      AND t2.key_2 = t2_old.key_2
     JOIN t2_new
       ON t2.key_1 = t2_new.key_1
      AND t2.key_2 = t2_new.key_2;
   DELETE t2_new;
   DELETE t2_old;
END;
```

Tip

テンポラリ・テーブルのローは、不要になったときに忘れず削除してください。後で競合に対処するときに、何回も繰り返し処理することになってしまわないようにするためです。

Tip

必要に応じて、複数の異なるテーブルに対する競合解決のロジックを単一のプロシージャに配置できます。テーブル名は、パラメータの 1 つとして resolve_conflict イベントに渡されるので、行うべき動作をコードで判断できます。

Note

通常の upload_update スクリプトはあるが upload_fetch スクリプトはない場合、競合は検出されず、upload_update が実行されます。これは「最後にアップロードされた更新が優先される」という状況です。upload_fetch スクリプトと通常の upload_update スクリプトの両方があるが、競合解決のスクリプトがない場合には、アップロードされた更新で競合が生じたものは無視されます。これは「最初の更新が優先される」という状況です。「最初の更新」とは、先に行われた更新の場合と、統合データベースに対して直接行われた更新の場合とがあります。

通常の競合の検出と解決のプロセス全体を、単一のストアド・プロシージャにマージして、拡張版 upload_update スクリプトから呼び出すことができます。次の例は、先ほどの例のすべてのスクリプトの代わりとなる、拡張版 upload_update スクリプトと ml_upload_update_t2 というプロシージャを示します。つまり、次のコードは、先ほどの upload_update、upload_old_row_insert、upload_new_row_insert、resolve_update の各スクリプトと、ml_resolve_conflict_t2 プロシージャの代わりとなります。拡張版 upload_update スクリプトからプロシージャに対して、新しい非キー値、プライマリ・キー・カラム、古い非キー値のそれぞれにつき、"?" パラメータ値が渡されます。

```
CALL ml_add_table_script ( '1', 't2', 'upload_update', '
CALL ml_upload_update_t2 ( ?, ?, ?, ?, ?, ? )' );
CREATE PROCEDURE ml_upload_update_t2 (
  IN @non_key_1     INTEGER,
  IN @non_key_2     INTEGER,
  IN @key_1         UNSIGNED BIGINT,
  IN @key_2         INTEGER,
  IN @old_non_key_1 INTEGER,
  IN @old_non_key_2 INTEGER )
BEGIN
   UPDATE t2
      SET t2.non_key_1 = t2.non_key_1 - @old_non_key_1 + @non_key_1,
          t2.non_key_2 = t2.non_key_2 - @old_non_key_2 + @non_key_2
    WHERE t2.key_1 = @key_1
      AND t2.key_2 = @key_2;
END;
```

続いて、「強制的競合」の説明に移ります。強制的競合とは、次の3つの条件が満たされた場合に生じるものです。1つ目は、挿入、削除、または更新がアップロードされたということです。2つ目は、その対象のテーブルに upload_insert、upload_delete、upload_update、upload_fetch の各スクリプトがないことです。3つ目は、upload_old_row_insert スクリプトと upload_new_row_insert スクリプトがあることです。resolve_conflict スクリプトもあってもよいですが、なくてもかまいません。

アップロードされた挿入に対して強制的競合が発生すると、リモート・データベースからの新しいローは upload_new_row_insert イベントが受け取ります。次に、resolve_conflict スクリプトが実行されますが、upload_old_row_insert イベントは実行されません。先ほどの例のように、スクリプトでテンポラリ・テーブルにローを挿入する場合、t2_new には1つのローが含まれているのに対し、t2_old は空という点から、resolve_conflict スクリプトでは、アップロードされた挿入によって自らが呼び出されたということを判断できます。

アップロードされた削除に対して強制的競合が発生すると、リモート・データベースからアップロードされた、削除されたロー全体を、upload_old_row_insert イベントが受け取ります。次に、resolve_conflict スクリプトが実行されますが、upload_new_row_insert イベントは実行されません。resolve_conflict スクリプトが実行されるときには、t2_old には1つのローがあるのに対し、t2_new は空です。

アップロードされた更新に対して強制的競合が発生すると、upload_old_row_insert イベントと upload_new_row_insert イベントの両方が呼び出され、resolve_conflict スクリプトが実行されるときには、t2_old と t2_new のそれぞれに1つのローがあります。

これら3つのイベントを使用すると、複雑な同期の問題を解決できます。たとえば、統合データベースとリモート・データベースのデータベース設計の違いへの対処などです。異なるテーブルの複数のローを組み合わせて1つにしたり、その逆を行ったりできます。あるテーブルに加わった変更を複数のテーブルに分散させることもできます。リモート・データベースで実行されたアクションを、統合データベースに到達した時点で変更することもできます。たとえば、更新と削除を挿入に変更して、詳細な

監査証跡としてすべてを記録することができます。こうしたロジックが可能なのは、強制的競合が発生したときには、3種のデータ（つまり、リモート・データベースの古いローと新しいロー、および統合データベースの現在のロー）をすべて利用可能だからです。

7.6.4.5　アップロード・エラーの処理

アップロード・エラーは、2つの点で競合とは異なります。1つは、エラーを通知なしで処理するためのメカニズムが組み込まれていない点、もう1つは、アップロードをロールバックして同期セッションを中止するというのがデフォルトの動作だという点です。この動作を変えるのは簡単ではありません。だからこそ、そもそもエラーが発生しないようにすることが重要です。

アップロード・エラーで最もありがちなのは、同期スクリプトのコーディング・ミスです。これらは簡単に修復できるのが普通であり、またアップロード全体がロールバックされていることから、統合データベースでスクリプトを修正して同期セッションを再度実行し直せば済みます。

Tip

データベース設計に手を加えるときには、Mobile Link 特有のエラーに注意してください。「Mobile Link 特有のエラー」とは、ソフトウェアの設計が原因で発生する誤りのことです。Mobile Link のケースで言えば、同じテーブルに対して複数の異なるスクリプトを作成する必要があるため、テーブル・レイアウトが変更になったときに、どれかのスクリプトを忘れてしまいがちです。たとえば、テーブルのカラムを追加または削除するときには、upload_insert、upload_update、upload_fetch、upload_old_row_insert、upload_new_row_insert、download_cursor の各スクリプトを確認してください。また、CREATE SYNCHRONIZATION PUBLICATION 文のカラムのリストも確認してください。プライマリ・キーの定義を修正する場合には、upload_update、upload_delete、download_delete_cursor の各スクリプトの他、シャドー・テーブルと削除トリガも確認してください。シャドー・テーブルについては、「7.6.4.7　削除のダウンロード」で説明します。

Tip

スキーマをごくわずかでも変更した場合には、同期を必ずテストしてください。一組のテスト・データベースと、すべての Mobile Link スクリプトを実行する一連の簡単なテスト・ケースを用意し、さらに、テストの実行方法と結果の確認方法を説明する Read me ファイルを作成します。ユーザ主導の回帰テストによってすべてのスクリプトを試したり細かな問題を発見したりできるものと期待しないでください。Mobile Link スクリプトでは、テストはきわめて重要です。ごく基本的な構文エラーであっても、スクリプトの実行時まで発覚しないからです。

もっと重大なアップロード・エラーとしては、実際のデータにまつわるものがあります。たとえば、プライマリ・キーの重複や参照整合性違反などです。大半のアプリケーションでは、こうしたエラーが生じないようにデータベースを設計しておくのが最適な対処法です。DEFAULT GLOBAL AUTOINCREMENT 機能と GLOBAL_DATABASE_ID オプションを使用すると、たとえばプライマ

リ・キーを確実にユニークにさせることができます。詳細については、1.8.2 項を参照してください。

リモート・データベースと統合データベースの両方で同じ外部キー関係が存在し、CREATE PUBLICATION 文で必要なテーブルをすべて指定した場合には、参照整合性違反は発生しません。スキーマの違いについては、対処の手間が必要です。たとえば、「7.4.1　CREATE PUBLICATION」で説明した TableOrder 拡張オプションや、「7.6.4.4　アップロード競合の処理」で説明した強制的競合スクリプトなどです。

しかし、いざというときには、たとえアップロード・エラーがあった場合でもノンストップでの操作が必要になるアプリケーションもあります。1 つの方法としては、不正なデータはスキップし、残りのデータは処理を続けるという手があります。これは、handle_error スクリプトで対応可能です。次は、すべてのエラーをスキップする例です。

```
CALL ml_add_connection_script ( '1', 'handle_error',
'CALL ml_handle_error ( ?, ?, ?, ?, ? )' );

CREATE PROCEDURE ml_handle_error (
  INOUT @action_code    INTEGER,
  IN    @error_code     INTEGER,
  IN    @error_message  LONG VARCHAR,
  IN    @ml_username    VARCHAR ( 128 ),
  IN    @table          VARCHAR ( 128 ) )
BEGIN
   SET @action_code = 1000; -- ローをスキップして継続
END;
```

もっと高度な handle_error スクリプトを作成し、発生したエラーの種類や関係するテーブルに応じて異なるアクションを行うようにすることも簡単です。アクション・コード・パラメータのデフォルトは 3000 です。アップロードをロールバックして同期セッションを中止するという意味です。これは、handle_error スクリプトがない場合のデフォルトのアクションでもあります。その他の値には、上の例でも使用した 1000（アップロードされたローでエラーの原因となったものをスキップし、残りのアップロードで処理を継続する）や、4000（アップロードをロールバックし、サーバを完全に停止する）があります。

後で分析するためにすべてのエラーを記録する方法の 1 つとしては、Mobile Link サーバを -o オプション付きで起動して、すべてのエラー・メッセージをテキスト・ファイルに書き込むという手があります。もう 1 つの方法としては、統合データベース上の独自のテーブルにエラー情報を挿入するという手があります。これを行える場所は 2 つあります。1 つは handle_error スクリプト、もう 1 つは report_error スクリプトです。report_error スクリプトに独自の INSERT を記述するメリットは、別の接続で実行されて直ちにコミットされるため、アップロードがロールバックされた場合でもローがそのまま残るという点です。一方、handle_error スクリプトに記述した INSERT は、その時点でアクション・コードが 3000 か 4000 に設定されている場合、または、後にアップロードのコミット前のある時点で 3000 か 4000 に設定された場合にロールバックされます。

次の例は、report_error スクリプトと、このスクリプトが使用するテーブルを示します。error_code カラムは INTEGER ではなく VARCHAR として定義されているので、このテーブルは、数ではなく英数字の SQLSTATE を受け取る report_ODBC_error スクリプトでも使用できます。

```
CREATE TABLE ml_error (
   ml_username     VARCHAR ( 128 ) NOT NULL,
   inserted_at     TIMESTAMP NOT NULL DEFAULT TIMESTAMP,
   unique_id       UNSIGNED BIGINT NOT NULL DEFAULT AUTOINCREMENT,
   action_code     INTEGER NOT NULL,
   error_code      VARCHAR ( 100 ) NOT NULL,
   error_message   LONG VARCHAR NOT NULL,
   table_name      VARCHAR ( 128 ) NOT NULL,
   PRIMARY KEY ( ml_username, inserted_at, unique_id ) );

CALL ml_add_connection_script  ( '1', 'report_error',
'CALL ml_report_error ( ?, ?, ?, ?, ? )' );

CREATE PROCEDURE ml_report_error (
  IN @action_code    INTEGER,
  IN @error_code     INTEGER,
  IN @error_message  LONG VARCHAR,
  IN @ml_username    VARCHAR ( 128 ),
  IN @table          VARCHAR ( 128 ) )
BEGIN
   INSERT ml_error
   VALUES ( @ml_username,
            DEFAULT,
            DEFAULT,
            @action_code,
            CAST ( COALESCE ( @error_code, 0 ) AS VARCHAR ( 100 ) ),
            COALESCE ( @error_message, '' ),
            COALESCE ( @table, '' ) );
END;
```

次は、プライマリ・キー違反がスキップされた後の ml_error ローの内容の例です。

```
'1', '2003 07 28 16:55:54.710000', 8, 1000, '-193',
'ODBC: [Sybase][ODBC Driver][Adaptive Server Anywhere]Integrity
   constraint violation: Primary key for table ''t1'' is not
   unique (ODBC State = 23000, Native error code = -193)', 't1'
```

Tip

最初に発生したエラーについての診断情報の記録のみを行い、後はセッションをロールバックして中止したいという場合には、handle_error スクリプトは省略し、上記のような report_error スクリプトのみを使用します。

アップロード・エラーを処理するもう1つの方法としては、アップロードされたローを受け取る基本スクリプトを変更するという手があります。たとえば、upload_insert スクリプトの INSERT 文で ON EXISTING SKIP 句を使用して、プライマリ・キー違反があるローをスキップすることができます。または、ON EXISTING UPDATE を使用して、失敗する INSERT をきちんと動作する UPDATE に変更します。当然ながら、これらの方法は、統合データベースが SQL Anywhere の場合のみ使用できます。Oracle など、他のソフトウェアを使用している場合には、「7.6.4.4　アップロード競合の処理」で説明した強制的競合スクリプトなどを使用する必要があり、もっと手間がかかります。

7.6.4.6　挿入と更新のダウンロード

ダウンロード・ストリームは、アップロード・ストリームとは違って、自作の Mobile Link スクリプトで完全に制御できます。ダウンロードされた削除については次の項で説明します。この項では、ダウンロード・ストリームの挿入部分と更新部分を構築する方法を説明します。

ダウンロードする各テーブルについて、download_cursor というスクリプトを作成する必要があります。この中では、リモート・データベースへの挿入または更新が必要なすべてのローを、統合データベースから選択します。どのローが挿入でどのローが更新かという区別については心配無用です。dbmlsync がダウンロード・ストリームを受け取ったときに、すべて自動的に制御されます。具体的には次のような動作です。すなわち、ダウンロードされたローのプライマリ・キーが、リモート・データベースに既に存在するローのプライマリ・キーに一致する場合、dbmlsync はこれを、ダウンロードされた更新として扱います。リモート・データベースのどのローにもプライマリ・キーが一致しない場合、挿入として処理します。これは、「アップサート（upsert）」と呼ばれることがあります。「必要に応じた更新（アップデート）または挿入（インサート）」という意味です。

Tip

Mobile Link 同期にかかわるどのローでも、プライマリ・キー値は更新しないでください。また、あるローを削除して、同じプライマリ・キー値のローを直ちに再挿入することもしないでください。Mobile Link は、挿入、更新、削除の対象のローを、プライマリ・キー値に基づいて判断します。キー値を変更することが必要なアプリケーションの場合には、テーブル上でそのキーは別個の UNIQUE 制約としておき、DEFAULT GLOBAL AUTOINCREMENT カラムを PRIMARY KEY として追加します。分散データベース環境でローの追跡を確実に行うためには、プライマリ・キーが変更されないことが大前提です。そうでない場合、混乱に陥るのは必至です。

最も簡単な download_cursor スクリプトは、"SELECT * FROM t" というものです。これは、すべてのカラムとローをリモート・データベースに送信します。dbmlsync によって、新しいローは自動的に挿入され、古いローは更新されます。実質的に、統合テーブル全体の「スナップショット」がダウンロードされることになります。これは、「スナップショット同期」と呼ばれることがよくあります。リモート・データベースでテーブルを読み込み専用として扱い、かつ統合データベースでローを削除しない場合、

スナップショット同期では、リモート・データベースのテーブルの内容全体が各同期のつど置換されることになります。

スナップショット同期は、サイズが小さくて短時間で変更できるテーブルならうまく機能するかもしれませんが、サイズが大きなテーブルの場合には、多大なネットワーク・トラフィックが生成されることになります。もっと一般的な手法の1つとしては、前回の最後のダウンロード以降に統合データベースで挿入または更新されたローのみをダウンロードするという方法があります。DEFAULT TIMESTAMP を指定した TIMESTAMP カラムを統合テーブルに用意すると、最初の "?" プレースホルダとして download_cursor スクリプトに渡される last_download パラメータを利用できます。これは「タイムスタンプ・ダウンロード」と呼ばれます。

```
<typical_download_cursor> ::= SELECT { <column_name> "," }
                                      <column_name>
                                FROM <consolidated_table_name>
                               WHERE <when_updated_column_name> "> ?"
<when_updated_column_name> ::= DEFAULT TIMESTAMP を指定した TIMESTAMP カラム
```

次に示すのは、簡単なテーブルと、それに対応するタイムスタンプ・ベースの download_cursor スクリプトの例です。t1 にローが挿入または更新されるたびに、DEFAULT TIMESTAMP の特別な機能によって、last_updated カラムが CURRENT TIMESTAMP に設定されます。このカラムは、WHERE 句でのみ指定し、SELECT のリストでは指定しません。リモート・データベースでは必要ないため、その中には含めないということです。last_updated が統合データベース上に存在する唯一の目的は、download_cursor スクリプトを制御することです。

```
CREATE TABLE t1 (
   key_1         UNSIGNED BIGINT NOT NULL DEFAULT GLOBAL AUTOINCREMENT ( 1000000 ),
   key_2         INTEGER NOT NULL DEFAULT 0,
   non_key_1     VARCHAR ( 100 ) NOT NULL DEFAULT '',
   non_key_2     VARCHAR ( 100 ) NOT NULL DEFAULT '',
   last_updated  TIMESTAMP NOT NULL DEFAULT TIMESTAMP,
   PRIMARY KEY ( key_1, key_2 ) );

CALL ml_add_table_script  ( '1', 't1', 'download_cursor', '
SELECT key_1,
       key_2,
       non_key_1,
       non_key_2
  FROM t1
 WHERE last_updated > ?' );
```

Note

last_download パラメータの初期値は 1900-01-01 です。

download_cursor スクリプトでは、複数の異なるテーブルからデータをジョインしたり、複雑な WHERE 句に基づいてローを選択したりなど、リモート・データベースの対象のテーブルに適用する結果セットを目的どおりに構築するために必要なことは、おおむねどんなことでも実行できます。従うべき唯一の規則は、当該テーブルの CREATE PUBLICATION と同じ数のカラムを SELECT リストに指定する必要があり、リモート・データベースの CREATE TABLE と同じデータ型（または互換性のあるデータ型）を同じ順序で指定するということです。多くの場合、これは簡単です。たいてい、両方のデータベースでテーブルはよく似ており、すべてのカラムを同期することになるからです。

しかし、アプリケーションによっては、スキーマが異なっていたり、別々のリモート・データベースに対してそれぞれ異なる組み合わせのローをダウンロードすることが必要な場合があります。Mobile Link では、こうした特別なケースに対応しやすくするためのしくみが用意されています。具体的には、現在の同期セッションの Mobile Link ユーザ名が、download_cursor スクリプトの第 2 パラメータで渡されます。Mobile Link ユーザ名をデータベース・カラムに格納し、このパラメータを WHERE 句の 2 番目の "?" プレースホルダとして参照することにより、異なるリモート・データベースに対するデータを切り分けることができます。

Tip

download_cursor スクリプトからはストアド・プロシージャを呼び出すことができます。ただし、そのプロシージャは、リモート・データベースのテーブルのダウンロード要件に適合する単一の結果セットを返すものでなければなりません。

次は、download_cursor スクリプトを作成するときの自由度の高さを示す、短いながらも込み入った例です。

```
CALL ml_add_table_script ( '1', 'tr4', 'download_cursor', '
SELECT tc3.key_3,
       tc2.non_key_1,
       tc3.non_key_1
  FROM tc1
  JOIN tc2 ON tc1.key_1 = tc2.key_1
  JOIN tc3 ON tc2.key_1 = tc3.key_1 AND tc2.key_2 = tc3.key_2
 WHERE tc3.last_update  > ?                    -- last_download
   AND tc1.db_id        = CAST ( ? AS BIGINT ) -- ML_username' );
```

この例のしくみを次に示します。

❶ このスクリプトは、リモート・データベースの tr4 という名前のテーブルにデータをダウンロードするためのものです。統合データベースにはその名前のテーブルはありませんが、スクリプトで構築する結果セットが tr4 に対応していれば問題ありません。

❷ SELECT で、統合データベースの tc1、tc2、tc3 の 3 つのテーブルをジョインします。これらはすべて、リモート・テーブル tr4 とは、名前もスキーマも異なります。Mobile Link スクリプトはリモート・データベースにはアクセスできず、統合データベースのテーブルしか参照できません。統合データベースのこれら 3 つのテーブルは、次のようなものです。

```
CREATE TABLE tc1 ( -- 統合データベースで作成
   key_1         BIGINT NOT NULL,
   db_id         BIGINT NOT NULL,
   PRIMARY KEY ( key_1 ) );

CREATE TABLE tc2 ( -- 統合データベースで作成
   key_1         BIGINT NOT NULL,
   key_2         BIGINT NOT NULL,
   non_key_1     BIGINT NOT NULL,
   PRIMARY KEY ( key_1, key_2 ),
   FOREIGN KEY ( key_1 ) REFERENCES tc1 );

CREATE TABLE tc3 ( -- 統合データベースで作成
   key_1         BIGINT NOT NULL,
   key_2         BIGINT NOT NULL,
   key_3         BIGINT NOT NULL UNIQUE,
   non_key_1     BIGINT NOT NULL,
   last_update   TIMESTAMP NOT NULL DEFAULT TIMESTAMP,
   PRIMARY KEY ( key_1, key_2, key_3 ),
   FOREIGN KEY ( key_1, key_2 ) REFERENCES tc2 );
```

❸ SELECT リストでは、tc2 と tc3 の 3 つのカラムを、tr4 の要件に一致する順序でピックアップします。これはきわめて重要な点です。tr4 で同期する対象のカラムは CREATE PUBLICATION で指定され、カラムの順序は tr4 の CREATE TABLE で指定されており、download_cursor の SELECT はそれらに合致する必要があります。次に示すのは、リモート・データベースでのテーブルとパブリケーションの定義です。

```
CREATE TABLE tr4 ( -- リモート・データベースで作成
   key_1         BIGINT NOT NULL,
   non_key_1     BIGINT NOT NULL,
   non_key_2     BIGINT NOT NULL,
   PRIMARY KEY ( key_1 ) );

CREATE PUBLICATION p1 (
   TABLE tr4 ( key_1,
               non_key_1,
               non_key_2 ) );
```

❹ download_cursor スクリプトの FROM 句では、tc1、tc2、tc3 を、外部キー関係に従ってジョイン

します。これは、正規化の解除の例です。つまり、download_cursor によって、統合データベースでの複数レベルの階層がフラットにされ、リモート・データベースの単一のテーブルになります。

❺ WHERE 句で、先ほど説明したタイムスタンプ・ダウンロードの手法を実装します。「tc3.last_update > ?」という部分です。

❻ WHERE 句では、2つ目の "?" プレースホルダを使用して、Mobile Link ユーザ名が一致するローのみに結果セットを限定しています。「tc1.db_id = CAST (? AS BIGINT)」という部分です。db_id カラムは、ダウンロードする各テーブルに格納しておくこともできますが、たいていは、親テーブルに格納しておいて、ジョインを使用して検索する形で十分です。

❼ WHERE 句で指定したいずれのカラムも、ダウンロードはされません。それどころか、一方のテーブル（tc1）は、SELECT リストでまったく指定していません。

❽ CAST 関数を使用して、このアプリケーションでは Mobile Link ユーザ名に数値を使用しているということを明示しています。実際には、Mobile Link ユーザ名のデータ型は VARCHAR(128) です。

Tip

download_cursor スクリプトで、last_download パラメータは必要ないが、Mobile Link ユーザ名のパラメータは参照したいという場合には、常に TRUE に評価されるダミーの式を使用して、last_download パラメータをスキップさせることができます。たとえば、'... WHERE ? IS NOT NULL AND db_id = ?' などです。こうする必要があるのは、"?" プレースホルダは位置に基づくものだからです。両方を省略したり、最初の 1 つだけを指定したりすることはできますが、2 つ目だけが必要な場合には、両方を指定する必要があります。

Tip

異なるリモート・データベースのそれぞれに対して送信したいローのセットに、重複する部分がある場合には、ダウンロードしたいローのプライマリ・キー値と Mobile Link ユーザ名とを関連付ける多対多関係の別個のテーブルに Mobile Link ユーザ名を格納できます。これらの関係は、必要に応じて、どれだけ柔軟でも複雑でもかまいません。download_cursor で適切な SELECT を記述できさえすれば OK です。

7.6.4.7 削除のダウンロード

Mobile Link で統合データベースからリモート・データベースに削除をダウンロードするのは比較的困難です。統合データベースからローが削除されると、次の同期セッションが行われるときには、そのローが存在したという形跡が残っていません。その点から問題が生じます。アップロード・プロセスとは違って、ダウンロードはトランザクション・ログからは構築されません。どのリモート・データベースからどのローを削除するかをコードで指定する必要があります。

この問題にはいくつかの解決法があります。1つ目は最も単純です。すなわち、何も削除しないとい

う方法です。ばかばかしい解決法に思えるかもしれませんが、アプリケーションによっては、実はそうでもありません。時として、ビジネス上の理由から、長期にわたってデータを保管しておくことが必要な場合があります。そして現在では、ディスク領域はきわめて安価に用意できるので、「長期」を「永遠」と見なして実装できるのです。

2つ目は、リモート・データベースの古いデータをアプリケーションで削除するという解決法です。この方法では、挿入と更新のみをダウンロードし、削除はダウンロードしません。どの場合にローの削除が必要なのかということを判断するのは、アプリケーションの役目です。この方法の問題点の1つは、リモート・データベースで明示的に実行された削除がトランザクション・ログに記録され、dbmlsyncが構築するアップロード・ストリームに含まれてしまうということです。こうした削除を統合データベースで繰り返したいかどうかは場合によります。繰り返したくない場合は、そのテーブルのupload_deleteスクリプトを割愛すれば済みますが、無駄なネットワーク・トラフィックが生じるのは避けられません。

この問題を回避するには、STOP SYNCHRONIZATION DELETEという特別なコマンドで、削除のアップロードを一時的に抑制します。このコマンドは、SQL Anywhere 9に対し、これ以降、現在の接続については、トランザクション・ログに記録された削除は、dbmlsyncを実行するときには完全に無視するという旨を通知します。START SYNCHRONIZATION DELETEコマンドを使用すると、通常の動作に戻すことができます。

次の例は、2年が経過したローをリモート・データベースから削除する方法を示します。その際、削除を統合データベースにアップロードすることはなく、ダウンロードの削除方法について心配する必要もありません。

```
STOP SYNCHRONIZATION DELETE;
DELETE t7
 WHERE t7.last_updated < DATEADD ( year, -2, CURRENT TIMESTAMP );
START SYNCHRONIZATION DELETE;
```

Note

STOP SYNCHRONIZATION DELETEコマンドでは、予期せぬ影響が生じることがあります。たとえば、STOP SYNCHRONIZATION DELETEが有効な状態で、リモート・データベースにローを挿入した後ですぐに削除した場合、次の同期処理でそのローが挿入としてアップロードされますが、リモート・データベースにはそのローは既に存在しません。その理由は、dbmlsyncがトランザクション・ログを処理するときに、挿入は対象に含めるのに対し、STOP SYNCHRONIZATION DELETEコマンドのために削除は無視するからです。

リモート・データベースからローを削除する方法の3つ目は、処理対象の各テーブルに対し、Mobile Linkのdownload_delete_cursorスクリプトを作成する方法です。このスクリプトでは、リモート・データベースから削除する各ローに対応するプライマリ・キー値を含む結果セットを生成する必要があります。この結果セットはダウンロード・ストリームに組み込まれ、dbmlsyncによって処理されます。

ここで、1つの難問が生じます。既に存在しないローのプライマリ・キーを含む結果セットを、いったいどのように生成すればよいのかということです。広く利用されている方法としては、ローが削除されるたびに、別個の「シャドー・テーブル」にローを挿入するという手があります。このシャドー・テーブルには、元のテーブルのプライマリ・キー・カラムに加え、いつ削除されたかを表すタイムスタンプ・カラムを含めますが、それ以外の非キー・カラムは必要ありません。そして、download_delete_cursorでこのシャドー・テーブルを使用して、既に存在しないローのプライマリ・キーをダウンロードします。

```
<typical_download_delete_cursor> ::= SELECT { <primary_key_column_name> "," }
                                           <primary_key_column_name>
                                     FROM <shadow_table_name>
                                    WHERE <when_deleted_column_name> "> ?"
<shadow_table_name>        ::= [ <owner_name> "." ] <table_name>
<when_deleted_column_name> ::= DEFAULT CURRENT TIMESTAMP を指定した TIMESTAMP カラム
```

次の例は、テーブルt1に対する典型的なdownload_delete_cursorスクリプトを示します。この中では、前回の同期セッション以降にDELETED_t1シャドー・テーブルに挿入されたローを選択します。

```
CALL ml_add_table_script ( '1', 't1', 'download_delete_cursor', '
SELECT key_1,
       key_2
  FROM DELETED_t1
 WHERE deleted_on > ? -- last_download ' );
```

シャドー・テーブルの定義を次に示します。プライマリ・キーにdeleted_onカラムが含まれているのは、統合データベースで同じプライマリ・キーの挿入と削除が複数回行われたという状況に対処するためです。

```
CREATE TABLE DELETED_t1 (
   key_1        UNSIGNED BIGINT NOT NULL,
   key_2        INTEGER NOT NULL,
   deleted_on   TIMESTAMP NOT NULL DEFAULT CURRENT TIMESTAMP,
   PRIMARY KEY ( key_1,
                 key_2,
                 deleted_on ) );
```

次は、シャドー・テーブルを管理するトリガの例を示します。このトリガでは、t1からローが削除されるたびに、DELETED_t1にローを挿入します。

```
CREATE TRIGGER trd_ml_t1 BEFORE DELETE ON t1
   REFERENCING OLD AS old_t1
   FOR EACH ROW
BEGIN
```

```
   INSERT DELETED_t1
   VALUES ( old_t1.key_1, old_t1.key_2, DEFAULT );
END; -- trd_ml_t1
```

シャドー・テーブルのロー自体は、すべてのリモート・データベースにダウンロードされたときに削除できます。これは、ml_subscriptionというMobile Linkシステム・テーブルを確認することで判断できます。このテーブルには、各リモート・データベースのlast_download_timeが記録されており、シャドー・テーブルのローのうちで最も古いlast_download_timeより前に挿入されたものは既に不要と判断できます。

次の例は、シャドー・テーブルの古いローのクリーンアップを実行するbegin_publicationスクリプトを示します。

```
CALL ml_add_connection_script ( '1', 'begin_publication',
'CALL ml_begin_publication ( ?, ?, ?, ?, ? )' );

CREATE PROCEDURE ml_begin_publication (
  INOUT @generation_number  INTEGER,
  IN    @ml_username        VARCHAR ( 128 ),
  IN    @publication_name   VARCHAR ( 128 ),
  IN    @last_upload        TIMESTAMP,
  IN    @last_download      TIMESTAMP )
BEGIN
   DECLARE @oldest_download   TIMESTAMP;
   SELECT MIN ( ml_subscription.last_download_time )
     INTO @oldest_download
     FROM ( SELECT a.last_download_time
              FROM ml_subscription AS a
             WHERE a.publication_name = @publication_name
               AND a.progress = ( SELECT MAX ( b.progress )
                                    FROM ml_subscription AS b
                                   WHERE b.user_id = a.user_id
                                     AND b.publication_name = @publication_name ) )
          AS ml_subscription ( last_download_time );
   DELETE DELETED_t1
    WHERE DELETED_t1.deleted_on < @oldest_download;
END; -- ml_begin_publication
```

Note

@oldest_downloadを取得するためのSELECTが複雑なのは、ml_subscriptionの「アクティブ」なローのみを検査して、削除および置換されたサブスクリプションを表す古い「非アクティブ」なローは除外する必要があるからです。ml_subscriptionの詳細については、「7.7　Mobile Linkのシステム・テーブル」を参照してください。

Note

ml_subscription.last_download_timeが確実に最新の値になる最も早い段階はbegin_synchronizationイベントで、これはSendDownloadACKの設定に関係ありません。しかし、このコードを記述する場所としては、begin_publicationイベントの方が適しています。その理由は、begin_publicationイベントはパブリケーション名をパラメータとして受け取るのに対し、begin_synchronizationは受け取らないからです。

シャドー・テーブルの方法が広く使われているのは、非常に効率がよいからです。また、アプリケーションの動作方法やアドホック・クエリの作成方法に変更を加える必要もありません。シャドー・テーブルは、あらゆる場所から隠ぺいされた存在です。ただし、シャドー・テーブルはきわめて冗長です。つまり、この方法で処理する各テーブルに対し、新しいテーブル、新しいトリガ、新しいMobile Linkスクリプトをそれぞれ1つずつ作成する必要があります。

シャドー・テーブルを一切使わずに済む方法もあります。各テーブルに「削除フラグ」を追加し、実際にそのローを削除する代わりにそのフラグを「オン」にするという方法です。次は、この方法を使用する場合のdownload_delete_cursorの例です。

```
CALL ml_add_table_script ( '1', 't1', 'download_delete_cursor', '
SELECT key_1,
       key_2
  FROM t1
 WHERE deleted = 'Y'
   AND last_updated > ? -- last_download' );
```

削除フラグを使用する方法では、シャドー・テーブルとトリガを別途用意する必要がありません。そして、begin_publicationスクリプトで同様のロジックを使用して、ダウンロードに必要なくなったローを実際に削除します。この方法の大きな欠点は、削除されたローをスキップするようにアプリケーション・プログラムを注意深く作成する必要があるということです。また、エンド・ユーザも、アドホック・レポートやその他のクエリを作成するときに注意が必要です。多くの環境では、削除フラグの方法を使用するのは、破滅への第1歩です。

リモート・データベースのテーブルのすべてのローを削除したい場合は、特別な形のdownload_delete_cursorを使用できます。具体的には、各プライマリ・キー・カラムをNULL値として単一のローを選択するだけです。

```
<typical_download_delete_truncate>  ::= SELECT { NULL, }
                                            NULL
```

次のdownload_delete_cursorスクリプトでは、t1のすべてのローが削除されます。この方法は、リモート・データベースで収集されたデータがアップロード後には必要ない場合に役立ちます。

```
CALL ml_add_table_script ( '1', 't1', 'download_delete_cursor', '
SELECT NULL,
       NULL' );
```

Tip

リモート・データベースにのみ存在するテーブルに対して全レコードの削除をするには、download_delete_cursor のトランケート版を使用できます。このテーブルに対しては、NULL 値を選択する download_delete_cursor だけを用意し、他のスクリプトはすべて割愛します。ローはアップロードされますが無視されます。また、download_delete_cursor の特別な NULL ロー以外は何もダウンロードされません。

7.6.4.8 ダウンロード・エラーの処理

この項では、4 種類のダウンロード・エラーについて説明します。1 つ目は、リモート・データベースにデータが到達するずっと前に、Mobile Link サーバによって検出される種類のエラーです。たとえば、データ型が誤っていたり、不正な NULL があったりなどの理由で、ダウンロード値がリモート・データベースのカラム定義に一致しない場合には、Mobile Link サーバは、統合データベースでダウンロード・プロセスを次のようにロールバックします。

```
E. 08/17 11:56:26. <1.5> [1]: Error: [-10038] A downloaded value for table t8
   (column #3) was either too big or invalid for the remote schema type
...
I. 08/17 12:35:51. <1.5> [1]: ROLLBACK Transaction: end_download
```

Mobile Link クライアントも、リモート・データベースでダウンロード・プロセスを次のようにロールバックします。

```
I. 08/17 12:35:59. ROLLBACK
E. 08/17 12:35:59. SQLCODE from MobiLink server is: -10038
E. 08/17 12:35:59. Message: A downloaded value for table t8 (column #3) was either
   too big or invalid for the remote schema type.  Table Name: t8
I. 08/17 12:35:59. Download stream processing failed
```

2 つ目のダウンロード・エラーは、リモート・データベースにデータが到達して初めて検出される種類のものです。たとえば、統合データベースでは有効だがリモート・データベースでは UNIQUE 制約に違反するカラム値の場合、Mobile Link クライアントでエラーとして次のように検出されます。

```
E. 08/21 09:42:35. SQL statement failed: (-196) Index 't8 UNIQUE (unique_1)' for
   table 't8' would not be unique
```

デフォルトでは、このエラーによって、リモート・データベースでダウンロードが失敗し、すべての変更がロールバックされます。しかし、SendDownloadACK 拡張オプションが'OFF'（デフォルト）の場合、この失敗はMobile Linkサーバには通知されず、サーバ側のダウンロード処理はコミットされます。ここで、管理に影響する2つの難題が生じます。1つは、このエラーを修正するために、エラーをどのように感知するかということです。もう1つは、修復を組み込んだダウンロード・ストリームを、次の同期でいかにして再送するかということです。

1つ目は、問題を報告するための連絡をだれもくれない場合や、リモート・コンピュータにアクセスできない場合には難問です。しかし、ダウンロード・エラーを自動的に記録して、次の同期でその情報をアップロードするようにリモート・データベースを設定することは可能です。

Mobile Link では、リモート・データベースに「フック・プロシージャ」という機能が用意されています。これは、統合データベースの Mobile Link イベントとスクリプトに似たものです。これらのストアド・プロシージャは、同期の際にリモート・データベースで発生する各イベントに対応する特別な名前を持ちます。最初は、これらのプロシージャは1つもありません。統合データベースの Mobile Link スクリプトも、自分が作成するまでは1つもありませんが、それと同じことです。そして、特別な名前を持つこれらのフック・プロシージャのいずれかを作成しておくと、対応するイベントか発生したときに呼び出されるという点も、Mobile Link スクリプトと同じです。

こうしたフック・プロシージャの1つに、sp_hook_dbmlsync_download_fatal_sql_error というものがあります。このプロシージャは、ダウンロードの段階で SQL 文が失敗した場合に呼び出されます。このフック・プロシージャは、統合データベースの report_error イベントと同様に、別個の接続で実行されます。つまり、このプロシージャでリモート・データベースに加えた変更は、たとえダウンロード・ストリーム全体がロールバックされる場合でもコミットされます。

sp_hook_dbmlsync_download_fatal_sql_error が呼び出されるときには、エラーについての多数の情報がこのプロシージャに渡されます。SQL エラー・コード、テーブル名、Mobile Link ユーザ名、パブリケーション名などです。この情報をリモート・データベースの独自のテーブルに挿入し、そのテーブルをパブリケーションに指定しておけば、そのエラーのレコードが次の同期の際にアップロードされます。

次は、リモート・データベースのエラー・テーブルの例です。これに対応するテーブルを統合データベースにも定義して、upload_insert スクリプトを用意する必要があります。

```
CREATE TABLE dbmlsync_sql_error (
   ml_username  VARCHAR ( 128 ) NOT NULL,
   inserted_at  TIMESTAMP NOT NULL DEFAULT TIMESTAMP,
   unique_id    UNSIGNED BIGINT NOT NULL DEFAULT GLOBAL AUTOINCREMENT ( 1000000 ),
   publication_name       VARCHAR ( 128 ) NOT NULL,
   table_name             VARCHAR ( 128 ) NOT NULL,
   sql_error_code         VARCHAR ( 10 ) NOT NULL,
   script_version         VARCHAR ( 128 ) NOT NULL,
   PRIMARY KEY ( ml_username, inserted_at, unique_id ) );
```

各フック・プロシージャはそれぞれ独自のパラメータを受け取りますが、その方法が通常とは異なります。具体的には、#hook_dict という名前の、小さな見慣れないテンポラリ・テーブルが作成され、フック・プロシージャが呼び出される直前に、Mobile Link クライアントによって中身が格納されます。各パラメータは、#hook_dict の個別のローで表されます。各ローには "name" と "value" という 2 つの文字列カラムがあり、"name" にはパラメータ名が、"value" には実際のパラメータ値が格納されます。全パラメータ名は SQL Anywhere 9 のオンラインヘルプに記載されており、各パラメータに対応する値は、単一の SELECT 文で取り出すことができます。

たとえば、「SELECT value FROM #hook_dict WHERE name = 'SQL error code'」という式では、'-196' などのエラー・コードが格納された、単一の文字列値が返ります。あるフック・プロシージャに 5 つのパラメータが渡され、5 つの値をすべて取得したい場合には、5 つの SELECT 文を別個に記述する必要があります。

Note

関連するフック・プロシージャで、sp_hook_dbmlsync_download_sql_error というものがあります。このプロシージャでは、エラーを修正して処理を継続できますが、こちらはお勧めできません。ある 1 つのローに影響するエラーが原因で、dbmlsync がそのテーブルのすべてのローをスキップする場合があるからです。sp_hook_dbmlsync_download_sql_error プロシージャをまったく作成しなかった場合のデフォルトのアクションは、sp_hook_dbmlsync_download_fatal_sql_error が存在する場合はこれを呼び出し、その後でダウンロードをロールバックするというものです。その方法をお勧めします。sp_hook_dbmlsync_download_sql_error を気にする必要はありません。

次に示すのは、致命的なダウンロード・エラーのフック・プロシージャの例です。これは、「7.6.4.5　アップロード・エラーの処理」で説明した ml_report_error プロシージャに似ています。

```
CREATE PROCEDURE sp_hook_dbmlsync_download_fatal_sql_error()
BEGIN
   INSERT dbmlsync_sql_error VALUES (
      ( SELECT value FROM #hook_dict WHERE name = 'MobiLink user' ),
      DEFAULT,
      DEFAULT,
      ( SELECT value FROM #hook_dict WHERE name = 'publication_0' ),
      ( SELECT value FROM #hook_dict WHERE name = 'table name' ),
      ( SELECT value FROM #hook_dict WHERE name = 'SQL error code' ),
      ( SELECT value FROM #hook_dict WHERE name = 'script version' ) );
END; -- sp_hook_dbmlsync_download_fatal_sql_error
```

Tip

#hook_dict のパラメータ名のコーディングには注意が必要です。単語の区切りがアンダースコアのもの（たとえば 'publication_0'）と、スペース区切りのもの（たとえば 'MobiLink user'）があります。また、

最初の（または唯一の）パブリケーションの番号は 1 ではなく 0 だということにも注意してください。パラメータ名のコーディングをミスした場合でも、エラーにはなりません。#hook_dict に一致するローがないために NULL 値が取得されるだけです。

これで、統合データベースの dbmlsync_sql_error テーブルを確認すれば、リモート・データベースでダウンロードが失敗したことを判断できます。先ほどの例では、カラム値がリモート・データベースの UNIQUE 制約に違反していました。この問題を修正する方法の 1 つとしては、統合データベースで値を変更するという手があります。ここで 2 つ目の疑問が沸き起こります。前にロールバックされた他のローすべてと、修復されたローを、次のダウンロードに確実に含めるにはどうすればよいかということです。

タイムスタンプ・ダウンロードの手法を使用する download_cursor スクリプトがあれば、答えは簡単です。単に、同期を再度実行すればよいのです。last_download パラメータは前回の同期と同じなので、ダウンロード・ストリーム用として同じローが選択されます。last_download パラメータは Mobile Link クライアントから得るもので、ダウンロードの失敗では変更されません。

一方、これより複雑なダウンロード手法を使用している場合、統合データベースでは前回のダウンロードがコミットされたがリモート・データベースではロールバックされたというときに、問題が生じることがあります。たとえば、複数のリモート・データベースに含まれている統合データベースのローの分割（サブセット）が、リモート・データベースによって異なる場合があります。そして、前回のダウンロードは、古いサブセットの削除と新しいサブセットの挿入によって分割を変更しようとするものだったということがあります。リモート・データベースではダウンロードが失敗したが統合データベースでは成功したという場合に、実際は分割が変更されていないのに、分割が変更されたという不正な情報が統合データベースに含まれている可能性があります。次の同期の実行時には、Mobile Link スクリプトで前回の失敗を検出して、同じ分割変更が次のダウンロード・ストリームで確実に送信されるように特別なアクションを行うことが必要な場合があります。

Tip

SendDownloadACK が 'OFF' の場合で、かつフック・プロシージャを使用してエラーを記録していない場合でも、統合データベースの Mobile Link スクリプトで、前回のダウンロードがリモート・データベースで失敗したかどうかを判断できます。それには、begin_publication スクリプトで、@last_upload パラメータと @last_download パラメータを比較するだけです。@last_upload が @last_download より大きい場合、サーバではすべてうまくいったと思っていたとしても、実は前回のダウンロードがリモート・データベースで失敗したということです。この方法を使用すれば、SendDownloadACK を 'OFF' に設定したままにでき、Mobile Link サーバと統合データベースの負荷を軽減できます。これは特に、低速なコンピュータでリモート・データベースが動作している場合に有用です。詳細と具体例については、「7.7　Mobile Link のシステム・テーブル」を参照してください。

Note

前の「ヒント」で説明した方法は、双方向同期でのみ使用でき、UploadOnly 拡張オプションが有効な場合には当てはまりません。この章全体が、すべての同期セッションが双方向であることを前提にしており、アップロード専用同期やダウンロード専用同期については説明していません。

3つ目のダウンロード・エラーは、Mobile Link クライアントによって検出され、デフォルトでは通知なしで処理される種類のものです。ダウンロード・ストリームによって参照整合性違反が生じる場合、違反の原因となる子テーブルまたは従属テーブルのすべてのローが自動的に削除されるため、参照整合性違反は回避されます。これは、リモート・データベースで外部キーがどのように定義されているかや、原因となるローがダウンロードされた中にあるのか、それともリモート・データベースに既に存在するのかとは関係なく行われます。

たとえば、tparent テーブルと tchild テーブルに親子関係がある状況で、download_delete_cursor が tparent のローに対する削除を送信した場合、Mobile Link クライアントは、tchild 内の対応するすべてのローを自動的に削除します。Mobile Link クライアントの診断ログに次のようなメッセージが生成されますが、これはエラーとは見なされず、ダウンロードはコミットされます。

```
I. 08/21 17:18:38. Resolving referential integrity violations on table tchild,
   role tparent
I. 08/21 17:18:38. delete from "DBA"."tchild" from "DBA"."tchild" ft where not
   exists ( select * from "DBA"."tparent" pt where  ( ft."key_1" = pt."key_1"
   or ft."key_1" is NULL )  )
I. 08/21 17:18:38. 1 rows deleted.
I. 08/21 17:18:38. COMMIT
```

言い換えると、ON DELETE RESTRICT と明示的に指定した場合でも、ON DELETE CASCADE を指定した場合と同様に外部キー関係が処理されます。ダウンロードした子ローに対応する親ローがない場合も、同じことが生じます。Mobile Link クライアントは、まず新しい子ローを挿入した後で、上記のメッセージと同種のロジックを使用してそのローを削除します。

この動作が気にいらない場合は、変更することが可能です。次の1行のフック・プロシージャをリモート・データベースに追加するだけです。Mobile Link クライアントは、このプロシージャが存在することを確認し、参照整合性エラーを検出したときにこれを呼び出します。

```
CREATE PROCEDURE sp_hook_dbmlsync_download_ri_violation()
BEGIN
   RAISERROR 19999;
END;
```

この特定の sp_hook_dbmlsync_download_ri_violation プロシージャが呼び出されると、RAISERROR 文の結果、Mobile Link クライアントはダウンロード・ストリームを次のようにロールバックします。

```
I. 08/22 10:21:22. execute "DBA".sp_hook_dbmlsync_download_ri_violation
E. 08/22 10:21:22. SQL statement failed: (-19999) RAISERROR executed:
E. 08/22 10:21:22. Error while executing hook procedure
   sp_hook_dbmlsync_download_ri_violation.
I. 08/22 10:21:22. ROLLBACK
I. 08/22 10:21:22. Download stream processing failed
I. 08/22 10:21:22. Sending a failure status to the MobiLink server.
```

ダウンロード・ストリームで発生した参照整合性違反についての診断情報を保存したい場合には、sp_hook_dbmlsync_download_log_ri_violation という名前のフック・プロシージャを別途作成できます。このプロシージャは、Mobile Link クライアントが違反を検出したときに実行されますが、別の接続が使用されるので、残りのダウンロードがロールバックされる場合でも、データベースに挿入した内容は自動的にコミットされます。

次に示す例は、参照整合性違反を記録するためのリモート・データベース・テーブルと、それに対して中身を設定する sp_hook_dbmlsync_download_log_ri_violation プロシージャです。

```
CREATE TABLE dbmlsync_ri_violation (
   ml_username  VARCHAR ( 128 ) NOT NULL,
   inserted_at  TIMESTAMP NOT NULL DEFAULT TIMESTAMP,
   unique_id    UNSIGNED BIGINT NOT NULL DEFAULT GLOBAL AUTOINCREMENT ( 1000000 ),
   publication_name       VARCHAR ( 128 ) NOT NULL,
   foreign_key_table_name VARCHAR ( 128 ) NOT NULL,
   primary_key_table_name VARCHAR ( 128 ) NOT NULL,
   role_name              VARCHAR ( 128 ) NOT NULL,
   script_version         VARCHAR ( 128 ) NOT NULL,
   PRIMARY KEY ( ml_username, inserted_at, unique_id ) );

CREATE PROCEDURE sp_hook_dbmlsync_download_log_ri_violation()
BEGIN
   INSERT dbmlsync_ri_violation VALUES (
      ( SELECT value FROM #hook_dict WHERE name = 'MobiLink user' ),
      DEFAULT,
      DEFAULT,
      ( SELECT value FROM #hook_dict WHERE name = 'publication_0' ),
      ( SELECT value FROM #hook_dict WHERE name = 'Foreign key table' ),
      ( SELECT value FROM #hook_dict WHERE name = 'Primary key table' ),
      ( SELECT value FROM #hook_dict WHERE name = 'Role name' ),
      ( SELECT value FROM #hook_dict WHERE name = 'script version' ) );
END; -- sp_hook_dbmlsync_download_log_ri_violation
```

4つ目のダウンロード・エラーも、参照整合性違反にまつわる種類のものですが、こちらは通知なしで処理されるものではありません。同期の対象でないテーブルに関連する参照整合性のエラーが検出された場合、致命的なエラーと診断され、sp_hook_dbmlsync_download_fatal_sql_error が呼び出さ

れます。たとえば、親テーブルはパブリケーションに含まれているが、子テーブルは含まれていない場合、対応する子ローを持つ親ローに対する削除をダウンロードしようとすると、これは失敗します。dbmlsync が子ローを通知なしで削除することはありません。同期処理に含まれていないからです。

この種のエラーは、ダウンロード・ストリームが終了に至って最後のコミットが実行されるまでは検出されないため、sp_hook_dbmlsync_download_fatal_sql_error に渡されるテーブル名は空です。これは難点です。問題を突き止めるうえで、フック・プロシージャが役に立たないからです。幸いなのは、たとえフック・プロシージャを作成しなくても、問題があることは少なくとも検出できるということです。

Section

7.7 Mobile Linkのシステム・テーブル

Mobile Link クライアントとサーバの両コンポーネントは、同期の状態についての情報を、リモート・データベースと統合データベースの両方のシステム・テーブルに保持します。これらのテーブルで最も重要なのは、リモート・データベースの SYSSYNC と、統合データベースの ml_subscription です。次に示すのはその定義です。

```
CREATE TABLE SYS.SYSSYNC (
   sync_id            UNSIGNED INT NOT NULL,
   type               CHAR ( 1 ) NOT NULL,
   publication_id     UNSIGNED INT NULL,
   progress           NUMERIC ( 20 ) NULL,
   site_name          CHAR ( 128 ) NULL,
   "option"           LONG VARCHAR NULL,
   server_connect     LONG VARCHAR NULL,
   server_conn_type   LONG VARCHAR NULL,
   last_download_time TIMESTAMP NULL,
   last_upload_time   TIMESTAMP NOT NULL DEFAULT 'jan-1-1900',
   created            NUMERIC ( 20 ) NULL,
   log_sent           NUMERIC ( 20 ) NULL,
   generation_number  INTEGER NOT NULL DEFAULT 0,
   extended_state     VARCHAR ( 1024 ) NOT NULL DEFAULT '',
   PRIMARY KEY ( sync_id ) );

CREATE TABLE dbo.ml_subscription (
   user_id            INTEGER NOT NULL,
   subscription_id    VARCHAR ( 128 ) NOT NULL,
   progress           NUMERIC ( 20 ) NOT NULL DEFAULT 0,
   publication_name   VARCHAR ( 128 ) NOT NULL DEFAULT '',
   last_upload_time   TIMESTAMP NOT NULL DEFAULT '1900/01/01 00:00:00',
```

```
    last_download_time TIMESTAMP NOT NULL DEFAULT '1900/01/01 00:00:00',
    PRIMARY KEY ( user_id,
                  subscription_id ) );
```

「7.4.5　SYSSYNC とパブリケーション・デフォルト」で説明したように、SYSSYNC テーブルには、各サブスクリプションに対してそれぞれ 1 つずつ、特別なローが格納されています。このローは、type カラムが 'D' で、かつ publication_id と site_name が null 以外の値であることにより識別できます。publication_id カラムは、対応するパブリケーション名が格納された SYSPUBLICATION のローを指し、site_name には、このリモート・データベースのユニークな Mobile Link ユーザ名が格納されています。たいていは、各リモート・データベースにつき、site_name 値は 1 つしかなく、またパブリケーションは 1 つしかありません。つまり、SYSSYNC には「サブスクリプション・ロー」が 1 つしかないことが大半です。sync_id カラムは人工プライマリ・キーであり、サブスクリプション・ローでは「サブスクリプション ID」として捉える必要があります。

統合データベースでは、対応するローが ml_subscription テーブルにあります。user_id カラムは、そのリモート・データベースに対応する Mobile Link ユーザ名が格納された ml_user テーブルのローを指し、その名前は SYSSYNC.site_name に一致します。ml_subscription.subscription_id カラムには、リモート・データベースの SYSSYNC.sync_id を文字列に変換した値が格納されます。多くの場合、ml_subscription には、各リモート・データベースにつき 1 つのローだけがあります。リモート・データベースに、組み合わせの異なるテーブルに対する 2 つの異なるサブスクリプションがある場合には、ml_subscription に含まれるローは 2 つになります。

SYSSYNC のサブスクリプション・ローには、統合データベースとリモート・データベースの間の同期の状態についての重要な情報も保持されます。progress カラムには、リモート・データベースのトランザクション・ログ・ファイルのバイト・オフセットが格納されます。最後に成功したアップロードで到達した位置です。この値は、統合データベースの ml_subscription テーブルにもアップロードおよび格納されるため、次の同期を開始する前の再チェックが簡単になります。値が一致する場合、次の同期では、リモート・トランザクション・ログのスキャンはその位置から開始されます。

値が一致しない場合、問題があるかどうかは状況によります。デフォルトの動作では、統合データベースの ml_subscription.progress カラムに格納されているログ・オフセット値が使用されます。リモート・データベースのトランザクション・ログにそのオフセットが見つからない場合、または有効なオフセットでない場合には、実際に問題があります。dbmlsync の -ra オプションか -rb オプションを使用して解決できる場合や、ml_subscription の値を修正して一致させることができる場合や、ml_subscription のローを削除することで問題を解決できる場合があります。あるいは、リモート・データベースでサブスクリプションを削除および再作成して、同期を改めて開始することが必要な場合もあります。

SYSSYNC.last_download_time カラムには、前回ダウンロードに成功した日付と時刻が格納されています。この日付と時刻は、リモート・サーバではなく統合データベース・サーバに対応するものです。これは、前回のアップロード・ストリームがコミットされた直後でかつダウンロード・ストリームの処理が始まる前に取得されるので、前回のダウンロードの開始後に行われた統合データベースの変更よりも

早い値です。したがって、最後のダウンロード以降に変更されたローを判断するには非常に有効であり、その目的のために、Mobile Link スクリプトのパラメータとして用意されています。この値は、リモート・データベースの SYSSYNC テーブルと、統合データベースの ml_subscription テーブルの両方に格納されています。

SYSSYNC には、前回成功したアップロードの日付と時刻を保持する last_upload_time カラムもあります。この値は、統合データベース・サーバの日付と時刻に対応するものであり、ml_subscription テーブルにも格納されています。最後のダウンロードがうまくいった場合、last_upload_time は last_download_time より小さくなります。アップロード・ストリームはダウンロードより前に処理されるからです。前回のアップロードはうまくいったがダウンロードは失敗した場合、last_upload_time は last_download_time より大きくなります。

次に示すのは、これらのカラムを表示するためにリモート・データベースで実行できるクエリです。

```
SELECT SYSSYNC.site_name,
       SYSSYNC.sync_id,
       SYSSYNC.progress,
       SYSPUBLICATION.publication_name,
       SYSSYNC.last_upload_time,
       SYSSYNC.last_download_time
  FROM SYSPUBLICATION
  JOIN SYSSYNC
    ON SYSPUBLICATION.publication_id = SYSSYNC.publication_id
 WHERE SYSSYNC.type = 'D'
   AND SYSSYNC.publication_id IS NOT NULL
   AND SYSSYNC.site_name IS NOT NULL;
```

次に示すのは、統合データベースの対応するクエリです。万事うまくいった場合、これら2つのクエリでは、同じサブスクリプションに対して同じ値が示されるはずです。

```
SELECT ml_user.name,
       ml_subscription.subscription_id,
       ml_subscription.progress,
       ml_subscription.publication_name,
       ml_subscription.last_upload_time,
       ml_subscription.last_download_time
  FROM ml_user
  JOIN ( SELECT a.user_id,
                a.subscription_id,
                a.progress,
                a.publication_name,
                a.last_upload_time,
                a.last_download_time
           FROM ml_subscription AS a
          WHERE a.progress = ( SELECT MAX ( b.progress )
                                 FROM ml_subscription AS b
```

```
                              WHERE b.user_id = a.user_id
                                AND b.publication_name = a.publication_name ) )
      AS ml_subscription
   ON ml_user.user_id = ml_subscription.user_id;
```

2番目のクエリの方が複雑なのは、リモート・データベースでサブスクリプションが削除および再作成された場合に、ml_subscription テーブルでは古いデータが自動的に削除されないからです。代わりに、次の同期で、user_id と publication_name が同じで subscription_id が異なる新しいローが ml_subscription に挿入されます。この新しいローは「アクティブ」なローで、有効な値を持つ唯一のローです。他のローは「非アクティブ」であり、既に有効ではありません。アクティブなローは、user_id と publication_name のそれぞれの組み合わせについて、progress カラムの値が最大です。

統合データベースでは、ml_subscription.last_upload_time カラムと last_download_time カラムは異なるタイミングで格納されます。last_upload_time カラムは、アップロード・ストリームがコミットされると直ちに更新され、その新しい値に対するクエリは、同じ同期に対する prepare_for_download やそれ以降の Mobile Link スクリプト、および次の同期に対する end_upload までのスクリプトで実行できます。

SendDownloadACK 拡張オプションが 'OFF'（デフォルト）の場合、last_download_time カラムは、次の同期が開始するまでは更新されません。その後で、すべての Mobile Link スクリプトで検索できるようになります。SendDownloadACK が 'ON' の場合、last_download_time カラムは、ダウンロード確認を受信すると直ちに更新され、新しい値の検索は、同じ同期に対する end_publication スクリプト、ならびに次の同期に対する end_download までのスクリプトで実行できます。

ml_subscription.last_upload_time と last_download_time を比較すると、前回のアップロードはうまくいったがダウンロードは失敗した、という状態かどうかを判断できます。ただし、この比較は、Mobile Link アップロード・スクリプトのいずれかで、end_upload までに行う必要があります。それ以降は、last_upload_time に新しい値が入り、比較は有効ではありません。

ml_subscription の last_upload_time と last_download_time を比較する場所として最適なのは begin_publication スクリプトです。実のところ、データベースに問い合わせる必要もありません。begin_publication スクリプトには、現在のサブスクリプションに対応する値がパラメータとして用意されているからです。

次の例は、前回のダウンロードが失敗したかどうかを SendDownloadACK の設定にかかわらず判断する begin_publication スクリプトを示します。また、同期がまだ行われていないという状況にも対処します。

```
CALL ml_add_connection_script ( '1', 'begin_publication',
'CALL ml_begin_publication ( ?, ?, ?, ?, ? )' );

CREATE PROCEDURE ml_begin_publication (
  INOUT @generation_number  INTEGER,
  IN    @ml_username        VARCHAR ( 128 ),
```

```
  IN    @publication_name   VARCHAR ( 128 ),
  IN    @last_upload        TIMESTAMP,
  IN    @last_download      TIMESTAMP )
BEGIN
   IF @last_upload > '1900-01-01 00:00:00' THEN
      IF @last_upload > @last_download THEN
         MESSAGE STRING (
            'Previous download FAILED for subscription to publication "',
            @publication_name,
            '" by MobiLink User "',
            @ml_username,
            '".' ) TO CONSOLE;
      ELSE
         MESSAGE STRING (
            'Previous download SUCCEEDED for subscription to publication "',
            @publication_name,
            '" by MobiLink user "',
            @ml_username,
            '".' ) TO CONSOLE;
      END IF
   ELSE
      MESSAGE STRING (
         'This is the FIRST SYNCHRONIZATION for subscription to publication "',
         @publication_name,
         '" by MobiLink user "',
         @ml_username,
         '".' ) TO CONSOLE;
   END IF;
END;
```

Note

begin_publication スクリプトは、「7.6.4.7　削除のダウンロード」の例で示した、シャドー・テーブルのローの削除を行うのと同じ場所です。しかし、その例では ml_subscription からの SELECT が必要でした。現在のリモート・データベースだけではなく、全リモート・データベースに対する最も古いダウンロードを判断する必要があったからです。

Section

7.8 Mobile Linkモニタ

Mobile Link モニタとは、動作中の Mobile Link サーバで何が行われているかを表示する管理ツールです。リモート・データベースで動作中の Mobile Link クライアントにはアクセスできませんが、同期セッションの統合データベース側については、グラフィック・モードとテキスト・モードの両方で、余すところなく表示されます。

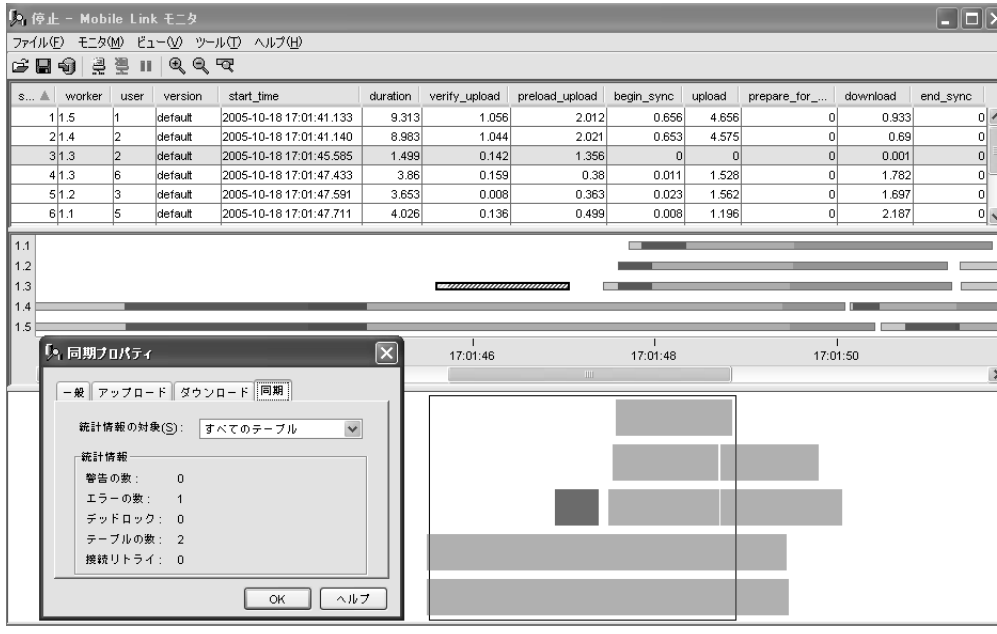


図 7-2　Mobile Link モニタ

Mobile Link モニタの画面は 3 つのウィンドウ枠で構成されます。上部の［詳細テーブル］ウィンドウ枠は、モニタが Mobile Link サーバに接続して以降に開始されたすべての同期セッションを示す、スクロールするリストです。中央の［チャート］ウィンドウ枠は、それらの同じセッションを鮮明に色分けして示したもので、時間の経過に合わせて右から左にスライドしていきます。下部の［概要］ウィンドウ枠には、全セッションのもっと大きなビューが、上の 2 つのウィンドウ枠のような詳細なしで表示されます。

次は、図 7-2 に表示されている内容の説明です。

- 上部のウィンドウ枠には、各セッションのMobile Linkユーザ名（[user]）、正確な開始時刻（[start time]）、セッション全体での経過秒数（[duration]）が示される。0と表示されているカラムは、対応するMobile Linkスクリプトが定義されていないか、またはなんらかの問題があったことを示すのが普通である
- 中央のウィンドウ枠には、各同期が色付きのバーで表示される。各バーは、アップロード・プロセスとダウンロード・プロセスの段階の違いに応じて色分けされている。デフォルトでは、アップロード段階はさまざまな濃度の緑色で示され、ダウンロード段階はさまざまな濃度の赤色で示される
- 中央のウィンドウ枠には、約10秒分のセッションが示される。この範囲は、［ビュー］－［移動］メニュー・オプションで変更できる
- 下部のウィンドウ枠には、モニタが情報の収集を開始して以降の時間全体を表す。下部のウィンドウ枠の中の四角い枠は、中央のウィンドウ枠に示されている時間に対応する。時間の経過と共に、この四角い枠はサイズが小さくなっていく。下部のウィンドウ枠は比較的長い時間間隔を表すのに対し、中央のウィンドウ枠は固定された時間間隔を表すからである
- 失敗したセッションは、中央のウィンドウ枠では斜線の模様付きで示され、下部のウィンドウ枠では鮮明な赤色で示される
- 成功したセッションは、［チャート］ウィンドウ枠ではべた塗りの色で表示され、下部のウィンドウ枠では鈍い灰色で表示される
- ［チャート・スクロールの一時停止］ボタンをクリックすると、中央と下部のウィンドウ枠の表示を停止できるが、情報の収集は継続される。この状態で、左右にスクロールして問題を探したり、個々のセッションを調査したりできる。上部のウィンドウ枠には新しいセッションが引き続き表示される。一時停止を解除すると、中央と下部のウィンドウ枠は直ちに最新の表示に戻る
- 中央と下部のウィンドウ枠では、Mobile Linkサーバで実行されている各「ワーカ・スレッド」に対してそれぞれ1行が割り当てられている。各ワーカ・スレッドは、同期要求を一度に1つ受け入れる。異なるリモート・データベースからのセッションが重複した場合、それぞれ別のワーカ・スレッドが割り当てられるため、それらを並列で処理してスループットを向上できる
- 色付きのバーをダブルクリックすると、個々のセッションについての詳細情報を参照できる。その場合、色付きのバーは太い輪郭で表示され、選択中であることが示される。上部のウィンドウ枠の対応する行も選択され、［同期プロパティ］ウィンドウが図7-2のようにポップアップ表示される
- ［同期プロパティ］ウィンドウでは、ローの数やエラーの数など、選択したセッションについての詳細情報が、複数のタブに表示される

図7-2では、ワーカ・スレッド1.4の失敗した同期が選択されています。［同期プロパティ］ウィンドウには、エラーが1つ検出されたことが示されています。上部のウィンドウ枠には、これはMobile Linkユーザ名"2"のセッションで、17:01:45に開始されたものであることが示されています。これら3つの情報（ワーカ・スレッド、Mobile Linkユーザ名、時刻）を使用して、Mobile Linkサーバの診断テキスト・ファイルで原因を探ることができます。

```
I. 08/26 12:57:39. <thread 1.4>: ASA synchronization request from: 2 (version: 1)
E. 08/26 12:57:39. <1.4> [2]: Error: [-10053] The user name '2' is already
  synchronizing.  Concurrent synchronizations using the same user name are
  not allowed
...
I. 08/26 12:57:39. <1.4> [2]: ROLLBACK Transaction: begin_synchronization
```

この場合は、同じリモート・データベースから2つの同期セッションが重複して開始されていました。それは認められません。次は、このエラーをMobile Linkクライアント側から見た場合を示します。

```
I. 08/26 12:57:39. Waiting for MobiLink to apply upload
I. 08/26 12:57:39. Download stream: received 10 bytes from MobiLink Server.
E. 08/26 12:57:39. The MobiLink user 2 was already being synchronized. Concurrent
   synchronizations for the same user are not allowed.
...
E. 08/26 12:57:40. Download aborted
```

Section

7.9 この章のまとめ

この章では、Mobile Link同期のクライアント・プログラムとサーバ・プログラムがどのように連係して、単一の統合データベースと多数のリモート・データベースの間でデータを分散させるかについて説明しました。また、パブリケーションとサブスクリプションに関するリモート・データベース設定、およびMobile Linkクライアントの動作の制御に利用できる多数のオプションについても説明しました。アップロード・ストリームの処理、ダウンロード・ストリームの生成、競合の処理を行うMobile Linkサーバ・スクリプトについても述べました。

アップロードとダウンロードの段階でエラーを処理する方法や、同期の状態を記録するMobile Linkシステム・テーブルの詳細についても説明しました。Mobile Linkモニタにも触れました。

次の章では、SQLコードをデータベース自体に格納されるモジュールにパッケージ化するというトピックに移ります。すなわち、プロシージャ、関数、トリガ、イベントについてです。

Chapter

8

パッケージ

Section

8.1 概要

この章では、ローカル宣言と例外処理ロジックを含むBEGINブロックにSQL文をまとめる方法について説明します。このBEGINブロックは、そのまま実行したり、後で実行するようにストアド・モジュールとしてデータベースに格納したりできます。ストアドSQLモジュールの形式としては、呼び出されるプロシージャ、式で参照される関数、INSERT操作、UPDATE操作、DELETE操作で自動的に起動されるトリガがあります。

SQL Anywhereは、別の接続で非同期に実行するイベントもサポートしています。イベントは、スケジュールに従って自動的に起動したり、特定の条件が発生したときに起動したり、TRIGGER EVENT文で明示的に起動したりできます。

また、この章では、BEGINブロックの中に複雑な論理を実装するための複数文の制御構造体IF、CASE、WHILEについても説明します。EXECUTE IMMEDIATE文では、SQL文を作成しながら実行できます。CREATE VARIABLE文では、1つの接続で実行されているすべてのモジュールの間でデータを渡せます。

Section

8.2 BEGINブロック

BEGINブロックは、SQL文をローカル宣言と例外処理ロジックと一緒にカプセル化するために使用します。BEGINブロックを使って、データベースに格納するSQLモジュール（プロシージャ、関数、トリガ、イベント）や、ISQLなどのクライアント・アプリケーションから直接実行するための文のバッチを作成します。

BEGINブロックには、ラベル付きとラベルなしの2つの形式があります。ラベルはLEAVE文のターゲットとして使用できます。

```
<begin_block>     ::= BEGIN [ [ NOT ] ATOMIC ]
                          [ <declaration_list> ]
                          [ <statement_list> ]
                          [ <exception_handler> ]
                      END
                    | <begin_label> ":"
                      BEGIN [ [ NOT ] ATOMIC ]
                          [ <declaration_list> ]
                          [ <statement_list> ]
```

```
                        [ <exception_handler> ]
                    END [ <begin_label> ]
<begin_label>     ::= <leave_statement> で使われる可能性のある <identifier>
<identifier>      ::= 「第1章　作成」の <identifier> を参照
<leave_statement> ::= LEAVE <begin_label>
                    | LEAVE <for_label>
                    | LEAVE <while_label>
<for_label>       ::= 「第6章　フェッチ」の <for_label> を参照
```

ATOMIC キーワードと NOT ATOMIC キーワードは、BEGIN ブロックのデフォルトの実行モードがアトミックか非アトミックかを指定します。アトミック・オペレーションとは、完全に実行されるか、まったく実行されないオペレーションです。アトミックな BEGIN ブロックの中の 1 つの文が失敗し、それによってブロックが異常終了した場合、その BEGIN ブロックの中のコードによって行われたデータベース変更はすべて取り消されます。COMMIT 文と ROLLBACK 文は、BEGIN ブロックのアトミックな性質と全体的に矛盾するので、アトミックな BEGIN ブロックの中では使えません。

非アトミックなコード・ブロックでは、それぞれの文が独立した実行単位として扱われるので、1 つの文が失敗しても、その他の文によって行われたデータベース変更は取り消されません。

Note

ROLLBACK TO SAVEPOINT 文はアトミック・オペレーションに入れられますが、制限があります。詳細については、「9.4　セーブポイントとサブトランザクション」を参照してください。

NOT ATOMIC がデフォルトです。ただし、実際の実行モードは実行時に決まります。BEGIN ブロックが大きなアトミック・オペレーションの一部として実行される場合、内部の BEGIN ブロックにも前述のアトミック・オペレーションの制約が適用されます。これは、NOT ATOMIC キーワードが内部の BEGIN ブロックで明示的に指定されている場合も同様です。トリガはすべてこのカテゴリに入ります。INSERT 文、UPDATE 文、DELETE 文はすべてアトミック・オペレーションなので、これらの文によって起動されるトリガはアトミック・オペレーションに含まれます。もう 1 つの例が、ATOMIC キーワードを使っている外部のプロシージャから呼び出されるプロシージャ内の BEGIN ブロックです。内部の BEGIN ブロックは、コーディング方法に関係なく、アトミック・オペレーションの一部として実行されます。

DECLARE 文を BEGIN ブロックの先頭、つまり実行可能な文の前に配置して、ローカル変数、カーソル、名前付き例外条件、ローカル・テンポラリ・テーブルを作成できます。カーソルの DECLARE 文については、「第6章　フェッチ」を参照してください。テンポラリ・テーブルについては、「1.15.2　ローカル・テンポラリ・テーブル」を参照してください。例外条件については、この章の次の項で説明します。

```
<declaration_list>              ::= <declaration> ";" { <declaration> ";" }
<declaration>                   ::= <declare_local_variable>
```

```
                                  | <declare_cursor>
                                  | <declare_exception_name>
                                  | <declare_local_temporary_table>
<declare_local_variable>        ::= DECLARE <local_variable_name> <data_type>
<local_variable_name>           ::= <identifier>
<data_type>                     ::=「第1章　作成」の<data_type>を参照
<declare_cursor>                ::=「第6章　フェッチ」の<declare_cursor>を参照
<declare_local_temporary_table> ::=「第1章　作成」の<declare_local_temporary_table>を参照
```

BEGINブロックの先頭で定義されているローカル変数は、ブロックが開始するときに作成され、終了するときに破棄されます。ローカル変数のスコープには、同じBEGINブロック内のすべての実行可能な文と、ネストされたBEGINブロック内のすべての文が含まれます。

スコープのネストはありません。同じ変数名を、ネストされたBEGINブロックの中で宣言されるローカル変数に使ったり、カーソルのFORループで作成される暗黙的なローカル変数の名前として使用したりすることはできません（カーソルのFORループの詳細については、「6.3　カーソルのFORループ」を参照してください）。ただし、カラムと同じ名前のローカル変数は、そのカラムが収められているテーブルを参照するSQL文の中では、「隠され」ます。次は、変数とカラムが同じ名前「scope」を共有するとどうなるかを示す例です。

```
CREATE TABLE t1 (
   key_1     INTEGER NOT NULL,
   scope     VARCHAR ( 1 ) NOT NULL,
   PRIMARY KEY ( key_1 ) );

INSERT t1 VALUES ( 1, 'B' );

BEGIN
   DECLARE scope VARCHAR ( 1 );
   SET scope = 'A';
   MESSAGE STRING ( 'scope #1: ', scope ) TO CONSOLE;
   MESSAGE STRING ( 'scope #2: ',
      ( SELECT scope FROM t1 WHERE key_1 = 1 ) ) TO CONSOLE;
   MESSAGE STRING ( 'scope #3: ', scope ) TO CONSOLE;
END;
```

1番目と3番目のMESSAGE文では値'A'が表示されるのに対し、2番目のMESSAGE文では、SELECT文が変数scopeではなくカラムt1.scopeを参照するため、'B'が表示されます。

Tip

カラムの名前と混同しないように、ローカル変数の名前の先頭には文字＠を付けてください（DECLARE @scope VARCHAR (1) など）。

BEGINブロック文のリストは、宣言のリストや例外ハンドラと同様に、オプションです。文のリストは、セミコロンで区切った複数の文からなる1つ以上の実行可能な文で構成されます。

```
<statement_list>       ::= <executable_statement>
                         | <statement_list> ";" <executable_statement>
                         | <statement_list> ";"
<executable_statement> ::= DECLARE と EXCEPTION 以外
```

BEGINブロックは、データベースの中に格納するプロシージャやトリガなどのモジュールを作成するために使われることがほとんどですが、アプリケーション・プログラムでも使用できます。次は、後で実行するためにBEGINブロックをデータベース・サーバに送るPowerBuilderスクリプトの例です。

```
string ls_sql

ls_sql &
   = "BEGIN " &
   + "   DECLARE @x VARCHAR ( 100 ); " &
   + "   SET @x = 'Hello, World'; " &
   + "   MESSAGE STRING ( @x ) TO CONSOLE; " &
   + "END;"

EXECUTE IMMEDIATE :ls_sql USING SQLCA;
```

Section

8.3 例外ハンドラ

各BEGINブロックの終わりに例外ハンドラを置くこともできます。例外ハンドラは、キーワードEXCEPTIONとそれに続く1つ以上のWHEN句で構成されます。各WHENキーワードの後ろには、1つ以上の例外条件、キーワードTHEN、1つ以上の実行可能な文が続きます。例外ハンドラの前にある実行可能な文のどこかで名前付き例外のいずれかが発生すると、制御のフローが対応するWHEN句の文に直ちに送られ、その文が終了した後でBEGINブロックの実行が終了します。

WHEN句で名前が指定される各例外条件は、SQLSTATEの単一の5文字値に対応し、例外の名前は、特殊な形式のDECLARE文を使ってBEGINブロックの先頭で定義します。次は、DECLARE文の構文と、例外ハンドラの構文です。

```
<declare_exception_name>  ::= DECLARE <exception_name>
                                  EXCEPTION FOR SQLSTATE <sqlstate_error_value>
<exception_name>          ::= <exception_handler> 内で使うための <identifier>
<sqlstate_error_value>    ::= SQLSTATE エラー値を含む文字列リテラル
```

```
<exception_handler>       ::= EXCEPTION <when_exception_list>
<when_exception_list>     ::= <when_exception> { <when_exception> }
<when_exception>          ::= WHEN <exception_name_list>
                                  THEN <statement_list>
                            | WHEN OTHERS
                                  THEN <statement_list>
<exception_name_list>     ::= <exception_name> { "," <exception_name> }
```

次は、実行すると「Data conversion error!」が表示される非常に簡単な例です。

```
BEGIN
DECLARE @date           DATE;
DECLARE @sqlstate_53018 EXCEPTION FOR SQLSTATE '53018';
SET @date = 'xxx';
EXCEPTION
   WHEN @sqlstate_53018 THEN
      MESSAGE STRING ( 'Data conversion error!' ) TO CLIENT;
END;
```

名前付き例外条件のリストの代わりにキーワードOTHERSを使って、上位のWHEN句で名前が指定されていない例外に対応するWHEN句を作成することもできます。WHEN OTHERS句は1つのみとし、この句の後ろにはWHEN句を置かないでください。最初のWHEN OTHERSの後ろにあるWHEN句は実行されません。

500を超えるさまざまな種類の例外があるので、WHEN OTHERSはまさに便利な道具です。次は、ゼロ除算例外をトラップして「SQLSTATE = 22012」が表示される例外です。

```
BEGIN
DECLARE @number         INTEGER;
DECLARE @date           DATE;
DECLARE @sqlstate_53018 EXCEPTION FOR SQLSTATE '53018';
SET @number = 1 / 0;
SET @date = 'xxx';
EXCEPTION
   WHEN @sqlstate_53018 THEN
      MESSAGE STRING ( 'Data conversion error!' ) TO CLIENT;
   WHEN OTHERS THEN
      MESSAGE STRING ( 'SQLSTATE = ', SQLSTATE ) TO CLIENT;
END;
```

特定の例外条件のための特殊なロジックなしで、すべての例外を単にトラップして診断する場合は、1つのWHEN OTHERS句だけで済みます。ERRORMSG関数をWHEN OTHERS句で呼び出せば、例外ハンドラがない場合に表示されていた正確なエラー・メッセージを取得できます。次は、「値xxxをデータ型timestampに変換できません」が表示される例です。

```
BEGIN
DECLARE @date DATE;
SET @date = 'xxx';
EXCEPTION
   WHEN OTHERS THEN
      MESSAGE ERRORMSG() TO CLIENT;
END;
```

WHEN 句の中で SQLSTATE 値と ERRORMSG() 値の両方を表示することは容易ではありません。なぜなら、一方の値の参照によってもう一方の値がデフォルト値に設定されることがあるためです。つまり、SQLSTATE の参照によって ERRORMSG() が空の文字列に設定されたり、ERRORMSG() の参照によって SQLSTATE が '00000' に設定されたりすることがあります。また、WHEN 句の中に BEGIN ブロックがあると、両方ともデフォルト値に設定されます。

この問題を解決するには、外側の BEGIN ブロックに 2 つのローカル変数と、WHEN 句が実行されたらすぐに SQLSTATE 値と ERRORMSG() 値の両方を取得する 1 つの文が必要です。これには INTO 句を持つ簡単な SELECT 文が使えます。次は、このしくみを示す BEGIN ブロックです。

```
BEGIN
DECLARE @date      DATE;
DECLARE @sqlstate  VARCHAR ( 5 );
DECLARE @errormsg  VARCHAR ( 32767 );
SET @date = 'xxx';
EXCEPTION
   WHEN OTHERS THEN
      SELECT SQLSTATE, ERRORMSG() INTO @sqlstate, @errormsg;
      MESSAGE STRING ( 'SQLSTATE = ', @sqlstate ) TO CLIENT;
      MESSAGE STRING ( 'ERRORMSG() = ', @errormsg ) TO CLIENT;
END;
```

この BEGIN ブロックが実行されると、例外に関する次の情報が表示されます。

```
SQLSTATE = 53018
ERRORMSG() = 値 xxx をデータ型 timestamp に変換できません。
```

576 個の 0 ではないすべての SQLSTATE 値が例外ハンドラで検出されるわけではありません。表 8-1 は、DECLARE で明示的に名前が付けられ、定義されていても、警告として扱われ、WHEN 句では処理できない 32 個の値を示します。これらの警告をトラップする場合は、警告を発生させる可能性がある実行可能な文の直後に、SQLSTATE 値を明示的に検査するコードを追加する必要があります。この種類の検査の例については、「6.2.4　OPEN カーソルと CLOSE カーソル」を参照してください。

表8-1　SQLSTATE別の警告条件

SQLSTATE	SQLCODE	警告メッセージ
01000	200	警告
01003	109	集合関数では、NULL値は無視されます
01004	101	値がトランケートされました
01S02	121	カーソル・オプションの値が変更されました
01W01	124	不正なデータ変換：行%2のカラム'%1'にNULLが挿入されました
01W02	102	テンポラリ・テーブルが使用されました
01W03	103	不正なデータ変換
01W04	104	最後に読み込まれた後で、ローは更新されています
01W05	105	プロシージャが完了しました
01W06	106	テーブル'%2'のカラム'%1'の値が変更されています
01W07	107	構文に言語の拡張機能がみつかりました
01W08	111	文を実行することができませんでした
01W09	112	もっと情報が必要です
01W10	110	トランザクション・ログ・バックアップ・ページの一部だけが、満杯になっています
01W11	113	ユーザ'%2'に対するデータベース・オプション'%1'は不正な設定です
01W12	114	'%1'への文字セット変換は実行できません。かわりに'%2'が使用されます
01W13	115	データベース・サーバでは文字セット'%1'を変換できません。変換が正しくない可能性があります
01W14	116	言語'%1'はサポートされていないため、'%2'を使用します
01W15	117	文字セット'%1'、言語'%2'はサポートされていません。代わりに言語'%3'が使用されます
01W16	118	無効なユーザ選択性の評価が指定されました
01W17	120	'%1'は認識できないオプションです
01W18	122	非確定的な結果が返されました
01W19	123	データベースがスキーマなしで作成されました
01W20	129	プランに仮想インデックスが含まれている可能性があります
01W21	130	ローを新しいスキーマ・フォーマットに変換できませんでした
01W23	132	文字列カラム'%1'のヒストグラムを出力できません
01W24	137	参照整合性を保つためにテーブル'%1'からローが削除されました
01WC1	125	無効なマルチバイトの入力文字があります
01WC2	126	無効なマルチバイトの出力文字があります
01WC3	127	入力文字をデータベース側文字セットに変換できなかったため、単純な置換を実行しました
01WC4	128	出力文字をアプリケーション側文字セットに変換できなかったため、単純な置換を実行しました
02000	100	ローが見つかりません
WB011	119	バックアップ・ログ'%1'が開けません

Section

8.4 代入文

代入文を使って、ローカル変数、接続レベルの変数、行レベルのトリガ構造体内のフィールドの3タイプのターゲットにデータ値をコピーできます。

```
<assignment_statement> ::= SET <assignment_target> "=" <expression>
<assignment_target>    ::= <local_variable_name>
                         | <connection_variable_name>
                         | <row_structure_name> "." <column_name>
<expression>           ::=「第3章　選択」の<expression>を参照
```

次は、ローカル変数をターゲットとして使う代入文の例です。

```
BEGIN
DECLARE @a INTEGER;
DECLARE @b TIMESTAMP;
DECLARE @c BIGINT;
DECLARE @d VARCHAR ( 7 );
DECLARE @e LONG BINARY;
SET @a = 1;
SET @b = CURRENT TIMESTAMP;
SET @c = ( SELECT COUNT(*) FROM customer )
       + ( SELECT COUNT(*) FROM customer );
SET @d = IF DOW ( CURRENT DATE ) IN ( 1, 7 )
            THEN 'weekend'
            ELSE 'weekday'
         ENDIF;
SET @e = xp_read_file ( 'hello.txt' );
END;
```

接続レベルの変数の詳細については、「8.14　CREATE VARIABLE」を参照してください。トリガ内の構造体については、「8.11　CREATE TRIGGER」を参照してください。式については、「3.10　式と演算子」を参照してください。

Section

8.5 IF文

IF 文を使って、BEGIN ブロックの中のどの文を実行するかを制御できます。各 IF 文は、ブール式、THEN 句、0 個以上の ELSEIF 句、オプションの ELSE 句、IF を終了するキーワード END IF で構成されます。

```
<if_statement>       ::= IF <boolean_expression> THEN
                             [ <statement_list> ]
                         { ELSEIF <boolean_expression> THEN
                             [ <statement_list> ] }
                         [ ELSE
                             [ <statement_list> ] ]
                        END IF
<boolean_expression> ::=「第3章　選択」の<boolean_expression>を参照
```

IF 文が実行されると、1 番目のブール式が評価されます。ブール式の結果が TRUE であると、THEN 句の文が実行され、その他の句は省略されます。結果が FALSE または UNKNOWN であると、TRUE の結果が検出されるまで、後続の各 ELSEIF 句のブール式が評価されます。TRUE の結果が検出されると、その ELSEIF 句の文が実行され、その他の句は省略されます。すべてのブール式の結果が FALSE または UNKNOWN の場合は、ELSE 句の式が実行されます（ELSE 句がある場合）。

次は、「@x IS NULL」を評価した結果が TRUE なので「ELSEIF」が表示される例です。

```
BEGIN
   DECLARE @x INTEGER;
   SET @x = NULL;
   IF @x = 1 THEN
      MESSAGE 'THEN' TO CLIENT;
   ELSEIF @x IS NULL THEN
      MESSAGE 'ELSEIF' TO CLIENT;
   ELSE
      MESSAGE 'ELSE' TO CLIENT;
   END IF;
END;
```

この例で、@x に 1 が入っている場合は「THEN」が表示され、@x に 2 が入っている場合は「ELSE」が表示されます。ブール式と TRUE 値、FALSE 値、UNKNOWN 値については、「3.12　ブール式と WHERE 句」を参照してください。

IF 文はネストできます。たとえば、THEN 句、ELSEIF 句、ELSE 句の文を、対応する END IF を持つ別の IF 文にできます。また、THEN 句、ELSEIF 句、ELSE 句の文の前後に BEGIN キーワードと

END キーワードを使う必要はありません。次は、ネストされた IF 文の例です。

```
BEGIN
   DECLARE @message VARCHAR ( 100 );
   IF DATEFORMAT ( CURRENT DATE, 'MMDD' ) = '1225' THEN
      SET @message = 'It''s Christmas Day';
      IF DOW ( CURRENT DATE ) IN ( 1, 7 ) THEN
         SET @message = @message + ' on a weekend.';
      ELSE
         SET @message = @message + ' on a weekday.';
      END IF;
   ELSEIF DOW ( CURRENT DATE ) IN ( 1, 7 ) THEN
      SET @message = 'It''s just another weekend.';
   ELSE
      SET @message = 'It''s just another weekday.';
   END IF;
   MESSAGE @message TO CLIENT;
END;
```

Section

8.6 CASE文

CASE 文には 2 つの形式があり、複数の ELSEIF 句を持つ IF の代替として利用できます。

```
<case_statement> ::= <basic_case_statement>
                   | <searched_case_statement>
```

どちらの形式も、CASE キーワード、1 つ以上の WHEN 句、オプションの ELSE 句、CASE 文を終了するキーワード END CASE で構成されます。

最初の形式の CASE 文では、CASE キーワードの後ろに式が入り、各 WHEN 句で定数値を指定し、その後ろにキーワード THEN と 1 つ以上の文が続きます。CASE 文の評価は 1 回で、結果は、後続の各 WHEN 句の定数値と比較されます。一致する値があるとすぐにその WHEN 句の文が実行され、その他の句は省略されます。

```
<basic_case_statement> ::= CASE <expression>
                             WHEN <when_constant> THEN
                                [ <statement_list> ]
                           { WHEN <when_constant> THEN
                                [ <statement_list> ] }
                           [ ELSE
                                [ <statement_list> ] ]
```

```
                          END CASE
<when_constant>        ::= <string_literal>
                         | [ "-" ] <number_literal>
                         | <special_literal>
                         | NULL
<string_literal>       ::= 一重引用符で囲まれた文字の並び
<number_literal>       ::= 整数、真数、または浮動小数点の数値リテラル
<special_literal>      ::=「第1章　作成」の<special_literal>を参照
```

各WHEN句内の定数は、リテラルの形式またはNULLキーワードの形式を取ることができます。リテラルでは、比較はブール式「<expression> = <when_constant>」と解釈され、NULLキーワードでは、比較は「<expression> IS NULL」と解釈されます。

TRUEの結果が検出されるまで、後続の各WHEN句のブール式が評価されます。TRUEの結果が検出されると、対応するTHEN句から後ろの文が実行され、その他の句は省略されます。すべてのブール式の結果がFALSEまたはUNKNOWNの場合は、ELSE句の式が実行されます（ELSE句がある場合）。

次は、「@x IS NULL」の評価結果がTRUEなので「CASE #3」が表示される例です。

```
BEGIN
   DECLARE @x INTEGER;
   SET @x = NULL;
   CASE @x
      WHEN 1 THEN
         MESSAGE 'CASE #1' TO CLIENT;
      WHEN -2 THEN
         MESSAGE 'CASE #2' TO CLIENT;
      WHEN NULL THEN
         MESSAGE 'CASE #3' TO CLIENT;
      ELSE
         MESSAGE 'CASE ELSE' TO CLIENT;
   END CASE;
END;
```

この例で、@xに1に入っている場合は「CASE #1」が表示され、@xに-2が入っている場合は「CASE #2」が表示され、それ以外の非NULL値が入っている場合は「CASE ELSE」が表示されます。

もう1つの形式のCASE文は、CASEキーワードから後ろの式を省略し、代わりに各WHEN句で完全なブール式を使います。TRUEの結果が検出されるまで、各ブール式が評価されます。TRUEの結果が検出されると、対応するTHEN句から後ろの文が実行され、その他の句は省略されます。すべてのブール式の結果がFALSEまたはUNKNOWNの場合は、ELSE句の式が実行されます（ELSE句がある場合）。

```
<searched_case_statement> ::= CASE
                                 WHEN <boolean_expression> THEN
```

```
                                    [ <statement_list> ]
                                { WHEN <boolean_expression> THEN
                                    [ <statement_list> ] }
                                [ ELSE
                                    [ <statement_list> ] ]
                               END CASE
```

次は、前述の例を、2番目の形式のCASE文を使うように変更したものです。

```
BEGIN
   DECLARE @x INTEGER;
   SET @x = NULL;
   CASE
      WHEN @x = 1 THEN
         MESSAGE 'CASE #1' TO CLIENT;
      WHEN @x = -2 THEN
         MESSAGE 'CASE #2' TO CLIENT;
      WHEN @x IS NULL THEN
         MESSAGE 'CASE #3' TO CLIENT;
      ELSE
         MESSAGE 'CASE ELSE' TO CLIENT;
   END CASE;
END;
```

単一の <expression> と一連の定数に制限されないので、2番目の形式のCASE文のほうが柔軟です。8.5項で説明したELSEIF句と同じように、各 <boolean_expression> にはまったく異なる値と演算子を含められます。

Section

8.7 WHILEループ

WHILEループは、ブール式とそれに続くLOOPキーワード、1つ以上の実行可能な文、ループを終了するキーワードEND LOOPで構成されます。WHILEループには、ラベル付きとラベルなしの2つの形式があります。ラベルはLEAVE文のターゲットとして使用できます。

```
<while_loop>  ::= WHILE <boolean_expression> LOOP
                     [ <statement_list> ]
                  END LOOP
                | <while_label> ":"
                  WHILE <boolean_expression> LOOP
                     [ <statement_list> ]
```

```
                END LOOP [ <while_label> ]
<while_label> ::= <leave_statement>で使われる可能性のある<identifier>
```

実行時にWHILEループが検出されると、ブール式が評価されます。ブール式の結果がFALSEまたはUNKNOWNであると、キーワードLOOPとキーワードEND LOOPの間の文が省略され、制御はEND LOOPの後ろの文に渡されます。ブール式の結果がTRUEであると、キーワードLOOPとキーワードEND LOOPの間の文が実行され、制御はWHILEの先頭に戻ります。ブール式の2度目の評価でも結果がTRUEの場合は、式が再度実行され、制御は再び文の先頭に戻ります。このプロセスは、ブール式の結果がFALSEまたはUNKNOWNになるまで繰り返されます。FALSEまたはUNKNOWNになると、制御はEND LOOPの後ろの文に渡されます。

次は、10回実行されるWHILEループの例です。

```
BEGIN
   DECLARE @loop_counter INTEGER;
   SET @loop_counter = 1;
   WHILE @loop_counter <= 10 LOOP
      MESSAGE STRING ( 'Loop pass # ', @loop_counter ) TO CLIENT;
      SET @loop_counter = @loop_counter + 1;
   END LOOP;
END;
```

次は、他の接続によってテーブルgo_aheadにローが挿入されるまで実行されるWHILEループです。WAITFOR文は、ループが速く回転してCPU時間をすべて消費してしまわないように3秒間一時停止します。実際には、このループは、他の接続からローがgo_aheadに提供されて他の作業に進むまで待機します。

```
BEGIN
   WHILE NOT EXISTS ( SELECT * FROM go_ahead ) LOOP
      WAITFOR DELAY '00:00:03';
   END LOOP;
END;
```

Section

8.8 EXECUTE IMMEDIATE

EXECUTE IMMEDIATE文をBEGINブロックの中で使うことによって、実行時に自動的に作成されるSQL文を実行できます。

```
<execute_immediate> ::= EXECUTE IMMEDIATE { <execute_option> } <basic_expression>

<execute_option>    ::= WITH QUOTES ON      -- 二重引用符で識別子を区切る
                      | WITH QUOTES         -- WITH QUOTES ONと同じ
                      | WITH QUOTES OFF     -- デフォルト; QUOTED_IDENTIFIERが適用される
                      | WITH ESCAPES ON     -- デフォルト; エスケープ・シーケンスを変換する
                      | WITH ESCAPES OFF    -- エスケープ・シーケンスを無視する
                      | WITH RESULT SET ON  -- 結果セットを許可する
                      | WITH RESULT SET OFF -- デフォルト; 結果セットを許可しない

<basic_expression>  ::=「第3章　選択」の<basic_expression>を参照
                        -- サブクエリではない式
```

EXECUTE IMMEDIATE 文が実行されると、式は1つ以上の SQL 文を含む文字列として評価され、その文字列の内容が実行されます。文字列式を評価する方法と実行する方法は、次のオプションで制御します。

- **WITH QUOTES ON** は、文字列式に埋め込まれた2つ1組の二重引用符を識別子デリミタとして解釈することを指定する
- **WITH QUOTES OFF** はデフォルトで、文字列式に埋め込まれた二重引用符の組を QUOTED_IDENTIFIER オプションに従って解釈することを指定する。QUOTED_IDENTIFIER オプションが 'OFF' に設定されている場合、二重引用符は、一重引用符と同様に、文字列リテラル・デリミタと解釈される。QUOTED_IDENTIFIER がデフォルトの 'ON' の場合、「t1」のような文字列は、埋め込みの文字列リテラルではなく識別子と解釈される。
  QUOTED_IDENTIFIER オプションが 'OFF' に設定されているが、二重引用符で囲まれた埋め込み文字列を EXECUTE IMMEDIATE で識別子として扱いたい場合は、WITH QUOTES ON を使う
- **WITH ESCAPES OFF** は、EXECUTE IMMEDIATE 文の実行時に文字列式内の円記号をエスケープ文字として扱わないことを指定する
- **WITH ESCAPES ON** はデフォルトで、EXECUTE IMMEDIATE 文の実行時に文字列式に埋め込まれた円記号をエスケープ文字として解釈することを指定する。たとえば、¥¥ は単一の円記号に変換され、¥n は改行文字に変換される
- **WITH RESULT SET ON** は、EXECUTE IMMEDIATE 文が1つ以上の結果セットを返せることを指定する
- **WITH RESULT SET OFF** はデフォルトで、EXECUTE IMMEDIATE が結果セットを返さないことを指定する

EXECUTE IMMEDIATE 文では、結果セットを返す SELECT を実行できます。次は、ISQL で実行すると employee テーブル内のすべてのローが表示される例です。

```
BEGIN
   DECLARE @table_name VARCHAR ( 128 );
   SET @table_name = 'employee';
   EXECUTE IMMEDIATE WITH RESULT SET ON STRING ( 'SELECT * FROM ', @table_name );
END;
```

同時に複数のSQL文を実行できます。次は、ISQLで実行すると2つの結果セットが表示される例です。

```
BEGIN
   EXECUTE IMMEDIATE WITH RESULT SET ON
      'SELECT * FROM employee; SELECT * FROM customer';
END;
```

BEGINブロックが含まれている限り、EXECUTE IMMEDIATEの中でもローカル変数を宣言できます。次は'Hello'が表示される例です。

```
BEGIN
   EXECUTE IMMEDIATE STRING ( '
      BEGIN
         DECLARE @x VARCHAR ( 10 );
         SET @x = ''Hello'';
         MESSAGE @x TO CLIENT;
      END;' );
END;
```

WITH ESCAPES OFFオプションでは、IMMEDIATE文が渡された文字列式を解析するときに文字列リテラルのエスケープ文字の解析を抑制できます。これは、文字列リテラル内で「¥」エスケープ文字を使う必要がないという意味ではなく、EXECUTE IMMEDIATE文を使えばエスケープ文字を二重にする必要がないという意味です。

デフォルトはWITH ESCAPES ONです。次は、問題のあるコードの例です。

```
BEGIN
   DECLARE @sql LONG VARCHAR;
   SET @sql = 'UNLOAD TABLE t1 TO ''C:¥¥temp¥¥new_t1.txt''';
   EXECUTE IMMEDIATE @sql;
END;
```

この例をISQLで実行すると、文字列リテラル'UNLOAD TABLE t1 TO ''C:¥¥temp¥¥new_t1.txt'''は、SQL Anywhereで解析されてから@sqlに代入されます。円記号はエスケープ文字として扱われ、2つ1組の円記号はそれぞれ単一の円記号と解釈されます。@sqlがEXECUTE IMMEDIATEに渡されると、再度解析されて、円記号は再びエスケープ文字として扱われます。そのため、¥nは改行文字と解釈されます。文の実行は失敗し、次の2行からなるエラー・メッセージが表示されます。

```
Cannot access file 'C:¥temp
ew_t1.txt' -- 不正な引数
```

WITH ESCAPES OFF 句は、エスケープ文字の2回目の解析を抑制します。次のコードは正常に実行されます。

```
BEGIN
   DECLARE @sql LONG VARCHAR;
   SET @sql = 'UNLOAD TABLE t1 TO ''C:¥¥temp¥¥new_t1.txt''';
   EXECUTE IMMEDIATE WITH ESCAPES OFF @sql;
END;
```

デフォルトの WITH ESCAPES ON では、次のように多くの円記号が必要です。

```
BEGIN
   DECLARE @sql LONG VARCHAR;
   SET @sql = 'UNLOAD TABLE t1 TO ''C:¥¥¥¥temp¥¥¥¥new_t1.txt''';
   EXECUTE IMMEDIATE @sql;
END;
```

エスケープ文字の解析をすべて抑制することは不可能です。SQL Anywhere は文字列リテラルを解析してから EXECUTE IMMEDIATE に渡すので、次の例は失敗し、前述と同じエラー・メッセージが表示されます。WITH ESCAPES OFF オプションが指定されていますが、¥n はすでに改行文字に変換されているからです。

```
BEGIN
   EXECUTE IMMEDIATE WITH ESCAPES OFF 'UNLOAD TABLE t1 TO ''C:¥temp¥new_t1.txt''';
END;
```

Note

コンテキストによっては、EXECUTE IMMEDIATE 文の実行が失敗して構文エラー「プロシージャ 'IMMEDIATE' が見つかりません」が発生することがあります。構文エラーが発生した場合は、EXECUTE IMMEDIATE を BEGIN ブロックの外に置いてください。

Section

8.9 CREATE PROCEDURE、CALL、RETURN

CREATE PROCEDURE 文では、CALL 文を介して後で実行するようにBEGIN ブロックをデータベースに格納しておくことができます。次は、CREATE PROCEDURE 文、CALL 文、RETURN 文の構文です。

```
<create_procedure>        ::= CREATE PROCEDURE [ <owner_name> "." ] <procedure_name>
                                "(" [ <parameter_list> ] ")"
                                [ RESULT "(" <result_set_column_list> ")" ]
                                <begin_block>
<owner_name>              ::= <identifier>
<procedure_name>          ::= <identifier>
<parameter_list> ::= <parameter_specification> { "," <parameter_specification> }
<parameter_specification> ::= [ <parameter_disposition> ]
                              <parameter_name>
                              <data_type>
                            [ <parameter_default> ]
<parameter_disposition>   ::= IN
                            | OUT
                            | INOUT  -- デフォルト
<parameter_name>          ::= <identifier>
<parameter_default>       ::= DEFAULT <expression>
<result_set_column_list>  ::= <result_set_column> { "," <result_set_column> }
<result_set_column>       ::= <identifier> <data_type>

<call_statement>     ::= [ <return_code> "=" ]
                           CALL [ <owner_name> "." ] <procedure_name>
                           [ "(" [ <call_argument_list> ] ")" ]
<return_code>        ::= INTEGER値を受け取るための<identifier>変数
<call_argument_list> ::= <call_argument> { "," <call_argument> }
<call_argument>      ::= <basic_expression>
                           | <parameter_name> "=" <basic_expression>

<return_statement>   ::= RETURN [ <expression> ]
```

Note

SQL Anywhere では、BEGIN ブロックにコードを指定するのではなく、別の言語で作成され外部のライブラリ（Windows の .dll ライブラリ、Netware の .nlm ライブラリ、Unix の共有ライブラリ）に格納されているストアド・プロシージャを参照するプロシージャを作成できます。また、BEGIN ブロックを別のデータベース内のストアド・プロシージャの参照と置き換えることも可能です。本書では、これらの方法の説明を割愛します。詳細については、SQL Anywhere のオンラインヘルプを参照してください。

<owner_name> と <procedure_name> の組み合わせは、データベース内のすべてのプロシージャとユーザ定義関数の中でユニークとします。関数の詳細については、「8.10　CREATE FUNCTION」を参照してください。

<parameter_list> は CREATE PROCEDURE 文ではオプションですが、カッコは必須です。プロシージャがパラメータを取らない場合は、() のみの空のリストを指定してください。ただし、CALL 文ではカッコを省略できます。

プロシージャは 1 つ以上の結果セットと、1 つの INTEGER データ型のリターン・コードを返します。次は、1 つの結果セットと 1 つのリターン・コードを返すプロシージャと、そのプロシージャが参照するテーブルと、CALL が入っている BEGIN ブロックの例です。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL,
   non_key_1     VARCHAR ( 100 ) NOT NULL DEFAULT '',
   PRIMARY KEY ( key_1 ) );

CREATE PROCEDURE p1()
BEGIN
   SELECT * FROM t1;
   RETURN 999;
END;

BEGIN
   DECLARE @return_code INTEGER;
   INSERT t1 VALUES ( 1, 'A' );
   @return_code = CALL p1;
   SELECT @return_code;
END;
```

このコードを ISQL で実行した場合の結果は次のとおりです。CALL 文の結果セットと SELECT @return_code の結果セットの 2 つが表示されます。

```
key_1  non_key_1
=====  =========
1      'A'

@return_code
============
999
```

RETURN 文は、プロシージャではオプションです。この文を使用する場合、戻り値式はオプションです。戻り値式を指定した場合は INTEGER データ型に変換されます。これは、プロシージャはこのデータ型しか返せないためです。プロシージャが明示的なリターン・コード値を指定する RETURN 文を実行しない場合、リターン・コードは 0 に設定されます。

CREATE PROCEDURE の <parameter_list> 内の各パラメータは、オプションの <parameter_disposition>、必須の <parameter_name> と <data_type>、オプションの DEFAULT 値の 4 つのコンポーネントから構成できます。

<parameter_disposition> には、次の 3 つのキーワードのいずれかを指定します。

- **IN** は、パラメータが読み込み専用であることを示す。プロシージャが呼び出されると、対応する引数値がプロシージャに渡されるが、プロシージャが戻ってもパラメータの変更は引数に反映されない
- **OUT** は、パラメータが戻り専用であることを示す。プロシージャが呼び出されると、対応する引数値は無視され、パラメータに指定されている DEFAULT 値も同様に無視される。つまり、プロシージャが呼び出されたときに最初からパラメータは NULL である。ただし、パラメータに割り当てられている値は、プロシージャが戻るときに対応する引数に戻される
- **INOUT** はデフォルトで、対応する引数値がプロシージャに渡され、プロシージャが戻るとパラメータ値の変更が引数に戻されることを示す

式が OUT パラメータまたは INOUT パラメータの引数として渡された場合は、プロシージャが戻るとパラメータ値の変更は破棄されます。OUT と INOUT の <parameter_disposition> は、対応する引数が単純な変数である場合のみ動作します。

各 <parameter_name> は、プロシージャの中のパラメータ名とローカル変数名に関してユニークとします。各パラメータの <data_type> には、「1.5　データ型」に記載されているデータ型のいずれかを指定します。

DEFAULT 句を IN パラメータまたは INOUT パラメータと一緒に使って、CALL 文で対応する引数が省略された場合にパラメータに割り当てるデフォルト値を指定できます。DEFAULT 句は OUT パラメータでは効果がありません。INOUT パラメータでも、デフォルトが有効となるのは出力値を受け取る引数がない場合だけなので、あまり意味がありません。

次は、<parameter_disposition> と DEFAULT 値の動作を示す例です。引数値は CALL の前後に表示され、パラメータはプロシージャの開始時と終了の直前に表示されます。

```
CREATE PROCEDURE p1 (
   IN    @p_in    VARCHAR ( 1 ) DEFAULT 'X',
   OUT   @p_out   VARCHAR ( 1 ) DEFAULT 'Y',
   INOUT @p_inout VARCHAR ( 1 ) DEFAULT 'Z' )
BEGIN
   SELECT 'entry' AS point, @p_in, @p_out, @p_inout;
   SET @p_in    = 'P';
   SET @p_out   = 'Q';
   SET @p_inout = 'R';
   SELECT 'exit' AS point, @p_in, @p_out, @p_inout;
END;

BEGIN
```

```
   DECLARE @in    VARCHAR ( 1 );
   DECLARE @out   VARCHAR ( 1 );
   DECLARE @inout VARCHAR ( 1 );
   SET @in    = 'A';
   SET @out   = 'B';
   SET @inout = 'C';
   SELECT 'before' AS point, @in, @out, @inout;
   CALL p1 ( @in, @out, @inout );
   SELECT 'after' AS point, @in, @out, @inout;
END;
```

4つのSELECT文の結果は次のとおりです。すべてのパラメータに引数が指定されているので、DEFAULT値はすべて無視されます。また、INパラメータの入力値はプロシージャに渡されますが、戻り値は無視されます。OUTパラメータの入力値は無視されますが、戻り値は返されます。INOUTパラメータは入力値を受け取り、出力値も対応する引数に返します。

```
point   @in  @out  @inout
======  ===  ====  ======
before  A    B     C
entry   A    NULL  C
exit    P    Q     R
after   A    Q     R
```

すべてのパラメータがDEFAULT値を持っているので、次のように、すべての引数をCALLから省略できます。

```
   CALL p1;
```

次は、引数が指定されないときの4つのSELECT文の結果を示します。DEFAULT値はINパラメータとINOUTパラメータには割り当てられますが、OUTパラメータでは無視されます。また、CALLにはパラメータの値を受け取る引数がないので、パラメータの値はいずれも呼び出し元に戻されません。

```
point   @in  @out  @inout
======  ===  ====  ======
before  A    B     C
entry   X    NULL  Z
exit    P    Q     R
after   A    B     C
```

引数とパラメータの対応は、通常は位置で決まります。1番目の引数は1番目のパラメータに対応し、2番目の引数は2番目のパラメータに対応し、以降の対応も同様です。また、CALLで「<parameter_name> = <basic_expression>」引数形式を使ってパラメータ名で対応を決める代替方法もあります。この形式を使うと、先行引数を省略したり、引数の順序を変更したりできます。次は、1番目の引数を省略して

残りの2つの引数の順序を変更する前述のプロシージャに対するCALLです。DEFAULT値はINパラメータに適用され、OUTとINOUTの出力値は正しい引数に返されます。

```
    CALL p1 ( @p_inout = @inout, @p_out = @out );
```

引数に2つの形式を組み合わせて使うこともできますが、形式を「<parameter_name> = <basic_expression>」に切り替えたら、引数リストの残りの引数にはその形式を使う必要があります。次は両方の形式を使った場合のCALLです。

```
    CALL p1 ( @in, @p_inout = @inout, @p_out = @out );
```

プロシージャは1つ以上の結果セットを呼び出し元に返せます。これは、INTO句がない1つ以上のSELECT文を使って行います。また、1つ以上の結果セットを呼び出し元プロシージャに返し、カスケードCALL文を介して呼び出し先プロシージャから後で返すこともできます。

プロシージャが1つの結果セット、またはカラムの数が同じ2つ以上の結果セットを返す場合は、CREATE PROCEDURE文でRESULT句を使って、返すカラム名とデータ型を明示的に指定できます。必要に応じてデータ変換が実行されます。

次は、2つのプロシージャを示す例です。p1は2つの結果セットを返し、p2はp1からの2つの結果セットと独自の3番目の結果セットを返します。RESULT句は、最後のリターンでINTEGERデータ型へのカラム'9'の変換を強制的に実行します。

```
CREATE PROCEDURE p1( )
BEGIN
   SELECT 1, 2, 3;
   SELECT 4, 5, 6;
END;

CREATE PROCEDURE p2()
RESULT ( a INTEGER, b INTEGER, c INTEGER )
BEGIN
   CALL p1();
   SELECT 7, 8, '9';
END;

CALL p2();
```

Note

ISQLでは複数の結果セットを表示できますが、本書では、クライアント・アプリケーションに2つ以上の結果セットを返す単一のプロシージャ・コールを処理するためのプログラミング方法の説明は割愛します。

プロシージャが返す結果セットが1つのみの場合は、RESULT句の有無に関係なく、クエリのFROM句からプロシージャを呼び出せます。詳細については、「3.7　SELECT FROMプロシージャ・コール」を参照してください。

Tip

よくわからない場合は、RESULT句を使ってプロシージャの結果セットの名前とデータ型を指定してください。RESULT句を省略すると、返されるカラムの名前とデータ型はSELECT文自体で決定され、予期しない結果セット（SELECTリスト内の式など）が返されることがあります。

デフォルトでは、プロシージャの中で発生した例外は呼び出し元に直ちに返されます。次は、例外ハンドラが入っているBEGINブロックから呼び出されるプロシージャと、データ変換エラーが発生するSET文の例です。

```
CREATE PROCEDURE p1()
BEGIN
   DECLARE @date      DATE;
   MESSAGE STRING ( 'Before error' ) TO CONSOLE;
   SET @date = 'xxx';
   MESSAGE STRING ( 'After error' ) TO CONSOLE;
END;

BEGIN
   DECLARE @sqlstate  VARCHAR ( 5 );
   DECLARE @errormsg  VARCHAR ( 32767 );
   MESSAGE STRING ( 'Before call' ) TO CONSOLE;
   CALL p1();
   MESSAGE STRING ( 'After call' ) TO CONSOLE;
EXCEPTION
   WHEN OTHERS THEN
      SELECT SQLSTATE, ERRORMSG() INTO @sqlstate, @errormsg;
      MESSAGE 'Error handled outside procedure' TO CONSOLE;
      MESSAGE STRING ( 'SQLSTATE = ', @sqlstate ) TO CONSOLE;
      MESSAGE STRING ( 'ERRORMSG() = ', @errormsg ) TO CONSOLE;
END;
```

出力は次のとおりです。例外は呼び出し元に直ちに返され、例外ハンドラのWHEN OTHERS句が処理されるので、プロシージャの最後のMESSAGE文は省略されます。

```
Before call
Before error
Error handled outside procedure
SQLSTATE = 53018
ERRORMSG() = 値xxxをデータ型timestampに変換できません。
```

プロシージャに例外ハンドラが入っている場合は、その例外ハンドラが制御を取得するので、呼び出し元に例外に関する情報は渡されません。次は、前述の例のコードで、例外ハンドラをプロシージャの中に移動したものです。

```
CREATE PROCEDURE p1()
BEGIN
   DECLARE @sqlstate  VARCHAR ( 5 );
   DECLARE @errormsg  VARCHAR ( 32767 );
   DECLARE @date      DATE;
   MESSAGE STRING ( 'Before error' ) TO CONSOLE;
   SET @date = 'xxx';
   MESSAGE STRING ( 'After error' ) TO CONSOLE;
EXCEPTION
   WHEN OTHERS THEN
      SELECT SQLSTATE, ERRORMSG() INTO @sqlstate, @errormsg;
      MESSAGE 'Error handled inside procedure' TO CONSOLE;
      MESSAGE STRING ( 'SQLSTATE = ', @sqlstate ) TO CONSOLE;
      MESSAGE STRING ( 'ERRORMSG() = ', @errormsg ) TO CONSOLE;
END;

BEGIN
   DECLARE @sqlstate  VARCHAR ( 5 );
   DECLARE @errormsg  VARCHAR ( 32767 );
   MESSAGE STRING ( 'Before call' ) TO CONSOLE;
   CALL p1();
   SELECT SQLSTATE, ERRORMSG() INTO @sqlstate, @errormsg;
   MESSAGE STRING ( 'After call' ) TO CONSOLE;
   MESSAGE STRING ( 'SQLSTATE = ', @sqlstate ) TO CONSOLE;
   MESSAGE STRING ( 'ERRORMSG() = ', @errormsg ) TO CONSOLE;
END;
```

出力は次のとおりです。プロシージャの中にある例外ハンドラは、例外が発生すると直ちに制御を取得し、プロシージャが呼び出し元に戻るとSQLSTATEの設定は'00000'に戻ります。

```
Before call
Before error
Error handled inside procedure
SQLSTATE = 53018
ERRORMSG() = 値 xxx をデータ型 timestamp に変換できません。
After call
SQLSTATE = 00000
ERRORMSG() =
```

プロシージャの中にある例外ハンドラで別の例外が発生すると、その2番目の例外は呼び出し元に戻されます。次は、前述のコードを、2つのエラーと2つの例外ハンドラを含むように変更したものです。

```
CREATE PROCEDURE p1()
BEGIN
   DECLARE @sqlstate  VARCHAR ( 5 );
   DECLARE @errormsg  VARCHAR ( 32767 );
   DECLARE @date      DATE;
   MESSAGE STRING ( 'Before error' ) TO CONSOLE;
   SET @date = 'xxx';
   MESSAGE STRING ( 'After error' ) TO CONSOLE;
EXCEPTION
   WHEN OTHERS THEN
      SELECT SQLSTATE, ERRORMSG() INTO @sqlstate, @errormsg;
      MESSAGE 'Error handled inside procedure' TO CONSOLE;
      MESSAGE STRING ( 'SQLSTATE = ', @sqlstate ) TO CONSOLE;
      MESSAGE STRING ( 'ERRORMSG() = ', @errormsg ) TO CONSOLE;
      SET @undefined = 0;
END;

BEGIN
   DECLARE @sqlstate  VARCHAR ( 5 );
   DECLARE @errormsg  VARCHAR ( 32767 );
   MESSAGE STRING ( 'Before call' ) TO CONSOLE;
   CALL p1();
   MESSAGE STRING ( 'After call' ) TO CONSOLE;
EXCEPTION
   WHEN OTHERS THEN
      SELECT SQLSTATE, ERRORMSG() INTO @sqlstate, @errormsg;
      MESSAGE 'Error handled outside procedure' TO CONSOLE;
      MESSAGE STRING ( 'SQLSTATE = ', @sqlstate ) TO CONSOLE;
      MESSAGE STRING ( 'ERRORMSG() = ', @errormsg ) TO CONSOLE;
END;
```

出力は次のとおりです。内部の例外ハンドラが制御を取得して最初の例外に関する情報が表示され、2番目の例外は呼び出し元に戻されて2番目の例外ハンドラによってその詳細が表示されます。

```
Before call
Before error
Error handled inside procedure
SQLSTATE = 53018
ERRORMSG() = 値 xxx をデータ型 timestamp に変換できません。
Error handled outside procedure
SQLSTATE = 42W14
ERRORMSG() = Variable '@undefined' not found
```

例外処理の詳細については、「9.5.1　SIGNAL と RESIGNAL」と「9.5.2　RAISERROR と CREATE MESSAGE」を参照してください。

Section

8.10 CREATE FUNCTION

CREATE FUNCTION 文を使って、ABS() や SUBSTRING() などの組み込みスカラ関数と同じように動作する、つまり 0 個以上の読み込み専用パラメータを受け入れ、単一の値を返し、式で参照できる、ユーザ定義関数を作成できます。

```
<create_function> ::= CREATE FUNCTION
                      [ <owner_name> "." ] <user_defined_function_name>
                      "(" [ <function_parameter_list> ] ")"
                      RETURNS <data_type>
                      <function_characteristic>
                      <begin_block> -- 値を返さなければならない
<user_defined_function_name> ::= <identifier>
<function_parameter_list>    ::= <function_parameter> { "," <function_parameter> }
<function_parameter>         ::= [ IN ]
                                   <parameter_name> <data_type>
                                 [ <parameter_default> ]
<function_characteristic>    ::= DETERMINISTIC         -- デフォルト
                                 | NOT DETERMINISTIC
<user_defined_function_call> ::= <user_defined_function_name>
                                   "(" [ <function_argument_list> ] ")"
<function_argument_list>     ::= <expression> { "," <expression> }
```

関数は値を返すと予期されるので、CREATE FUNCTION 文には、戻り値のデータ型を指定する RETURNS 句が必要です。関数とプロシージャの違いはこれだけではありません。次に、プロシージャと異なる関数の特性を示します。

- 有効な <parameter_disposition> は IN のみで、これがデフォルトである。OUT パラメータと INOUT パラメータは指定できない
- 関数は結果セットを返せず、RESULT 句も持てない
- デフォルトの戻り値は 0 ではなく NULL である
- 引数とパラメータの対応は厳密に位置で決まる
- 関数は DETERMINISTIC または NOT DETERMINISTIC として定義できる

Note

関数はプロシージャと同様に CALL 文で呼び出せます。その場合は、上記の違いはいずれも当てはまりません。ただし、本書では、CALL を使って関数を呼び出す方法の説明は割愛します。プロシージャが必要な場合は、CREATE PROCEDURE を使います。

次は、数値の階乗を計算する関数と、組み込みテーブル RowGenerator を使ってその関数を 10 回呼び出す SELECT 文の例です。この関数は自分自身を再帰的に呼び出すことに注意してください。

```
CREATE FUNCTION factorial ( IN @n UNSIGNED BIGINT )
RETURNS UNSIGNED BIGINT
BEGIN
   DECLARE @f UNSIGNED BIGINT;
   CASE
      WHEN @n <= 1 THEN SET @f = 1;
      WHEN @n > 20 THEN SET @f = 0; -- 大きすぎる
      ELSE SET @f = @n * factorial ( @n - 1 );
   END CASE;
   RETURN @f;
END;

SELECT row_num AS n,
       factorial ( n )
  FROM RowGenerator
 WHERE row_num <= 10
 ORDER BY row_num;
```

出力は次のとおりです。

```
n   factorial(n)
==  ============
1   1
2   2
3   6
4   24
5   120
6   720
7   5040
8   40320
9   362880
10  3628800
```

Note

システム・テーブル RowGenerator は、1 ～ 255 の番号が割り振られた 255 個のローが入っている単純な小さいテーブルです。これは、この例のように、何もない状態から複数のローの結果セットを生成する場合に非常に便利です。

DEFAULT 句を関数パラメータと一緒に使って、対応する引数が関数参照で省略された場合にそのパラメータに割り当てるデフォルトを指定できます。引数とパラメータの対応は厳密に関数参照で位置が決まります。1 番目の引数は 1 番目のパラメータに対応し、2 番目の引数は 2 番目のパラメータに対応し、以降の対応も同様です。関数の引数を指定するための代替方法「<parameter_name> =

<basic_expression>」はありません。つまり、引数値は右からしか省略できません。たとえば、関数にDEFAULT 値を持つ2つのパラメータがある場合は、2つの引数とも省略する、または1番目の引数を指定して2番目の引数を省略することはできますが、1番目の引数を省略して2番目の引数を指定したり、引数の順序を変更したりすることはできません。

次は、2つのパラメータを予期しているのに1つの引数しか渡されない関数の例です。さらに、この関数には、RETURNS 句はありますが、実際の RETURN 文はありません。

```
CREATE FUNCTION f1 (
   @p1    VARCHAR ( 1 ) DEFAULT 'X',
   @p2    VARCHAR ( 1 ) DEFAULT 'Y' )
RETURNS VARCHAR ( 1 )
BEGIN
   MESSAGE STRING ( 'entry  ', @p1, ' ', @p2 ) TO CONSOLE;
   SET @p1 = 'P';
   SET @p2 = 'Q';
   MESSAGE STRING ( 'exit   ', @p1, ' ', @p2 ) TO CONSOLE;
END;

BEGIN
   DECLARE @a1     VARCHAR ( 1 );
   DECLARE @return VARCHAR ( 1 );
   SET @a1 = 'A';
   MESSAGE STRING ( 'before ', @a1 ) TO CONSOLE;
   SET @return = f1 ( @a1 );
   MESSAGE STRING ( 'after  ', @a1 ) TO CONSOLE;
   MESSAGE STRING ( 'return ', COALESCE ( @return, 'NULL' ) ) TO CONSOLE;
END;
```

出力は次のとおりです。1番目のパラメータは関数呼び出しから値を受け取り、2番目のパラメータはDEFAULT 値を取得します。また、戻り値は NULL です。

```
before A
entry  A Y
exit   P Q
after  A
return NULL
```

入力パラメータ値によって戻り値がユニークに決定される場合、その関数は決定的であるといいます。言い換えれば、同じ入力から常に同じ出力が生成されます。SQL Anywhere では、関数はデフォルトで決定的と見なされます。つまり、SQL Anywhere は、関数を呼び出すときのパラメータ値が同じ場合、実際にはその関数を再度実行する必要がないことを意味します。前回の呼び出しで計算された戻り値を使えるので、時間を節約し、パフォーマンスを向上できます。

入力パラメータが同じでも戻り値が異なる場合、つまり関数がテーブル更新などの副次的な作用を実

行する場合、その関数は非決定的であるといいます。この種の関数では、CREATE PROCEDURE 文でキーワード NOT DETERMINISTIC を明示的に指定し、実行時に関数参照が検出されるたびに実際にその関数を実行するように SQL Anywhere を強制する必要があります。このキーワードを指定しないと、間違った戻り値が使われたり、予期される副次的な作用が発生しなかったりする可能性があります。

次は、入力パラメータに関係なく、ランダムな値を返すことが予期される NOT DETERMINISTIC 関数の例です。

```
CREATE FUNCTION f1 ( @n INTEGER )
RETURNS INTEGER
NOT DETERMINISTIC
BEGIN
   RETURN CAST ( RAND() * 10000 AS INTEGER );
END;

SELECT row_num, f1 ( 1 )
  FROM RowGenerator
 ORDER BY row_num;
```

次は、この関数を呼び出す SELECT によって生成される最初の 5 つのローです。各ローの値が異なりますが、これが正しい出力です。

```
row_num  f1(1)
=======  =====
1        4775
2        4864
3        4285
4        8194
5        6299
```

NOT DETERMINISTIC 句が指定されていない場合の出力は次のとおりです。入力パラメータが変わらないので、SQL Anywhere は戻り値が同じであると見なします。

```
row_num  f1(1)
=======  =====
1        9842
2        9842
3        9842
4        9842
5        9842
```

DETERMINISTIC 関数に渡す古いパラメータ値と新しいパラメータ値を比較するため、SQL Anywhere は文字列値を、大文字と小文字を区別するものとして扱います。次は、NOT

DETERMINISTIC 句を省き、VARCHAR (1) パラメータを受け取り、'A' と 'a' によって 2 度呼び出されるように前述の関数を変更したものです。

```
CREATE FUNCTION f1 ( @s VARCHAR ( 1 ) )
RETURNS INTEGER
BEGIN
   RETURN CAST ( RAND() * 10000 AS INTEGER );
END;

SELECT f1 ( 'A' ), f1 ( 'a' );
```

デフォルトでは、SQL Anywhere の文字列比較は大文字と小文字の区別がないので、ブール式 'A' = 'a' の結果は TRUE です。ただし、この例では、'A' と 'a' は別の文字と見なされるので 2 つの関数の戻り値は異なります。

```
f1('A')  f1('a')
=======  =======
7648     4417
```

Tip

ユーザ定義関数を FROM 句、WHERE 句、HAVING 句から呼び出す場合は、DETERMINISTIC として定義されているかどうかに関係なく、実行回数を特定できません。これは、SQL Anywhere は別の形式に書き換えることによってクエリを自由に最適化できるためで、仮にそうなった場合、クエリの各部分が最終的に何回実行されるかを予測するのは非常に困難です。

Section

8.11 CREATE TRIGGER

「トリガ」は、データベースに格納される特殊な種類のBEGINブロックです。各トリガは単一のテーブルに関連付けられ、指定されたINSERT操作、UPDATE操作、DELETE操作のいずれかがそのテーブルで実行されると自動的に起動されます。トリガは基本的に、起動時期に従って3つのタイプに分かれます。ローごとの操作の実行前、ローごとの操作の実行後、INSERT文、UPDATE文、DELETE文の全体が完了した後に1回、の3つです。

```
<create_trigger> ::= <create_before_row_trigger>
                   | <create_after_row_trigger>
                   | <create_after_statement_trigger>
```

次は、1番目の形式「before row」トリガの完全な構文です。

```
<create_before_row_trigger> ::= CREATE TRIGGER <trigger_name>
                                BEFORE
                                <fired_by>
                                [ ORDER <order_number> ]
                                ON [ <owner_name> "." ] <table_name>
                                [ <referencing_as_structures> ]
                                FOR EACH ROW
                                [ WHEN "(" <boolean_expression> ")" ]
                                <begin_block>
<trigger_name>              ::= <identifier>
<fired_by>                  ::= <simple_fired_by> { "," <simple_fired_by> }
                              | UPDATE OF <column_name> { "," <column_name> }
<simple_fired_by>           ::= INSERT
                              | DELETE
                              | UPDATE
<column_name>               ::= <identifier>
<order_number>              ::= 1～32767の範囲の整数リテラル
<table_name>                ::= <identifier>
<referencing_as_structures> ::= REFERENCING { <as_structure> } <as_structure>
<as_structure>              ::= OLD AS <row_structure_name>
                              | NEW AS <row_structure_name>
<row_structure_name>        ::= シングルローの構造体の名前を示す<identifier>
```

CREATE TRIGGER文はかなり複雑です。次に、before rowトリガの作成方法を句ごとに説明します。

- **<trigger_name>**は対応するテーブルに対してユニークにする。各トリガも、起動時期、起動理由、起動順序、対応するテーブルでユニークにする。これらの特性はすべて以下の句で定義される

- **BEFORE** キーワードは、このトリガを起動する時期を指定する。このキーワードがあると、このトリガは「before row」トリガとして識別される
- **<fired_by>** 句は、このトリガを起動する理由を指定する。キーワード INSERT、UPDATE、DELETE のうちの 1 つ以上、または 1 つの UPDATE OF 句とそれに続くカンマで区切られたカラム名のリストで構成される。キーワード INSERT、UPDATE、DELETE は、このトリガをそれぞれ INSERT 操作、UPDATE 操作、DELETE 操作ごとに起動することを意味する。UPDATE OF 句は、UPDATE 操作が名前付きカラムのいずれかを指定している場合のみこのトリガを起動することを意味する
- **ORDER** 句は、各種トリガを起動する順番を指定する。この句は、同じ時期に同じ理由で同じテーブルに対し起動するトリガが 2 つ以上ある場合（たとえば、before row 形式の INSERT トリガが 2 つある場合）のみ必要となる
- **ON <table_name>** 句は単一のテーブルを識別する場合に必要。テーブルの <owner_name> はオプション。トリガの所有者は対応するテーブルと同じである。実際、各トリガは、外部キーやその他の制約と同様に、対応するテーブルの 1 つの属性と見なされるので、対応するテーブルから切り離せない
- **REFERENCING** 句では、WHEN 句と BEGIN ブロックのコードがローの古いバージョンと新しいバージョンを参照するときに使う 1 つまたは 2 つの構造体を指定できる。OLD AS 構造体には、トリガを起動した操作が実行される前にデータベースに存在するローのカラムごとに 1 つのフィールドが暗黙的に入り、NEW AS 構造体には操作の実行後に存在するローが入る。SQL Anywhere でテーブル以外の構造化データが実装される場所は、これらの古いロー用の構造体と新しいロー用の構造体しかない。デフォルトでは、これらの構造体はないので、REFERENCING 句を使って定義する。OLD AS 構造体は、DELETE 操作と UPDATE 操作によって起動されるトリガに対して定義でき、NEW AS 構造体は、INSERT 操作と UPDATE 操作によって定義されるトリガに対して定義できる
- **FOR EACH ROW** 句は、このトリガを起動する回数を指定する。この句がある場合も、このトリガは「before row」トリガとして識別される。たとえば、1000 個のローに適用される 1 つの SQL 文では、対応する before row トリガが 1000 回起動される
- オプションの **WHEN** 句では、ブール式を指定して、トリガの実際の起動条件を細かく制限できる。ブール式は、前述の REFERENCING 句で定義されている古い構造体と新しい構造体内のフィールド（カラム）を参照できる。BEGIN ブロックは、その他の稼働条件を満たし、ブール式が評価されると TRUE を返すローごとに実行される

Note

REFERENCING の OLD AS 構造体は、INSERT トリガであっても、DELETE 操作または UPDATE 操作によって引き起こされる INSERT トリガであれば定義可能です。同様に、NEW AS 構造体は、DELETE トリガであっても、INSERT 操作または UPDATE 操作によって引き起こされる DELETE トリガであれば定義可能です。ただし、これらの構造体は、定義されていても、トリガの起動時に必ずし

も存在するとは限りません。トリガがINSERT操作によって起動されるときに実際に作成されるのはNEW AS構造体のみなので、OLD AS構造体への参照は失敗します。同様に、トリガがDELETEによって起動されるときに作成されるのは、OLD AS構造体のみです。トリガがUPDATE操作によって起動されたときは、OLD AS構造体とNEW AS構造体の両方とも存在します。

次は、非常に簡単なbefore rowトリガと、このトリガを合計6回起動する4つのSQL文の例です。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL PRIMARY KEY,
   non_key_1     VARCHAR ( 100 ) NOT NULL );

CREATE TRIGGER triud_t1
BEFORE INSERT, DELETE, UPDATE
ON t1
FOR EACH ROW
BEGIN
   MESSAGE STRING ( 'Trigger triud_t1 fired.' ) TO CONSOLE;
END;

INSERT t1 VALUES ( 1, 'first row' );
INSERT t1 VALUES ( 2, 'second row' );
UPDATE t1 SET non_key_1 = 'xxx';
DELETE t1;
```

出力は次のとおりです。このトリガはFOR EACH ROWとして定義されたので、各INSERTによって1回ずつ、1つのUPDATE文によって2回、1つのDELETEによって2回の、合計6回起動されました。

```
Trigger triud_t1 fired.
Trigger triud_t1 fired.
Trigger triud_t1 fired.
Trigger triud_t1 fired.
Trigger triud_t1 fired.
Trigger triud_t1 fired.
```

次は、同じトリガを、トリガを起動したSQL操作の種類に応じて別のコードを実行するように修正した例です。

```
CREATE TRIGGER triud_t1
BEFORE INSERT, DELETE, UPDATE
ON t1
FOR EACH ROW
BEGIN
   CASE
      WHEN INSERTING THEN MESSAGE 'Inserting t1.' TO CONSOLE;
```

```
      WHEN UPDATING  THEN MESSAGE 'Updating t1.' TO CONSOLE;
      WHEN DELETING  THEN MESSAGE 'Deleting t1.' TO CONSOLE;
   END CASE;
END;

INSERT t1 VALUES ( 1, 'first row' );
INSERT t1 VALUES ( 2, 'second row' );
UPDATE t1 SET non_key_1 = 'xxx';
DELETE t1;
```

出力は次のとおりです。特殊なトリガ述部INSERTING、DELETING、UPDATINGの詳細については、「3.12.7　トリガ述部」を参照してください。

```
Inserting t1.
Inserting t1.
Updating t1.
Updating t1.
Deleting t1.
Deleting t1.
```

Tip

INSERT トリガと DETELE トリガで特殊なトリガ述部 INSERTING、DELETING、UPDATING を参照する場合は、IF 式と CASE 式ではなく、IF 文と CASE 文を使います。これは、INSERT によってトリガが起動されるときは REFERENCING の OLD AS 構造体が定義されず、DELETE によってトリガが起動されるときは NEW AS 構造体が定義されないためです。IF 式と CASE 式の中の THEN 式と ELSE 式は、評価されない場合でも常に解析され、未定義のロー構造体があるとエラーが発生します。IF 文と CASE 文では同じことが当てはまりません。THEN 分岐と ELSE 分岐は選択されなければ評価されず、解析もされません。このような状況で IF 文と CASE 文は動作するのに、IF 式と CASE 式が失敗するのは、そのためです。

UPDATE で指定される新しいカラム値が古いカラム値と同じ場合は、before row 形式の UPDATE トリガが起動されます。これは、UPDATE OF 句で指定されるカラムを参照するが、指定する値は変わらない、という UPDATE の場合も同じです。また、ロー構造体には、UPDATE OF リストから除外されているカラムも含め、古いローと新しいローのすべてのカラム値が入ります。その他のカラムはすべて WHEN 句で指定できます。次は、UPDATE OF 句と WHEN 句の両方が指定されている before row トリガと、すべての非キー・カラムの最終値を変更するコードの例です。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL PRIMARY KEY,
   non_key_1     VARCHAR ( 100 ) NOT NULL,
   non_key_2     VARCHAR ( 100 ) NOT NULL );
```

```
CREATE TRIGGER triud_t1
BEFORE UPDATE OF non_key_1
ON t1
REFERENCING OLD AS old_t1
            NEW AS new_t1
FOR EACH ROW
WHEN ( old_t1.non_key_2 = 'xxx' )
BEGIN
   MESSAGE 'Updating t1...' TO CONSOLE;
   MESSAGE STRING ( '  Old row:   ',
      old_t1.key_1, ', ',
      old_t1.non_key_1, ', ',
      old_t1.non_key_2 ) TO CONSOLE;
   MESSAGE STRING ( '  New row:   ',
      new_t1.key_1, ', ',
      new_t1.non_key_1, ', ',
      new_t1.non_key_2 ) TO CONSOLE;
   SET new_t1.non_key_1 = 'ccc';
   SET new_t1.non_key_2 = 'ddd';
   MESSAGE STRING ( '  Final row: ',
      new_t1.key_1, ', ',
      new_t1.non_key_1, ', ',
      new_t1.non_key_2 ) TO CONSOLE;
END;

INSERT t1 VALUES ( 1, 'ppp', 'aaa' );
INSERT t1 VALUES ( 2, 'qqq', 'bbb' );
UPDATE t1 SET non_key_2 = 'xxx' WHERE key_1 = 1;
UPDATE t1 SET non_key_1 = 'zzz' WHERE key_1 = 2;
UPDATE t1 SET non_key_1 = 'yyy';
SELECT * FROM t1 ORDER BY key_1;
```

この例の1番目のUPDATEでは、SET句がトリガのUPDATE OF句で指定されていないカラムを指定しているので、トリガは起動されません。2番目のUPDATEでは、t1.non_key_2の古い値が'bbb'で、トリガのWHEN句と一致しないので、トリガは起動されません。3番目のUPDATEは、t1の2つのローを変更しますが、UPDATE OF句とWHEN句の両方と一致する更新は1番目のローの更新のみなので、そのローの更新によってのみトリガは起動されます。次に、トリガの中のコードによって両方の非キー・カラム値が変更され、old、new、finalの3つすべてのバージョンのローが表示されます。出力は次のとおりです。

```
Updating t1...
  Old row:   1, ppp, xxx
  New row:   1, yyy, xxx
  Final row: 1, ccc, ddd
```

すべての更新が完了した後の最終的なSELECTの出力は次のとおりです。

```
key_1  non_key_1  non_key_2
=====  =========  =========
1      'ccc'      'ddd'
2      'yyy'      'bbb'
```

Tip

before row形式のCREATE TRIGGERが最もコーディングしやすいため、非常によく使われます。たとえば、before row形式のUPDATEトリガでは、無限再帰を気にしないで新しいローを修正できます。その他の2つのタイプのトリガでの更新は、ロー構造体ではなく対応するテーブルに直接行う必要があります。ネストされた更新によって同じトリガが再帰的に起動される可能性があるので、再帰が暴走していないことを確認する追加コードが必要です。

2番目の形式のトリガの構文では、1つだけ異なるキーワードがあります。キーワードAFTERは、このトリガがロー操作の完了後に起動されることを指定します。

```
<create_after_row_trigger> ::= CREATE TRIGGER <trigger_name>
                                AFTER
                                <fired_by>
                                [ ORDER <order_number> ]
                                ON [ <owner_name> "." ] <table_name>
                                [ <referencing_as_structures> ]
                                FOR EACH ROW
                                [ WHEN "(" <boolean_expression> ")" ]
                                <begin_block>
```

after rowトリガの動作はbefore rowトリガとほとんど同じですが、次の3つの点が異なります。

- after row形式のUPDATEトリガは、どのカラム値も変更されていないローに対しては起動されない
- after row形式のUPDATE OFトリガは、UPDATE OF句で指定されているどのカラムも変更されていないローに対しては起動されない
- REFERENCINGのNEW AS構造体内の値は、ロー操作はすでに実行済みなので修正できない

3番目の形式のトリガの構文では、キーワードAFTERとFOR EACH STATEMENTを使って、トリガを引き起こすINSERT文、UPDATE文、DELETE文が終了し、関係するすべてのローに適用された後で1回起動されるトリガを定義します。

```
<create_after_statement_trigger> ::= CREATE TRIGGER <trigger_name>
                                     AFTER
                                     <fired_by>
                                     [ ORDER <order_number> ]
                                     ON [ <owner_name> "." ] <table_name>
                                     [ <referencing_as_tables> ]
                                     [ FOR EACH STATEMENT ]
                                     <begin_block>
<referencing_as_tables> ::= REFERENCING { <as_table> } <as_table>
<as_table>              ::= OLD AS <as_table_name>
                          | NEW AS <as_table_name>
<as_table_name>         ::= 読み込み専用テンポラリ・テーブルの名前を示す <identifier>
```

次に、after statement トリガの特性のうち、after row トリガと異なる点を示します。

- REFERENCING の OLD AS 句と NEW AS 句は、シングルローの構造体ではなく、マルチローのテンポラリ・テーブルを定義する
- REFERENCING の OLD AS テンポラリ・テーブルには、トリガを起動させた文によって影響を受けたローが含まれ、データはその文の実行前のものである
- REFERENCING の NEW AS テンポラリ・テーブルには、トリガを起動させた文によって影響を受けたローが含まれ、データはその文の実行後かつこのトリガの開始前のものである
- REFERENCING の NEW AS テンポラリ・テーブル自体は読み込み専用だが、トリガの中にある UPDATE 文のジョインに使用できる
- after statement トリガでは WHEN 句を使えない
- REFERENCING の OLD AS テンポラリ・テーブルと NEW AS テンポラリ・テーブルは、トリガを引き起こす文がテーブルのどのローにも影響しない場合、空になりうる。他の条件が満たされていれば after statement トリガは必ず起動される。たとえば、UPDATE 文に、トリガの UPDATE OF 句で指定されているカラムの 1 つ以上を指定する SET 句がある場合は、UPDATE 文の WHERE 句がどのローとも一致しない場合でも、UPDATE OF トリガは起動される
- after statement 形式の UPDATE トリガまたは UPDATE OF トリガ内の REFERENCING の OLD AS テンポラリ・テーブルと NEW AS テンポラリ・テーブルには、カラム値が実際には変更されていないローは入らない。つまり、テンポラリ・テーブルが空になったり、または UPDATE 文の WHERE 句と一致したローの数よりテンポラリ・テーブル内のローの数が少なくなったりすることがある

after statement トリガが起動されるときの規則と、起動された場合の、REFERENCING の OLD AS テンポラリ・テーブルと NEW AS テンポラリ・テーブル内のローの数の規則は、かなり複雑です。次に、この規則を簡単にまとめた、before row トリガと after row トリガを含む 2 つのテーブルを示します。各テーブルの入り口で、「このトリガを起動するか」と「after statement トリガでは、REFERENCING のテンポラリ・テーブルにローをいくつ表示するか」の 2 つの質問に答えます。わかりやすくするため、

これらのテーブルでは、1つのローまたは0個のローのいずれかと一致するUPDATE文を使います。

最初のテーブルは、普通のUPDATEトリガ、つまり特殊なUPDATE OF句を使わないトリガ用です。このクラスのトリガが起動されるかどうかは、WHERE句がローと一致するかと、SET句が異なるカラム値を指定するかによって決まります。

```
UPDATE Trigger Fired?
   WHERE clause matches row:    yes         yes          no
   SET clause specifies value:  different   same         n/a
                                ==========  ===========  ===========
      BEFORE UPDATE ROW         yes         yes          no
      AFTER UPDATE ROW          yes         no           no
      AFTER UPDATE STATEMENT    yes, 1 row  yes, 0 rows  yes, 0 rows
```

2番目のテーブルは、UPDATE OF句を持つトリガ用です。このクラスのトリガが起動されるかどうかは、WHERE句がローと一致するか、SET句が指定するカラムがUPDATE OF句でも指定されているか、SET句が異なるカラム値を指定するかによって決まります。

```
UPDATE OF Trigger Fired?
   WHERE clause matches row:      yes         yes          yes   no           no
   SET clause matches UPDATE OF:  yes         yes          no    yes          no
   SET clause specifies value:    different   same         -     -            -
                                  ==========  ===========  ====  ===========  ====
      BEFORE UPDATE OF ROW        yes         yes          no    no           no
      AFTER UPDATE OF ROW         yes         no           no    no           no
      AFTER UPDATE OF STATEMENT   yes, 1 row  yes, 0 rows  no    yes, 0 rows  no
```

次は、2つのローと一致するUPDATE文によって起動されるafter statementトリガの例です。トリガのBEGINブロックにはカーソルのFORループとMESSAGE文があり、REFERENCINGのOLD ASテンポラリ・テーブルとNEW ASテンポラリ・テーブルの内容全体が表示されます。

このトリガには、テーブルを再び直接更新することによってトリガを引き起こすUPDATE文による変更を上書きするUPDATE文も入っています。これによってトリガは再帰的に起動されるので、再帰の暴走を防ぐために次の2つのステップを取ります。まず、トリガの中のUPDATE文には、直前のトリガの実行によってすでに変更されたローと一致しないWHERE句が含まれています。2番目に、トリガのBEGINブロックの最初の文が、REFERENCINGのOLD ASテンポラリ・テーブル内のローの数を検査するIF文になっています。そのテンポラリ・テーブルが空の場合（どのローとも一致しないUPDATEによってトリガが起動されると空になります）、LEAVE文がトリガを終了し、トリガは自分自身を再び起動するチャンスがなくなります。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL PRIMARY KEY,
   non_key_1     VARCHAR ( 100 ) NOT NULL,
```

```
   non_key_2     VARCHAR ( 100 ) NOT NULL );

CREATE TRIGGER tru_t1
AFTER UPDATE OF non_key_1
ON t1
REFERENCING OLD AS old_t1
            NEW AS new_t1
FOR EACH STATEMENT
this_trigger:
BEGIN
   MESSAGE 'Updating t1...' TO CONSOLE;
   IF NOT EXISTS ( SELECT * FROM old_t1 ) THEN
      MESSAGE '...no rows updated.' TO CONSOLE;
      LEAVE this_trigger;
   END IF;

   FOR f1 AS c1 NO SCROLL CURSOR FOR
   SELECT old_t1.key_1        AS @key_1,
          old_t1.non_key_1    AS @non_key_1,
          old_t1.non_key_2    AS @non_key_2
     FROM old_t1
    ORDER BY old_t1.key_1
   DO
      MESSAGE STRING ( '  Old row:   ',
         @key_1, ', ',
         @non_key_1, ', ',
         @non_key_2 ) TO CONSOLE;
   END FOR;

   FOR f2 AS c2 NO SCROLL CURSOR FOR
   SELECT new_t1.key_1        AS @key_1,
          new_t1.non_key_1    AS @non_key_1,
          new_t1.non_key_2    AS @non_key_2
     FROM new_t1
    ORDER BY new_t1.key_1
   DO
      MESSAGE STRING ( '  New row:   ',
         @key_1, ', ',
         @non_key_1, ', ',
         @non_key_2 ) TO CONSOLE;
   END FOR;

   UPDATE t1
          INNER JOIN new_t1
                  ON new_t1.key_1 = t1.key_1
      SET t1.non_key_1 = 'ccc',
          t1.non_key_2 = 'ddd'
    WHERE t1.non_key_1 <> 'ccc'
       OR t1.non_key_2 <> 'ddd';
```

```
   FOR f4 AS c4 NO SCROLL CURSOR FOR
   SELECT t1.key_1        AS @key_1,
          t1.non_key_1    AS @non_key_1,
          t1.non_key_2    AS @non_key_2
     FROM t1
          INNER JOIN new_t1
                  ON new_t1.key_1 = t1.key_1
    ORDER BY t1.key_1
   DO
      MESSAGE STRING ( 'Final row: ',
         @key_1, ', ',
         @non_key_1, ', ',
         @non_key_2 ) TO CONSOLE;
   END FOR;

END;

INSERT t1 VALUES ( 1, 'ppp', 'aaa' );
INSERT t1 VALUES ( 2, 'qqq', 'bbb' );
UPDATE t1 SET non_key_1 = 'yyy';
SELECT * FROM t1 ORDER BY key_1;
```

Note

トリガの暴走はしばらく続き、自分自身を何度も繰り返し起動しますが、最終的にはエラーが検出されてSQLSTATEが'42W29'に設定され、「プロシージャまたはトリガの呼び出しのネストが深すぎます」が出力されます。

MESSAGEの出力を見ると、トリガが合計3回起動されていることがわかります。外部のUPDATEによって1回、ローを2度目に変更する最初のトリガ実行内のUPDATEによって1回、どのローとも一致しない2番目のトリガ実行内のUPDATEによって1回の、合計3回です。

```
Updating t1...
  Old row:   1, ppp, aaa
  Old row:   2, qqq, bbb
  New row:   1, yyy, aaa
  New row:   2, yyy, bbb
Updating t1...
  Old row:   1, yyy, aaa
  Old row:   2, yyy, bbb
  New row:   1, ccc, ddd
  New row:   2, ccc, ddd
Updating t1...
...no rows updated.
Final row: 1, ccc, ddd
```

```
Final row: 2, ccc, ddd
Final row: 1, ccc, ddd
Final row: 2, ccc, ddd
```

SELECT の出力では、テーブルの最終的な内容が示されます。

```
key_1  non_key_1  non_key_2
=====  =========  =========
1      'ccc'      'ddd'
2      'ccc'      'ddd'
```

トリガは、複雑な整合性検査と、非正規化されたデータベース設計での計算にも使えます。次は、子テーブルでローが挿入、更新、削除されるたびに親テーブルの実行合計を更新するトリガです。INSERT ごとに、child.non_key_3 に挿入された値が対応する parent.non_key_3 に加算され、DELETE ごとに、削除された値が減算され、UPDATE ごとに、古い値が減算され新しい値が加算されます。

```
CREATE TRIGGER tr_child
BEFORE INSERT, DELETE, UPDATE
ORDER 1 ON child
REFERENCING OLD AS old_child
            NEW AS new_child
FOR EACH ROW
BEGIN
   CASE
      WHEN INSERTING THEN
         UPDATE parent
            SET parent.non_key_3
              = parent.non_key_3
              + new_child.non_key_3
          WHERE parent.key_1 = new_child.key_1;
      WHEN UPDATING THEN
         UPDATE parent
            SET parent.non_key_3
              = parent.non_key_3
              - old_child.non_key_3
              + new_child.non_key_3
          WHERE parent.key_1 = old_child.key_1;
      WHEN DELETING THEN
         UPDATE parent
            SET parent.non_key_3
              = parent.non_key_3
              - old_child.non_key_3
          WHERE parent.key_1 = old_child.key_1;
   END CASE;
END;
```

Tip

トリガの作成は避けてください。トリガの作成は、コーディングしにくい、わかりにくい、テストしにくい、デバッグしにくい、エラーやパフォーマンス問題が発生しやすい作業です。SQL Anywhere には、主キー制約と外部キー制約、UNIQUE 制約、CHECK 制約、計算カラムや、TIMESTAMP、LAST USER、AUTOINCREMENT、GLOBAL AUTOINCREMENT のような DEFAULT 値など、トリガの作成を避けるために使える機能がたくさんあります。これらの機能については、「第 1 章　作成」を参照してください。

Section

8.12 CREATE EVENT

「イベント」は、データベースに格納される特殊な種類の BEGIN ブロックです。各イベントが関連付けられる対象は、SQL Anywhere が検出可能な名前付きの出来事や条件、または SQL Anywhere が実行可能なスケジュールです。イベントは、SQL Anywhere によって自動的に実行できる点ではトリガに少し似ています。しかし、トリガと異なり、イベントはデータベースのテーブルには関連付けられず、自動的に起動できるだけでなく明示的にも実行できます。

イベントは大きく 3 つの種類に分かれます。名前付きの条件またはイベント・タイプに関連付けられるタイプ型イベント、クロックやカレンダ・スケジュールに従って実行されるスケジュール型イベント、TRIGGER EVENT 文（8.13 項参照）を介して明示的に実行されるユーザ定義イベントの 3 つです。

```
<create_event> ::= <create_typed_event>
                 | <create_scheduled_event>
                 | <create_user_defined_event>
```

タイプ型イベントは、14 種類ある条件またはイベント・タイプのいずれかに関連付けられます。これらのイベント・タイプの大半は SQL Anywhere が検出できる特定の出来事に関連付けられ、イベントが発生するとすぐに反応します。たとえば、「Connect」イベント・タイプは、ユーザが接続に成功したことを表します。これらのイベント・タイプのうち、DBDiskSpace、LogDiskSpace、ServerIdle、TempDiskSpace の 4 つには、SQL Anywhere が 30 秒おきに実行するアクティブ・ポーリングが必要です。

```
<create_typed_event> ::= CREATE EVENT <event_name>
                           TYPE <event_type>
                           [ <event_where_clause> ]
                           HANDLER <begin_block>
<event_name>         ::= <identifier>
```

```
<event_type>          ::= BackupEnd           -- バックアップが完了
                        | "Connect"           -- ユーザ接続が成功
                        | ConnectFailed       -- ユーザ接続が失敗
                        | DatabaseStart       -- データベースが起動
                        | DBDiskSpace         -- 30秒おきに検査
                        | "Disconnect"        -- ユーザ接続が切断
                        | GlobalAutoincrement -- 範囲の終わりに近い
                        | GrowDB              -- データベース・ファイルが拡張
                        | GrowLog             -- トランザクション・ログが拡張
                        | GrowTemp            -- テンポラリ・ファイルが拡張
                        | LogDiskSpace        -- 30秒おきに検査
                        | "RAISERROR"         -- RAISERRORが発行
                        | ServerIdle          -- 30秒おきに検査
                        | TempDiskSpace       -- 30秒おきに検査
```

イベントのWHERE句を使えば、タイプ型イベントが実際に実行される条件を制限できます。イベント・タイプごとに測定単位が関連付けられていて、実際の数値は、EVENT_CONDITION関数を呼び出して観測します。数値比較述部とAND演算子による簡単なブール式をWHERE句で使うことで、その観測値とリテラル値を比較できます。

```
<event_where_clause>        ::= WHERE <event_predicate>
                                { AND <event_predicate> }
<event_predicate>           ::= EVENT_CONDITION "(" <event_condition_name> ")"
                                  <event_comparison_operator>
                                  <event_condition_value>
<event_condition_name>      ::= 'DBFreePercent'   -- DBDiskSpace用
                              | 'DBFreeSpace'     -- DBDiskSpace用、MB単位
                              | 'DBSize'          -- GrowDB用、MB単位
                              | 'ErrorNumber'     -- "RAISERROR"用
                              | 'IdleTime'        -- ServerIdle用、秒単位
                              | 'Interval'        -- すべてに対応、秒単位
                              | 'LogFreePercent'  -- LogDiskSpace用
                              | 'LogFreeSpace'    -- LogDiskSpace用、MB単位
                              | 'LogSize'         -- GrowLog用、MB単位
                              | 'RemainingValues' -- GlobalAutoincrement用
                              | 'TempFreePercent' -- TempDiskSpace用
                              | 'TempFreeSpace'   -- TempDiskSpace用、MB単位
                              | 'TempSize'        -- GrowTemp用、MB単位
<event_comparison_operator> ::= "-"
                              | "<"
                              | ">"
                              | "!="
                              | "<="
                              | ">="
<event_condition_value>     ::= 比較のための整数リテラル値
```

Note

CREATE EVENT 文には他のキーワードもあります。詳細については、SQL Anywhere のオンラインヘルプを参照してください。DISABLE キーワードでは、ALTER EVENT 文が ENABLE を指定するまで、何があっても自動的に実行されないイベントを作成できます。デフォルトでは、イベントは有効（ENABLE）です。本書では、ALTER EVENT 文の説明を割愛します。また、AT CONSOLIDATED 句と AT REMOTE 句を使って、イベントを SQL Remote 環境のどこで実行するかを制御できます。本書では、Mobile Link のみを説明し、SQL Remote は扱わないので、これらの AT 句の説明も割愛します。

EVENT_CONDITION パラメータとして使用可能なものは、文字列リテラルの <event_condition_name> の値だけです。大文字と小文字の区別はありませんが、構文チェックは行われます。スペルミスがあったり、式を使おうとしたりすると、CREATE EVENT 文は失敗します。

EVENT_CONDITION の戻り値は数値です。'Interval' を除き、各イベント条件名が適用されるイベント・タイプは 1 つのみです。適用外のイベント・タイプで使われた条件名に対しては、EVENT_CONDITION は 0 を返します。

EVENT_CONDITION 関数は、前述の WHERE 句でしか呼び出せません。イベントの BEGIN ブロックの中で同じ情報が必要な場合は、EVENT_PARAMETER 関数を呼び出します。

EVENT_PARAMETER は、EVENT_CONDITION と同じすべての条件名と、次に示す一部の定義済みパラメータを受け入れます。

```
<event_parameter_function_call> ::= EVENT_PARAMETER
                                    "(" <event_parameter_name_string> ")"
<event_parameter_name_string>   ::= string expression containing an
                                    <event_parameter_name>
<event_parameter_name>          ::= DBFreePercent    -- EVENT_CONDITION から
                                  | DBFreeSpace
                                  | DBSize
                                  | ErrorNumber
                                  | IdleTime
                                  | Interval
                                  | LogFreePercent
                                  | LogFreeSpace
                                  | LogSize
                                  | RemainingValues
                                  | TempFreePercent
                                  | TempFreeSpace
                                  | TempSize
                                  | AppInfo          -- その他の定義済み名
                                  | ConnectionID
                                  | DisconnectReason
                                  | EventName
                                  | Executions
                                  | NumActive
                                  | ScheduleName
```

```
                                    | TableName
                                    | User
                                    | <user_defined_event_parameter_name>
<user_defined_event_parameter_name> ::= <identifier>
```

EVENT_PARAMETERの引数は、イベント・パラメータの名前の入った文字列です。たとえば、EVENT_PARAMETER ('User')は、このイベントを起動したユーザIDを返します。EVENT_CONDITIONの引数とは異なり、EVENT_PARAMETERには式を渡すこともできますが、式の結果が上記の定義済みパラメータ名のいずれか、またはユーザ定義パラメータ名である必要があります。

EVENT_PARAMETERの戻り値はVARCHAR (254)です。英数字値と数値はすべて文字列として返されます。デフォルト値は、定義済みの英数字パラメータの場合は空の文字列''、定義済みの数値パラメータの場合は'0'、TRIGGER EVENT文に値が指定されていないユーザ定義パラメータの場合はNULLです。ユーザ定義パラメータの詳細については、「8.13　TRIGGER EVENT」を参照してください。

次は、WHERE句を使ってサーバのアイドル状態が60秒間続いたらすぐに実行を開始するタイプ型イベント・ハンドラServerIdleの例です。

```
CREATE EVENT ev_ServerIdle
TYPE ServerIdle
WHERE EVENT_CONDITION ( 'IdleTime' ) >= 60
HANDLER BEGIN
   MESSAGE STRING (
      'The server has been idle for ',
      EVENT_PARAMETER ( 'IdleTime' ),
      ' seconds.' ) TO CONSOLE;
END;
```

このイベント・ハンドラの出力は次のとおりです。SQL Anywhereは30秒おきにこのタイプのイベントをポーリングしますが、WHERE句の条件により、アイドルになった最初の30秒のイベント・ハンドラは実行されませんでした。

```
The server has been idle for 60 seconds.
The server has been idle for 90 seconds.
The server has been idle for 120 seconds.
The server has been idle for 150 seconds.
The server has been idle for 180 seconds.
The server has been idle for 210 seconds.
The server has been idle for 240 seconds.
The server has been idle for 270 seconds.
```

CREATE EVENT文はDBA権限が付与されたユーザしか実行できません。イベントが実行される際、イベントを作成したユーザIDを使って別の新しい接続をデータベースに対して確立し、そのユーザの権限でイベントが実行されます。この別の接続は非同期で実行されます。言い換えると、イベント

のBEGINブロックの実行は、このイベントを直接的に実行させた接続も含め、他の接続でのコードの実行とコーディネートされません。

Tip

イベント・ハンドラの中でエラーが検出されていないかどうか、エンジン・コンソール・ウィンドウで確認してください。たとえば、「'ev_ServerIdle' のハンドラによって SQLSTATE '52003 が発生しました」は、「カラムが見つかりませんでした」という意味です。イベント実行ごとに別々の内部接続が使われるので、イベントのBEGINブロックでエラーが発生しても、そのエラー・メッセージを受け取る「クライアント・アプリケーション」がない場合、SQL Anywhereにはコンソール・ウィンドウ以外にエラーを送る場所がありません。ISQL文とTRIGGER EVENT文を使ってイベントをテストする場合でも、エラー・メッセージはコンソール・ウィンドウでしか確認できません。ISQLの［メッセージ］ウィンドウには表示されません。

次は、別個の接続とその非同期的な性質を示す例です。まず、次のCREATE EVENT文をユーザ「Admin1」で実行します。イベント自体に対する接続番号とユーザIDを表示するMESSAGE文があります。また、2つのEVENT_PARAMETER呼び出しにより、このイベントを実行させているもう1つの接続の接続番号とユーザも表示されます。

```
CREATE EVENT ev_Connect
TYPE "Connect"
HANDLER BEGIN
   MESSAGE STRING ( 'Connection event...' );
   MESSAGE STRING ( 'Event connection:      ', CONNECTION_PROPERTY ( 'Number' ) );
   MESSAGE STRING ( 'Event user:            ', CURRENT USER );
   MESSAGE STRING ( 'Triggering connection: ', EVENT_PARAMETER( 'ConnectionID' ) );
   MESSAGE STRING ( 'Triggering user:       ', EVENT_PARAMETER( 'User' ) );
   MESSAGE STRING ( CURRENT TIMESTAMP, ' ', CURRENT USER, ' Event waiting...' );
   WAITFOR DELAY '00:00:30';
   MESSAGE STRING ( CURRENT TIMESTAMP, ' ', CURRENT USER, ' ...event complete.' );
END;
```

この例の2番目のステップとして、ユーザ「User1」でデータベースに接続して直ちに次の文を実行します。

```
MESSAGE STRING ( CURRENT TIMESTAMP, ' ', CURRENT USER, ' Connected OK.' );
```

出力は次のとおりです。イベントの中にある最初の6つのMESSAGE文は、User1がデータベースに接続するとすぐに実行されます。そのとき、WAITFOR文によりイベントは30秒間一時停止します。ただし、接続イベントが実行中だからといってUser1の接続が遅延することはありません。逆に、User1は、接続イベントが最後のMESSAGE文を実行して終了するのを待たずに、「Connected OK」

MESSAGE 文をすぐに実行できます。

```
Connection event...
Event connection:      200824710
Event user:            ADMIN1
Triggering connection: 1778456925
Triggering user:       User1
2004-01-11 12:29:29.157 ADMIN1 Event waiting...
2004-01-11 12:29:31.661 User1 Connected OK.
2004-01-11 12:29:59.240 ADMIN1 ...event complete.
```

タイプ型イベントは再入可能で、並列で実行できます。この例では、2 番目の接続は、最初の実行が終了する前に同じイベントの 2 度目の起動を行えます。

Tip

イベントの中の CURRENT USER は、このイベントを実行させた接続のユーザ ID ではなく、イベントの作成者です。イベントの中で CONNECTION_PROPERTY を呼び出すときに他の接続のプロパティが必要な場合は、その接続番号を明示的に指定する必要があることに注意してください。

Tip

実行する順序を問わない限り、同じタイプの 2 つのタイプ型イベントを作成しないでください。タイプ型イベントを実行する順序を指定する記述がないだけでなく、イベントは非同期で実行されるので、最初に起動されたイベントの終了が最後にならないとも限りません。

スケジュール型イベントには TYPE 句も WHERE 句もありませんが、1 つ以上の SCHEDULE 項目があります。

```
<create_scheduled_event>   ::= CREATE EVENT <event_name>
                                   <event_schedule_list>
                                   HANDLER <begin_block>
<event_schedule_list>      ::= <event_schedule_item> { "," <event_schedule_item> }
<event_schedule_item>      ::= SCHEDULE [ <event_schedule_item_name> ]
                                   <event_start_times>
                                   [ <event_repeat_every> ]
                                   [ <event_on_days> ]
                                   [ START DATE <event_start_date> ]
<event_schedule_item_name> ::= <identifier> -- スケージュル時間が複数ある場合に必要
<event_start_times>        ::= START TIME <first_scheduled_time>
                             | BETWEEN <first_scheduled_time> AND <ending_time>
<first_scheduled_time>     ::= 開始時刻を示す文字列リテラル
<ending_time>              ::= イベントが発生しない場合の終了時刻を示す文字列リテラル
```

```
<event_repeat_every>       ::= EVERY <schedule_interval> HOURS
                             | EVERY <schedule_interval> MINUTES
                             | EVERY <schedule_interval> SECONDS
<schedule_interval>        ::= 時、分、または秒の整数リテラル数
<event_on_days>            ::= ON "(" <day_name>   { "," <day_name>   ")" }
                             | ON "(" <day_number> { "," <day_number> ")" }
<day_name>                 ::= 曜日を示す文字列リテラル
<day_number>               ::= その月における日を示す整数リテラル
<event_start_date>         ::= 開始日を示す文字列リテラル
```

イベントの各SCHEDULE項目は、次のコンポーネントから構成されます。

- **identifier**でスケジュール項目に名前を付けられる。この名前はEVENT_PARAMETER('ScheduleName')を介して実行時に入手できるので、イベント・ハンドラのコードはイベントがどのスケジュール項目によって起動されたかを判別できる。この名前はイベントに複数のSCHEDULE項目がある場合に必要となる
- **START TIME**句は、イベントを最初に起動する正確な時刻を指定する
- **BETWEEN**句は、イベントを最初に起動する時間（START TIMEと同様）と、イベントの起動を終了する時間の2つの時間を指定する
- **EVERY**句は、イベントを複数回起動することと、時間、分、秒単位の実行間隔を指定する
- **ON**句は、イベントを起動する曜日または日付を指定する
- **START DATE**句は、イベントを最初に起動する正確な日付を指定する

EVERY句とON句の両方が省略された場合、イベントの起動は1回です。EVERYが指定されONが省略された場合は、有効なすべての日を指定するデフォルトのON句が取られます。EVERYが省略されONが指定された場合は、イベントは指定された日ごとに1回起動されます。EVERYとONの両方が指定された場合は、イベントは指定された日の計算された時刻に起動されます。

次は、2つのSCHEDULE項目ですべての句を使っている例です。

```
CREATE EVENT ev_repeater
SCHEDULE sched_10
   START TIME '14:40:01'
   EVERY 10 SECONDS
   ON ( 'Monday', 'Sunday', 'Tuesday' )
   START DATE '2004-01-11',
SCHEDULE sched_17
   BETWEEN '14:40:02' AND '20:00'
   EVERY 17 SECONDS
   ON ( 'Wednesday', 'Sunday' )
   START DATE '2004-01-11'
HANDLER BEGIN
   MESSAGE STRING (
```

```
      'Event ',
      EVENT_PARAMETER ( 'EventName' ),
      ' fired at ',
      CURRENT TIMESTAMP,
      ' because of schedule ',
      EVENT_PARAMETER ( 'ScheduleName' ) ) TO CONSOLE;
END;
```

次は、SCHEDULE 項目「sched_10」によってイベントが START TIME の指定時刻 14:40:01 に最初に起動され、それ以降は EVERY 10 SECONDS 句に従って 14:40:11、14:40:21、14:40:31 などに起動されることを示す出力です。また、SCHEDULE 項目「sched_17」によってイベントが BETWEEN の最初の指定時刻 14:40:02 に起動され、それ以降は EVERY 17 SECONDS 句に従って 14:40:19、14:40:36、14:40:53 などに起動されることも示されています。

```
Event ev_repeater fired at 2004-01-11 14:40:01.048 because of schedule sched_10
Event ev_repeater fired at 2004-01-11 14:40:02.050 because of schedule sched_17
Event ev_repeater fired at 2004-01-11 14:40:11.083 because of schedule sched_10
Event ev_repeater fired at 2004-01-11 14:40:19.014 because of schedule sched_17
Event ev_repeater fired at 2004-01-11 14:40:21.017 because of schedule sched_10
Event ev_repeater fired at 2004-01-11 14:40:31.051 because of schedule sched_10
Event ev_repeater fired at 2004-01-11 14:40:36.079 because of schedule sched_17
Event ev_repeater fired at 2004-01-11 14:40:41.096 because of schedule sched_10
Event ev_repeater fired at 2004-01-11 14:40:51.030 because of schedule sched_10
Event ev_repeater fired at 2004-01-11 14:40:53.033 because of schedule sched_17
Event ev_repeater fired at 2004-01-11 14:41:01.055 because of schedule sched_10
Event ev_repeater fired at 2004-01-11 14:41:10.088 because of schedule sched_17
```

スケジュール型イベントは、スケジュールが明らかな重複を示している場合でも、繰り返し逐次実行されます。このため、実際の間隔と、EVERY 句で指定されている間隔が異なることがあります。たとえば、イベントが EVERY 10 SECONDS で指定されているが、イベントが起動するたびに実行の完了まで 15 秒かかる場合、間隔ポイントは 1 つおきに省略され、実際のスケジュールは EVERY 20 SECONDS が指定された場合と同じになります。

システム時間を常に監視しながらイベントの実行の開始が決められているのではなく、次にイベントを開始するまでの残り時間が計算されて管理されます。ワンタイム・イベントでは、この計算は、CREATE EVENT 文または ALTER EVENT 文が実行されたときに行われ、イベントの起動前にデータベースが停止して再起動した場合、再計算されます。繰り返しイベントが初めて起動されたときも同様です。繰り返しイベントの 2 回目以降の起動では、前回の実行が終了したときに計算が行われ、データベースが停止して再起動した場合、再計算されます。

Note

計算された経過時間が 1 時間を超えると、SQL Anywhere は 1 時間後に再計算を強制実行します。この再計算は、残りの経過時間が 1 時間未満になるまで、1 時間ごとに繰り返されます。これにより、夏時間の開始または終了のためにシステム・クロックが調整されたときも、イベントは予定どおり起動されます。

Tip

スケジュール型イベントが午前 0 時などの特定の時間に実際に発生していることをテストするためにシステム・クロックを調整する場合は、システム・クロックの調整後に、イベントの削除と作成、またはイベントの変更を行ってください。サーバをいったん停止してから起動してもかまいません。サーバが実行している間にシステム時間を調整し、スケジュール型イベントの経過時間を強制的に再計算することを指定しないと、そのイベントの次の起動時間が CURRENT TIMESTAMP と一致しない可能性があります。

タイプ型イベントとスケジュール型イベントを組み合わせることで管理タスクを自動化できます。次は、毎週日曜日の午前 0 時にデータベース・バックアップを実行してトランザクション・ログの名前を変更するスケジュール型イベントと、バックアップが完了したらすぐにテーブルを再編成するタイプ型イベントの例です。

```
CREATE EVENT ev_backup
SCHEDULE
   START TIME '00:00:00'
   ON ( 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Sunday' )
HANDLER BEGIN
   MESSAGE STRING (
      EVENT_PARAMETER ( 'EventName' ),
      ' started at ',
      CURRENT TIMESTAMP ) TO CONSOLE;
   BACKUP DATABASE DIRECTORY 'c:¥¥backup'
      TRANSACTION LOG RENAME MATCH
      WITH COMMENT 'ev_backup';
   MESSAGE STRING (
      EVENT_PARAMETER ( 'EventName' ),
      ' finished at ',
      CURRENT TIMESTAMP ) TO CONSOLE;
END;

CREATE EVENT ev_reorganize
TYPE BackupEnd
HANDLER BEGIN
   MESSAGE STRING (
      EVENT_PARAMETER ( 'EventName' ),
```

```
      ' started at ',
      CURRENT TIMESTAMP ) TO CONSOLE;
   REORGANIZE TABLE t1;
   REORGANIZE TABLE t1 PRIMARY KEY;
   MESSAGE STRING (
      EVENT_PARAMETER ( 'EventName' ),
      ' finished at ',
      CURRENT TIMESTAMP ) TO CONSOLE;
END;
```

出力は次のとおりです。午前0時にev_backupイベントが起動してBACKUP DATABASE文を実行し、それにより進行に従って多くのチェックポイント操作が強制実行されます。バックアップが完了すると、TYPE BackupEndで定義されているためすぐにev_reorganizeイベントが起動されます。このイベントは2つのREORGANIZE TABLE文を実行し、それによりチェックポイントも強制実行されます。

```
ev_backup started at 2004-01-12 00:00:00.003
Starting checkpoint of "test8" (test8.db) at Mon Jan 12 2004 00:00
Finished checkpoint of "test8" (test8.db) at Mon Jan 12 2004 00:00
Starting checkpoint of "test8" (test8.db) at Mon Jan 12 2004 00:00
Finished checkpoint of "test8" (test8.db) at Mon Jan 12 2004 00:00
Starting checkpoint of "test8" (test8.db) at Mon Jan 12 2004 00:00
Finished checkpoint of "test8" (test8.db) at Mon Jan 12 2004 00:00
ev_backup finished at 2004-01-12 00:00:01.044
ev_reorganize started at 2004-01-12 00:00:01.044
Starting checkpoint of "test8" (test8.db) at Mon Jan 12 2004 00:00
Finished checkpoint of "test8" (test8.db) at Mon Jan 12 2004 00:00
Starting checkpoint of "test8" (test8.db) at Mon Jan 12 2004 00:00
Finished checkpoint of "test8" (test8.db) at Mon Jan 12 2004 00:00
ev_reorganize finished at 2004-01-12 00:00:01.124
```

ev_reorganizeを起動するのはev_backupイベントではなく、ev_backupの中にあるBACKUP文です。ev_backupイベントのBACKUP文の後ろに時間のかかるコードが入っている場合、ev_reorganizeイベントはev_backupが終了する前に起動します。このイベントのカスケードは、2番目のトリガが1番目のトリガに入っているINSERT文、UPDATE文、DELETE文によって起動されるトリガのカスケードと似ています。

BACKUP DATABASE文の詳細については、「9.12　バックアップ」を参照してください。REORGANIZE TABLE文の詳細については、「10.6.3　テーブルの再編成」を参照してください。

ユーザ定義イベントの作成に、TYPE句、WHERE句、SCHEDULE句は使いません。

```
<create_user_defined_event> ::= CREATE EVENT <event_name>
                                HANDLER <begin_block>
```

ユーザ定義イベントを実行する唯一の方法は、TRIGGER EVENT 文を使用することです。ユーザ定義イベントは自動的に起動されません。ユーザ定義イベントは、TRIGGER EVENT 文が CALL 文に相当するという意味においてはプロシージャと似ていますが、プロシージャは CALL と同じ接続で同期で実行されるのに対し、イベントは独自の接続で非同期で実行されます。ユーザ定義イベントと TRIGGER EVENT 文については、次の項で詳しく説明します。

Section

8.13 TRIGGER EVENT

TRIGGER EVENT 文を使って、タイプ型イベントとスケジュール型イベントをテストしたり、通常の処理の一部としてオンデマンドでユーザ定義イベントを起動したりできます。

```
<trigger_event>              ::= TRIGGER EVENT <event_name>
                                   [ <event_parameter_list> ]
<event_parameter_list>       ::= "(" <event_parameter_assignment> { ","
                                   <event_parameter_assignment> } ")"
<event_parameter_assignment> ::= <event_parameter_name> "=" <event_parameter_value>
<event_parameter_value>      ::= 254 バイト以内の文字列
```

TRIGGER EVENT 文は、イベントの TYPE 句、WHERE 句、SCHEDULE 句の内容に関係なく、イベントを強制的に実行します。たとえば次の文は、午前 0 時以外でも、前項で説明した ev_backup イベントを起動します。

```
TRIGGER EVENT ev_backup;
```

TRIGGER EVENT 文では、値をイベントに渡せます。これらの値は、イベントの BEGIN ブロックの中にある EVENT_PARAMETER の呼び出しで取得できます。次のイベントを用いて、TRIGGER EVENT 文のいろいろな使い方を見てみましょう。この ev_DBDiskSpace イベントでは、DBFree Percent パラメータと DBFreeSpace パラメータが表示されます。

```
CREATE EVENT ev_DBDiskSpace
   TYPE DBDiskSpace
   WHERE EVENT_CONDITION ( 'DBFreePercent' ) < 20
HANDLER BEGIN
   MESSAGE STRING ( 'ev_DBDiskSpace started at ', CURRENT TIMESTAMP );
   MESSAGE STRING ( 'DBFreePercent: ', EVENT_PARAMETER ( 'DBFreePercent' ) );
   MESSAGE STRING ( 'DBFreeSpace  : ', EVENT_PARAMETER ( 'DBFreeSpace' ) );
END;
```

通常では、DBFreeSpaceの測定値が20%を下回ると、SQL Anywhereはこのイベントを30秒おきに実行します。出力は次のとおりです。

```
ev_DBDiskSpace started at 2004-01-12 13:39:56.495
DBFreePercent: 9
DBFreeSpace  : 2664
```

次のTRIGGER EVENTでは、DBFreeSpaceではなくDBFreePercentの値のみ（DBFreeSpaceはない）を渡します。

```
TRIGGER EVENT ev_DBDiskSpace ( DBFreePercent = '15' );
```

対応する出力は次のとおりです。TRIGGER EVENTを使うと、パラメータ値は自動的に取得されないので、DBFreeSpaceは、数値の定義済みパラメータのデフォルトである0になります。

```
ev_DBDiskSpace started at 2004-01-12 13:40:30.564
DBFreePercent: 15
DBFreeSpace  : 0
```

次は、両方の測定値を渡す例です。

```
TRIGGER EVENT ev_DBDiskSpace ( DBFreePercent = '15', DBFreeSpace = '111' );
```

出力は次のとおりです。TRIGGER EVENTを使うときは、イベント・ハンドラに重要なすべてのパラメータの値を渡してください。

```
ev_DBDiskSpace started at 2004-01-12 13:41:09.710
DBFreePercent: 15
DBFreeSpace  : 111
```

TRIGGER EVENT文で指定されたパラメータは、イベントのWHERE句でのEVENT_CONDITION呼び出しによって戻されるパラメータと同じですが、TRIGGER EVENTではWHERE句は無視され、WHERE句と一致しない値がTRIGGER EVENTに指定された場合でもイベントは実行されます。次は、WHERE句と一致しない値にパラメータを設定したTRIGGER EVENT文です。

```
TRIGGER EVENT ev_DBDiskSpace ( DBFreePercent = '50', DBFreeSpace = '111' );
```

対応する出力は次のとおりです。

```
ev_DBDiskSpace started at 2004-01-12 13:41:40.975
```

```
DBFreePercent: 50
DBFreeSpace  : 111
```

イベントのTRIGGER EVENT文では、あらゆるイベント条件とイベント・パラメータの名前を指定でき、文字列値の最大許容文字数は254です。SQL Anywhereは、TRIGGER EVENTから渡された値についてはエラー・チェックをまったく実行しません。たとえば、イベントが通常実行されるときにはパラメータ値が常に数値であっても、DBFreePercentに'xxx'を渡せます。

Tip

TRIGGER EVENTは、WHERE句の有無にかかわらず、タイプ型イベントや、SCHEDULE句のあるイベントのテストにはあまり適していません。これは、TRIGGER EVENT文が作り出すテスト環境は、現実を反映するとは限らないまったく人工的なものだからです。適切なテストを行うには、イベントを実行させる実際の条件を設定し、イベントが期待どおりに実際に実行されることを確認してください。

Note

TRIGGER EVENT文を実行しても、自動的にスケジュールされているイベントの次の実行時刻に影響はありません。

TRIGGER EVENTを使って、ユーザ定義イベントを実行したり、ユーザ定義パラメータをイベントのBEGINブロックに渡したりできます。この方法を使えば、1つのコード・ブロックを別々の接続で非同期に実行できます。次は、テスト・データを生成するためにバックグラウンドで実行するイベントの例です。挿入するローの数は、ユーザ定義パラメータ@row_countの値を返すEVENT_PARAMETERの呼び出しで提供されます。

```
CREATE EVENT ev_generate
HANDLER BEGIN
   DECLARE @row_count INTEGER;
   DECLARE @row_counter INTEGER;
   SET TEMPORARY OPTION BACKGROUND_PRIORITY = 'ON';
   MESSAGE STRING ( 'ev_generate started at ', CURRENT TIMESTAMP );
   SET @row_count = CAST ( EVENT_PARAMETER ( '@row_count' ) AS INTEGER );
   SET @row_counter = 0;
   WHILE @row_counter < @row_count LOOP
      SET @row_counter = @row_counter + 1;
      INSERT t1 VALUES (
         @row_counter,
         CAST ( RAND() * 1000000 AS INTEGER ) );
      IF MOD ( @row_counter, 10000 ) = 0 THEN
         COMMIT;
         MESSAGE STRING ( 'ev_generate COMMIT at ', CURRENT TIMESTAMP );
```

```
      END IF;
   END LOOP;
   COMMIT;
   MESSAGE STRING ( 'ev_generate ended at ', CURRENT TIMESTAMP );
END;
```

次は、テスト・データとして100,000のローを要求するTRIGGER EVENTと、この接続が制御を再取得したときに表示されるMESSAGE文の例です。

```
TRIGGER EVENT ev_generate ( @row_count = '100000' );
MESSAGE STRING ( 'Control regained after TRIGGER EVENT at ', CURRENT TIMESTAMP );
```

出力には、制御がTRIGGER EVENT文を実行した接続に直ちに戻されたことと、ev_generateイベントがバックグラウンドで引き続き実行されることが示されています。

```
ev_generate started at 2004-01-12 17:26:14.940
Control regained after TRIGGER EVENT at 2004-01-12 17:26:14.980
ev_generate COMMIT at 2004-01-12 17:26:16.112
ev_generate COMMIT at 2004-01-12 17:26:17.063
ev_generate COMMIT at 2004-01-12 17:26:18.034
ev_generate COMMIT at 2004-01-12 17:26:18.946
ev_generate COMMIT at 2004-01-12 17:26:19.817
ev_generate COMMIT at 2004-01-12 17:26:20.718
ev_generate COMMIT at 2004-01-12 17:26:21.670
ev_generate COMMIT at 2004-01-12 17:26:22.541
ev_generate COMMIT at 2004-01-12 17:26:24.414
ev_generate COMMIT at 2004-01-12 17:26:25.465
ev_generate ended at 2004-01-12 17:26:25.465
```

TRIGGER EVENT文で指定されるパラメータ名はローカル変数に似ていますが、周囲のコードにあるその他の名前とは関係ありません。次は、その事実を示す例です。このイベントはEVENT_PARAMETERを呼び出してユーザ定義パラメータ'@p'の値を取得し、その値をローカル変数@pに代入して、次の結果を表示します。

```
CREATE EVENT ev_test
HANDLER BEGIN
   DECLARE @p VARCHAR ( 128 );
   SET @p = COALESCE ( EVENT_PARAMETER ( '@p' ), 'NULL' );
   MESSAGE STRING ( '@p passed to event: ', @p );
END;
```

次は、TRIGGER EVENT (@p = @v)を実行して値をイベントに渡すコードです。このコードにはローカル変数@pもありますが、このコンテキストではローカル変数@pは、TRIGGER EVENTで指定さ

れる @p とは何の関係もありません。

```
BEGIN
   DECLARE @p VARCHAR ( 128 );
   DECLARE @v VARCHAR ( 254 );
   SET @p = 'hello';
   SET @v = 'world';
   MESSAGE STRING ( '@p before event:    ', @p );
   TRIGGER EVENT ev_test ( @p = @v );
   MESSAGE STRING ( '@p after event:     ', @p );
END;
```

出力は次のとおりです。外部の BEGIN ブロックのローカル変数 @p に、TRIGGER EVENT 文のパラメータ指定 @p = @v は適用されません。

```
@p before event:    hello
@p passed to event: world
@p after event:     hello
```

Section

8.14 CREATE VARIABLE

SQL Anywhere は、CREATE VARIABLE 文を使って、接続レベルの変数を作成できます。この種の変数は、いったん作成されると同じ接続で実行されているどの SQL コードからも参照できるため、「グローバル変数」といいます。これには、プロシージャ、トリガ、クライアント・アプリケーションから SQL Anywhere に渡される SQL 文などが含まれますが、イベントはグローバル変数ではありません。

```
<create_connection_variable> ::= CREATE VARIABLE
                                    <connection_variable_name> <data_type>
<connection_variable_name>   ::= <identifier>
<data_type>                  ::=「第 1 章　作成」の <data_type> を参照
```

接続レベルの変数はいったん作成されると、明示的に削除されるか接続が終了するまで、存在し続けます。ただし、接続レベルの変数は、作成元の接続が異なるとまったく別のものになるので、本当の意味で「グローバル」ではありません。同じ名前であっても、データ型と値が異なることがあります。

VAREXISTS 関数を使って、特定の接続レベルの変数があるかどうかを判別できます。VAREXISTS は、接続レベルの変数の名前が入っている 1 つの文字列パラメータを取り、変数が存在する場合は 1、存在しない場合は 0 を返します。次は、接続レベルの変数がすでに存在する場合にいったん削除してから作成するコードの例です。

```
IF VAREXISTS ( '@g_user_id' ) = 1 THEN
   DROP VARIABLE @g_user_id;
END IF;
CREATE VARIABLE @g_user_id VARCHAR ( 128 );
```

接続レベルの変数と同じ名前のローカル変数がBEGINブロックの中で宣言されていることがあり、その場合は実行期間の間、接続レベルの変数は「隠され」ます。次の例では、3つのSELECT文によって'xxx'、'yyy'、'xxx'が表示され、接続レベルの変数がBEGINブロックの中で表示されていないことがわかります。

```
CREATE VARIABLE @g_user_id VARCHAR ( 128 );
SET @g_user_id = 'xxx';
SELECT @g_user_id;
BEGIN
   DECLARE @g_user_id VARCHAR ( 128 );
   SET @g_user_id = 'yyy';
   SELECT @g_user_id;
END;
SELECT @g_user_id;
```

Section

8.15 この章のまとめ

この章では、IF、CASE、WHILEの制御構造体、ローカル宣言、例外処理ロジックなど、複数のSQL文を含むBEGINブロックの作成方法について説明しました。BEGINブロックから作成されるストアド・プロシージャ、関数、トリガ、イベントの4種類のストアドSQLモジュールについても説明しました。

また、SQLコマンドの動的な作成と実行用のEXECUTE IMMEDIATE文と、接続レベルの変数の定義に使うCREATE VARIABLE文についても説明しました。

次の章では、データベースの構築や障害からのシステムの保護などについて説明します。

Chapter

9

保護

Section

9.1 概要

この章は、システムの障害、不正なユーザからのアクセス、システム処理が固まってしまうような事態、およびそれらの予防策に関係します。

つまり、お使いのデータベースをさまざまな悪い事象から保護するあらゆる手段について説明します。

9.2 項では、SET OPTION 文について説明します。この文は保護のためだけに使われるものではありませんが、データベースと接続のオプションの多くが保護と安全性に関わる面を制御します。

9.3 項では、BEGIN TRANSACTION、COMMIT、ROLLBACK を使ったトランザクション制御について説明します。また、サーバ側の CHAINED オプションやクライアント側のオートコミット・モードによって、トランザクション処理がどのような影響を受けるかについても説明します。

9.4 項では、SAVEPOINT、RELEASE SAVEPOINT、ROLLBACK TO SAVEPOINT を使用することで、ネストされたサブトランザクションを SQL Anywhere で実装する方法を説明します。

9.5 項の「エラー処理」では、SQL コードによって、クライアント・アプリケーションに対する問題の明示的な通知を可能にするさまざまな方法を説明します。具体的には、SIGNAL、RAISERROR、ROLLBACK TRIGGER について説明します。

9.6 項と 9.7 項ではロック、ブロック、独立性レベルによって、別々の接続が同じデータを同時に処理することで発生する矛盾からデータベースを保護する方法を説明します。9.8 項では、2 種類のデッドロックについて説明します。互いにブロックし合う 2 つ以上の接続によって発生する環状のデッドロックと、ブロックされる接続が多すぎて SQL Anywhere が処理しきれないときに発生する「すべてのスレッドがブロックされる」デッドロックです。9.9 項では、ミューテックスすなわち相互排他操作について説明します。また、複数の CPU を使用する環境において、ミューテックスがパフォーマンスの損失をどのようにもたらすかについても説明します。

9.10 項では、GRANT CONNECT 文を使ってデータベースのユーザ ID を作成する方法を説明します。その次の 3 つのサブセクションでは、他の形式の GRANT 文を使用して、テーブルとビューの選択、更新、それにストアド・プロシージャと関数の実行のためのパーミッションを含め、さまざまな権限を個々のユーザ ID に与える方法を説明します。9.10.4 では、GRANT RESOURCE、GRANT DBA、GRANT REMOTE DBA を用いる権限について引き続き説明を行います。9.10.5 では、ユーザ・グループを使用して、管理と SQL プログラミングの両方を簡単にする方法を説明します。

9.11 項では、ロギングとリカバリが SQL Anywhere においてどのような働きをするかを、トランザクション・ログ、チェックポイント・ログ、ロールバック・ログを含めて説明します。9.12 項ではデータベースのバックアップ手順の設定方法を、9.13 項ではデータベースをバックアップからリストアする方法を、それぞれ説明します。9.14 項では、必要なときに機能することを確かめるためにデータベースを検証する方法を説明します。

Section

9.2 オプション

SQL Anywhere の動作の多くは、オプションという組み込みパラメータによって制御されます。この項では、オプションがどのように格納されるか、またその値をどのように変更できるかを、例を示しながら説明します。なお、この章の他の場所や他の章であっても、説明されている話題にとって重要と思われるオプションについては、その場で取り上げました。

オプションの基本的な種類には、グローバルとローカルの2つがあります。グローバル・オプションは、個々の接続ではなく、データベースまたはサーバの全体に適用されます。たとえば、AUDITING オプションを使用すると、データベース内の監査機能の有効と無効を切り替えることができ、その影響はすべての接続に対して同じです。一方、ローカル・オプションは個々の接続に適用されます。たとえば、BACKGROUND_PRIORITY オプションを使用して、実行している特定の接続の優先度を上げたり下げたりすることができます。

ほとんどのオプションは、事実上ローカルなものです。表 9-1 は、数少ないグローバル・オプションの一覧です【監注 7】。

表 9-1　グローバル・オプション

オプション名
ANSI_PERMISSIONS
AUDITING
AUDITING_OPTIONS
CHECKPOINT_TIME
DATABASE_AUTHENTICATION
GLOBAL_DATABASE_ID
JAVA_NAMESPACE_SIZE
JAVA_PAGE_BUFFER_SIZE
LOGIN_MODE
MAX_HASH_SIZE
MAX_WORK_TABLE_HASH_SIZE
MIN_PASSWORD_LENGTH
MIN_TABLE_SIZE_FOR_HISTOGRAM【監注 8】
OPTIMIZATION_WORKLOAD
PINNED_CURSOR_PERCENT_OF_CACHE
PRESERVE_SOURCE_FORMAT

【監注 7】この他に、LOG_DEADLOCKS、TEMP_SPACE_LIMIT_CHECK、INTEGRATED_SEVER_NAME、WEBSERVICE_NAMESPACE_HOST が該当する。
【監注 8】deprecated

オプション名
RECOVERY_TIME
RI_TRIGGER_TIME
TRUNCATE_DATE_VALUES【監注9】
TRUNCATE_TIMESTAMP_VALUES
TRUNCATE_WITH_AUTO_COMMIT

Note

SQL Anywhere オンラインヘルプでは、グローバル対ローカルではなく、別の分類スキームを使用しています。SQL Anywhere オンラインヘルプは全体的な目的に従って分類されています。つまり、Transact SQL 互換性オプション、ISQL オプション、ANSI 互換性オプションといった分類です。本書では、さまざまな SET OPTION 文がどのように働くかの説明を容易にするためにグローバル対ローカルという分類を使用します。

次の 4 階層において、同一オプションに対して別々の値を指定できます。

- 内部システム・デフォルト値は、グローバルとローカルのすべてのオプションに存在し、SQL Anywhere の操作に重要な意味を持つ。この値は変更できないが、階層の下位レベルに指定する値で上書きできる。対応するパブリックな値が削除された場合にのみ、この値が使用される。本書ではパブリック・デフォルト値が常に存在することを前提とするため、内部システム・デフォルト値についてはこれ以上説明しない
- パブリック・デフォルト値は、グローバルとローカルのオプションに存在し、SYSOPTION カタログ・テーブルに格納される。グローバル・オプションの場合は、この値が適用される。ローカル・オプションの場合は、階層の下位レベルに明示的な値が指定されていないとき、この値が使用される。すなわち、「パブリック」とは、個々のユーザや接続とは反対に、対象のすべてを意味する
- ユーザ・デフォルト値の設定は任意であり、この値はローカル・オプションにだけ存在し得る。ユーザ・デフォルト値は個別のユーザ ID に対応付けられ、この値もまた SYSOPTION テーブルに格納される。新規データベースの場合、初期状態では SYSOPTION にユーザ・デフォルト値は存在しない
- ローカル・オプションの現在の値は、接続の確立時に初期化され、一時的に変更されることがある。現在の値は SYSOPTION テーブルには格納されない

【監注 9】deprecated

Note

新しい接続が確立されるたびに、SQL Anywhere は sp_login_environment という組み込みプロシージャを呼び出します。通信プロトコルが TDS の場合、このプロシージャは、続いて sp_tsql_environment プロシージャを呼び出します。sp_tsql_environment プロシージャは、Transact SQL 互換性を管理するためにいくつかのオプションを明示的に設定します。TDS プロトコルは、Sybase jConnect を備えた JDBC または Sybase Open Connect ライブラリを用いる接続に対して使用されます。もしも TDS を使用しているが Transact SQL 互換性に関心がない場合、SQL Anywhere オンラインヘルプで "sp_tsql_environment" を検索して、このオプションが希望どおりの値に設定されていることを確認してください。ただし、ODBC、OLE DB、または Embedded SQL を使用してデータベースに接続する場合は、sp_tsql_environment プロシージャは呼び出されないので、その点を心配する必要はありません。

Note

データベースに接続した後、ISQL は自らの目的のためにいくつかのオプションを明示的に設定します。ISQL オプションについては SQL Anywhere オンラインヘルプで説明されているため、本書では詳しく説明しません。

パブリック、ユーザ、現在の値の各レベルのオプション値は、SET OPTION 文の 3 種類の形式を使って変更できます。

```
<set_option> ::= <set_public_default_option>
               | <set_user_default_local_option>
               | <set_temporary_local_option>
```

次の構文は、グローバルとローカルのオプションをパブリック・レベルで変更します。

```
<set_public_default_option> ::= SET [ EXISTING ] OPTION
                                  PUBLIC "." <option_name> "=" [ <option_value> ]
<option_name>  ::= <identifier> -- 通常は存在するオプションの名前
<option_value> ::= オプション値として格納される文字列リテラル
                 | 文字列値として格納される数値リテラル
                 | 文字列値としてそのまま格納される <identifier>
                 | ON   -- 'ON' として格納
                 | OFF  -- 'OFF' として格納
                 | NULL -- このレベルのエントリを削除
<identifier>   ::=「第 1 章　作成」の <identifier> を参照
```

Note

上で説明した <option_value> 構文は、3 つの形式の SET OPTION 文すべてで使用されます。ただし、パブリック・デフォルト・レベルで NULL 値が使用されることはほとんどありません。NULL 値の使用

は、より下位のユーザや現在の値のレベルに指定された値を削除する目的で、これらのレベルでのみ使用してください。また、ユーザ・レベルに値が存在する場合は、パブリック・デフォルト値を削除できません。

Note

<option_value> 構文は、3 つの形式の SET OPTION 文のすべてで完全に省略されることがあります。省略された場合、NULL 値が指定された場合と同じで、対応するレベルのエントリを削除するのと同じ影響を与えます。本書では、明示的な NULL 値の表示を行います。

ほとんどのパブリック・デフォルト・オプションの設定は、変更の必要がありません。デフォルト設定の大部分が注意深く決められていることは SQL Anywhere の大きな長所の 1 つであり、ユーザが設定に悩まされることはありません。

しかし、デフォルト値によっては変更の候補に挙がるものもあります。以下に、パブリックな設定を、永続的に別の値に変えるための SET 文の例をいくつか示します。

```
SET EXISTING OPTION PUBLIC.ANSI_INTEGER_OVERFLOW        = 'ON';
SET EXISTING OPTION PUBLIC.CLOSE_ON_ENDTRANS            = 'OFF';
SET EXISTING OPTION PUBLIC.FLOAT_AS_DOUBLE              = 'ON';
SET EXISTING OPTION PUBLIC.MIN_TABLE_SIZE_FOR_HISTOGRAM = '100';
SET EXISTING OPTION PUBLIC.STRING_RTRUNCATION           = 'ON';
```

以下に、これらの設定の意味を説明します。

- ANSI_INTEGER_OVERFLOW = 'ON' は、INSERT 文によって整数カラムに許容範囲外の値を格納しようとする場合、不正な値を格納しないでエラーを発生させることを意味する
- CLOSE_ON_ENDTRANS = 'OFF' は、COMMIT または ROLLBACK の操作の副次的効果としてカーソルが閉じられるのを防ぐことを意味する
- FLOAT_AS_DOUBLE = 'ON' は、明示的な精度の指定がない場合、CREATE TABLE 文に強制的に FLOAT データ型を単精度ではなく倍精度として解釈させることを意味する
- MIN_TABLE_SIZE_FOR_HISTOGRAM = '100' は、強制的に SQL Anywhere に小さなテーブルのパフォーマンス統計値を管理させることを意味する【監注 10】
- STRING_RTRUNCATION = 'ON' は、INSERT 文によってブランク以外の文字のトランケートを文字列値の右端から行おうとする場合、何も通知せずにトランケートを行って値をカラムに入れるのではなく、エラーを発生させることを意味する

EXISTING キーワードの使用は任意ですが、使用することを強くおすすめします。このキーワードに

【監注 10】このオプションは deprecated になっていて、ユーザは変更できない。ローが 5 以上あるテーブルは常に統計値が管理される。

よって、SQL Anywhereがオプション名のスペルミスを新しいオプション名と解釈するのを防げます。

Note

ユーザ定義オプションを作成できますが、本書では説明しません。ユーザ定義オプションとは、どの定義済みオプションとも異なるオプション名を持つオプションです。

SET文の2番目の形式は、ローカル・オプションの設定、変更、削除をユーザ・レベルで行うために使用されます。

```
<set_user_default_local_option>
   ::= SET OPTION <user_id> "." <option_name> "=" [ <option_value> ]

<user_id> ::= <identifier>
```

NULLでないオプション値が指定された場合、ユーザ名とオプション名に対応するSYSOPTIONテーブル内にまだローが存在しないか、すでに存在しても更新されるときはローが1つ挿入されます。NULLがオプション値として指定された場合、SYSOPTIONテーブル内に対応するローが存在すれば削除され、存在しなければ何も起こりません。

次に示すのは、時間に対する要求の厳しい処理を実行するユーザのために、BLOCKINGオプションを'OFF'に設定する例です。

```
SET OPTION User1.BLOCKING = 'OFF';
```

Note

この項では、オプションの設定方法を中心に説明しています。BLOCKINGオプションの詳細については、「9.7　ブロックと独立性レベル」を参照してください。

SET文の3番目の形式は、ローカル・オプションの現在の値を変更するために使用されます。

```
<set_temporary_local_option> ::= SET TEMPORARY OPTION
                                    <option_name> "=" [ <option_value> ]
```

SET TEMPORARY OPTIONは、SYSOPTIONテーブルの内容には影響しません。NULLでないオプション値が指定された場合、その値はすぐに現在の接続に影響を与えます。NULLがオプション値として指定された場合、現在の設定は、そのオプションに対して実行された他のどのSET TEMPORARY OPTION文よりも前に有効だった値に復元されます。

グローバル・オプションのパブリック設定を変更した場合、その変更はすぐにすべてのユーザと接続に影響を与えます。これはグローバル・オプションの性質です。反対に、ローカル・オプションの変更は、

どのような変更であっても、すでに開かれている他の接続において有効な現在の設定に影響を与えることは決してありません。つまり、ある接続のローカル・オプションでは、すでに実行中の別の接続の設定にすぐに影響を与えるような設定はできません。

とても複雑なルールですが、ローカル・オプションの変更が現在の接続に与える影響は、すぐに有効になる場合もあればそうでない場合もあります。最終的な結果は、実行されるSET文の種類、SYSOPTIONテーブルの現在の内容、SET文で指定されるオプション値がNULLかそうでないかによって変わります。以下の例では、新規データベースへのある接続においてANSI_INTEGER_OVERFLOWという同一のローカル・オプションに対する変更を14回連続して行っています。最初の状態では、パブリック設定がデフォルト値'OFF'になっており、SYSOPTIONテーブルのDBAユーザには何も設定がありません。DBAユーザが接続すると、現在の設定はパブリック設定に初期化されるため、'OFF'になります。

次に、14回の連続したSET文の番号付きリストと共に、各文の右側のカラムにその実行結果を示します。

```
                                                                  この接続に対する
番号   文                                                         現在の設定
==  ==========================================================  ===================
1   SET EXISTING OPTION PUBLIC.ANSI_INTEGER_OVERFLOW = 'ON'          On
2   SET OPTION DBA.ANSI_INTEGER_OVERFLOW             = 'OFF'         Off
3   SET EXISTING OPTION PUBLIC.ANSI_INTEGER_OVERFLOW = 'ON'          Off
4   SET OPTION DBA.ANSI_INTEGER_OVERFLOW             = NULL          On
5   SET EXISTING OPTION PUBLIC.ANSI_INTEGER_OVERFLOW = 'OFF'         Off
6   SET OPTION DBA.ANSI_INTEGER_OVERFLOW             = 'ON'          On
7   SET TEMPORARY OPTION ANSI_INTEGER_OVERFLOW       = 'OFF'         Off
8   SET OPTION DBA.ANSI_INTEGER_OVERFLOW             = 'ON'          On
9   SET EXISTING OPTION PUBLIC.ANSI_INTEGER_OVERFLOW = 'OFF'         On
10  SET TEMPORARY OPTION ANSI_INTEGER_OVERFLOW       = NULL          On
11  SET OPTION DBA.ANSI_INTEGER_OVERFLOW             = NULL          Off
12  SET TEMPORARY OPTION ANSI_INTEGER_OVERFLOW       = 'ON'          On
13  SET EXISTING OPTION PUBLIC.ANSI_INTEGER_OVERFLOW = 'OFF'         On
14  SET TEMPORARY OPTION ANSI_INTEGER_OVERFLOW       = NULL          Off
```

以下に、これらの変更がどのように働くかを説明します。同じSET文であっても、現在の設定に異なる影響を与える場合があるなど、意外な結果もあります。

❶ ユーザ設定またはテンポラリ設定に有効なものがなかったため、新しいパブリック設定はすぐに有効になった

❷ 現在接続中のユーザに対するユーザ設定の変更はいつでもすぐに有効になる

❸ ユーザ設定が有効になっているため、新しいパブリック設定を行っても現在の設定は変更されなかった

❹ ユーザ設定が削除されたため、先ほどのパブリック設定が有効になった

❺ 上記 1. の変更と同じ理由により、新しいパブリック設定がすぐに有効になった
❻ 新しいユーザ設定はいつでもすぐに有効になるため、ここでもそうなった
❼ テンポラリの変更はいつでもすぐに有効になる
❽ テンポラリの変更が行われた後ではあるが、上記 2. の変更と同じように新しいユーザ設定が有効になった
❾ 上記 3. の変更と同じ理由により、新しいパブリック設定を行っても現在の設定は変更されなかった
❿ テンポラリ設定を削除しても影響はなかった
⓫ ユーザ設定を削除することによって、現在の設定が先ほどのパブリック設定に復元された
⓬ 上記 7. の変更と同じように、テンポラリ設定はいつでもすぐに有効になる
⓭ 今度はテンポラリ設定が存在していたので、新しいパブリック設定を行っても現在の設定は変更されなかった
⓮ 今度はテンポラリ設定を削除したことによって、現在の設定が先ほどのパブリック設定に復元された

Tip

オプションの設定は簡潔に行うようにします。一度、前もってパブリックとユーザの各デフォルト設定を変更したら、これらのデフォルト設定の変更は処理の途中で行いません。その代わりに、処理の途中で変更を行う場合には SET TEMPORARY OPTION 文を使用します。また、古い設定を復元するには、NULL 値を使用せずに新しい値を明示的に指定するようにします。

Section

9.3 トランザクション

データベースのトランザクションとは、データベースの整合性を保証しながら、正しく要求を満たすための 1 つのまとまりとして処理される一連の SQL 文です。トランザクションの設計は、アプリケーション設計の重要な部分であり、データベースの正当性と整合性だけでなくパフォーマンスにも深刻な影響を与えます。

トランザクションは 4 つの要件を満たさなければなりません。すなわち、アトミック性、一貫性、独立性、持続性であり、それぞれの英語の頭文字をとって "ACID" と記されます。

- **アトミックである**——トランザクションによってデータベースに加えられるすべての変更は、トランザクションの正常終了時に完了していなければならない。トランザクションが失敗した場合

は、データベースにいかなる変更も加えられないものとする。トランザクション途中の任意の時点での失敗によって、そのトランザクション全体が失敗する。失敗は、ROLLBACK のようなSQL 文によって明示的に発生させられる場合や、システム・クラッシュのような外部イベントによって暗黙的に発生する場合がある。たとえば、sales_order ローと、それに対応する 1 つ以上の sales_order_item ローとを ASADEMO データベースに挿入するトランザクションの途中で 1 箇所でも失敗が生じると、トランザクション完了時には、これらのローはいずれも存在しない。

- **一貫性がある**——トランザクションの完了時には、データに関するすべての制約が満たされていなければならない。この制約には、データベース・スキーマ内で定義されるもの以外に、アプリケーション・プログラムによって管理される制約も含まれる。
- **独立性がある**——別々のトランザクションによって、同一のデータが同時に変更されてはならない。つまり、同一データに対するすべての変更は互いに独立していなければならない。SQL Anywhere での独立性の最小単位はローである。別々のトランザクションどうしが、他のトランザクションによって加えられた変更の影響を確認するかどうかを考慮する場合も独立性が問題になることがある。この点に対して適用される ISOLATION_LEVEL オプションについては、この章の後の部分で説明する。しかし、更新に関するかぎり、独立性はオプションではない。あるトランザクションによって、いったんテーブル内のローが変更されると、SQL Anywhere は、そのトランザクションが完了するまでは、他のどのトランザクションに対してもそのローの変更を許可しない。
- **持続性がある**——正常に終了するトランザクションによってデータベースに加えられるすべての変更は、持続的でなければならず、このトランザクションによる変更は、それ以降に生じる失敗の影響を受けてはならない。つまり、COMMIT は永続的でなければならない。

たとえば、あるトランザクションによって、sales_order ローと、それに対応する 1 つ以上の sales_order_item ローとを ASADEMO データベースに挿入する場合を考えます。このトランザクションはアトミックです。トランザクションの成功時には挿入されたローのすべてがデータベースに存在し、失敗時にはどのローもデータベースに残らないからです。このトランザクションには一貫性があります。どの sales_order ローに対しても少なくとも 1 つの対応する sales_order_item のローが存在しなければならないというアプリケーションの制約と、どの sales_order_item ローも 1 つの対応する sales_order のローを持たなければならないというデータベースの外部キー制約とを満たすからです。このトランザクションは独立性を満たします。トランザクションが正常に完了するまでは、挿入されたローの削除または更新を行えるデータベースの接続が他に存在しないからです。このトランザクションには持続性があります。いったんトランザクションが正常に完了すると、挿入されたローがデータベースに残るからです。

それぞれのトランザクションには開始と終了があり、単一の接続におけるコンテキストの範囲内で実行されます。1 つの接続において 2 つ以上のトランザクションが実行されることがあります。ただし、シリアル形式で 1 つずつ順に実行されます。別々の接続において、それぞれのトランザクションが同時に実行されることがあります。これらのトランザクションは互いに影響を与えることができますが、1 つのトランザクションが 2 つ以上の接続にわたって実行されることはありません。

Note

複合文でない個々のSQL文は、事実上アトミックです。これは、そうしたSQL文が失敗した場合には、すでにデータベースに加えられたどのような変更も自動的に取り消されることを意味します。これは、2つ以上のローを操作する単一のINSERT、UPDATE、DELETEの各文に適用されます。1つ以上のローに影響を与えた後でこれらの文の1つが失敗した場合、その影響のすべてが自動的に取り消されます。このアクティビティは、トランザクション制御とは別のものです。ある文の失敗によってトランザクションが自動的にロールバックされることはありません。SQL文のアトミックな性質は、セーブポイントを介して内部的に実装されています。これについては、「9.4　セーブポイントとサブトランザクション」で説明します。

3種類のSQL文、BEGIN TRANSACTION、COMMIT、ROLLBACKは、トランザクションの開始と終了を明示的に制御するために使用されます。こうした制御は、すべての場合において絶対的なわけではありません。ROLLBACK文は常にトランザクションを終了させますが、BEGIN TRANSACTION文は必ずしもトランザクションを開始させるわけではなく、COMMIT文は必ずしもトランザクションを終了させるわけではありません。

```
<begin_transaction> ::= BEGIN TRAN        -- すべての形式は等価
                      | BEGIN TRANSACTION
                      | BEGIN TRAN <transaction_name>
                      | BEGIN TRANSACTION <transaction_name>
<transaction_name>  ::= <identifier>      -- いかなる目的にも使用されない

<commit>            ::= COMMIT            -- すべての形式は等価
                      | COMMIT WORK
                      | COMMIT TRAN
                      | COMMIT TRANSACTION
                      | COMMIT TRAN <transaction_name>
                      | COMMIT TRANSACTION <transaction_name>

<rollback>          ::= ROLLBACK          -- すべての形式は等価
                      | ROLLBACK WORK
                      | ROLLBACK TRAN
                      | ROLLBACK TRANSACTION
                      | ROLLBACK TRAN <transaction_name>
                      | ROLLBACK TRANSACTION <transaction_name>
```

Tip

BEGIN TRANSACTION、COMMIT、またはROLLBACKの各文のトランザクション名のパラメータにこだわらないようにします。これらのパラメータは、SQL Anywhereに影響を与えません。また、各文のための異なった形式は等価なものです。完全な構文をここに示したのは、こうした異なった形式がマニュアルとユーティリティ・プログラムの出力に表示されて、不要な混乱を招くことがよくあるから

です。

SQL Anywhere には、「連鎖モード」と「非連鎖モード」という 2 つのトランザクション制御のモードがあります。

- 「連鎖モード」では、INSERT、UPDATE、または DELETE、あるいはロックを取得する SELECT であれば、どの文によってもトランザクションが暗黙的に開始される。
  このトランザクションは、明示的な COMMIT または ROLLBACK 文の実行時、あるいはトランザクションの失敗時に終了する
- 「非連鎖モード」では、BEGIN TRANSACTION 文によってトランザクションが明示的に開始される。こうしたトランザクションは、明示的な COMMIT または ROLLBACK 文の実行時、あるいはトランザクションの失敗時に終了する。BEGIN TRANSACTION 文が 1 つも実行されない場合、それぞれの文はそれ自身がトランザクションとして実行され、正常に動作するときは暗黙的なコミットを、失敗するときは暗黙的なロールバックを伴う

次に示すのは、連鎖モードで実行される 2 つの UPDATE 文の簡単な例です。SET TEMPORARY OPTION CHAINED = 'ON' という文は、連鎖モードが有効であることを明確に示すために使用されています。

```
SET TEMPORARY OPTION CHAINED = 'ON';
UPDATE t1 SET non_key_1 = 'xxx' WHERE key_1 = 1;
UPDATE t1 SET non_key_1 = 'yyy' WHERE key_1 = 2;
COMMIT;
```

トランザクション・ログ・ファイルを調べれば、どのタイミングでトランザクションが開始し、終了したかがわかります。dbtran.exe というユーティリティ・プログラムを使用すると、このログ・ファイルを判読可能な SQL 文に変換できます。以下にコマンド・ラインは、dbtran.exe を実行します。コミットされていないものも入れてすべての操作を含める -a オプション、接続のパラメータを指定する -c オプション、UPDATE 文を ANSI 形式で生成する -s オプション、確認メッセージを表示せずに出力ファイルを上書きする -y オプションをそれぞれ使用しています。最後のパラメータは、出力テキスト・ファイルとして test9_log.sql を指定しています。

```
"%ASANY9%¥win32¥dbtran.exe" -a -c "DSN=test9" -s -y test9_log.sql
```

先ほど示した UPDATE と COMMIT 文の場合に、どのような出力が得られるかを以下に示します。BEGIN TRANSACTION のエントリは、トランザクションが開始されてから最初の UPDATE が行われたことを示しています。

```
--BEGIN TRANSACTION-1001-0000402114
BEGIN TRANSACTION
go
--UPDATE-1001-0000402115
UPDATE DBA.t1
   SET non_key_1='xxx'
 WHERE key_1=1
go
--UPDATE-1001-0000402126
UPDATE DBA.t1
   SET non_key_1='yyy'
 WHERE key_1=2
go
--COMMIT-1001-0000402137
COMMIT WORK
go
```

次は、先ほどと同じ2つのUPDATE文であり、今度は、明示的なCOMMITを使わずに非連鎖モードで実行します。

```
SET TEMPORARY OPTION CHAINED = 'OFF';
UPDATE t1 SET non_key_1 = 'xxx' WHERE key_1 = 1;
UPDATE t1 SET non_key_1 = 'yyy' WHERE key_1 = 2;
```

今回のdbtran.exeの出力は、2つの別々のBEGIN TRANSACTIONとCOMMIT操作が実行されたことを示しています。

```
--BEGIN TRANSACTION-1001-0000402237
BEGIN TRANSACTION
go
--UPDATE-1001-0000402238
UPDATE DBA.t1
   SET non_key_1='xxx'
 WHERE key_1=1
go
--COMMIT-1001-0000402249
COMMIT WORK
go
--BEGIN TRANSACTION-1001-0000402250
BEGIN TRANSACTION
go
--UPDATE-1001-0000402251
UPDATE DBA.t1
   SET non_key_1='yyy'
 WHERE key_1=2
go
--COMMIT-1001-0000402262
```

```
COMMIT WORK
go
```

非連鎖モードを使ったアプリケーションでは、明示的なBEGIN TRANSACTION文を発行しないかぎり、トランザクションの設計またはスコープを制御できません。以下に示すのは、制御を行えるように先ほどの例を変更して、両方のUPDATE文を強制的に1つのトランザクションに含めるようにした例です。

```
SET TEMPORARY OPTION CHAINED = 'OFF';
BEGIN TRANSACTION;
UPDATE t1 SET non_key_1 = 'xxx' WHERE key_1 = 1;
UPDATE t1 SET non_key_1 = 'yyy' WHERE key_1 = 2;
COMMIT;
```

以下に示す、非連鎖モードでBEGIN TRANSACTIONを使用した場合のdbtran.exeの出力は、先ほどの連鎖モードを使用した最初の例の場合とまったく同様で、BEGIN TRANSACTIONが1つ、UPDATEが2つ、COMMITが1つ、それぞれ含まれています。

```
--BEGIN TRANSACTION-1001-0000402314
BEGIN TRANSACTION
go
--UPDATE-1001-0000402315
UPDATE DBA.t1
   SET non_key_1='xxx'
 WHERE key_1=1
go
--UPDATE-1001-0000402326
UPDATE DBA.t1
   SET non_key_1='yyy'
 WHERE key_1=2
go
--COMMIT-1001-0000402337
COMMIT WORK
go
```

Note

dbtran.exeの出力におけるBEGIN TRANSACTIONエントリは、明示的なBEGIN TRANSACTION文が実行された時点ではなく、トランザクションが実際に開始されたところをマークするのに役立ちます。細かいことですが、dbtran.exeによる出力を読む際に重要となります。トランザクションのモードにかかわらず、アプリケーション・コード内の明示的なBEGIN TRANSACTION文は、トランザクション・ログ内のBEGIN TRANSACTIONエントリに対応することもあれば、そうでないこともあります。

連鎖モードはマニュアル・モードとも呼ばれます。トランザクションの終了をマークするために、明示的なCOMMITとROLLBACK文が必要なためです。非連鎖モードはオートコミット・モードとも呼ばれます。明示的なBEGIN TRANSACTION文が1つも実行されていなくても、成功した文のそれぞれが自動的にコミットされるためです。ただし、オートコミット・モードには次の2種類があります。

- 「サーバ側オートコミット・モード」は、CHAINEDオプションを'OFF'に設定して上記の非連鎖モードにした場合に、SQL Anywhereデータベース・エンジンによって実装される。この場合、明示的なBEGIN TRANSACTION文は実行されない。トランザクションの開始と終了は、データベース・エンジンの内部で自動的に行われる。BEGIN TRANSACTION、COMMIT、またはROLLBACK文がクライアント側から送られることはない
- 「クライアント側オートコミット・モード」は、ODBCやJDBCのようなクライアント・データベース・インタフェースのソフトウェアによって実装される。インタフェースのAutoCommitフラグが設定されていると、INSERT、UPDATE、DELETEの各文の後に明示的なCOMMIT文がSQL Anywhereに送られる。サーバ側のオートコミットも有効になっている場合は、SQL AnywhereはCOMMIT文を受け取るまでにコミットを完了しているため、これらの追加されたCOMMITは何の影響も与えない。サーバ側のオートコミットが有効になっていない場合は、クライアント側のオートコミットによって送られたCOMMIT文は影響を与える

次の表は、サーバ側とクライアント側のオートコミット設定における4つの組み合わせのもとで、ODBCを使ったPowerBuilder【監注11】アプリケーションによってUPDATE文が発行された場合にどうなるかを示します。1番目のカラムは、ODBC経由でデータベースに接続するときにPowerBuilderアプリケーションが使用するクライアント側SQLCA.AutoCommitの設定です。2番目のカラムは、SQL Anywhereが使用するサーバ側CHAINEDオプションの設定です。3番目のカラムは、クライアント・サーバ・インタフェースを介してODBCからSQL Anywhereに実際に送られたSQL文です。4番目のカラムは、SQL Anywhereによって実行された内部操作を、トランザクション・ログに記録されたのと同じように表しています。

クライアント側 ODBCのAutoCommit	サーバ側 CHAINEDモード	送信された文	実行された操作
FALSE	OFF	UPDATE	Begin, update, commit
TRUE	OFF	UPDATE, COMMIT	Begin, update, commit
TRUE	ON	UPDATE, COMMIT	Begin, update, commit
FALSE	ON	UPDATE	Update

オートコミット・モードは、上記の最初の3つの組み合わせで有効になっています。重要なのは、オートコミットがクライアント側のものかサーバ側のものかではなく、データベースの変更のそれぞれが、

【監注11】サイベース社の統合開発環境

それ自体トランザクションとして扱われていることです。オートコミットが有効になっている状態では、あるアプリケーションのトランザクションが複数の INSERT、UPDATE、または DELETE 文にまたがることは不可能です。

Tip

オートコミットは決して使用しないようにします。トランザクションを明示的に制御するようにアプリケーションを設計します。上記の表の最後の行に示した設定を使用して、CHAINED オプションは常に 'ON' に、クライアント側の AutoCommit フラグはすべて FALSE に設定し、トランザクションの終了時に COMMIT または ROLLBACK 文を明示的に行うようにします。

SQL Anywhere は、COMMIT 文の実行時に実際にコミット操作を行うこともあれば行わないこともあります。これは @@TRANCOUNT という組み込みの接続レベル変数の現在の値によって決まります。@@TRANCOUNT の値が 0 または 1 の場合に COMMIT 文が実行されると、SQL Anywhere はコミットを行います。@@TRANCOUNT が 2 以上の場合、SQL Anywhere は COMMIT 文を無視します。以下に、@@TRANCOUNT の値がどのように変更されて使われるかの詳細を示します。

- @@TRANCOUNT の値は、接続が開始されると 0 に設定され、トランザクションの終了時も常に 0 に戻される
- 非連鎖モードでは、明示的な BEGIN TRANSACTION 文 1 つにつき @@TRANCOUNT の値が 1 だけ増える
- 連鎖モードでは、暗黙的なトランザクションが開始されると、@@TRANCOUNT の値は 0 になり、この値はすぐには変更されない。しかし、次の明示的な BEGIN TRANSACTION 文によって @@TRANCOUNT の値は 2 に設定され、以降の BEGIN TRANSACTION 文によって @@TRANCOUNT の値は 1 ずつ増える
- 連鎖モードでは、トランザクションが暗黙的に開始される前に明示的な BEGIN TRANSACTION 文が実行されると @@TRANCOUNT の値は 1 に設定される。以降の BEGIN TRANSACTION 文によって @@TRANCOUNT の値は 1 ずつ増える
- 連鎖モードと非連鎖モードのどちらでも、@@TRANCOUNT の値が 0 になっていないかぎり、COMMIT 文 1 つの実行につき @@TRANCOUNT の値は 1 ずつ減る。@@TRANCOUNT の値が 0 の場合に COMMIT 文が実行されると、実際のコミット操作が行われる。すべてのトランザクションによるデータベース変更は永続的であり、トランザクションが保持するロックがすべて解放されると、そのトランザクションは終了する。@@TRANCOUNT の値が 0 でない場合に COMMIT 文が実行されても、それ以上は何も起こらない。未処理のままになっているデータベース変更とロックに関するかぎり、COMMIT 文は無視され、トランザクションは進行中のままである
- 連鎖モードと非連鎖モードどちらでも、ROLLBACK 文は、@@TRANCOUNT の値は 0 に設定し、そのトランザクションによるすべてのデータベース変更のロールバックとすべてのロックの解放を

行って、トランザクションを終了させる。これは、@@TRANCOUNT の現在の値に関係なく起こる

「ネストされたトランザクション」という用語は、@@TRANCOUNT の値が 2 以上になるときに使用されることがあります。しかし、この用語は誤解を招く恐れがあります。データベース変更とロックに関するかぎり、最も外側のトランザクションだけが意味を持つためです。@@TRANCOUNT の値が 2 以上になると、COMMIT 文が @@TRANCOUNT の値を下げる以外の動作を行うことは絶対にありません。ネストされたトランザクションとは、外側のトランザクションで行われた変更がロールバックされても、内部のトランザクションで行われた変更が永続的になる場合があることを意味します。しかし、SQL Anywhere ではこうしたことは起こり得ず、ネストされたトランザクションというものは存在しません。

Tip

BEGIN TRANSACTION 文は一切使用しないようにします。その代わりに連鎖モードを使用します。SQL Anywhere に各トランザクションを暗黙的に開始させ、明示的な COMMIT と ROLLBACK 文を使用して各トランザクションを終了させます。また、余計な BEGIN TRANSACTION 文で COMMIT 文を無効にしないようにします。

以下に示すのは、簡単なトランザクションの例です。ASADEMO データベースの sales_order テーブルに親ローが挿入され、対応する子ローが sales_order_items テーブルに挿入されます。最初の INSERT によってトランザクションが開始されます。INSERT 文が両方とも正常に動作すると、トランザクションは COMMIT 文によって終了します。外部キー違反が検出されると、SQLSTATE が '23503' に設定され、例外ハンドラの ROLLBACK によってトランザクションが終了します。

```
BEGIN
   DECLARE @errormsg  VARCHAR ( 32767 );
   DECLARE error_23503 EXCEPTION FOR SQLSTATE '23503';

   INSERT sales_order ( id, cust_id, order_date, fin_code_id, region, sales_rep )
      VALUES ( 1, 101, CURRENT DATE, 'r1', 'Eastern', 299 );

   INSERT sales_order_items ( id , line_id, prod_id, quantity, ship_date )
      VALUES ( 1, 1, 999, 12, DATEADD ( DAY, 1, CURRENT DATE ) );

   COMMIT;
   MESSAGE 'COMMIT OK.' TO CONSOLE;

EXCEPTION
   WHEN error_23503 THEN
      SET @errormsg = ERRORMSG();
```

```
        ROLLBACK;
        MESSAGE 'ROLLBACK after error.' TO CONSOLE;
        MESSAGE STRING ( 'ERRORMSG() = ', @errormsg ) TO CONSOLE;
END;
```

特にこの場合は、prod_id の値 999 によって外部キー違反が発生し、トランザクションは ROLLBACK によって終了します。以下は、MESSAGE 文からの出力です。

```
ROLLBACK after error.
ERRORMSG() =「テーブル 'sales_order_items' の外部キー 'ky_prod_id' に対応するプライマリ・キーの
    値がありません」
```

ASADEMO データベース内のテーブルの詳細については、「3.6　マルチテーブル・ジョイン」を参照してください。BEGIN ブロックと例外ハンドラの詳細については、「第 8 章　パッケージ」の 8.2 項と 8.3 項を参照してください。

Tip

トランザクション設計の際、副次的にコミットを行う SQL 文に注意します。SQL Anywhere のシステム・カタログの更新を行う文はすべてこのカテゴリに分類され、進行中のトランザクションは終了します。たとえば、SET OPTION、GRANT、CREATE SYNCHRONIZATION USER のような文はすべてコミットを行い、グローバル永続テーブルとグローバル・テンポラリ・テーブルで使用される CREATE TABLE と CREATE INDEX もコミットを実行します。LOAD TABLE 文も、TRUNCATE_WITH_AUTO_COMMIT オプションが ON の場合の TRUNCATE TABLE も、コミットを行います。TRUNCATE TABLE の詳細については、5.6 項を参照してください。しかし、スキーマに影響を与える文には、コミットを行わないものもあります。CREATE DATABASE はコミットを行いません。これはおそらく CREATE DATABASE が現在のデータベースに影響を与えず、ローカル・テンポラリ・テーブルに関わるどのような文も現在のデータベースに影響を与えないためです。SET TEMPORARY OPTION 文もまたコミットを行わないため、トランザクションの進行中に接続オプションを変更できます。

Section

9.4 セーブポイントとサブトランザクション

セーブポイントは、部分的なロールバックを実装するための柔軟な手法を提供します。「savepoint（セーブポイント）」とは、現在の接続でのトランザクション処理途中でマークを付けるサブトランザクションの開始時点です。サブトランザクションは、トランザクション全体のロールバックまたは終了を行わずにロールバックできますが、トランザクション全体に対するコミットと終了を行わないかぎりコミットできません。

```
<savepoint>              ::= SAVEPOINT [ <savepoint_name> ]
<savepoint_name>         ::= <identifier> -- RELEASE と ROLLBACK で使用

<rollback_to_savepoint> ::= ROLLBACK TO SAVEPOINT [ <savepoint_name> ]

<release_savepoint>      ::= RELEASE SAVEPOINT [ <savepoint_name> ]
```

SAVEPOINT 文は、セーブポイントを作成します。オプションとしてセーブポイント名を指定できます。後続の SAVEPOINT 文によってネストされたサブトランザクションを作成でき、名前を与えることもできます。

セーブポイントには、アクティブまたは非アクティブの状態があります。アクティブなセーブポイントとは、SAVEPOINT 文によって設定された後、次の RELEASE SAVEPOINT、ROLLBACK TO SAVEPOINT、ROLLBACK、または COMMIT 文によってまだ非アクティブにされていないものです。

ROLLBACK TO SAVEPOINT 文は、特定のアクティブ・セーブポイント以降に行われた変更のすべてをロールバックしてからセーブポイントを非アクティブにします。セーブポイント名が指定されない場合、ROLLBACK TO SAVEPOINT 文は最後に設定されたアクティブなセーブポイントを暗黙的に参照します。ROLLBACK TO SAVEPOINT 文によって、セーブポイントに名前を付けることができます。この場合、そのセーブポイント以降に行われた変更のすべてがロールバックされ、名前を付けたセーブポイントに加えて、そのセーブポイント以降に設定されたすべてのセーブポイントが非アクティブになります。RELEASE SAVEPOINT は、どのような変更もロールバックしません。セーブポイントを非アクティブにするだけです。セーブポイント名が指定されない場合、RELEASE SAVEPOINT 文は、最後に設定されたアクティブなセーブポイントを暗黙的に参照します。RELEASE SAVEPOINT 文によって、セーブポイントに名前を付けることができます。この場合、名前を付けたセーブポイントに加えて、そのセーブポイント以降に設定されたすべてのセーブポイントが非アクティブになります。

非アクティブなセーブポイントを参照する ROLLBACK TO SAVEPOINT または RELEASE SAVEPOINT 文を実行すると、エラーになります。

ROLLBACK と COMMIT 文はどちらも、すべてのサブトランザクションと共にトランザクション全体を終了させ、そのトランザクションのすべてのセーブポイントを非アクティブにします。

以下に示すのは、後で説明するセーブポイント処理の3つの例で使用する簡単なテーブルです。それぞれの例は、その前の例の終了時の値ではなく、ここに示した2つのローと同じ値で始まることに注意してください。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL PRIMARY KEY,
   non_key_1     VARCHAR ( 100 ) NOT NULL );

INSERT t1 VALUES ( 1, 'AAA' );
INSERT t1 VALUES ( 2, 'BBB' );
COMMIT;
```

最初の例は、1番目のローを2回、2番目のローを1回、それぞれ更新し、セーブポイントを1つ設定するトランザクションです。ROLLBACK TO SAVEPOINT文は、1番目のローに対する2回目の更新をロールバックしますが、1回目の更新はロールバックしません。また、このROLLBACK TO SAVEPOINT文はトランザクションを終了させず、以降のSQL文にも影響しないため、2番目のローに対する更新とCOMMIT文は正常に実行されます。

```
UPDATE t1 SET non_key_1 = '1' WHERE key_1 = 1;
SAVEPOINT;
UPDATE t1 SET non_key_1 = '2' WHERE key_1 = 1;
ROLLBACK TO SAVEPOINT;
UPDATE t1 SET non_key_1 = 'XXX' WHERE key_1 = 2;
COMMIT;
SELECT * FROM t1;
```

以下は、最後のSELECTからの出力です。1番目のローに対する1回目の更新と2番目のローに対する更新はコミットされていますが、1番目のローに対する2回目の更新はコミットされていません。

```
key_1  non_key_1
=====  =========
1      '1'
2      'XXX'
```

第2の例は、ネストされたセーブポイントで、ネストを示すためのインデントを付けています。この例のROLLBACK TO SAVEPOINT文は、2番目のSAVEPOINT文を暗黙的に参照していることに注意してください。

```
UPDATE t1 SET non_key_1 = '3' WHERE key_1 = 1;
SAVEPOINT;
   UPDATE t1 SET non_key_1 = '4' WHERE key_1 = 1;
   SAVEPOINT;
```

```
      UPDATE t1 SET non_key_1 = '5' WHERE key_1 = 1;
   ROLLBACK TO SAVEPOINT;
   UPDATE t1 SET non_key_1 = 'YYY' WHERE key_1 = 2;
COMMIT;
SELECT * FROM t1;
```

以下は、SELECTからの出力です。ROLLBACK TO SAVEPOINT文によって、1番目のローに対する3回目の更新はロールバックされていますが、最初の2回の更新はロールバックされていないこと、また、2番目のローに対する更新が正常に行われていることがわかります。

```
key_1  non_key_1
=====  =========
1      '4'
2      'YYY'
```

第3の例は、名前付きのセーブポイントです。このスクリプトでは、テーブルt1の1番目のローの更新を進めるなかで、4つのネストされたセーブポイントを設定しています。ROLLBACK TO SAVEPOINT s7という文は、2番目のセーブポイントに遡ってそれ以降に行われたすべての変更をロールバックします。セーブポイント名を使用することで、その間に起こったセーブポイントを明示的に処理する必要がなくなります。その後、このスクリプトは2番目のローを更新し、COMMITによってトランザクションを終了します。

```
UPDATE t1 SET non_key_1 = '6' WHERE key_1 = 1;
SAVEPOINT s6;
   UPDATE t1 SET non_key_1 = '7' WHERE key_1 = 1;
   SAVEPOINT s7;
      UPDATE t1 SET non_key_1 = '8' WHERE key_1 = 1;
      SAVEPOINT s8;
         UPDATE t1 SET non_key_1 = '9' WHERE key_1 = 1;
         SAVEPOINT s9;
            UPDATE t1 SET non_key_1 = '10' WHERE key_1 = 1;
   ROLLBACK TO SAVEPOINT s7;
   UPDATE t1 SET non_key_1 = 'ZZZ' WHERE key_1 = 2;
COMMIT;
SELECT * FROM t1;
```

以下は、SELECTからの出力です。1番目のローに対する最後の3回の更新はすべてロールバックされ、その他の変更はコミットされていることがわかります。

```
key_1  non_key_1
=====  =========
1      '7'
2      'ZZZ'
```

RELEASE SAVEPOINT 文を使用して、異なるセーブポイント間の関係を SQL モジュールの呼び出し構造に合わせることができます。各モジュールが SAVEPOINT 文で始まり、ROLLBACK TO SAVEPOINT または RELEASE SAVEPOINT 文で終わる場合、ネストされたサブトランザクションを作成することなく、連続して異なるモジュールを呼び出せます。しかし、あるモジュールに別のモジュールに対するネストされた呼び出しが含まれているときは、ネストされたサブトランザクションが作成されます。

以下に示す別の例では、最初の UPDATE が、SAVEPOINT と RELEASE SAVEPOINT 文によるサブトランザクションの中にあります。2つめの UPDATE は、先ほどのサブトランザクションに続く第2のサブトランザクションの中にあります。このサブトランザクションは、最初のサブトランザクション内にネストされていません。その前の RELEASE SAVEPOINT 文によって最初のサブトランザクションが終了しているためです。3つめの UPDATE は、第3のサブトランザクションの中にあります。このサブトランザクションは、第2のトランザクション内にネストされています。第3のサブトランザクションは RELEASE SAVEPOINT 文によって終了し、第2のサブトランザクションは ROLLBACK TO SAVEPOINT 文によって終了します。第3のサブトランザクションはネストされているため、第2と第3のサブトランザクションにおいて行われた変更はすべてロールバックされます。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL PRIMARY KEY,
   non_key_1     VARCHAR ( 100 ) NOT NULL,
   non_key_2     VARCHAR ( 100 ) NOT NULL,
   non_key_3     VARCHAR ( 100 ) NOT NULL );

INSERT t1 VALUES ( 1, 'AAA', 'BBB', 'CCC' );
COMMIT;

SAVEPOINT;
   UPDATE t1 SET non_key_1 = 'xxx' WHERE key_1 = 1;
RELEASE SAVEPOINT;
SAVEPOINT;
   UPDATE t1 SET non_key_2 = 'yyy' WHERE key_1 = 1;
   SAVEPOINT;
      UPDATE t1 SET non_key_3 = 'zzz' WHERE key_1 = 1;
   RELEASE SAVEPOINT;
ROLLBACK TO SAVEPOINT;
COMMIT;
SELECT * FROM t1;
```

以下は、SELECT からの出力です。第1のサブトランザクションによる変更はコミットされていますが、第2と第3のサブトランザクションによる変更はコミットされていないことがわかります。

```
key_1  non_key_1  non_key_2  non_key_3
=====  =========  =========  =========
1      'xxx'      'BBB'      'CCC'
```

シングルユーザ・アプリケーションを作成する際、セーブポイントを使用すると、ユーザによる変更の取り消し機能の実装を簡素化できます。変更すべき内容を別途保存して後から適用するのではなく、SAVEPOINT 文を実行して直接データベースを更新できます。ユーザが変更の取り消しを決めた場合、アプリケーションは ROLLBACK TO SAVEPOINT 文を実行できます。連続して存在するネストされたサブトランザクションの組み合わせを変えれば、異なるアプリケーション・ダイアログに対応できます。この方法はマルチユーザ環境には向いていません。ユーザが次の行動を決めるまでの間ずっとロックが保持され、トランザクションの実行が長引くと、他のユーザとの間で競合が起こる可能性があるからです。しかし、シングルユーザ環境では、ロックはそれほど問題になりません。競合する心配のある他のユーザがいないためです。

Tip

マルチユーザ環境では、トランザクションが開始されてから、COMMIT または ROLLBACK によって終了するまでは、キーボードまたはマウスの制御を決してユーザに与えないようにします。データベースの変更によってローがロックされ、他のユーザが作業できなくなります。トランザクションの処理時間はできるだけ短くしてください。特に、エラー検出時は、診断メッセージをユーザに表示した後ではなく、表示する前に ROLLBACK 文を実行します。

Note

「シングルユーザ」と「マルチユーザ」という用語が一般的に使われていますが、「シングル接続」と「マルチ接続」という表現のほうが、より正確です。トランザクションとロックに関するかぎり、数える対象はユーザではなくトランザクションだからです。このことは、すべての接続が同じユーザ ID を指定する場合にも、実行中のアプリケーションが 1 つだけの場合にもあてはまります。データベースへの接続が複数あれば、トランザクションが重複する可能性があるためです。

Note

SQL Anywhere では、セーブポイントを内部で使用して、複合文でない SQL 文のアトミック性とアトミックな BEGIN ブロックを実現しています。内部のセーブポイントはクライアント・アプリケーションからは見えません。しかし、失敗した INSERT、UPDATE、または DELETE 文が必ずしも進行中のトランザクションに影響を与えることなく自動的に取り消されるのは、こうしたセーブポイントの影響です。

Section

9.5 エラー処理

SQL Anywhere は、SQL モジュールが自らの実行を中断してユーザ定義の例外またはエラーの処理を行える方法をいくつか提供しています。こうした方法には、次の項で説明する SIGNAL と RESIGNAL 文、その後の 2 つの項で説明する RAISERROR と ROLLBACK TRIGGER 文があります。

9.5.1 SIGNALとRESIGNAL

SIGNAL と RESIGNAL 文は、指定された SQLSTATE の値を使って例外条件を明示的に発生させます。

```
<signal_exception>   ::= SIGNAL <exception_name>
<exception_name>     ::=「第 8 章　パッケージ」の <exception_name> を参照
<resignal_exception> ::= RESIGNAL
```

Note

プロシージャと他の BEGIN ブロック内にある例外ハンドラの概要については、「8.3　例外ハンドラ」を参照してください。

SIGNAL 文を使用して、外部キー違反を表す '23503' などの定義済み SQLSTATE 値を明示的に発生させることができます。また、SIGNAL 文を使用して、'99000' から '99999' までの範囲のユーザ定義 SQLSTATE 値を発生させることもできます。

SIGNAL 文を実行すると、以降の文はすべてスキップされ、制御はすぐに現在の BEGIN ブロック内の例外ハンドラに移されます。現在のブロックに例外ハンドラがない場合、そのブロックは終了させられ、呼び出し元または外側の BEGIN ブロックの例外ハンドラに例外が渡されます。そのブロックにも例外ハンドラがないときは、例外ハンドラが見つかるまで例外は先のほうへと渡されていきます。例外がクライアント・アプリケーションに到達するとエラーになります。

Note

SIGNAL 文は、「警告」として定義された SQLSTATE 値は扱いません。「エラー」として定義されたものだけを扱います。つまり、「8.3　例外ハンドラ」の「表 8-1　SQLSTATE 別の警告条件」にリストされたどの SQLSTATE 値も参照しない SIGNAL 文は無視されます。

例外ハンドラ内のWHEN句によって例外処理を行う場合、デフォルトでは、SQLSTATE値は'00000'にリセットされて例外は外に渡されません。WHEN句の中でRESIGNAL文を使用すると、この動作を上書きできます。WHEN句の実行は終了させられ、呼び出し元または外側のBEGINブロックに例外が渡されます。これは、例外を段階的に処理できることを意味します。たとえば、クライアント・アプリケーションへと例外を渡している間に内側のブロックでは問題に関する診断情報を収集できます。

以下の例は、SIGNALとRESIGNALがどのような働きをするかを示します。プロシージャp_error1はp_error2によって呼び出され、p_error2は外側のBEGINブロックによって呼び出されます。p_error1に制御が移ると、SIGNAL文が実行されてSQLSTATEの値がユーザ定義の例外'99001'に設定されます。p_error1には例外ハンドラがないため、呼び出し元のブロックであるp_error2の例外ハンドラに制御が移ります。p_error2のWHEN句によってRESIGNAL文が実行されると、外側のBEGINブロックへと制御が移ります。その途中、複数のMESSAGE文によって制御の流れがトレースされます。

```
CREATE PROCEDURE p_error1 ()
BEGIN
   DECLARE error_99001 EXCEPTION FOR SQLSTATE '99001';
   MESSAGE 'p_error1 before SIGNAL' TO CONSOLE;
   SIGNAL error_99001;
   MESSAGE 'p_error1 after SIGNAL' TO CONSOLE;
END;

CREATE PROCEDURE p_error2 ()
BEGIN
   DECLARE error_99001 EXCEPTION FOR SQLSTATE '99001';
   MESSAGE 'p_error2 before CALL p_error1' TO CONSOLE;
   CALL p_error1();
   MESSAGE 'p_error2 after CALL p_error1' TO CONSOLE;
EXCEPTION
   WHEN error_99001 THEN
      MESSAGE STRING ( 'p_error2 WHEN error_99001' ) TO CONSOLE;
      RESIGNAL;
END;

BEGIN
   DECLARE error_99001 EXCEPTION FOR SQLSTATE '99001';
   MESSAGE 'Before CALL p_error2' TO CONSOLE;
   CALL p_error2();
   MESSAGE 'After CALL p_error2' TO CONSOLE;
EXCEPTION
   WHEN error_99001 THEN
      MESSAGE STRING ( 'Outer WHEN error_99001' ) TO CONSOLE;
END;
```

以下に、このときの出力を示します。いったん例外が発生すると通常の制御の流れが中断されるため、すべてのMESSAGE文が実行されるわけではありません。

```
Before CALL p_error2
p_error2 before CALL p_error1
p_error1 before SIGNAL
p_error2 WHEN error_99001
Outer WHEN error_99001
```

BEGINブロックと例外ハンドラの詳細については、「第8章　パッケージ」の8.2と8.3項を参照してください。

以下に示すのは、Javaプログラムから呼び出されるストアド・プロシージャの例です。このストアド・プロシージャは、結果セットを返した後、ユーザ定義の例外を発生させるためにSIGNAL文を実行します。

```
CREATE PROCEDURE p1()
RESULT (
   key_1 INTEGER,
   non_key_1 VARCHAR ( 100 ) )
BEGIN
   DECLARE error_99001 EXCEPTION FOR SQLSTATE '99001';
   MESSAGE 'p1 before SELECT';
   SELECT key_1,
          non_key_1
     FROM t1
    ORDER BY key_1;
   MESSAGE 'p1 after SELECT';
   SIGNAL error_99001;
   MESSAGE 'p1 after SIGNAL';
END;
```

以下に示すのは、ストアド・プロシージャp1を呼び出すCatchError.javaアプリケーションです。

```
import java.sql.*;
public class CatchError {
  public static void main( String args[] ) {
    try {
      String driver = "ianywhere.ml.jdbcodbc.IDriver";
      Class.forName ( driver );
      String url = "jdbc:odbc:driver=Adaptive Server Anywhere 9.0;"
                 + "ENG=test9;DBN=test9;UID=DBA;PWD=SQL";
      Connection conn = DriverManager.getConnection ( url );
      String sql =  "CALL p1()" ;
      Statement stmtSql = conn.createStatement();
      stmtSql.execute ( sql );
```

```
      ResultSet result = stmtSql.getResultSet();
      while ( result.next() ) {
        Integer key_1 = new Integer ( result.getInt ( 1 ) );
        String non_key_1 = result.getString ( 2 );
        System.out.println (
           "key_1 = "
             + key_1.toString()
             + ", non_key_1 = '"
             + non_key_1
             + "'" );
      } // while
      Statement stmtMsg = conn.createStatement();
      String message =  "MESSAGE 'CatchError.java after display'" ;
      stmtMsg.execute ( message );
      stmtSql.getMoreResults();
    } // try
    catch ( SQLException e ) {
      Integer errorCode;
      errorCode = new Integer ( e.getErrorCode() );
      System.out.println ( "SQLState:  " + e.getSQLState() );
      System.out.println ( "ErrorCode: " + errorCode.toString()  );
      System.out.println ( "Message:   " + e.getMessage() );
      System.out.println ( "Stack trace..." );
      e.printStackTrace();
    }
    catch ( Exception e ) {
      System.out.println ( "Error: " + e.getMessage() );
      e.printStackTrace();
    }
  } // main
} // class CatchError
```

このCatchError.javaアプリケーションでは、SQL Anywhereに付属している、パフォーマンスに優れた新しいiAnywhere JDBCドライバを使用しています。データベースに接続した後、このプログラムはjava.sql.Statement.execute()メソッドを介してストアド・プロシージャp1を呼び出します。次に、結果セットのローを表示し、SQLのMESSAGE文を実行してデータベースのコンソール・ウィンドウに進行状況を記録します。最後に、getMoreResults()メソッドを呼び出すことによって、ストアド・プロシージャp1にSELECT文以降のSQL文を実行させます。この例では、getMoreResultsの呼び出しが非常に重要です。SELECTの後にSIGNAL文が続いているからです。getMoreResults()を呼び出さないかぎり、このSIGNAL文は実行されません。このアプリケーションにはcatchブロックが2つ含まれています。1つはSQLエラーを取得するもので、もう1つはそれ以外のエラーを取得します。

以下のWindowsバッチ・ファイルは、Javaアプリケーションのコンパイルと実行を行うものです。Sun JDKバージョン1.4.2_03のコンパイラjavac.exeとランタイム・ユーティリティjava.exeが使われています。また、CLASSPATHにはiAnywhere JDBCドライバjobdc.jarのファイル以外に現在のフォルダが設定されています。

```
C:¥j2sdk1.4.2_03¥bin¥javac.exe CatchError.java
SET CLASSPATH=.;%ASANY9%¥java¥jodbc.jar
C:¥j2sdk1.4.2_03¥bin¥java.exe CatchError
```

以下の出力は、先ほどのJavaアプリケーションによるものです。結果セットの後に最初のcatchブロックからの出力が続いています。ユーザ定義によるSQLSTATE値'99001'が、「ドライバ固有のエラー」を意味するHY000というSQLStateに変わっていること、SQLのErrorCodeの値が「ユーザ定義エラー」を意味する-297になっていることに注意してください。

```
key_1 = 1, non_key_1 = 'AAA'
key_1 = 2, non_key_1 = 'BBB'
SQLState:  HY000
ErrorCode: -297
Message:   [Sybase][ODBC Driver][Adaptive Server Anywhere]ユーザ定義の例外が通知されました。
Stack trace...
java.sql.SQLException: [Sybase][ODBC Driver][Adaptive Server Anywhere]ユーザ定義の例外が通知さ
れました。
        at ianywhere.ml.jdbcodbc.IStatement.getMoreResults(Native Method)
        at CatchError.main(CatchError.java:27)
```

以下の出力は、データベースのコンソール・ウィンドウに表示されたものです。"CatchError.java after display"というJavaアプリケーション・メッセージの後に、ストアド・プロシージャによる"p1 after SELECT"というメッセージが出力されていることに注意してください。これは、SELECTの後にSIGNAL文があるため、ストアド・プロシージャのSIGNAL文を含む部分に到達するにはgetMoreResultsの呼び出しが必要であることを示しています。

```
p1 before SELECT
CatchError.java after display
p1 after SELECT
```

9.5.2 RAISERRORとCREATE MESSAGE

SIGNALとRESIGNAL文に比べると、RAISERRORはとても行儀の良いSQL文です。SIGNALは不作法に処理を中断してすぐに例外ハンドラに制御を渡しますが、RAISERRORはただ診断情報の記録を行うと、次の文に進みます。以降のSQL文では、この診断情報を確認し、情報があればアクションを実行することになりますが、こうした処理は完全にオプションです。ほとんどの場合はこの情報がクライアント・アプリケーションに渡されるだけです。

RAISERROR文には4つのフォーマットがあります。

```
<raiserror> ::= RAISERROR <message_number>
              | RAISERROR <message_number> <message_text>
```

```
              | RAISERROR <message_number> <message_text> "," <substitution_list>
              | RAISERROR <high_message_number> "," <substitution_list>
<message_number> ::= 17000～99999の範囲の整数リテラル
                   | 同じ範囲の整数値をとる変数の<identifier>
<message_text>   ::= <simple_expression> -- プレースホルダ%nn!を含めることが可能
<simple_expression>   ::=「第3章　選択」の<simple_expression>を参照
                          -- サブクエリではない。IFまたはCASEでは始まらない
<substitution_list>   ::= { <substitution_value> "," } <substitution_value>
<substitution_value>  ::= <simple_expression>
<high_message_number> ::= 20000～99999の範囲の整数リテラル
                        | 同じ範囲の整数値をとる変数の<identifier>
```

RAISERRORの最初の形式は、17000～99999の範囲のメッセージ番号を指定するだけのものです。この番号には整数のリテラルまたは変数が使えます。ただし、式は使えません。次は、簡単なRAISERROR文の例です。

```
RAISERROR 17000;
```

図9-1は、ISQLでRAISERRORを実行した時に表示されるダイアログ・ボックスです。「RAISERRORが実行されました：」というメッセージが表示されていますが、それ以外のメッセージ・テキストは含まれていません。

図9-1　ISQLにおけるRAISERRORダイアログ・ボックス

メッセージ番号が20000～99999の範囲内にある場合、CREATE MESSAGE文を使用して、RAISERRORによって自動的に表示されるテキスト・メッセージを前もって格納しておくことができます。CREATE_MESSAGE文は、SYSUSERMESSAGESというSQL Anywhereカタログ・テーブルにローを1つ挿入します。

```
<create_message>  ::= CREATE MESSAGE <high_message_number> AS <message_literal>
<message_literal> ::= 255バイト長までの文字列リテラル
```

次は、CREATE MESSAGE 文の例で、メッセージ・テキストの格納場所を表示する SELECT が後に続いています。

```
CREATE MESSAGE 20000 AS 'Demonstration message.';
SELECT *
  FROM SYSUSERMESSAGES
 WHERE error = 20000;
```

この結果、SYSUSERMESSAGES 内にできたローの内容を以下に示します。

```
error  uid  description              langid
=====  ===  =======================  ======
20000  1    'Demonstration message.'  0
```

Tip

ISQL を使って RAISERROR 文を含む SQL スクリプトをテストする場合、特別な ISQL オプション ON_ERROR を一時的に変更するには、SET TEMPORARY OPTION ON_ERROR = 'CONTINUE' という文を使用します。ただし、実際のエラーを見逃してしまう可能性があるため、その変更を永続的なものにすることは避けます。

以下の例は、SYSUSERMESSAGES に格納されたメッセージ・テキストを表示する RAISERROR 文です。特別な ISQL オプションである ON_ERROR を一時的に 'CONTINUE' に設定すれば、RAISERROR の実行後も、ISQL はダイアログ・ボックスを表示させることなく実行を続けます。

```
SET TEMPORARY OPTION ON_ERROR = 'CONTINUE';
RAISERROR 20000;
```

RAISERROR 文が実行されると、［ISQL 結果］ウィンドウ枠の［メッセージ］タブに次のように表示されます。「Demonstration message」というメッセージ・テキストが自動的に表示されていることに注意してください。

```
Line 1, column 1
Could not execute statement.
[Sybase][ODBC Driver][Adaptive Server Anywhere]
RAISERROR が実行されました：  Demonstration message.
（エラーの後に続く）
```

RAISERROR 文の 2 番目の形式は、メッセージ番号の他にメッセージ・テキストも明示的に指定します。この場合、SYSUSERMESSAGES テーブルに格納されていた内容は、指定されたメッセージ・テキストで上書きされます。次の例は、メッセージ番号は先ほどと同じですがメッセージ・テキストが異なります。メッセージ番号とメッセージ・テキストの間にカンマがないことに注意してください。

```
RAISERROR 20000 'Different message.';
```

今回 ISQL によって表示される内容は次のとおりです。

```
RAISERROR executed:  Different message.
```

RAISERROR 文の 3 番目の形式では、%n! という特別な番号付きプレースホルダを含むメッセージ・テキスト・パラメータを使用し、その後に、このプレースホルダに代入する値のカンマ区切りリストを記します。リストの最初の値はプレースホルダ %1! に、2 番目の値は %2! に、というようにそれぞれの値が各プレースホルダに代入されます。次の例にはプレースホルダが 2 つあります。メッセージ・テキストの後にカンマが必要なことに注意してください。

```
RAISERROR 20000 'Diagnostic number = %1!, string = "%2!".', 999, 'AAA';
```

ISQL 実行の結果、次のような出力が得られます。

```
RAISERROR executed:  Diagnostic number = 999, string = "AAA".
```

RAISERROR 文の 4 番目の形式では、メッセージ・テキストを省略して、代入する値のリストだけを与えます。この場合、プレースホルダ %n! を含んだメッセージを格納するには、CREATE MESSAGE 文を使用します。以下に示すのは、先ほどの例で使ったのと同じメッセージ・テキストを格納する CREATE MESSAGE と、先ほどとまったく同じ出力を生成する RAISERROR の例です。この形式の RAISERRROR では、メッセージ番号の後にカンマがあることに注意してください。

```
CREATE MESSAGE 20000 AS 'Diagnostic number = %1!, string = "%2!".';
RAISERROR 20000, 999, 'AAA';
```

SYSUSERMESSAGES に格納するメッセージ・テキストは、最大 255 バイトに制限されています。しかし、メッセージがアプリケーション・プログラムに表示されるか返されるときに、SQL Anywhere が格納するのは、プレフィックス「RAISERROR が実行されました：」の後の 233 バイト分の文字までです。この制限はプレースホルダへの代入がすべて終わった後に適用されます。

RAISERROR 文の実行後も、SQL の処理は通常どおり継続されます。メッセージ番号は他の SQL 文でも組み込みの接続レベルの変数 @@ERROR から参照でき、メッセージ・テキストは組み込み関数 ERRORMSG() を呼び出すことによって参照できます。BEGIN ブロックの例外ハンドラは実行されません。SQL コードによって RAISERROR 文が通知したエラーを処理する必要がある場合は、@@ERROR 変数を明示的にチェックします。

Tip

CONTINUE_AFTER_RAISERROR オプションのデフォルト設定は変更しないで 'ON' のままにしておきます。変更すると、BEGIN ブロックの内側にある RAISERROR 文がこの項の説明どおりに動作しなくなります。

以下の示すのは、2つのストアド・プロシージャと1つの BEGIN ブロックからなる例です。BEGIN ブロックは p_error2 を、p_error2 は p_error1 をそれぞれ呼び出し、p_error1 は RAISERROR 文を実行します。途中、@@ERROR の値がチェックされ、数々の MESSAGE 文によってデータベースのコンソール・ウィンドウに進行状況が表示されます。RAISERROR 文はメッセージ番号と代入値のための変数を使用して、実行時に得られるあらゆる診断情報を表示します。この構文では RAISERROR 文のメッセージ・テキストに式を使うこともできます。

```
CREATE PROCEDURE p_error1 ()
BEGIN
   DECLARE @error INTEGER;
   DECLARE @diagnostic_number INTEGER;
   DECLARE @diagnostic_string VARCHAR ( 10 );
   MESSAGE 'p_error1 before RAISERROR';
   SET @error = 20000;
   SET @diagnostic_number = 999;
   SET @diagnostic_string = 'AAA';
   RAISERROR @error
      'Diagnostic number = %1!, string = "%2!".',
      @diagnostic_number,
      @diagnostic_string;
   MESSAGE 'p_error1 after RAISERROR';
EXCEPTION
   WHEN OTHERS THEN
      MESSAGE STRING ( 'p_error1 exception handler' );
END;

CREATE PROCEDURE p_error2 ()
BEGIN
   MESSAGE 'p_error2 before CALL p_error1';
   CALL p_error1();
   MESSAGE 'p_error2 after CALL p_error1';
   IF @@ERROR <> 0 THEN
      MESSAGE 'Error detected in p_error2...';
      MESSAGE STRING ( '   @@ERROR    = ', @@ERROR );
      MESSAGE STRING ( '   ERRORMSG() = ', ERRORMSG() );
   END IF;
EXCEPTION
   WHEN OTHERS THEN
      MESSAGE STRING ( 'p_error2 exception handler' );
END;
```

```
BEGIN
   MESSAGE 'Before CALL p_error2';
   CALL p_error2();
   MESSAGE 'After CALL p_error2';
   IF @@ERROR <> 0 THEN
      MESSAGE 'Error detected in outer block...';
      MESSAGE STRING ( '   @@ERROR    = ', @@ERROR );
      MESSAGE STRING ( '   ERRORMSG() = ', ERRORMSG() );
   END IF;
EXCEPTION
   WHEN OTHERS THEN
      MESSAGE STRING ( 'Outer block exception handler' );
END;
```

上記の例から得られる出力は以下のようになります。それぞれのCALLとRAISERRORの後も実行が正常に継続されたこと、実行の途中で@@ERRORとERRORMSG()の値が保存されていたこと、例外ハンドラはまったく実行されなかったことに注意してください。

```
p_error2 before CALL p_error1
p_error1 before RAISERROR
p_error1 after RAISERROR
p_error2 after CALL p_error1
Error detected in p_error2...
   @@ERROR    = 20000
   ERRORMSG() = RAISERROR が実行されました :  Diagnostic number = 999, string = "AAA".
After CALL p_error2
Error detected in outer block...
   @@ERROR    = 20000
   ERRORMSG() = RAISERROR が実行されました :  Diagnostic number = 999, string = "AAA".
```

以下に示すのは、Javaアプリケーションから呼び出されるストアド・プロシージャの別の例です。アプリケーションは以前示したのと同じCatchError.javaですが、ストアド・プロシージャは以前と異なり、SIGNALの代わりにRAISERROR文を実行します。

```
CREATE PROCEDURE p1()
RESULT (
   key_1 INTEGER,
   non_key_1 VARCHAR ( 100 ) )
BEGIN
   DECLARE @error INTEGER;
   DECLARE @diagnostic_number INTEGER;
   DECLARE @diagnostic_string VARCHAR ( 10 );

   MESSAGE 'p1 before SELECT';
   SELECT key_1,
```

```
        non_key_1
    FROM t1
   ORDER BY key_1;
  MESSAGE 'p1 after SELECT';

  SET @error = 20000;
  SET @diagnostic_number = 999;
  SET @diagnostic_string = 'AAA';
  RAISERROR @error
     'Diagnostic number = %1!, string = "%2!".',
     @diagnostic_number,
     @diagnostic_string;
  MESSAGE 'p1 after RAISERROR';

END;
```

以下の出力は、新しいプロシージャ p1 がデータベースに格納されている場合に、以前示した Java アプリケーション CatchError.java によって表示されたものです。以前と同様、結果セットの後に最初の catch ブロックからの出力が続き、SQLState も「ドライバ固有のエラー」を意味する HY000 という値で前と同じです。ただし、今回は SQL の ErrorCode の値が -20000 になっています。これは RAISERRROR 文で指定された、負の値を持つメッセージ番号です。また、診断データを含む完全な形の RAISERROR メッセージが表示されています。このことは、クライアント・アプリケーションに情報を渡すという点では RAISERROR のほうが SIGNAL より強力なことを示しています。

```
key_1 = 1, non_key_1 = 'AAA'
key_1 = 2, non_key_1 = 'BBB'
SQLState:  HY000
ErrorCode: -20000
Message:   [Sybase][ODBC Driver][Adaptive Server Anywhere]RAISERROR が実行されました :
Diagnostic number = 999, string = "AAA".
Stack trace...
java.sql.SQLException: [Sybase][ODBC Driver][Adaptive Server Anywhere]RAISERROR が実行されました :  Diagnostic number = 999, string = "AAA".
        at ianywhere.ml.jdbcodbc.IStatement.getMoreResults(Native Method)
        at CatchError.main(CatchError.java:27)
```

以下の出力は、CatchError.java が新しいプロシージャ p1 を呼び出したときにデータベースのコンソール・ウィンドウに表示されたものです。SIGNAL 文を使った前のテストとは異なり、このプロシージャの実行は RAISERROR 文の実行後も続きました。しかし、SELECT 以降の文を実行するには、やはり getMoreResults() の呼び出しが必要です。

```
p1 before SELECT
CatchError.java after display
p1 after SELECT
p1 after RAISERROR
```

以下に示すのは、例外をRAISERRORメッセージに変えるためにBEGINブロックの内側で使用できる例外ハンドラの例です。このコードの利点は、クライアント・アプリケーションに渡されるエラー・メッセージの中に元のSQLCODE、SQLSTATE、ERRORMSG()の値がすべて保存されることです。

```
CREATE PROCEDURE p1()
RESULT (
   key_1 INTEGER,
   non_key_1 VARCHAR ( 100 ) )
BEGIN
   DECLARE @sqlcode   INTEGER;
   DECLARE @sqlstate  VARCHAR ( 5 );
   DECLARE @errormsg  VARCHAR ( 32767 );
   DECLARE error_99001 EXCEPTION FOR SQLSTATE '99001';
   SIGNAL error_99001;
EXCEPTION
   WHEN OTHERS THEN
      SELECT SQLCODE, SQLSTATE, ERRORMSG()
        INTO @sqlcode, @sqlstate, @errormsg;
      RAISERROR 99999
         'SQLCODE = %1!, SQLSTATE = %2!, ERRORMSG() = %3!',
         @sqlcode, @sqlstate, @errormsg;
END;
```

以下の出力は、上記のプロシージャを呼び出したときにJavaアプリケーションCatchError.javaによって表示されたものです。今回は、元のSQLSTATEの値'99001'をクライアント・アプリケーションが使用できます。

```
SQLState:  HY000
ErrorCode: -99999
Message:   [Sybase][ODBC Driver][Adaptive Server Anywhere]RAISERRORが実行されました：
SQLCODE =  -297, SQLSTATE = 99001, ERRORMSG() = ユーザ定義の例外が通知されました。
```

この方法は組み込みSQLSTATEの例外にも使えます。クライアントに返されるエラー・メッセージには、エラーの時点で利用可能な元の診断情報がすべて含まれます。以下の出力は、呼び出し時にデータ変換の例外を発生させるプロシージャの中で上記の例外ハンドラを使用したときにJavaアプリケーションCatchError.javaによって表示されたものです。

```
SQLState:  HY000
ErrorCode: -99999
Message:   [Sybase][ODBC Driver][Adaptive Server Anywhere]RAISERRORが実行されました：
SQLCODE = -157, SQLSTATE = 53018, ERRORMSG() = 値 ??? をデータ型 ??? に変換できません。
```

9.5.3 ROLLBACK TRIGGER

ROLLBACK TRIGGER 文をトリガ内部で使用すると、次の大きく異なる 2 つの操作のどちらかを実行できます。内側のトリガ操作を何の通知もなくロールバックしてから何も起こらなかったかのように処理を継続するか、外側のトリガ操作をロールバックしてクライアント・アプリケーションにエラーを返すかのどちらかです。以下に、この 2 つの操作のための構文を示します。最初の形式はロールバック時に何も通知しない方法（サイレント・ロールバック）を行い、2 番目の形式はエラーを返すものです。

```
<rollback_trigger> ::= ROLLBACK TRIGGER
                     | ROLLBACK TRIGGER WITH <raiserror>
```

以下に示すのは、ROLLBACK TRIGGER を使ったローカルなサイレント・ロールバックの例です。テーブル t1 にはトリガはありませんが、テーブル t2 にはテーブル t3 に影響を与える UPDATE 文を含む更新トリガが、テーブル t3 には ROLLBACK TRIGGER 文を含む更新トリガがあります。このトランザクションには、テーブル t1 と t2 を変更する 2 つの UPDATE 文があり、その後に COMMIT 文が続きます。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL PRIMARY KEY,
   non_key_1     VARCHAR ( 100 ) NOT NULL );

CREATE TABLE t2 (
   key_2         INTEGER NOT NULL PRIMARY KEY,
   non_key_2     VARCHAR ( 100 ) NOT NULL );

CREATE TABLE t3 (
   key_3         INTEGER NOT NULL PRIMARY KEY,
   non_key_3     VARCHAR ( 100 ) NOT NULL );

INSERT t1 VALUES ( 1, 'AAA' );
INSERT t2 VALUES ( 1, 'AAA' );
INSERT t3 VALUES ( 1, 'AAA' );
COMMIT;

CREATE TRIGGER tru_t2 BEFORE UPDATE ON t2
FOR EACH ROW
BEGIN
   UPDATE t3 SET non_key_3 = 'ZZZ' WHERE key_3 = 1;

END;

CREATE TRIGGER tru_t3 BEFORE UPDATE ON t3
FOR EACH ROW
BEGIN
```

```
   ROLLBACK TRIGGER;
END;

UPDATE t1 SET non_key_1 = 'XXX' WHERE key_1 = 1;
UPDATE t2 SET non_key_2 = 'YYY' WHERE key_2 = 1;
COMMIT;

SELECT * FROM t1;
SELECT * FROM t2;
SELECT * FROM t3;
```

Note

特に断りがないかぎり、本書のすべての例ではCHAINEDオプションがデフォルト値'ON'に設定されているものとします【監注12】。CHAINEDオプションの詳細については、「9.3　トランザクション」を参照してください。

上記のスクリプトを実行すると、ROLLBACK TRIGGERは直接トリガを起動した操作だけに影響するロールバックを行います。この例の場合はテーブルt3のUPDATEがロールバックされます。ただし、テーブルt2の更新は影響を受けません。ROLLBACK TRIGGERを含むトリガを直接起動しなかったためです。代わりにテーブルt2のトリガが起動され、その結果、テーブルt3のトリガが起動されます。また、テーブルt1の更新もROLLBACK TRIGGERの影響を受けません。次のように、3つのSELECT文は、テーブルt1とt2は更新され、t3は更新されなかったことを示しています。

```
key_1  non_key_1
=====  =========
1      'XXX'

key_2  non_key_2
=====  =========
1      'YYY'

key_3  non_key_3
=====  =========
1      'AAA'
```

ROLLBACK TRIGGERの2番目の形式は、RAISERROR文とROLLBACK TRIGGER文を結合したものです。以下に示すのは、先ほどの例にあったトリガの1つを変更して、ROLLBACK TRIGGERにWITH RAISERRORを加えたものです。

【監注12】Open ClientおよびJDBC接続時は、CHAINEDオプションのデフォルト値は'OFF'。

```
CREATE TRIGGER tru_t3 BEFORE UPDATE ON t3
FOR EACH ROW
BEGIN
   ROLLBACK TRIGGER WITH RAISERROR 99999 'ROLLBACK TRIGGER in tru_t3';
END;
```

以下のISQLテスト・スクリプトはこの新しいトリガの影響を示しています。ISQLオプションON_ERRORが'CONTINUE'に設定されているため、エラーの後もISQLは実行を継続します。また、UPDATE文がBEGINブロック内にあるため、ISQLはこれらのUPDATE文を、SQL Anywhereに渡される単一の実行可能バッチとして扱います。また、SQL文のうち、どれが実行され、どれがバイパスされたかを示すMESSAGE文が含まれています。

```
SET TEMPORARY OPTION ON_ERROR = 'CONTINUE';
BEGIN
   MESSAGE '1';
   UPDATE t1 SET non_key_1 = 'XXX' WHERE key_1 = 1;
   MESSAGE '2';
   UPDATE t2 SET non_key_2 = 'YYY' WHERE key_2 = 1;
   MESSAGE '3';
   COMMIT;
   MESSAGE '4';
END;
MESSAGE '5';

SELECT * FROM t1;
SELECT * FROM t2;
SELECT * FROM t3;
```

次のメッセージは、上記のスクリプトの実行時に［ISQLメッセージ］タブに表示されたものです。

```
RAISERRORが実行されました： ROLLBACK TRIGGER in tru_t3
```

以下の出力は、データベースのコンソール・ウィンドウに表示されたものです。2番目のUPDATE文がエラーになったため、COMMITが実行されなかったことがわかります。このエラーの発生により、SQL AnywhereはBEGINブロック内の残りの文をバイパスし、クライアントであるISQLに制御を返しています。

```
1
2
5
```

先ほどの例とは異なり、テーブルt3の変更と同様にテーブルt2の変更もロールバックされていることが、以下に示す3つのSELECT文の出力からわかります。ROLLBACK TRIGGERのこの形式は、

ネストされたトリガ以外に外側のトリガを起動した変更もロールバックします。ただし、テーブル t1 の変更は有効なままです。ROLLBACK TRIGGER のどちらの形式も、部分的なロールバックを行うため、t1 の変更はロールバック対象に含まれなかったのです。

```
key_1  non_key_1
=====  =========
1      'XXX'

key_2  non_key_2
=====  =========
1      'AAA'

key_3  non_key_3
=====  =========
1      'AAA'
```

上記スクリプトの実行終了時もトランザクションはまだ有効であることに対する注意が重要です。COMMIT または ROLLBACK のどちらを実行するかはクライアント（ISQL）次第です。t2 と t3 の変更がロールバックされてもテーブル t1 の変更はコミットされていない状態のままです。

ROLLBACK TRIGGER 文の 2 つの形式は、以前、SQL Anywhere には「ネストされたトランザクションは存在しない」と述べたことと一部矛盾します。厳密に言うと、存在しないのは部分的なコミットの類です。ネストされたトランザクションについて語られるとき、たいていは部分的なコミットに言及していますが、この部分的なコミットはサポートされていません。ただし、限られた形式の部分的ロールバックは ROLLBACK TRIGGER と、「9.4　セーブポイントとサブトランザクション」で説明したもっと柔軟な機能によってサポートされています。

Section

9.6 ロック

全体的な効率性を改善するために、マルチユーザ環境では異なるトランザクションどうしを重複させることができます。たとえば、UPDATE の処理を行った SQL Anywhere が同じトランザクション内の次のコマンドを受け取るのを待っている間に、別のトランザクション内では SELECT に到達した場合、すぐに SQL Anywhere はその SELECT を処理しようとします。仮に、SQL Anywhere が一度に 1 つのトランザクションの処理しか行わないとすると、他の処理を割り込ませることはできません。しかし、実際には、データベース・エンジンは混雑した環境において重複する数百のトランザクションの間で実行を切り替えることができます。

SQL Anywhere がトランザクションを重複して処理できることを「同時実行性」といいます。同時実行性は、トランザクションの基本的要件である一貫性や独立性と矛盾することがあります。たとえば、

重複する2つのトランザクションに同じローの更新を許した場合、異なるトランザクションによる変更は互いに独立していなければならないという要件に違反します。別の例として、最終的な結果に一貫性を持たせるために検索と更新の間でデータに変更が起こらないよう要求されるトランザクションの設計をとりあげます。最初のトランザクションで取り出されたデータを、重複する別のトランザクションが変更すると、たとえ最初のトランザクションによる更新前に別のトランザクションが変更をコミットした場合であっても、この要件に違反します。

SQL Anywhereは、ロックを使用して、同時実行性を許すと同時に独立性と一貫性を保持します。「ロック」とは、SQL Anywhereによって管理されるシステム内部テーブルに格納されるデータです。各ロックは、特定の接続による処理の進行が可能になる前に満たされなければならない要件を表し、論理的にはその接続と単一のローまたはテーブルの間の一時的な関係として実装されます。ロックが存在する間は、他のいかなる接続もそのテーブルまたはローに対して一定の操作を行えません。

ある接続が処理を進めるためにロックを必要とするとき、接続からロックが要求された、といいます。そのロックがSQL Anywhereによって作成されると、「要求が承諾された」「ロックを取得した」といい、先ほどの接続は処理を進めることができます。他の競合するロックがすでに存在するためにSQL Anywhereによってロックが作成されないとき、「要求がブロックされた」「ロックが取得できなかった」といい、先ほどの接続は処理を進めることができません。

ロックは大きく短期と長期の2つに分類されます。「short-term lock（短期ロック）」は単一のSQL文の間またはそれ以下の間しか保持されません。一方、「long-term lock（長期ロック）」はそれより長い間、たいていは1つのトランザクションの終了まで保持されます。この章では、長期ロックに限定して説明を進めます。管理の観点からは短期ロックは見えないものだからです。この章では、特に断りがないかぎり、「ロック」という用語は「長期ロック」を意味します。

組み込みプロシージャsa_locksを使用すると、その時点で保持されているすべてのロックを表示できます。sa_locksを呼び出す例を次に示します。

```
CALL sa_locks();
```

以下に示すのはsa_locksの実行結果です。各エントリは、特定のテーブルまたはローに対応付けられた1つ以上のロックを表します。connectionのカラムは、ロックを保持している接続を識別するものです。user_idのカラムは、接続の作成に使用されたユーザidを表します。table_nameは、ロックが対応付けられているテーブルを表します。lock_typeは、そのエントリによって表されるロックの種類を表します。lock_nameのカラムには、テーブル全体に対応付けられているsa_locksのエントリを表す内部のロー識別子またはNULLが記されています。

```
connection  user_id  table_name  lock_type  lock_name
==========  =======  ==========  =========  =========
508116521   DBA      DBA.t1      E          473
508116521   DBA      DBA.t3b     EPA*       4294967836
508116521   DBA      DBA.t1b     EPA0000    4294967834
```

```
508116521   DBA     DBA.t1u     EPA0001    12884902403
508116521   DBA     DBA.t1n     EPT        528
508116521   DBA     DBA.t3      S          4294967821
508116521   DBA     DBA.t1      SPA0000    1095216660986
508116521   DBA     DBA.t1u     SPA0001    1095216661028
508116521   DBA     DBA.t3n     SPT        553
508116521   DBA     DBA.e4b     E          NULL
508116521   DBA     DBA.e4      EPT        NULL
508116521   DBA     DBA.t2n     S          NULL
508116521   DBA     DBA.e1b     SAT        NULL
508116521   DBA     DBA.e3      SPAT       NULL
508116521   DBA     DBA.t2b     SPT        NULL
```

以下では、lock_name カラムの値が NULL でない場合、lock_type カラムの値がどのような意味を持つかを説明します。

- "E" は「exclusive row write lock（排他ロー書き込みロック）」を表す。同じローに対する排他書き込みまたは共有読み込みのロックを、他の接続のどれかが持っている場合、この種類のロックは付与されない。排他ロー書き込みロックがいったん取得されると、他の接続は、このローに対するどのような種類のロックも取得できなくなる
- "S" は「shared row read lock（共有ロー読み込みロック）」を表す。この種類のロックは、他の接続に付与された、同じローに対する別の共有ロー読み込みロックと共存することができる
- "P" は「insert row position lock（挿入ロー位置ロック）」または「anti-phantom row position lock（対幻ロー位置ロック）」を表し、lock_name カラムによって識別されるローの直前の位置にローを挿入する権利を予約する。このローの位置は、特定のインデックス順、逐次テーブル・スキャン順、またはテーブル上のインデックスと逐次のすべての順序付け、という 3 つの方法のどれかによって決まる。挿入ロー位置ロックと同時に、排他ロー書き込みロックまたは共有ロー読み込みロックが常に付与される
- "A" は「anti-insert row position lock（対挿入ロー位置ロック）」または「phantom row position lock（幻ロー位置ロック）を表し、lock_name カラムで識別されるローの直前の位置に他の接続によってローが挿入されるのを防ぐ。このローの位置も挿入ロックの場合と同じように決まる。対挿入ロー位置ロックと同時に、排他ロー書き込みロックまたは共有ロー読み込みロックが常に付与される。また、対挿入ロックと挿入ロックは同時に付与されることがある。たとえば、"EPA" や "SPA" の組み合わせは、同じローに対応付けられた 3 つのロックが、sa_locks 出力において 1 つのエントリで表されることを意味する
- 0000 または 0001 のような 4 桁の整数は、挿入ロー位置ロックと対挿入ロー位置ロックのためのローの順序付けを決めるために使用されるインデックスを識別するためのものである
- "T" は、逐次テーブルを使用して挿入ロー位置ロックと対挿入ロー位置ロックのためのローの順序付けを決めることを表す
- アスタリスク（*）は、挿入ロックと対挿入ロックをインデックスと逐次のすべての順序に適用する

ことを表す

以下では、lock_name カラムの値が NULL の場合、lock_type カラムの値がどのような意味を持つかを説明します。

- "E" は、排他テーブル・スキーマ・ロックを表す
- "S" は、共有テーブル・スキーマ・ロックを表す
- "PT" は、テーブル・コンテンツ・更新インテントロックを表す
- "AT" は、テーブル・コンテンツ読み込みロックを表す
- "PAT" は、更新インテントと読み込みの 2 つのテーブル・コンテンツ・ロックの組み合わせを表す

以下に、先ほどの sa_locks の実行結果における lock_type と lock_name のすべての組み合わせを、上記の定義によるロックの説明と共に示します。

表 9-2　lock_type と lock_name の組み合わせ

lock_type	lock_name	説明
E	473	排他ロー書き込みロック
EPA*	4294967836	排他ロー書き込みロック、すべての順序での挿入ロー位置ロックと対挿入ロー位置ロック
EPA0000	4294967834	排他ロー書き込みロック、インデックス 0000 での挿入ロー位置ロックと対挿入ロー位置ロック
EPA0001	12884902403	排他ロー書き込みロック、インデックス 0001 での挿入ロー位置ロックと対挿入ロー位置ロック
EPT	528	排他ロー書き込みロック、逐次順序での対挿入ロー位置ロック
S	4294967821	共有ロー・ロック
SPA0000	1095216660986	排他ロー読み込みロック、インデックス 0000 での挿入ロー位置ロックと対挿入ロー位置ロック
SPA0001	1095216661028	排他ロー読み込みロック、インデックス 0001 での挿入ロー位置ロックと対挿入ロー位置ロック
SPT	553	排他ローの読み込みロック、逐次順序での対挿入ロー位置ロック
E	(NULL)	排他テーブル・スキーマ・ロック
EPT	(NULL)	排他テーブル・スキーマ・ロック、テーブル・コンテンツ・更新インテントロック
S	(NULL)	共有テーブル・スキーマ・ロック
SAT	(NULL)	共有テーブル・スキーマ・ロック、テーブル・コンテンツ・読み込みロック
SPAT	(NULL)	共有テーブル・スキーマ・ロック、テーブル・コンテンツ・読み込みロックとテーブル・コンテンツ・更新インテントロック
SPT	(NULL)	排他テーブル・スキーマ・ロック、テーブル・コンテンツ・更新インテントロック

単一接続の場合は、同じテーブルまたはローに対する異なる種類のロックの取得は妨げられません。競合は異なる接続どうしの間でしか起こらないからです。たとえば、ある接続があるローの位置に対する対挿入ロックを持っている場合、別の接続は同じローの位置に対する挿入ロックを取得できません。しかし、単一接続の場合は同じ位置に対してどちらの種類のロックも取得できます。

接続がもはやロックを必要としなくなったとき、ロックは解放されたといい、SQL Anywhere は内部のロック・テーブルからそのエントリを削除します。ほとんどのロックは、ある接続によって取得されると、その接続が次にCOMMIT またはROLLBACK の操作を行うまで持続します。しかし、ロックによっては、それ以前に解放されるものやそれ以降も持続するものがあります。たとえば、独立性レベル1でカーソル安定性を確保するために、FETCH 操作によって取得された読み込みロックは、次のローがフェッチされるとすぐ解放されます。また、WITH HOLD 句を使ったLOCK TABLE 文によって取得された排他テーブル・ロックはCOMMIT の後も持続します。実際、そのテーブルの削除と再作成が行われると、そのテーブル・ロックは自動的に復活し、接続が切断されるまで解放されません。カーソル安定性については次の項で説明し、そこではLOCK TABLE 文によって可能になるパフォーマンスの向上についても説明します。

しかし、実用的な目的であれば、トランザクション中に取得されるすべてのロー・ロックは、そのトランザクションがCOMMIT またはROLLBACK によって終了するまで保持され、その終了時にまとめて解放されます。これは、成功する文だけでなく失敗する文にもあてはまります。INSERT、UPDATE、DELETE のような単一のSQL 文は、事実上アトミックです。このことは、こうした文が失敗すると、その文によるデータベースの変更のすべてが自動的に取り消されることを意味します。しかし、これはロックには適用されません。失敗した文によって取得されたロックは、トランザクションが終了しても持続します。

Section

9.7 ブロックと独立性レベル

付与できないロックを接続が要求すると、「ブロック」が起こります。デフォルトでは、ブロックされた接続はすべての競合するロックが解放されるまで待機させられます。データベース・オプションBLOCKING を 'OFF' に設定すると、ブロックされた操作はすぐにキャンセルされ、ブロックされた接続にはエラーが返されます。しかし、ブロックされた操作のキャンセルは自動的なロールバックを意味するものではありません。影響を受けた接続は処理を進め、失敗した文のそれまでの処理の間に取得されたロックを含め、以前に取得していたロックをそのまま保持する場合があります。

1つの接続が一度に保持するロックの数には、0から数百万までの幅があります。実際の数は、2つの大きな要因によって左右されます。それは、現在のトランザクションの間に実行されたSQL 操作の種類と、各操作が実行されたときのISOLATION_ LEVEL データベース・オプションの設定です。操作によっては、UPDATE のように独立性レベルに関係なくロックを要求するものがあります。SELECT のよう

な他の操作では、独立性レベルによってロックを要求するかしないかが決まります。

「独立性レベル」の値は0、1、2、または3のどれかであり、その接続が他の接続によって実行される操作から保護される度合いを表します。

- 独立性レベル0は、重複するデータ変更、スキーマ変更と重複するデータ検索、デッドロック状況を回避する。図9-2から9-5と図9-20では、独立性レベル0の場合に重複するトランザクションがどのような影響を受けるかを示す
- 独立性レベル1は、独立性レベル0が提供する保護に加え、ダーティ・リードとカーソル不安定性を回避する。図9-6から9-9で、独立性レベル1の影響を示す
- 独立性レベル2は、独立性レベル0と1の提供する保護に加え、繰り返し不可能読み出しと更新不安定性を回避する。図9-10から9-13では、独立性レベル2において繰り返し可能読み出しと更新安定性がどのように達成されるかを示す
- 独立性レベル3は、独立性レベル0、1、2の提供する保護に加え、幻ローと特定の形式の更新内容消失を回避する。図9-14から9-17では、独立性レベル3の影響を示す

独立性レベル2と3では、ロック数が最大に、保護レベルが最高になりますが、同時実行性のレベルは最低になります。図9-18と9-19では、高い独立性レベルが同時実行性にどのように影響するかを示します。

9.7.1 独立性レベル0

デフォルトでは独立性レベルは0です。この場合、ロックの数は最小に、同時実行性は最高レベルになりますが、一貫性が失われます。なお、独立性レベルを高くすれば一貫性が失われるのを防げます。

図9-2は、ロックとブロックを説明するいくつかの例のうちで最初のものです。こうした例のすべてには、2つの接続、1つのテーブル、独立性レベルのいろいろな値が関係しています。以下のスクリプトは、テーブルを作成して5つのローに値を入れるものです。このスクリプトが図9-2から9-20を説明するときの開始点になります。

```
CREATE TABLE DBA.t1 (
   k1     INTEGER NOT NULL PRIMARY KEY,
   c1     VARCHAR ( 100 ) NOT NULL );

INSERT t1 VALUES ( 1, 'clean' );
INSERT t1 VALUES ( 3, 'clean' );
INSERT t1 VALUES ( 5, 'clean' );
INSERT t1 VALUES ( 7, 'clean' );
INSERT t1 VALUES ( 9, 'clean' );
COMMIT;
```

図9-2は、接続Aがローを更新した後COMMITまたはROLLBACKを行う前に、接続Bが同じローの更新と削除をしようとするとどうなるかを示しています。接続Aはこのローに対する排他書き込みロックを保持しているため、接続Bによる操作はどちらもブロックされます。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 0				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 0			
3	UPDATE t1 SET c1 = 'winner' WHERE k1 = 5				A Write (E) A Schema (S)
4		UPDATE t1 SET c1 = 'loser' WHERE k1 = 5	Blocked: 42W18 User 'A' has the row in 't1' locked		A Write (E) A Schema (S) B Schema (S)
5		DELETE t1 WHERE k1 = 5	Blocked: 42W18 User 'A' has the row in 't1' locked		A Write (E) A Schema (S) B Schema (S)

図9-2　UPDATEによる別のUPDATEとDELETEのブロック

以下に、図9-2以降の図に表示される6つのカラムを説明します。

- **ステップ番号1、2、3...**は、2つの接続のどちらかで個別に実行された各SQLコマンドの実行順序を表す。それぞれの図のステップ1と2は、各接続のISOLATION_LEVELに明示的に設定される値を示す。図9-2の場合、独立性レベルは問題にならない。UPDATEは常にUPDATEまたはDELETEをブロックするからである
- **Connection A（接続A）**のカラムは、一方の接続で実行される各SQL文を表す
- **Connection B（接続B）**のカラムは、もう一方の接続で実行されるSQL文を表す
- **Comment（コメント）**のカラムは、そのステップの完了時に起こる興味深い状況を説明する。図9-2の場合、ステップ4と5で接続BによるUPDATEとDELETEの実行がブロックされることを示している。これらの図のうち1つを除くすべての図では、その目的上、どちらの接続もBLOCKINGオプションが'OFF'に設定されているために待機は起こらない。ブロックされた文はすぐにキャンセルされ、SQLSTATEにはエラーを示す'42W18'が設定される。ロールバックもどのようなロックの解放もブロックによっては起こらないことに注意が必要である
- **c1 value（c1の値のカラム）**は、特定のローに対するSELECTまたはFETCHを行うステップにおけるt1.c1カラムの値を示す。この値は図9-3以降で重要になる
- **Locks Held by A & B（AとBが保持するロック）**のカラムは、各ステップの実行後に接続AとBが保持しているすべてのロックを表す。このカラムに示されるのは、この時点で存在するロックであって、必ずしもこのステップで取得されたロックとはかぎらない。たとえば、ステップ3で初めて表示される書き込みロックは、ステップ3で取得されてステップ4と5でも持続する。各ロックの説明の前にある文字AまたはBは、どちらの接続がそのロックを保持しているかを表す

ロックの説明は、AとBが保持するロックのカラムに簡潔に記しました。これらの図の目的は、ロック、ブロック、独立性レベルが同時実行性と一貫性にどのように影響するかを説明することであって、SQL Anywhereにおけるロック管理の内部の働きを説明することではないからです。以下は、簡潔化された記述のリストと、それらの意味を9.6項における定義の点から説明したものです。

- **Write［E］（書き込み（排他））**は、排他ロー書き込みロックを表す
- **Read［S］（読み込み（共有））**は、共有ロー読み込みロックを表す
- **Anti-insert［S］（対挿入（共有））**は、共有ロー読み込みロックと対挿入ロー位置ロックの組み合わせを表す
- **Anti-insert + Insert［S］（対挿入＋挿入（共有））**は、共有ロー読み込みロックと対挿入ロー位置ロックに挿入ロー位置ロックを加えた組み合わせを表す
- **Schema［S］（スキーマ（共有））**は、共有テーブル・スキーマ・ロックを意味し、これにはテーブル・コンテンツ・更新インテントロックを備えたものとそうでないものがある

Note

図9-2から9-20では、連鎖モードを想定して、トランザクションの開始と終了を明示的に示していません。連鎖モードについては「9.3　トランザクション」で説明しています。連鎖モードでは、接続Aと接続Bの両カラムに記される、最初のINSERT、UPDATE、またはDELETE文、あるいは、ロックを取得するSELECT文によってトランザクションが暗黙的に開始されます。これらのトランザクションは、明示的なCOMMITまたはROLLBACK文の実行によって終了します。

図9-3は、接続Aによって削除されたローは、接続Aによるその変更のコミットの前に、接続Bによって再挿入できないことを示します。これは独立性レベルに関係なくあてはまります。接続Aはこの削除のロールバックを実行できなければなりません。そのため、事実上は接続Aによるそのロー自体の再挿入が可能でなければなりません。仮に、接続Bによるこのローの再挿入が認められると、接続Aのロールバックによってプライマリ・キーの競合が起こることになります。この場合、接続Bの挿入がブロックされます。接続Aは、他の接続によるこのローの再挿入を防ぐための対挿入ロックのほか、このローに対する書き込みロックを保持しているからです。また、接続Aは、ロールバック時にそのローを再挿入できるように挿入ロックも保持しています。待機を行うか、後でこの挿入を再び試みるかは接続Bの自由です。接続Aが削除による変更をコミットすると、接続Bはこのローを挿入できますが、接続Aがこの削除をロールバックすると、接続Bの挿入は失敗します。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 0				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 0			
3	DELETE t1 WHERE k1 = 5				A Write (E) A Anti-insert + Insert (S) A Schema (S)
4		INSERT t1 VALUES (5, 'reinserted')	Blocked: 42W18 User 'A' has the row in 't1' locked		A Write (E) A Anti-insert + Insert (S) A Schema (S) B Schema (S)

図 9-3　DELETE による INSERT のブロック

図 9-3 に示したシナリオは、テーブル t1 のプライマリ・キーの存在によって変わってきます。仮にプライマリ・キーが存在しなければ、ステップ 3 において接続 A は対挿入ロックと挿入ロックを取得できず、ステップ 4 におけるブロックは起こりません。そのため、接続 B はこのローを挿入できたことになります。

図 9-4 は、ローの挿入を行った接続 A がその変更をコミットするまでは、接続 B は、独立性レベルに関係なく、そのローの更新も削除も行えないことを示します。接続 A はコミットまたはロールバックを行うまでは新しいローを完全に制御しています。つまり、接続 A は、それまで干渉を起こすことなく、そのローに対する他の操作を自由に行えなければなりません。接続 B による更新または削除は、確実にこの干渉のカテゴリに分類されます。図 9-3 の場合、接続 B は待機するか、後で再び操作を試みるかのどちらかを自由に行えます。接続 A がコミットを行った場合、以降の更新と削除の操作が有効になります。接続 A が挿入をロールバックした場合、接続 B は更新も削除も行えなくなります。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 0				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 0			
3	INSERT t1 VALUES (4, 'new')				A Write (E) A Schema (S)
4		UPDATE t1 SET c1 = 'different' WHERE k1 = 4	Blocked: 42W18 User 'A' has the row in 't1' locked		A Write (E) A Schema (S) B Schema (S)
5		DELETE t1 WHERE k1 = 4	Blocked: 42W18 User 'A' has the row in 't1' locked		A Write (E) A Schema (S) B Schema (S)

図 9-4　INSERT による UPDATE、DELETE のブロック

図 9-5 は、独立性レベルが 0 であってもテーブルのスキーマ・ロックを取得する簡単な SELECT を示

します。これらのロックは、スキーマの変更を防ぐ以外は他のどの接続にも影響を与えません。この例では、接続 A の SELECT によって接続 B はインデックスの作成ができなくなります。独立性レベル 0 で実行されるアプリケーションは、ローを取り出した後のコミットをめったに行いません。これは、混雑した環境では、ほとんどのテーブルが永続的なスキーマ・ロックの影響を受けてスキーマの変更が困難になることを意味します。これと反対の影響はさらに大きなものになります。いったんスキーマの変更が開始されると、他の接続はこのスキーマの変更が完了するまで影響を受けたテーブルに対して何もできなくなります。重要な時期におけるスキーマの変更はおすすめしません。また、本書では、スキーマの変更によって生じるロックとブロックについてこれ以上は説明しません。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 0				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 0			
3	SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5			clean	A Schema (S)
4		CREATE INDEX xc1 ON t1 (c1)	Blocked: 42W18 User 'A' has the row in 't1' locked		A Schema (S)
5	COMMIT				
6		CREATE INDEX xc1 ON t1 (c1)	Unblocked		

図 9-5　SELECT によるスキーマ変更のブロック

9.7.2　独立性レベル1

図 9-6 は、独立性レベル 0 では許可される同時実行動作の最初の例として「ダーティ・リード」をとりあげます。ステップ 3 で接続 A が更新したローは、ステップ 4 で接続 B によってすぐに読み込まれています。これを「ダーティ・リード」といいます。接続 A による変更はまだコミットされていないためです。最終的にこの変更がロールバックされると、接続 B はステップ 4 においてダーティ・データを使って作業を行うことになります。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 0				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 0			
3	UPDATE t1 SET c1 = 'dirty' WHERE k1 = 5				A Write (E) A Schema (S)
4		SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5	Dirty read	dirty	A Write (E) A Schema (S) B Schema (S)
5	ROLLBACK				B Schema (S)
6		SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5	Original value	clean	B Schema (S)

図 9-6　独立性レベル 0 では許されるダーティ・リード

図 9-7 は、独立性レベル 1 で実行中の接続においてダーティ・リードがどのように回避されるかを示します。ステップ 4 の SELECT はブロックされます。接続 A がこのローに対する書き込みロックを保持しており、書き込みロックは独立性レベル 1 での読み込みをブロックするためです。SQL Anywhere では、実際にはすでに存在しない変更前の値を返すという解決策を実行はせずに、ダーティ・リードを完全にブロックします。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 0				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 1			
3	UPDATE t1 SET c1 = 'dirty' WHERE k1 = 5				A Write (E) A Schema (S)
4		SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5	Blocked: 42W18 User 'A' has the row in 't1' locked		A Write (E) A Schema (S) B Schema (S)
5	ROLLBACK				B Schema (S)
6		SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5	Unblocked	clean	B Schema (S)

図 9-7　独立性レベル 1 では回避されるダーティ・リード

図 9-7 は、ダーティ・リードを回避するためには長期にわたってロックを保持する必要がないことを示しています。ステップ 4 で接続 B がブロックされたのは、SELECT の途中でそのローに対する短期ロックを取得しようした試みが接続 A の書き込みロックに抵触したためです。この短期ロックは、A と B が保持するロックのカラムには記されません。この短期ロックは付与されておらず、sa_locks は、呼び出された時点（これらの例では、各ステップの終了時）で付与されているロックだけを表示するからです。短期ロックのメカニズムによって独立性レベル 1 のダーティ・リードは回避されます。

ダーティ・リードは必ずしも悪いものではなく、良いか悪いかはアプリケーションに依存します。たとえば、ある接続がカラム X を更新した後、別の接続が同じローのカラム Y を読み込む場合、これはアプリケーションの観点からは「ダーティ・リード」とみなされないことがあります。それにもかかわらず、この操作は独立性レベル 1 では実行できません。もう 1 つ考慮すべき点は、更新のほとんどは、ロールバックされずにコミットされるということです。なぜなら更新が未コミットの状態だからといって、それが必ずしもアプリケーション的なデータ不良を意味するものではないからです。

図 9-8 は、独立性レベル 0 で許可される同時実行動作のもう 1 つとして、「cursor instability（カーソル不安定性）」をとりあげます。ステップ 7 で接続 B は k1 = 5 の状態でローをフェッチし、ステップ 8 でそのローを変更した接続 A は、ステップ 9 でその変更をすぐコミットしています。ステップ 10 で接続 B が行った同じローの更新はブロックされません。このローに対する書き込みロックを接続 A はもう保持していないためです。しかし、接続 A による変更は期待どおりには実行されていません。SET c1 = c1 + 'er' の句によって、"clean" が "cleaner" に変更されずに "dirty" が "dirtyer" に変更されています。最後の誤った（不幸な？）結果がステップ 13 に示されています。この干渉の形式は、もう 1 つの接続がカー

ソル・ループで最後にフェッチされたローの変更を許されるため、「カーソル不安定性」と呼ばれます。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 0				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 0			
3		DECLARE cur1 CURSOR FOR SELECT k1, c1 FROM t1 ORDER BY k1 FOR READ ONLY			
4		OPEN cur1			B Schema (S)
5		FETCH cur1 INTO @k1, @c1		clean	B Schema (S)
6		FETCH cur1 INTO @k1, @c1		clean	B Schema (S)
7		FETCH cur1 INTO @k1, @c1		clean	B Schema (S)
8	UPDATE t1 SET c1 = 'dirty' WHERE k1 = 5		Allowed		A Write (E) A Schema (S) B Schema (S)
9	COMMIT				B Schema (S)
10		UPDATE t1 SET c1 = c1 + 'er' WHERE k1 = 5	Unstable		B Write (E) B Schema (S)
11		CLOSE cur1			B Write (E) B Schema (S)
12		COMMIT			
13		SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5	Incorrect result	dirtyer	B Schema (S)

図9-8　独立性レベル0では保証されないカーソル安定性

図9-9は、独立性レベル1によってカーソル安定性がどのように保証されるかを示します。ステップ7で接続Bがローをフェッチすると、ステップ8での接続Aによる更新はブロックされます。ステップ9で接続Bが行う更新は期待どおりの結果になり、ステップ11に示すように"clean"は"cleaner"に変更されます。

独立性レベル1でのカーソル安定性は、各ローをフェッチする毎にに設定される読み込みロックによって実行されます。たとえば、ステップ7で接続Bが取得する読み込みロックは、ステップ8で接続Aが書き込みロックを取得しようとする試みをブロックします。

こうした読み込みロックのそれぞれは、次のローがフェッチされてそのローに対する新しい読み込みロックが取得されるとすぐに解放されます。このようなカーソル安定性を実現するために、フェッチの都度読み込みロックを解放する動作は、「すべてのロー・ロックはトランザクション終了まで保持される」という原則の例外にあたります。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 0				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 1			
3		DECLARE cur1 CURSOR FOR SELECT k1, c1 FROM t1 ORDER BY k1 FOR READ ONLY			
4		OPEN cur1			B Schema (S)
5		FETCH cur1 INTO @k1, @c1		clean	B Read (S) B Schema (S)
6		FETCH cur1 INTO @k1, @c1		clean	B Read (S) B Schema (S)
7		FETCH cur1 INTO @k1, @c1		clean	B Read (S) B Schema (S)
8	UPDATE t1 SET c1 = 'dirty' WHERE k1 = 5		Blocked: 42W18 User 'B' has the row in 't1' locked		A Schema (S) B Read (S) B Schema (S)
9		UPDATE t1 SET c1 = c1 + 'er' WHERE k1 = 5	Stable		A Schema (S) B Write (E) B Read (S) B Schema (S)
10		CLOSE cur1			A Schema (S) B Write (E) B Schema (S)
11		SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5	Interim result	cleaner	A Schema (S) B Write (E) B Schema (S)
12		COMMIT			A Schema (S)
13	UPDATE t1 SET c1 = 'dirty' WHERE k1 = 5		Unblocked		A Write (E) A Schema (S)
14	COMMIT				
15	SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5		Final result	dirty	A Schema (S)

図9-9　独立性レベル1で保証されるカーソル安定性

図9-9のシナリオはステップ15まで続いており、いったん接続Bが読み込みロックを解放すると、最終的に接続Aは変更を行えることがわかります。ロック、ブロック、独立性レベルは競合するトランザクションにのみ影響を与え、競合しないトランザクション、または連続したトランザクションによって加えられる変更に対する保護は行いません。

また、ロックとブロックは同じトランザクションによって加えられる変更に対する保護も行いません。たとえば、単一のトランザクションが2つの異なるカーソルを同時に開くことがありますが、一方のカーソルが取得したロックによって、他方のカーソルが行った変更による干渉が防がれることはありません。

9.7.3　独立性レベル2

図9-10では、独立性レベル0と1では起こる可能性のある「繰り返し不可能読み出し」と呼ばれる干渉の形式をとりあげます。ステップ3と6において、接続Aは同じローを2回取り出し、2つの異なる結果を得ています。接続Aによって実行される2つのSELECT文の間で、接続Bがこのローの更新とその変更のコミットを行ったためです。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 1				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 1			
3	SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5			clean	A　Schema (S)
4		UPDATE t1 SET c1 = 'different' WHERE k1 = 5	Allowed		A　Schema (S) B Write (E) B Schema (S)
5		COMMIT			A　Schema (S)
6	SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5		Not repeated	different	A　Schema (S)

図9-10　独立性レベル1以下では保証されない繰り返し可能読み出し

図9-10に示した繰り返し不可能読み出しは、たとえ独立性レベルが1に設定されていても起こります。ステップ6では残っている書き込みロックがないため、図9-7でダーティ・リードを回避したメカニズムが働かないのです。また、ステップ3のSELECT文でも、図9-9のFETCH文と同様に長期の読み込みロックの取得をしなかったため、いずれもカーソル安定性が役に立っていません。

図9-10のステップ3では接続Aが、ダーティ・リードを回避するために短期ロックを取得したことに注意してください。しかし、この短期ロックは、SELECT文の終了時に解放されているため、ステップ4の接続Bによる書き込みロックの取得をブロックしていません。

図9-11は、読み込みが繰り返し可能であることを保証するには、2以上の独立性レベルが必要なことを示します。独立性レベルが2の場合、ステップ3で接続Aは取り出したローに対する読み込みロックを取得し、ステップ4における接続Bの書き込みロック取得を防ぎます。これで、ステップ5の2つめのSELECTは、最初のSELECTと同じ値を返します。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 2				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 1			
3	SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5			clean	A Read (S) A Schema (S)
4		UPDATE t1 SET c1 = 'different' WHERE k1 = 5	Blocked: 42W18 User 'A' has the row in 't1' locked		A Read (S) A Schema (S) B Schema (S)
5	SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5		Repeated result	clean	A Read (S) A Schema (S) B Schema (S)
6	COMMIT				B Schema (S)
7		UPDATE t1 SET c1 = 'different' WHERE k1 = 5	Unblocked		B Write (E) B Schema (S)
8		COMMIT			
9	SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5		Final result	different	A Read (S) A Schema (S)

図9-11　独立性レベル2で保証される繰り返し可能読み出し

図9-11のステップ6から9では、連続したトランザクションが独立性レベルの影響を受けないことをもう一度示します。接続Aが読み込みロックを解放するとすぐに、接続BはUPDATEの実行が可能になります。

図9-12では、独立性レベル0または1で起こる可能性のある干渉のもう1つの形式、「unstable update（不安定な更新）」をとりあげます。接続Bはステップ3で値"clean"を取り出し、接続Aはステップ4でこの値を"dirty"に更新して、ステップ5でこの変更をコミットします。ステップ6で接続Bは同じローの更新が可能になります。もはや接続Aは書き込みロックを保持していないからです。この2つめの更新はSET c1 = c1 + 'er'という句を使用するため、ステップ8における最終的な値は"dirtyer"になります。ただし、接続Bの観点からは、c1の現在値は"clean"だったはずで、新しい値は"cleaner"になるはずでした。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 1				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 1			
3		SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5		clean	B Schema (S)
4	UPDATE t1 SET c1 = 'dirty' WHERE k1 = 5		Allowed		A Write (E) A Schema (S) B Schema (S)
5	COMMIT				B Schema (S)
6		UPDATE t1 SET c1 = c1 + 'er' WHERE k1 = 5	Unstable		B Write (E) B Schema (S)
7		COMMIT			
8		SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5	Incorrect result	dirtyer	B Schema (S)

図9-12　独立性レベル1以下では保証されないUPDATEの安定性

図9-12のステップ6のUPDATEが、仮にSET c1 = @c1 + 'er'に変更された場合を考えます。@c1がステップ3で取り出されたカラムの値を保持している変数だとすると、ステップ8における最終的な値は"cleaner"になります。これは、接続Bの観点では予測されたとおりの結果ですが、接続Aにとってはそうではありません。この場合の矛盾は、「lost update（更新内容消失）」という形式にあたり、以前のデータに基づいた更新をあるトランザクションに許したために、もう一方のトランザクションの更新が消失するというものです。接続Aの観点からは、最終結果は"cleaner"または"dirtyer"ではなく"dirty"になるはずだったわけです。

図9-13は、独立性レベル2を使用すれば、不安定な読み込みを防ぐことができ、更新内容消失も回避されることを示します。このメカニズムは、繰り返し可能な読み込みを保証するために図9-11で使用されたものと同じです。独立性レベル2で実行中の接続は、取得する各ローに読み込みロックを取得し、この読み込みロックによって他の接続は書き込みロックを取得できなくなります。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION LEVEL = 1				
2		SET TEMPORARY OPTION ISOLATION LEVEL = 2			
3		SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5		clean	B Read (S) B Schema (S)
4	UPDATE t1 SET c1 = 'dirty' WHERE k1 = 5		Blocked: 42W18 User 'B' has the row in 't1' locked		A Schema (S) B Read (S) B Schema (S)
5		UPDATE t1 SET c1 = c1 + 'er' WHERE k1 = 5	Stable		A Schema (S) B Write (E) B Schema (S)
6		SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5	Interim result	cleaner	A Schema (S) B Write (E) B Schema (S)
7		COMMIT			A Schema (S)
8	UPDATE t1 SET c1 = 'dirty' WHERE k1 = 5		Unblocked		A Write (E) A Schema (S)
9	COMMIT				
10		SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5	Final result	dirty	B Read (S) B Schema (S)

図9-13　独立性レベル2で保証されるUPDATEの安定性

9.7.4　独立性レベル3

図9-14では、独立性レベル0、1、または2で起こる可能性のある干渉の形式、「phantom row（幻ロー）」を取り上げます。接続Aは、ステップ3で特定の選択基準に合うローを取り出し、ステップ6ではこれと同じSELECT文を使ってまったく異なるローを取り出します。この新しい幻ローは、接続Bによって挿入されたものです。接続Bはこの挿入とそのコミットをステップ4と5で行っています。接続Aは独立性レベル2で実行されるため、ステップ3で読み込みロックを取得していますが、読み込みロックでは新しいローの挿入は防げません。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 2				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 2			
3	SELECT FIRST c1 INTO @c1 FROM t1 WHERE k1 < 7 ORDER BY k1 DESC			clean	A Read (S) A Schema (S)
4		INSERT t1 VALUES (6, 'phantom')	Allowed		A Read (S) A Schema (S) B Write (E) B Schema (S)
5		COMMIT			A Read (S) A Schema (S)
6	SELECT FIRST c1 INTO @c1 FROM t1 WHERE k1 < 7 ORDER BY k1 DESC		Phantom row	phantom	A Read (S) A Read (S) A Schema (S)

図9-14　独立性レベル2以下では許される幻ロー

図9-15は、独立性レベル3によって幻ローの出現を回避する方法を示します。ステップ3で接続Aは、それ以降は選択基準を満たすどのようなローの挿入をも防ぐ対挿入ロックを取得します。このため、ステップ4で接続Bはブロックされ、その結果、ステップ5での幻ローの出現は回避されます。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 3				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 2			
3	SELECT FIRST c1 INTO @c1 FROM t1 WHERE k1 < 7 ORDER BY k1 DESC			clean	A Anti-insert (S) A Anti-insert (S) A Schema (S)
4		INSERT t1 VALUES (6, 'phantom')	Blocked: 42W18 User 'A' has the row in 't1' locked		A Anti-insert (S) A Anti-insert (S) A Schema (S) B Schema (S)
5	SELECT FIRST c1 INTO @c1 FROM t1 WHERE k1 < 7 ORDER BY k1 DESC		Old row	clean	A Anti-insert (S) A Anti-insert (S) A Schema (S) B Schema (S)
6	COMMIT				B Schema (S)
7		INSERT t1 VALUES (6, 'phantom')	Unblocked		B Write (E) B Schema (S)
8		COMMIT			
9	SELECT FIRST c1 INTO @c1 FROM t1 WHERE k1 < 7 ORDER BY k1 DESC		New row	phantom	A Anti-insert (S) A Anti-insert (S) A Schema (S)

図9-15　独立性レベル3で回避される幻ロー

Tip

高い独立性レベルで実行される、カーソル・フェッチ・ループ内側のCOMMIT文に注意します。COMMITの実行時にカーソルを開いておくためのWITH HOLD句を使うだけでは、COMMIT後になんらかのロー・ロックが保持されることを意味しません。実際、そうしたロー・ロックは保持されません。他の接続で実行されるSQL文による干渉からカーソル・ループ内側の処理を保護するために高い独立性レベルを使用する場合、各COMMITは、その時点までに取得されたすべてのロックによる保護を解放します。

図9-16では、独立性レベル0、1、または2で起こる可能性がある干渉のもう1つの形式、「suppressed update（更新の抑制）」をとりあげます。ステップ3で接続Aがローを削除し、ステップ4では同じローの更新を接続Bが試みます。独立性レベルが2以下の場合、WHERE句がどのローにも一致しないために何も起こらないことを除けば、この更新に問題はありません。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 2				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 2			
3	DELETE t1 WHERE k1 = 5				A Write (E) A Anti-insert + Insert (S) A Schema (S)
4		UPDATE t1 SET c1 = 'different' WHERE k1 = 5	Suppressed		A Write (E) A Anti-insert + Insert (S) A Schema (S) B Schema (S)
5		COMMIT			A Write (E) A Anti-insert + Insert (S) A Schema (S)
6	ROLLBACK				
7		SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5	Original value	clean	B Read (S) B Schema (S)

図9-16　独立性レベル2以下でDELETEによって抑制されるUPDATE

かなり前の図9-3に、あるローのDELETEを行うと、競合するトランザクション内にある、同じローに対する後続のINSERTが常にブロックされるという例を示しました。しかし、図9-16からは、DELETEを行っても、異なる接続による後続のUPDATEはブロックされず、ただ「何も行わない」操作に変わることがはっきりとわかります。

図9-16のステップ6で、接続Aは削除をロールバックしてカラムc1を元の値"clean"に戻しています。接続Bの観点からは、ステップ7のSELECTによって返される値は"different"になるはずですが、実際はそうなっていません。

Tip

UPDATE の結果をチェックするときは、「エラーがない」ことと「正しく動作した」ことを混同しないようにします。アプリケーションは、SELECT @@ROWCOUNT を使用して、UPDATE によって実際に影響を受けたローの個数を取り出し、個数が 0 かつそれが好ましくない場合にはアクションを実行できます。@@ROWCOUNT の値は、UPDATE の直後に参照してください。それ以降では SELECT などの後続の SQL 文によって値が変更される可能性があるためです。

図 9-17 に、独立性レベル 3 で、競合するトランザクションが削除されたローの更新をブロックする問題 (更新の抑制) がどのように回避されるかを示します。ステップ 6 に示すように、ブロックされた接続は、待機するか、後で更新を再試行するかを選択できます。この状況では、独立性レベル 2 と 3 の違いは、取得されるロック数ではなく、取得されなかったロックに現れます。図 9-17 のステップ 4 で接続 B は失われたローによって残されたギャップに対して対挿入ロックの取得を試みますが、接続 A が同じギャップに対して挿入ロックを保持しているためにブロックされます。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 2				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 3			
3	DELETE t1 WHERE k1 = 5				A Write (E) A Anti-insert + Insert (S) A Schema (S)
4		UPDATE t1 SET c1 = 'different' WHERE k1 = 5	Blocked: 42W18 User 'A' has the row in 't1' locked		A Write (E) A Anti-insert + Insert (S) A Schema (S) B Schema (S)
5	ROLLBACK				B Schema (S)
6		UPDATE t1 SET c1 = 'different' WHERE k1 = 5	Unblocked		B Write (E) B Schema (S)
7		COMMIT			
8		SELECT c1 INTO @c1 FROM t1 WHERE k1 = 5	Updated value	different	B Read (S) B Schema (S)

図 9-17　独立性レベル 3 で DELETE によってブロックされる UPDATE

Note

これらの図には、付与されたロックだけが記されています。すなわち、ロックの取得を試みた接続が既存のロックにブロックされたために取得されなかったロックは、図には明示されていません。たとえば、図 9-17 のステップ 4 では、接続 B による対挿入ロックは取得されていないために記されていません。これらの図は組み込みプロシージャ sa_locks を使って作成されていますが、このプロシージャでは存在しないロックは表示されないからです。この例において、仮に、ステップ 3 と 4 の間で接続 A が COMMIT を実行したとすると、接続 B がステップ 4 で実行する UPDATE は、削除されたローによって残されたギャップの対挿入ロックの取得に成功し、sa_locks の呼び出しによってこのロックが表示されることになります。

Note

図9-16と9-17との違いは、接続Aの使用する独立性レベルではなく、接続Bの独立性レベルにあります。つまり、接続Aは、仮にその独立性レベル0であったとしても、やはり対挿入ロックと挿入ロックを取得することになります。

独立性レベル2と3で実行されるSELECT文は驚くほど多くのロックを取得する可能性があります。たとえば、独立性レベル0または1を使って、ASADEMOデータベースに対する以下のクエリを実行すると、75のローが返されてもスキーマ・ロックはわずかに1つしか取得されません。しかし、独立性レベル2の場合、このスキーマ・ロックに加えて75の読み込みロックが取得され、ロー1つにつき読み込みロックが1つ返されます。このことは、こうしたロックがCOMMITまたはROLLBACKによって解放されるまでは、他の接続はこれらのローをどれ1つとして更新できないことを意味します。

```
SELECT *
  FROM sales_order_items
 WHERE quantity = 48;
```

図9-18は、独立性レベル2において多数のロックを取得する別のクエリです。ステップ3のSELECTの実行によって、テーブルt1のローの数がカウントされると同時に、テーブルのすべての単一ローに対して読み込みロックが取得されます。このため、ステップ4で接続Bが試みる更新はブロックされます。実際、他の接続が試みる、このテーブルのどのローに対する更新も削除もすべてブロックされます。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 2				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 0			
3	SELECT COUNT(*) INTO @c1 FROM t1		Single value	5	A Read (S) A Read (S) A Read (S) A Read (S) A Read (S) A Schema (S)
4		UPDATE t1 SET c1 = 'different' WHERE k1 = 5	Blocked: 42W18 User 'A' has the row in 't1' locked		A Read (S) A Read (S) A Read (S) A Read (S) A Read (S) A Schema (S) B Schema (S)

図9-18　独立性レベル2での極端なロックの例

Tip

独立性レベル2と3を使う場合は特に、トランザクションを短くするよう心がけてください。他のSQL文とは別にしたSELECT文をトランザクションに含め、そのSELECTのすぐ後にCOMMITを入れることで、ロックが保持される時間を短縮できることがあります。

独立性レベル3におけるSELECTは、以下のクエリの各テーブルに対して対挿入ロックを取得します。

- インデックス・スキャンを使用してテーブルにSELECTを実行する場合、読み込んだ各ローに対して直前への挿入を防ぐための対挿入ロックが1つ取得され、さらに、結果セットの最後への挿入を防ぐための対挿入ロックが1つ追加される。こうした理由により、図9-15のステップ3には対挿入ロックが2つ記されている。これらは、カラムk1のプライマリ・キー・インデックスを使用して取得したローに対するロックと追加のロックである
- このテーブルにインデックスが存在しないか、どのインデックスも選択条件を満たすことができないために、インデックス・スキャンが使用されない場合、テーブル内のありとあらゆるローのための対挿入ロックが1つ取得され、さらにその最後に追加ロックが1つ取得される。仮にカラムk1にインデックスが存在しなかった場合、テーブルt1には5つのローがあるため、図9-15のステップ3には取得された6つの対挿入ロックが記されることになる

独立性レベル3では影響が非常に大きくなる可能性があります。たとえば、以下のSELECTをASADEMOデータベースに対して実行すると、ローは75個しか返されませんが、このテーブルには1097のローが存在してquantitiyカラムにインデックスがないために、1098個の対挿入ロックが取得されます。この簡単なクエリーは、これらのロックがCOMMITまたはROLLBACKによって解放されるまでは、sales_order_itemsテーブルのどのローに対する挿入、更新、削除であっても他の接続によるこうした変更をすべてブロックします。

```
SET TEMPORARY OPTION ISOLATION_LEVEL = '3';
SELECT *
  FROM sales_order_items
 WHERE quantity = 48;
```

独立性レベル3では、SQL Anywhereは検査されるすべてのローに対してロックを取得するため、通常はもっと多くのロックが取得されます。一方、独立性レベル2では、ローが最終的な結果セットに寄与する場合にのみ、そのローに対するロックが取得されます。この違いは、逐次スキャンが必要な場合に最も顕著になります。

図9-19に、独立性レベル3において極端なロックが行われる別の例を示します。ステップ3のSELECTは何も返しませんが、テーブルのすべての単一ローに対して対挿入ロックを取得します。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 3				
2		SET TEMPORARY OPTION ISOLATION_LEVEL = 0			
3	SELECT FIRST c1 INTO @c1 FROM t1 WHERE c1 <> 'clean' ORDER BY c1		Empty result		A Anti-insert (S) A Anti-insert (S) A Anti-insert (S) A Anti-insert (S) A Anti-insert (S) A Anti-insert (S) A Schema (S)
4		UPDATE t1 SET c1 = 'different' WHERE k1 = 5	Blocked: 42W18 User 'A' has the row in 't1' locked		A Anti-insert (S) A Anti-insert (S) A Anti-insert (S) A Anti-insert (S) A Anti-insert (S) A Anti-insert (S) A Schema (S) B Schema (S)

図9-19　独立性レベル3での極端なロックの例

Tip

ISOLATION_LEVEL データベース・オプションの設定は、トランザクションの実行中に動的に変更できます。干渉に対して高いレベルの保護を必要とするSQL文の実行前には高いレベルを、あまり保護が必要ではなく、したがってそれほど多くのロックを必要としない文には低いレベルを設定できます。FROM句のNOLOCKとREADCOMMITTEDのような「テーブル・ヒント」を使うことによって同じクエリ内でテーブルごとに異なる独立性レベルを指定することもできます。この構文の詳細については、「3.3　FROM句」を参照してください。

LOCK TABLE文をIN EXCLUSIVE MODE句と共に使用すると、1つのテーブルについて取得されるロック数を大幅に削減することができます。たとえば、テーブルt2に100,000個のローが含まれている場合、次のSELECT文は、独立性レベル3の処理に従って100,002個のロックを取得します。

```
SET TEMPORARY OPTION ISOLATION_LEVEL = '3';
SELECT COUNT(*)
  FROM t2;
```

次のようにLOCK TABLE文を追加すると、ロック数は実に1にまで削減されます。

```
SET TEMPORARY OPTION ISOLATION_LEVEL = '3';
LOCK TABLE t2 IN EXCLUSIVE MODE;
SELECT COUNT(*)
  FROM t2;
```

また、LOCK TABLE 文は、独立性レベルが低い場合にも、更新の操作に役立ちます。たとえば、次の UPDATE 文はテーブル t2 内の 100,000 個のローをすべて変更しますが、あるテストでは、LOCK TABLE 文を追加することによって実行速度が 3 倍になりました。

```
SET TEMPORARY OPTION ISOLATION_LEVEL = '0';
LOCK TABLE t2 IN EXCLUSIVE MODE;
UPDATE t2 SET non_key_1 = 'xxx';
```

ただし、マルチユーザ環境での LOCK TABLE 文の使用には十分な注意が必要です。LOCK TABLE を使ったトランザクションはできるだけすぐにコミットを行うようにします。

Section

9.8 デッドロック

図 9-20 は、「cyclical deadlock（環状デッドロック）」と呼ばれる状態の例です。ステップ 1 と 3 では、どちらの接続とも独立性レベルを 0 に設定しています。どの独立性レベルでも環状デッドロックが発生する可能性があることを示すためです。ステップ 2 と 4 では、BLOCKING オプションを 'ON' に設定しています。これにより、それぞれの接続は、もう一方の接続の保持するロックによってブロックされた場合にすぐに例外を発生させずに強制的に待機させられます。

	Connection A	Connection B	Comment	c1 Value	Locks Held by A & B
1	SET TEMPORARY OPTION ISOLATION_LEVEL = 0				
2	SET TEMPORARY OPTION BLOCKING = 'ON'				
3		SET TEMPORARY OPTION ISOLATION_LEVEL = 0			
4		SET TEMPORARY OPTION BLOCKING = 'ON'			
5	UPDATE t1 SET c1 = 'changed by A' WHERE k1 = 1		A gets lock		A Write (E) A Schema (S)
6		UPDATE t1 SET c1 = 'changed by B' WHERE k1 = 3	B gets lock		A Write (E) A Schema (S) B Write (E) B Schema (S)
7	UPDATE t1 SET c1 = 'changed by A' WHERE k1 = 3		A blocked		A Write (E) A Schema (S) B Write (E) B Schema (S)
8		UPDATE t1 SET c1 = 'changed by B' WHERE k1 = 1	Deadlock: 40001 Deadlock detected		A Write (E) A Schema (S) B Write (E) B Schema (S)
9		ROLLBACK	A unblocked		A Write (E) A Write (E) A Schema (S)
10		SELECT c1 INTO @c1 FROM t1 WHERE k1 = 1	Final result	changed by A	A Write (E) A Write (E) A Schema (S) B Schema (S)
11		SELECT c1 INTO @c1 FROM t1 WHERE k1 = 3	Final result	changed by A	A Write (E) A Write (E) A Schema (S) B Schema (S)

図 9-20　独立性レベル 0 における環状デッドロック

Note

大部分のアプリケーションでは、BLOCKINGオプションにデフォルト値'ON'を使用してください。ほとんどのブロックは有効期間が短く、目の前の失敗に対処するよりも来たるべき成功を待つほうが容易です。なお、これまでの図ではBLOCKINGオプションは'OFF'であると想定していましたが、これはただロックとブロックがどのように作用するかを説明するためです。

図9-20のステップ5と6でそれぞれの接続はローを更新します。ステップ7で接続Aは、ステップ6で接続Bが更新したローを更新しようとしますが、接続Aによるこの試みはブロックされます。ステップ8で接続Bは、ステップ5で接続Aが更新したローを更新しようとします。SQL Anywhereはこの時点で環状デッドロック状態を検出します。接続Aはブロックされたまま接続Bがロックを解放するのを待ち、接続Bもブロックされたまま接続Aの処理が終了するのを待っている状態です。このブロックどうしの循環またはサイクルが環状デッドロックと呼ばれるもので、どちらの接続も処理を進められません。そこで、SQL Anywhereは、両方の接続を永遠に待機させるのではなく、ステップ8の更新を自動的にキャンセルし、SQLSTATE値'40001'により問題を接続Bに知らせます。

デフォルトでは、SQL Anywhereは環状デッドロックであるSQLSTATE値'40001'を特別な方法で処理します。現在の操作の終了時にSQLSTATE値がまだ'40001'が設定されている場合、SQL Anywhereはその接続に対するROLLBACKを自動的に行ってからクライアント・アプリケーションに戻ります。このデフォルト動作は、SQLSTATE'40001'を取得し、例外を先へ渡すためのRESIGNAL文を実行しない例外ハンドラを持つBEGINブロックを使用することによって回避できます。この場合はSQLSTATEが再び'00000'に設定されてからクライアント・アプリケーションに戻るため、SQL Anywhereは自動ROLLBACKを行いません。影響を受けた接続は、このROLLBACKの有無に関わらず、自由に処理を進められます。ROLLBACKが行われた場合、ブロックされていたロックがなくなるため、もう一方の接続もまた処理を進めることができます。ROLLBACKが行われなかった場合、もう一方の接続はブロックされたままです。例外ハンドラを持つBEGINブロックの詳細については、「8.3 例外ハンドラ」を参照してください。RESIGNAL文と、例外ハンドラのより多くの例については、9.5.1と9.5.2を参照してください。

Note

'40001'以外のSQLSTATEに対しては、SQL Anywhereは自動ROLLBACKを行いません。また、自動ROLLBACKが実行されるためには、実際の環状デッドロックの状態になっている必要はありません。SIGNAL文を使用してSQLSTATEを'40001'に設定すれば、自動ROLLBACKを実行できます。ただし、現在の操作完了前に例外ハンドラまたは他のロジックによってSQLSTATEが他の値に設定される場合を除きます。

図9-20に示した例では、ステップ9に明示的な別のROLLBACKがあるため、接続Bが行った変更はすべてロールバックされます。これにより、接続Aは、ステップ9で取得した2つめの書き込みロッ

クを使ってすぐに処理を進められます。ステップ10と11のSELECT文の結果からわかるように、この環状デッドロックの競合では接続Aが勝者であり、最終結果には接続Aによる操作が反映されています。

SQL Anywhereで環状デッドロックが発生することはきわめてまれです。ほとんどの操作でロー・ロックが使用されるからです。SQL Anywhereにはページ・ロックの類は存在しません。また、たとえロー・ロックが数百万個になっても、テーブル・ロックに「昇格」することは決してありません。

異なる接続で実行されても常に同じ操作を同じ順序で行なうようにトランザクションを設計することによって、多くの環状デッドロックを回避できます。たとえば、図9-20の環状デッドロックの原因は、重複するトランザクションが同じローを異なる順序で更新したことにあります。仮に、重複するトランザクションが同じローを同じ順序で更新したとすると、環状デッドロックの危険性はありません。一方の処理が終了するまで他方の接続がブロックされるだけであり、その後で他方の接続も処理を進められたはずです。

独立性レベルが高くなると、環状デッドロックが起こりやすくなります。ブロックの原因となるロックが増えるためです。たとえば、独立性レベルが2または3の場合、同じローに対するSELECTを行う2つの接続は、両方とも共有読み込みロー・ロックを取得します。なお、独立性レベル2における共有読み込みロー・ロックの例については、図9-11のステップ3を参照してください。両方の接続がその後このローのUPDATEを行おうとすると、一方はブロックされ、もう一方によって環状デッドロックのエラーが起こります。このシナリオの場合、1つの解決策は、それぞれの接続に、まずUPDATEを、次にSELECTを実行させることです【監注13】。こうすれば、先にUPDATEを実行した接続が処理を進めることができ、もう一方の接続はすぐにブロックされるため、デッドロックは起こりません。

Tip

BLOCKING_TIMEOUTオプションを0以外の値に設定すると、環状デッドロックの発生時に接続による作業の繰り返しが容易になります。BLOCKING_TIMEOUTのデフォルト値は0で、この値は「永遠に待つ」ことを意味します。環状デッドロックが発生し、そこにBLOCKING_TIMEOUTが0以外の値に設定されている接続が1つ以上が関係している場合、0でない最小の値を持つ接続がエラーを受け取ります。これは、処理を進めるべき重要な更新をある接続が行い、もう一方の接続が後で容易に再実行できるレポート生成を行う場合に役立ちます。

「thread deadlock（スレッド・デッドロック）」と呼ばれる別の種類のデッドロックが発生するのは、オペレーティング・システムのすべてのタスク、またはSQL Anywhereエンジンで使用可能なすべての実行スレッドが、ブロックされた接続に占有された場合です。SQL Anywhereエンジンは、内部でスレッド・プールを利用しています。スレッド・プールを行うと、接続の数がスレッド数を超えることが可能です。1つのスレッドでアクティブな接続のそれぞれが実行されている一方で、どの時点でもアイドル状態の接続があり、そうした接続のための作業はどのスレッドでも実行されていないためです。アイドル状態になった接続は、使っていたスレッドを解放して別の接続のためにフリー・スレッドのプールに返します。ただし、アクティブな接続がブロックされるとスレッドは解放されません。すべてのス

【監注13】FROM句にWITH (XLOCK)を付ければ、排他ロックをかけつつSELECTできる。

レッドがブロックされた接続に占有された場合にスレッド・デッドロックという状態が起こります。、この時点ではどの作業も進められません。すべてのスレッドを永遠に待機させることのないように、SQL Anywhere はブロックされた操作の 1 つをキャンセルし、SQLSTATE'40W06' とエラー・メッセージ「すべてのスレッドがブロックされています。」によってその接続に問題を通知します。

デフォルトでは、SQL Anywhere ネットワーク・サーバ dbsrv9.exe はプールに 20 個のスレッドを持ち、パーソナル・サーバ dbeng9.exe は 10 個のスレッドを持っています。このスレッド数は、同時に処理できる接続の数を制限するのではなく、一度に処理できるアクティブ状態にある接続の数を制限します。また、多数のロックを取得して頻繁にブロックを起こす、ビジー状態にある接続が増えると、スレッド・ロックが起こる可能性が出てきます。

次のクエリは、組み込みの sa_conn_info プロシージャを使用して、ブロックされた接続とブロックしている接続をすべて表示します。

```
SELECT NUMBER(*) AS "#",
       Name,
       UserId,
       Number,
       BlockedOn
  FROM sa_conn_info() AS conn1
 WHERE BlockedOn <> 0
    OR EXISTS ( SELECT *
                  FROM sa_conn_info() AS conn2
                 WHERE conn2.BlockedOn = conn1.Number )
 ORDER BY BlockedOn,
       Name,
       UserId,
       Number;
```

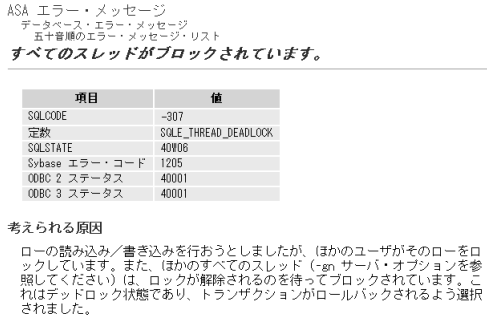
次の例は、20 スレッドをサポートし、同一テーブル内の同じローを同時に更新しようとする 25 個の接続を持つサーバで上記のクエリを実行したときの出力です。1 つの接続がこの更新に成功し、それ以降に行おうとした 19 回の更新はブロックされています。スレッド・デッドロックが起こした 21 回目の試行は、残り 4 回の試行と同様にキャンセルされています。以下の出力には、19 個のブロックされた接続とそれらをブロックする 1 つの接続が示されています。

```
#   Name  UserId  Number       BlockedOn   LockName
==  ====  ======  ===========  =========   ========
1   C01   C01     1447092880   0           0
2   C02   C02     2016944313   1447092880  445
3   C03   C03     1579014964   1447092880  445
4   C04   C04     1141085615   1447092880  445
5   C05   C05     439312098    1447092880  445
6   C06   C06     571234182    1447092880  445
7   C07   C07     133304833    1447092880  445
8   C08   C08     1710937048   1447092880  445
9   C09   C09     265226917    1447092880  445
```

```
10  C10   C10     835078350    1447092880  445
11  C11   C11     1842859132   1447092880  445
12  C12   C12     1273007699   1447092880  445
13  C13   C13     954498130    1447092880  445
14  C14   C14     703156266    1447092880  445
15  C15   C15     1382749      1447092880  445
16  C20   C20     1524349563   1447092880  445
17  C21   C21     1404929783   1447092880  445
18  C22   C22     1974781216   1447092880  445
19  C23   C23     384646697    1447092880  445
20  C24   C24     516568781    1447092880  445
```

以下に、上記のカラムの意味を説明します。# カラムは行番号です。Name と UserID のカラムは接続名とユーザ ID をそれぞれ表します。Number カラムには、それぞれの接続をユニークに識別する番号が記されています。Blocked On カラムには、その接続をブロックしている接続の番号が記されています。LockName にはそのブロックに対するロックをユニークに識別する番号が記されています。接続がブロックされていない場合、BlockedON と LockName は 0 になります。

前に述べたように、スレッド・デッドロックを検出して操作をキャンセルすると、SQL Anywhere は SQLSTATE を '40W06' に設定します。この場合、以前の環状デッドロックのところで説明した自動 ROLLBACK を SQL Anywhere は行いません。ただし、アプリケーションの観点からは、環状デッドロックの場合と同じ '40001' が SQLSTATE の値として返される場合があります。ODBC など、ある種のクライアント・インタフェースでは、ある変換処理を経て SQLSTATE の値が返されるためです。こうした代替的な SQLSTATE 値についての説明は SQL Anywhere オンラインヘルプにあります。図 9-21 は、SQL Anywhere オンラインヘルプにおけるスレッド・デッドロックの説明です。SQLCODE は -307 に、エンジン内部の SQLSTATE は '40W06' になっています。しかし、ODBC の Version 2 または 3 のインタフェースを使用するアプリケーションに返される SQLSTATE の値は '40001' に変わります。このことは "ODBC 2 ステータス" と "ODBC 3 ステータス" ラベルの付いた項目からわかります。


ASA エラー・メッセージ
データベース・エラー・メッセージ
五十音順のエラー・メッセージ・リスト

すべてのスレッドがブロックされています。

項目	値
SQLCODE	-307
定数	SQLE_THREAD_DEADLOCK
SQLSTATE	40W06
Sybase エラー・コード	1205
ODBC 2 ステータス	40001
ODBC 3 ステータス	40001

考えられる原因

ローの読み込み／書き込みを行おうとしましたが、ほかのユーザがそのローをロックしています。また、（ほかのすべてのスレッド（-gn サーバ・オプションを参照してください）は、ロックが解除されるのを待ってブロックされています。これはデッドロック状態であり、トランザクションがロールバックされるよう選択されました。

図 9-21　SQLSTATE'40W06' に対する SQL Anywhere オンラインヘルプ

スレッド・デッドロックの発生によって、ビジー状態のサーバが単純により多くのスレッドを必要としているのだとわかることがあります。dbsrv9 の -gn コマンド・ライン・オプションを使用すれば、ス

レッド数を増やせます。しかし、スレッド・デッドロックがアプリケーション設計の不備を示していることもあります。接続が多すぎるためにリソースが不足し、競合が起こる場合がこれにあたります。先ほどの例では、ブロックされた19の接続すべてが同じデータベース・オブジェクトを取得しようとしているのが自明です。実際には、異なるユーザ全員が同時に同じ作業を行おうとすることはないでしょうが、利用可能なスレッド数を増やすだけでもブロックされる接続数が増えることがあります。

たとえば、次に使用可能なプライマリ・キーの値の計算を、DEFAULT AUTOINCREMENTを使用せずに行うために、単一テーブル内のローを1つ更新するアプリケーションを考えます。新しいプライマリ・キーを計算しようとする接続が多すぎて競合が発生すると、このアプリケーションは容易にスレッド・ロックに陥ります。実運用の観点からは、こうした接続は異なる作業を行っているはずです。そのため、このスレッド・デッドロックは設計の不備による人為的なものといえます。

Section

9.9 ミューテックス

SQL Anywhereエンジンは、SQL操作の処理に複数のCPUを使用できます。1つ1つの操作は、複数のCPUに分割して割り当てられるのではなく、1つのCPUで処理されます。しかし、SQL Anywhereでは、2つ以上の接続から来る複数の要求を同時に処理できます。

理論的には、n個のCPUによって、同時に起こるn個の要求を、1つのCPUが1つの要求を処理するのと同じ時間で、処理できるはずです。たとえば、1つのCPUが1つの要求を10秒で処理する場合、2つのCPUを使えば2つの要求を10秒の時間で処理できるはずです。

現実的にはこれは不可能です。常にオーバヘッドがかかるため、同時に起こる2つの要求を処理するには10秒よりも長くかかります。SQL Anywhereがこの2つの要求を12～13秒で処理しているなら、1つのCPUで20秒かけて処理するよりもかなり良いといえます。

しかし、2つの要求の処理に20秒かかっているとすれば、オーバヘッドに大きな問題があるはずです。この場合はCPU追加の利益がまったくありません。2つの要求の処理に20秒より長くかかっているとすれば、深刻な問題です。この場合はむしろCPUを追加しないほうが良いことになります。

複数のCPUによるオーバヘッドの問題が起こる原因は、「mutexes（ミューテックス）」、すなわち、相互排他操作にあります。ミューテックスとは、SQL Anywhereのようなマルチスレッド・プログラムが共有の内部データ構造を競合や破壊から保護するためのメカニズムです。ミューテックスはロー・ロックに似ていますが、次のような違いがあります。

- ミューテックスはロックよりも頻繁に発生する
- ミューテックスはロックほど長く続かない
- ミューテックスは、ロックやブロックの影響を受けない読み込み専用クエリにも影響を与えることができる

- ミューテックスは、単一CPUよりも複数CPUの場合に大きな問題になる
- ミューテックスによる待ち時間が、ある程度の時間よりも長くなると、コンボイ（1つを除くすべてのデータベース接続が共有リソースを待機している状態）が起こる可能性がある
- ミューテックスを表示するツール、またはミューテックスによって起こる競合を直接測定するツールがない
- 要求レベル・ログを使用して、複数CPUでうまく動作しないSQL文を探せる

コンボイは、別々の接続が同じ内部データ構造に繰り返しアクセスする必要がある場合に起こります。データに対する作業にかかる時間がミューテックスのためのチェックと待機にかかる時間の割に短い場合、ただ1つの接続がそのデータに対して作業を行い、他の接続はすべて待機している状況が起こることがあります。順番を待つ列の先頭にある接続が作業の一部を終えて他の接続に制御を譲った後、もう一度同じデータにアクセスしようとすると、この接続はコンボイを再結合して再び順番を待つ必要があります。この状況における全体のスループットは、複数CPUのほうが単一CPUの場合よりも悪くなる可能性があります。

Tip

ありもしない問題を探さないようにしてください。ミューテックスでのコンボイがパフォーマンスの問題を起こすことはめったにありません。ミューテックス自体はとても一般的なものです。ミューテックスは、SQL Anywhereだけでなく、あらゆるマルチスレッド・ソフトウェアで使われていますが、普通は害のないものです。

競合の問題が起こってその原因と考えられるロックとブロックを削除したときは、SQL Anywhereの要求レベル・ログの機能を使用すると、ミューテックスの状況証拠を探すことができます。以下にそのステップを示します。

❶ 複数のCPUの使用時にスループットが低下する、または少なくとも期待したほどスループットが良くないことが明らかになるように負荷をかけます。
❷ 要求レベル・ログをオンにして、1つの接続からその負荷となる処理を実行します。この機能の詳細については、「10.2　要求レベルのロギング」を参照してください。
❸ 組み込みプロシージャsa_get_request_timesを実行して、要求レベル・ログ・ファイルの分析と組み込みのテンポラリ・テーブルsatmp_request_timeへの結果の保存を行います。satmp_request_timeの内容を同じスキーマを持つ別のテーブルにコピーして、ステップ7で利用できるようにします。
❹ 要求レベル・ログをオフにし、その出力ファイルを次のステップに備えて削除します。
❺ 要求レベル・ログを再びオンにして、先ほどの負荷を複数の接続から同時に実行します。
❻ ステップ3をもう一度行って、2回目のテストのsatmp_request_timeの内容を別のテーブルに保

存します。

❼ ステップ3と6のデータを比較します。そのときは、ステップ5での複数の接続による実行時に、ステップ2での1つの接続による実行時よりも時間のかかったSQL文を探します。ミューテックス上のフォールトの有無に関わらず、探し出したSQL文はスループットを低下させており、調べる価値があります。

複数CPU使用時のスループットの問題は、アプリケーションの処理方法を変更することで解決できる場合があります。具体的には次のような変更です。

- 多数の異なる接続で、同じローの小さなサブセットを同時に参照しないようにする
- それぞれの接続で、同じ順序で同じデータを何度も取り出さないようにする

1つの方法は、繰り返し取り出すのではなく、アクティブ状態で読み込んだこのデータをクライアントにコピーすることです。もう1つの方法は、首尾よくこのデータを各接続にとってローカルな別のデータにするために、グローバル・テンポラリ・テーブルにコピーすることです。

Section

9.10 GRANT CONNECT

SQL Anywhereにおいて、接続はクライアント・アプリケーションと1つのデータベースの間をつなぐリンクです。実際の作業に必要な接続を最初に確立するために、クライアント・アプリケーションはユーザIDとパスワードを1つずつ指定しなければなりません。このユーザIDとパスワードは事前に定義されてSYSUSERPERMカタログ・テーブルに格納されている必要があります。新規のSQL Anywhereデータベースを作成すると、そういうユーザIDとパスワードの組が予め1つ存在します。DBAとSQLです。この情報で十分なアプリケーションを作成しているのであれば、この章をとばして「9.11　ロギングとリカバリ」に進んでかまいません。

GRANT文を使用すると、ユーザIDの作成と、ユーザIDに対する次の処理の許可ができます。

- データベースへの接続
- テーブルからのローの選択
- テーブルへのローの挿入
- テーブル内のローの更新
- テーブルからのローの削除
- プロシージャまたは関数の実行

- テーブルと他のオブジェクトの所有
- テーブルとビューの作成
- テーブル内のインデックスと外部キーの作成
- テーブルのスキーマの変更
- 他のユーザ ID への権限の付与
- グループとしての動作
- グループのメンバとしての権限の継承
- 管理タスクの実行
- Windows の統合化ログインを利用したデータベース接続

以下に、GRANT 文を使用して、ユーザ ID を作成してデータベースに接続するパーミッションを与える方法を示します。

```
<create_user>              ::= GRANT CONNECT TO <user_id_list>
                                 [ IDENTIFIED BY <positional_password_list> ]
<user_id_list>             ::= <user_id> { "," <user_id> }
<positional_password_list> ::= <password> { "," <password> }
<password>                 ::= <identifier> -- パスワードとして使用
                             | パスワードを含む文字列リテラル
                             | 空文字列リテラル
                             | 二重引用符で囲んだ空文字列
```

ユーザ ID リストには、1 つ以上の SQL Anywhere 識別子を含めることができます。IDENTIFIED BY 句はオプションで、ユーザ ID に割り当てるパスワードの指定に使用します。複数のユーザ ID とパスワードを指定する場合、両者の対応はその位置で決まります。つまり、リストの最初のパスワードが最初のユーザ ID に割り当てられ、2 番目のパスワードが 2 番目のユーザ ID に割り当てられる、というように対応します。

データベースに接続するには、ユーザ ID がパスワードを持っていなければなりません。パスワードは空文字列でもかまいませんが、IDENTIFIED BY 句の中で明示的に指定する必要があります。以下のいくつかの例は、新しいユーザ ID を作成する GRANT 文です。ユーザ ID の A、B、"C 2"、E はパスワードを持つのでデータベースへの接続に使用できますが、D はパスワードを持たないので接続できません。

```
GRANT CONNECT TO A, B, "C 2" IDENTIFIED BY SQL, '', 'QWER ASDF';
GRANT CONNECT TO D;
GRANT CONNECT TO E IDENTIFIED BY "";
```

Note

本書では、すべてのデータベースが大文字と小文字を区別しないと想定しています。デフォルトでも大文字と小文字を区別しない設定になっています。これは、パスワードでも大文字と小文字が区別されな

いことを意味します。たとえば、上記のようにパスワードを大文字で定義した場合でも、データベース接続時には同じ文字列を小文字で入力できます。ただし、大文字と小文字を区別するデータベースに接続する場合は、ユーザ ID では大文字と小文字が区別されなくてもパスワード文字列では大文字と小文字が区別されます【監注 14】。

上記のように、ユーザ ID は、二重引用符の囲みはあってもなくてもかまいませんが、SQL Anywhere 識別子としてコード化されます。一方、パスワード文字列は、いくつかの形式を使ってコード化できます。この形式には、二重引用符の囲みありまたは二重引用符の囲みなしの識別子、一重引用符で囲まれた文字列リテラル、または二重引用符で囲まれた空文字列があります。

以下の 4 つの ISQL コマンド・ラインは、先ほど定義した A、B、"C 2"、E というユーザ ID を使ってデータベースに接続するためのものです。ユーザ ID とパスワードの値は -c に続く接続文字列の中で指定されていることに注意してください。"C 2"、"QWER ASDF"、あるいはユーザ ID の B または E のパスワードとして使われる空文字列の前後にデリミタは必要ありません。

```
"%ASANY9%¥win32¥dbisql.exe" -c "ENG=test9;DBN=test9;UID=A;PWD=sql;CON=A"
"%ASANY9%¥win32¥dbisql.exe" -c "ENG=test9;DBN=test9;UID=B;PWD=;CON=B"
"%ASANY9%¥win32¥dbisql.exe" -c "ENG=test9;DBN=test9;UID=C 2;PWD=QWER ASDF"
"%ASANY9%¥win32¥dbisql.exe" -c "ENG=test9;DBN=test9;UID=E;PWD="
```

Tip

上記の最初と 2 番目の ISQL コマンド・ラインに示したように、データベースへの接続時に「接続名」を指定するには、CON 接続パラメータを使用します。接続名は、組み込み関数呼び出し CONNECTION_PROPERTY ('名前') を介してアプリケーション・プログラムでも使用できます。接続名は、組み込みプロシージャ sa_conn_info によって返される情報に含められます。接続名を使用すると、同じユーザ ID を使用する接続どうしを区別できます。また、ISQL で接続名を使用すると、タイトル・バーに接続名が表示されます。図 9-22 は、接続名 A と B を使用したものはウィンドウとタスク・バー・アイテムの対応が明確ですが、ユーザ ID が E のものではサーバ名が使われているために対応がわかりにくいことを示しています。

【監注 14】原文の誤り。SQL Anywhere では、文字列を比較するときの大文字・小文字の区別方法は、3 つに分類される。
①データ――デフォルトでは区別されないが、データベース作成時に区別するように設定できる。
②パスワード――デフォルトでは区別されないが、データベース作成時に区別するように設定できる。①の設定とは独立して働く。
③識別子――①②の設定にかかわらず、常に区別されない。ユーザ ID もここに分類される。

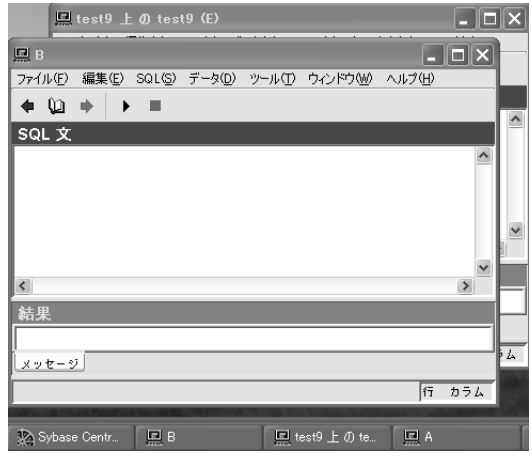


図 9-22　ISQL タスク・バー・タイトルに表示される接続名

GRANT をはじめとする SQL 文で指定された識別子を参照するとき、「ユーザ」、「ユーザ ID」、「ユーザ名」という用語が同じ意味で使われることがよくあります。しかし、SQL Anywhere カタログ内部の SYSUSERPERM テーブルでは、ユーザ ID とユーザ名のカラムが区別されています。SYSUSERPERM.user_name カラムには、'DBA'、'SYS'、'bcarter' のような英数字によるユニークな識別子の文字列が入りますが、user_id カラムには、1 や 101 のようなユニークな符号なし整数の値が入ります。

SYSUSERPERM.user_id の値は、新しいユーザのための GRANT 文が実行されるときに SQL Anywhere によって割り当てられます。他のカタログ・テーブル内の関連するカラムには、文字列である user_name の値ではなく、この user_id の値が格納されます。たとえば、SYSTABLE.creator のカラムには、各テーブルの所有者を識別するために user_id の数値が格納されます。SQL Anywhere は、このようなユーザ ID とユーザ名の値を操作する方法をいくつか提供しています。

- CURRENT USER は現在接続されている SYSUSERPERM.user_name を返す。たとえば、'DBA' など
- USER_NAME() は CURRENT USER と同じ働きをする
- USER_ID() は現在接続されている SYSUSERPERM.user_id を返す。たとえば、1 が返される
- USER_NAME(1) は数値の user_id を対応する user_name に変換する。たとえば、'DBA' が返される
- USER_ID('DBA') は user_name を対応する数値 user_id に変換する。たとえば、1 が返される
- USER_ID (CURRENT USER) は USER_ID() と同じ働きをする

また、GRANT CONNECT 文を使用して、特定のユーザ ID が持つパスワードの変更または完全な削除ができます。現在の接続で使用されていないユーザ ID のパスワードを変更するには特別なパーミッ

ションが必要です。しかし、現在接続されているユーザIDのパスワードはいつでも変更できます。つまり、他のユーザのパスワードを変更するには「データベース管理者」の権限が必要ですが、自分自身のパスワードはいつでも変更できます。

次のGRANT文の例は、ユーザIDのBを使って確立した接続で使用できます。この文は、ユーザIDのBが持つパスワードを変更します。パスワードを変更しても現在の接続は処理を進められますが、Bを使用する次の接続では、新しいパスワードの入力が必要になります。

```
GRANT CONNECT TO B IDENTIFIED BY ASDFASDF;
```

Tip

既存のユーザIDを無効にするには、REVOKE CONNECTではなく、パスワードを指定しないGRANT CONNECT文を使用します。REVOKE文は、GRANT文とは反対に、ユーザIDから特定の権限を取り消すことができます。ただし、ユーザIDが完全に削除されてしまうため、REVOKE CONNECTの実行には十分な注意が必要です。そのユーザIDがテーブルの所有者になっている場合、そうしたテーブルとテーブル内のデータのすべてがトリガやインデックスと共に消去されます。また、そのユーザIDが所有するプロシージャと関数も消去されます。本書ではREVOKEやDROPのような文は扱わないというルールでしたが、このヒントはその例外です。詳細については、SQL Anywhereオンラインヘルプの REVOKE文を参照してください。

Tip

ユーザIDの自己破壊に注意してください。たとえば、現在の接続におけるユーザIDがBの場合、GRANT CONNECT TO BというIDENTIFIED BY句のない文によって、Bのパスワードは削除されます。以後はBを使って接続できなくなります。このとき、現在の接続は処理を進めることができます。しかし、他のユーザがGRANT文を使用してBのパスワードを定義しないかぎり、それ以降の接続は失敗します。

Tip

組み込みユーザIDの"dbo"は、自分自身の目的では使用しないようにします。SQL Anywhereでは、dboは実際のユーザIDであり、Adaptive Server EnterpriseまたはMicrosoft SQL Serverのdboのように「データベース所有者」を表す特別なエイリアスではありません。そのため、新しいテーブルの所有者としてdboを使いたいかもしれませんが、使わないでください。アプリケーション開発ソフトウェアによっては、特別なシステム・ユーザID"SYS"の使用を制限するのと同じように"dbo"の使用を制限するものがあるからです。

9.10.1 テーブル権限

これまで指定してきたユーザIDは、パーミッションを取得しないかぎり、限られた働きしかできません。特に、パスワードを持たないユーザIDにできることはただ1つ、テーブルと他のデータベース・オブジェクトの所有者になることだけです。そもそも接続ができないため、テーブルの作成はもちろん参照さえ自分だけではできません。

パスワードを持つユーザIDにはもう少しできることがあります。接続の他、ほとんどのシステム・カタログ・テーブルの検索ができます。ただし、他のユーザが所有するテーブルでクエリや他の操作を実行するには、さらにパーミッションが必要です。次のGRANT文の形式を使用して、パーミッションを与えることができます。

```
<grant_table_privileges> ::= GRANT <table_privileges>
                             ON [ <owner_name> "." ] <table_name>
                             TO <user_id_list>
                             [ WITH GRANT OPTION ]
<table_privileges>  ::= ALL
                      | <table_privilege> { "," <table_privilege> }
<table_privilege>  ::= ALTER
                      | DELETE
                      | INDEX [ "(" [ <column_name_list> ] ")" ]
                      | INSERT
                      | REFERENCES [ "(" [ <column_name_list> ] ")" ]
                      | SELECT [ "(" [ <column_name_list> ] ")" ]
                      | UPDATE [ "(" [ <column_name_list> ] ")" ]
<owner_name>       ::= <identifier>
<table_name>       ::= <identifier>
<column_name_list> ::= <column_name> { "," <column_name> }
<column_name>      ::= <identifier>
```

次の例に、ユーザIDのDBAを使った接続によって作成と初期化を行ったテーブルと、ユーザIDのAを作成するGRANT CONNECT文を示します。これらのオブジェクトは以降のさまざまな例の出発点になります。

```
CREATE TABLE t1 (
   key_1         INTEGER NOT NULL,
   key_2         INTEGER NOT NULL,
   non_key_1     VARCHAR ( 100 ) NOT NULL,
   non_key_2     VARCHAR ( 100 ) NOT NULL,
   PRIMARY KEY ( key_1, key_2 ) );

INSERT t1 VALUES ( 1, 1, 'xxx', 'yyy' );
INSERT t1 VALUES ( 1, 2, 'xxx', 'yyy' );
GRANT CONNECT TO A IDENTIFIED BY SQL;
```

Note

本書では、「ユーザ ID の X を使って確立した接続で実行される」の省略形として「ユーザ ID の X によって実行される」と「X によって実行される」といった表現を用いることがあります。

表 9-3 は、GRANT 文と実行が許される SQL 文の例です。最初のカラムはそれぞれの例に与えた番号です。2 番目のカラムは、ユーザ ID の DBA によって実行される GRANT 文で、それぞれテーブル DBA.t1 に対する特定の操作を実行するパーミッションをユーザ ID の A に与えるものです。3 番目のカラムは、対応する GRANT 文によってユーザ ID の A が実行できるようになる SQL 文の例です。

表 9-3　GRANT 文の例

#	権限を付与する GRANT 文	実行可能になる SQL 文
1.	GRANT ALTER ON t1 TO A;	ALTER TABLE DBA.t1 ADD non_key_3 VARCHAR (100) DEFAULT '';
2.	GRANT DELETE ON t1 TO A;	DELETE DBA.t1;

3.	GRANT SELECT (key_1, key_2)
	  ON t1 TO A;
	GRANT DELETE ON t1 TO A;
		DELETE DBA.t1
		  WHERE key_1 = 1
		  AND key_2 = 1;

4.	GRANT INSERT ON t1 TO A;	INSERT DBA.t1
		  (key_1, key_2, non_key_1, non_key_2)
		VALUES
		  (1, 3, 'aaa', 'bbb');

5.	GRANT REFERENCES ON t1 TO A;	CREATE INDEX x1 ON DBA.t1 (non_key_1);

6.	GRANT SELECT (non_key_2)
	  ON t1 TO A;
		SELECT non_key_2
		  FROM DBA.t1;

7.	GRANT UPDATE (non_key_2)
	  ON t1 TO A;
		UPDATE DBA.t1
		  SET non_key_2 = 'ppp';

表 9-3 の例 1 は、DBA.t1 のスキーマ変更を行う ALTER TABLE 文の実行をユーザ ID の A に許可します。この権限は、影響力のより小さい他の操作に拡張されない特別な権限です。たとえば、テーブルに対して ALTER を実行するパーミッションには、ローに対する UPDATE はもちろん、データに対する SELECT を実行するパーミッションも付随しません。

例 2 は、テーブルに対する DELETE の実行に必要な、DELETE を行うパーミッションです。

例 3 は、DELETE を行うパーミッションでは、単一ローに対する DELETE が実行できないことを示しています。この場合は DELETE 文の WHERE 句でカラム t1.key_1 と t1.key_2 を指定するために、これらのカラムの SELECT を行うパーミッションも必要です。

例 4 は、ローの INSERT を実行するには、INSERT パーミッションが必要なことを示しています。普通は、ローの SELECT を実行できないユーザ ID が、ローの INSERT を実行できることはまずありませんが、ありえないわけではありません。

例 5 は、DBA.t1 上にインデックスを作成するには、REFERENCES または INDEX の権限が必要な

ことを示しています。ALTER 権限と同じように、REFERENCES はその権限だけを有する特異な権限です。

例 6 は、テーブル内のカラムの一部またはすべてに対して SELECT を実行するためのパーミッションで、とてもよく使われます。WHERE と FROM をはじめとする句や select リストにおいてカラムを指定するには、このパーミッションも必要です。GRANT SELECT 文において column name リストを省略した場合、または、空の column name リスト "()" を指定した場合、すべてのカラムに対して SELECT を行うパーミッションが要求されたものとみなされます。

表 9-3 の例 7 は、UPDATE 文の SET 句においてカラムの一部またはすべてを指定するには、UPDATE パーミッションが必要なことを示しています。

Note

コンテキストによっては、CREATE TRIGGER、LOAD TABLE、UNLOAD SELECT、UNLOAD TABLE、一部の形式の ALTER TABLE などの特定の SQL 文を実行するために権限のさまざまな組み合わせが必要になることがあります。たとえば、ALTER TABLE を使用して外部キー関係を作成するには、子テーブルに対する ALTER 権限と親テーブルに対する REFERENCES 権限の両方が必要です。詳細については、SQL Anywhere オンラインヘルプのそれぞれの文の説明を参照してください。

別々の権限を 1 つの GRANT 文の中で結合できます。次の例は、DBA.t1 のローに対する選択と削除だけをユーザ ID の A に許可し、それ以外の操作は許可しません。

```
GRANT SELECT, DELETE ON DBA.t1 TO A;
```

キーワード ALL はすべての権限を表す省略形です。たとえば、次の 2 つの GRANT 文は同じ働きをします。

```
GRANT ALL ON DBA.t1 TO A;
GRANT ALTER, DELETE, INSERT, REFERENCES, SELECT, UPDATE ON DBA.t1 TO A;
```

WITH GRANT OPTION 句を使用すると、ユーザ ID は指定した権限を他のユーザ ID に付与できるようになります。たとえば、ユーザ ID の DBA は次の文を実行することによって、ユーザ ID の A には ALTER、DELETE、INSERT、REFERENCES、SELECT、UPDATE のすべてをテーブル DBA.t1 に対して実行する権限が与えられ、さらにこれらの権限を他のユーザ ID に付与するパーミッションも与えられます。

```
GRANT ALL ON DBA.t1 TO A WITH GRANT OPTION;
```

これでユーザ ID の A は、以下の文を実行することによって、DBA.t1 テーブルに対する SELECT 権限をユーザ ID の E に付与できます。SELECT 権限は A に付与された権限のサブセットであることに注

意してください。また、この文にはWITH GRANT OPTIONを追加できることにも注意してください。仮に、この句が含まれていた場合は、ユーザIDのEもまた他のユーザIDにSELECT権限を付与できることになります。

```
GRANT SELECT ON DBA.t1 TO E;
```

Note

所有権を持っていればパーミッションを付与できます。たとえば、ユーザIDのAをテーブルt1の所有者にすると、誰がそのテーブルを作成したかに関係なく、そのテーブルに対してあらゆる操作を実行できるパーミッションと、そのテーブルに対する権限を他のユーザIDに付与できるパーミッションをAは取得します。これは、GRANT ALL ON A.t1 to A WITH GRANT OPTIONという文を実行した場合とおよそ同じ結果です。ただし、まったく同じではありません。権限とパーミッションの違いについては「9.10.5　GRANT GROUP」で説明します。

9.10.2　ビュー権限

GRANT文の次の形式を使用すると、テーブルのときと同じようにビューに関する権限を付与できます。

```
<grant_view_privileges> ::= GRANT <view_privileges>
                              ON [ <owner_name> "." ] <view_name>
                              TO <user_id_list>
                              [ WITH GRANT OPTION ]
<view_privileges>       ::= ALL
                          | <view_privilege> { "," <view_privilege> }
<view_privilege>        ::= DELETE
                          | INSERT
                          | SELECT
                          | UPDATE [ <column_name_list> ]
<view_name>             ::= <identifier>
```

基本となるテーブルの代わりにビューに対する権限を付与することによって、操作の許可についてのより高度な制御が可能になります。具体的には、ビューを使用すると、GRANT文が提供するカラム・レベルの権限に加えてロー・レベルの権限も付与できます。

以下に、ユーザIDのDBAを使用して作成と初期化を行ったビューの例と、ユーザIDのBを作成するGRANT CONNECT文を示します。ビューv1は以前の例で示したテーブルt1に基づいており、t1.key_2 = 1を満たすローを一致させるWHERE句を指定しています。このビューv1とユーザIDのBは、以降の例の出発点になります。

```
CREATE VIEW v1 AS
SELECT *
  FROM t1
 WHERE key_2 = 1
  WITH CHECK OPTION;

GRANT CONNECT TO B IDENTIFIED BY SQL;
```

ビューの作成の詳細については、「3.23　CREATE VIEW」を参照してください。

表 9-4 は、ビュー DBA.v1 に関連する GRANT 文の例です。2 番目のカラムはユーザ ID の DBA によって実行される GRANT 文です。3 番目のカラムはユーザ ID の B が実行できるようになる SQL 文です。

表 9-4　ビューを伴う GRANT 文の例

#	権限を付与する GRANT 文	実行可能になる SQL 文
1.	GRANT DELETE ON v1 TO B;	DELETE DBA.v1;
2.	GRANT INSERT ON v1 TO B;	INSERT DBA.v1 (key_1, key_2, non_key_1, non_key_2) VALUES (2, 1, 'aaa', 'bbb');
3.	GRANT SELECT ON v1 TO B;	SELECT * FROM DBA.v1;
4.	GRANT UPDATE (non_key_1) ON v1 TO B;	UPDATE DBA.v1 SET non_key_1 = 'zzz';
5.	GRANT UPDATE (non_key_1) ON t1 TO B;	UPDATE DBA.v1 SET non_key_1 = 'zzz';

表 9-4 の例 1 は、ローに対する DELETE をビューから実行するには DELETE 権限が必要なことを示しています。この場合、この DELETE 文は、ビューの WHERE 句による指定どおり、基本となるテーブル DBA.t1 において key_2 = 1 を満たすローだけを削除します。

例 2 は、ビュー内のローに対して INSERT を実行するには INSERT を行うパーミッションが必要なことを示しています。カラム key_2 に対して指定できる値は 1 だけです。ビュー定義の WITH CHECK OPTION によって、このビューに挿入される新しいローは WHERE 句を満たさなければならないと指定されているためです。

例 3 では、SELECT 権限によって、テーブル内のすべてのカラムに対する SELECT を実行するパーミッションが与えられています。テーブルの SELECT 権限とは異なり、ビューの場合は column name リストを指定できません。この SELECT 権限は、SELECT、UPDATE をはじめとする文の WHERE、FROM といった句においてカラムを指定するためにも必要です。

例 4 は、UPDATE 文の SET 句においてカラムの一部またはすべてを指定するには、UPDATE パーミッションが必要なことを示しています。GRANT UPDATE 文で column name リストを省略した場合、または空の column name リスト "()" を指定した場合、すべてのカラムに対する UPDATE パーミッションが要求されたものとみなされます。この場合、この UPDATE 文は、基本となるテーブル DBA.t1 に

おいて key_2 = 1 を満たすローだけを更新します。ビューの WHERE 句によってそう指定されており、ビュー定義に WITH CHECK OPTION 句が含まれているためです。

表 9-4 の例 5 は、基本となるテーブルを使用するためのパーミッションが、そのテーブル上で定義されたビューに拡張されることを示しています。このことは、基本となるテーブルを使用するパーミッションがすでに付与されていれば、ビューを使用するパーミッションについて心配する必要がないことを意味します。しかし、この逆は成立しません。たとえば、例 4 に示した、ビュー DBA.v1 に関する GRANT UPDATE では、ユーザ ID の B は UPDATE 文においてテーブル DBA.t1 の明示的な指定ができません。

キーワード ALL は、ビューに適用される権限のすべてを表す省略形です。また、WITH GRANT OPTION は、ビューに対してもテーブル権限のときと同様の働きをします。

9.10.3 実行権限

プロシージャと関数を実行するパーミッションを与えるには、GRANT 文の以下の形式を使用します。

```
<grant_execute_privilege>    ::= GRANT EXECUTE ON [ <owner_name> "." ]
                                    <procedure_name>
                                    TO <user_id_list>
                                 GRANT EXECUTE ON [ <owner_name> "." ]
                                    <user_defined_function_name>
                                    TO <user_id_list>
<procedure_name>             ::=「第 8 章　パッケージ」の <procedure_name> を参照
<user_defined_function_name> ::=「第 8 章　パッケージ」の <user_defined_function_name> を参照
```

次に示すのは、ユーザ ID の DBA によって作成されるプロシージャの例と、このプロシージャの呼び出しをユーザ ID の B に許可する GRANT EXECUTE 文です。

```
CREATE PROCEDURE p1()
BEGIN
   SELECT *
     FROM t1
    WHERE key_2 = 1;
END;

GRANT CONNECT TO C IDENTIFIED BY SQL;
GRANT EXECUTE ON p1 TO C;
```

プロシージャまたは関数内の SQL 文は、そのプロシージャまたは関数を呼び出したユーザ ID のパーミッションではなく、それらの所有者であるユーザ ID のパーミッションで実行されます。これは混乱を生じさせる制限ではなく、特定のユーザ ID が実行できる操作に対して高度な制御が行えるようにする重要な機能です。

たとえば、上記のGRANT EXECUTE文は、テーブル全体についてのSELECTパーミッションを付与することなく、テーブルt1からローのサブセットを取得する能力をユーザIDのCに与えます。ユーザIDのCはp1のCALLを実行できるため、以下に示す最初の文を実行できます。また、DBAはテーブルDBA.t1に対するSELECTを実行できます。しかし、CはDBA.t1に対して直接SELECTを実行することを許可されていないため、2番目の文は失敗します。

```
CALL DBA.p1();        -- 成功
SELECT * FROM DBA.t1; -- 失敗
```

ユーザIDのCは、次の文も実行できます。この文はSELECT文として書けますが、パーミッションの決定という目的では、FROM句におけるプロシージャ参照はCALL文にしても同じことになります。

```
SELECT * FROM DBA.p1(); -- 成功
```

セキュリティを考慮した環境では、INSERT、UPDATE、DELETEのすべての操作がストアド・プロシージャ内に置かれ、GRANT EXECUTE権限がユーザIDに与えるパーミッションでしかない場合があります。こうしたストアド・プロシージャは、あらゆる追加の編集と監査の機能を実行できます。この方法を使うと、ユーザに対しては必要な変更を許しながら、Excelのような汎用ツールによる特定の更新を止めさせることができます。

トリガ内のSQL文は、対応するテーブルの所有者になっているユーザIDのパーミッションで実行されます。別々のパーミッションが必要な場合の代替案として、必要なパーミッションを持つユーザIDが所有するプロシージャにトリガの内容を移し、このプロシージャをトリガから呼び出す方法があります。

Note

プロシージャ、関数、またはトリガの内部にある文は、その所有者の権限ではなく、その所有者のパーミッションで実行されます。パーミッションと権限の違いについては、「9.10.5　GRANT GROUP」で説明します。

イベント内のSQL文は、そのイベントの所有者のパーミッションで実行されます。イベントの場合、所有者は作成者と同じであり、イベント起動時に所有者は無制限のDBA権限を持っていなければなりません。実際、イベントはどんなことも可能です。DBA権限については次の項で説明します。

9.10.4　特別権限

次の3種類の特別権限が使用できます。

- RESOURCE 権限は、使用されるユーザ ID に対して、そのユーザ ID 自身が所有することになるテーブルやデータベース・オブジェクトの作成を許可する
- DBA 権限（ユーザ ID の DBA と混同しないこと）は、ユーザ ID に対して SQL Anywhere がサポートするあらゆる操作の実行を許可する
- REMOTE DBA 権限は、一定のコンテキストの範囲内でのみ DBA 権限と同じ権限を持つ

次に示すのは、RESOURCE 権限を付与する GRANT 文の形式です。

```
<grant_resource_privilege> ::= GRANT RESOURCE TO <user_id_list>
```

RESOURCE 権限は、所有者のユーザ ID が明示的または暗黙的に指定されているテーブル、ビュー、プロシージャ、関数に対する CREATE と DROP の実行をそれらの所有者であるユーザに許可します。つまり、RESOURCE 権限を持つユーザ ID はそれ自身のテーブルを作成できます。

前に述べたように、所有権を持っていればパーミッションを取得していることになります。たとえば、ユーザ ID の A が RESOURCE 権限を使用してテーブル A.t1 を作成すると、A はそのテーブルに対してあらゆる操作を実行できるパーミッションと共に、そのテーブルに対する権限を他のユーザ ID に付与できるパーミッションも取得します。こうした追加のパーミッションは、権限として明示的に GRANT を実行しなくても自動的にユーザ ID の A に与えられます。パーミッションと権限はわずかに異なりますが、その詳細については、「9.10.5　GRANT GROUP」で説明します。

RESOURCE 権限は、所有者として他のユーザ ID が指定されているデータベース・オブジェクトの作成は許可しません。このためには DBA の特別権限が必要です。表 9-5 に、RESOURCE、DBA、各種 CREATE 文の実行に必要な最小権限の違いを示します。

表 9-5　CREATE 文の実行に必要な権限

CREATE 文	必要な最小の権限
CREATE DATABASE	DBA; 例外についてはオンラインヘルプを参照
CREATE DATATYPE	RESOURCE
CREATE DBSPACE	DBA
CREATE EVENT	DBA; TRIGGER_EVENT も必要
CREATE EXISTING TABLE	RESOURCE
CREATE EXTERNLOGIN	DBA; 例外についてはオンラインヘルプを参照
CREATE FUNCTION	RESOURCE
CREATE INDEX	REFERENCES
CREATE MESSAGE	RESOURCE
CREATE PROCEDURE	RESOURCE
CREATE PUBLICATION	DBA
CREATE SERVER	DBA
CREATE SERVICE	DBA

CREATE 文	必要な最小の権限
CREATE STATISTICS	DBA
CREATE SUBSCRIPTION	DBA
CREATE SYNCHRONIZATION USER	DBA
CREATE TABLE	RESOURCE
CREATE TRIGGER	ALTER
CREATE VARIABLE	任意の接続で実行可能
CREATE VIEW	RESOURCE と SELECT

次に、DBA 権限を付与する方法を示します。

```
<grant_dba_privilege> ::= GRANT DBA TO <user_id_list>
```

DBA よりも強力な権限はありません。DBA は、新しい SQL Anywhere データベースが作成されたときに、同じく DBA と呼ばれるユーザ ID が保持する権限です。セキュリティを考慮しない場合、DBA はあらゆる状況に対処できる唯一の権限、唯一のユーザ ID です。

Tip

DBA 権限を持つ予備のユーザ ID を少なくとも 1 つは作成しておきます。こうした予備のユーザ ID は、DBA 権限の不慮の損失に対するバックアップとしての保護を提供します。DBA 権限を使用できるユーザ ID は、常に少なくとも 1 つ存在するようにします。

セキュリティへの配慮はしなければならないが、一般ユーザが使うユーザ ID から、通常ならば DBA 権限が要求される操作を実行しなければならない場合は、REMOTE DBA の特別権限が役に立つでしょう。次に示すのは、REMOTE DBA 権限を付与するための GRANT 文の形式です。

```
<grant_remote_dba_privilege> ::= GRANT REMOTE DBA TO <user_id_list>
```

REMOTE DBA 権限を持つユーザ ID を使用すると、次のプログラムからデータベースに接続できます。

- **dbmlsync.exe** —— Mobile Link 同期クライアント
- **dbvalid.exe** ——データベース検証ユーティリティ
- **dbbackup.exe** ——データベース・バックアップ・ユーティリティ

REMOTE DBA 権限を持つユーザ ID を用いて確立した接続では、DROP CONNECTION、VALIDATE TABLE、VALIDATE INDEX の各文を実行できます。

次に示すのは、データベースに接続できる他は REMOTE DBA 権限しか持たないユーザ ID の例です。

```
GRANT CONNECT TO SYNCH IDENTIFIED BY SQL;
GRANT REMOTE DBA TO SYNCH;
```

ユーザ ID の SYNCH を使っても、アプリケーション・プログラムまたは ISQL からは、一切の変更はもちろん、データの SELECT も実行できません。しかし、次のコマンド・ラインに示すように Mobile Link 同期クライアントの実行はできます。

```
"%ASANY9%¥win32¥dbmlsync.exe" -c "ENG=remo;DBN=remo;UID=SYNCH;PWD=SQL"
```

9.10.5 GRANT GROUP

GROUP 権限を使用すると 2 つの問題を解決できます。まず、多数のユーザ ID に対して権限の組み合わせを割り当てるのが容易になります。また、テーブルとビューの参照時に所有者の名前を明示的に指定する必要がなくなります。

SQL Anywhere のユーザ・グループは、GROUP 権限を持つだけのユーザ ID です。次に示すのは、通常のユーザ ID をグループにする GRANT 文の形式です。

```
<create_user_group> ::= GRANT GROUP TO <group_name_list>
<group_name_list>   ::= <user_id_list> -- グループでもあるユーザ
```

次に示すのは、他のユーザ ID をグループに加える GRANT 文の構文です。

```
<add_user_to_group> ::= GRANT MEMBERSHIP IN GROUP <group_name_list>
                            TO <user_id_list>
```

次の権限は、グループ・メンバシップを通して継承されます。

- テーブルに対する ALTER、DELETE、INSERT、REFERENCES、SELECT、UPDATE の各権限
- ビューに対する DELETE、INSERT、SELECT、UPDATE の各権限
- プロシージャと関数に対する EXECUTE 権限
- グループ内の MEMBERSHIP

次の権限は、グループ・メンバシップを通して継承されません。

- RESOURCE
- DBA
- REMOTE DBA

- GROUP
- テーブルとビューに対する権限の WITH GRANT OPTION

グループ MEMBERSHIP は継承されますが、GROUP 権限は継承されないことに注意してください。たとえば、ユーザ ID の A が CHILD グループのメンバであり、CHILD は PARENT グループのメンバである場合、A は PARENT グループのメンバでもあるので PARENT の権限を継承できます。しかし、ユーザ ID の A 自身はグループではないので、CHILD または PARENT の GROUP 権限は継承しません。

また、WITH GRANT OPTION 句によって与えられた権限は、グループ・メンバシップを通して継承されません。たとえば、WITH GRANT OPTION 句を用いてユーザ・グループ G にある権限を付与したからといって、そのグループのメンバは自動的にその権限を他のユーザ ID に付与できることにはなりません。

次の例は、DBA ユーザ ID によって実行される一連の GRANT 文です。これらの文はメンバ数が 10 のグループを設定します。

```
GRANT CONNECT TO READ_ONLY;
GRANT GROUP TO READ_ONLY;
GRANT SELECT ON t1 TO READ_ONLY;
GRANT SELECT ON t2 TO READ_ONLY;
GRANT SELECT ON t3 TO READ_ONLY;
GRANT MEMBERSHIP IN GROUP READ_ONLY TO A, B, C, D, E, F, G, H, I, J;
GRANT SELECT ON t4 TO READ_ONLY;
GRANT SELECT ON t5 TO READ_ONLY;
GRANT SELECT ON t6 TO READ_ONLY;
```

以下に、この 9 つの GRANT 文がどのような働きをするかを示します。

- GRANT CONNECT 文は、READ_ONLY というユーザ ID を作成する。このユーザ ID はパスワードを持たないため、データベースの接続には使えない。これはグループとして使用されるだけのユーザ ID にもあてはまる一般則である
- GRANT GROUP 文は、READ_ONLY ユーザ ID をグループとして使用することを許可する。つまり、READ_ONLY が GRANT MEMBERSHIP 文で使えるようになる
- 次の 3 つの GRANT SELECT 文は、DBA. t 1 から DBA.t3 までのテーブルに対する SELECT 権限を READ_ONLY に与える。READ_ONLY は接続に使えないため、実際にはどのような SELECT の実行にも使用できない。この SELECT 権限は、他のユーザ ID にグループ・メンバシップを通して権限を渡せるだけのものである
- GRANT MEMBERSHIP 文は、A から J までの 10 個のユーザ ID を READ_ONLY というグループに加える。これらのユーザ ID は、READ_ONLY に付与されていた SELECT 権限を自動的に継承する
- 最後の 3 つの GRANT SELECT 文は、メンバ追加の前後どちらでもこのグループに権限を付与で

きることを示す。これらの権限はすぐにすべてのグループメンバで有効になる

これで10個のユーザIDすべてが、次の文を実行するパーミッションを取得しました。

```
SELECT * FROM DBA.t1;
```

ユーザ・グループの主な利点は、1つのユーザID（グループ）に複数の権限を付与するほうが、個々のユーザID（メンバ）に権限を付与するより容易であることです。

ユーザ・グループの別の利点は、権限の組み合わせを容易に割り当てられることです。たとえば、DBAによって実行される次のGRANT文はDATA_ENTRYという2つめのグループを設定します。このグループのメンバは、テーブルDBA. t 1とDBA. t 3に対するINSERTとUPDATEの各操作を実行できます。

```
GRANT CONNECT TO DATA_ENTRY;
GRANT GROUP TO DATA_ENTRY;
GRANT INSERT, UPDATE ON t1 TO DATA_ENTRY;
GRANT INSERT, UPDATE ON t3 TO DATA_ENTRY;
GRANT MEMBERSHIP IN GROUP DATA_ENTRY TO A, E, H;
```

これでユーザIDのA、E、HはREAD_ONLYとDATA_ENTRYの両グループのメンバになったため、SELECT、INSERT、UPDATEの3つの操作すべてをテーブルDBA.t1とDBA.t3に対して実行できます。ただし、A、E、Hは他のテーブルに対してはSELECT操作しかできず、その他のユーザIDはどのテーブルに対してもSELECTしか実行できません。

ユーザ・グループのもう1つの利点は、テーブルとビューの参照時に所有者の名前を指定する必要がないことです。たとえば、上記のGRANT文でテーブルDBA.t1に対するSELECTのパーミッションをユーザIDのAに与えたにもかかわらず、次の文をAが実行しようとすると、「テーブルt1は見つかりませんでした」というエラーが起こります。

```
SELECT * FROM t1;
```

しかし、次のGRANT文をユーザIDのDBAが実行すると、もはやユーザIDのAはテーブル名"t1"を所有者名"DBA"で修飾する必要がなくなります。

```
GRANT GROUP TO DBA;
GRANT MEMBERSHIP IN GROUP DBA TO A;
```

これでユーザIDのAは次のSELECTを実行できます。

```
SELECT * FROM t1;
```

Note

前に説明したとおり、DBA 権限はグループ・メンバシップを通して継承されません。

SQL 文が実行されるとき、SQL Anywhere は、所有者の名前を省略したテーブルまたはビューへの参照のそれぞれについて完全に修飾された名前を判別します。以下に、この作業を実行するステップを示します。

❶ 一致する名前を持つローカル・テンポラリ・テーブルが存在する場合は、検索を終了します。そのテンポラリ・テーブルが使用すべきものです。
❷ 一致する名前を持つ、グローバルなテンポラリ・テーブルまたは永続的なテーブルもしくはビューのうち、現在のユーザ ID が所有するものが存在する場合、検索を終了します。そのテーブルまたはビューが使用すべきものです。
❸ 一致する名前を持つテーブルまたはビューのうち、現在のユーザ ID がメンバになっているグループが所有するものが1つしか存在しない場合、検索を終了します。そのテーブルまたはビューが使用すべきものです。この検索は、現在のユーザ ID がメンバになっているグループの間に存在し得るいかなる階層関係からも影響を受けません。この検索ではすべてのグループが同等に扱われます。
❹ ステップ3で、一致する名前を持つテーブルまたはビューが2つ以上見つかり、そのそれぞれが現在のユーザ ID がメンバになっている別のグループによって所有されている場合、「テーブル名 'x' はあいまいです」というエラーになります。
❺ 一致する名前を持つテーブルまたはビューが存在しない場合、「テーブル 'x' は見つかりませんでした」というエラーになります。

この章では、「パーミッション」と「権限」という言葉をほとんど同じ意味で使用しています。しかし、両者には以下に説明するような違いがあります。

- ユーザ ID が操作を実行するには、その操作を実行するパーミッションが必要である
- 権限は GRANT 文を介して明示的に付与されるか、グループ・メンバシップを通して継承される
- 権限には1つ以上のパーミッションが含まれる。たとえば、SELECT 権限を有するユーザ ID は、テーブルに対する SELELCT パーミッションを持ち、DBA 権限を有するユーザ ID はどのような操作も行えるパーミッションを持つ
- 上記の項目の逆は成立しない。つまり、パーミッションを持つことは必ずしも対応する権限を持つことにはならない。たとえば、テーブルの所有者はそのテーブルに対する SELECT を実行するパーミッションを自動的に取得するが、明示的に付与されないかぎり、その SELECT 権限は持たない。
- 権限は他の権限に包含されない。たとえば、DBA 権限を有するユーザ ID は、たとえ任意のテーブルに対する SELECT のパーミッションを持っていたとしても、任意のテーブルに対する

SELECT 権限を自動的に持つことはない

- 権限は、パーミッションと異なり、グループ・メンバシップを通して継承される

最後の項目は、ユーザ ID を DBA グループのメンバにしても安全な理由を示しています。DBA ユーザ ID は、DBA 以外のどのような権限も持たず、DBA 権限はグループ・メンバシップを通して継承されません。したがって、実際には、DBA グループのメンバはそれ以上のどのような権限も継承しません。DBA グループのメンバが持つのは、DBA が所有するテーブルを参照するときに所有者の名前を省略できる能力だけです。

Section

9.11 ロギングとリカバリ

SQL Anywhere は、システム障害とメディア障害によって生じるデータの損失と矛盾からリカバリを行うために、効率的で信頼性の高いメカニズムを提供しています。この項では、データベース・エンジンに組み込まれている、システム障害によって生じる問題を処理するための基本的なリカバリ・メカニズムについて説明します。次の 2 つの項では、この基本的なメカニズムを拡張してファイルの失敗に対処できるようにした、バックアップとリストアの各プロシージャについて説明します。

データベースの稼働中、データは 7 つの異なる場所に格納されます。

- メモリ内キャッシュ
- 他の dbspace ファイルを含む、ディスク上のデータベース・ファイル
- ディスク上のトランザクション・ログ・ファイル
- オプションとしてのディスク上のミラー・ログ・ファイル
- データベース・ファイル内のチェックポイント・ログ
- キャッシュとデータベース・ファイル内のロールバック・ログ
- ディスク上のテンポラリ・ファイル

これらのデータ格納場所は基本的なリカバリ・メカニズムにとって非常に重要です。以下では、それぞれについて説明します。

「cache（キャッシュ）」には、データベース・ファイルからメモリに読み込まれたテーブルとインデックスの各ページが格納されます。キャッシュは常に最新の状態に保たれます。具体的には、SQL 文によるデータベースの論理的な変更は、コミットされたものもされていないものも含めて、このキャッシュ内のテーブルとインデックスのページにすぐに反映されます。

「database files（データベース・ファイル）」には、テーブルとインデックスのページが格納されます。そのうち SYSTEM dbspace データベース・ファイルには、チェックポイント・ログとロールバック・ロ

グの各ページも格納されます。変更されたテーブルとインデックスのページは、チェックポイント前およびチェックポイント中に物理的なデータベース・ファイルに書き込まれます。SQL 文によって論理的なデータベースの変更が最初に行われたとき、または、COMMIT 文が実行されたときには、これらの変更は必ずしもこうした物理的なデータベース・ファイルには書き込まれません。実際、COMMIT 操作と物理的なデータベース・ファイル内のデータの間には直接の対応はありません。コミットされていない変更がデータベース・ファイルに書き込まれることもあれば、コミットされた変更がすぐには書き込まれないこともあります。

データベース・ファイルはチェックポイントの後に最新の状態になります。具体的には、物理的なデータベース・ファイル内のすべてのテーブルとインデックスのページは、チェックポイント完了時点ではいつでも、キャッシュ内のデータに対して最新の状態であることが保証されます。

物理的なデータベース・ファイルにはコミットされていない変更が含まれる可能性があることに注意してください。コミットされていないデータは独立性レベル 0 で実行中の接続からは参照できますが、独立性レベル 1 以上で実行中の接続からは参照できません。独立性レベルの詳細については、「9.7　ブロックと独立性レベル」を参照してください。

「transaction log file（トランザクション・ログ・ファイル）」は「redo log（再実行ログ）」とも呼ばれ、このログ・ファイルの作成時点以降にデータベースに加えられた論理的な変更の逐次的な記録を持っています。コミットされた変更とコミットされていない変更の両方が、物理的なトランザクション・ログ・ファイルに書き込まれる可能性があります。また、これらの変更は、対応する接続で COMMIT が実行される前に書き込まれる場合があります。COMMIT と CHECKPOINT の操作も、他の変更と同様にトランザクション・ログに記録されます。

Note

ここでの説明では、トランザクション・ログが使用されることを想定しています。トランザクション・ログは信頼性だけでなくパフォーマンスも向上させます。また、ここでの説明では、DELAYED_COMMIT データベース・オプションがデフォルトの 'OFF' のままであることを想定しています。この設定によっても信頼性が確保されます。

トランザクション・ログ・ファイルは COMMIT の実行後に最新の状態になります。具体的には、特定のトランザクションが正常に完了する COMMIT 終了の後、接続に制御が返されるまでに、そのトランザクションによる変更が物理的なトランザクション・ログに書き込まれます。実際、すべての変更は、コミットされたものもそうでないものも、接続のどれかが COMMIT を実行するたびにトランザクション・ログに書き込まれることが保証されています。

トランザクション・ログは、この項で後に述べるリカバリ処理の 2 番目のステップで使用されます。

トランザクション・ログには完全なリカバリに必要な最小限のデータが格納されます。空のデータベースを初期化してトランザクション・ログを適用すれば、時間はかかるかもしれませんが、データベース全体を最初から再構築できます。

「mirror log file（ミラー・ログ・ファイル）」は、オプションとして使用される、トランザクション・ログ・ファイルのコピーです。ミラー・ログ・ファイルを指定すると、このファイルはトランザクション・ログ・ファイルと同一内容の冗長なコピーとして自動的に管理されます。ミラー・ログが自動的にリカバリ処理で使用されることはありませんが、トランザクション・ログの消失または破壊時には、その代替ログとしてリカバリ処理で使用できます。

「checkpoint log（チェックポイント・ログ）」は、「page level undo log（ページ・レベル取り消しログ）」とも呼ばれ、前のチェックポイント以降に変更されたデータベース・テーブル、インデックス、ロールバック・ログ・ページの更新前イメージまたは変更前コピーの逐次的な記録を格納します。対応するテーブル、インデックス、またはロールバック・ログ・ページが変更されて前のチェックポイント以降で初めてファイルに書き込まれるちょうど前に、それぞれのチェックポイント・ログ・ページが物理的なデータベース・ファイルに書き込まれます。各チェックポイント・ログは、たとえ対応するページが2回以上変更されても一度の書き込みしか必要としません。必要とされる変更されていないページのコピーは1つだけだからです。

チェックポイント・ログはいつでも最新の状態にあります。具体的には、チェックポイント・ログ・ページはデータベース・ファイルにしか存在せず、対応するテーブルとインデックスのページが変更されてデータベース・ファイルに書き込まれるときに作成されるだけです。チェックポイントが設定された直後はチェックポイント・ログが空の状態であり、空でないのはチェックポイントとチェックポイントの合間だけであることに注意してください。

チェックポイント・ログ・ページは、SYSTEM dbspace データベース・ファイルの最後に書き込まれます。ファイルが大きくなるとページがファイルの最後になるように場所を移されます。チェックポイントの設定が正常に完了するとチェックポイント・ログ内のデータは廃棄され、データベースがシャットダウンされるとその領域のリカバリが行われます。

チェックポイント・ログによって、データベース・サーバは、更新されたテーブルとインデックスのページのデータベース・ファイルへの書き込みを遅らさせることが可能になります。つまり、COMMITが実行されるたびに、データベース・ファイル自体への書き込みを行わずに済みます。COMMIT 実行時はいつでも、トランザクション・ログに変更内容が存在することが保証されているため、変更されたページのすべてをデータベースに書き込む前にサーバが実行を停止しても、それらが失われることはありません。データベース・ファイルへのページ書き込みを遅らせることによって、より効率的なディスク操作が可能になり、全体のパフォーマンスが向上します。ただし、チェックポイントとチェックポイントの間のどこかでサーバが実行を停止した場合、データベース・ファイル内のテーブルとインデックスのページが不安定または矛盾のある状態になることがあります。このとき、最新の状態にあるデータとそうでないデータが混在することになります。チェックポイント・ログが存在するのはこのためであり、この項において後で説明するリカバリ処理の最初のステップで使用されます。

「rollback log（ロールバック・ログ）」は、「undo log（取り消しログ）」または「row level undo log（ロー・レベル取り消しログ）」とも呼ばれます。このログは、それぞれの接続がリバース操作の逐次的な記録を保持するために管理されます。リバース操作とは、各接続における1つ前のCOMMIT またはROLLBACK の実行時点以降にその接続がデータベースに加えた論理的な変更を取り消す操作です。

ロールバック・ログ・ページは、テーブルやデータ・ページと一緒にSYSTEM dbspaceデータベース・ファイルに格納されており、その書き込みはチェックポイント前およびチェックポイント中に行われます。ロールバック・ログ・ページの物理的なデータベース・ファイルへの書き込みは、SQL文によるデータベースの最初の論理的な変更時に行われるとはかぎりません。ロールバック・ログ・ページはチェックポイント・ログに含まれていることに注意してください。

ロールバック・ログはチェックポイントの後で最新の状態になります。具体的には、物理的なデータベース・ファイル内にある現在のロールバック・ログ・ページのすべては、チェックポイント完了時はいつでも最新の状態であることが保証されています。

ロールバック・ログ内のデータには、チェックポイントの前後に加えられた変更が含まれている場合があります。チェックポイントと特定のトランザクションの終了とは対応していません。

ロールバック・ログ内のデータは、接続におけるROLLBACK実行時に、変更をロールバックする通常の操作の途中で使用され、その後廃棄されます。接続においてCOMMITが実行されたときにも廃棄されます。

また、ロールバック・ログは、リカバリ処理の3番目のステップでも使用されます。

「temporary file（テンポラリ・ファイル）」は、テンポラリ・テーブル・データとテンポラリ・テーブル上のインデックスとを含むテンポラリ・データのために使用されます。このファイルは決して、永続的なテーブルもしくは、チェックポイント・ログ・ページ、ロールバック・ログ・ページまたはインデックス・ページを保持するためには使用されません。

テンポラリ・テーブルに対する変更内容はトランザクション・ログに記録されません。また、その変更によってチェックポイント・ログになんらかのページが書き込まれることもありません。ただし、テンポラリ・テーブルの作成時にNOT TRANSACTIONAL句が指定されていないかぎり、テンポラリ・テーブルの変更によって、ロールバック・ログにエントリが書き込まれます。

ロールバック・ログに記録されたテンポラリ・テーブルの変更内容は、通常のROLLBACKの処理においてのみ使用され、リカバリ処理において処理されることはありません。つまり、リカバリ処理によってテンポラリ・テーブルは再作成されません。

「recovery process（リカバリ処理）」は、データベースの開始時にデータの矛盾を修復するためにデータベース・エンジンによって実行されます。このリカバリ処理は以下の状況で行われます。

- 最後のチェックポイントが成功せずに終了した後でデータベースを起動するとき、自動的に行われる
- もしバックアップ・コピーが元のデータベースの起動中に行われた場合、つまりオンライン・バックアップの場合、そのバックアップ・コピーを使用してデータベースを起動すると、自動的に行われる。なおオンライン・バックアップの手順に関しては次項で説明する
- トランザクション・ログの適用を要求する-aパラメータを用いてデータベースを起動するとき、明示的に行われる
- トランザクション・ログを使用せずにデータベースの起動を要求する-fパラメータを用いてデータベースを起動するとき、明示的に行われる

以下に示すのは、リカバリ処理の実行時にデータベース・エンジンがコンソール・ウィンドウに表示する内容です。

```
データベース "test9" (E:¥test9¥test9.db) を Sat Feb 21 2004 12:15 に起動しています。
データベースのリカバリ処理中です。
    Sat Feb 21 2004 12:14 での最終チェックポイント
    チェックポイント・ログ ...
    トランザクション・ログ: test9.log...
    ロールバック・ログ ...
    チェックポイントを設定中 ...
Sat Feb 21 2004 12:15 に "test9" (test9.db) のチェックポイントを開始しています。
Sat Feb 21 2004 12:15 に "test9" (test9.db) のチェックポイントを終了しました。
リカバリ処理が完了しました。
```

リカバリ処理は、次の4つのステップで構成されています。

❶ データベースを、成功した1つ前のチェックポイントの状態にリストアするために、チェックポイント・ログ内のページを使用して、対応する変更後のテーブル、インデックス、ロールバック・ログ・ページのすべてを上書きします。この時点では、コミットされていない変更がデータベース内に存在する可能性があること、また、コミットされていないトランザクションのそれぞれについて、前回の成功したチェックポイントから見て最新の状態のロールバック・ログが存在することに注意してください。

❷ 前回の成功したチェックポイント以降にトランザクション・ログ・ファイルに記録された論理的な再実行操作のすべてを適用して、シャットダウンされた時の状態にデータベースをリストアします。この処理によって、前回の成功したチェックポイント後に行われた変更に対応するロールバック・ログの一部も再作成されます。この段階で行われたCOMMITとROLLBACKの操作はいずれも、対応するトランザクションのためのロールバック・ログを消去すること、さらに、コミットされていない変更によって、これらの新しいトランザクションのための新しいロールバック・ログが作成されることに注意してください。

❸ 残りのロールバック・ログに記録されている論理的な取り消し操作のすべてをデータベースに適用して、コミットされていないアクティブなトランザクションがあればすべてロールバックを行い、データベースを一貫性のある正しい状態にします。実行中に長い間コミットされなかったトランザクションの場合、前回の成功したチェックポイント前後からの取り消し操作がロールバック・ログに含まれる可能性があることに注意してください。前回の成功したチェックポイント前の状態からのロールバック・ログの一部は上記のステップ1でリストアされ、前回の成功したチェックポイント後の状態からのロールバック・ログの一部はステップ2で再構築されます。

❹ チェックポイントを実行します。これでデータベースは使用できる状態になります。-aまたは-fパラメータを指定してリカバリ処理を開始した場合、データベース・エンジンはこの時点で自動的にシャットダウンを行った後、-aまたは-fパラメータの指定なしで再起動され、通常のオペレーションを再開できるようになります。

Section

9.12 バックアップ

SQL Anywhere は、データベースのバックアップ・コピーを作成するためのさまざまな機能を提供しています。以降の5つのサブセクションでは、こうした機能を使って次の5種類のバックアップを作成する方法を説明します。

- フル・オフライン・イメージ・バックアップは、データベース停止中にデータベースとトランザクション・ログ・ファイルに対して行われる
- フル・オンライン・イメージ・バックアップは、データベース実行中にデータベースと各種ログ・ファイルに対して行われる
- 差分オンライン・ログ・バックアップは、前回のフル・バックアップ以降に使用されたログ・ファイルに対して行われる
- インクリメンタル・オンライン・ログ・バックアップは、前回のバックアップ以降にのみ使用されたログ・ファイルに対して行われる
- ライブ・ログ・バックアップはトランザクション・ログ・データのリモート・コンピュータ上のファイルへの継続的なコピーを行う

9.12.1 フル・オフライン・イメージ・バックアップ

「full offline image backup（フル・オフライン・イメージ・バックアップ）」は、データベースとトランザクション・ログ・ファイルのファイル・コピーであり、データベースが実行されていない間に行われます。このバックアップ・ファイルは通常のオペレーティング・システムのファイル・コピー・コマンドを使用して作成されます。フル・オフライン・イメージ・バックアップは、データベースのシャットダウンが可能で、データベースとトランザクション・ログ・ファイルの両方に対する完全なコピーを作成するだけの十分な領域がある場合に適しています。

Note

データベースが正常に停止した場合、トランザクション・ログに対するバックアップ・コピーの作成は必ずしも必要ではありません。データベース停止時に行われた最後のチェックポイントによって、データベース・ファイルのバックアップ・コピーがトランザクション・ファイルに記録されたすべてのアクティビティを反映していることが保証されるためです。しかし、データベースが最後のチェックポイントを行わずに停止した場合、完全なリストアを行うにはバックアップ・トランザクション・ログが必要です。トランザクション・ログを含めたバックアップの実施は常に安全であり、またそれが必要なことも多いため、この項では「データベースのみ」のバックアップについては説明しません。

この項の例では、test9.db という名前のデータベース・ファイルを使います。このデータベースはトランザクション・ログ（test9.log）とミラー・ログ（test9.mgl）の両方を使用します。ディスク障害の影響を軽減するために、ミラー・ログは、トランザクション・ログとデータベース・ファイルとは別の物理的なディスク・ドライブに格納されます。

次の Windows コマンド・ラインは、このデータベースを作成してデータベース・サーバを起動するためのものです。

```
"%ASANY9%¥win32¥dbinit.exe" -m C:¥mirror¥test9.mlg -p 4096 test9.db
"%ASANY9%¥win32¥dbsrv9.exe" -x tcpip test9.db
```

-m パラメータは、データベースのミラー・ログ作成を dbinit.exe に指示します。-p パラメータは、db.init に対してデータベースのページ・サイズを 4K に指定します。-x パラメータは、デフォルトの「共有メモリ」つまりローカルな接続プロトコルに加えて、TCP/IP が唯一のクライアント・サーバ間プロトコルとして使用されることを dbsrv9.exe に伝えます。

特に断りがないかぎり、この項に示すコマンド・ラインは、データベース・サーバが置かれるメインまたはプライマリ・コンピュータで実行されるものとします。コマンド実行の開始フォルダは "E:¥test9" であり、ここに現在のデータベースとトランザクション・ログ・ファイルが置かれます。ミラー・ログはメイン・コンピュータの "C:¥mirror" フォルダに、バックアップ・ファイル類はメイン・コンピュータとは別のリモート・コンピュータの "G:¥bkup" フォルダに、それぞれ作られます。G ドライブは別のコンピュータの "remote_test9" フォルダに対応付けられています。"G:¥bkup" のフル・パスは、UNC フォーマットで "¥¥Dell180¥DELL180G¥remote_test9¥bkup" です。

Tip

バックアップ・ファイルは別のコンピュータに置くことができ、また、おそらくそうするのが良いでしょう。ただし、通常の要求処理中にデータベース・エンジンが使用するすべてのファイルは、データベース・エンジンを実行しているのと同じ CPU に対してローカルに付与された物理ドライブ上になければなりません。いかに LAN が高速であっても、データベース、トランザクション・ログ、またはテンポラリ・ファイルを別のコンピュータに置かないようにします。

以下に示すのは、フル・オフライン・イメージ・バックアップを行う Windows バッチ・ファイルの例です。

```
SET CONNECTION="ENG=test9;DBN=test9;UID=dba;PWD=sql"
"%ASANY9%¥win32¥dbisql.exe" -c %CONNECTION% STOP ENGINE test9 UNCONDITIONALLY
RENAME G:¥bkup¥test9.db  old_test9.db
RENAME G:¥bkup¥test9.log old_test9.log
RENAME G:¥bkup¥test9.mlg old_test9.mlg
IF EXIST G:¥bkup¥test9.db  GOTO ERROR
```

```
IF EXIST G:¥bkup¥test9.log GOTO ERROR
IF EXIST G:¥bkup¥test9.mlg GOTO ERROR
COPY test9.db            G:¥bkup¥test9.db
COPY test9.log           G:¥bkup¥test9.log
COPY C:¥mirror¥test9.mlg G:¥bkup¥test9.mlg
ECHO N | COMP test9.db G:¥bkup¥test9.db
IF ERRORLEVEL 1 GOTO ERROR
ECHO N | COMP test9.log G:¥bkup¥test9.log
IF ERRORLEVEL 1 GOTO ERROR
ECHO N | COMP C:¥mirror¥test9.mlg G:¥bkup¥test9.mlg
IF ERRORLEVEL 1 GOTO ERROR
ERASE G:¥bkup¥old_test9.db
ERASE G:¥bkup¥old_test9.log
ERASE G:¥bkup¥old_test9.mlg
"%ASANY9%¥win32¥dbsrv9.exe" -x tcpip test9.db
GOTO END
:ERROR
PAUSE Backup process failed.
:END
```

Windows コマンドのヘルプを参照するには、コマンド・ウィンドウまたは「DOS プロンプト」ウィンドウを開いて HELP と入力します。また、HELP IF のように、その後にコマンド名も入力できます。お使いの Windows のバージョンでこの指定が効かない場合は、IF /? のように /? オプションを試してください。

以下に、このバッチ・ファイルがフル・オフライン・イメージ・バックアップをどのように行うかを示します。

- SET コマンドは、次のコマンドで使用する接続文字列を保持するためのローカル環境変数を定義する
- 2 行目のコマンド・ラインは、ISQL をバッチ・モードで実行し、STOP ENGINE 文によって test9 というデータベース・エンジンを現在接続中のユーザによる切断を待たずに停止させる。STOP ENGINE 文と、ISQL コマンド・ライン上での SQL 文の実行の詳細については、オンラインヘルプを参照
- 3 つの RENAME コマンドは、前回のバックアップ・ファイルがすでに存在する場合に上書きされないようにファイル名を変更する。バックアップ作業がすべてうまく行った場合、これらのファイルは後の処理で削除される
- 3 つの IF EXIST コマンドは、RENAME コマンドによってファイル名が変更されたことを確認する。元のファイル名が 1 つでもまだ使われていればバックアップ処理全体を中止する
- 3 つの COPY コマンドは、.db、.log、.mlg の各ファイルを別のコンピュータ上の bkup というフォ

ルダにコピーする

- その次の6つのコマンドは、元のファイルと新しいバックアップ・コピーとの比較を行う。"ECHO N |"のそれぞれは、「ほかのファイルを比較しますか (Y/N)?」というプロンプトに自動的に答えるために、対応するCOMPコマンドに"N"という文字をパイプする。COMPコマンドはファイルの比較を行う。IF ERRORLEVELコマンドは、COMPコマンドの返り値が1以上の場合にバックアップ処理全体を中止する
- 3つのERASEコマンドは、前に名前を変更した古いバックアップ・ファイルを削除する
- その次のコマンドは、現在のデータベースとトランザクション・ログ・ファイルを使って、もう一度データベース・エンジンを起動する
- 最後の4行は、状況に応じてエラー・メッセージの表示またはその省略を行う

9.12.2 フル・オンライン・イメージ・バックアップ

dbbackup.exeプログラムまたはBACKUP DATABASE文を使用することによって、データベースの実行中にバックアップを行うことが可能です。この項の例には、オンライン・バックアップのためのBACKUP DATABASE文だけを示します。BACKUP DATABASE文は、ISQL、アプリケーション・プログラム、内部でスケジュールされたイベントのそれぞれから実行できるという点でdbbackupよりも柔軟性が高いからです。スケジュールされたイベントの詳細については、「8.12 CREATE EVENT」を参照してください。

前に説明した「オフライン・バックアップ」に対し、BACKUP DATABASE文によって作成されたバックアップを「オンライン・バックアップ」と呼びます。オフライン・バックアップとの重要な違いは、バックアップ処理中にユーザが変更を行う可能性があるため、データベースとトランザクション・ログ・ファイルのオンライン・バックアップ・コピーが一致しない場合があることです。この点は、9.13項で説明するリストア処理に影響を与えます。

BACKUP DATABASE文には2つのフォーマットがあります。1つは.dbと.logの2つのファイル群で別々のイメージ・コピーを作成するために使用します。もう1つは単一の「アーカイブ」ファイルを作成するために使用します。アーカイブ・ファイルを作成するBACKUP DATABASEの形式については、SQL Anywhereオンラインヘルプで説明されています。

以下に示すのは、オンライン・イメージ・ファイルを作成するBACKUP DATABASE文の構文です。

```
<backup_database_to_image_files> ::= BACKUP DATABASE { <image_option> }
<image_option>      ::= DIRECTORY <image_destination> -- 必須
                      | DBFILE ONLY                   -- この2つの句は
                      | TRANSACTION LOG ONLY          -- 相互排他的
                      | TRANSACTION LOG RENAME
                      | TRANSACTION LOG RENAME MATCH  -- TRUNCATEと競合
                      | TRANSACTION LOG TRUNCATE      -- RENAMEを上書き
                      | WAIT BEFORE START             -- 場合によりこの2つの句は
                      | WAIT AFTER END                -- 一緒に使用される
```

```
                    | WITH COMMENT <backup_comment>
<image_destination> ::= 文字列リテラルのパスとフォルダ
<backup_comment>    ::= %ASANY9%¥win32¥backup.sybに書き込む文字列リテラル
```

BACKUP DATABASE文は、バックアップ対象のデータベースにつながった接続において、REMOTE DBAまたはDBA権限を持つユーザIDを使って実行します。さまざまなオプションを使用することで、次のような異なる種類のオンライン・イメージ・バックアップを実行できます。

- フル・オンライン・イメージ・バックアップは、データベースとトランザクション・ファイルのバックアップ・コピーを作成する
- 差分オンライン・ログ・バックアップは、最後のフル・バックアップ以降に作られたトランザクション・ログのエントリをバックアップする
- インクリメンタル・オンライン・ログ・バックアップは、最後のフル・バックアップまたはインクリメンタル・バックアップ以降に作られたトランザクション・ログのエントリをバックアップする

「full online image backup（フル・オンライン・イメージ・バックアップ）」は、dbbackup.exeプログラムまたはBACKUP DATABASE文を使用して、データベースの実行中にデータベース・ファイルとトランザクション・ログ・ファイルに対して別々のバックアップ・コピーを作成します。このバックアップ処理では、開始時にチェックポイント操作が行われます。データベースのバックアップ・ファイルは、この時点のデータベースのイメージになります。しかし、トランザクション・ログのバックアップ・ファイルは、バックアップ処理の終了時に存在するトランザクション・ログのイメージになります。

ユーザはデータベースの更新を継続できますが、コピーが行われている間はデータベース・ファイルに対する変更は一切行われません。この場合、キャッシュ内でのみ変更が行われます。キャッシュが一杯の場合、テンポラリ・ファイルが使用されます。このことは、結果として得られるトランザクション・ログのバックアップ・ファイルには、対応するデータベースのバックアップ・ファイルよりも新しいデータが含まれている可能性があることを意味します。SQL Anywhereは9.11項で説明したようにリカバリ処理を行わなければならないため、完全なリストアには両方のバックアップファイルが必要です。

以下に示すのは、フル・オンライン・イメージ・バックアップを作成するBACKUP DATABASE文の例です。DIRECTORY句だけが指定されていますが、これはすべてのイメージに対してBACKUP DATABASE文を実行するために必要だからです。DBFILE ONLYとTRANSACTION LOG ONLY句は意図的に省略しています。こうすることで、SQL Anywhereはデータベースとトランザクション・ログをともにバックアップ対象に含めるため、フル・バックアップになります。わかりやすくするために、他のすべての句も省略しています。

```
BACKUP DATABASE DIRECTORY 'G:¥bkup';
```

以下に、各種のBACKUP DATABASE句がどのような働きをするかを示します。

- DIRECTORY 句は、バックアップ・ファイルの書き込み先となるフォルダを指定する
- DBFILE ONLY 句は、データベース・ファイルのみをバックアップするように指定する。トランザクション・ログの重要性も十分に高いので、このオプションについてはこれ以上説明しない
- TRANSACTION LOG ONLY 句は、トランザクション・ログ・ファイルのみをバックアップするように指定する。この句は、差分およびインクリメンタル方式のオンライン・ログ・バックアップのためのものであり、次の2つの項で説明する
- TRANSACTION LOG RENAME 句は、バックアップ完了時に現在のトランザクション・ログ・ファイルを閉じてファイル名を YYMMDDXX.LOG という形式に変更し、元のファイル名で新しいトランザクション・ログ・ファイルを開始する。YYMMDD にはその日の年月日が、XX には、その日の最初のバックアップならば AA、2回目のバックアップならば AB というように、それぞれ文字列が入る。TRANSACTION LOG RENAME または TRANSACTION LOG RENAME MATCH 句は、どちらもインクリメンタル・オンライン・ログ・バックアップで使用されるため、この章の後の部分で説明するが、TRANSACTION LOG RENAME MATCH の使用をおすすめする
- TRANSACTION LOG RENAME MATCH は、現在のトランザクション・ログ・ファイルのバックアップ・コピーもまた、ファイル名が変更された現在のトランザクション・ログに合わせて YYMMDDXX.LOG 形式のファイル名に変更されることを除けば、TRANSACTION LOG RENAME と同じである。デフォルトでは、現在のトランザクション・ログのバックアップ・コピーの名前は、現在のトランザクション・ログと同じになるため、BACKUP DATABASE 文の実行を繰り返すと、古いバックアップ・コピーは上書きされる。MATCH キーワードを使えば、バックアップ・ログ・ファイルのコピーである2つの永続的な YYMMDDXX.LOG ファイルを、現在のフォルダとバックアップ・フォルダに1つずつ作成できる
- TRANSACTION LOG TRUNCATE は、BACKUP DATABASE 文の完了時にトランザクション・ログ・ファイルを閉じて削除し、元のファイル名で新しいトランザクション・ログ・ファイルを作成するように指定する。トランザクション・ログのバックアップ・コピーは、DIRECTORY 句で指定されるフォルダに作成されるが、元のファイル名と同じ名前になる。TRANSACTION LOG RENAME 句のほうが柔軟性が高いため、TRANSACTION TRUNCATE についてはこれ以上説明しない
- WAIT BEFORE START 句は、すべてのトランザクションが COMMIT または ROLLBACK 操作によって終了するまで SQL Anywhere に待機するように指示する。この結果として作成されるデータベース・バックアップ・ファイルには、ロールバック・ログの情報は含まれない。検証チェックを実行するためにデータベース・バックアップ・ファイルを読み込み専用モードで開始する場合に便利である。ただし、別々の接続が新しい重複するトランザクションの開始を続けようとする場合、バックアップ処理は WAIT BEFORE START によって待ち状態になり、いつまでも開始されない。検証処理については、9.14 項で説明する
- WAIT AFTER END 句は、すべてのトランザクションが COMMIT または ROLLBACK 操作によって終了するのを待ってから、トランザクション・ログ・ファイルのバックアップを終了するように

SQL Anywhere に指示する。別々の接続が新しい重複するトランザクションの開始を続けようとする場合、バックアップ処理は WAIT AFTER END によって待ち状態になり、いつまでも開始されない。WAIT AFTER END 句は、現在のログ・ファイルの名前変更またはトランケートが行われた場合のみ有効である。他の場合は影響がなく、バックアップは待ち状態になることなく終了する。

WAIT BEFORE START と WAIT AFTER END はともに同一の BACKUP DATABASE 文に対して指定できる。この場合、両方の影響を受けて、新しいトランザクションはバックアップの開始後かつ終了前に開始される。これらのトランザクションによる変更は、データベースのバックアップ・ファイルには含まれないが、トランザクション・ログのバックアップ・ファイルには含まれる可能性があることに注意すること。実際、WAIT AFTER END 句の指定時には、トランザクション・ログのバックアップ・ファイルにはこうしたトランザクションによる変更が含まれることになる

- WITH COMMENT 句を使用すると、バックアップ操作によって、データベース・エンジンと同じフォルダにある backup.syb というテキスト・ファイルに追加されるメッセージに含める文字列リテラルを指定できる。Windows の場合、このテキスト・ファイルのデフォルトのロケーションは C:¥Program Files¥Sybase¥SQL Anywhere 9¥win32¥backup.syb になる。このファイルは、WITH COMMENT 句を使用するかどうかに関わらず更新される。この句を使った例を次に示す

```
BACKUP DATABASE DIRECTORY 'G:¥bkup' WITH COMMENT 'Ad hoc';
```

この文によって backup.syb ファイルに追加された内容を次に示します。

```
BACKUP, 2.0, test9.db, ASAn, '2004-03-17 09:29:00.000', DBA, Full, Imag, G:¥bkup,
   'Ad hoc'
```

9.12.3 差分オンライン・ログ・バックアップ

「differential online log backup（差分オンライン・ログ・バックアップ）」は、dbbackup.exe または BACKUP DATABASE 文を使用して、データベースの実行中にトランザクション・ログのバックアップ・コピーを作成します。「差分」という言葉は、バックアップが行われている現在のトランザクション・ログが、少なくとも最後にフル・バックアップが作成されたとき以来ずっと使用され、その時点以降の変更内容をすべて含んでいることを意味します。

差分バックアップをもう一度行うと、トランザクション・ログに対する前回の差分バックアップ・コピーは必要なくなります。新しい差分バックファイルにはそのとき以降の変更内容とデータがすべて含まれるためです。

完全なリストアを行うには、トランザクション・ログに対する最新の差分バックアップ・コピーに加えて、データベース・ファイルとトランザクション・ログに対する前回のフル・オンライン・バックアップが必要です。

差分オンライン・ログのBACKUP DATABASE文では、TRANSACTION LOG ONLY句を指定しますが、TRANSACTION LOG RENAMEとTRANSACTION LOG RENAME MATCH句は省略します。以下の例には、3つのBACKUP DATABASE文があります。最初の1つがフル・オンライン・イメージ・バックアップの処理を開始するもの、その後に差分オンライン・ログ・バックアップを連続して行う2つの文が続きます。

```
BACKUP DATABASE DIRECTORY 'G:¥bkup' WITH COMMENT 'Start';
BACKUP DATABASE DIRECTORY 'G:¥bkup' TRANSACTION LOG ONLY WITH COMMENT 'Diff 1';
BACKUP DATABASE DIRECTORY 'G:¥bkup' TRANSACTION LOG ONLY WITH COMMENT 'Diff 2';
```

以下に示すのは、上記の3つのBACKUP DATABASE文によってbackup.sysファイルに追加されたエントリです。

```
BACKUP, 2.0, test9.db, ASAn, '2004-03-17 09:31:00.000', DBA, Full, Imag, G:¥bkup,
   'Start'
BACKUP, 2.0, test9.db, ASAn, '2004-03-17 09:31:00.000', DBA, LGFO, Imag, G:¥bkup,
   'Diff 1'
BACKUP, 2.0, test9.db, ASAn, '2004-03-17 09:31:00.000', DBA, LGFO, Imag, G:¥bkup,
   'Diff 2'
```

差分オンライン・ログ・バックアップが実行されるたびに、現在のトランザクション・ログの新しいバックアップ・コピーが作成され、この新しいコピーはそれより前に作成されたバックアップ・コピーを上書きします。現在のトランザクション・ログはそのまま残り、更新が行われるたびにサイズが大きくなっていきます。

9.12.4 インクリメンタル・オンライン・ログ・バックアップ

「incremental online log backup（インクリメンタル・オンライン・ログ・バックアップ）」もまた、dbbackup.exeまたはBACKUP DATABASE文を使用して、データベースの実行中にトランザクション・ログのバックアップ・コピーを作成します。「インクリメンタル」という言葉は、バックアップ対象である現在のトランザクション・ログが前回のバックアップ操作以降でのみ使用されていることを意味します。また、前回のバックアップ操作の時点で、空のトランザクション・ログ・ファイルが新しく作成されています。トランザクション・ログのインクリメンタル・バックアップ・コピーには、前回のバックアップ操作以降の変更内容だけが含まれます。完全なリストアを行うには、トランザクション・ログに対するすべてのインクリメンタル・バックアップ・コピーに加え、データベース・ファイルとトランザクション・ログに対する前回のフル・オンライン・バックアップが必要です。

インクリメンタル・オンライン・ログのBACKUP DATABASE文では、TRANSACTION LOG ONLYに加えて、TRANSACTION LOG RENAMEまたはTRANSACTION LOG RENAME MATCH句を指定します。以下の例には、3つのBACKUP DATABASE文があります。最初の1つが

フル・オンライン・イメージ・バックアップの処理を開始するもの、その後にインクリメンタル・オンライン・ログ・バックアップを連続して行う2つの文が続きます。3つのBACKUP DATABASE文のすべてでTRANSACTION LOG RENAME MATCH句を使用しているため、BACKUP DATABASE文を実行するたびに、現在のトランザクション・ログに対してバックアップ、ファイルのクローズ、名前変更、再起動が行われます。

```
BACKUP DATABASE DIRECTORY 'G:¥bkup'
   TRANSACTION LOG RENAME MATCH;

BACKUP DATABASE DIRECTORY 'G:¥bkup'
   TRANSACTION LOG ONLY
   TRANSACTION LOG RENAME MATCH;

BACKUP DATABASE DIRECTORY 'G:¥bkup'
   TRANSACTION LOG ONLY
   TRANSACTION LOG RENAME MATCH;
```

以下に示すのは、上記3つのBACKUP DATABASE実行後の関連するファイルです。

```
test9.db              - 現在のデータベース・ファイル
test9.log             - 現在のトランザクション・ログ・ファイルの最新版
040317AA.LOG          - 1回目のバックアップでできたトランザクション・ログ・ファイルを名前変更
したもの
040317AA.LOG          - 2回目のバックアップでできたトランザクション・ログ・ファイルを名前変更
したもの
040317AA.LOG          - 3回目のバックアップでできたトランザクション・ログ・ファイルを名前変更
したもの

C:¥mirror¥test9.mlg   - 現在のミラー・ログ・ファイルの最新版
040317AA.LOG          - 1回目のバックアップでできたミラー・ログ・ファイルを名前変更したもの
040317AA.LOG          - 2回目のバックアップでできたミラー・ログ・ファイルを名前変更したもの
040317AA.LOG          - 3回目のバックアップでできたミラー・ログ・ファイルを名前変更したもの

G:¥bkup¥test9.db      - 1回目のバックアップでできたバックアップ・データベース・ファイル
G:¥bkup¥test9.db      - 1回目のバックアップでできたバックアップ・トランザクション・ファイル
G:¥bkup¥test9.db      - 2回目のバックアップでできたバックアップ・トランザクション・ファイル
G:¥bkup¥test9.db      - 3回目のバックアップでできたバックアップ・トランザクション・ファイル
```

Note

BACKUP DATABASEコマンドによって、現在のトランザクション・ログ・ファイルの場合と同じように、現在のミラー・ログ・ファイルでも名前の変更と再起動が行われますが、ミラー・ログ・ファイルのバックアップ・コピーは作成されません。これは問題になりません。いずれにしてもミラー・ログ・ファイルは、実際には対応するトランザクション・ログの単なるコピーに過ぎず、コピーは3つもあれば十分だからです。

9.12.5　ライブ・ログ・バックアップ

ライブ・ログ・バックアップは、dbbackup.exe を使用して、トランザクション・ログ・データをリモート・コンピュータ上のファイルに継続的にコピーします。ライブ・ログ・バックアップ・ファイルは、メイン・コンピュータ上にある現在のトランザクション・ログに対して時間上の遅れがあります。しかし、特に、高速な LAN で 2 台のコンピュータが接続されている場合は大きな遅れにはなりません。他のバックアップ・ファイルがリモート・コンピュータに書き込まれていても、ライブ・ログ・バックアップ・ファイルが管理されていれば、メイン・コンピュータ全体が失われた場合に、リモート・コンピュータを使ってデータベースを起動できます。現在のトランザクション・ログとライブ・ログ・バックアップの間のタイム・ラグによるデータの損失はわずかです。

以下に示すのは、dbbackup.exe をリモート・コンピュータ上で起動する Windows バッチ・ファイルの例です。このバッチ・ファイルはリモート・コンピュータ上で実行されます。開始フォルダは remote_test9 で、前に述べた、メイン・コンピュータ上の G: ドライブに対応づけられているフォルダと同じものです。ローカル環境変数 CONNECTION によって dbbackup で使用される接続の文字列が保持され、LINK パラメータによって dbbackup.exe はメイン・コンピュータ上で実行中のデータベースに LAN 経由で接続できます。-l パラメータは、ライブ・ログ・バックアップが remote_test9¥bkup フォルダ内の live_test9.log というファイルに書き込まれるように指定します。また、最後のパラメータ bkup は、すべての dbbackup コマンド・ラインの最後にバックアップ・フォルダを指定する必要がある、という要求を満たします。

```
SET CONNECTION="ENG=test9;DBN=test9;UID=dba;PWD=sql;LINKS=TCPIP(HOST=TSUNAMI)"
"%ASANY9%¥win32¥dbbackup.exe" -c %CONNECTION% -l bkup¥live_test9.log bkup
```

以下に示すのは、リモートコンピュータ上で dbbackup.exe をしばらく実行した後にコマンド・ウィンドウに表示される内容です。メイン・コンピュータ上で 3 回の BACKUP DATABASE が続けて実行された後、データベースの更新がいくつか行われています。

```
Adaptive Server Anywhere バックアップ・ユーティリティ バージョン 9.0.1.1751
 (1 of 1 pages, 100% complete)
 (1 of 1 pages, 100% complete)
Transaction log truncated by backup - 再起動中...
 (1 of 1 pages, 100% complete)
 (1 of 1 pages, 100% complete)
Transaction log truncated by backup - 再起動中...
 (1 of 1 pages, 100% complete)
 (1 of 1 pages, 100% complete)
Transaction log truncated by backup - 再起動中...
 (1 of 1 pages, 100% complete)
 (2 of 2 pages, 100% complete)
 (3 of 3 pages, 100% complete)
 (4 of 4 pages, 100% complete)
トランザクション・ログのライブ・バックアップは次のページの準備ができています...
```

メイン・コンピュータ上のバックアップ操作によって、現在のトランザクション・ログの名前変更と再起動が行われると、リモート・コンピュータ上で実行中の dbbackup.exe プログラムは、ライブ・ログ・バックアップ・ファイルの内容をいったん消去してから再び書き込みを開始します。このことは、ライブ・ログ・バックアップが、現在処理されているトランザクション・ログの単なるコピーであることを意味し、現在のトランザクション・ログ自身もまた再起動されている以上、問題にならないことを意味します。メイン・コンピュータで実行される他のバックアップ操作は、そのバックアップ・ファイルをリモート・コンピュータにも書き込むので、データベースの起動に必要なすべてのものがリモート・コンピュータ上で利用可能になります。

Note

ライブ・ログ・バックアップなどのバックアップ操作によって、出力ファイルをデータベース・エンジン実行中のコンピュータから別のコンピュータのディスク・ドライブに LAN 経由で書き込むことができます。ただし、アクティブ・データベース、トランザクション・ログ、ミラー・ログ、テンポラリ・ファイルはすべて、データベース・エンジンを実行中のコンピュータにローカルに付与されているディスク・ドライブに配置する必要があります。LAN を経由した入出力は認められていません。このコンテキストでは、ミラー・ログは「バックアップ・ファイル」ではなく、アクティブなトランザクション・ログに対する、冗長でありながらもアクティブなコピーです。

次の項では、この項に示したバックアップの例で作成したファイルを使って、データベースを障害からリストアする方法を説明します。

Section

9.13 リストア

「リストア」とは、データベースの起動と実行に必要なリカバリ処理を実行し、データベースを最新の状態にするために必要なトランザクション・ログを適用しながら、現在のデータベース・ファイルをバックアップ・コピーで置き換える処理です。

Tip

自動化されたリストアのようなものはありません。バックアップ処理は自動化できます。リストアも自動化できるかもしれませんが、どのようなリストアを実行するにしても念入りな調査と注意が必要です。

以下では、データベースのリストアを行うステップの概略をまず示し、その後に例をいくつか示します。

❶ パニックに陥らないようにします。
❷ 事前に計画を行います。具体的には、使用可能なバックアップ・ファイルと実際に使用するバックアップ・ファイルを判断し、どのステップにおいてどのような順番で使うかを決定します。
❸ 上書きされるファイルは名前の変更またはコピーをしておきます。データベースのリストアでは、特にステップ1にしたがうのが困難なことが多く、失敗が起こりやすいため、これは非常に重要です。
❹ データベースのリストアまたはトランザクション・ログ・ファイルの適用もしくはその両方を、ステップ2と3で準備した計画にしたがって実行します。

例1：現在のデータベースとトランザクション・ログがどちらも使用できません。この項の最初に説明した、データベースとトランザクション・ログの両方のフル・オフライン・イメージ・バックアップが最新のバックアップです。以下のWindowsバッチ・ファイルは、そのバックアップを実行したものです。リストアに使用するバックアップ・ファイルG:¥bkup¥test9.dbとG:¥bkup¥test9.log、それにミラー・ログのバックアップは、このバッチ・ファイルによって作成されたものです。

```
SET CONNECTION="ENG=test9;DBN=test9;UID=dba;PWD=sql"
"%ASANY9%¥win32¥dbisql.exe" -c %CONNECTION% STOP ENGINE test9 UNCONDITIONALLY
RENAME G:¥bkup¥test9.db  old_test9.db
RENAME G:¥bkup¥test9.log old_test9.log
RENAME G:¥bkup¥test9.mlg old_test9.mlg
IF EXIST G:¥bkup¥test9.db  GOTO ERROR
IF EXIST G:¥bkup¥test9.log GOTO ERROR
IF EXIST G:¥bkup¥test9.mlg GOTO ERROR
COPY test9.db            G:¥bkup¥test9.db
COPY test9.log           G:¥bkup¥test9.log
COPY C:¥mirror¥test9.mlg G:¥bkup¥test9.mlg
ECHO N | COMP test9.db G:¥bkup¥test9.db
IF ERRORLEVEL 1 GOTO ERROR
ECHO N | COMP test9.log G:¥bkup¥test9.log
IF ERRORLEVEL 1 GOTO ERROR
ECHO N | COMP C:¥mirror¥test9.mlg G:¥bkup¥test9.mlg
IF ERRORLEVEL 1 GOTO ERROR
ERASE G:¥bkup¥old_test9.db
ERASE G:¥bkup¥old_test9.log
ERASE G:¥bkup¥old_test9.mlg
"%ASANY9%¥win32¥dbsrv9.exe" -x tcpip test9.db
GOTO END
:ERROR
PAUSE Backup process failed.
:END
```

この状況で期待できる最善の結果は、前回のバックアップ時の状態にデータベースをリストアすることです。それ以降の更新内容はすべて失われます。次のWindowsバッチ・ファイルは、例1に適した、簡単なフル・リストアを実行します。

```
ATTRIB -R test9.db
ATTRIB -R test9.log
ATTRIB -R C:¥mirror¥test9.mlg
RENAME test9.db            old_test9.db
RENAME test9.log           old_test9.log
RENAME C:¥mirror¥test9.mlg old_test9.mlg
COPY G:¥bkup¥test9.db  test9.db
COPY G:¥bkup¥test9.log test9.log
COPY G:¥bkup¥test9.mlg C:¥mirror¥test9.mlg
"%ASANY9%¥win32¥dbsrv9.exe" -o ex_1_console.txt -x tcpip test9.db
```

以下に、このバッチ・ファイルが例1の状況に対してどのような働きをするかを説明します。

- 3つのATTRIBコマンドは、.db、.log、.mlgの各ファイルの名前を変更できるように「読み込み専用」設定をリセットする
- 3つのRENAMEコマンドは、「上書きされるファイルは名前変更またはコピーをしておく」という先ほどのルールに従っている
- 3つのCOPYコマンドは、.db、.log、.mlgの各バックアップ・ファイルをリモート・コンピュータから現在のフォルダとミラー・フォルダにリストアする。ミラー・ログ・ファイルのリストアは実際には必要ないので、以降の例ではわざわざ行わない
- 最後のコマンドは、リストアされたデータベースとトランザクション・ログ・ファイルを使って、データベース・エンジンをもう一度起動する。-oオプションを指定すると、データベースのコンソール・ウィンドウに表示されるメッセージがファイルにも書き込まれる

例2：現在のデータベースは使用不可能ですが、現在のトランザクション・ファイルはまだ使用できます。この項の最初に説明した、データベースとトランザクション・ログの両方のフル・オフライン・イメージ・バックアップが最新のバックアップです。次の文が、そのバックアップを実行してG:¥bkup¥test9.dbとG:¥bkup¥test9.logの両ファイルを作成したものです。

```
BACKUP DATABASE DIRECTORY 'G:¥bkup';
```

この場合、データベースのバックアップ・ファイルをバックアップ・フォルダから現在のフォルダにコピーし、そのデータベースに現在のトランザクション・ログ・ファイルを適用することによって、データベースをより最新に近い状態に持っていきます。コミットされたトランザクションはすべてリカバリされますが、障害の発生時にコミットされていなかった変更はすべて消失します。次のWindowsバッチ・ファイルは、例2に適したリストアを実行します。

```
ATTRIB -R test9.db
RENAME test9.db old_test9.db
```

```
COPY test9.log old_test9.log
COPY G:¥bkup¥test9.db test9.db
"%ASANY9%¥win32¥dbsrv9.exe" -o ex_2_console.txt test9.db -a G:¥bkup¥test9.log
"%ASANY9%¥win32¥dbsrv9.exe" -o ex_2_console.txt test9.db -a test9.log
"%ASANY9%¥win32¥dbsrv9.exe" -o ex_2_console.txt -x tcpip test9.db
```

以下に、このバッチ・ファイルが例2の状況に対してどのような働きをするかを説明します。

- ATTRIBコマンドは、現在の.dbファイルの「読み込み専用」設定をリセットする。この例では、現在のログ・ファイルをそのままにしておく
- RENAMEコマンドと最初のCOPYコマンドは、「上書きされるファイルは名前変更またはコピーをしておく」という先ほどのルールに従っている。データベース・ファイルは、バックアップ・コピーで上書きされ、現在のトランザクション・ログは、最後の処理におけるサーバの起動時になって更新される
- 2番目のCOPYコマンドは、バックアップの.dbファイルをリモート・コンピュータのバックアップ・フォルダから現在のフォルダにリストアする
- 次のコマンドは、"-a G:¥bkup¥test9.log"というオプションを指定してdbsrv9.exeを実行する。このオプションによって、バックアップの.logファイルが、リストアされたばかりの.dbファイルに適用される。コミットされた変更内容のうち、.logファイルには存在するがデータベース自身には含まれていないものすべてがデータベースに適用される。この処理が必要なのは、元のバックアップがオンラインのBACKUP文によって実行されているため、対応するバックアップ・データベース・ファイルよりもバックアップ・トランザクション・ログのほうがより新しい状態にあり得るからである。-aオプションを指定して実行されるデータベース・エンジンは、まるでバッチ・ユーティリティ・プログラムのように操作を行い、ロール・フォワードの処理が完了するとすぐに停止する
- 最後から2番目のコマンドは、"-a test9.log"というオプションを指定してdbsrv9.exeを実行する。これにより、現在のログ・ファイルがデータベースに適用される。この実行によって、データベースは、バックアップ後のコミットされた変更に対しては最新の状態になる
- 最後のコマンドは、リストアされた.dbファイルと現在のログ・ファイルを使ってデータベース・エンジンをもう一度起動する

Note

ほとんどのリストア処理では、先ほど示したように、データベースのバックアップ・ファイルと同時に作成されたトランザクション・ログのバックアップ・ファイルが、dbsrv9の-aオプションを使って適用される最初のログです。この例に関しては、リカバリに必要な情報がすべて現在のトランザクション・ログに含まれているので、このステップは必要ありません。つまり、"-a G:¥bkup¥test9.log"というオプションを用いたdbsrv9.exeコマンドは省略できます。しかし、実行しても害はなく、通常は必要になるため、この例にも記してあります。

次の出力は、例2の最後の3つの処理を実行したときにデータベースのコンソール・ウィンドウに表示されたものです。

```
I. 03/17 09:21:27. Adaptive Server Anywhere Network Server Version 9.0.0.1270
...
I. 03/17 09:21:27. Starting database "test9" ... at Wed Mar 17 2004 09:21
I. 03/17 09:21:27. データベースのリカバリ処理中です。
I. 03/17 09:21:27.     Wed Mar 17 2004 09:17 での最終チェックポイント
I. 03/17 09:21:27.     チェックポイント・ログ...
I. 03/17 09:21:27.     トランザクション・ログ: G:¥bkup¥test9.log...
I. 03/17 09:21:27.     ロールバック・ログ...
I. 03/17 09:21:27.     チェックポイントを設定中...
I. 03/17 09:21:27. Wed Mar 17 2004 09:21 に "test9" (test9.db) のチェックポイントを開始しています。
I. 03/17 09:21:27. Wed Mar 17 2004 09:21 に "test9" (test9.db) のチェックポイントを終了しました。
I. 03/17 09:21:27. リカバリ処理が完了しました。
I. 03/17 09:21:27. Wed Mar 17 2004 09:21 でデータベース・サーバが停止しました。
...
I. 03/17 09:21:27. Starting database "test9" ... at Wed Mar 17 2004 09:21
I. 03/17 09:21:27. データベースのリカバリ処理中です。
I. 03/17 09:21:27.     Wed Mar 17 2004 09:21 での最終チェックポイント
I. 03/17 09:21:27.     チェックポイント・ログ...
I. 03/17 09:21:27.     トランザクション・ログ: test9.log...
I. 03/17 09:21:27.     ロールバック・ログ...
I. 03/17 09:21:27.     チェックポイントを設定中...
I. 03/17 09:21:28. Wed Mar 17 2004 09:21 に "test9" のチェックポイントを開始しています。
I. 03/17 09:21:28. Wed Mar 17 2004 09:21 に "test9" のチェックポイントを終了しました。
I. 03/17 09:21:28. リカバリ処理が完了しました。
I. 03/17 09:21:28. Wed Mar 17 2004 09:21 でデータベース・サーバが停止しました。
...
I. 03/17 09:21:28. Starting database "test9" ... at Wed Mar 17 2004 09:21
I. 03/17 09:21:28. トランザクション・ログ: test9.log
I. 03/17 09:21:28. トランザクション・ログミラー: C:¥mirror¥test9.mlg
I. 03/17 09:21:28. Wed Mar 17 2004 09:21 に "test9" のチェックポイントを開始しています。
I. 03/17 09:21:28. Wed Mar 17 2004 09:21 に "test9" のチェックポイントを終了しました。
I. 03/17 09:21:28. Wed Mar 17 2004 09:21 に データベース "test9" (test9.db) が起動しました。
I. 03/17 09:21:28. Wed Mar 17 2004 09:21 に起動したデータベース・サーバ
...
I. 03/17 09:21:36. 要求を受信中です。
```

上に示したリストアでは、障害発生時にコミットされていた変更のすべてがリカバリされています。そのようなすべての変更がトランザクション・ログに含まれていたためです。トランザクション・ログに含まれていれば、コミットされていない変更のリカバリも可能です。これは、コミットされていない変更が行われた後に、他のどれかの接続でCOMMITが実行された場合にもあてはまります。つまり、COMMITが実行されると、すべての変更がトランザクション・ログに書き込まれます。

以下に、dbtran.exeユーティリティを使用して、トランザクション・ログ・ファイルの分析と、そのログに記録された変更に対応するSQL文の生成を行う方法を示します。-aオプションは、コミットされ

ていない操作を出力に含めるようにdbtran.exeに指示します。また、入力されるトランザクション・ログ・ファイルと出力されるテキスト・ファイルの2つが指定されています。

```
"%ASANY9%¥win32¥dbtran.exe" -a old_test9.log old_test9.sql
```

以下に示すのは、dbtran.exeユーティリティによって生成された出力テキスト・ファイルからの引用です。ここには、INSERT文が含まれていますが、このコミットされていない操作をリカバリしたい場合は、ISQLでこの文を使用できます。

```
--INSERT-1001-0000385084
INSERT INTO DBA.t1(key_1,non_key_1)
VALUES (9999,'Lost uncommitted insert')
```

例3：現在のデータベースは使用不可能ですが、現在のトランザクション・ファイルはまだ使用できます。バックアップとして、名前が変更された、以前のフル・オンライン・イメージ・バックアップと再起動されたトランザクション・ログ、それにその後2回分のインクリメンタル・ログ・バックアップがあります。以下に示すのは、これらのバックアップを作成した文です。

```
BACKUP DATABASE DIRECTORY 'G:¥bkup'
   TRANSACTION LOG RENAME MATCH;

BACKUP DATABASE DIRECTORY 'G:¥bkup'
   TRANSACTION LOG ONLY
   TRANSACTION LOG RENAME MATCH;

BACKUP DATABASE DIRECTORY 'G:¥bkup'
   TRANSACTION LOG ONLY
   TRANSACTION LOG RENAME MATCH;
```

この場合、リモートのバックアップ・フォルダからバックアップ・データベース・ファイルをコピーしてから、トランザクション・ログ全体を適用してデータベースを最新の状態に持っていきます。以下のWindowsバッチ・ファイルは、例3に適したリストアを実行します。

```
ATTRIB -R test9.db
RENAME test9.db old_test9.db
COPY test9.log old_test9.log
COPY G:¥bkup¥test9.db
"%ASANY9%¥win32¥dbsrv9.exe" -o ex_3_console.txt test9.db -a G:¥bkup¥040317AA.LOG
"%ASANY9%¥win32¥dbsrv9.exe" -o ex_3_console.txt test9.db -a G:¥bkup¥040317AB.LOG
"%ASANY9%¥win32¥dbsrv9.exe" -o ex_3_console.txt test9.db -a G:¥bkup¥040317AC.LOG
"%ASANY9%¥win32¥dbsrv9.exe" -o ex_3_console.txt test9.db -a test9.log
"%ASANY9%¥win32¥dbsrv9.exe" -o ex_3_console.txt -x tcpip test9.db
```

以下に、このバッチ・ファイルが、例3に対してどのような働きをするかを説明します。

- ATTRIBコマンドは、現在の.dbファイルの「読み込み専用」設定をリセットする
- RENAMEコマンドは、「上書きされるファイルは名前変更またはコピーをしておく」という先ほどのルールに従っている。すべての処理が順調にいった場合、これら"old*.*"ファイルはすべて削除される
- 2番目のCOPYコマンドは、バックアップの.dbファイルを、リモート・コンピュータのバックアップ・フォルダから現在のフォルダにコピーする
- 次の3つのコマンドは、dbsrv9.exeを-aオプションを用いて実行し、最も古い3つのトランザクション・ログ・バックアップを古いものから順に適用する
- 最後から2番目のコマンドは、-aオプションを付けてdbsrv9.exeを実行し、現在のトランザクション・ログを適用することによって、コミットされたトランザクションに関するかぎり、データベースを最新の状態にする
- 最後のコマンドは、リストアされた.dbファイルと現在のログ・ファイルを使ってデータベース・エンジンをもう一度起動する

以下の出力は、5回のdbsrv9.exeを実行する例3の処理の間にデータベースのコンソール・ウィンドウに表示されたものです。

```
I. 03/17 09:44:00. Starting database "test9" ... at Wed Mar 17 2004 09:44
...
I. 03/17 09:44:00.     トランザクション・ログ: G:¥bkup¥040317AA.LOG...
...
I. 03/17 09:44:01. Wed Mar 17 2004 09:44 に "test9" のデータベースを開始しています。
...
I. 03/17 09:44:01.     トランザクション・ログ: G:¥bkup¥040317AB.LOG...
...
I. 03/17 09:44:01. Wed Mar 17 2004 09:44 に "test9" のデータベースを開始しています。
...
I. 03/17 09:44:01.     トランザクション・ログ: G:¥bkup¥040317AC.LOG...
...
I. 03/17 09:44:01. Wed Mar 17 2004 09:44 に "test9" のデータベースを開始しています。
...
I. 03/17 09:44:02.     トランザクション・ログ: test9.log...
...
I. 03/17 09:44:02. Wed Mar 17 2004 09:44 に "test9" のデータベースを開始しています。
I. 03/17 09:44:02. トランザクション・ログ: test9.log
...
I. 03/17 09:44:10. 要求を受信中です。
```

例4：メイン・コンピュータが使用できません。バックアップは、例3に示したのと同じものの他に、リモート・コンピュータ上で実行中のライブ・ログ・バックアップが使用できます。以下に示すのは、

ライブ・ログ・バックアップを開始するためにリモート・コンピュータ上で実行するコマンドです。

```
SET CONNECTION="ENG=test9;DBN=test9;UID=dba;PWD=sql;LINKS=TCPIP(HOST=TSUNAMI)"
"%ASANY9%¥win32¥dbbackup.exe" -c %CONNECTION% -l bkup¥live_test9.log bkup
```

以下は、バックアップを作成するためにメイン・コンピュータ上で実行する文です。

```
BACKUP DATABASE DIRECTORY 'G:¥bkup'
   TRANSACTION LOG RENAME MATCH;

BACKUP DATABASE DIRECTORY 'G:¥bkup'
   TRANSACTION LOG ONLY
   TRANSACTION LOG RENAME MATCH;

BACKUP DATABASE DIRECTORY 'G:¥bkup'
   TRANSACTION LOG ONLY
   TRANSACTION LOG RENAME MATCH;
```

この場合、リモート・コンピュータ上でリストア処理を行います。以下の Windows バッチ・ファイルは、例 4 に適したリストアを実行します。

```
COPY bkup¥test9.db
COPY bkup¥live_test9.log test9.log
"%ASANY9%¥win32¥dbsrv9.exe" -o ex_4_console.txt test9.db -a bkup¥040317AD.LOG
"%ASANY9%¥win32¥dbsrv9.exe" -o ex_4_console.txt test9.db -a bkup¥040317AE.LOG
"%ASANY9%¥win32¥dbsrv9.exe" -o ex_4_console.txt test9.db -a bkup¥040317AF.LOG
"%ASANY9%¥win32¥dbsrv9.exe" -o ex_4_console.txt test9.db -a test9.log
"%ASANY9%¥win32¥dbsrv9.exe" -o ex_4_console.txt -x tcpip test9.db
```

このバッチ・ファイルが例 4 に対してどのような働きをするかを以下に説明します。

- 最初の COPY コマンドは、バックアップの .db ファイルを、リモート・コンピュータのバックアップ・フォルダから現在のフォルダにコピーする。これらのコマンドはすべて、リモート・コンピュータ上で実行されるため、バックアップ・ファイルは "G:¥bkup" ではなく、単純に "bkup" として参照されていることに注意する
- 2 番目の COPY コマンドは、ライブ・ログ・バックアップをバックアップ・フォルダから現在のフォルダにコピーし、現在のトランザクション・ログとして使用するためにファイル名を "test9.log" に変更する
- 次の 3 つのコマンドは、dbsrv9.exe を -a オプションを用いて実行し、最も古い 3 つのトランザクション・ログ・バックアップを古いものから順に適用する
- 最後から 2 番目のコマンドは、-a オプションを付けて dbsrv9.exe を実行し、元はライブ・ログ・バックアップ・ファイルだった現在のトランザクション・ログを適用する。これにより、コミットされ

たトランザクションのうち、ライブ・ログ・バックアップ・ファイルに書き込まれたものすべてに関するかぎり、データベースは最新の状態になる

- 最後のコマンドは、リストアされた.dbファイルと現在のログ・ファイルを使ってデータベース・エンジンをもう一度起動する。これでクライアントは、リモート・コンピュータ上のサーバに接続できる。そのため、クライアントが使用する接続文字列の変更が必要になる場合とならない場合があるが、この問題についてはここでは触れない

Section

9.14 検証

データベースを確実に保護したい場合は、バックアップ・データベース・ファイルとバックアップ・トランザクション・ログのすべてに対して、それらが作成されたらすぐに有効性のチェックを行うようにします。

データベースをチェックするには2つの方法があります。dbvalid.exeユーティリティ・プログラムを実行するか、一連のVALIDATE TABLEとVALIDATE INDEX文を実行するかのどちらかです。どちらの方法でも、データベースを起動しておく必要があります。

以下に示す2つのWindowsバッチ・ファイルは、dbvalid.exeを実行する処理を自動化します。copy_database_to_validate.batという最初のバッチ・ファイルは、データベース起動のたびに行われる変更によって元のコピーが影響を受けないようにデータベース・ファイルのテンポラリ・コピーを作成します。次に、-nオプションを使ってdblog.exeを実行し、コピーしたデータベースのトランザクション・ログとミラー・ログの両ファイルの使用をオフにします。さらに、-fオプションを使ってdbsrv9.exeを実行し、コピーしたデータベースのリカバリを、ログファイルを一切適用しないで強制的に行います。最後に、dbsrv9.exeを使って、コピーしたデータベースを起動します。

```
ATTRIB -R temp_%1.db
COPY /Y %1.db temp_%1.db
"%ASANY9%¥win32¥dblog.exe" -n temp_%1.db
"%ASANY9%¥win32¥dbsrv9.exe" -o console.txt temp_%1.db -f
"%ASANY9%¥win32¥dbsrv9.exe" -o console.txt temp_%1.db
```

validate_database_copy.batという、もう1つのWindows batch fileは、データベースのテンポラリ・コピーに対してdbvalid.exeを実行します。

```
@ECHO OFF
SET CONNECTION="ENG=temp_%1;DBN=temp_%1;UID=dba;PWD=sql"
ECHO ***** DBVALID %CONNECTION% >>validate.txt
DATE /T >>validate.txt
```

```
TIME /T >>validate.txt
"%ASANY9%¥win32¥dbvalid.exe" -c %CONNECTION% -f -o validate.txt
IF NOT ERRORLEVEL 1 GOTO OK
ECHO ON
REM ***** ERROR: DATABASE IS INVALID *****
GOTO END
:OK
ECHO ON
ECHO OK >>validate.txt
```

以下に、validate_database_copy.bat ファイルがどのような働きをするかを示します。

- ECHO OFF コマンドは、表示内容の出力をオフにする
- SET コマンドは、接続文字列を保持するためのローカル環境変数を作成する
- ECHO、DATE、TIME の各コマンドは、validate.txt ファイルへの情報の追加を開始する
- その次のコマンドは、-f オプションと -o オプションを使って dbvalid.exe を実行する。-f オプションはすべてのテーブルのフル・チェックを行い、-o オプションは表示内容の出力を validate.txt ファイルに追加する。また、-c オプションを使用することによって、実行中のデータベース、この場合は元のデータベースのテンポラリ・ファイル、への接続を行う
- IF コマンドは、dbvalid.exe からのリターン・コードをチェックする。リターン・コードの値 0 はすべて正常であること、それ以外の値は問題があることを意味する。この IF コマンドは、「(リターン・コード >= 1)でなければ OK ラベルにジャンプし、他の場合は次のコマンドを続けて実行する」と解釈できる
- 残りのコマンドは、リターン・コードの値に応じて「エラー（ERROR)」または「データベースは正常（DATABASE IS OK)」を表示する

以下の例は、上で説明した 2 つのバッチ・ファイルの実行方法を示します。まずは正常なデータベースに対して実行し、次に破壊されたデータベースに対して実行します。2 つのバッチ・ファイルとも、データベース・ファイル名の .db 拡張子を除いたファイル名部分をパラメータとしてとります。

```
copy_database_to_validate valid_test9
validate_database_copy valid_test9

copy_database_to_validate invalid_test9
validate_database_copy invalid_test9
```

以下に示すのは、validate_database_copy.bat を正常なデータベースに対して実行したときの表示内容です。

```
Adaptive Server Anywhere 検証ユーティリティ バージョン 9.0.0.1270
エラーは報告されていません。
```

```
E:¥validate>ECHO OK  1>>validate.txt
E:¥validate>REM ***** DATABASE IS OK *****
```

以下に示すのは、問題のあるデータベース、具体的には、インデックスが破壊されたデータベースに対して validate_database_copy.bat を実行したときの表示内容です。

```
Adaptive Server Anywhere 検証ユーティリティ バージョン 9.0.0.1270
DBA.t1 を検証しています。
実行時 SQL エラーです -- インデックス "x1" にインデックス・エントリがありません
エラーが 1 件報告されました。
E:¥validate>REM ***** ERROR: DATABASE IS INVALID *****
```

以下に、validate_database_copy.bat を上記のように2回実行した後の validate.txt ファイルの内容を示します。データベース接続パラメータ、日時、検証結果が記録されています。

```
***** DBVALID "ENG=temp_valid_test9;DBN=temp_valid_test9;UID=dba;PWD=sql"
Wed 03/17/2004
 8:19a
Adaptive Server Anywhere 検証ユーティリティ バージョン 9.0.0.1270
エラーは報告されていません
OK
***** DBVALID "ENG=temp_invalid_test9;DBN=temp_invalid_test9;UID=dba;PWD=sql"
Wed 03/17/2004
 8:19a
Adaptive Server Anywhere 検証ユーティリティ バージョン 9.0.0.1270
実行時 SQL エラーです -- インデックス "x1" にインデックス・エントリがありません
エラーが 1 件報告されました。
```

次に示すのは、VALIDATE TABLE 文の構文です。

```
<validate_table> ::= VALIDATE TABLE [ <owner_name> "." ] <table_name>
                        [ <with_check> ]

<with_check>     ::= WITH DATA CHECK    -- データ・チェックを追加
                   | WITH EXPRESS CHECK -- データ・チェックと高速なインデックス・チェックを追加
                   | WITH INDEX CHECK   -- フル・インデックス・チェックを追加
                   | WITH FULL CHECK    -- データ・チェックとフル・インデックス・チェックを追加
```

どの WITH 句も指定しない場合、VALIDATE TABLE 文は基本的なローとインデックスのチェックを行います。各種の WITH 句は、次のようにチェックの動作を拡張します。

- WITH DATA CHECK は、BLOB ページの追加チェックを行う
- WITH EXPRESS CHECK は、WITH DATA のチェックに加えて、より詳細なインデックス・チェックを行う

- WITH INDEX CHECK は、テーブルの全インデックスに対し、VALIDATE INDEX 文と同じ拡張インデックス・チェックを行う
- WITH FULL CHECK は、WITH DATA と WITH INDEX のチェックを組み合わせて、最も徹底したチェックを行う

以下に示すのは、先ほどの例で dbvalid.exe によってエラーが検出されたものと同じデータベースに対して実行した VALIDATE TABLE 文の例です。

```
VALIDATE TABLE t1;
```

この VALIDATE TABLE 文の結果、SQLSTATE の値が '40000' になり、同じエラー・メッセージ「実行時 SQL エラーです – インデックス "x1" にインデックス・エントリがありません」が生成されました。

この VALIDATE INDEX 文は、基本的なチェックのほか、単一インデックスの妥当性のチェックも行い、すべてのインデックスが実際にテーブル内のローに対応しているかの確認と、インデックスが外部キー上にある場合は親テーブル内の対応するローが実際に存在するかの確認を行います。

VALIDATE INDEX には 2 つの異なる形式があります。1 つはプライマリ・キー・インデックス用、もう 1 つは他の種類のインデックス用です。その構文は次のとおりです。

```
<validate_primary_key> ::= VALIDATE INDEX
                           [ [ <owner_name> "." ] <table_name> "." ]
                           <table_name>
<validate_other_index> ::= VALIDATE INDEX
                           [ [ <owner_name> "." ] <table_name> "." ]
                           <index_name>
<index_name>           ::= <identifier>
```

次に示すのは、テーブル t1 のプライマリ・キー・インデックスをチェックする VALIDATE INDEX 文の例です。このインデックスは正常なので、この文は SQLSTATE を '00000' に設定します。

```
VALIDATE INDEX DBA.t1.t1;
```

以下に示すのは、テーブル t1 の x1 というインデックスをチェックする VALIDATE INDEX 文の例です。この文を、先ほどの VALIDATE TABLE 文の例と同じデータベースに対して実行すると、やはり SQLSTATE が '40000' に設定され、見つからないインデックス・エントリについての同じエラー・メッセージが生成されます。

```
VALIDATE INDEX t1.x1;
```

次に示すのは、テーブル t2 上で役割名 fk2 を持つ外部キーをチェックする VALIDATE INDEX 文の例です。

```
VALIDATE INDEX t2.fk2;
```

この場合、テーブル上のあるロー内の外部キー・カラムの値が破壊されており、VALIDATE INDEX によって次のエラー・メッセージが生成されます。

```
「テーブル "t2" の外部キー "fk2" が無効です。」
「テーブル "t1" のプライマリ・キーまたは一意性制約 "t1" にエントリがありません。」
```

ログを SQL コマンドに変換しようとする dbtran.exe ユーティリティを使用することによって、トランザクション・ログ・ファイルの妥当性をチェックできます。この試みが成功すれば、ログに問題はありません。失敗した場合、このログはリカバリの目的では使用できません。

以下に示すのは、トランザクション・ログ・ファイル指定をパラメータとして指定するコマンド・ラインから呼び出せる check_log.bat という Windows バッチ・ファイルの例です。このバッチ・ファイルは、エラー・メッセージを validate.txt というテキスト・ファイルに追加する -o オプション、出力 SQL ファイルを上書きする -y オプション、バッチ・ファイルのパラメータ値を表す %1 表記、dummy.sql という出力 SQL ファイルをそれぞれ指定して、dbtran.exe を実行します。

```
ECHO OFF
ECHO ***** DBTRAN %1 >>validate.txt
DATE /T >>validate.txt
TIME /T >>validate.txt
"%ASANY9%¥win32¥dbtran.exe" -o validate.txt -y %1 dummy.sql
IF NOT ERRORLEVEL 1 GOTO OK
ECHO ON
REM ***** ERROR: LOG IS INVALID *****
GOTO END
:OK
ECHO ON
ECHO OK >>validate.txt
REM ***** LOG IS OK *****
:END
```

次の 2 つの Windows コマンド・ラインは、正常なトランザクション・ログと破損したログに対して 1 回ずつ check_log.bat を呼び出します。

```
CALL check_log 040226AB.LOG
CALL check_log 040226AC.LOG
```

上記の check_log.bat の最初の呼び出しでは「***** ログは正常です *****（***** LOG IS OK *****）」と表示され、2 つめの呼び出しでは「***** エラー：ログが不正です（***** ERROR: LOG IS INVALID *****）*****」と表示されます。以下に、これら 2 回の呼び出しを行った後の validate.txt ファイルの内容を示します。

```
***** DBTRAN 040226AB.LOG
Fri 02/27/2004
10:17a
Adaptive Server Anywhere ログ変換ユーティリティ バージョン 9.0.0.1270
トランザクション・ログ "040226AB.LOG" はオフセット 0000380624 で開始します。
トランザクション・ログはオフセット 0000385294 で停止します。
OK
***** DBTRAN 040226AC.LOG
Fri 02/27/2004
10:17a
Adaptive Server Anywhere ログ変換ユーティリティ バージョン 9.0.0.1270
トランザクション・ログ "040226AC.LOG" はオフセット 0000380624 で開始します。
ログ・ファイルが破損しています (オペレーションは無効です)。
ログの破損はオフセット 0000385082 で開始しています。
オフセット 0000385082 でのログ・オペレーションには、オフセット 0000385083 で不良データがあります。
```

Section

9.15 この章のまとめ

この章では、SQL Anywhere データベースの整合性を保護するために用いられるさまざまな方法と機能について説明しました。

9.2 項では、ローカルとグローバルのデータベース・オプションについて説明し、また、その値が内部デフォルト、パブリック・デフォルト、ユーザ・デフォルト、特定の接続において現在使用中、の 4 つの異なるレベルにどのように存在するかについても説明しました。

9.3 項では、トランザクションの "ACID" プロパティと呼ばれる、アトミック性、一貫性、独立性、持続性について説明しました。また、BEGIN TRANSACTION、COMMIT、ROLLBACK を利用したトランザクション制御の詳細についても、サーバ側とクライアント側のオートコミット・モードと共に説明しました。

9.4 項では、セーブポイントについて説明し、また、部分的なロールバックを可能にするネストされたサブトランザクションの形式が、セーブポイントを使ってどのように実現できるかについても説明しました。

9.5 項とそのサブセクションでは、SIGNAL、RESIGNAL、RAISERROR、CREATE MESSAGE、ROLLBACK TRIGGER の各文を利用してクライアント・アプリケーションに問題を明示的にレポートする方法を説明しました。

9.6 と 9.7 項では、ロック、ブロック、データベースの一貫性と同時実行性とのトレードオフについて説明し、全体のスループット低下という代償を払って独立性レベルを高くすることによってどのように矛盾を回避できるかを示しました。9.8 項では、環状デッドロック、スレッド・デッドロックについて説明し、SQL Anywhere がこれらをどのように処理するか、どうすれば根本的な問題に対処できるかを説明しました。9.9 項では、ミューテックスによって複数 CPU 環境のスループットがいかに低下し得る

かを説明しました。

次の項とそのサブセクションでは、接続、ユーザ ID、権限の関係について説明し、GRANT 文のさまざまな形式を利用してユーザIDを作成する方法とユーザIDに各種の権限を与える方法を示しました。9.10.5 のサブセクションでは、ユーザ・グループに権限を継承させる方法、パーミッションと権限の違いについて説明し、また、ユーザ・グループを利用して、テーブルとビューの参照時に所有者の名前を明示的に指定せずに済む方法を説明しました。

9.11 項では、トランザクション、チェックポイント、リカバリ・ログがどのような働きをするか、COMMIT と CHECKPOINT 操作の途中で何が起こっているか、SQL Anywhere によるデータベースの起動時にどのようにログが使われるかを含め、ロギングとリカバリのさまざまな側面を説明しました。9.12 から 9.14 の 3 つの項では、データベースのバックアップとリストアの各処理について説明し、また、バックアップ・ファイルの検証によって、データベースのリストアが必要なときにバックアップ・ファイルが利用可能か確かめる方法についても説明しました。

次の章では、保護からパフォーマンスへと話題を移して、SQL Anywhere データベースのパフォーマンスを向上させるためのさまざまな方法と取り組みを紹介します。

Chapter

10

チューニング

Section

10.1 概要

「コンピューティングでもたらされる罪悪は、（必ずしも達成されるとは限らない）効率の追求という名目の下に成されているものが最大を占め、その中には単なる無知に起因するものもある」。

Carnegie-Mellon University（カーネギーメロン大学）の William Wulf（ウイリアム・ウルフ）は、1972 年に ACM の席上で『A Case Against the GOTO』という論文を発表しましたが、上記の引用はその中の一文です。ここに述べられている見解は、今日見られるような、誤って最適化されたプログラムやデータベースにも、そのまま当てはまります。

次の引用は、単なる観察でなく現実を基にしているという点でより実用的な意見であり、守るべき 2 つのルールを述べています。なおこのルールは、Michael A. Jackson が Associated Press 社より 1975 年に出版した『構造的プログラム設計の原理』から抜粋しました。。

最適化についてのルール

ルール 1. 最適化を行ってはいけない。

ルール 2.（熟練者になっても）やはり最適化を行ってはいけない。

アプリケーションやデータベースは、高速で動作することよりも、所定の目的を正しく果たし管理が容易なことが重要であり、パフォーマンスを改善しようとする多くの試みは、かえってバグを発生させたりメンテナンスの負担を増やしてしまいます。とは言うものの、この章の表題にもあるように、SQL Anywhere データベースのパフォーマンスを改善する必要に迫られる場合もあるので、ここではそのための手法やアプローチ、関連するヒントやテクニックを紹介します。ただし、特にパフォーマンス面での不満が生じていなければ、この章は読み飛ばしてください。「壊れていないものを直すべからず」という諺にあるとおりです。

最初のトピックは要求レベルのロギングと呼ばれるもので、これは、クライアント・アプリケーションから出されたどの SQL 文がデータベース・サーバの処理時間を専有しているかを確認する作業です。こうした作業によって、ごく簡単な変更を加えるだけでパフォーマンスを向上できる箇所が、アプリケーション中に見つかる場合もあります。そこまで順調に行かない場合は、この章で説明する方法を用いて、要求レベルのロギングで特定されたクエリをより詳細に分析することになります。

2 番目のトピックはインデックス・コンサルタントという機能を利用するもので、これを用いると、インデックスの構築によって実行時の負荷を軽減できるかが確認できます。実行に時間のかかるストアド・プロシージャやトリガがある場合は、プロファイリングの項で説明する方法を用いると、モジュール内部で速度低下の原因となっている部位を特定できたり、要求レベルのロギングやインデックス・コンサルタントでは示されない詳細な情報を得られる場合があります。グラフィカルなプランの項では、SQL

Anywhereのクエリ・エンジンにおいてパフォーマンス面での障害になるクエリの検討法を説明します。

10.6項では、ファイル、テーブル、インデックスの断片化および、これらの問題の対処法を説明します。インデックスはこの章全体に関係する問題ですが、特にCREATE INDEX文については、独立した項を設けて詳しく説明します。その他には、SQL Anywhereが管理するデータベースのパフォーマンス・カウンタの項を設けています。また、他の項で取り上げなかったヒントや手法については最後の項で一覧します。

Section

10.2 要求レベルのロギング

SQL Anywhereのデータベース・エンジンには要求レベルのロギングと呼ばれる機能が用意されており、クライアント・アプリケーションから出される要求のトレース結果をテキスト・ファイルに出力させることができます。この出力を確認することで、最も時間を要しているSQL文を特定し、問題の箇所をピンポイントで修正できます。

次の例は、組み込みストアド・プロシージャのsa_server_optionをISQLから呼び出して、要求レベルのロギングをオンにする方法を示しています。最初の呼び出しで出力先のテキスト・ファイルを指定し、次の呼び出しでロギングを開始しています。

```
CALL sa_server_option ( 'Request_level_log_file', 'C:¥¥temp¥¥rlog.txt' );
CALL sa_server_option ( 'Request_level_logging', 'SQL+hostvars' );【監注15】
```

sa_server_optionプロシージャでは、使用するオプション名と値を、2つの文字列パラメータを用いて指定します。

最初の呼び出しにあるファイル指定用の「C:¥¥temp¥¥rlog.txt」は、データベース・サーバが動作しているコンピュータからの相対パスで記述します。既にログ・ファイルが存在している場合は既存ファイルにログが追加され、そうでない場合は新規にファイルが作成されます。

Tip

要求レベルのロギングで得られる出力ファイルは、UNCフォーマット【監注16】を指定して別のコンピュータに移すようなことはせず、データベース・サーバが動作するコンピュータ上に残してください。作成されたファイルは、後から別の場所にコピーすることも、サーバ上で分析することもできます。

【監注15】9.0.2.3182より、新しいオプションが追加され、時間のかかったクエリのみロギングすることもできる。
CALL sa_server_option ('RememberLastPlan', 'ON');
CALL sa_server_option ('LogExpensiveQueries', '1000'); -- 単位はミリ秒
【監注16】「¥¥サーバ名¥共有リソース名」の形式のこと。

2番目の呼び出しでは、出力ファイルを開き、記録プロセスを開始すると同時に、記録レベルの詳細を指定しています。指定できる記録レベルは、「SQL」(SQL 文を出力ファイルに記録する)、「SQL+hostvars」(SQL 文とホスト変数値を合わせて記録する)、「ALL」(クライアントからサーバに出される SQL 以外のトラフィックも記録する)です。最初の2つの指定値がパフォーマンス分析をする場合によく使用されるのに対して、最後の「ALL」は非常に大量の出力が生成されるため、パフォーマンス分析よりもデバッグに適しています。

ロギングを停止させるには、次のように指定して、sa_server_option を再度呼び出します。

```
CALL sa_server_option ( 'Request_level_logging', 'NONE' );
```

「NONE」オプションを指定すると、サーバがロギングを停止すると同時に、ログ・ファイルを閉じるので、その後は、ワードパッドなどのテキスト・エディタでログ・ファイルを開くことができます。

Tip

得られたログ・ファイルに、次回以降の出力を追加させたくない場合は、このファイルを削除しておくか、別名で保存しておきます。

次に示すロギング・ファイルの抜粋は、4つの接続を持つデータベースを対象として、要求レベルのロギングを試験的に短時間実行した結果です。4分間の出力でログ・ファイルは2,400 行以上になり、サイズは 270K に達しています。この抜粋は、1つの SELECT 文に関して生成されたものです。

```
12/04 17:43:18.073 ** REQUEST conn: 305282592  STMT_PREPARE            "SELECT *
   FROM child AS c   WHERE c.non_key_4 LIKE '0000000007%';  "
12/04 17:43:18.073 ** DONE    conn: 305282592  STMT_PREPARE            Stmt=65548
12/04 17:43:18.074 ** REQUEST conn: 305282592  STMT_EXECUTE            Stmt=-1
12/04 17:43:18.074 ** WARNING conn: 305282592  code: 111 "Statement cannot be executed"
12/04 17:43:18.074 ** DONE    conn: 305282592  STMT_EXECUTE
12/04 17:43:18.075 ** REQUEST conn: 305282592  CURSOR_OPEN             Stmt=65548
12/04 17:43:18.075 ** DONE    conn: 305282592  CURSOR_OPEN             Crsr=65549
12/04 17:43:58.400 ** WARNING conn: 305282592  code: 100 "Row not found"
12/04 17:43:58.401 ** REQUEST conn: 305282592  CURSOR_CLOSE            Crsr=65549
12/04 17:43:58.401 ** DONE    conn: 305282592  CURSOR_CLOSE
12/04 17:43:58.409 ** REQUEST conn: 305282592  STMT_DROP               Stmt=65548
12/04 17:43:58.409 ** DONE    conn: 305282592  STMT_DROP
```

この抜粋を見ると、発せられた SELECT 文の完全なテキストが確認できます。また、プロセスが 17:43:18 に開始して 17:43:58 に終了していることもわかります。

Note

要求レベルのロギングで生じるオーバヘッドは、アクティブな接続数が数個の場合にはそれほどではありませんが、アクティブな接続数が増えるほど大きくなっていきます。特に「Request_level_logging」に「ALL」を指定すると、高負荷なサーバのパフォーマンスに悪影響を及ぼす場合があります。これは、すべての接続に関するログ・データを 1 つのテキスト・ファイルに書き込むという処理が、サーバに要求されるためです。

要求レベルのロギングについては、良い点と悪い点があります。悪い点は、次に挙げるいくつかの理由で、出力ファイルの扱いに難があることです。1 番目の問題点はファイルが巨大化しがちなことで、高負荷なサーバであれば、ごく短時間の実行結果でも数ギガバイトに達してしまいます。2 番目の問題点は、ファイルに出力される内容が冗長なことで、クライアント・アプリケーションから発せられた単一の SQL 文に関する情報が、ファイル上の複数行に分割されてしまいます。3 番目の問題点は、各 SQL 文が改行されることなく 1 行形式で出力されることです（先に見た SELECT 文の抜粋は、紙面に合わせるために後から改行したもので、ファイル上では改行されていません）。4 番目の問題点は接続番号が出力されないことで、内部の接続ハンドルは "305282592" のような形式で確認できますが、これだけでは該当する SQL 文と発信元のアプリケーションを付き合わせるのは困難です。最後の問題点は各 SQL 文の経過時間が出力されないことで、たとえば実行に 40 秒以上要する SELECT を特定したければ、必要な計算を自分でするしかありません。

良い点は、要求レベルのロギングで得られる出力については、その分析や要約をするために、各種のストアド・プロシージャが SQL Anywhere に用意されていることです。これらのうち、最初に紹介するのは sa_get_request_times と呼ばれるストアド・プロシージャです。このストアド・プロシージャは、要求レベルのロギングで得られる出力ファイルを読み込んで各種の処理を行います。具体的には、複数の行に分かれていた SQL 文ごとの情報を単一のエントリにまとめる、SQL 文ごとの経過時間を計算する、接続番号を接続ハンドルに対応付ける、satmp_request_time という組み込みの GLOBAL TEMPORARY TABLE に結果を出力する、などです。

satmp_request_time のスキーマは、次のとおりです。

```
CREATE GLOBAL TEMPORARY TABLE dbo.satmp_request_time (
   req_id        INTEGER NOT NULL,
   conn_id       UNSIGNED INT NULL,
   conn_handle   UNSIGNED INT NULL,
   stmt_num      INTEGER NULL,
   millisecs     INTEGER NOT NULL,
   stmt_id       INTEGER NULL,
   stmt          LONG VARCHAR NOT NULL,
   prefix        LONG VARCHAR NULL,
   PRIMARY KEY ( req_id ) )
   ON COMMIT PRESERVE ROWS;
```

satmp_request_time の各ローは、1 つの SQL 文に対応します。req_id カラムには、要求レベルのロギングの出力ファイル上での各 SQL 文に対応する最初の行番号が収められるので、このテーブルを時系列でソートする場合に利用できます。conn_id カラムには、conn_handle に格納されるハンドルに対する実際の接続番号が収められます。stmt_num カラムには、要求レベルのロギングの出力ファイル上で「Stmt=65548」のような形式で表示されている、内部処理用の「文番号」エントリが収められます。stmt_id および prefix カラムには、sa_get_request_times プロシージャが値を収められません。特に重要なのは stmt および millisecs の 2 つのカラムで、前者には実際の SQL 文を記述したテキストが、後者にはその経過時間が収められます。

次の例は、先ほど抜粋を示した要求レベルのロギングの出力ファイルに対して、sa_get_request_times による処理を施すものです。ここでは SELECT 文を用いて satmp_request_time を表示していますが、2,400 行もあったテキスト・ファイルの内容が 215 ローのテーブルにまとめられています。

```
CALL sa_get_request_times ( 'C:¥¥temp¥¥rlog.txt' );

SELECT req_id,
       conn_id,
       conn_handle,
       stmt_num,
       millisecs,
       stmt
  FROM satmp_request_time
 ORDER BY req_id;
```

satmp_request_time の最初の 3 ローと、先の抜粋で表示されていた SELECT 文は次のように表示されます。

```
req_id  conn_id     conn_handle  stmt_num  millisecs stmt
======  ==========  ===========  ========  ========= ============================
5       1473734206  305182584    65536     3         'SELECT @@version, if ''A''...
11      1473734206  305182584    65537     6         'SET TEMPORARY OPTION ...
17      1473734206  305182584    65538     0         'SELECT connection_property...
...
1297    1939687630  305282592    65548     40326     'SELECT *     FROM child ...
```

Tip

satmp_request_time テーブルの各ローが、オリジナルの出力ファイルのどの部分に対応するかは、req_id カラムに収められている行番号または stmt_num カラムの値を利用することで特定できます。たとえば、上記の 4 番目のローに対応するログ・ファイル行を検索するのであれば、ワードパッドの検索機能で「Stmt=65548」と指定します。ただし、複数のデータベースを運用しているサーバの場合、データベースごとに独自に文番号が割り当てられるので、同じ文番号が複数存在する可能性があります。

次の SELECT 文は、処理時間の長い順に、上位 10 個の文を一覧します。

```
SELECT TOP 10
       millisecs,
       stmt
  FROM satmp_request_time
 ORDER BY millisecs DESC;
```

結果は、次のように出力されます。

```
millisecs  stmt
=========  ========================================================================
111813     'SELECT c.key_1,         c.key_2,         c.non_key_3, ...
41195      'SELECT *    FROM child AS c   WHERE c.non_key_4 LIKE ''0000000005%''; '
40326      'SELECT *    FROM child AS c   WHERE c.non_key_4 LIKE ''0000000007%''; '
19595      'SELECT p.key_1,         p.non_key_3,          p.non_key_5 ...
17674      'call "dba".p_non_key_3'
257        'call "dba".p_parent_child'
218        'SELECT c.key_1,         c.key_2,         c.non_key_3, ...
217        'SELECT c.key_1,         c.key_2,         c.non_key_3, ...
216        'SELECT c.key_1,         c.key_2,         c.non_key_3, ...
216        'SELECT c.key_1,         c.key_2,         c.non_key_3, ...
```

これらのストアド・プロシージャやクエリは、要求レベルのロギングの出力ファイルを作成したデータベースやサーバ上で実行する必要はありません。作成後の出力ファイルは、他のコンピュータに移動して分析できます。sa_get_request_times などの組み込みプロシージャおよび satmp_request_time などのテーブルは、すべての SQL Anywhere データベースに用意されているので、作成直後で空のデータベースであっても、他のサーバから移動した要求レベルのロギングの出力ファイルを分析できます。

2 番目に紹介する組み込みストアド・プロシージャは sa_get_request_profile と呼ばれるもので、これは sa_get_request_times のすべての機能に加えて、次の 4 つの処理を段階的に行えます。最初は、COMMIT と ROLLBACK の実行に要した時間を集計して、satmp_request_time の単一ローにまとめる機能です。2 つ目は、WHERE 句の部分を削除したうえで、「類似」した文の先頭部を satmp_request_time.prefix カラムに収める機能です。3 つ目は、stmt_id に値を割り当てる機能で、この値は prefix 値が一致するすべての行で同じ値になります。最後に、satmp_request_time テーブルのデータをコピーして、satmp_request_profile と呼ばれる第 2 のテーブルに要約する機能です。

次の例は、先ほど抜粋を示した要求レベルのロギングの出力ファイルに対して、sa_get_request_profile による処理を施すものです。ここでは SELECT 文を用いて satmp_request_profile を表示していますが、2,400 行もあったテキスト・ファイルの内容が 17 ローのテーブルにまとめられています。

```
CALL sa_get_request_profile ( 'C:¥¥temp¥¥rlog.txt' );
SELECT *
  FROM satmp_request_profile;
```

結果セットは、次のとおりです。satmp_request_profile.uses カラムは prefix に一致する SQL 文が何回実行されたかを示し、total_ms、avg_ms、max_ms カラムは 1 つの文の実行に要した総時間、平均時間、最大時間をそれぞれ示します。

```
stmt_id  uses  total_ms  avg_ms  max_ms  prefix
=======  ====  ========  ======  ======  ========================================
1        2     3         1       2       'SELECT @@version, if ''A''<>''a'' then...
2        2     31        15      19      'SET TEMPORARY OPTION Time_format = ...
3        2     1         0       1       'SELECT connection_property( ...
4        2     1         0       1       'SELECT db_name()'
5        2     1         0       1       'SELECT @@SERVERNAME'
6        2     8         4       6       'SELECT (SELECT width FROM ...
7        2     28        14      15      'SELECT DISTINCT if domain_name = ...
8        97    10773     111     133     'SELECT customer.company_name, ...
9        1     17674     17674   17674   'call "dba".p_non_key_3'
10       10    113742    11374   111813  'SELECT c.key_1,         c.key_2, ...
11       2     81521     40760   41195   'SELECT *     FROM child AS c   '
12       30    21056     701     19595   'SELECT p.key_1,         p.non_key_3, ...
13       28    3067      109     174     'SELECT *     FROM parent AS p   '
14       15    1457      97      257     'call "dba".p_parent_child'
15       15    1304      86      148     'call "dba".p_parent_child_b'
16       1     0         0       0       'CALL sa_server_option ( ...
17       2     0         0       0       'COMMIT'
```

このような類似した SQL 文の実行に要した時間の要約情報は、処理に長時間を要するオペレーションがどのクライアント・アプリケーションから出されているかを特定したい場合に有用です。たとえば、アプリケーションが誤った種類のクエリを実行していた、または 1 つのオペレーションを繰り返し実行していたなどの問題点が判明すれば、アプリケーション・コードの一部を変更するだけで、処理速度が改善されることがあります。

ただし、一般には、正しいクエリが実行されているのになぜか過剰な実行時間を要しているというケースが多く、そうした場合は SQL 文に関する「prefix」以外の情報が必要となります。より具体的に言えば、WHERE 句を含めた SELECT 文の全体を確認することです。それには、SELECT 文を読みやすいフォーマットに整形できると便利です。

こうした処理に利用できるのが、SQL Anywhere に用意されている 3 番目の組み込みストアド・プロシージャである sa_statement_text で、これは SELECT 文を含む文字列を読み込み、複数の行に分割することで、読みやすいフォーマットに整形してくれます。次の例は、sa_statement_text の呼び出し方および、その結果セットを示したものです。

```
CALL sa_statement_text
   ( 'SELECT * FROM child AS c WHERE c.non_key_4 LIKE ''0000000007%''' );

stmt_text
====================================
SELECT *
        FROM child AS c
WHERE c.non_key_4 LIKE ''0000000007%''
```

現状では、sa_statement_text は関数ではなくプロシージャとして記述されており、複数の改行コードで分割された1つの文字列ではなく、複数のローから成る結果セットを返すので、操作性の面で難があります。しかし、次のような関数を定義することで、sa_statement_text の機能をより使いやすい形で利用できます。

```
CREATE FUNCTION f_formatted_statement ( IN @raw_statement LONG VARCHAR )
RETURNS LONG VARCHAR
NOT DETERMINISTIC
BEGIN
   DECLARE @formatted_statement LONG VARCHAR;
   SET @formatted_statement = '';
   FOR fstmt AS cstmt CURSOR FOR
   SELECT sa_statement_text.stmt_text AS @formatted_line
     FROM sa_statement_text ( @raw_statement )
   DO
      SET @formatted_statement = STRING (
         @formatted_statement,
         '¥x0d¥x0a',
         @formatted_line );
   END FOR;
   RETURN @formatted_statement;
END;
```

このユーザ定義関数 f_formatted_statement では、入力パラメータとして未フォーマット状態の SQL 文を受け取り、sa_statement_text プロシージャに渡します。sa_statement_text がフォーマット処理した結果セットは、カーソルの FOR ループでローごとに処理し、キャリッジ・リターンとラインフィードを示す '¥x0d¥x0a' を付加して、フォーマット後のすべての行を結合します。カーソルの FOR ループの詳細については「第6章　フェッチ」を、CREATE FUNCTION 文の詳細については「第8章　パッケージ」を参照してください。

次の例は、f_formatted_statement の使用法を示すもので、ここでは UNLOAD SELECT 文を用いてテキスト・ファイルを生成しています。

```
UNLOAD SELECT f_formatted_statement
   ( 'SELECT * FROM child AS c WHERE c.non_key_4 LIKE ''0000000007%''' )
TO 'C:¥¥temp¥¥sql.txt' QUOTES OFF ESCAPES OFF;
```

結果は、次のように出力されます。f_formatted_statement から返されるのは1つの文字列ですが、ファイルの内容は4つの行に分割されています（テキスト3行と先頭部の改行）。

```
SELECT *
        FROM child AS c
WHERE c.non_key_4 LIKE '0000000007%'
```

新規に定義したf_formatted_statement関数をsa_get_request_timesと組み合わせることで、次のようなp_summarize_request_timesプロシージャを作成することもできます。

```
CREATE PROCEDURE p_summarize_request_times ( IN @log_filespec LONG VARCHAR )
BEGIN
   CALL sa_get_request_times ( @log_filespec );
   SELECT NUMBER(*)                                          AS stmt_#,
          COUNT(*)                                           AS uses,
          SUM ( satmp_request_time.millisecs )               AS total_ms,
          CAST ( ROUND ( AVG ( satmp_request_time.millisecs ),
                         0 ) AS BIGINT )                     AS avg_ms,
          MAX ( satmp_request_time.millisecs )               AS max_ms,
          f_formatted_statement ( satmp_request_time.stmt ) AS stmt
     FROM satmp_request_time
    GROUP BY satmp_request_time.stmt
   HAVING total_ms >= 100
    ORDER BY total_ms DESC;
END;
```

このp_summarize_request_timesプロシージャは、入力パラメータとして要求レベルのロギングの出力ファイルを受け取り、これをsa_get_request_times組み込みプロシージャに渡すことで、satmp_request_timeテーブルに情報を収めています。次のSELECT文では、GROUP BY句を用いて、個々のSQL文（WHERE句を含めた）の経過時間を集計しています。そして、f_formatted_statementを呼び出して、各SQL文を複数の行に分割します。結果セットは、トータルの経過時間の大きい順にソートし、NUMBER(*)関数を用いて各ローに「文番号」を割り当てます。HAVING句では、トータル経過時間が10分の1秒以上である文のみを出力するよう制限をかけています。

次の例はp_summarize_request_timesの使用法を示すもので、フォーマットしたレポートをUNLOAD SELECT …FROM句を用いてファイルに出力します。UNLOAD SELECTの詳細については「3.25　UNLOAD TABLEとUNLOAD SELECT」を参照してください。

```
UNLOAD
SELECT STRING ( '-- Statement ',
                stmt_#,
                ': ',
                uses,
                ' uses, ',
```

```
                total_ms,
                ' ms total, ',
                avg_ms,
                ' ms average, ',
                max_ms,
                ' ms maximum time ',
                stmt,
                '¥x0d¥x0a' )
  FROM p_summarize_request_times ( 'C:¥¥temp¥¥rlog.txt' )
TO 'C:¥¥temp¥¥rlog_summary.txt' QUOTES OFF ESCAPES OFF;
```

これで得られるテキスト・ファイル rlog_summary.txt には、12 個の SQL 文に関する情報が要約されています。次に、その最初の 5 個を抜粋して示します。これらは、4 個の SELECT 文と 1 個のプロシージャ呼び出しに関するものです。

```
-- Statement 1: 1 uses, 111813 ms total, 111813 ms average, 111813 ms maximum time
SELECT c.key_1,
         c.key_2,
         c.non_key_3,
         c.non_key_5
        FROM child AS c
WHERE c.non_key_5 BETWEEN '1983-01-01'
AND   '1992-01-01 12:59:59'
ORDER BY c.non_key_5;

-- Statement 2: 1 uses, 41195 ms total, 41195 ms average, 41195 ms maximum time
SELECT *
        FROM child AS c
WHERE c.non_key_4 LIKE '0000000005%';

-- Statement 3: 1 uses, 40326 ms total, 40326 ms average, 40326 ms maximum time
SELECT *
        FROM child AS c
WHERE c.non_key_4 LIKE '0000000007%';

-- Statement 4: 1 uses, 19595 ms total, 19595 ms average, 19595 ms maximum time
SELECT p.key_1,
         p.non_key_3,
         p.non_key_5
        FROM parent AS p
WHERE p.non_key_5 BETWEEN '1983-01-01'
AND   '1992-01-01 12:59:59'
ORDER BY p.key_1;

-- Statement 5: 1 uses, 17674 ms total, 17674 ms average, 17674 ms maximum time
call "dba".p_non_key_3
```

上記の Statement 5 の部分を見ると、要求レベルのログからは、クライアント・アプリケーションが

直接呼び出したプロシージャの実行時間についておおよその情報がわかりますが、プロシージャのどの部分の処理に時間がかかっているかについては、何の情報も得られません。また、トリガについての情報や、他のプロシージャやトリガから呼び出されるネストされたプロシージャについての情報を得ることもできません。プロシージャやトリガ内部での処理を確認する方法については、「10.4　プロファイリング」を参照してください。

要求レベルのロギングは、特定のサーバで実行されるすべてのSQL処理についての情報を集めるのに使用されることが多く、どのクライアント接続からSQLが発行されたか、あるいは該当する接続がどのデータベースを使用しているかは問題にしていません。たとえば、上記の例の場合、4つの接続および、1つのサーバ上にある2つのデータベースが使用されていることだけがわかります。

ただし、要求レベルのロギングの出力ファイルにフィルタをかけて、特定の接続から出された要求だけに絞り込むこともできます。こうした手法が有用なのは、サーバの負荷が高く、多数の接続が同じような処理を行っている場合です。内容の重複したログ・データが数ギガバイトに及んで記録されたり、統計を収集する時間が制限されたりすることなく、特定の接続に注目して、その状態を長時間にわたってモニタリングできるのは大きなメリットです。

1つの接続に絞って要求レベルのロギングを行うには、まず、該当する接続番号を調べる必要があります。こうした情報の取得にあたっては、sa_conn_infoストアド・プロシージャを次のように使用することで、現在使用中のすべての接続番号を一覧させることができます。

```
SELECT sa_conn_info.number AS connection_number,
       sa_conn_info.userid AS user_id,
       IF connection_number = CONNECTION_PROPERTY ( 'Number' )
          THEN 'this connection'
          ELSE 'different connection'
       ENDIF AS relationship
FROM sa_conn_info();
```

これで得られる結果セットでは、すべての接続とそのユーザIDだけでなく、現在の接続がどれであるかも確認できます。

```
connection_number  user_id   relationship
=================  ========  ====================
1864165868         DBA       this connection
286533653          bcarter   different connection
856385086          mkammer   different connection
383362151          ggreaves  different connection
```

組み込みストアド・プロシージャのsa_server_optionを用いると、要求レベルのロギングを接続別にフィルタリングすることができます。その場合は、第1パラメータにオプション名'Requests_for_connection'を、第2パラメータに接続番号を指定します。

次の例は、単一の接続に対する要求レベルのロギングを開始するプロシージャ呼び出しです。ここで

は、接続番号として 383362151 を指定しています。また、ロギングの停止も、同様のプロシージャ呼び出しで行っています。

```
CALL sa_server_option ( 'Request_level_log_file', 'C:¥¥temp¥¥rlog_single.txt' );
CALL sa_server_option ( 'Requests_for_connection', 383362151 );
CALL sa_server_option ( 'Request_level_logging', 'SQL+hostvars' );
-- 接続 383362151 からの要求をログに記録開始。
CALL sa_server_option ( 'Request_level_logging', 'NONE' );
```

要求レベルのロギングのフィルタリングを接続レベルで終了させるには、次のプロシージャ呼び出しを使用します。

```
CALL sa_server_option ( 'Requests_for_connection', -1 );
```

Tip

フィルタリングを終了させるには、「CALL sa_server_option ('Requests_for_connection', -1)」を実行する必要があります。'Requests_for_connection' と接続番号を指定して sa_server_option を呼び出すと、該当する接続番号が他の呼び出しで変更されるか、サーバが再起動するか、-1 を指定して明示的に終了させるまで、フィルタリング指定が維持されます。

sa_server_option の呼び出しは、要求レベルのロギングをデータベース単位でフィルタリングする場合にも使用できます。それにはまず、使用するデータベース番号を調べておく必要がありますが、次のような SELECT 文を用いることで、サーバ上で動作しているすべてのデータベース番号と名前を一覧させることができます。

```
SELECT sa_db_properties.number AS database_number,
       sa_db_properties.value  AS database_name,
       IF database_number = CONNECTION_PROPERTY ( 'DBNumber' )
          THEN 'this database'
          ELSE 'different database'
       ENDIF AS relationship
  FROM sa_db_properties()
 WHERE sa_db_properties.PropName = 'Name'
 ORDER BY database_number;
```

結果セットを確認すると、個々のデータベースの識別情報および、現在の接続が使用しているデータベースを特定できます。

```
database_number  database_name  relationship
===============  =============  ==================
0                asademo        different database
1                volume         this database
```

ストアド・プロシージャの sa_server_option を用いると、要求レベルのロギングをデータベース別にフィ

ルタリングすることができます。その場合は、第1パラメータにオプション名'Requests_for_database'を、第2パラメータにデータベース番号を指定します。

次の例は、単一のデータベースに対する要求レベルのロギングを開始するプロシージャ呼び出しです。ここでは、データベース番号として0を指定しています。またロギングの停止も、同様のプロシージャ呼び出しで行っています。

```
CALL sa_server_option ( 'Request_level_log_file', 'C:¥¥temp¥¥rdb.txt' );
CALL sa_server_option ( 'Requests_for_database', 0 );
CALL sa_server_option ( 'Request_level_logging', 'SQL+hostvars' );
-- データベース0の要求をログに記録開始。
CALL sa_server_option ( 'Request_level_logging', 'NONE' );
```

要求レベルのロギングのフィルタリングを、データベース・レベルで終了させるには、次のプロシージャ呼び出しを使用します。

```
CALL sa_server_option ( 'Requests_for_database', -1 );
```

Tip

フィルタリングを終了させるには、「CALL sa_server_option ('Requests_for_database', -1)」を実行する必要があります。また、接続のフィルタリングとデータベースのフィルタリングを併用した場合は、フィルタリング指定の組み合わせを間違えると、想定外の要求レベルのロギングが解除されてしまうので、注意が必要です。

Section

10.3 インデックス・コンサルタント

要求レベルのロギングの出力ファイルを検討した結果、いくつかのクエリ処理に長時間を要していることが判明し、インデックスを追加することで事態が改善できると考えられる場合は、インデックス・コンサルタントが役に立ちます。

動作中のデータベースでインデックス・コンサルタントを使用するには、Sybase Centralでデータベースに接続して、ツリー・ビューからデータベースを選択し、右クリックでポップアップ・メニューを表示させて［インデックス・コンサルタント］をクリックします（図10-1を参照）。

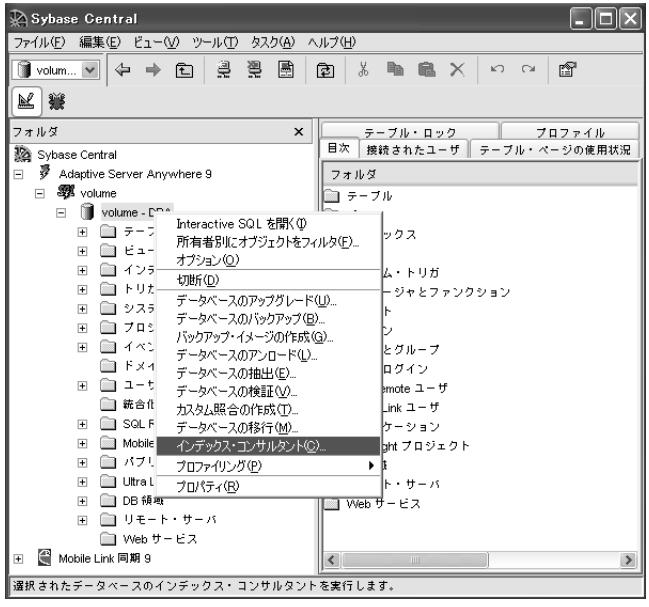


図 10-1　Sybase Central でのインデックス・コンサルタントの起動

インデックス・コンサルタントのウィザードが起動します。最初のウィンドウでは、新規分析の開始を指定しますが、後で検討する場合にそなえて、保存用の名前を入力しておきます（図 10-2 を参照）。

図 10-2　インデックス・コンサルタントでの新規分析の開始

ウィザードの最初のウィンドウで［次へ］ボタンをクリックすると、図 10-3 のステータス・ウィンドウが表示されます。これ以降、インデックス・コンサルタントのセッションは、［完了］ボタンがクリック

されるまで、該当データベースで実行されるすべてのクエリについてのモニタと情報収集を継続します。負荷を手動で動作させる場合は、この時点で別の接続を用いて負荷を開始します。既存の接続でデータベース処理が実行されている場合は、インデックス・コンサルタントによるモニタリングが行われます。

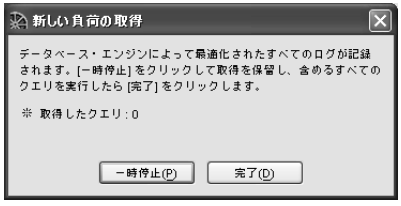


図 10-3　インデックス・コンサルタントによる新規負荷の取得開始

表示の変化に多少時間がかかりますが、[取得したクエリ] のカウントが増加することで、動作状況を確認できます。必要なクエリのサンプル情報をインデックス・コンサルタントが収集し終わったと判断したら（図 10-4 を参照）、[完了] ボタンをクリックして、データ収集を終了させます。

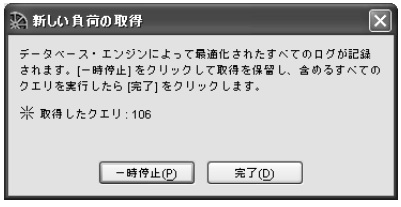


図 10-4　インデックス・コンサルタントによるデータ収集の完了

収集直後のデータをインデックス・コンサルタントに分析させるには、事前にいくつかの質問に答えておく必要があります。1 番目の質問はインデックスに関するものです（図 10-5 を参照）。ここでは、クラスタード・インデックスを作成するか、負荷の軽減に役に立たないと思われる既存のインデックスを廃棄させるかを指定します。

図 10-5　インデックス・コンサルタントのインデックス設定

2 番目の質問はディスクの使用量に関するものです（図 10-6 を参照）。ここでは、インデックスにサイズ制限を課し、その制限内で処理をさせるかどうかを指定します。

図 10-6　インデックス・コンサルタントでのディスク使用量の指定オプション

質問への解答後、[次へ] ボタンをクリックすると、インデックス・コンサルタントによる分析が始まり、図 10-7 のステータス・ウィンドウが表示されます。

図 10-7　負荷分析中のインデックス・コンサルタント

その後 [インデックス・コンサルタント] ウィザードは、分析結果の詳細を、図 10-8 の [推奨インデックス] タブなどの形式で表示します。

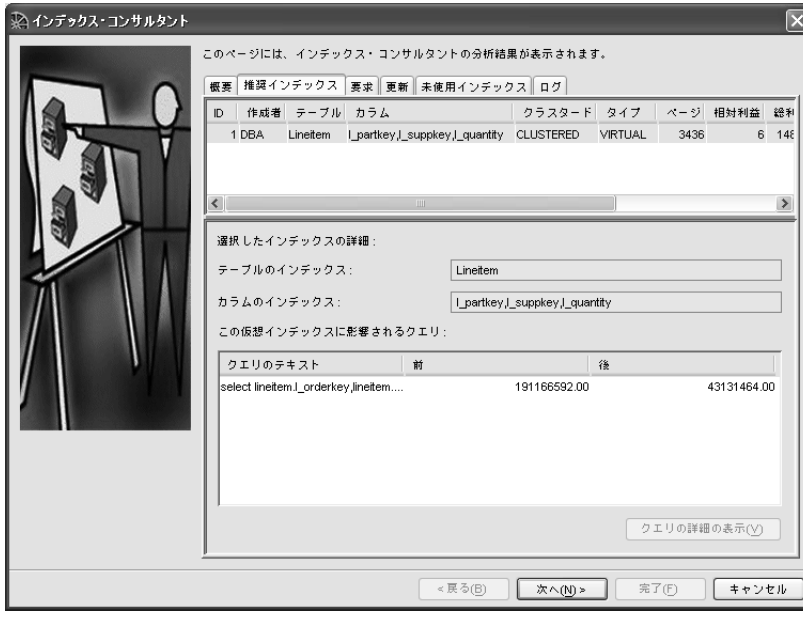


図 10-8　インデックス・コンサルタントによる推奨インデックスの提示

[推奨インデックス] タブには、必要と思われるインデックスと、それぞれの新規インデックスの影響を受けるクエリが一覧されます。新規インデックスに関連する特定のクエリを選択して、[クエリ詳細の表示] をクリックすると、その詳細情報が表示されます。

最後に［インデックス・コンサルタント］ウィザードは、図10-9にあるような［推奨インデックス］のスクリプトを表示します。このスクリプトは、DROP INDEXコマンドによる未使用インデックスの破棄、CREATE INDEXコマンドによる新規インデックスの作成に加え、クラスタード・インデックスの順序に基づいてテーブルのローを再ソートするために、新規のクラスタード・インデックスに対してそれぞれ実行するREORGANIZE TABLEコマンドから構成されます。スクリプトをテキスト・ファイルに保存しておき後から実行する場合は［保存］ボタンを、直ちに変更を行う場合は［スクリプトの実行］ボタンをクリックします。

Tip

インデックスには、その目的がよくわかるような名前を付けてください。インデックス・コンサルタントは図10-9にあるような［推奨インデックス］のスクリプトを提示しますが、負荷分析の名前を基にしたインデックス名が付けられているので、必要であれば［スクリプトの実行］ボタンをクリックする前に変更できます。

図10-9　インデックス・コンサルタントの提示する［推奨インデックス］スクリプト

Section

10.4 プロファイリング

プロファイリングとは、ストアド・プロシージャ、関数、トリガのどの処理に時間を要しているかを確認する機能のことです。動作中のデータベースでプロファイリングを使用するには、Sybase Central でデータベースに接続して、ツリー・ビューからデータベースを選択し、右クリックでポップアップ・メニューを表示させて［プロファイリング］－［プロファイリングの開始］をクリックします（図 10-10 を参照）。

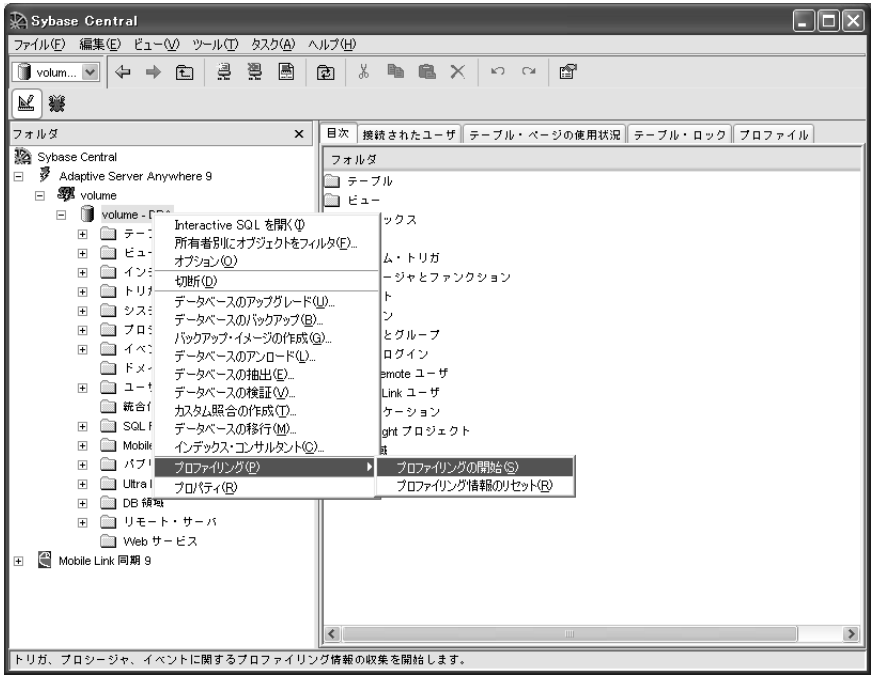


図 10-10　Sybase Central でのプロファイリングの起動

これ以降、SQL Anywhere は、［プロファイリング］－［プロファイリングの停止］がクリックされるまで、該当データベースのストアド・プロシージャやトリガにおいて、それぞれの文の実行に要した時間を追跡します。

プロファイル情報を確認するには、Sybase Central のツリー・ビューでデータベースを選択してから、［プロファイル］タブをクリックします。情報を再表示させる場合は、［F5］キーを押してください。［プロファイル］タブには、プロファイリングの実行中に処理されたすべてのプロシージャ、関数、トリガが一覧されます（図 10-11 を参照）。

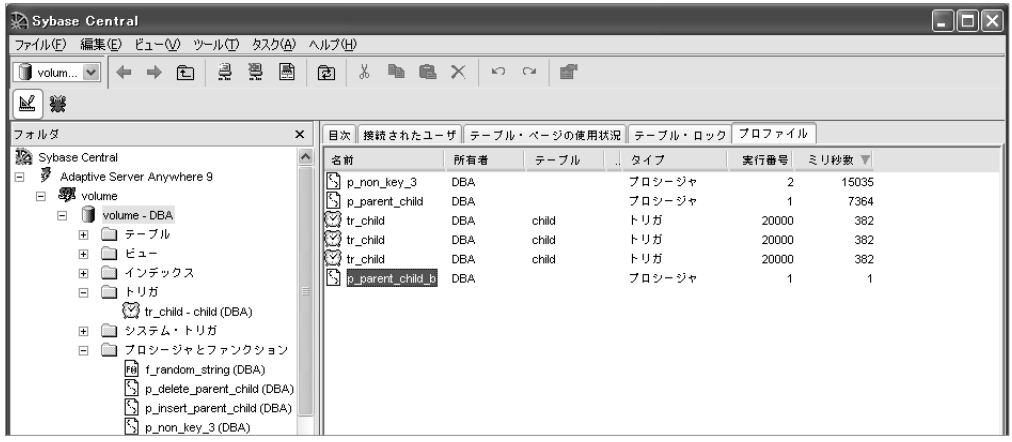


図 10-11　プロファイリングで表示されるプロシージャとトリガの一覧

Note

プロファイリングで収集されるのは、プロファイリングを行うデータベース内での情報です。要求レベルのロギングの場合とは違い、データを別のコンピュータに移動してから異なるエンジンを用いて分析することはできないので、プロファイリングを行ったデータベースとの接続を維持しておく必要があります。

ストアド・プロシージャやトリガ内部での所用時間を確認するには、［プロファイル］タブで、該当するエントリをダブルクリックします。たとえば図 10-12 の場合、p_non_key_3 プロシージャの所要時間が SQL コードの文別に表示されており、UPDATE 文の実行に 15 秒を要していることが確認できます。

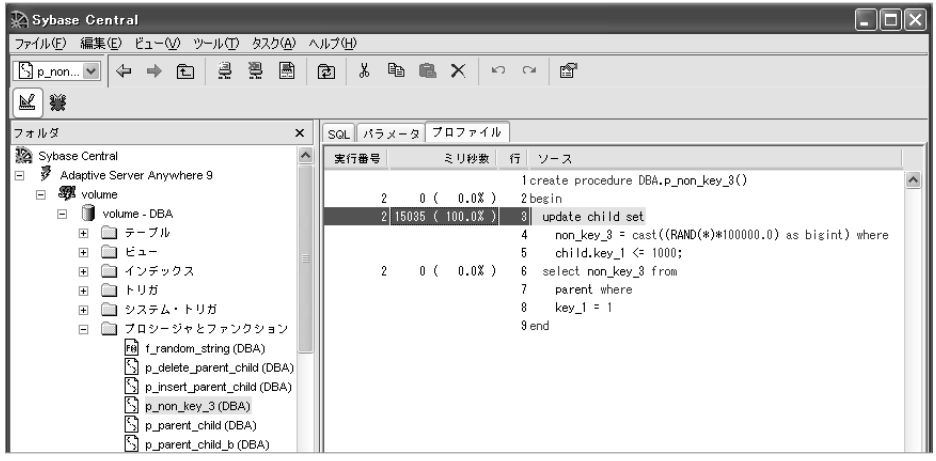


図 10-12　ストアド・プロシージャの実行プロファイル

p_non_key_3 プロシージャで、UPDATE 文の実行に要したとされている時間は、そのすべてが子のテーブルの更新に費やされたのではなく、実際には、子のローが更新されるごとに起動される tr_child トリガでもその一部が費やされています。図 10-13 を見ると、tr_child にある別の UPDATE 文の実行に 0.4 秒が費やされていることが確認できますが、この UPDATE 文は子テーブルではなく親テーブルを処理するものです。

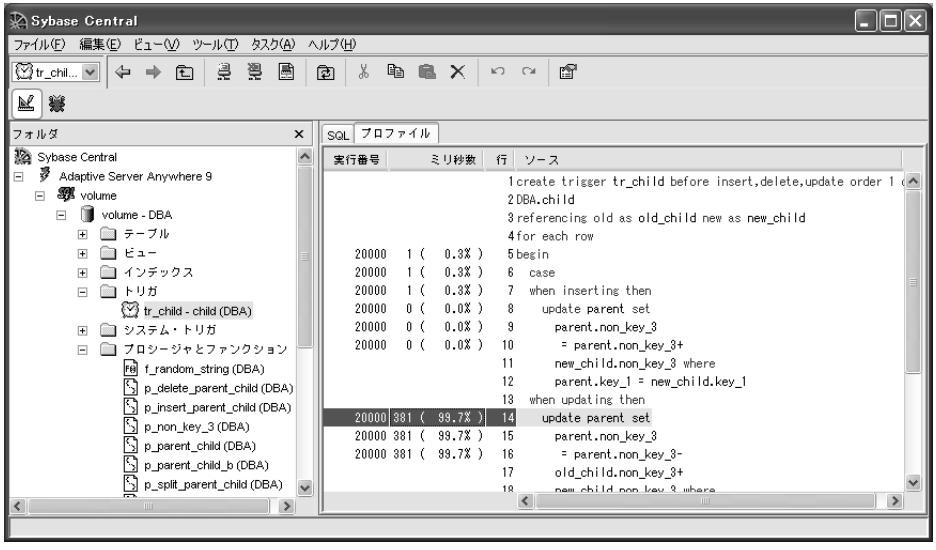


図 10-13　トリガの実行プロファイル

Section

10.5 グラフィカルなプラン

SELECT を始め、クエリを含む INSERT、UPDATE、DELETE などの操作に対しては SQL Anywhere が「実行プラン」を選択しますが、こうした実行プランを確認するには、グラフィカルなプランという機能を使用します。実行プランがパフォーマンス面で果たす役割は重要です。通常は、現在のデータベースの状態に応じて SQL Anywhere が最適な実行プランを自動的に選択しますが、不適切なものが選ばれる場合もあります。そのため、実行プランの詳細を確認することでパフォーマンス上の問題点が特定でき、より適した実行プランが選択されるよう、スキーマやクエリを変更できる場合があります。

グラフィカルなプランは ISQL から表示させます。ISQL のメニュー・オプションで［ツール］－［オプション］を選んで、［プラン］をクリックすると、図 10-14 のようなオプション・ウィンドウが表示されます。

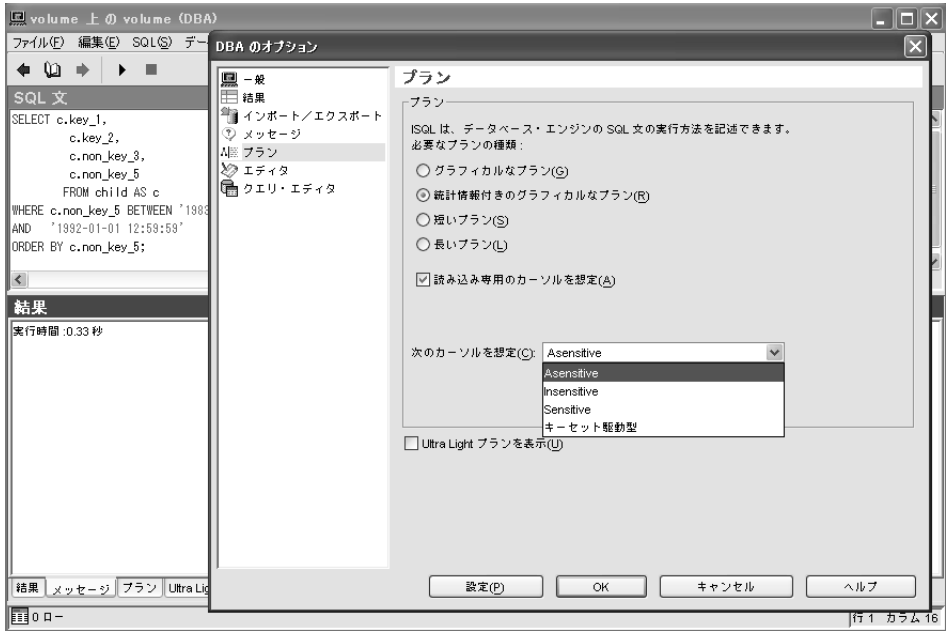


図 10-14　グラフィカルなプランの設定オプション

図 10-14 の指定は、次の意味を持ちます。

- **［統計情報付きのグラフィカルなプラン］ラジオ・ボタン**――実行時の統計情報および予測値を取得するための指定。このオプションを選択すると、ISQL はクエリを実行するが、結果セットは表示されない。
- **［読み込み専用のカーソルを想定］チェック・ボックス**――実際にクエリを発したアプリケーションで使われたものと、カーソルの種類を合わせるための指定。デフォルトでは更新可能なカーソルのプランが生成されるが、これは読み込み専用カーソルのプランと種類が異なり非効率な場合がある。
- **［次のカーソルを想定］の［Asensitive］**――これも上記と同様の指定で、実際にクエリを発したアプリケーションで使われたものと、カーソルの種類を合わせる。
- **［Ultra Light プランを表示］チェック・ボックス**――このチェック・ボックスをオフにしてあるのは、プランの表示を高速化するため。Ultra Light プランは、Ultra Light アプリケーションのクエリを最適化する場合は非常に有用だが、ここでの例には当てはまらない。

Tip

アプリケーション中で読み込み専用に用いている SELECT 文に対しては、必ず［読み込み専用のカーソルを想定］チェック・ボックスをオンにしてください。この指定の有無によって、ISQL の表示するプランに大きな違いが生じ、パフォーマンスの改善に必要な情報が得られない場合があります。

ISQLの［SQL文］ウィンドウ枠にクエリをペーストしてから［Shift + F5］キーを押すか、メニュー・オプションで［SQL］−［プランの取得］を選ぶと、グラフィカルなプランが表示されます。たとえば、「10.2　要求レベルのロギング」でサンプルとして取り上げたSQL文の中で、処理時間が最大であったものを見てみましょう。次のサンプルは、処理時間が最大である文のSQLコードをp_summarize_request_timesストアド・プロシージャで出力させたものです。

```
-- Statement 1: 1 uses, 111813 ms total, 111813 ms average, 111813 ms maximum time
SELECT c.key_1,
         c.key_2,
         c.non_key_3,
         c.non_key_5
        FROM child AS c
WHERE c.non_key_5 BETWEEN '1983-01-01'
AND   '1992-01-01 12:59:59'
ORDER BY c.non_key_5;
```

図10-15は、これと同じSQL文をISQLで表示したものです。［結果］ウィンドウ枠の［プラン］タブにグラフィカルなプランが表示されています。ウィンドウ枠の左側には、全体の概要が4つのアイコンでグラフィカル表示されています。［SELECT］アイコンはクエリ全体を示し、［Work］アイコンは結果セットの作成に用いたテンポラリ・ワーク・テーブルを示し、［Sort］アイコンはORDER BYによるソート処理を示し、「c」とラベルされたアイコンは子テーブルからのローの選択に用いたプロセスを示しています。ウィンドウ枠の右側には、選択中のアイコンの詳細が表示されます。図中の「c」アイコンの詳細では、子テーブルが連続的にスキャンされており、このクエリによって子テーブルから25,000ローが返されるとSQL Anywhereが予測していることが確認できます。

図10-15　SELECTによるテーブル・スキャンのグラフィカルなプラン

グラフィカルなプランに表示される統計データをスクロールしてゆくと、膨大な量の情報が取得できます。たとえば、図10-16を見ると、このクエリで実際に返されたのは748ローだけであり、WHERE句の述部で一致したローは2.99%だけであったことがわかります。選択性が非常に高い述部については、インデックスの存在が大きく影響する場合があります。たとえば、non_key_5カラムにインデックスが付けられていれば、まったく異なった実行プランが得られて、より高いパフォーマンスが達成できた可能性があります。

図10-16　グラフィカルなプランの示す統計および述部情報とヘルプ・ウィンドウ

図10-16にはコンテキスト別の［Interactive SQL ヘルプ］ウィンドウも表示されていますが、これを呼び出すには、右マウス・ボタンを押してからポップアップ・メニューの［ヘルプ］を選びます。この例のヘルプ・ウィンドウには、テーブル・スキャンが実行されたこと、そして、その多くは「適切なインデックスが定義されていない」ときに行われる、ということが表示されています。

グラフィカルなプランの表示は、INSERT、UPDATE、DELETE文でも行えます。図10-17はUPDATE文に対するプランで、図10-12にあるプロファイリングからコピーしてペーストしたものです。

図 10-17　UPDATE のインデックス・スキャンに関するグラフィカルなプラン

図 10-17 を見ると、ここでの複数ローの UPDATE 文はキーセット駆動型カーソルで実装されていること、子テーブルの検索にはプライマリ・キー・インデックスが使用されていることがわかります。

グラフィカルなプランに表示された内容は、ISQL のメニュー・オプション［ファイル］－［名前を付けて保存］を選んでファイル・フォーマットに XML を指定（*.XML）することで、XML ファイルに保存しておくことができます。こうして作成される XML ファイルには、グラフィカルなプランに表示されたすべての情報が保存されます。保存しておいた情報は ISQL で表示させることができ、この場合、オリジナルのデータベースにアクセスする必要はありません。

グラフィカルなプランの情報を記録した XML ファイルは、ISQL を介さずに作成することもできます。組み込み関数 GRAPHICAL_PLAN は、SQL 文を含む文字列を解析し、グラフィカルなプランの情報を XML フォーマットの文字列として返します。この GRAPHICAL_PLAN に UNLOAD SELECT を併用すると、ストアド・プロシージャやトリガに含まれる文について、その実行プランを保存できます。次の例は、その具体的な構文です。

```
<write_graphical_plan_to_file> ::= UNLOAD SELECT GRAPHICAL_PLAN
                                     "(" <string_containing_SQL_statement> ","
                                         <statistics_level> ","
                                         <cursor_type_parameter> ","
                                         <update_status> ")"
                                   TO <xml_output_filespec> ESCAPES OFF QUOTES OFF
```

```
<string_containing_SQL_statement> ::= クエリを含んだ文字リテラルまたは変数
<statistics_level>              ::= 0 -- 予測のみ
                                  | 1 -- 実際の統計の要約を含む
                                  | 2 -- 実際の統計の詳細を含む
<cursor_type_parameter>         ::= 'asensitive'
                                  | 'insensitive'
                                  | 'sensitive'
                                  | 'keyset-driven'
<update_status>                 ::= 'READ-ONLY'
                                  | 'FOR UPDATE'
<xml_output_filespec> ::= サーバからの相対位置でファイルを指定する文字リテラル
```

この方法は、実行時に確定するホスト変数を用いる文の情報を取得する場合に有用です。次の例は、プロシージャのパラメータを参照する WHERE 句を SELECT に指定したストアド・プロシージャです。

```
CREATE PROCEDURE p_test_temp ( IN @from_date TIMESTAMP )
BEGIN
SELECT c.key_1,
       c.key_2,
       c.non_key_3,
       c.non_key_5
 FROM child AS c
WHERE c.non_key_5 BETWEEN @from_date
  AND '1992-01-01 12:59:59'
ORDER BY c.non_key_5;
END;
```

プロシージャにUNLOAD SELECT GRAPHICAL_PLAN 文を追加するステップは、次のとおりです。

❶ SELECT 部をコピーしたものを 2 回ペーストし、2 つ目の部分に次のステップで変更を加えます。
❷ SELECT 部を一重引用符で囲み、埋め込まれている引用符をすべて二重化して、この SELECT 部を有効な 1 つの文字列リテラルにしておきます。
❸ ホスト変数がある場合は、文字列リテラルを分割し、STRING 関数を用いて、実行時に実際のホスト変数の値を文字列に挿入します。必要に応じて、タイムスタンプ・リテラルの引用符など、ホスト変数を囲む一重引用符を二重化しておきます。
❹ 処理した SELECT 部の前後に、GRAPHICAL_PLAN の呼び出しと関連パラメータ、UNLOAD SELECT のキーワード、TO によるファイル指定、ESCAPES OFF および QUOTES OFF のオプションを追加します。

次の例は、前述のプロシージャに UNLOAD SELECT GRAPHICAL_PLAN を追加して得られるコードです。これを実行すると、SELECT 文は 1 つの文字列値として GRAPHICAL_PLAN に渡されます。GRAPHICAL_PLAN からは XML フォーマットの実行プランを含む文字列が返され、その文字列が

c:¥temp¥plan_for_embedded_select.xml のファイルに出力されます。

```
CREATE PROCEDURE p_test_temp ( IN @from_date TIMESTAMP )
BEGIN

SELECT c.key_1,
       c.key_2,
       c.non_key_3,
       c.non_key_5
 FROM child AS c
WHERE c.non_key_5 BETWEEN @from_date
  AND '1992-01-01 12:59:59'
ORDER BY c.non_key_5;

UNLOAD SELECT GRAPHICAL_PLAN ( STRING (
'SELECT c.key_1,
       c.key_2,
       c.non_key_3,
       c.non_key_5
  FROM child AS c
 WHERE c.non_key_5 BETWEEN ''',
@from_date,
''' AND ''1992-01-01 12:59:59''
 ORDER BY c.non_key_5;' ),
2, 'asensitive', 'READ-ONLY' )
TO 'c:¥¥temp¥¥plan_for_embedded_select.xml' ESCAPES OFF QUOTES OFF;
END;
```

このプロシージャを実行して得られるファイル plan_for_embedded_select.xml は、ISQL のメニュー・オプション［ファイル］－［開く］を選び、ファイル・フォーマットに XML を指定（*.XML）することで表示できます。

Section

10.6 断片化と再編成

どれだけ慎重に SQL 文を構築しても、ファイル、テーブル、インデックスのいずれかが断片化すると、パフォーマンスの悪化は回避できません。この項では、これら 3 種類の断片化および、それらに対する 4 つの対処法を解説します。

- ファイルの断片化およびディスクの断片化解消ユーティリティ
- テーブルの断片化およびページ・データの物理的編成

- SQL 文を用いたテーブルの再編成
- インデックスの断片化および、深さと密度がパフォーマンスに及ぼす影響
- SQL 文を用いたインデックスの再編成
- アンロードと再ロードを用いた、データベース、テーブル、インデックスの再編成

10.6.1 ファイルの断片化

データベースの物理ファイルを格納するディスク・ドライブが重度に断片化していると、複数のロケーションからデータを収集することになるため、大幅に処理速度が低下します。組み込み関数 DB_PROPERTY を次の例のように使用すると、ファイルの断片化がデータベース・ファイルに影響しているかどうかを確認できます。

```
SELECT DB_PROPERTY ( 'DBFileFragments' ) AS db_file_fragments;
```

理想的なのはまったく断片化していない状態ですが、実際に問題となるのは重度に断片化している場合です。断片化したディスクの解決法はオペレーティング・システムごとに異なります。図 10-18 は、SQL Anywhere のデータベースを含む 13,556 個に断片化したハード・ドライブを Windows 2000 のディスク デフラグ ツールで検査した場合の表示です。

図 10-18　Windows 2000 のディスク デフラグ ツールでの検査結果

ディスクの断片化は、他の種類の断片化より優先的に対処しておく必要があります。オペレーティング・システムによっては、ディスクの断片化解消プログラムとその他の断片化解消プログラムとが、個別のユーティリティに分かれている場合があります。

Tip

ディスクの断片化解消プログラムの中には、既存のファイルは処理できるが空き領域は整理できないタイプがあり、その場合はなんらかのファイルを作成してからでないと、有効に機能しません。たとえば、Windows 2000のディスク デフラグ ツールの場合、このプログラムを実行してから大容量のデータベース・ファイルを作成すると、ファイルが重度に断片化することがあるので、正しい手順としては、最初にデータベースを作成してからディスク デフラグ ツールを実行します。

データベースが巨大化していくと予想される場合は、事前にデータベース・ファイルに空き領域を割り付けてから、ディスクの断片化解消を行います。これは、データベース・ファイル用の連続領域を確保しておくことで、ディスク断片化の再発を防止する措置です。具体的な操作としては、ALTER DBSPACEコマンドを実行します。ALTER DBSPACEコマンドの構文は次のとおりです。

```
<add_space_to_database_file> ::= ALTER DBSPACE <database_file_reference>
                                     ADD <number_of_units> [ <units> ]
<database_file_reference>    ::= SYSTEM       -- プライマリ・データベースの .DB ファイル
                               | TRANSLOG     -- トランザクションの .LOG ファイル
                               | TEMPORARY    -- データベースのテンポラリ・ファイル
                               | <identifier> -- セカンダリ・データベース・ファイルの DBSPACE 名
<number_of_units>            ::= 追加する <units> の整数リテラル値
<units>                      ::= PAGES -- デフォルト
                               | KB
                               | MB
                               | GB
                               | TB
```

次に示すのは、動作中のデータベースに100メガバイトの空き領域を追加する例です。この種のコマンドの実行に要する時間は非常に短く、この場合は数秒程度で完了します。

```
ALTER DBSPACE SYSTEM ADD 100 MB;
```

組み込みストアド・プロシージャのsa_disk_free_spaceを使用すると、個々の物理ファイルが置かれているディスク上にどれだけの空き領域があるかを確認できます。次のクエリは、ディスクの空き領域および各ファイルのファイル仕様を表示するもので、nameカラムの内容は「SYSTEM」がメイン・データベース・ファイル、「Transaction Log」が.logファイル、「Transaction Log Mirror」がミラー・ログ、「Temporary File」がテンポラリ・ファイルを示し、その他はセカンダリのDB領域ファイルの名前（CREATE DBSPACE文で指定したもの）を示します。

```
SELECT sa_disk_free_space.dbspace_name                                  AS name,
       sa_disk_free_space.free_space                                    AS bytes,
       CAST ( TRUNCNUM ( bytes / ( 1024 ), 0 ) AS BIGINT )              AS KB,
       CAST ( TRUNCNUM ( bytes / ( 1024 * 1024 ), 0 ) AS BIGINT )       AS MB,
```

```
       CAST ( TRUNCNUM ( bytes / ( 1024 * 1024 * 1024 ), 0 ) AS BIGINT ) AS GB,
       CASE name
          WHEN 'SYSTEM'                THEN DB_PROPERTY ( 'File' )
          WHEN 'Transaction Log'       THEN DB_PROPERTY ( 'LogName' )
          WHEN 'Transaction Log Mirror' THEN DB_PROPERTY ( 'LogMirrorName' )
          WHEN 'Temporary File'        THEN DB_PROPERTY ( 'TempFileName' )
          ELSE ( SELECT SYSFILE.file_name
                   FROM SYSFILE
                  WHERE SYSFILE.dbspace_name = name )
       END AS file
  FROM sa_disk_free_space();
```

このクエリの出力例を次に示します。名前および空き領域の情報はsa_disk_free_spaceプロシージャで取得しており、ファイル指定の情報は、セカンダリのDB領域「extend」についてはSYSFILEテーブルから、その他のファイルについては組み込み関数DB_PROPERTYで取得しています。

```
name                    bytes        KB        MB     GB  file
================        ==========   ========  =====  ==  ======================
SYSTEM                  3546898432   3463768   3382   3   e:¥¥xdb¥¥demo.db
extend                  12201697280  11915720  11636  11  c:¥¥xtend¥¥extend.db
Transaction Log         12201697280  11915720  11636  11  c:¥¥xlog¥¥demo.log
Transaction Log Mirror  3546898432   3463768   3382   3   e:¥¥xmirror¥¥demo.mlg
Temporary File          3546898432   3463768   3382   3   e:¥¥xtemp¥¥asat0000.tmp
```

Note

sa_disk_free_spaceで取得されるfree_spaceが示すディスクの空き領域とは、特定のファイルに使用されていない領域のことであり、データベース・ファイル内部の空き領域の意味ではありません。このため、同一ディスク上にあるデータベース・ファイルに対してsa_disk_free_spaceを使用すると、すべて同じfree_spaceの値が返されます。

Note

動作中のデータベースにはすべて固有のテンポラリ・ファイルが存在し、これらは必要時に作成され、不必要になった段階で削除されます。

10.6.2 テーブルの断片化

ディスクが断片化しておらず、すべてのデータベース・ファイルがディスク上の連続した領域に格納されていても、データベース内のテーブル・データが重度に断片化している場合があります。ディスクの断片化と同様、テーブルの断片化もパフォーマンスを悪化させるので、その原因を検討しておく必要

があります。

1つのローを挿入すると、そのデータは、単一ページ上の連続した領域内に格納されます。こうしたページが既に別のローのデータを格納していて十分な空き領域が取れない場合は、空のページが使用されます。つまり、ローのデータが1つのページより大きい場合、または255バイトを超過したデータを持つ長いカラムが1つまたは複数存在する場合、こうしたローは2つ以上のページに分割されますが、1つのロー全体が単一のページ内に挿入しきれなくなるのは、このような場合だけです。

Note

異なるテーブルのデータが、同一のページ上に格納されることはありません。また、新規ローが挿入される際は、テーブルのPCTFREEの設定が関係します。空き容量がPCTFREE設定を下回るようなページには新規ローは挿入されません。別の言い方をすれば、PCTFREEで規定された空き容量は後のUPDATE時に利用され、INSERT時に使われることはありません。PCTFREEの詳細については「1.12 空き領域」を参照してください。

UPDATEオペレーションによって、1つのローのサイズがオリジナルのページを超過してしまう場合があります。こうした現象が発生する状況は2つあります。1つ目は、1つまたは複数の短いカラムが該当ページに残された空き領域より大きくなった場合です。2つ目は、1つまたは複数の長いカラムが255バイトを超過した場合です。1つ目の場合、ローは分割されます。2つ目の場合、255バイトを超過した段階でblobカラムが分割されます。

ローの分割は好ましくない状況であり、特に短いカラムに起因したローの分割には注意が必要です。ローの分割はテーブルの断片化を引き起こし、それが重度になれば、処理速度の大幅な低下をもたらします。SELECTでは分割されたローを集めなければならず、UPDATEでは新たにローを分割しなければならないからです。

SQL Anywhereには、テーブルの断片化の度合いを、次の定義に基づいて計測する機能が用意されています。

- 「短いカラム」とは、サイズが254バイトより大きくならないカラムのこと。「長いカラム」とは、これより大きくなるカラムのこと。どちらのタイプのカラムも断片化の原因となるが、短いカラムの引き起こす断片化の方が影響が大きい
- 「ロー・セグメント」とは、1つのページ内に格納されているローの一部または全体のこと。ただし、長いカラムの254バイトを超える部分はこれに該当しない。1つのローは、1つ以上のロー・セグメントで構成される
- 「テーブル・ページ」とは、1つまたは複数のロー・データについて、その先頭部のロー・セグメントを格納しているページのこと
- 各ページに格納される先頭部のロー・セグメントは、その所在が「不変」である。これは、ローの先頭部のロー・セグメントが置かれるページは、テーブル・ページに格納された段階で固定さ

れ、その後行われるUPDATEオペレーションによっても、格納先が別ページに変更されることはないことを意味する。つまり、UPDATEオペレーションによってローが最初のページに収まり切らなくなった場合、そのローの一部が、別ページにある他のロー・セグメントに移動されるが、ローの先頭部だけは、常に最初に配置されたページに留まり続ける。ただし、次の項で解説するREORGANIZE TABLE文を用いた場合のみこの限りではない

- 「ロー残部」とは、ローの中で、先頭のロー・セグメントに収まり切らない部分のこと。ただし、長いカラムの254バイトを超える部分はこれに該当しない。1つのロー残部は、1つまたは複数のロー・セグメントで構成される。ロー残部が生じるのは、短いカラムによるロー分割が行われた場合である
- 「blob継続」とは、ロー内の長いカラムの254バイトを超える部分のこと。長いカラムでは、それぞれ溢れた分のデータがblob継続として分割して扱われる。blob継続が生じるのは、長いカラムによるロー分割が行われた場合である。なお、サイズが255バイトちょうどになるカラムにもblob継続が必要となるが、この場合のblob継続には空のデータが収められる。つまり、長いカラムの先頭255バイト分は、常に短いカラムと一緒にロー・セグメント内に格納される
- 「拡張ページ」とは、ロー残部またはblob継続、あるいはその両方のデータを格納したページのこと。拡張ページは「blobアリーナ」とも呼ばれるが、blobに限定されるものではない。テーブル・ページとは異なり、拡張ページのデータは動的な再編成が行え、各ロー残部をカラム順で連続的に格納できる。また、個々のblob継続は独立して管理されるが、これらも連続的に格納される。ただし、SQL Anywhereはテーブル・ビットマップを用いて拡張ページを効率的に探すことはできない（テーブル・ビットマップはテーブル・ページでのみ使用される）

Tip

UPDATEのパフォーマンスが徐々に低下していった場合、実行プランやインデックスには原因がないことがあります。既に重度に断片化したテーブルで短いカラムのサイズが増加すると、分割状態にあるローがさらに細分化されてゆきます。このため、ロー残部データに連続した順番でアクセスしようとすると、SQL Anywhereは複数のページを切り替えなければならないので、処理に時間を要することがあります。

組み込みストアド・プロシージャのsa_table_fragmentationを用いると、各テーブルにおけるロー当たりの平均ロー・セグメント数を確認できます。次に示すのは、sa_table_fragmentationを使って、DBAに属するすべてのテーブルの状態を表示する例です。

```
CHECKPOINT;
SELECT *
  FROM sa_table_fragmentation ( owner_name = 'DBA' );
```

Note

通常は明示的に CHECKPOINT 文を記述する必要はありません。この例で CHECKPOINT 文を使用したのは、SYSTABLE システム・カタログ・テーブルのロー・カウントを強制的に最新の状態に更新してから、sa_table_fragmentation を実行するためです。

sa_table_fragmentation による計算結果は、次の出力例のような形で得られます。ロー当たりのロー・セグメント数は segs_per_row カラムに示されます。この segs_per_row の値は、1 が最適な状態であり、1 を超えると断片化の発生を意味するので、たとえば 1.5 であれば半数のローが断片化していると読み取れます。この出力例を見ると、child、parent、twide という 3 つのテーブルは非常に高い segs_per_row 値を示しています。この結果から、child と parent では、その大部分のローで短いカラムのロー分割によるロー残部が生じており、また 1 つのローしかない twide では、複数回の分割が行われていることがおおよそ把握できます。残りの normal と tblob テーブルについては、segs_per_row が 1 という最適値を示しているので、ロー残部は生じていません。

```
TableName  rows   row_segments  segs_per_row
=========  =====  ============  ============
child      25000  49073         1.96292
normal     1      1             1.0
parent     5000   9999          1.9998
tblob      1      1             1.0
twide      1      4             4.0
```

Note

大量のディスク I/O が伴うことがあるので、sa_table_fragmentation の実行は時間がかかる場合があります。特に、大きなテーブルが断片化していて、ページの多くがキャッシュされていない場合に顕著になります。

その他の有用な組み込みプロシージャとして sa_table_stats があり、これを次の例のように使用することで、各テーブルのテーブル・ページと拡張ページの数を確認できます。

```
SELECT sa_table_stats.table_name       AS table_name,
       sa_table_stats.count            AS rows,
       sa_table_stats.table_page_count AS table_pages,
       sa_table_stats.ext_page_count   AS ext_pages
  FROM sa_table_stats()
 WHERE sa_table_stats.creator = 'DBA'
 ORDER BY table_name;
```

sa_table_stats の出力例は次のようになります。この例に用いたテーブル群は先に見たものと同じで

すが、そこから予想されるとおり、ロー残部が多い child、parent、twide の 3 つのテーブルは、拡張ページの数も多いことがわかります。その一方で tblob テーブルを見ると、先に確認したようにロー残部は存在しないのに、拡張ページの数は 104 個という高い値になっています。これは、tblob にロー残部はないものの、blob 継続を必要とする長いカラムが 1 つ存在しているためです。

```
table_name  rows   table_pages  ext_pages
==========  =====  ===========  =========
child       25000  3704         49073
normal      1      1            1
parent      5000   360          10013
tblob       1      1            104
twide       1      1            3
```

次のプロシージャは、sa_table_fragmentation と sa_table_stats プロシージャの出力を並べて表示させるもので、ロー・セグメントと拡張ページの情報をまとめて確認できます。

```
CREATE PROCEDURE p_table_fragmentation ( IN @owner_name VARCHAR ( 128 ) )
RESULT ( table_name       VARCHAR ( 128 ),
         rows             UNSIGNED INTEGER,
         row_segments     UNSIGNED BIGINT,
         segments_per_row DOUBLE,
         table_pages      UNSIGNED BIGINT,
         extension_pages  UNSIGNED BIGINT )
BEGIN
SELECT *
  INTO #sa_table_fragmentation
  FROM sa_table_fragmentation ( owner_name = @owner_name );

SELECT #sa_table_fragmentation.TableName     AS table_name,
       #sa_table_fragmentation.rows          AS rows,
       #sa_table_fragmentation.row_segments  AS row_segments,
       #sa_table_fragmentation.segs_per_row  AS segments_per_row,
       sa_table_stats.table_page_count       AS table_pages,
       sa_table_stats.ext_page_count         AS extension_pages
  FROM #sa_table_fragmentation
       INNER JOIN sa_table_stats()
               ON sa_table_stats.table_name = #sa_table_fragmentation.TableName
 WHERE sa_table_stats.creator = @owner_name
 ORDER BY table_name;
END;
SELECT * FROM p_table_fragmentation ( 'DBA' );
```

この p_table_fragmentation による出力は次のようになり、今回も先ほどと同じ 5 つのテーブルを対象にしています。segments_per_row が大きく extension_pages と row_segments がほぼ一致しているテーブル（この場合は child、parent、twide）については、ロー残部による断片化が生じているものと

考えられ、逆に segments_per_row が 1 に近い値を示しているのに拡張ページの数が多いテーブル（この場合は tblob）については、blob 継続による断片化が生じているものと考えられます。

```
table_name  rows   row_segments  segments_per_row  table_pages  extension_pages
==========  =====  ============  ================  ===========  ===============
child       25000  49073         1.96292           3704         49073
normal      1      1             1.0               1            1
parent      5000   9999          1.9998            360          10013
tblob       1      1             1.0               1            104
twide       1      4             4.0               1            3
```

Note

sa_table_stats プロシージャは現行のマニュアルには記載されていませんが、主に現在のキャッシュ情報を確認することを目的に用意されています。前述の p_table_fragmentation プロシージャの場合、sa_table_stats はテーブル・ページと拡張ページの情報を取得するためにだけ呼び出しています。データベースの開始直後はキャッシュが空なので、こうした情報は取得できません。そのため、このプロシージャでは、最初に sa_table_fragmentation を呼び出して強制的にキャッシュに情報を取り込んでから sa_table_stats を呼び出すことで、必要な情報を取得しています。

次のクエリは、ロー当たりの平均セグメント数が 1.05 以上であるか、拡張ページとロー・セグメントとの比が 2 以上であるか、またはその両方の条件に該当するテーブルについて「Fragmentation!（断片化!）」と報告するものです。1 番目の条件ではロー残部によって断片化が発生していることがわかり、2 番目の条件では blob 継続によって断片化が発生していることがわかります。なお、ここで用いた 1.05 および 2 という基準値は任意に設定したものなので、実際にパフォーマンスが低下し始めた場合の数値などに、必要に応じて変更してください。

```
SELECT table_name,
       segments_per_row,
       CAST ( IF rows = 0
                 THEN 0
                 ELSE CAST ( extension_pages AS DOUBLE ) / row_segments
              ENDIF AS FLOAT ) AS extension_pages_per_segment,
       IF segments_per_row >= 1.05
          THEN 'Fragmentation!'
          ELSE IF extension_pages_per_segment >= 2
               THEN 'Fragmentation!'
               ELSE ''
          ENDIF
       ENDIF AS fragmentation
  FROM p_table_fragmentation ( 'DBA' )
 ORDER BY table_name;
```

このクエリを実行すると、「Fragmentation!」という報告が次のように出力され、断片化していないテーブルがnormalだけであることを簡単に確認できます。

```
table_name  segments_per_row  extension_pages_per_segment  fragmentation
==========  ================  ===========================  ==============
child       1.96292             1.0                        Fragmentation!
normal      1.0                 1.0
parent      1.9998              1.0014001                  Fragmentation!
tblob       1.0               104.0                        Fragmentation!
twide       4.0                 0.75                       Fragmentation!
```

Note

拡張ページはロー残部とblob継続の両方で使用されますが、どちらで使用されているかを確認する方法はありません。前述の「Fragmentation!」クエリの機能は完全なものではありませんが、問題のあるテーブルを見逃したり、誤った警告を表示する頻度はそれほど高くないはずです。

10.6.3 テーブルの再編成

テーブルの断片化への対策の1つは、定期的にREORGANIZE TABLE文を実行することです。REORGANIZE TABLE文の構文は、次のとおりです。

```
<reorganize_table> ::= REORGANIZE TABLE [ <owner_name> "."] <table_name>
```

このREORGANIZE TABLE文は、ローの削除と再挿入をすることで、すべてのテーブルのページを再構築します。ローの挿入順の指定には、クラスタード・インデックスが存在する場合はそれが用いられ、存在しない場合はプライマリ・キーが用いられます。このため、プライマリ・キーもクラスタード・インデックスも存在しないテーブルの場合、REORGANIZE TABLEは機能しません。

REORGANIZE TABLEのプロセスでは、ある程度まとまった量のローを対象として削除と再挿入が行われ、その間、該当テーブルに排他ロックがかけられます。それらのローに対する処理が完了すると、いったん排他ロックが解除され、再編成プロセスでブロックされていたその他のプロセスがあれば、必要な処理が行えるようになります。そして、しばらくすると、再編成プロセスが再びテーブルに排他ロックをかけ、次の一群のローの処理を開始します。このプロセスが元で他の接続が完全に使用不能になることはありませんが、高負荷なサーバの同時実行性を低下させることはあるので、REORGANIZE TABLEの実行は負荷の高い時間帯を避けるべきです。

Tip

作業完了まで長時間を要するREORGANIZE TABLE文を実施する場合は、その直前に「SET

TEMPORARY OPTION BACKGROUND_PRIORITY = 'ON'」という文を実行してください。これは、優先度の高い他の接続がある場合、それらの作業が進められるよう、この接続の持つ制御をある程度ゆずってもかまわない、ということを意味します。実際この方法はREORGANIZE TABLEに限定されるものではなく、作業量は多いが迅速な応答は必要としないタイプの接続全般に適用できます。

REORGANIZE TABLE文は、トリガの起動やトランザクション・ログへの書き込みは何も行いません。論理的な観点からすると、これはデータをページ間で移動させているだけであり、データの内容的な変更は行わないからです。ただし、データベースのページは変更されるので、次のチェックポイントでこれらをデータベース・ファイルに書き出すため、チェックポイント・ログは大きくなってゆきます。

この項で用いた形式のREORGANIZE TABLEは、対象テーブルのインデックスには何の作用も及ぼしません。インデックスの断片化の詳細については、次の項を参照してください。

次の例は、REORGANIZE TABLEの効果を確認するため、この文の実行後にp_table_fragmentationを呼び出すものです。

```
REORGANIZE TABLE child;
CHECKPOINT;
SELECT * FROM p_table_fragmentation ( 'DBA' );
```

p_table_fragmentationが報告するchildテーブルの状態は、REORGANIZE TABLEの実施前後で、次のように変化しています。segments_per_rowは理想値である1.0にまで下がっており、ロー残部がすべて取り除かれたことがわかります。同様に、拡張ページは実施前の半分に低下していますが、この残りの拡張ページはblob継続に必要な分です。

```
table_name  rows   row_segments  segments_per_row  table_pages  extension_pages
==========  =====  ============  ================  ===========  ===============
child       25000  49073         1.96292           3704         49073      -- 実施前
child       25000  25000         1.0               25001        25008      -- 実施後
```

テーブルの断片化を事前に予防する、または発生時の影響を最小限に抑えるには、いくつかの方法があります。

- ローを空の状態で挿入して、空でないダミー値ですぐに更新するという処理をしない。INSERTをする時点で実際の値が確定しているのであれば、それを使用する
- 最終的にNULL以外の値を用いて更新するカラムには、NULL値を入れない。NULL値のカラムは記憶領域をまったく消費しないので、その後NULL以外の値を入れて更新すると、必然的にロー・サイズの増加が発生する
- 最終的に長い文字列に更新するカラムには、空ないし短い文字列を入れない。実際に記憶させる文字列を挿入するか、初期値としてブランクを埋め込んでおく

- ロー・サイズは更新をするごとに増えるので、多数のローを繰り返し更新するような操作は避ける。通常この種の処理はテーブルを重度に断片化させるので、更新をするごとに処理完了までの時間が長くなり、最終的には実用に耐えないレベルまで処理速度が低下する可能性がある
- 短いカラムの更新でロー・サイズが増える傾向にあるテーブルについては、PCTFREE の指定値を大きくしておく。PCTFREE の詳細については「1.12　空き領域」を参照
- 使用頻度の高い短いカラムはローの先頭部に配置しておき、ローの分割時にこれらのカラムのデータが拡張ページに移動するのを防ぎ、オリジナルのテーブル・ページに止まるようにする

10.6.4　インデックスの断片化

インデックスの断片化には、ファイルやテーブルの断片化と異なる側面と類似した側面があります。インデックスの断片化とファイルやテーブルの断片化とでは、発生原因および測定法が異なるので、その対策も異なりますが、パフォーマンスの面では、いずれも好ましくない影響を及ぼすという点で同じです。

インデックスのエントリは、テーブルのデータと同様に、ページ上に格納されます。すべてのインデックスは単一の最上位ページ(レベル1)から始まり、このページはほぼ確実にキャッシュに取り込まれます。エントリが追加されるに従い、ページのツリー構造が広く浅く広がる形で、レベル2以降の階層が形成されていきます。こうしたレベルの数は「深さ」と呼ばれます。このレベルの数が大きいほど、隣接レベル間でのページ移動に余分のディスクI/Oが必要になるため、インデックスの処理速度は低下します。

最適なパフォーマンスが得られるのは、すべてのページがインデックス・エントリで満たされ、ツリー全体のバランスが取れている場合です。インデックスの深さを決めるのは、基本的にはインデックス・エントリのサイズとデータベースのページ・サイズです。ただし、インデックスのバランスが崩れたりページの一部が空になると、実際のデータ量から必要とされる値よりも、インデックスの深さが過剰に大きくなります。これがインデックスの断片化で、こうした状態に陥るとパフォーマンスが低下します。

インデックスの断片化の度合いは、2つの組み込みストアド・プロシージャを用いて計測できます。その1つはsa_index_levelsと呼ばれるものです。次の例では、このプロシージャを利用してDBAに属するすべてのテーブルのインデックスを表示させています。

```
SELECT *
  FROM sa_index_levels ( owner_name = 'DBA' )
 ORDER BY sa_index_levels.TableName,
       sa_index_levels.IndexName;
```

sa_index_levelsからは次のような出力が得られます。ここで用いたサンプル・データベースの場合、5つのテーブルのプライマリ・キー・インデックスおよび1つの外部キー・インデックスが表示されています。ここで、childテーブルのプライマリ・キー・インデックスが3レベルであるという情報は、あまりよくない兆候です。データベースのページ・サイズが4K、プライマリ・キーはデータ消費の少ないINTEGERカラム、テーブル中のロー数が高々25,000という場合は2レベルもあれば十分なはずなのに、

余分のレベルが追加されているということは、インデックスの使用に要するディスクI/Oが倍増している可能性があります。

```
TableName  IndexName  Levels
=========  =========  ======
child      child      3      -- childテーブルのプライマリ・キー・インデックス
child      parent     2      -- childテーブルの外部キー・インデックス
normal     normal     1      -- normalテーブルのプライマリ・キー・インデックス
parent     parent     2      -- parentテーブルのプライマリ・キー・インデックス
tblob      tblob      1      -- tblobテーブルのプライマリ・キー・インデックス
twide      twide      1      -- twideテーブルのプライマリ・キー・インデックス
```

Tip

インデックスは、2レベルなら良好、3レベルなら不良、4レベルなら劣悪です。

もう1つの組み込みストアド・プロシージャはsa_index_densityと呼ばれるものです。これはインデックス・ページが平均してどの程度満たされているかの指標である、密度という情報を各インデックスごとに取得します。また、このプロシージャは各インデックスのリーフ・ページ数、つまりインデックス・エントリを持つインデックス・ページの数も取得できます。sa_index_densityは次の例のように使用します。

```
SELECT *
  FROM sa_index_density ( owner_name = 'DBA' )
 ORDER BY sa_index_density.TableName,
       sa_index_density.IndexName;
```

密度の値は、1.0に近ければ良好、0.25ならかなり悪いと見なせます。次の例は、先と同じ6つのインデックスに対するsa_index_densityの出力例です。childテーブルのプライマリ・キー・インデックスは密度が0.25を下回っていますが、これは各ページを平均するとほとんど空であることを意味し、そのためインデックスが2レベルでなく3レベルになっていたことがわかります。

```
TableName  IndexName  LeafPages  Density
=========  =========  =========  ========
child      child      461        0.249984
child      parent     218        0.263677
normal     normal     1          0.014893
parent     parent     34         0.478322
tblob      tblob      1          0.014893
twide      twide      1          0.014893
```

次のプロシージャは、sa_index_levelsとsa_index_densityを組み合わせて、より扱いやすい形に出

力を整えたものです。またここでは、SYSTABLEからロー・カウントを取得して、「concerns」カラムに潜在的な問題の有無を表示させています。

```
CREATE PROCEDURE p_index_fragmentation ( IN @owner_name VARCHAR ( 128 ) )
RESULT ( table_name       VARCHAR ( 128 ),
         index_name       VARCHAR ( 128 ),
         rows             UNSIGNED BIGINT,
         leaf_pages       UNSIGNED INTEGER,
         levels           INTEGER,
         density          NUMERIC ( 8, 6 ),
         concerns         VARCHAR ( 100 ) )
BEGIN
SELECT sa_index_levels.TableName  AS table_name,
       sa_index_levels.IndexName  AS index_name,
       SYSTABLE.count             AS rows,
       sa_index_density.LeafPages AS leaf_pages,
       sa_index_levels.Levels     AS levels,
       sa_index_density.Density   AS density,
       STRING (
           IF levels > 2
              THEN 'deep'
              ELSE ''
           ENDIF,
              IF levels > 1 AND density < 0.5
                 THEN IF levels > 2
                         THEN ', low density'
                         ELSE 'low density'
                      ENDIF
                 ELSE ''
              ENDIF ) AS concerns
  FROM sa_index_levels ( owner_name = @owner_name )
       INNER JOIN sa_index_density ( owner_name = @owner_name )
               ON sa_index_density.TableName = sa_index_levels.TableName
               AND sa_index_density.IndexName = sa_index_levels.IndexName
       INNER JOIN SYSTABLE
               ON SYSTABLE.table_name = sa_index_density.TableName
 WHERE USER_NAME ( SYSTABLE.creator ) = 'DBA'
 ORDER BY table_name,
       index_name;
END;
```

このp_index_fragmentationは、次のように使用します。ここでCHECKPOINT文を使用しているのは、強制的に最新のロー・カウント値をSYSTABLE.countカラムに格納させるためです。

```
CHECKPOINT;
CALL p_index_fragmentation ( 'DBA' );
```

p_index_fragmentation による出力は、次の例のようになります。「concerns」カラムで「deep」と表示されているのは、レベルが2より大きい場合で、「low density」と表示されているのは、レベルが1より大きいインデックスで密度が0.5より小さい場合です。

```
table_name  index_name  rows   leaf_pages  levels  density   concerns
==========  ==========  =====  ==========  ======  ========  ================
child       child       25000  461         3       0.249984  deep, low density
child       parent      25000  218         2       0.263677  low density
normal      normal      1      1           1       0.014893
parent      parent      5000   34          2       0.478322  low density
tblob       tblob       1      1           1       0.014893
twide       twide       1      1           1       0.014893
```

ローが削除されても、SQL Anywhere はインデックスのバランスを再調整することはしません。これがインデックスが断片化する最大で唯一の理由であり、上記の例にある child および parent テーブルのインデックスが断片化したのもこのためです。レベルと空き率の高いインデックス・ページが数多く含まれる巨大なインデックスが使用されると、そのクエリ処理のパフォーマンスは確実に悪化します。

10.6.5 インデックスの再編成

インデックスの再編成には、次の3つの形式の REORGANIZE TABLE 文を使用できます。

```
<reorganize_index> ::= REORGANIZE TABLE [ <owner_name> "."] <table_name>
                           PRIMARY KEY
                     | REORGANIZE TABLE [ <owner_name> "."] <table_name>
                           FOREIGN KEY <constraint_or_role_name>
                     | REORGANIZE TABLE [ <owner_name> "."] <table_name>
                           INDEX <index_name>
<constraint_or_role_name> ::= CONSTRAINT または役割名として使用する <identifier>
<identifier>              ::=「第1章　作成」の <identifier> を参照
```

1番目の形式にあるキーワード PRIMARY KEY は、再編成の対象がプライマリ・キー・インデックスであることを指定します。2番目の形式にあるキーワード FOREIGN KEY は、外部キー関係の役割名と組み合わせることで、再編成の対象が外部キー・インデックスであることを指定します。3番目の形式にあるキーワード INDEX は、インデックス名と組み合わせることで、再編成の対象が通常の非キー・インデックスであることを指定します。なお、これらはいずれも「REORGANIZE TABLE」という名前の文を使用していますが、実際に変更されるのはテーブル・データではありません。クラスタード・インデックスが使用されている場合であっても、操作の対象はあくまでインデックス・ページであり、テーブル・ページは何の変更も受けません。

Note

3番目の形式のREORGANIZE TABLEは、UNIQUE制約によって自動的に作成されたインデックスの再編成に使用することもできます。たとえば、childテーブルに「UNIQUE (key_2, key_3)」という制約が指定されていた場合、関連するインデックスを再編成すには、「REORGANIZE TABLE child INDEX "child UNIQUE (key_2,key_3)"」とします。使用すべきインデックス名については、前項で解説したsa_index_densityプロシージャを用いれば簡単に確認できます。

次の3つのコードはREORGANIZE TABLE文の使用例で、ここでは前項で取り上げた3種類の断片化インデックスを再編成しています。外部キーの役割名は、デフォルトで関係先のテーブル名が使用されます(ここではparent)。この例の場合childテーブルの外部キー関係にあるテーブルの名前は「parent」なので、2番目のREORGANIZE TABLEにおける役割名としてもこの名前を用いています。

```
REORGANIZE TABLE child PRIMARY KEY;
REORGANIZE TABLE child FOREIGN KEY parent;
REORGANIZE TABLE parent PRIMARY KEY;
```

3つのインデックスの再編成後にp_index_fragmentationプロシージャを呼び出すと、次のような出力が得られます。これを見ると、リーフ・ページ数が減少し、密度が高まっていますが、これはインデックス・エントリをまとめてページを充填するようにした結果です。childテーブルの外部キー・インデックスについてはレベルが3から2に下がっており、すべての「concerns」カラムから潜在的な問題を告げる表示が消えています。

```
table_name  index_name  rows   leaf_pages  levels  density   concerns
==========  ==========  =====  ==========  ======  ========  =================
child       child       25000  111         2       1.001269
child       parent      25000  56          2       0.993517
parent      parent      5000   16          2       1.003250
```

REORGANIZE TABLEによるインデックスの再編成は、テーブルそのものの再編成ほど長時間を要しないので、他の接続のパフォーマンスに悪影響を及ぼす可能性もそれほど高くありません。この処理は、複数のインデックス・ページ間でインデックス・エントリを移動させるのではなく、インデックスを最初から再構築します。このため、高負荷なサーバ上でもインデックスを再編成できる場合があります。

10.6.6 アンロード／再ロードによるデータベースの再構築

データベース全体を再構築する究極の方法としては、すべてのデータをいったんアンロードしてからデータベースを最初から再作成した後に全データを再ロードするという、力押し的な手法があります。これは、データベース・ファイルで使用されるディスク領域を削減する唯一の方法でもあり、その他の方法では、ファイル内データの再構築はできても、ファイル・サイズの削減までは行えません。

アンロード／再ロードによるデータベースの再構築は、次のステップで行えます。なおこのステップは、ある程度の期間データベースをオフラインにすることを前提にしています。

❶ 処理対象のデータベースから、他のすべてのユーザの接続を切断します。
❷ dbunload.exe を用いて処理対象のデータベースをアンロードします。
❸ 処理対象のデータベースを停止します。
❹ 処理対象のデータベースをバックアップしてから削除します。
❺ dbinit.exe を用いて新しいデータベースを作成します。
❻ 新しいデータベースを起動します。
❼ ISQL で ALTER DBSPACE を実行して、新しいデータベース・ファイルのサイズを増やします。
❽ ハード・ドライブの断片化を解消します。
❾ 必要に応じて reload.sql ファイルの確認と編集をします。
❿ ISQL で reload.sql を実行して、新しいデータベースをロードします。
⓫ 異常がないかを確認します。
⓬ 新しいデータベースを使用可能にします。

次に、各ステップの詳細を説明します。

ステップ1：処理対象のデータベースから、他のすべてのユーザの接続を切断します。ここでは、すべてのデータをアンロードしてデータベース全体を削除しますが、他の接続経由の変更でロール・バックされるものや、テーブルのアンロード後にコミットされる変更は失われます。また、他の接続がロックをかけているとアンロード・プロセスがブロックされる場合や、その逆も発生します。

Tip

ネットワーク環境の場合、他のユーザがデータベースに接続できないようにする手段として、ネットワーク・サーバdbsrv9.exeを停止してからスタンドアロン・エンジンdbeng9.exeを用いてデータベース・ファイルを再起動する、という方法が使えます。スタンドアロン・エンジンでは、ネットワーク経由の接続はできません【監注17】。

次の例は、dbeng9.exe を用いてデータベースを起動する Windows バッチ・ファイルです。「-x none」パラメータは、エンジンの高速起動用に共有メモリ接続のサポートだけをロードさせる指定です（TCP/IP などは除外）。

```
"%ASANY9%¥win32¥dbeng9.exe" -x none volume.db
```

【監注17】「dbsrv9 -x tcpip(LocalOnly=yes)」のようにLocalOnlyオプションを使用すれば、ローカルマシンのみ接続可能なネットワーク・サーバが起動する。

ステップ2：SQL Anywhereに用意されているdbunload.exeプログラムを用いて、処理対象のデータベースにあるデータのアンロードとテキスト・ファイルへの保存、および、新規データベースへの再ロードに用いるreload.sqlコマンド・ファイルの構築を行います。可能であれば、アンロードしたデータ・ファイルを格納するディスク・ドライブと、データベース本体のファイルを格納するディスク・ドライブとで、異なる物理ドライブを使用できると、アンロードおよび再ロードの処理が高速化されます。

次の例は、dbunload.exeを実行するWindowsバッチ・ファイルです。-cパラメータは、dbunload.exeによるデータベース接続法に関する指定です。ここでODBC DSNを用いているのは、他の接続パラメータを検出するためだけの便宜上の処置で、実際にODBCを用いて接続するわけではありません。-rオプションは、後にデータベースの再ロードに用いる.sqlコマンド・ファイルの格納先で、dbunload.exeを実行するコンピュータに対する相対位置で指定します。最後のパラメータは、アンロードしたテーブル・データの格納先で、テキスト形式の.datファイルを収めるドライブとフォルダを指定します。このパラメータは、dbunload.exeの実行コンピュータではなく、データベースの実行コンピュータに対する相対位置で指定します。この例では、わかりやすさと速度を考慮して、データベース・エンジンが動作しているコンピュータ上でdbunload.exeを実行します。また、reload.sqlファイルおよびアンロードしたデータ・ファイルの格納先についても共通のc:¥tempを使用するよう指定しています。

```
"%ASANY9%¥win32¥dbunload.exe" -c "DSN=volume" -j 3 -r c:¥temp¥reload.sql c:¥temp
```

これによって生成されるreload.sqlファイルには、データベース・スキーマの再作成に必要となるCREATE TABLEをはじめ、その他すべてのSQL文が記述されています。その中には、dbunload.exeがテーブル・データの格納に用いた.datファイルの参照情報と共に、各テーブル用のLOAD TABLE文も指定されます。次に示すのは、実際に生成されたreload.sqlファイルの一部で、テーブル名「parent」に対するLOAD TABLE文を抜粋したものです。ファイル参照先はc:¥temp¥428.datであることがわかります。

```
LOAD TABLE "DBA"."parent" ("key_1", "non_key_1_1", "non_key_1_2", "non_key_1_3",
   "non_key_1_4", "non_key_1_5", "non_key_1_6", "non_key_1_7", "non_key_1_8",
   "non_key_1_9", "non_key_1_10", "non_key_1_11", "non_key_1_12", "non_key_2",
   "non_key_3", "index_1", "index_2" )
      FROM 'c:¥¥temp¥¥428.dat'
      FORMAT 'ASCII' QUOTES ON
      ORDER OFF ESCAPES ON
      CHECK CONSTRAINTS OFF COMPUTES OFF
      STRIP OFF DELIMITED BY ','
```

LOAD TABLE文の詳細については「2.3　LOAD TABLE」を参照してください。

上記のdbunload.exeのコマンド・ラインで用いた「-j 3」オプションは、相互依存関係にある複数のビュー定義に対する処理の方法を指定しています。たとえば、あるビューが別のビューを参照する場合、後者のビューがreload.sql中で後から定義されるようになっていると、最初のCREATE VIEW文の段階で処理が失敗します。-jオプションは、すべてのCREATE VIEW文のコピーを複数作成させる指定

で（この場合は3つ）、これによってどのビューも、いずれか1つのCREATE VIEW文で処理が成功するようになります。残りのCREATE VIEW文については、ビューが既に作成されている、または作成されていないビューを参照しようとしている状況になるため、これらの処理は失敗します。ここではすべてのビューが正常に作成できれば問題はないので、これらの失敗は特に気にする必要はありません。

ステップ3：データベース・サーバを停止するか、または処理対象のデータベースだけをサーバ上で停止させます。以降、処理対象のデータベースは不要になり、次のステップではデータベースのバックアップと削除を行うので、その前に停止しておく必要があります。

ステップ4：処理対象のデータベースをバックアップしてから、データベース・ファイルおよびトランザクション・ログを削除します。ここでの操作は、ファイル名を変更するか、別のディスク・ドライブやフォルダに移動しても同じ結果が得られます。

ステップ5：dbinit.exeを用いて新しいデータベースを作成します。このプログラムは、データベース・エンジンと同一のコンピュータ上で実行する必要があります。次の例は、ページ・サイズ4Kでデータベースを作成するWindowsバッチ・ファイルです。

```
"%ASANY9%¥win32¥dbinit.exe" -p 4096 volume.db
```

Tip

dbinit.exeには、-pオプションを用いてページ・サイズを明示的に指定する必要があります。データベース・ファイルのアンロードと再ロードをしない限り、いったん指定したページ・サイズは変更できません。必要量が不明であれば、4096または4Kを指定してください。デフォルトのページ・サイズでは2Kが適用されますが、通常は最適な結果が得られません。その他の指定可能な値である1K、16K、32Kなどはさらに問題があるので、可能な限り使用しないでください。ただし、8192または8Kという指定で良好な結果が得られる場合もあります。これは、1つのページにより多くのインデックス・エントリを収めることで、巨大なテーブルに付随するインデックスのレベルを低くできるような場合です。なお、1つのページに収められるインデックス・エントリの数に制限はありませんが、1ページ中のデータ数には最大255ローという制限があります。つまり、ローの平均長が16バイトであれば4Kのページ・サイズがちょうどいっぱいになりますが、平均長32バイト以上のローを運用するのでなければ、8Kのページ・サイズを用意しても無駄になるだけです。その他の目安としては、高負荷なテーブルのロー数が500,000以上になるまで8Kのページ・サイズを検討する必要はない、という経験則があります。

Tip

同じサーバ上で複数のデータベースを運用する場合は、すべてのデータベースでページ・サイズを共通にしてください。こうしておかないと、小サイズのページをメモリにロードするのに大きなページ・サイズが使われて、キャッシュ領域が無駄に使用される状況が生じます。2つのデータベースを異なるページ・サイズで運用する場合は、サーバを2つ用意することで、それぞれのキャッシュを効率的に管理できます。動的キャッシュ・サイズ決定を用いれば、高負荷な方のサーバを優先してキャッシュを大きくし、他方のキャッシュを小さくすることが可能です。データベースのページ・サイズが同じであればサーバを2つ用意する必要はありません。1つのサーバ上で2つのデータベースを運用してください。

ステップ6：新しいデータベースを起動しますが、この時点ではまだ他のユーザからの接続は許可しません。

ステップ7：ISQLでALTER DBSPACEを実行して、新しいデータベース・ファイルのサイズを増やします。これは、次のステップで行うディスクの断片化解消後にデータベース用の連続領域を確保するための措置で、ディスクの断片化による悪影響を防止します。また、データベース・ファイルを小さな増分で何度も繰り返し拡張する必要がなくなるので、再ロードの処理も高速化します。

次の例は、ISQLを対話型モードで開始させるWindowsバッチ・ファイルです。ここでODBC DSNを用いているのは接続パラメータ（サーバおよびデータベース名、ユーザID、パスワード）を取得するための処置で、実際にISQLがODBCを用いて接続するわけではありません。

```
"%ASANY9%¥win32¥dbisql.exe" -c "DSN=volume"
```

次の例は、ALTER DBSPACE文を用いてメインのデータベース・ファイルに800メガバイトを追加する場合のコードです。

```
ALTER DBSPACE SYSTEM ADD 800 MB;
```

ALTER DBSPACEの詳細については、この章の「10.6.1　ファイルの断片化」を参照してください。

ステップ8：ハード・ドライブの断片化解消を実行します。ディスクが断片化するとパフォーマンスが悪化するので、この機会に断片化を取り除いておきます。この処理をデータベース・サイズを拡張（ステップ7）した後に実施するのは、ディスクの断片化解消ツールによっては、既存のファイルに対してのみ有効な場合があるためです。

ステップ9：reload.sqlファイルの論理構成を確認し、必要に応じて編集を加えます。先のステップ2以降であれば、この処理はどの段階で実施しても同じであり、行う行わないも任意です。ただし、データベースを長期間使用していると「スキーマ変更」が生じて、データベース設計上の不整合性やエラーが出てきます。reload.sqlというテキスト・ファイルの形でスキーマ全体を確認できるのは、検証や修正を加えるよい機会でもあります。

問題の性質によっては、無用なCHECK制約を解除する、不要なユーザIDを削除する、オプション設定を修正するなどの簡単な措置で対処できるものがあります。しかし、その他の問題はそれほど簡単に解決できません。たとえば、テーブルにカラムを追加することはできても、CREATE TABLE文にあるカラムを削除した場合は、対応するLOAD TABLE文の変更も検討する必要があります。特殊なキーワード「filler()」を用いてカラムの読み込みをスキップさせる方法の詳細については「2.3　LOAD TABLE」を参照してください。

Tip

この段階で、データベース・オプションのOPTIMIZATION_GOALの設定値を入念にチェックしてください。reload.sqlファイル中に「SET OPTION "PUBLIC"."OPTIMIZATION_GOAL" = 'all-rows'」という文が存在し、これが自分にとって必要な設定であることを確認します（通常は必要です）。再ロードをするとこのオプションが'first-row'に設定される場合があるので、アンロードおよび再ロードをして古いバージョンからアップグレードした後は、これらの値の確認が必要です。

ステップ10：ISQLでreload.sqlを実行して、データベースを再ロードします。ステップ2および8（アンロードと断片化解消）も時間のかかる処理ですが、おそらくはすべてのステップでここが最も長時間を要する処理のはずです。次の例は、ISQLをバッチ・モードで開始させるWindowsバッチ・ファイルです。ここでは、ユーザによる介入なしにreload.sqlファイルを即座に実行させています。

```
"%ASANY9%¥win32¥dbisql.exe" -c "DSN=volume" c:¥temp¥reload.sql
```

Tip

dbunload.exeの-ac、-an、-arオプションは使用しないでください。これらは半自動的にアンロードと再ロードを行うオプションですが、かなりの確率で問題が発生するため、かえって非効率です。これらはいわばオールオアナッシング式のアプローチであり、すべての処理が順調に進めば問題はないのですが、処理の途中で何かが失敗すると、すべてを最初からやり直すよう何度も要求されるので、その点が非常に厄介です。それに対して、ここで解説するステップバイステップ式のアプローチならば、問題が生じても最初からすべてを繰り返すのではなく、その1つ前の段階からやり直すだけで済みます。こうしたアプローチの違いは、アンロードや再ロードに数時間を要する大型のデータベースを扱う場合や、限られた時間内に作業を完了させなければならない場合に、特に顕著になります。

ステップ11：異常がないかを確認します。次の例は、ファイル、テーブル、インデックスが断片化していないかを確認するための文で、ISQLから実行できます。

```
SELECT DB_PROPERTY ( 'DBFileFragments' ) AS db_file_fragments;
CHECKPOINT;
SELECT * FROM p_table_fragmentation ( 'DBA' );
CALL p_index_fragmentation ( 'DBA' );
```

これを実行すると、次のような出力が得られます。まずわかることは、ディスク上に存在するのは1つの連続領域であり、その中に800MBのデータベース・ファイルが収まっているので、これは好ましい状態です。次に、いずれのアプリケーション・テーブルも各ローの持つロー・セグメントは1つだけであることがわかります。つまり、短いカラムによるロー分割が行われていないことを意味するので、これも好ましい状態です。その一方で、拡張ページは多数存在していますが、この場合は長いカラムの値を格納する必要性から生じたものです（blob）。最後にわかるのは、すべてのインデックスがレベル2

以内に収まっていること、そしてこれらの密度が1に近いことであり、いずれもインデックスが好ましい状態にあることを示しています。

```
db_file_fragments
=================
1

table_name  rows   row_segments  segments_per_row  table_pages  extension_pages
==========  =====  ============  ================  ===========  ===============
child       25000  25000         1.0               25000        25000
parent      5000   5000          1.0               5000         5000

table_name  index_name  rows   leaf_pages  levels  density   concerns
==========  ==========  =====  ==========  ======  ========  =================
child       child       25000  116         2       0.958616
child       parent      25000  58          2       0.959599
parent      parent      5000   17          2       0.944925
```

ステップ12：この段階で、新しいデータベースを他のユーザが使用できるようにします。通常の起動に用いているのがdbsrv9.exeであれば、これを使用します。次の例は、TCP/IP接続をサポートするネットワーク・サーバを起動させるWindowsバッチ・ファイルです。

```
"%ASANY9%¥win32¥dbsrv9.exe" -x tcpip volume.db
```

Section

10.7 インデックスの作成

インデックスはクエリのパフォーマンスを向上させますが、そうした効果には、FROM、WHERE、HAVING句の述部評価の高速化、必要となるテンポラリ・ワーク・テーブルの削減、ORDER BYやGROUP BY句によるソートの省略、MAXやMINなどの集合関数計算の高速化、および独立性レベルを高めた場合に必要なロック数の削減など、さまざまな要因が寄与しています。

インデックスの中には自動的に生成されるものがあり、PRIMARY KEY制約やUNIQUE制約を定義するとユニーク・インデックスが作成され、個々の外部キー制約にはユニークでないインデックスが作成されます。それ以外のインデックス作成には、次の例のように明示的な指定が必要です。

```
<create_index>      ::= CREATE
                        [ UNIQUE ]
                        [ CLUSTERED | NONCLUSTERED ]
                        INDEX <index_name>
```

```
                          ON [ <owner_name> "."] <table_name>
                          <index_column_list>
                          [ <in_dbspace_clause> ]
<index_name>        ::= テーブル中でユニークな値を指す <identifier>
<owner_name>        ::= <identifier>
<table_name>        ::= <identifier>
<index_column_list> ::= "(" <index_column> { "," <index_column> } ")"
<index_column>      ::= <existing_column_name> [ ASC | DESC ]
                      | <builtin_function_call> AS <new_column_name>
<builtin_function_call>  ::= <builtin_function_name>
                             "(" [ <function_argument_list> ] ")"
<builtin_function_name>  ::= SQL Anywhere のスカラ関数名を指す <identifier>
<function_argument_list> ::= <expression> { "," <expression> }
<expression>             ::=「第3章　選択」の <expression> を参照
<existing_column_name>   ::= テーブル中の既存カラムを指す <identifier>
<new_column_name> ::= テーブルに追加する COMPUTE カラムの名前を指す <identifier>
<in_dbspace_clause> ::= ( IN | ON ) ( DEFAULT | <dbspace_name> )
<dbspace_name>      ::= <identifier> -- DEFAULT 名は SYSTEM
```

明示的に作成するインデックスには、<index_name> として、テーブル内でユニークな値を割り当てる必要があります。この制限は、SQL Anywhere が自動作成するインデックス名には当てはまりません。このように作成されるインデックス名は、組み込みプロシージャの sa_index_levels や sa_index_density、または「10.2　インデックスの断片化」で説明した p_index_fragmentation で確認できます。これらのインデックス名は、次の処理で作成されます。

- PRIMARY KEY のインデックス名は、常にテーブル名と同じ名前になる。これは明示的に CONSTRAINT 名を指定しても変更されない
- FOREIGN KEY のインデックス名は、役割名が定義されていればその名前が使われ、CONSTRAINT 名が定義されていればその名前が使われる。それ以外の場合は、外部キー関係にある親テーブル名前が使われる
- UNIQUE 制約のインデックス名は、CONSTRAINT 名が定義されていればその名前が使われる。それ以外の場合は「t1 UNIQUE (c1,c2)」のような名前となる。ここで「t1」はテーブル名、「c1,c2」は UNIQUE 制約のカラム名リストである

Tip

ユーザ指定するインデックスには、自動的に作成されるインデックスの名前と重ならないもので、意味のわかりやすい名前を付けてください。インデックスの作成目的がわかりやすい名前を付けておくと、後で確認する場合や、sa_index_levels などのプロシージャでの出力結果を解釈する場合に便利です。

1つのテーブル内において、1つまたは複数のカラムに対してインデックスを作成します。2つのインデックスが同じカラム群を参照している場合、これらのカラムの組み合わせおよびカラムの順序が同じ

で、各カラムのソート指定（ASC や DESC）も同じであれば、両者は実質的に同一のインデックスであり、単に重複して存在しているだけです。それ以外の場合、これらのインデックスはそれぞれ別のものであり、異なる目的に有効に利用されます。

UNIQUE キーワードは、インデックスのカラムに対し、テーブル中のローがすべて異なる値でなければならないことを指定します。ただし、インデックスのカラムにおける NULL 値は、他のいずれのローの内容にも一致しない値として扱われ、NULL 値どうしも異なる値として見なされます。このため、NULL 値を許可するカラムを元に UNIQUE インデックスを作成すると、文字どおりの「ユニーク」な状態にならない場合があります。たとえば、次の INSERT 文を実行してもエラーになりませんが、これはインデックスのカラムの 1 つが NULL 入力可能であるため、複数の NULL 値があっても「ユニーク」という条件が満たされるからです。

```
CREATE TABLE t1 (
   key_1       INTEGER NOT NULL PRIMARY KEY,
   ikey_1      INTEGER NOT NULL,
   ikey_2      INTEGER NULL );
CREATE UNIQUE INDEX index_1 ON t1 ( ikey_1, ikey_2 );
INSERT t1 VALUES ( 1, 1, 1 );
INSERT t1 VALUES ( 2, 1, NULL );
INSERT t1 VALUES ( 3, 1, NULL );
```

Note

SQL Anywhere では、UNIQUE インデックスで複数の NULL 値の存在を許すよう拡張されていますが、これは ANSI SQL:1999 標準とは異なっています。

UNIQUE インデックスを作成するカラムの NOT NULL 指定は、より厳しい制限をカラム値に課すため、高いパフォーマンスを必要とするクエリでよく使用されます。

Note

UNIQUE 制約を用いると、CREATE TABLE で NULL 入力可能を宣言した場合でも、すべてのカラム値を NOT NULL としたうえで UNIQUE インデックスが生成されます。これは PRIMARY KEY 制約でも同様で、この場合も NULL 値を許可しない UNIQUE インデックスが生成されます。

CREATE INDEX で UNIQUE キーワードを指定しないと、ユニークでないインデックスが作成され、インデックスのカラムで複数のローが同一値を持てるようになります。この種類のインデックスが使用される例としては外部キーがありますが、これは子テーブルにある複数のローが親テーブルのローにある共通の値を持つという状況が生じうるからです。ユニークでないインデックスの用途としては、ソートや検索を行う場合も挙げられます。

マルチカラム・インデックスの場合、インデックスの用途とカラムの順番が密接な関係にあります。たとえば次の例では、姓と名前を基にインデックスを作成したうえで「Robert」という名前のデータを検索していますが、実のところこのインデックスは検索の高速化になんら寄与しておらず、極論すると、紙媒体にアルファベット順に印刷された電話帳で検索する方が早いかもしれません。

```
CREATE TABLE phone_book (
   last_name      VARCHAR ( 100 ),
   first_name     VARCHAR ( 100 ),
   phone_number   VARCHAR ( 20 ) PRIMARY KEY );
CREATE INDEX book_sort ON phone_book ( last_name, first_name );
SELECT *
  FROM phone_book
 WHERE first_name = 'Robert';
```

ISQLの［ツール］－［オプション］－［プラン］タブで［長いプラン］を選択して［SQL］－［プランの取得］メニュー・オプションを選ぶか、［Shift + F5］キーを押すと、簡潔なテキストにまとめられた実行プランを確認できます。この機能を上記のクエリに対して用いるとISQLからは次のような結果が返されます。これを見ると、クエリの述部の実行にあたっては完全なテーブル・スキャンが行われ、book_sortインデックスは使用されない、ということがわかります。

```
( Plan [ Total Cost Estimate: 0 ]
  ( TableScan phone_book[ phone_book.first_name =  'Robert' : 5% Guess ] )
)
```

この場合のクエリ処理を高速化するには、次のようにfirst_nameを1番目としたインデックスを作成するか、またはfirst_nameを唯一のカラムに指定したインデックスを作成する必要があります。

```
CREATE INDEX first_name_sort ON phone_book ( first_name, last_name );
```

これでISQLレポートを確認すると、テーブル・スキャンの代わりにインデックス・スキャンが行われることがわかります。

```
( Plan [ Total Cost Estimate: 0 ]
  ( IndexScan phone_book first_name_sort )
)
```

インデックスに用いたカラムの値は、デフォルトでは昇順（ASC）にソートされたうえでインデックスに格納されます。SQL Anywhereでは、ORDER BY句でDESCが指定されたインデックス・カラムであっても、必要に応じて昇順インデックスが自動的に用いられるので、インデックスの定義時にASCとDESCのどちらを指定すべきかで特に悩む必要はありません。例外は、複数のカラムでのソートです。各カラムに異なるソート順を指定した場合、ORDER BY指定でのASCおよびDESCキーワードが一

致しているインデックスが優先的に使用されます。

次の例では、先の book_sort インデックスに含まれるカラムを ORDER BY 句に指定したものです。book_sort インデックスで指定したように両カラムともに ASC のソート順を指定するのではなく、それぞれのカラムに ASC と DESC という逆方向のキーワードを指定しています。

```
SELECT *
  FROM phone_book
 ORDER BY last_name ASC,
       first_name DESC;
```

ISQL で実行プランを確認すると、今回は完全なテーブル・スキャンに加えて、テンポラリ・ワーク・テーブルおよびソート処理が追加されていますが、これは book_sort インデックスが利用されていないためです。

```
( Plan [ Total Cost Estimate: .0377095 ]
  ( WorkTable
    ( Sort
      ( TableScan phone_book )
    )
  )
)
```

有効的に活用されるインデックスを作成するのであれば、次の book_sort2 のように、ORDER BY での ASC および DESC 指定に対応させたインデックス定義が必要です。

```
CREATE INDEX book_sort2 ON phone_book ( last_name, first_name DESC );
```

これで実行プランを確認すると、テーブル・スキャン、ワーク・テーブル、ソート処理が使用されなくなり、インデックス・スキャンだけが利用されるよう改善されたことがわかります。

```
( Plan [ Total Cost Estimate: .000645 ]
  ( IndexScan phone_book book_sort2 )
)
```

インデックスを CLUSTERED として定義すると、SQL Anywhere はこのインデックスのエントリと同じ物理順で、実際のローのデータを格納しようとします。この機能が特に有用なのは、「過去 10 年分のエントリを格納したテーブルから、本年 1 月第 1 週のアカウントに該当するエントリをすべて表示しろ」というような、インデックスのカラム値から限られた範囲内のデータを抽出するクエリの場合です。

テーブルは1つの順序でしかソートできないので、CLUSTERED 指定できるインデックスも 1 つのテーブルにつき1つだけです。新しいローが挿入されると、SQL Anywhere は、インデックス値が隣接するローどうしをできるだけ同じ物理ページに格納しようとします。ただし実際には、より多数のローが挿入されるに従い、インデックスの順序とローの実際の順序との間の食い違いが大きくなっていきます。さらに、

既に多数のローが存在するテーブルでクラスタード・インデックスを作成すると、ローが再配列されるのは、該当テーブルにREORGANIZE TABLE文が実行された後になります。REORGANIZE TABLEの詳細については、「10.6.3　テーブルの再編成」を参照してください。

Tip

通常、プライマリ・キーは、クラスタード・インデックスの候補とはなりません。たとえば、ASADEMOのsales_order_itemsテーブルのプライマリ・キーは、注文のIDとline_idを基に作成されています。このプライマリ・キーのインデックスは、1つのローをランダムに検索する場合には有効ですが、両方のカラムを対象とした範囲クエリには不向きです。逆に、ship_dateの値が特定の期間内にあるsales_order_itemsを抽出する、というよくあるタイプのクエリについては、ship_dateをクラスタード・インデックスに用いると高速化できます。

次の例は、先に10.3項で取り上げたインデックス・コンサルタントで生成したCREATE INDEX文を抜粋したものです。この例では、いずれのクラスタード・インデックスについても、その作成後にREORGANIZE TABLE文を実行して、ローの物理順をインデックスどおりに並べ替えている点に注目してください。

```
CREATE  INDEX "ixc_volume_test4_1" ON "DBA"."parent" ( non_key_5 );

CREATE CLUSTERED INDEX "ixc_volume_test4_2" ON "DBA"."parent" ( non_key_4 );
REORGANIZE TABLE "DBA"."parent";

CREATE  INDEX "ixc_volume_test4_3" ON "DBA"."child" ( key_1 ,non_key_5 );

CREATE  INDEX "ixc_volume_test4_4" ON "DBA"."child" ( non_key_5 );

CREATE CLUSTERED INDEX "ixc_volume_test4_5" ON "DBA"."child" ( non_key_4 );
REORGANIZE TABLE "DBA"."child";
```

1つのクエリ処理にあたってSQL Anywhereが使用するインデックスは、クエリ中の各テーブルで最大1個だけです。同じテーブルを対象にしても、クエリが異なれば使用されるインデックスも異なります。また、同一クエリで同じテーブルが異なるエイリアス名で2度使用されていると、これらは異なるテーブルとして扱われるので、用いられるインデックスも異なる場合があります。

個々のインデックスは、その維持にコストがかかります。これは、INSERTおよびDELETE文を実行するごとにインデックス・ページの変更が必要になるという意味であり、そうした事情は、インデックスのカラム値を変更するUPDATE文でも同様です。インデックスがクエリ処理に与える恩恵が大きければ、こうしたコストの存在は無視できますが、揮発性のテーブルを扱う場合には留意する必要があります。また、どのようなクエリ処理にも寄与しないインデックスがある場合、その維持に要するコストは無駄でしかありません。

特定のインデックスが有用であるかどうかは、インデックス・カラムのサイズ、インデックス・カラ

ムの順序、各インデックス・エントリに実際に格納されるインデックス・カラムのデータ量、作成されたインデックス・エントリの選択性など、複数の要因に左右されます。SQL Anywhere では、インデックス・カラムのすべてのデータがインデックス・エントリに必ずしも格納されるわけではありません。安易にインデックスを作成すると、どのようなクエリ処理の高速化にも寄与せず、維持コストがかかるだけの無用の長物と化す場合があります。

各インデックスに宣言されるデータ幅は、インデックスの各カラムで宣言された最大長に1を加えた値の総和になります。余分の1バイトは、各カラムのカラム長フィールドの分です。SQL Anywhere におけるインデックス・エントリの物理的な格納形式には、フル・インデックス、圧縮インデックス、部分インデックスという3つの種類があります。それぞれの形式の意味とその用途について、次に説明します。

- 「フル・インデックス」が作成されるのは、宣言されたデータ幅が10バイト以下の場合。フル・インデックスの場合、インデックス・カラムのすべての内容がインデックス・エントリに格納される。たとえば、1つのINTEGERカラムを基にインデックスを作成すると、宣言されるデータ幅は1 + 4 = 5バイトとなるので、5バイトのデータすべてがインデックス・エントリに格納される
- 「圧縮インデックス」が作成されるのは、宣言されたデータ幅が11から249バイトの間になる場合。圧縮インデックスの場合、インデックス・エントリのサイズを節約するため、インデックス・カラムのすべての内容が圧縮される。たとえば、VARCHAR (3) のカラムとVARCHAR (100) のカラムを基にしてインデックスを作成すると、宣言されるデータ幅は「1 + 3 + 1 + 100 = 105」バイトとなるので、カラムの内容には大幅な圧縮が加えられ、105バイトより小さな値としてインデックス・エントリに格納される。実際、圧縮インデックスはフル・インデックスより小さなサイズとなる場合がある
- 「部分インデックス」が作成されるのは、宣言されたデータ幅が250バイト以上の場合。部分インデックスの場合、カラムの内容は圧縮するのではなくトランケート（切り捨て）が行われ、最終的には宣言されたデータ幅の先頭10バイト分だけがインデックス・エントリに格納される。たとえば、1つのVARCHAR (249) のカラムを基にしてインデックスを作成すると、宣言されるデータ幅は「1 + 249」バイトとなるので、フィールド長の1バイトとカラムの内容の先頭9バイト分がインデックス・エントリに格納される

部分インデックスはフル・インデックスの一種で、インデックス・エントリで先頭10バイトの超過部がトランケートされる（切り捨てられる）という点以外は同じです。ただし、トランケートされる対象は、個々のカラム値ではなくインデックス・エントリ全体で評価されるので注意が必要です。たとえば、INTEGERのカラムとVARCHAR (300) のカラム各1つから作成されるインデックスの場合、宣言されるデータ幅は「1 + 4 + 1 + 300 = 306」となって圧縮インデックスの上限値249を超過するので、部分インデックスの10バイト長エントリが用いられます。このときINTEGERカラムの値はすべて格納されますが、VARCHAR (300) カラムの値は先頭4バイト分だけがエントリに収められます。

インデックス値のトランケーションは、該当インデックスが用いられるクエリ処理のパフォーマンス

に大きな影響を及ぼします。仮にインデックス情報の相違点がすべてトランケートされる部分に偏在していると、これらはインデックス・エントリに格納されず、インデックス・カラムの先頭部がすべて同じ値となるので、SQL Anywhere はテーブル・ローの値を取得して、インデックス・カラムが本来示すべき値が何であったかを確認し直す必要があります。インデックスのエントリ値だけでは解決できず該当ローのカラム値を調べ直す処理は「完全比較」と呼ばれますが、SQL Anywhere が何度こうした処理を強要されたかは、ISQL で次の SELECT 文を実行することで確認できます。

```
SELECT DB_PROPERTY ( 'FullCompare' );
```

この「DB_PROPERTY ('FullCompare')」の値が増加していくと、部分インデックスの副作用として、パフォーマンスが低下していきます。特定のクエリで完全比較が何回実施されたかは、「10.5　グラフィカルなプラン」で説明した［統計情報付きのグラフィカルなプラン］で確認できます。部分インデックスを用いて 1 つのローを特定するのに、10 回以上の完全比較が行われるのも珍しいことではありません。テーブル・ページがキャッシュされていない限り、これらの完全比較をするごとにディスクからのデータ読み込みが必要となる可能性が生じます。

また、次の項で説明する Windows パフォーマンス モニタを用いると、データベース全体で行われる完全比較の回数を追跡できます。

部分インデックスではインデックスの使用目的がまったく果たせない、ということではありません。これらのインデックス・エントリは、フル・インデックスのようにオリジナルの値をすべて保持するわけではありませんが、常にフル・インデックスに相当するカラム値でソートされた状態で格納されます。ただし、インデックス・カラムを用いた比較処理をする場合は、すべてのカラム値がインデックス・エントリに格納されている方が有利であり、部分インデックスよりもフル・インデックスおよび圧縮インデックスの方が高速に処理されます。

Section

10.8 データベースのパフォーマンス・カウンタ

SQL Anywhere の処理内容は、各種オペレーションの実行数やイベントの発生数をカウントすることで、随時追跡されています。これらのカウンタ値は、3つの組み込み関数（PROPERTY、DB_PROPERTY、CONNECTION_PROPERTY）および3つの組み込みプロシージャ（sa_eng_properties、sa_db_properties、sa_conn_properties）を通じて取得できます。

PROPERTY 関数は、指定されたプロパティ値をデータベース・サーバ・レベルで返します。DB_PROPERTY 関数も現在のデータベースのプロパティ値を返しますが、プロパティを取得するデータベース番号を指定すれば、同一サーバ上にある別のデータベースに関する情報も取得できます。CONNECTION_PROPERTY 関数も現在のデータベースのプロパティ値を返しますが、プロパティを

取得する接続番号を指定すれば、別の接続に関する情報も取得できます。パフォーマンス・カウンタの値を確認するには、これらの関数の戻り値を取得する必要があります。

次に示すのは、これら3つの関数の使用法です。PROPERTY の呼び出しでサーバのキャッシュ・サイズ（キロバイト）を、DB_PROPERTY の呼び出しで現在のデータベースで行われたディスク書き込みの回数を、CONNECTION_PROPERTY の呼び出しで現在の接続で行われたインデックスの完全比較の回数を取得しています。

```
SELECT PROPERTY ( 'CurrentCacheSize' )       AS server_cache_size_in_K,
       DB_PROPERTY ( 'DiskWrite' )           AS database_disk_writes,
       CONNECTION_PROPERTY ( 'FullCompare' ) AS connection_full_compares;
```

このクエリを実行すると、次のような結果が得られます。

```
server_cache_size_in_K  database_disk_writes  connection_full_compares
======================  ====================  =======================
130680                  26926                 10909818
```

3つの組み込みプロシージャは、取得したすべてのプロパティの名前と値を、複数ローの結果セットとして返します。sa_eng_properties プロシージャからはサーバ・レベルの 90 個のプロパティ値を、sa_db_properties プロシージャからは各データベースに関する 135 個のプロパティ値を、sa_conn_properties プロシージャからは各接続に関する 196 個のプロパティ値を取得できます【監注 18】。パフォーマンス・カウンタの値も、すべてこれらプロパティ値の一部として取得されます。次に、これら3つのプロシージャの呼び出し方の例を示します。

```
CALL sa_eng_properties();  -- すべてのサーバ・プロパティ
CALL sa_db_properties();   -- すべてのデータベースに関するデータベース・プロパティ
CALL sa_conn_properties(); -- すべての接続に関する接続プロパティ
```

次の例は、CREATE VIEW と SELECT を用いてサーバ・レベルおよびデータベース・レベルのパフォーマンス・カウンタ値のみを抽出して、1つのリストにまとめるものです。ここでは、数値のみを選択することでパフォーマンス・カウンタでないプロパティ値の大半を除外し、PROPERTY ('Name ') および DB_NAME (Number) 関数を用いてサーバとデータベースの名前を抽出しています。

```
CREATE VIEW v_show_counters AS
SELECT CAST ( STRING (
                 '1. Server ',
                 PROPERTY ( 'Name' ) )
              AS VARCHAR ( 200 ) )     AS property_type,
       PropName                        AS name,
```

【監注 18】9.0.2.3228 では、それぞれ、117 個、139 個、225 個になっている。

```
       Value                          AS value,
       PropDescription                AS description
  FROM sa_eng_properties()
 WHERE ISNUMERIC ( value ) = 1
UNION ALL
SELECT CAST ( STRING (
                 '2. DB ',
                 DB_NAME ( Number ) )
              AS VARCHAR ( 200 ) )    AS property_type,
       PropName                       AS name,
       Value                          AS value,
       PropDescription                AS description
  FROM sa_db_properties()
 WHERE ISNUMERIC ( value ) = 1
 ORDER BY 1, 2;

SELECT * FROM v_show_counters;
```

次のリストは、上のSELECTで返される結果セットの一部を抜粋したものです。このリストを見ると、キャッシュの読み込み数とキャッシュのヒット数がほぼ一致しているので、キャッシュの使用状況は良好であると確認できます。その一方で、インデックスの使用状況については、過剰な数の完全比較が行われているので、1つまたは複数のインデックスに設計上の問題があることが伺えます。

```
property_type     name              value     description
===============   ===============   ========  ================================
1. Server volume  CacheHitsEng      26845056  Cache Hits
1. Server volume  CacheReadEng      26845293  Cache reads
1. Server volume  CurrentCacheSize  130680    Current cache size in kilobytes
1. Server volume  DiskReadEng       470       Disk reads
2. DB volume      CacheHits         26842887  Cache Hits
2. DB volume      CacheRead         26843046  Cache reads
2. DB volume      DiskRead          378       Disk reads
2. DB volume      FullCompare       20061691  Number of comparisons beyond the
                                              hash value
2. DB volume      IndLookup         1584417   Number of index lookups
```

Windowsパフォーマンス モニタを用いると、個々のパフォーマンス・カウンタの値を継続的に追跡できます。たとえば、パフォーマンス モニタでインデックスの完全比較の発生数をグラフ表示させるには、次の手順に従います。

❶［スタート］−［プログラム］−［管理ツール］−［パフォーマンス］を選んでWindowsパフォーマンス モニタを起動します。

❷インデックスの完全比較数をモニタするには、右マウス・ボタンを押して[カウンタの追加]を選び、図10-19にある［カウンタの追加］ダイアログ・ボックスを表示させます。

❸［パフォーマンスオブジェクト］の一覧から［ASA 9データベース］を選びます。

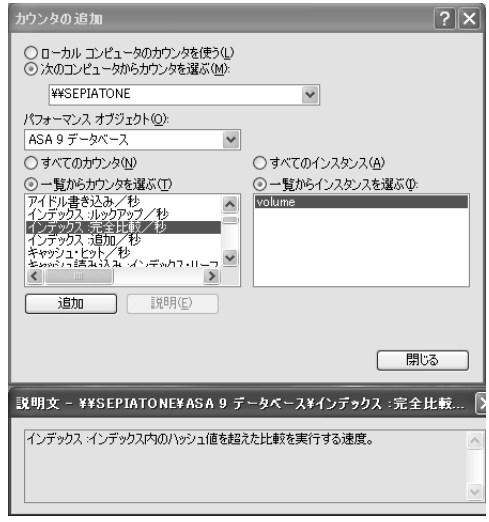


図 10-19　パフォーマンス モニタへのカウンタの追加

❹［一覧からカウンタを選ぶ］で［インデックス：完全比較／秒］を選びます。

❺［一覧からインスタンスを選ぶ］でモニタするデータベースを選びます。

❻［説明］ボタンを押して、選択中のカウンタの説明を確認します。

❼［追加］ボタンを押してから［閉じる］ボタンを押し、モニタのウィンドウに戻ります。

❽ グラフのプロパティを調整するには、右マウス・ボタンを押して［プロパティ］を選択し、［システムモニタのプロパティ］－［データ］タブを選んで、図 10-20 のウィンドウを表示させます。

図 10-20　パフォーマンス モニタの表示色とスケールの調整

❾ 各カウンタ・ラインの［カラー］および［幅］を選びます。

❿ 各カウンタ・ラインがグラフ・ウィンドウに収まるよう［スケール］を調整します。

⓫［グラフ］タブを選んで、各カウンタ・ラインがグラフ・ウィンドウに収まるよう［垂直スケール］－［最大］を調整します。

⓬［コンソール］－［名前を付けて保存］を選んで、パフォーマンス モニタの設定を Microsoft Management Console ファイル（*.msc）に保存します。設定情報を読み込むには、［名前を付けて保存］－［開く］を選びます。

パフォーマンス モニタには図 10-21 のような情報が表示されます。このグラフを見ると、1 秒間の完全比較数がピーク時には 200,000 を超えており、1 つまたは複数のインデックスの設計に重大な問題があることが伺えます。

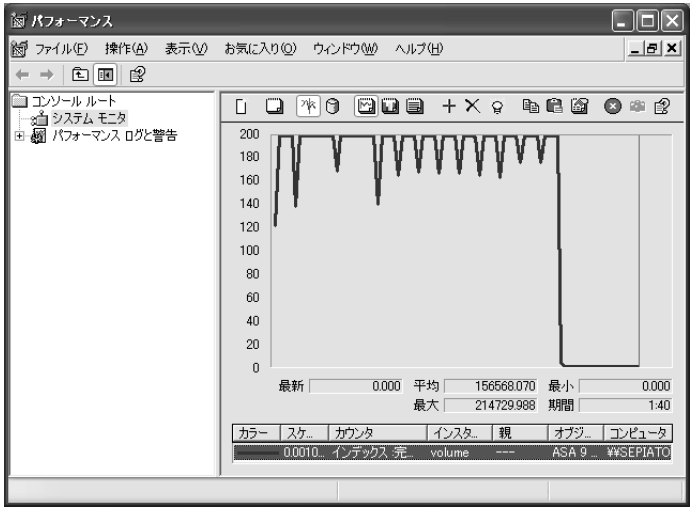


図 10-21　パフォーマンス モニタによる 1 秒間の完全比較数の表示

Section

10.9 ヒントとテクニック

パフォーマンスの向上に関しては、その他にも多数の方法が存在します。これらは、いずれも有用な手法であり、検討しておく価値も高いのですが、この章で個別の項を立てて解説するほど大げさなものでもありません。そこでこの項では、これまで言及しなかったこれらのヒントやテクニックをまとめて紹介します。なお、次のリストには参照用の番号が振られていますが、内容的には順不同です。

❶ COUNT(*) の代わりに EXISTS を使用する。
❷ UNION ALL を使用する。
❸ データベース設計を正規化する。
❹ 検索引数不可能な述部がないか確認する。
❺ シータ・ジョインがないか確認する。
❻ ユーザ定義関数の参照に注意する。
❼ OR の代わりに UNION が使えないか検討する。
❽ 更新をするときは随時 COMMIT を入れる。
❾ SET ROWCOUNT の指定をする。
❿ データベース・サーバの割り当てキャッシュ・メモリを増やす。
⓫ .log ファイルは必ず使用する。
⓬ RAID 1+0 の使用を検討する。
⓭ 異なる物理ディスク間でのファイル分散を検討する。
⓮ プライマリ・キーは必ず定義する。
⓯ 使用頻度の高いカラムを先頭に配置する。
⓰ 外部キー関係を明示的に指定する。
⓱ 一意性制約を明示的に指定する。
⓲ 不要なトリガのカスケード化に注意する。
⓳ 不要な CHECK 制約に注意する。
⓴ DEFAULT TIMESTAMP および DEFAULT LAST USER を利用する。
㉑ DEFAULT AUTOINCREMENT を利用する。
㉒ カラムに NOT NULL を指定する。
㉓ NOT TRANSACTIONAL 句を指定する。
㉔ MIN_TABLE_SIZE_FOR_HISTOGRAM を '100' に設定する。
㉕ CREATE STATISTICS を指定する。
㉖ DROP STATISTICS の使用を避ける。
㉗ テンポラリ・データには永続テーブルを使用しない。
㉘ オプティマイザを有効に活用する。
㉙ テーブル・ローを直接アプリケーションに渡さない。
㉚ トランザクション設計を把握する。
㉛ 接続と切断を繰り返さない。
㉜ カーソルを FOR READ ONLY として定義する。
㉝ ROW_COUNTS は 'ON' に設定しない。
㉞ ISOLATION_LEVEL はデフォルトの '0' とする。
㉟ 明示的な選択性推定の使用は避ける。
㊱ dbupgrad.exe は使用しない。

前ページのリストの具体的な内容は、次のとおりです。

❶ COUNT(*) の代わりに EXISTS が使用できないかを検討します。存在するローの数を知りたければ COUNT(*) を使う以外ありませんが、ロー・カウントが 0 であるかだけを確認したければ、EXISTS を用いることで、通常はより高速に処理されます。次の例は、SELECT と IF を併用することで、指定条件に対する該当ローの有無を 'Y' と 'N' で表示させるものです。

```
SELECT IF EXISTS ( SELECT *
                     FROM sales_order_items
                    WHERE prod_id = 401 )
       THEN 'Y'
       ELSE 'N'
       ENDIF;
```

❷ UNION ALL が使用できないかを検討します。通常の UNION 演算子だと、結合した結果セットをカラム別にソートしてから重複分の削除チェックを施す場合がありますが、UNION ALL だと余分な処理がすべて省略されます。重複が生じないことがあらかじめわかっていれば、UNION ALL を使うことで高速化できます。また多少の重複が生じる場合でも、これらの削除や除外をアプリケーション・プログラム側で行う方が速く処理できることがあります。

❸ ロー分割を削減するには、データベース設計を正規化します。一般に正規化を行うと、ローの長い少数のテーブルが、ローの短い多数のテーブルに分割されて、ローの短いテーブルのロー分割が減少します。正規化の詳細については「1.16　設計の正規化」を、ロー分割の詳細については「10.6.2　テーブルの断片化」を参照してください。

❹ クエリの実行速度が遅い場合は、検索引数不可能な述部がないかを確認します。「検索引数可能(sargable)」というのは聞き慣れない用語かもしれませんが、これはインデックスを有効活用した検索のできる述部のことを意味します（search argument-able の略)。つまり、検索引数可能であればいいのですが、問題は検索引数不可能の場合です。たとえば t1.key_1 がプライマリ・キーである場合、「t1.key_1 = 100」という述部は検索引数可能ですが、これはプライマリ・キーのインデックスに単独で一致するエントリがあるからです。これに対して「t1.key_1 <> 100」という述部は検索引数不可能であって、key_1 のインデックス情報で効率的に処理できる内容ではありません。同様に、「LIKE 'xx%'」はインデックスで効率的に処理できるので検索引数可能であり、「LIKE '%xx'」はインデックス情報を有効活用できないので検索引数不可能です。検索引数不可能な述部は、完全に削除できる場合や、クエリを書き換えて影響を最小限化できる場合があります。

❺ クエリの実行速度が遅い場合は、シータ・ジョインがないかを確認します。「シータ」とは、等号「=」以外の演算子のことです。たとえば、「child.key_1 <= parent.key_1」という述部は、外部キー関係を用いたシータ・ジョインの例です。シータ・ジョインには、マージ・ジョインやハッシュ・ジョインのアルゴリズムを実装できないので、パフォーマンスを悪化させる可能性があります。これらが問題の原因となっている場合は、シータ・ジョインをなくすようクエリを変更できないか検討します。

❻ WHERE、HAVING、FROM 句の述部内など、クエリ中でユーザ定義関数を参照する場合には注意が必要です。ユーザ定義関数の内部動作については、通常のクエリとは異なる最適化が必要とされる場合が多く、またこれらの関数が実際にどの程度の頻度で呼び出されるかは予測困難です。特にクエリやテンポラリ・テーブルを含む関数は要注意で、低速な関数が何百万回も呼び出されるとパフォーマンスが悪化します。

❼ OR の代わりに UNION が使えないかを検討します。SELECT 文で OR 条件を用いるよりも、条件別に 2 つの処理として分割し、得られた結果セットを UNION 演算子を用いて結合する方がよい場合があります。たとえば ASADEMO データベースの場合、次のクエリを実行させると、id と quantity カラムには個別のインデックスが定義されているのに、オプティマイザはテーブルの完全スキャンを行います。

```
SELECT *
  FROM sales_order_items
 WHERE id BETWEEN 3000 AND 3002
    OR quantity = 12;
```

これに対してクエリを次のように分割すると、インデックスを用いた処理が行われるようになり、UNION による結果も上記と同じものが得られます。

```
SELECT *
  FROM sales_order_items
 WHERE id BETWEEN 3000 AND 3002
UNION
SELECT *
  FROM sales_order_items
 WHERE quantity = 12;
```

❽ 長時間実行させるバッチの更新処理には、随時 COMMIT を入れておきます。多数のロックがかけられても他のユーザの処理を妨害しない場合がありますが、ロールバック・ログが巨大化すると、データベース・ファイルに余分なページが追加され、これに付随して無用なディスク I/O が生じます。これらのページは、最終的に COMMIT が実行された段階で削除されます。

❾ ロック数やロールバック・ログのサイズが肥大化しないうちに随時 COMMIT を実行する場合は、SET ROWCOUNT 1000 などを用いて、UPDATE や DELETE の処理対象となるロー数を制限しておきます。次の例では WHILE ループを用いて、未更新状態のローがなくなるまで UPDATE 文を繰り返し実行させています。この場合は 1000 ローごとに COMMIT が実行されますが、こうした制限は最後の SET ROWCOUNT 0 文で解除されます。

```
BEGIN
   DECLARE @updated_count INTEGER;
   SET ROWCOUNT 1000;
```

```
   UPDATE line_item
      SET supplier_id = 1099
    WHERE supplier_id = 99;
   SET @updated_count = @@ROWCOUNT;
   WHILE @updated_count > 0 LOOP
      COMMIT;
      MESSAGE 'COMMIT performed' TO CLIENT;
      UPDATE line_item
         SET supplier_id = 1099
       WHERE supplier_id = 99;
       SET @updated_count = @@ROWCOUNT;
   END LOOP;
   COMMIT;
   SET ROWCOUNT 0;
END;
```

⑩ データベース・サーバには、多量のキャッシュ・メモリを割り当てておきます。ディスクI/Oの所用時間を削減する最善の方法は、ディスクI/Oそのものを不要にすることで、サーバ・キャッシュはまさにそうした目的で用意されています。データベース・サーバには、多量のRAMを搭載した専用のコンピュータを割り当てて、すべてのメモリをサーバに割り当てます。

⑪ .logファイルは常に使用するようにします。トランザクション・ログを使用した場合でも、通常のCOMMITオペレーションが必要とするのはログ・ファイル末尾への連続的な書き込みだけです。データベース・ファイルの更新にランダムなディスクI/Oを行う分だけ処理に時間のかかるCHECKPOINTオペレーションについても、散発的に実施されるだけです。トランザクション・ログを使用しないと、COMMITごとにCHECKPOINTオペレーションが行われ、高負荷なサーバの場合はディスクI/Oが急増する可能性があります。トランザクション・ログの詳細については「9.11 ロギングとリカバリ」を参照してください。

⑫ RAIDを導入する場合はRAID 1+0(別名RAID 10)の採用を検討します。ハードウェアのパフォーマンスはここで解説できる範疇を超えていますが、一般にデータベース・パフォーマンスを向上させる最善の方法は、RAID 1+0の導入だと言われています。

⑬ RAIDを導入しないのであれば、トランザクション・ログやテンポラリ・ファイルなどを異なる物理ディスク間で分散させて、ディスクI/Oを高速化させることを検討します。ミラー・ログの配置先はトランザクション・ログとは異なる物理ドライブにすべきですが、そうでなければミラー・ログの作成そのものを取りやめます。ミラー・ログの作成はディスクI/Oの増加をもたらしますし、トランザクション・ログと同じ物理ドライブに配置しても、該当ドライブに障害が発生すれば両方のログは同時に失われるので、作る意味がありません。テンポラリ・ファイルの配置先は、ASTMP環境変数で制御できます。その他のファイルの配置先は、dbinit.exeとdblog.exeプログラム、CREATE DATABASE、ALTER DATABASE、CREATE DBSPACE、ALTER DBSPACE文で制御できます。

⑭ プライマリ・キーは、常に定義しておきます。プライマリ・キー・インデックスは、クエリを最適化するときにデータベース・エンジンが必ず使用するものです。プライマリ・キーが存在しないと、

各種のパフォーマンス向上に支障が生じ、ハード・シャットダウン後の自動リカバリ・プロセスも低速化します。

⓯ 使用頻度の高いカラムや短いカラムは、ローの先頭に配置しておきます。これはロー分割の影響を抑制します。詳細については「10.6.2　テーブルの断片化」を参照してください。また、使用頻度の高いカラムを検索するときに、同一ページ中の他のカラム・データをスキップする処理を省けるため、パフォーマンス向上に寄与します。

⓰ 外部キー関係を用いる箇所は、明示化しておきます。2つのテーブルが親子関係にある場合は、その関係を外部キー制約を用いて明示的に指定します。この結果得られるインデックスは、テーブル間ジョインの最適化に用いられることがあります。また、オプティマイザが外部キー関係を利用してジョインの結果セットのサイズを評価することで、実行プランの品質も向上します。

⓱ 一意性制約が必要な箇所は、明示化しておきます。特定のカラムのデータを一意にする場合は、カラムまたはインデックスにUNIQUE制約を明示的に指定します。データベース・エンジンは、こうして作成されるインデックスを用いてクエリの最適化を行います。

⓲ トリガをカスケード化すると、処理を遅くする場合があるので注意が必要です。重層化された複数のトリガ中にコードを埋め込むと、挿入、更新、削除の処理が遅くなります。

⓳ カラムやテーブルのCHECK制約は、処理を遅くする場合があるので注意が必要です。CHECK制約にサブクエリが関係していると、それらに関連するカラム値が変更されるごとにそのサブクエリが評価されます。

⓴ DEFAULT TIMESTAMPおよびDEFAULT LAST USERで済む処理を、トリガで代行しないようにします。これらは特殊なDEFAULT値ですが、トリガで同様の情報を取得すると、余分な時間がかかります。

㉑ DEFAULT AUTOINCREMENTで済む処理を、キー・プール・テーブルや独自の手法で代行しないようにします。これらの特殊なDEFAULT値は、必要な処理を高速かつ確実に実行でき、ロックおよびブロックの重複や競合も起こしません。

㉒ カラムには可能な限りNOT NULLを指定します。NULL入力可能なカラムがあると、データベース・エンジンによるクエリの最適化が難しくなるので、カラムはできるだけNOT NULLとしておきます。

㉓ テンポラリ・テーブルには可能な限りNOT TRANSACTIONAL句を指定します。1つのアトミック・オペレーションやトランザクション中でテンポラリ・テーブルの作成／使用／削除が行われる場合、これらのデータはロールバック・ログに書き込む必要がないので、NOT TRANSACTIONAL句を指定することでパフォーマンスが向上します。

㉔ MIN_TABLE_SIZE_FOR_HISTOGRAMデータベース・オプションの設定値を'100'に変更します【監注19】。これは、SQL Anywhereの管理する重要なクエリ最適化情報を、巨大なテーブルだけでなく、ロー数が100を超える程度のテーブルも対象にさせるための指定です。実際の情報は、SYSCOLSTATテーブルに保持されます。小さなテーブルであれば問題は生じないとは言い切れ

【監注19】この機能はdeprecatedになり、5以上のローがあるテーブルに対しては常に統計情報が管理される。

ないので、MIN_TABLE_SIZE_FOR_HISTOGRAM のデフォルト値 '1000' は大きすぎます。

㉕ 問題が生じているテーブルについては、CREATE STATISTICS 文を指定して、SQL Anywhere によるヒストグラム作成を強制します。作成されたヒストグラムは、SQL Anywhere によって更新され続け、その後の SELECT 処理時における最適な実行プランの選択に利用されます。ただし、こうしたヒストグラムは、処理されるロー数が少ない INSERT、UPDATE、DELETE 文を実行しても作成されない場合があります。CREATE STATISTICS および LOAD TABLE 文は強制的にヒストグラムを作成させるので、パフォーマンスが大きく改善される場合があります。

㉖ DROP STATISTICS 文は使用しないようにします。これを指定すると、クエリ・オプティマイザの機能が阻害され、有効に機能しなくなります。

㉗ テンポラリ・データには永続テーブルを使用しないようにします。永続テーブルに対する変更はトランザクション・ログに記録され、INSERT や DELETE 文を使用すると、書き込みが 2 度発生します。これに対してテンポラリ・テーブルのデータはトランザクション・ログに書き込まれないので、テンポラリ・データを扱うにはテンポラリ・テーブルの方が適しています。

㉘ オプティマイザ任せにした方がいい場合は、無理をしてテンポラリ・テーブルや独自のカーソル・ループを用意する必要はありません。クエリを構築するときは、できるだけ 1 つの SELECT 文ごとに 1 つのクエリにまとめます。たとえば、1 つの長大な SELECT 文にまとめるには手間がかかりすぎるクエリを扱う場合や、長大な SELECT 文を構築したが意図したとおりに動作しないという場合は、全体を複数の短いクエリに分割して、これらをオプティマイザに処理させる方が良好な結果が得られるケースがあります。そのような状況に遭遇したら無理をせず、作業処理を分割して個別に処理させるべきです。

㉙ テーブルのローを直接アプリケーションに渡すのは避け、ジョインやフィルタリング用のコードで処理させます。FROM と WHERE 句を用いて SQL Anywhere に処理させる方が高速です。

㉚ トランザクション設計を把握したうえで、クライアント側の「オートコミット」オプションを無効にし、データベースの CHAINED オプションをデフォルト値の 'ON' にしておき、アプリケーションが必要とする時点で COMMIT 文を明示的に実行させます。「オートコミット」オプションが有効にされているか、CHAINED が 'OFF' にされていると、COMMIT が過剰に実行されるようになり、パフォーマンスが低下します。詳細については「9.3　トランザクション」を参照してください。

㉛ データベースとの接続と切断を繰り返さないようにします。通常のアプリケーションが必要とする接続は 1 つだけ、または 2 つまでなので、不要になるまで接続は維持したままにしておきます。

㉜ カーソルはできるだけ FOR READ ONLY として定義し、可能であれば NO SCROLL またはデフォルトの DYNAMIC SCROLL に指定します。パフォーマンス的に最も優れているのは、読み込み専用の asensitive カーソルです。カーソル・タイプの詳細については「6.2.1　DECLARE CURSOR FOR select」を参照してください。

㉝ ROW_COUNTS データベース・オプションは 'ON' に設定しないようにします。これを指定すると、SQL Anywhere はロー数を計算するときと、結果セットを返すときの計 2 回クエリを実行することになります。

㉞ ISOLATION_LEVEL はできるだけデフォルトの '0' としておき、必要に応じて '1' とします。独立

性レベルを '2' や '3' の高い値にすると、マルチユーザ環境でのパフォーマンスが低下します。同時制御メカニズムとしては、多数のロックをかけてシステムを停滞させる「悲観的」なスキームではなく、「楽観的」なものを採用すべきです【監注 20】。独立性レベルの詳細については、「9.7　ブロックと独立性レベル」を参照してください。

㉟ 明示的な選択性推定によって特定のインデックスを使用するよう強制するのは避けてください。状況によっては、インデックスよりもテーブル・スキャンの方が高速に処理される場合があり、何よりも、指定したインデックスが最適なものであるとは限りません。選択性推定を使用するのであれば、これを用いることで実際にクエリ処理が高速化されることを事前に確認してからにします。

㊱ データベースのアップグレードに dbupgrad.exe は使用しないようにします。こうした処理には「10.6.6　アンロード／再ロードによるデータベースの再構築」で説明したアンロード／再ロードの手法が使えます。アップグレード・ユーティリティはシステム・カタログ・テーブルに論理的な変更を加えるだけで、データベース・ファイルの物理的な強化は行わないので、古いファイルの場合、アップグレードで改善された重要な機能やパフォーマンスの恩恵が受けられない可能性があります。現状のデータベースで使用できない機能を確認するには、Sybase Central の［データベースのプロパティ］ダイアログ・ボックスの［設定］タブを開いて、データベース機能のリストを表示させます。図 10-22 に表示されているデータベースは、SQL Anywhere 7 によって作成したものを dbupgrad.exe を用いてバージョン 9 にアップグレードしたものです。リスト中の×印は、重要な機能の一部が使えないことを示しており、そうした機能を使用可能にするには、10.6.6 項で説明したアンロード／再ロードを行う必要があります。

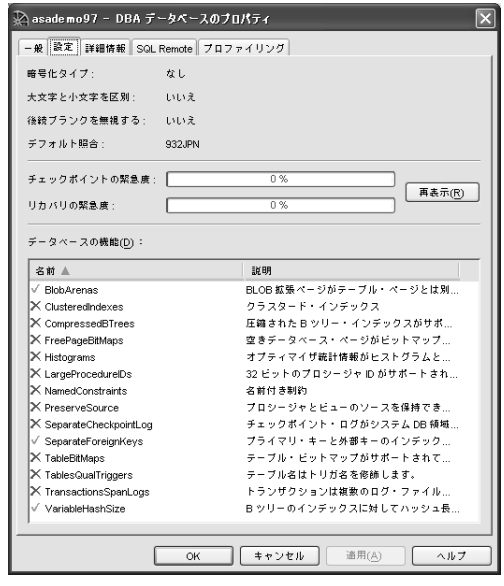


図 10-22　dbupgrad.exe での処理後に使用不可能と表示された機能

【監注 20】楽観ロックはアプリケーション側で実装する必要がある。たとえば、ローごとに更新回数を記録するようにして、コミット時にその回数を比べる。

Section

10.10 この章のまとめ

この章では、SQL Anywhere データベースのパフォーマンス改善にあたって検討すべき点および、具体的な各種の方法やアプローチを説明しました。ここでは、要求レベルのロギング、インデックス・コンサルタント、プロファイリング、グラフィカルなプランなど、SQL Anywhere に用意されている主要な機能を取り上げました。

ファイル、テーブル、インデックス・レベルにおける断片化については、断片化の度合いの測定法と対処法を含めて、それぞれ独立した項を立てて説明し、またアンロードと再ロードによるデータベース再編成については、個別の項を設けたうえで、安全かつ効率的な実行ステップを説明しました。3 種類の物理的なインデックスの実装法については、CREATE INDEX の項で詳細に説明してあり、組み込み機能を用いたデータベースのパフォーマンス測定については、Windows パフォーマンス モニタの使用法も含めてパフォーマンス・カウンタの項で説明しました。最後の項では、パフォーマンスの向上において重要な意味を持つ、各種の簡単なヒントおよびテクニックをまとめて説明しました。

本書での説明はここで終えますが、何か不明な点やコメントなどがあれば、Breck Carter まで問い合わせてください【監注 21】。

【監注 21】原書についての記述。本書へのご意見、ご質問、ご指摘などについては、ii ページに記載のとおり。

付属CD-ROMについて

収録内容

付属CD-ROMのDisc 1にはSQL Anywhere Studio 9.0.2 Windows版Developer Edition（32-bit/Windows CE）が、Disc 2には同Linux版Developer Edition（32-bit）が収録されています。また、Disc 1、Disc 2ともにHTMLディレクトリには、BNF_Syntax_from_SQL_Anywhere_Studio_9.htmlおよびCode_Examples_from_SQL_Anywhere_Studio_9.htmlの2つのHTMLファイルが収録されています。BNF_Syntax_from_SQL_Anywhere_Studio_9.htmlファイルには、本書に記載されているBNFが、Code_Examples_from_SQL_Anywhere_Studio_9.htmlにはコード例が含まれています。 どちらもHTML形式なので参照が容易です。また、章と項ごとに見やすく編成してあります。

Developer Editionについて

SQL Anywhere Studio 9.0.2 Developer Editionは、社内評価や開発を目的としたSQL Anywhereの利用に対して無償でライセンスするもので、有償の製品版と比べて機能制限はありません。詳細は「特定製品に関するライセンス条件」をお読みください。Developer Editionのインストール時にはライセンスキーが必要となります。アイエニウェア・ソリューションズ社のWebサイト（http://www.ianywhere.jp/）からユーザ登録（無償）を行うと、キーが発行されます。なお、キーを入力せずにインストールすると、Developer Editionではなく60日間有効の評価版となるのでご注意ください。

SQL Anywhereのインストール

SQL Anywhere（Windows版）のインストールを開始するには、setup.exeを実行して、画面の指示に従います。インストール中、iAnywhereのWebサイトに移動してインストール・キーを入手するかどうかをたずねるメッセージが表示されます。［はい］をクリックすると、アイエニウェア・ソリューションズ社のWebサイトがブラウザに表示されます。そのウィンドウで、インストール・キーを受け取るための登録を行うことができます。［いいえ］をクリックするとインストールが継続し、ライセンス契約の画面に移ります。

［登録キーまたはインストール・キー］のウィンドウが表示されたら、iAnywhereから入手したインストール・キーを入力するか、60日間の評価版を使用する場合は無効なキー（25桁のコードではないもの）を入力してください。 無効なキーを入力すると、有効なキーを入力するよう促すダイアログ・ボックスが表示されます。 無効なキーをもう一度入力して、［次へ］をクリックします。 再度ダイアログ・ボックスが表示され、ソフトウェアが60日間の評価版でインストールされることを伝えるメッセージが表示されます。また、インストールを続行するかどうかをたずねます。

Linux 版の場合は、CUI（Character User Interface）によるインストールとなります。setup スクリプトを実行して CUI インストーラを起動し、画面の指示に従います。

特定製品に関するライセンス条件

SQL Anywhere®Developer Edition

ご使用にあたって

「ソフトウェア・ライセンス契約」に規定されたライセンス条件に加え、以下に追加した権利と制限が、第 1 条に規定する「本プログラム」に適用されるものとします。

注意 : SQL Anywhere Developer Edition は無償で提供されます。ただし、本ライセンス条件への同意により、配布、再許諾その他いかなる方法であっても本プログラムを第三者に譲渡または使用許諾しないことに同意したものとみなされます 。

第 1 条（定義）使用される用語は、特に指定しない限り、アイエニウェア・ソリューションズ株式会社（以下「 当社 」といいます）のソフトウェア・ライセンス契約に定めるものと同義です。本ライセンス条件において「本プログラム」は、SQL Anywhere Studio およびそのすべてのコンポーネントを意味します。このバージョンの SQL Anywhere Studio は、以下のコンポーネントから構成されています。

- Adaptive Server®Anywhere DBMS（クライアント・バージョンおよびサーババージョン）
- Adaptive Server®Anywhere 同期付きパーソナル・データベース
- Adaptive Server®Anywhere DBMS Ultra Light エディション
- Mobile Link 同期サーバ
- Sybase Central ™
- jConnect®5.5
- Physical Architect ™（評価版）
- InfoMaker®（評価版）

注意 : 上記のコンポーネントは、ある特定のオペレーティング・システムについて提供されていない場合があります。たとえば、Ultra Light は Linux 版では提供しておりません。製品構成および対応プラットフォームについては、各製品のマニュアルをご覧ください。

本プログラムのメディア・パッケージには、複数の異なるオペレーティング・システムに対応する SQL Anywhere Studio のサーバおよびクライアント・コンポーネントの複数のバージョンが含まれることがありますが、お客様は SQL Anywhere Studio のライセンス許諾を受けた SQL Anywhere Studio のサーバおよびクライアントのコンポーネントのそれぞれのコピーを（1）つのオペレーティング・システムに

のみインストールすることができます。

第2条（各コンポーネントの個別使用）前述のとおり、SQL Anywhere Developer Edition は、いくつかのコンポーネントソフトウェアからなる製品です。以下において特に認める場合を除き、本プログラムのいずれのコンポーネントも本プログラムのライセンスを取得している一または複数の特定の開発用シート（たとえばPC、(1) ユーザのワークステーション、またはリアルタイム・デバイスなど）上での使用に限定されます。本プログラムのコンポーネントをそれぞれ別のまたは追加のシートで使用することはできません。

第3条（開発者によるプログラムの使用）このSQL Anywhere Developer Edition は社内評価、開発、および本プログラムと互換性のあるアプリケーションをテストすることのみを目的として、(1) コピーを単一のシートで使用することのみ許諾されています。通常の運用環境またはWebサーバ上での使用等これ以外のいかなる目的の場合でも、SQL Anywhere Developer Edition またはそのいずれのコンポーネントを配布または他の形で使用する場合は、事前に当社より配布ライセンスを取得する必要があります。本プログラムに含まれるクライアント・シートの (1) ライセンスでは、1 つのオペレーティング・システムを実行している (1) 台のマシンに Adaptive Server Anywhere DBMS をインストールし、(1) シートから Adaptive Server Anywhere サーバ・コンポーネントにアクセスすることを許諾しています。また接続された単一のクライアント・シートに以下のコンポーネントをインストールすることができます：(a) Adaptive Server Anywhere クライアント・ソフトウェア (b) jConnect、(c) Adaptive Server Anywhere 同期付パーソナル・データベース。上記のシートの同期は、Mobile Link または SQL Remote のどちらかを使用して実装することができます。また、Mobile Link 同期サーバ (1) コピーをインストールして SQL Anywhere データベースと同期を行う目的で使用することができます。Ultra Light データベースと Mobile Link クライアント・ソフトウェアのコピーをそれぞれ 1 つ 1 台のクライアント・デバイス上で使用することができます。Adaptive Server Anywhere および Mobile Link と連携して使用する場合にのみ、Sybase Central を (1) コピー使用できます。前述のプログラムは、Adaptive Server Anywhere サーバと Mobile Link コンポーネントがインストールされているサーバ・マシンまたは Adaptive Server Anywhere サーバへのアクセスがライセンスされているクライアント・シートのいずれかに限られます。

Java オプション： パッケージに含まれている Adaptive Server Anywhere のデータベース内 Java オプションは、個別ライセンスを必要に応じて別途購入しなければインストールまたは使用することはできません。

SQL Anywhere 配布プラットフォーム： ユーザ・ツールの使用: Linux など特定のコンピュータ・プラットフォームでは、SQL Anywhere の一部のコア・コンポーネントのみが使用可能です。特定のユーザ・ツールを使用できないプラットフォームでライセンスを取得済みの SQL Anywhere Studio の各コピーについては、Windows を実行する Intel（または互換性のある）マシン 1 台で、このようなユーザ・

ツール（例:Sybase Central）を実行することが許諾されています。この Intel マシンで Adaptive Server Anywhere のデータベース・サーバを実行することは許可されていません。

セキュリティ・オプション：本プログラムに含まれている Adaptive Server Anywhere 用 128 ビット Certicom 暗号化は、個別のライセンスを別途購入することなくインストールしたり使用することはできません。必要なライセンスを購入した場合であっても、Certicom 暗号化は SQL Anywhere と併用した場合に限り使用することができます。これらの暗号化プログラムやその関連マニュアルは暗号に関するものであるため、厳密な規制の対象となりますが、お客様が作成したアプリケーション、Certicom 暗号化プログラム、または関連ドキュメントに必要な輸出許可または輸入許可の取得は、お客様が行うものとします。

サンプル・コード：サンプル・コード（コード例、サンプル・アプリケーション、マニュアルに記載されたその他素材など）として提供された、このプログラムの一部を構成するソース・コードのバージョンについては、必要に応じて、開発環境内で使用したり、修正することができます。そのようなソース・コードのバージョンを第三者に配布することはできません。当社およびそのサプライヤの著作権または特許権のいずれにおいても、明示的なライセンス許可を与えていません。

評価版ソフトウェア: このプログラムの評価版（上記に示す）については、開発環境内で評価する場合（アプリケーションのテストや開発以外の目的を含むが、これだけに限定されない）および Adaptive Server Anywhere の使用と連携する場合にのみ、(1) 台のマシンで各コピーを (1) つ使用できます。前述のプログラムの使用は、Adaptive Server Anywhere サーバ・コンポーネントがインストールされているサーバ・マシンまたは Adaptive Server Anywhere サーバへのアクセスがライセンスされているクライアント・シートのいずれかに限定されます。これ以外のいかなる目的（アプリケーションのテストと開発、一般の運用環境または Web サーバでの使用を含むが、これだけに限定されない）の場合も、これらのプログラムのフルセットまたは一部を配布する前に、その目的のために配布ライセンスを当社から取得する必要があります。

第 4 条　本契約は、メンテナンスその他のサービスや、本プログラムのアップデートまたは新しいバージョンの供給を受ける権利をお客様に与えるものではありません。メンテナンスおよびサポートの提供は , 無償のものもあります。

第 5 条（期間）本ライセンスはその終了時まで有効です。お客様は、プログラムおよび関連ドキュメントを破棄することによって、いつでも本ライセンスを終了させることができます。また、本ライセンスは、お客様が本契約条項の何れかに違反する行為（不作為を含みます）があった場合にも終了します。この場合、お客様はプログラムおよびドキュメントを破棄することに同意していただきます。本プログラムのライセンス契約に記載されているすべての保証の放棄および責任制限は、本プログラムおよびそのコンポーネントに適用され、本契約の終了後も存続します。

第 6 条（追加製品）本プログラムには以下のバージョンのサード・パーティ製品が同梱されており（追加製品）、以降に示す諸条件がさらに適用されます。

Microsoft®Visual C++®、バージョン 6.0、Runtime ファイル ; Installer Components of Microsoft Data Access Components 2.7; Java Runtime Environment、Java 2 Standard Edition、J2SE ™バージョン 1.4.1; Microsoft HTML Help Support ファイル、Palm Conduit Manager DLL; Merant ODBC ドライバ ; Microsoft XML Parser 4.0。

6.1 本プログラムとの併用
追加製品は、本プログラムと同時に用いることのみを目的に使用できます。

6.2 追加製品の配布可能ファイル
一部の追加製品には、配布可能ファイルとして再配布してもよいファイルが含まれていますが SQL Anywhere Developer Edition との連携においてはその権利は許可しておりません。また、追加製品の所有者によってそのような権利が独自に認められる場合を除き、これらのファイルのいずれか、または追加製品を配布することはできません。

6.3 Java Platform Interface および Java Mark
上記の規定の他、お客様は、Java Platform Interface（ JPI：“java” パッケージまたは “java” パッケージのサブパッケージに含まれているクラスとして識別）内でクラスを追加作成したり、JPI のクラスを結果的に追加または変更したりすることはできません。お客様は (i) Java 環境の機能を拡張する API を作成し、(ii) そのような追加 API を実行する追加のソフトウェアを開発する目的でその API を第三者のソフトウェア・ベンダーに提供する場合には、全ての開発者がそのような API を無償で利用できるよう、正確な仕様を、速やかに、適切な形で広く公開しなければなりません。お客様はいかなる “java” クラスにも名前を付けることはできず、また Sun が合理的に要求する命名規則に従う必要があります。本ライセンスでは、Sun の名称、商標、ロゴ（をお客様が使用することをいっさい許諾しておらず、Java の商標および Java 関連のすべての商標、ロゴ、Coffee Cup および Duke（ Java Mark）などのアイコンが Sun の所有物であることを了解するとともに、(i) Java Trademark Guidelines（http://java.sun.com/trademarks.html）を遵守し、(ii) Java Mark に関して Sun が所有する権利を侵害または相反する行為をいっさい行わず、(iii) 任意の Java Mark についてお客様が取得したいっさいの権利を Sun へ譲渡することを含め、Sun のこれらの権利を保護するよう協力することに同意することになります。

6.4 追加製品のサポート
当社では、同梱したバージョンの追加製品を当社による保証は追加せずに現状のままお客様に提供しています。これらの追加製品の保証は、それらのサプライヤから提供されることもあります。当社では同梱した追加製品の本プログラムへの統合はサポートしますが、追加製品そのものに対するサポートや不

具合の修正、または新規リリースへの対応を行う義務はいっさいありません。このようなサービスや製品の提供は、追加製品の各サプライヤから受けられることもあります。

6.5 本プログラムおよび追加製品に対して適用される当社のソフトウェア・ライセンス契約に定めるその他の免責条項および損害の限定に加え、すべての追加製品に対して次の規定が適用されます。いかなる場合も、当社およびその関連会社は、追加製品起因あるいは関連して発生した直接の損害に関して一切責任を負いません。

第7条（その他）本プログラムを日本以外の国でダウンロードおよびインストールする場合、当社に変わりSybase Inc. およびその子会社（以下「サイベース」といいます）から使用許諾される場合があります。この場合、本契約書における「当社」は「サイベース」に読み替えられるものとします。

サンプル・コードの使用に関するライセンス条件

付属CD-ROMを開封してBNF_Syntax_from_SQL_Anywhere_Studio_9.htmlおよびCode_Examples_from_SQL_Anywhere_Studio_9.htmlをご使用になる前に、以下のサンプル・コードの使用に関するライセンス条件をお読みください。

1 CD-ROMのパッケージを開くと、この「サンプル・コードの使用に関するライセンス条件」の各条項を読み、これらに従うことに同意したものとみなされます。
2 CD-ROMと本書に収録されているサンプル・コードは、どちらも米著作権法と国際著作権条約によって著作権が保護されています。また、これらの所有権は原書出版社であるWordware Publishing, Inc. に帰属します。サンプル・コードの著作権は、作者が所有します。
3 CD-ROMまたは本書のすべてのサンプル・コードのいかなる部分も、Wordware Publishing, Inc. またはサンプル・コードの作者の書面による許諾なしに、一般のフォーラム（WWWページ、FTPサイト、掲示板、インターネット・ニュース・グループなど）に公開することは許可されません。
4 収録されたサンプル・コードは、この契約に記載された方法を除き、デコンパイル、リバース・エンジニアリング、ディスアセンブル、派生著作物の作成、またはその他の方法で使用することは許可されません。
5 CD-ROMおよび本書に収録されているサンプル・コードについて、販売元はいかなる保障も行いません。Wordware Publishing, Inc. と作者は、CD-ROMおよびこれに含まれるサンプル・コードあるいは本書に記載された技術やサンプル・プログラムに実装された技術における欠陥についての市販性、および特定目的との適合性の暗示的保障を含む（ただし必ずしもこれらに限定されない）明示または暗示のその他すべての保障を明確に拒否するものとします。いかなる場合でも、特殊または偶発的、間接的、その他の損害を含む（ただし必ずしもこれらに限定されない）、利益の損失またはその他の私的損害、業務上の損害、およびそれらの主張に対して、Wordware

Publishing, Inc. とその販売店、販売元、および作者は、賠償の責めを負わないものとします。

6 CD-ROM に収録されたサンプル・コードは、バックアップの目的で一度だけコピーを作成することができるものとします。すべてのサンプル・コードは、ハードディスクへコピーすることができるものとします。ただし、同時に 2 台以上のコンピュータに存在させることは許可されません。2 台目のコンピュータにコピーできるのは、1 台目のコンピュータにコピーした内容を削除した場合に限られるものとします。

7 サンプル・コードのいかなる部分も、Wordware Publishing, Inc. またはサンプル・コードの所有者の書面による許諾なしに、CD-ROM、ディスク、インターネット・ディストリビューションなどを含む（ただし必ずしもこれらに限定されない）媒体上のサンプル・コード、ユーティリティ、ヘルプ・ファイル、サンプル・プログラム、フリーウェア、シェアウェア、評価版プログラムのコンパイルに含めることは許可されません。

索引

記号

!< ····· 150
!= ····· 150
!> ····· 150
" ····· 3, 9
····· 9
#hook_dict ····· 324
% ····· 157
%ASANY9% ····· 287
& ····· 134
( ) ····· 3, 118, 134, 135
* ····· 132, 134
+ ····· 134
- ····· 134, 157
-- ····· 3
.dll ライブラリ ····· 354
.nlm ライブラリ ····· 354
/ ····· 134
::= ····· 3
< ····· 150
<= ····· 150
<> ····· 3, 150
= ····· 150
> ····· 150
>= ····· 150
? ····· 297
@ ····· 271
@@ERROR ····· 425
@@ROWCOUNT ····· 451
@@TRANCOUNT ····· 410
[^] ····· 157
[ ] ····· 3, 9, 157
^ ····· 134
_ ····· 157
{ } ····· 3
| ····· 3, 134
|| ····· 134, 174
~ ····· 134
¥ ····· 16, 77
¥¥ ····· 17
¥n ····· 17

数字

1NF ····· 54
2NF ····· 55
3NF ····· 56
4NF ····· 58
5NF ····· 59

A

ACID ····· 403
after delete トリガ ····· 235
after row トリガ ····· 372
after statement トリガ ····· 217, 235, 373
after update トリガ ····· 217
ALL ····· 150, 151, 164
ALTER DBSPACE ····· 540, 557
ALTER EVENT ····· 380
ALTER TABLE ····· 65
AND ····· 148
ANSI_INTEGER_OVERFLOW ····· 400
ANSI_PERMISSIONS ····· 397
ANY ····· 150, 151
ASTMP ····· 574
ATTRIB ····· 497, 498
AUDITING ····· 397
AUDITING_OPTIONS ····· 397
authenticate_parameters ····· 285, 296
authenticate_user ····· 284, 296
authenticate_user_hashed ····· 296
AVG( ) ····· 163, 164

B

BACKGROUND_PRIORITY ····· 397, 548
BACKUP DATABASE ····· 387, 488
DBFILE ONLY ····· 488, 490
DIRECTORY ····· 488, 490
TRANSACTION LOG ONLY ····· 488, 490
TRANSACTION LOG RENAME ····· 488, 490
TRANSACTION LOG RENAME MATCH ····· 488, 490
TRANSACTION LOG TRUNCATE ····· 488, 490
WAIT AFTER END ····· 488, 490
WAIT BEFORE START ····· 488, 490
WITH COMMENT ····· 489, 491
BCNF ····· 57
before delete トリガ ····· 235
before row トリガ ····· 368
before update トリガ ····· 217
BEGIN ····· 338
ATOMIC ····· 339
NOT ATOMIC ····· 339
例外ハンドラ ····· 341
ローカル変数 ····· 339
begin_connection ····· 295
begin_connection_autocommit ····· 295
begin_download ····· 296, 299
begin_download_deletes ····· 299
begin_download_rows ····· 299
begin_publication ····· 296, 320, 321, 331
begin_synchronization ····· 296, 297, 299, 321
begin_upload ····· 296, 299
begin_upload_deletes ····· 299
begin_upload_rows ····· 299
BEGIN TRAN ····· 405
BEGIN TRANSACTION ····· 405
BETWEEN ····· 155
blob アリーナ ····· 543
blob 継続 ····· 11, 543
BLOCKING ····· 401, 437, 455
BLOCKING_TIMEOUT ····· 457
BNF 表記法 ····· 2

C

CALL ····· 354, 355
CASE ····· 137, 347
CAST( ) ····· 140, 141
CHAINED ····· 406, 431
CHECK ON COMMIT ····· 38, 39

CHECKPOINT_TIME ····· 397
CLOSE ····· 267
CLOSE_ON_ENDTRANS ····· 400
CLUSTERED ····· 32
COALESCE( ) ····· 141, 142
COMMIT ····· 405
COMMIT TRAN ····· 405
COMMIT TRANSACTION ····· 405
COMMIT WORK ····· 405
CONNECTION_PROPERTY( ) ····· 383, 464, 566
CONTINUE_AFTER_RAISERROR ···· 426
COPY ····· 497, 498
COUNT(*) ····· 163, 165, 174, 572
COVAR_POP ····· 168
COVAR_SAMP ····· 168
CREATE DBSPACE ····· 9
CREATE EVENT ····· 378
BETWEEN ····· 383, 384
DISABLE ····· 380
EVERY ····· 384
HANDLER ····· 378
ON ····· 384
SCHEDULE ····· 383
START DATE ····· 384
START TIME ····· 383, 384
TYPE ····· 378
WHERE ····· 379
CREATE EXISTING TABLE ····· 44
CREATE EXTERNLOGIN ····· 42
CREATE FUNCTION ····· 140, 362
DETERMINISTIC ····· 362
NOT DETERMINISTIC ····· 362
RETURNS ····· 362
CREATE INDEX ····· 559
CREATE MESSAGE ····· 423
CREATE PROCEDURE ····· 126, 354
DEFAULT ····· 354, 356
IN ····· 354, 356
INOUT ····· 354, 356
OUT ····· 354, 356
RESULT ····· 128, 354
CREATE PUBLICATION ····· 281, 282
CREATE SERVER ····· 41, 42
CREATE SYNCHRONIZATION SUBSCRIPTION ····· 281, 285, 293
CREATE SYNCHRONIZATION USER ····· 281, 283
CREATE TABLE
····· 49
AT ····· 42
DEFAULT ····· 68
CREATE TRIGGER ····· 367
AFTER ····· 372
BEFORE ····· 367, 368
DELETE ····· 367, 368
DELETING ····· 370
FOR EACH ROW ····· 367, 368, 369, 372
FOR EACH STATEMENT ····· 372
INSERT ····· 367, 368
INSERTING ····· 370
NEW AS ····· 367, 368
OLD AS ····· 367, 368
ON ····· 367, 368
ORDER ····· 367, 368
REFERENCING ····· 367, 368
UPDATE ····· 367, 368
UPDATE OF ····· 367, 368
UPDATING ····· 370
WHEN ····· 367, 368
CREATE VARIABLE ····· 392
CREATE VIEW ····· 190
WITH ····· 194
WITH CHECK OPTION ····· 190, 192
CROSS JOIN ····· 108
CURRENT DATE ····· 135
CURRENT TIME ····· 13
CURRENT TIMESTAMP ····· 13
CURRENT USER ····· 383, 465

D

DATABASE_AUTHENTICATION ····· 397
DATE ····· 504
DB_PROPERTY( ) ····· 539, 541, 566
dbbackup.exe ····· 475, 488, 494
dbeng9.exe ····· 458, 554
DBFreePercent ····· 388
DBFreeSpace ····· 388
dbinit.exe ····· 556
dbisql.exe ····· 84
dblog.exe ····· 503
dbmlsrv9.exe ····· 293
設定 ····· 294
dbmlsync.exe ····· 280, 286, 475
オプション ····· 288
dbmlsync_sql_error ····· 325
dbo ····· 466
dbspace ····· 480
dbsrv9.exe ····· 458, 554
-a オプション ····· 498
dbtran.exe ····· 82, 406
-a オプション ····· 499
dbunload.exe ····· 555
dbupgrad.exe ····· 577
dbvalid.exe ····· 475, 503
DECLARE CURSOR FOR ····· 258
DECLARE CURSOR FOR CALL ····· 265
DECLARE CURSOR USING ····· 264
DECLARE LOCAL TEMPORARY TABLE ····· 50, 51
DEFAULT GLOBAL AUTOINCREMENT ····· 310, 313
DEFAULT 値 ····· 21
DEFAULT AUTOINCREMENT ····· 22
DEFAULT GLOBAL AUTOINCREMENT 23
DEFAULT LAST USER ····· 26
DEFAULT TIMESTAMP ····· 26
DEFAULT UTC TIMESTAMP ····· 26
特殊更新デフォルト ····· 26
内部システム・デフォルト値 ····· 398
パブリック・デフォルト値 ····· 398
ユーザ・デフォルト値 ····· 398
リテラル・デフォルト ····· 24
DELAYED_COMMIT ····· 481
DELETE ····· 32
WHERE CURRENT OF ····· 244
カーソル指向型 ····· 244
集合指向型 ····· 237
DELETING ····· 159
DISTINCT ····· 164
download_cursor ····· 300, 313
last_download パラメータ ····· 314
download_delete_cursor ····· 300, 318, 321
download_statistics ····· 296, 299

E

ECHO ····· 504
ECHO OFF ····· 504
Embedded SQL ····· 256
end_connection ····· 295
end_download ····· 296, 299

end_download_deletes ····· 299
end_download_rows ····· 299
end_publication ····· 296
end_synchronization ····· 296, 299
end_upload ····· 296, 299
end_upload_deletes ····· 299
end_upload_rows ····· 299
ERRORMSG ····· 342
ESCAPE ····· 158
EVENT_CONDITION( ) ····· 379
EVENT_PARAMETER( ) ····· 380
EXCEPT ····· 184
EXCEPTION ····· 341
EXCEPTION FOR SQLSTATE ····· 341
EXECUTE IMMEDIATE ····· 350
EXISTS ····· 153, 572
EXPRTYPE( ) ····· 142

F

FALSE ····· 137, 147, 149
FASTFIRSTROW ····· 107
FETCH ····· 254, 268
ABSOLUTE ····· 269
FIRST ····· 269
LAST ····· 269
NEXT ····· 268
PRIOR ····· 269
RELATIVE ····· 269
filler( ) ····· 75
FIRST ····· 179
FLOAT_AS_DOUBLE ····· 15, 400
FULL OUTER JOIN ····· 108, 113

G

GLOBAL_DATABASE_ID ····· 310, 397
GRANT ····· 467
ALL ····· 467, 470
ALTER ····· 467, 468
DBA ····· 474
DELETE ····· 467, 468, 470, 471
EXECUTE ····· 472
GROUP ····· 476
INDEX ····· 467
INSERT ····· 467, 468, 470, 471
MEMBERSHIP IN GROUP ····· 476
REFERENCES ····· 467, 468
REMOTE DBA ····· 474
RESOURCE ····· 474
SELECT ····· 467, 468, 470, 471
UPDATE ····· 467, 468, 470, 471
WITH GRANT OPTION ····· 467, 469, 470
GRANT CONNECT ····· 462
IDENTIFIED BY ····· 463
GRAPHICAL_PLAN ····· 536
GROUP BY ····· 99, 100, 160
GROUP BY ROLLUP ··99, 100, 160, 171
GROUPING ····· 163

H

handle_error ····· 296, 311
handle_odbc_error ····· 296
HAVING ····· 175
HOLDLOCK ····· 107

I

iAnywhere JDBC ドライバ ····· 421
IF ····· 137, 149, 346, 504
IF EXIST ····· 487
IN ····· 154
INNER JOIN ····· 108, 109
INPUT ····· 84
BY NAME ····· 84, 87
BY ORDER ····· 84, 87
COLUMN WIDTHS ····· 84, 87
DELIMITED BY ····· 84, 88
END ····· 85
ESCAPE CHARACTER ····· 84, 88
FORMAT ····· 84
NOSTRIP ····· 84, 88
PROMPT ····· 84
INSERT ····· 64
DEFAULT ····· 65, 66
ON EXISTING ERROR ····· 65, 67
ON EXISTING SKIP ····· 65, 67, 313
ON EXISTING UPDATE ···· 65, 67, 313
select を使用した〜 ····· 69
VALUES ····· 65
WITH AUTO NAME ····· 64, 72
指定のカラムへの〜 ····· 67
INSERTING ····· 159
INTEGRATED_SEVER_NAME ····· 397
Interactive SQL ····· 84
INTERSECT ····· 184
INTO ····· 183
IS ····· 148
ISOLATION_LEVEL ····· 107, 437, 454
ISQL ····· 84
IS NOT ····· 148
IS NULL ····· 156

J

JAVA_NAMESPACE_SIZE ····· 397
JAVA_PAGE_BUFFER_SIZE ····· 397
JOIN ····· 108

L

LATERAL ····· 130
LEAVE ····· 338
LEFT( ) ····· 141, 144
LEFT OUTER JOIN ····· 108, 110
LENGTH( ) ····· 141, 145
LIKE ····· 157
LIST( ) ····· 163, 168
LOAD TABLE ····· 74
BCP フォーマット ····· 78
CHECK CONSTRAINTS ····· 74
CHECK CONSTRAINTS OFF ····· 77
COMPUTES ····· 74
COMPUTES OFF ····· 77
DEFAULTS ····· 74
DEFAULTS ON ····· 77
DELIMITED BY ····· 74, 77
ESCAPE CHARACTER ····· 74, 77
ESCAPES ····· 74, 78
filler( ) ····· 75
FORMAT ( ) ····· 74
HEXADECIMAL ····· 74, 78
ORDER ····· 74, 78
PCTFREE ····· 74, 78
QUOTES ····· 74, 78
STRIP ····· 74, 78
WITH CHECKPOINT ····· 74, 78
LOCATE( ) ····· 141, 145
LOCK TABLE ····· 437, 454
LOG_DEADLOCKS ····· 397

LOGIN_MODE ····· 397
LOWER( ) ····· 141, 145
LTRIM( ) ····· 141, 145

M

MAX_HASH_SIZE ····· 397
MAX_WORK_TABLE_HASH_SIZE ···· 397
MAX( ) ····· 163, 166
MESSAGE ····· 130, 132
MIN_PASSWORD_LENGTH ····· 397
MIN_TABLE_SIZE_FOR_HISTOGRAM ····· 397, 400, 575
MIN( ) ····· 163, 166
ml_add_connection_script ····· 295, 303
ml_add_table_script ····· 299, 303
ml_subscription ····· 320, 328
Mobile Link ····· 274, 275
Mobile Link スクリプト ····· 279
Mobile Link 特有のエラー ····· 310
Mobile Link モニタ ····· 333
Mobile Link ユーザ名 ····· 283
upsert ····· 313
アーティクル ····· 282
アップサート ····· 313
エラーの処理 ····· 310, 322
競合の処理 ····· 304
強制的競合 ····· 309
クライアント ····· 280, 281
サーバ ····· 293
サブスクリプション ····· 285
システム・テーブル ····· 328
シャドー・テーブル ····· 319
スクリプトのテスト ····· 298
セッション・レベル・イベント ····· 279
セッション・レベルのスクリプト ···· 296
接続レベル・イベント ····· 280
接続レベルのスクリプト ····· 295
タイムスタンプ・ダウンロード ····· 314
テーブル・レベル・イベント ····· 279
テーブル・レベルのスクリプト ····· 298
パブリケーション ····· 282
フック・プロシージャ ····· 323
ロー・レベル・イベント ····· 279
MOD( ) ····· 256
modify_last_download_timestamp ·· 297
modify_next_last_download_timestamp ····· 297
modify_user ····· 297

N

NOLOCK ····· 107, 454
NONCLUSTERED ····· 32
NOT ····· 148
NOT EXISTS ····· 154
NOT LIKE ····· 157
NOT NULL ····· 38
NOT TRANSACTIONAL ····· 48
NOW(*) ····· 140, 141
NULL ····· 4, 118, 134, 156, 161, 174
NULL プロパティ ····· 28
NUMBER(*) ····· 94, 104, 140, 181

O

ON_ERROR ····· 424, 432
ON COMMIT PRESERVE ROWS ····· 47
ON DELETE ····· 32, 38
ON DELETE CASCADE ····· 32
ON DELETE RESTRICT ····· 32
ON DELETE SET DEFAULT ····· 32
ON DELETE SET NULL ····· 32
ON UPDATE ····· 32, 38
ON UPDATE CASCADE ····· 32
ON UPDATE RESTRICT ····· 32
ON UPDATE SET DEFAULT ····· 32
ON UPDATE SET NULL ····· 32
OPEN ····· 267
OPTIMIZATION_GOAL ····· 107, 558
OPTIMIZATION_WORKLOAD ····· 397
OR ····· 148
ORDER BY ····· 93, 176
OTHERS ····· 342
OUTPUT ····· 208
APPEND ····· 208, 211
COLUMN WIDTHS ····· 208, 211
DELIMITED BY ····· 208, 211
ESCAPE CHARACTER ····· 208, 211
FORMAT ····· 208, 211
HEXADECIMAL ····· 208, 212
QUOTE ····· 208, 212
VERBOSE ····· 208, 212

P

PCTFREE ····· 542
PI ····· 140
PINNED_CURSOR_PERCENT_OF_CACHE ····· 397
PowerBuilder ····· 409
prepare_for_download ····· 297
PRESERVE_SOURCE_FORMAT ····· 397
PROPERTY( ) ····· 566

R

RAISERROR ····· 422
READCOMMITTED ····· 107, 454
READUNCOMMITTED ····· 107
RECOVERY_TIME ····· 398
REFERENCES ····· 38
REGR_AVRX ····· 168
REGR_AVRY ····· 168
REGR_COUNT ····· 168
REGR_INTERCEPT ····· 168
REGR_R2 ····· 168
REGR_SLOPE ····· 168
REGR_SXX ····· 168
REGR_SXY ····· 168
REGR_SYY ····· 168
RELEASE SAVEPOINT ····· 413
RENAME ····· 487, 497, 498
REORGANIZE TABLE ····· 387, 547, 552
REPEAT( ) ····· 141, 145
REPEATABLEREAD ····· 107
REPLACE( ) ····· 145
REPLICATE( ) ····· 141
Replication Server ····· 274
report_error ····· 297, 311
report_odbc_error ····· 297, 312
RESIGNAL ····· 418
resolve_conflict ····· 300, 304, 306
RESULT ····· 358
RETURN ····· 354, 355
REVOKE CONNECT ····· 466
REWRITE( ) ····· 152
RI_TRIGGER_TIME ····· 398
RIGHT( ) ····· 141, 144
RIGHT OUTER JOIN ····· 108, 112
ROLLBACK ····· 405
ROLLBACK TO SAVEPOINT ···· 339, 413

ROLLBACK TRAN ····· 405
ROLLBACK TRANSACTION ····· 405
ROLLBACK TRIGGER ····· 430
ROLLBACK WORK ····· 405
ROLLUP ····· 171
ROW_COUNTS ····· 576
ROWCOUNT ····· 573
RowGenerator ····· 363
RTRIM ( ) ····· 141, 145

S

sa_conn_info ····· 458, 464, 522
sa_conn_properties ····· 566
sa_db_properties ····· 566
sa_disk_free_space ····· 540
sa_eng_properties ····· 566
sa_get_request_profile ····· 517
sa_get_request_times ····· 461, 515
sa_index_density ····· 550, 560
sa_index_levels ····· 549, 560
sa_locks ····· 434
sa_server_option ····· 513
Requests_for_connection ····· 522
Requests_for_database ····· 524
sa_statement_text ····· 518
sa_table_fragmentation ····· 543
sa_table_stats ····· 544, 546
satmp_request_profile ····· 517
satmp_request_time ····· 461, 515
SAVEPOINT ····· 413
SELECT ····· 92
ALL ····· 103
AS ····· 106, 133
CROSS JOIN ····· 106, 108
DISTINCT ····· 93, 94, 103, 178
EXCEPT ····· 93
FIRST ····· 179
FOR READ ONLY ····· 93
FOR UPDATE ····· 93
FOR UPDATE BY ····· 93
FOR XML ····· 93, 104
FROM ····· 94, 96, 105
FULL JOIN ····· 106
FULL OUTER JOIN ····· 106, 108, 113
GROUPING ····· 94, 101
GROUP BY ····· 94, 99, 100, 160
GROUP BY ROLLUP ·· 94, 99, 100, 160
HAVING ····· 94, 100, 101, 175
INNER JOIN ····· 106, 108, 109
INTERSECT ····· 93
INTO ····· 94, 183
JOIN ····· 106
KEY ····· 106
LATERAL ····· 130
LEFT JOIN ····· 106
LEFT OUTER JOIN ····· 106, 108, 110
NATURAL ····· 106
ON ····· 106, 109, 114
ORDER BY ····· 93, 94, 102, 176
RIGHT JOIN ····· 106
RIGHT OUTER JOIN ···· 106, 108, 112
select リスト ····· 132
START AT ····· 103, 179
TOP ····· 179
UNION ····· 93
WHERE ····· 98, 147
WITH ····· 93, 107, 128, 194
プロシージャ・コール ····· 125
SELECT INTO # ····· 52
SendDownloadACK ····· 280, 323, 331
SERIALIZABLE ····· 107
SET ····· 149, 345, 504
SET OPTION ····· 399
SET TEMPORARY OPTION ····· 401
SIGNAL ····· 418
SOME ····· 150, 151
sp_hook_dbmlsync_download_fatal_sql_error ····· 323
sp_hook_dbmlsync_download_log_ri_violation ····· 327
sp_hook_dbmlsync_download_ri_violation ····· 326
sp_hook_dbmlsync_download_sql_error ····· 324
sp_login_environment ····· 399
sp_tsql_environment ····· 399
SQL ····· 5
SQL Anywhere
グローバル・オプション ····· 397
ニュースグループ ····· 68
ローカル・オプション ····· 397
SQL Remote ····· 274
SQLSTATE ····· 217, 344, 418, 459
STDDEV ····· 163, 168
STDDEV_POP ····· 163, 168
STDDEV_SAMP ····· 163, 168
STOP ENGINE ····· 487
STOP SYNCHRONIZATION DELETE ·· 318
STRING( ) ····· 141, 146, 174
STRING_RTRUNCATION ····· 12, 400
SUBSTR( ) ····· 141, 144
SUM( ) ····· 163, 167
synchronization_statistics ····· 297, 299
SYS ····· 466
SYS.DUMMY ····· 93, 136
SYSCOLSTAT ····· 575
SYSFOREIGNKEY ····· 190
SYSOPTION ····· 398
SYSPUBLICATION ····· 329
SYSSYNC ····· 291, 328
SYSTABLE ····· 190
SYSUSERMESSAGES ····· 424
SYSUSERPERM ····· 462, 465

T

TableOrder ····· 282
TEMP_SPACE_LIMIT_CHECK ····· 397
TIME ····· 504
time_statistics ····· 297, 299
TODAY ····· 140
TOP ····· 179
TRACEBACK ····· 140
TRIGGER EVENT ····· 388
TRIM( ) ····· 141, 145
TRUE ····· 137, 147, 149
TRUNCATE_DATE_VALUES ····· 398
TRUNCATE_TIMESTAMP_VALUES ··· 398
TRUNCATE_WITH_AUTO_COMMIT ····· 248, 398, 412

U

UNC ····· 76
UNION ····· 184
UNION ALL ····· 185, 187
UNKNOWN ····· 137, 147, 149
UNLOAD SELECT ····· 203
APPEND ····· 203, 204
DELIMITED BY ····· 203, 204
ESCAPE CHARACTER ····· 204
ESCAPES ····· 203, 204
FORMAT ····· 203, 204
HEXADECIMAL ····· 203, 204

ORDER ····· 204
QUOTES ····· 203, 204
UNLOAD TABLE ····· 203
ORDER ····· 203
UPDATE ····· 32, 159, 216
SET ····· 216
WHERE CURRENT OF ····· 228
カーソル指向型 ····· 228
集合指向型 ····· 220
UPDATING ····· 159
upload_delete ····· 300, 303
upload_fetch ····· 300, 305, 308
upload_insert ····· 300, 301
upload_new_row_insert ····· 300, 306
upload_old_row_insert ····· 300, 306
upload_statistics ····· 297, 299
upload_update ···· 300, 301, 305, 308
UploadOnly ····· 326
UPPER( ) ····· 141, 145
upsert ····· 313
USER_ID( ) ····· 465
USER_NAME( ) ····· 191, 465

V

VALIDATE INDEX ····· 503, 506
VALIDATE TABLE ····· 503, 505
WITH DATA CHECK ····· 505
WITH EXPRESS CHECK ····· 505
WITH FULL CHECK ····· 505, 506
WITH INDEX CHECK ····· 505, 506
VAR_POP ····· 163, 168
VAR_SAMP ····· 163, 168
VARIANCE ····· 163, 168

W

WAITFOR ····· 350
WEBSERVICE_NAMESPACE_HOST ·· 397
WHEN OTHERS ····· 342
WHERE ····· 147
WHILE ····· 349
WITH ····· 93, 194
RECURSIVE ····· 194, 195

X

XLOCK ····· 457
XML ····· 105

あ

アーティクル ····· 282
空き領域 ····· 35
アップサート ····· 313
アトミック・オペレーション ····· 339
アトミック性 ····· 403

い

一貫性 ····· 403
イベント ····· 378, 392
インクリメンタル・オンライン・ログ・バックアップ ····· 485, 489, 492
インデックス
圧縮インデックス ····· 565
再編成 ····· 552
作成 ····· 559
部分インデックス ····· 565
フル・インデックス ····· 565
インデックス・コンサルタント ····· 524

え

エイリアス ····· 96, 133, 137
エスケープ文字 ····· 77, 158
エラー処理 ····· 418
演算子 ····· 134

お

オートコミット・モード ····· 409

か

カーソル ····· 254
CLOSE ····· 267
DECLARE CURSOR FOR ····· 258
DECLARE CURSOR FOR CALL ····· 265
DECLARE CURSOR USING ····· 264
DYNAMIC SCROLL ····· 259, 261
FETCH ····· 254, 268
FOR ループ ····· 269
INSENSITIVE ····· 259, 261
NO SCROLL ····· 259, 261
OPEN ····· 267
SCROLL ····· 259, 261
SENSITIVE ····· 259, 261
カーソル不安定性 ····· 443
感知性 ····· 259
更新可能性 ····· 259
拡張ページ ····· 11, 543
加算 ····· 134
仮想カラム ····· 96
仮想テーブル ····· 105, 108
カラム ····· 18
CHECK 制約 ····· 29, 30
NOT NULL 制約 ····· 29
PRIMARY KEY 制約 ····· 30, 31
UNIQUE 制約 ····· 32
外部キー制約 ····· 31
カラム制約 ····· 28
プロパティ ····· 18
環状デッドロック ····· 455
関数 ····· 362
決定的 ····· 364
完全比較 ····· 566
カンマ・ジョイン演算子 ···· 106, 130, 131

き

キャッシュ ····· 480
共有テーブル・スキーマ・ロック ····· 436
共有ライブラリ ····· 354
共有ロー読み込みロック ····· 435

く

グラフィカルなプラン ····· 532
繰り返し不可能読み出し ····· 446
グローバル・テンポラリ・テーブル ····· 7
グローバル永続テーブル ····· 7
グローバル変数 ····· 392

け

警告 ····· 418
計算カラム ····· 19

権限 479
検索引数不可能 572
減算 134
幻ロー 448
幻ロー位置ロック 435

こ

更新可能なビュー 192
更新内容消失 448
更新の抑制 450
コンボイ 461

さ

再帰ユニオン 195
再実行ログ 481
サイレント・ロールバック 430
削除可能なビュー 192
サブスクリプション 285
サブトランザクション 413
差分オンライン・ログ・バックアップ 485, 489, 491

し

シータ・ジョイン 572
式 134
識別子 9
時刻型 13
システム・カタログ 4
持続性 403
実行権限 472
シャドー・テーブル 319
集合関数 100, 140, 163
ジョイン 106, 108, 118
乗算 134
冗長性 53
除算 134
シングル接続 417
シングルユーザ 417
シンクロナイゼーション 274
真数型 14

す

スカラ関数 140, 141
スター・ジョイン 124
ストアド・プロシージャ 125, 354
スナップショット同期 313
スレッド・デッドロック 457
スレッド・プール 457

せ

正規化 53
第 1 正規形 54
第 2 正規形 55
第 3 正規形 56
第 4 正規形 58
第 5 正規形 59
ボイスコッド正規形 57
生成規則 2
セーブポイント 413
接続名 464

そ

相関サブクエリ 153
挿入可能なビュー 192
挿入ロー位置ロック 435
ソート
BINARY と CHARACTER での違い 12

た

ダーティ・リード 442
対幻ロー位置ロック 435
対挿入ロー位置ロック 435
代入文 345
短期ロック 434
探索条件 148
断片化 538

ち

チェックポイント・ログ 482
抽出テーブル 114
長期ロック 434

て

ディメンジョン・テーブル 124
データ型 10
BIGINT 15
BINARY 10
BIT 15
CHARACTER 10
DATE 13
DATETIME 34
DECIMAL 14
DOUBLE 15
FLOAT 15
IMAGE 34
INT 15
INTEGER 15
LONG BINARY 10
LONG VARCHAR 10
MONEY 34
NUMERIC 14
OLDBIT 34
REAL 15
SMALLDATETIME 34
SMALLINT 15
SMALLMONEY 34
SYSNAME 34
TEXT 34
TIME 13
TIMESTAMP 13
TINYINT 15
UNIQUEIDENTIFIER 34
UNIQUEIDENTIFIERSTR 34
UNSIGNED BIGINT 15
UNSIGNED INT 15
UNSIGNED INTEGER 15
UNSIGNED SMALLINT 15
UNSIGNED TINYINT 15
XML 34
時刻型 13
真数型 14
操作規則 10
バイナリ整数型 15
日付型 13
物理記憶領域 10
浮動小数点数型 14
プロパティ 10
文字列型 10
ユーザ定義データ型 33
リテラル 16
データベース・ファイル 480
データベース所有者 466

データ変換……72
テーブル……4, 6
5つのタイプ……6
グローバル・テンポラリ・テーブル……7, 46
グローバル永続テーブル……7, 8
テンポラリ・テーブル……6, 46
プロキシ・テーブル……7
ベース・テーブル……7
リモート・テーブル……7
ローカル・テンポラリ・テーブル…7, 48
テーブル・コンテンツ・更新インテントロック……436
テーブル・コンテンツ読み込みロック…436
テーブル・ヒント……454
テーブル・ページ……542
テーブル権限……467
テーブル制約……35
FOREIGN KEY 制約……37
PRIMARY KEY 制約……36
UNIQUE 制約……39
CHECK 制約……36
デッドロック……455
テンポラリ・テーブル……6, 46
NOT TRANSACTIONAL……83
テンポラリ・ビュー……194
テンポラリ・ファイル……483
配置先……574

と

同期……274
スナップショット同期……313
統計関数……101, 168
同時実行性……433
特別権限……473
独立性……403
独立性レベル……437
トランザクション……403
オートコミット・モード……409
ネストされたトランザクション……411
非連鎖モード……406
マニュアル・モード……409
連鎖モード……406
トランザクション・ログ・ファイル……406, 481
トリガ……159, 367
after delete トリガ……235
after row トリガ……372
after statement トリガ…217, 235, 373
after update トリガ……217
before delete トリガ……235
before row トリガ……368
before update トリガ……217
取り消しログ……482

な

長いカラム……542

は

パーミッション……479
排他テーブル・スキーマ・ロック……436
排他ロー書き込みロック……435
バイナリ整数型……15
バックアップ……485
インクリメンタル・オンライン・ログ・バックアップ……485, 489, 492
差分オンライン・ログ・バックアップ……485, 489, 491
フル・オフライン・イメージ・バックアップ……485
フル・オンライン・イメージ・バックアップ……485, 488, 489
ライブ・ログ・バックアップ…485, 494
パフォーマンス・カウンタ……566
パブリケーション……282
パブリック・デフォルト値……398
バルク・コピー・プログラム（bcp.exe）…78

ひ

比較……150
BINARY と CHARACTER での違い……12
大文字と小文字を区別……12
日付型……13
ビュー……5, 190
ビュー権限……470
非連鎖モード……406
ヒント……107

ふ

ファクト・テーブル……124
不安定な更新……447
ブール式……147
フック・プロシージャ……323
浮動小数点数型……15
プライマリ・キー……4
フル・オフライン・イメージ・バックアップ……485
フル・オンライン・イメージ・バックアップ……485, 488
プロキシ・テーブル……40, 274
プロシージャ……354, 362
ブロック……437
プロファイリング……530
分断統治法……122

へ

ページ・レベル取り消しログ……482
変数……392

ま

マニュアル・モード……409
マルチ接続……417
マルチテーブル・ジョイン……118
マルチユーザ……417

み

短いカラム……542
ミューテックス……460
ミラー・ログ・ファイル……482

も

文字列型

ゆ

ユーザ……465
dbo……466
SYS……466
作成……463
パスワード……463
無効……466
ユーザ・デフォルト値……398

よ

要求レベルのロギング…… 513

ら

ライブ・ログ・バックアップ…… 485, 494
楽観ロック…… 577
ラテラル・プロシージャ・コール…… 130

り

リカバリ…… 480
リストア…… 495
リモート・データ・アクセス……40
リモート・テーブル……40
リレーショナル・データベース…… 2, 4
　コッドの 12 のルール……4
リレーション……4

れ

例外…… 359
例外ハンドラ…… 341
連結…… 134
連鎖モード…… 406

ろ

ロー・セグメント…… 542
ロー・レベル取り消しログ…… 482
ローカル・テンポラリ・テーブル
ローカル・ビュー…… 194
ロー残部…… 543
ロールバック・ログ…… 482
ロギング…… 480
　開始…… 513
　停止…… 514
　フィルタリング…… 522
　要求レベルのロギング…… 513
ロック…… 433
　解放…… 437
　共有テーブル・スキーマ・ロック…… 436
　共有ロー読み込みロック…… 435
　幻ロー位置ロック…… 435
　取得…… 434
　昇格…… 457
　挿入ロー位置ロック…… 435
　対幻ロー位置ロック…… 435
　対挿入ロー位置ロック…… 435
　短期ロック…… 434
　長期ロック…… 434
　テーブル・コンテンツ・更新インテントロック…… 436
　テーブル・コンテンツ読み込みロック…… 436
　排他テーブル・スキーマ・ロック…… 436
　排他ロー書き込みロック…… 435

SQL ／コマンド構文

<action>……29
<add_space_to_database_file>…… 540
<add_user_to_group>…… 476
<aggregate_builtin_function_call>…… 163
<alias_name> ……115, 225
<alias_name_list>…… 115
<all_values_list> ……65
<alphabetic> ……8
<argument> ……125, 266
<argument_list> ……125, 266
<article> …… 282
<article_list> …… 282
<as_structure> …… 367
<as_table> …… 373
<as_table_name> …… 373
<assignment_statement> …… 345
<assignment_target> …… 345
<backup_comment> …… 489
<backup_database_to_image_files> …… 488
<basic_case_expression> …… 138
<basic_case_statement> …… 347
<basic_expression> …… 134
<begin_block> …… 338
<begin_label> …… 339
<begin_transaction> …… 405
<between_predicate> …… 155
<boolean_expression> ……147, 149
<builtin_data_type> ……10
<builtin_function_call> …… 560
<builtin_function_name> …… 560
<builtin_user_defined_data_type>…34
<call_argument>…… 354
<call_argument_list>…… 354
<call_statement> …… 354
<case_expression> …… 138
<case_statement> …… 347
<char_type> ……10
<close_cursor>…… 267
<clustering> ……29
<column_constraint>……28
<column_definition> ……9, 18
<column_definition_list> ……9, 18
<column_definition_term> ……9, 18
<column_list>…… 282
<column_name> ……9, 67, 159, 216, 367, 467
<column_name_list> …… 36, 67, 467
<column_name_literal> …… 159
<column_property> ……18
<column_reference> ……134, 225
<column_width> ……85
<column_width_list> ……85
<commit> …… 405
<commit_action> ……47
<comparison_operator> …… 150
<comparison_predicate> …… 150
<concurrency_setting> ……93
<connection_script_name> …… 295
<connection_string> ……41
<connection_variable_name> …… 392
<consolidated_table_name> …… 301
<constant_expression> ……27
<constant_function_call> ……27
<constraint_name> ……29
<constraint_or_prefix> ……37
<constraint_or_role_name> …… 552
<constraint_prefix> ……29
<correlation_name> ……106, 225, 240
<create_after_row_trigger> …… 372
<create_after_statement_trigger>…373
<create_before_row_trigger> …… 367
<create_connection_variable> …… 392
<create_domain> ……33
<create_event> …… 378
<create_external_login> ……42
<create_function> …… 362
<create_global_permanent_table>…8
<create_global_temporary_table>…47
<create_index> …… 559
<create_local_temporary_table> ……49
<create_message> …… 423

<create_procedure> ····· 354
<create_proxy_table> ·····44
<create_publication> ····· 282
<create_remote_and_proxy_table>···43
<create_scheduled_event> ····· 383
<create_server>·····41
<create_synch_subscription> ····· 286
<create_synch_user>····· 283
<create_table>·····6
<create_trigger> ····· 367
<create_typed_event>····· 378
<create_user> ····· 463
<create_user_defined_event> ····· 387
<create_user_group> ····· 476
<create_view> ····· 190
<current_values_table_name> ···· 305
<cursor_declaration> ····· 270
<cursor_for_call> ····· 266
<cursor_for_select> ····· 259
<cursor_name> ·····229, 244, 259
<cursor_positioning>····· 268
<cursor_select_variable> ····· 264
<cursor_type> ····· 259
<cursor_type_parameter> ····· 537
<cursor_using_select>····· 264
<data_type> ·····9, 356
<database_file_reference> ····· 540
<date_time_type>·····13
<day_name> ····· 384
<day_number> ····· 384
<dbmlsync_command> ····· 288
<dbmlsync_connection_option> ··· 288
<dbmlsync_extended_options> ···· 288
<dbmlsync_interface_option> ····· 288
<dbmlsync_option> ····· 288
<dbmlsync_option_list> ····· 288
<dbmlsync_session_option>····· 288
<dbspace_name>·····2, 8, 560
<declaration>····· 339
<declaration_list>····· 339
<declare_cursor> ····· 258
<declare_cursor_for_call>····· 266
<declare_cursor_for_select>····· 259
<declare_cursor_using_select>···· 264
<declare_exception_name> ····· 341
<declare_local_temporary_table> ···50
<declare_local_variable> ····· 340
<default_value> ·····21
<delete_where_current_of_cursor>
····· 244
<derived_column_name_list> ····· 115
<derived_table> ····· 115
<domain_definition>·····33
<domain_property>·····33
<domain_property_list>·····33
<double_precision>·····15
<end_of_input_marker>·····85
<ending_time> ····· 383
<escape_character> ····· 74, 203
<event_comparison_operator>···· 379
<event_condition_name> ····379, 380
<event_condition_value>····· 379
<event_name>····· 378
<event_on_days> ····· 384
<event_parameter_assignment> ·· 388
<event_parameter_function_call>·· 380
<event_parameter_list> ····· 388
<event_parameter_name> ····· 380
<event_parameter_name_string>·· 380
<event_parameter_value> ····· 388
<event_predicate> ····· 379
<event_repeat_every> ····· 384
<event_schedule_item> ····· 383
<event_schedule_item_name> ···· 383
<event_schedule_list>····· 383
<event_start_date> ····· 384
<event_start_times>····· 383
<event_type> ····· 379
<event_where_clause>····· 379
<exact_keyword> ·····14
<exact_numeric_type> ·····14
<exact_scale>·····14
<exception_handler> ····· 342
<exception_name> ····· 341
<exception_name_list>····· 342
<executable_statement> ····· 341
<execute_immediate> ····· 351
<execute_option>····· 351
<existing_column_name> ····· 560
<exists_predicate> ····· 153
<expression> ····· 134
<extended_option> ····· 284
<extended_option_list> ····· 284
<extended_option_name> ····· 284
<extended_option_value>····· 284
<fetch_cursor>····· 268
<fetch_into_list> ····· 268
<file_input_option_list> ·····84
<fired_by> ·····367, 368
<first_scheduled_time>····· 383
<float_numeric_type> ·····15
<for_clause>·····93
<for_cursor_loop> ····· 269
<for_intent_clause> ····· 93, 259
<for_label> ····· 270
<for_loop_body_statements>····· 270
<for_name> ····· 270
<for_xml_clause> ·····93
<foreign_key_table_constraint> ·····37
<free_percent> ·····35, 74
<from_clause>·····105, 240
<full_outer_join>····· 106
<function_argument_list>
·····140, 362, 560
<function_call> ····· 140
<function_characteristic>····· 362
<function_parameter> ····· 362
<function_parameter_list> ····· 362
<grant_dba_privilege> ····· 475
<grant_execute_privilege> ····· 472
<grant_remote_dba_privilege>···· 475
<grant_resource_privilege> ····· 474
<grant_table_privileges>····· 467
<grant_view_privileges> ····· 470
<group_by_clause> ·····160, 171
<group_by_list>·····160, 171
<group_name_list> ····· 476
<having_clause>····· 175
<hexadecimal_option>····· 209
<high_message_number>····· 423
<hint> ····· 107
<hint_list> ····· 107
<hints> ····· 107
<identifier>·····8
<if_expression> ····· 137
<if_statement>····· 346
<image_destination> ····· 489
<image_option> ····· 488
<in_dbspace_clause> ·····2, 8, 560
<in_predicate>····· 154
<index_column> ····· 560
<index_column_list> ····· 560
<index_name>·····506, 560
<initial_query_specification> ····· 196

<inline_data> 85
<inline_input_option_list> 84
<inner_join> 106
<input_delimiter> 85
<input_escape_character> 85
<input_file> 84
<input_format> 85
<input_format_name> 85
<input_name> 74
<input_name_list> 74
<input_option> 84
<input_option_list> 84
<insert_all_values> 65
<insert_named_values> 67
<insert_select_all_columns> 69
<insert_select_auto_name> 72
<insert_select_column_list> 71
<integer_type> 15
<isolation_level> 267
<isql_inline_input> 84
<isql_input> 84
<isql_input_from_file> 84
<isql_input_with_prompt> 84
<isql_output> 208
<join_operator> 106, 118
<lateral_derived_table> 130
<leave_statement> 339
<left_outer_join> 106
<like_predicate> 157
<list_delimiter> 163
<load_delimiter> 74
<load_filespec> 74
<load_option> 74
<load_option_list> 74
<load_table> 74
<local_user_id> 42
<local_variable_name> 340
<local_view> 194
<local_view_column_name_list> .. 194
<local_view_list> 194
<local_view_name> 194
<maximum_length> 11, 14
<maximum_row_count> 179, 225
<message_literal> 423
<message_number> 423
<message_text> 423
<MobiLink_client_setup> 281
<MobiLink_protocol> 284
<MobiLink_protocol_parameters> .. 284
<MobiLink_script> 295
<MobiLink_user> 284
<MobiLink_user_list> 284
<move_to_row_number> 268
<move_to_row_offset> 268
<multi_row_where_clause> .. 219, 236
<named_values_list> 67
<new_column_name> 560
<non_primary_key_column_name> .. 302
<non_temporary_alphabetic> 8
<non_temporary_identifier> 8
<null_test_predicate> 156
<number_literal> 22, 134, 348
<number_of_units> 540
<numeric> 8
<old_values_table_name> 307
<on_action> 29
<on_condition> 106
<on_condition_shorthand> ... 106, 118
<on_existing> 65
<open_cursor> 267
<option_name> 399
<option_value> 399, 400
<order_by_clause> 176
<order_by_item> 176
<order_number> 367
<ordinary_builtin_function_call> .. 140
<ordinary_builtin_function_name> 140
<outer_query_specification> 196
<output_column_width> 208
<output_column_width_list> 208
<output_delimiter> 208
<output_escape_character> 208
<output_file> 208
<output_format> 208
<output_format_name> 208
<output_option> 208
<output_quote> 209
<owner_name> 8, 65, 106, 216, 234, 266, 354, 467, 560
<parameter_default> 354
<parameter_disposition> 354, 356
<parameter_list> 354, 356
<parameter_name> 266, 354, 356
<parameter_specification> 354
<partition_size> 22
<password> 463
<permanent_table_name> 8
<positional_password_list> 463
<predicate> 147
<primary_key_column_name> 302
<primary_key_table_constraint> 36
<procedure_name> 125, 266, 354
<procedure_reference> 125
<prompt_input_option_list> 84
<proxy_column> 44
<proxy_column_list> 44
<publication> 282
<publication_list> 286
<publication_name> 282
<quantifier> 150
<query_expression> 93, 184
<query_operator> 93, 184
<query_specification> 93, 178
<raiserror> 422
<recursive_query_specification> ... 196
<referencing_as_structures> 367
<referencing_as_tables> 373
<release_savepoint> 413
<remote_location> 43
<remote_password> 42
<remote_table_name> 299
<remote_user_id> 42
<reorganize_index> 552
<reorganize_table> 547
<resignal_exception> 418
<result_definition> 125
<result_definition_list> 125
<result_set_column> 354
<result_set_column_list> 354
<return_code> 354
<return_statement> 354
<right_outer_join> 106
<role_name> 38
<rollback> 405
<rollback_to_savepoint> 413
<rollback_trigger> 430
<row_count> 240
<row_number> 240
<row_range> 179, 225, 240
<row_script_name> 300
<row_structure_name> 367
<savepoint> 413
<savepoint_name> 413
<schedule_interval> 384

<script_version> 295
<searched_case_expression> 139
<searched_case_statement> 348
<select> 69, 93, 184, 194, 259
<select_all_columns> 69
<select_auto_name> 72
<select_column_list> 71
<select_for_unload> 203
<select_into> 183
<select_into_local_temporary_table> 52
<select_into_variable_list> 183
<select_item> 132
<select_item_alias> 132
<select_item_number> 176
<select_list> 132
<server_access_class> 41
<server_local_name> 41
<session_script_name> 296
<set_clause> 225
<set_delete> 237, 240
<set_item> 225
<set_option> 399
<set_public_default_option> 399
<set_temporary_local_option> 401
<set_update> 220, 225
<set_user_default_local_option> .. 401
<shadow_table_name> 319
<signal_exception> 418
<simple_expression> 134
<simple_fired_by> 367
<simple_set_clause> 216
<simple_set_delete> 240
<simple_set_item> 216
<single_precision> 15
<single_row_where_clause> .. 216, 234
<special_builtin_function_call> 140
<special_literal> 25
<special_update_default> 26
<sqlstate_error_value> 341
<start_at_row_number> 179, 225
<statement_list> 341
<statistics_level> 537
<store_MobiLink_connection_script> 295
<store_MobiLink_row_script> 300
<store_MobiLink_script> 295
<store_MobiLink_session_script> .. 296
<store_MobiLink_table_script> 299
<store_publication_defaults> 293
<string_containing_SQL_statement> 537
<string_literal> 22, 132, 348
<string_type> 10
<subquery> 93
<substitution_list> 423
<substitution_value> 423
<table_check_constraint> 36
<table_constraint> 36
<table_element> 9, 35
<table_element_list> 8
<table_expression> 106, 118
<table_expression_list> 105
<table_name> 29, 47, 50, 106, 216, 234, 367, 467, 560
<table_or_view_list> 228
<table_or_view_reference> .. 229, 240
<table_privilege> 467
<table_privileges> 467
<table_property> 35
<table_reference> 106
<table_script_name> 299
<table_specification> 105
<table_term> 106, 118
<target_table_name> 65
<temporary_table_name> 49, 52
<transaction_log_folder> 288
<transaction_name> 405
<trigger_event> 388
<trigger_name> 367
<trigger_predicate> 159
<truth_value> 147
<typical_download_cursor> 314
<typical_download_delete_cursor> 319
<typical_download_delete_truncate> 321
<typical_extended_upload_update> 306
<typical_lateral_procedure_call> .. 130
<typical_multi_row_delete> 236
<typical_multi_row_update> 219
<typical_recursive_union> 195
<typical_single_row_delete> 234
<typical_single_row_update> 216
<typical_upload_delete> 304
<typical_upload_fetch> 305
<typical_upload_insert> 301
<typical_upload_new_row_insert> . 307
<typical_upload_old_row_insert> .. 307
<typical_upload_update> 302
<unique_table_constraint> 39
<units> 540
<unload> 203
<unload_delimiter> 203
<unload_filespec> 203
<unload_select> 203
<unload_select_option> 203
<unload_table> 203
<unload_table_option> 203
<update_status> 537
<update_where_current_of_cursor> 228
<user_defined_data_type> 33
<user_defined_event_parameter_name> 381
<user_defined_function_call> 140, 362
<user_defined_function_name> 140, 362
<user_estimate> 147
<user_id> 401
<user_id_list> 463
<user_name_list> 3
<validate_other_index> 506
<validate_primary_key> 506
<validate_table> 505
<value> 65
<value_list> 65
<variable_reference> 134
<view_column_name_list> 190
<view_name> 106, 240, 470
<view_privilege> 470
<view_privileges> 470
<view_reference> 106
<when_constant> 348
<when_deleted_column_name> ... 319
<when_exception> 342
<when_exception_list> 342
<when_updated_column_name> .. 314
<where_clause> 147
<where_current_of_clause> .. 229, 244
<while_label> 350
<while_loop> 349
<with_check> 505
<with_clause> 194
<write_graphical_plan_to_file> 536
<xml_output_filespec> 537

装幀・アートディレクション：　犬飼健二
デザイン：　春田沙哉子（犬飼デザインサイト）
編集協力・本文デザイン・DTP：株式会社トップスタジオ

SQL Anywhere® Studio 9 デベロッパーズガイド

2006 年 2 月 27 日　初版第 1 刷発行

著者　Breck Carter（ブレック・カーター）
訳者　株式会社トップスタジオ
監修　アイエニウェア・ソリューションズ株式会社
発行人　速水 浩二
発行所　株式会社翔泳社（http://www.seshop.com）
印刷・製本　株式会社 廣済堂

本書は著作権法上の保護を受けています。本書の一部または全部について（ソフトウェアおよびプログラムを含む）、株式会社 翔泳社から文書による許諾を得ずに、いかなる方法においても無断で複写、複製することは禁じられています。

本書へのお問い合わせについては、ii ページに記載の内容をお読みください。

落丁・乱丁はお取り替えいたします。03-5362-3705 までご連絡ください。

ISBN4-7981-1018-3　Printed in Japan