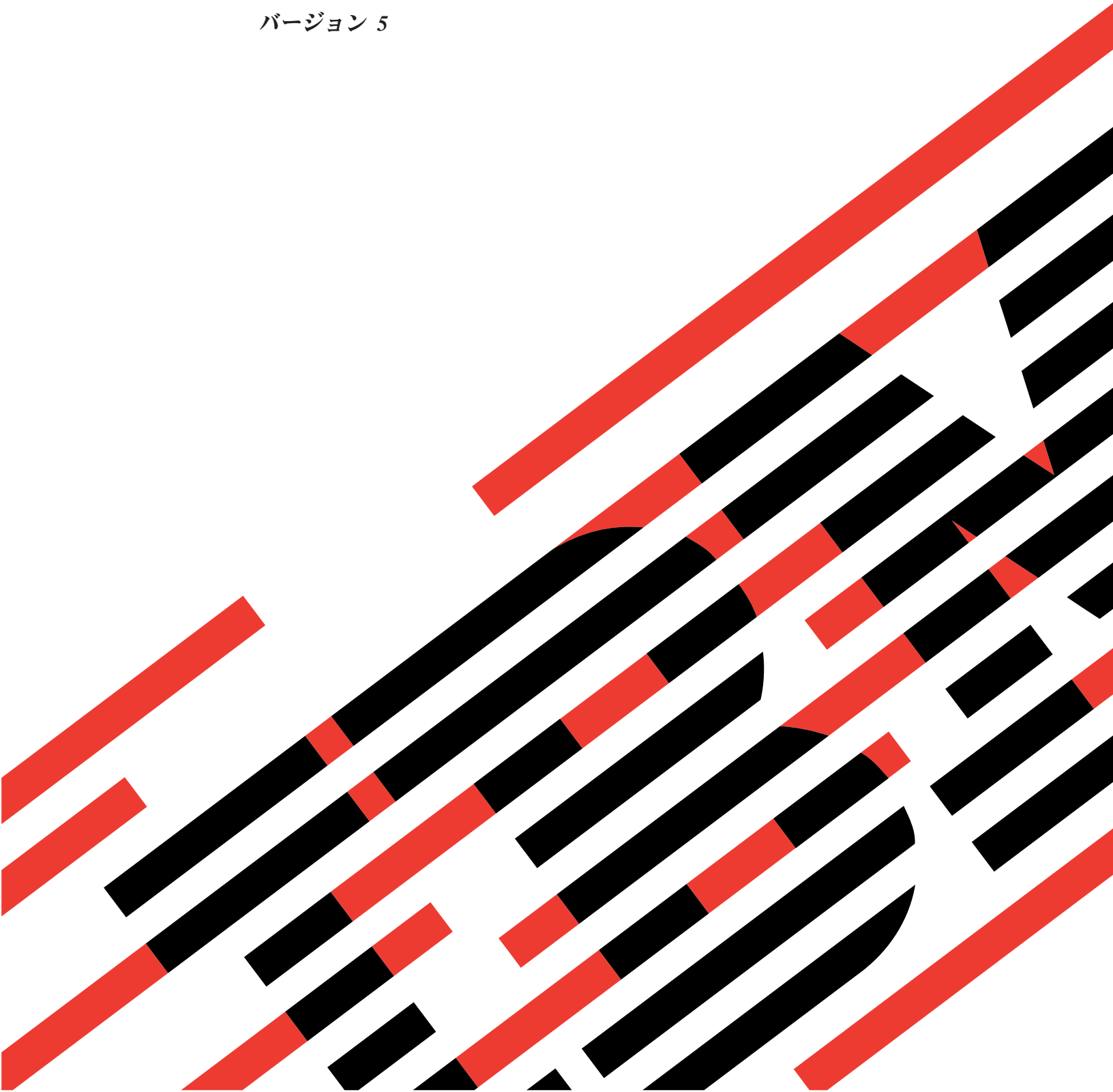
IBM

eserver

iSeries

DB2 Universal Database for iSeries SQL プログラミング 概念

バージョン 5



IBM

@server

iSeries

DB2 Universal Database for iSeries SQL プログラミング 概念

バージョン 5

本マニュアルに関するご意見やご感想は、次の URL からお送りください。今後の参考にさせていただきます。

http://www.ibm.com/jp/manuals/main/mail.html

なお、日本 IBM 発行のマニュアルはインターネット経由でもご購入いただけます。詳しくは

http://www.ibm.com/jp/manuals/ の「ご注文について」をご覧ください。

(URL は、変更になる場合があります)

原　典： RBAF–Y000–03
iSeries
DB2 Universal Database for iSeries Programming Concepts
Version 5

発　行： 日本アイ・ビー・エム株式会社

担　当： ナショナル・ランゲージ・サポート

第1刷 2002.8

目次

DB2 UDB for iSeries SQL プログラミング概念について

本書は、プログラマーおよびデータベース管理者に以下の基本的な SQL プログラミング概念について説明します。

- DB2 UDB for iSeries ライセンス・プログラムを使用する方法
- データベース内のデータにアクセスする方法
- SQL ステートメントを含むアプリケーション・プログラムを準備、実行、およびテストする方法

アプリケーション・プログラミング環境において、DB2 UDB for iSeries SQL のインプリメンテーションを行うための指針と例に関する詳細については、iSeries Information Center 内の下記の資料を参照してください。

- SQL 解説書
- DB2 UDB for iSeries ホスト言語での SQL プログラミング
- DB2 UDB for iSeries データベース・パフォーマンスおよび Query 最適化
- SQL 呼び出しレベル・インターフェース
- SQL メッセージおよびコード

本書の詳細については、以下のトピックを参照してください。

- 『本書の対象読者』
- 『SQL ステートメントの例に関する前提事項』
- x ページの『コードに関する特記事項』
- xi ページの『構文図の見方』
- xii ページの『SQL プログラミング概念 V5R2 版の新規情報』

本書の対象読者

本書が対象とする読者は、COBOL for iSeries、ILE COBOL for iSeries、iSeries PL/I、ILE C for iSeries、ILE C++、REXX、RPG III (RPG for iSeries の一部)、または、ILE RPG for iSeries 言語に精通していて、これらの言語を用いてプログラミングを行うことができ、基本的なデータベース・アプリケーションについて理解することができるアプリケーション・プログラマーおよびデータベース管理者の方々です。

以下のセクションも参照してください。

- 『SQL ステートメントの例に関する前提事項』
- x ページの『コードに関する特記事項』
- xi ページの『構文図の見方』

SQL ステートメントの例に関する前提事項

本書で示されている SQL ステートメントの例は、付録 A DB2 UDB for iSeries サンプル表に基づいており、以下の事項を前提としています。

- それらの例は、対話式 SQL 環境で使用されるか、あるいは ILE C または COBOL で書かれています。COBOL プログラム内での SQL ステートメントの区切りには、EXEC SQL および END-EXEC が使用されています。COBOL プログラムの中で SQL ステートメントを使用する方法は、『COBOL アプリケーションでの SQL ステートメントのコーディング方法』で説明されています。ILE C プログラムの中で SQL ステートメントを使用する方法は、『C アプリケーションでの SQL ステートメントのコーディング方法』で説明されています。
- 各 SQL ステートメントの例は、ステートメントの文節ごとに行を変えて、数行にまたがって示されています。
- SQL のキーワードは太字で示されています。
- 付録 A DB2 UDB for iSeries サンプル表に記載されている表名は、スキーマ CORPDATA を使用します。サンプル表にない表名は、ユーザーが作成するスキーマを使用しなければなりません。
- 計算対象の列は、括弧 () と大括弧 [] で囲まれています。
- SQL の命名規則が使用されています。
- APOST および APOSTSQL 事前コンパイラー・オプションは、COBOL でのデフォルト・オプションではありませんが想定されています。SQL およびホスト言語ステートメント内の文字ストリング・リテラルは、アポストロフィ (') によって区切られています。
- 特に断りがない限り、*HEX の分類順序が使用されています。
- どの例でも、通常、SQL ステートメントの構文全体が示されているわけではありません。本書に記載されているステートメントの完全な説明と構文については、SQL 解説書を参照してください。

上記の前提と異なる例が提示されている場合は、必ずその旨が記述されています。

本書はアプリケーション・プログラマーを対象としているため、ほとんどの例はアプリケーション・プログラムの中で書かれているものとして示されています。ただし、若干の変更を加えれば、対話式 SQL を使用して対話式で実行することができる例も多数あります。対話式 SQL を使用する場合の SQL ステートメントの構文は、同じステートメントをプログラムに組み込む場合の形式と若干異なります。

コードに関する特記事項

本書には、プログラミングの例が含まれています。

IBM は、お客様に、すべてのプログラム・コードのサンプルを使用することができる非独占的な著作使用権を許諾します。お客様は、このサンプル・コードから、お客様独自の特別のニーズに合わせた類似のプログラムを作成することができます。

すべてのサンプル・コードは、例として示す目的でのみ、IBM により提供されます。 このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。 従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。

ここに含まれるすべてのプログラムは、現存するままの状態で提供され、いかなる保証条件も適用されません。商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任の保証の適用も一切ありません。

構文図の見方

本書では、以下ように定義された構造を使用して構文を記述します。

- 構文図は矢印に従って左から右へ、上から下へと見ます。

  ▶▶── 記号はステートメントの始まりを示します。

  ──▶ 記号は、ステートメントの構文が次の行へ続くことを示します。

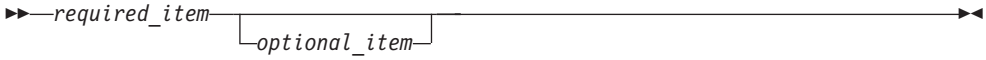
  ▶── 記号は、ステートメントが前の行から続いていることを示します。

  ──▶◀ 記号はステートメントの終わりを示します。

  完結したステートメント以外の構文単位の図は、▶── 記号で始まり、──▶ 記号で終わります。

- required_item (必須項目) は水平線 (主線) 上に示されています。

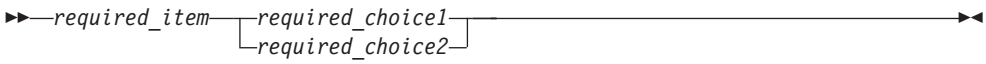
- オプショナル項目は主線より下に示されています。



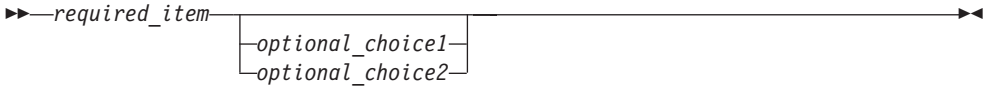
  optional_item が主線より上に示されている場合、その項目はステートメントの実行に対しては何の効果も持たず、読みやすさのためだけに使用されています。

- 2 つ以上の項目から選択できる場合には、それらは上下に重ねて (スタックされて) 示されています。

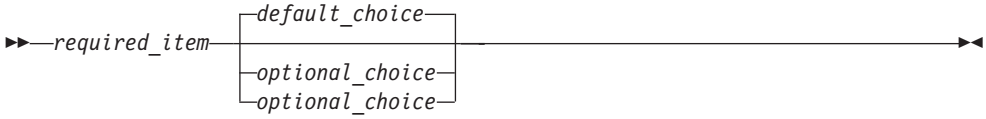
  項目の 1 つを選択しなくてはならない 場合は、スタックの内の 1 つの項目が主線上に示されます。



  項目の 1 つを選択することが任意の場合は、スタック全体が主線より下に示されています。



  項目の 1 つが省略時値である場合には、それが主線より上に示され、残りの選択項目は主線より下に示されています。



- 主線より上の左に戻る矢印は、反復可能な項目であることを示します。

  反復矢印にコンマが含まれている場合は、反復される各項目をコンマで区切らなくてはなりません。

スタックより上にある反復矢印は、スタック内の項目を反復できることを示します。

- キーワードは英大文字で示されます (たとえば、FROM)。それらは示されているとおり正確につづらなければなりません。変数はすべて小文字で示されています (たとえば、*column-name*)。それらはユーザー提供の名前または値を表します。
- 句読記号、括弧、算術演算子、あるいは他にそのような記号が示されている場合には、それらを構文の一部として入力しなくてはなりません。

SQL プログラミング概念 V5R2 版の新規情報

本リリースでは本書に以下の主な変更がありました。

- ユーザー定義表関数。詳細については、 217 ページの『ユーザー定義関数 (UDF)』を参照してください。
- 隔離解除されたユーザー定義関数。詳細については、 263 ページの『関数を隔離または隔離解除にする』を参照してください。
- 保管ポイント。詳細については、 344 ページの『保管ポイント』を参照してください。
- 一時表。
- 識別列。詳細については、 54 ページの『識別列の作成および変更』を参照してください。
- スカラー副選択。
- SQL プロシージャーのデバッグ。詳細については、 179 ページの『ストアード・プロシージャーのデバッグ』 を参照してください。

第 1 章 DB2 UDB for iSeries 構造化照会言語 (SQL) の紹介

本章では、DB2 UDB for iSeries および DB2 UDB Query Manager and SQL Development Kit バージョン 5 ライセンス・プログラムを用いて、iSeries サーバーで構造化照会言語 (SQL) を実際に使用する方法について説明します。SQL では、リレーショナル・モデルのデータに基づいて情報を管理します。SQL ステートメントは、高水準言語の中に組み込むか、動的に準備して実行するか、あるいは対話式で実行することができます。

SQL はステートメントおよび文節から構成され、これによってデータベースのデータで何をしたいか、またどのような条件のもとでそれを実行したいかを記述します。

本章では、以下の事柄について説明します。

- 『SQL の概念』
- 6 ページの『SQL オブジェクト』
- 11 ページの『アプリケーション・プログラム・オブジェクト』

SQL は、IBM 分散リレーショナル・データベース・アーキテクチャー (DRDA*) を使用して遠隔リレーショナル・データベースのデータにアクセスすることができます。この機能については、本書の 『第 21 章 分散リレーショナル・データベース機能』で説明されています。DRDA の詳細については、分散データベース・プログラミングに記載されています。

SQL の概念

DB2 UDB for iSeries SQL は、以下の主要部分から構成されています。

- SQL 実行時サポート

  SQL 実行時サポートは、SQL ステートメントを解析し、任意の SQL ステートメントを実行します。このサポートは OS/400 ライセンス・プログラムの一部であり、このサポートによって、DB2 UDB Query Manager and SQL Development Kit ライセンス・プログラムが導入されていないシステム上で、SQL ステートメントが含まれているアプリケーションを実行することができます。
- SQL 事前コンパイラー

  SQL 事前コンパイラーは、ホスト言語の組み込み SQL ステートメントの事前コンパイルをサポートします。以下の言語がサポートされています。
  - ILE C
  - ILE C++ for iSeries
  - ILE COBOL
  - COBOL for iSeries
  - iSeries PL/I
  - RPG III (RPG for iSeries の一部)
  - ILE RPG

SQL ホスト言語事前コンパイラーは、SQL ステートメントを含むアプリケーション・プログラムを準備します。次にホスト言語コンパイラーは事前コンパイルされたホスト・ソース・プログラムをコンパイルします。事前コンパイルの詳細については、*ホスト言語での SQL プログラミング* の SQL ステートメントでのプログラムの準備および実行を参照してください。事前コンパイラーのサポートは、DB2 UDB Query Manager and SQL Development Kit ライセンス・プログラムの一部です。

- SQL 対話式インターフェース

  SQL 対話式インターフェースにより、SQL ステートメントの作成と実行を行うことができます。対話式 SQL に関する詳細については、『第 17 章 対話式 SQL の使用』を参照してください。対話式 SQL は、DB2 UDB Query Manager and SQL Development Kit ライセンス・プログラムの一部です。

- Run SQL スクリプト

  iSeries ナビゲーターの「Run SQL Scripts」ウィンドウを使用すると、SQL ステートメントのスクリプトを作成、編集、実行、および、トラブルシューティングすることができます。「Run SQL Scripts」は、iSeries ナビゲーターの一部です。詳細については、298 ページの『「SQL スクリプトの実行」を使用したデータベースの照会』を参照してください。

- SQL ステートメント実行 CL コマンド

  RUNSQLSTM により、ソース・ファイルに格納されている一連の SQL ステートメントを実行することができます。 SQL ステートメントの実行コマンドに関する詳細については、『第 18 章 SQL ステートメント処理プログラムの使用』を参照してください。

- DB2 Query Manager for iSeries

  DB2 Query Manager for iSeries は、プロンプト方式の対話式インターフェースを提供します。これにより、ユーザーは、データの作成、データの追加、データの保守、およびデータベースに関する報告書の作成を行うことができます。Query Manager は、DB2 UDB Query Manager and SQL Development Kit ライセンス・プログラムの一部です。詳細については、Query Manager ご使用の手引きを参照してください。

- SQL REXX インターフェース

  SQL REXX インターフェースを使用すると、REXX プロシージャーの中で SQL ステートメントを実行することができます。 REXX プロシージャー内での SQL ステートメントの使用に関する詳細については、*ホスト言語での SQL プログラミング*のREXX アプリケーションでの SQL ステートメントのコーディングを参照してください。

- SQL 呼び出しレベル・インターフェース

  DB2 UDB for iSeriesは、SQL 呼び出しレベル・インターフェースをサポートします。これにより、どの ILE 言語のユーザーも、システムが提供するサービス・プログラムへのプロシージャー呼び出しを介して直接 SQL 機能にアクセスすることができます。SQL 呼び出しレベル・インターフェースを使用すると、事前コンパイルを必要とせずに全 SQL 機能を実行することができます。このインターフェースは、SQL ステートメントの準備、SQL ステートメントの実行、データ

行の取り出し、および拡張機能 (カタログへのアクセスや、プログラム変数の出力列へのバインドなど) を実行するための標準セットのプロシージャー呼び出しです。

使用可能なすべての機能の詳細とそれらの構文については、SQL 呼び出しレベル・インターフェースを参照してください。

- QSQPRCED API

  このアプリケーション・プログラム・インターフェース (API) は、拡張動的 SQL 関数を提供します。 SQL ステートメントは、SQL パッケージにしてから、この API を使用して実行することができます。この API によってパッケージにされたステートメントは、パッケージまたはステートメントが明示的に除去されるまで存続します。QSQPRCED は、OS/400 ライセンス・プログラムの一部です。QSQPRCED API についての詳細は、iSeries Information Center のプログラミング・セクションの中の QSQPRCEDを参照してください。API の一般情報については、iSeries Information Center の中の OS/400 API を参照してください。

- QSQCHKS API

  この API は、SQL ステートメントの構文を検査します。QSQCHKS は、OS/400 ライセンス・プログラムの一部です。QSQCHKS API についての詳細は、iSeries Information Center のプログラミング・セクションの中の QSQCHKS を参照してください。API の一般情報については、iSeries Information Center の中の OS/400 API を参照してください。

- DB2 マルチ・システム

  オペレーティング・システムのこの機能を使用すると、複数のサーバー間でデータを分散させることができます。DB2 マルチ・システムについての詳細は、DB2 マルチ・システムを参照してください。

- DB2 SMP

  オペレーティング・システムのこの機能により、照会最適化プログラムに、データを取り出すための追加の方式 (並列処理を含む) が提供されます。マルチプロセス (SMP) は、メモリーおよびディスク資源を共用する複数のプロセッサー (CPU および入出力処理機構) が 1 つの終了結果を得るために同時に作動する、1 つのシステムで実施される並列処理の形式です。この並列処理とは、データベース・マネージャーが 1 回の照会で同時に 2 つ以上 (あるいはすべて) のシステム・プロセッサーを作動させることができるということを意味します。 並列処理の制御方法の詳細については、データベース・パフォーマンスおよび *Query 最適化*の並列処理の制御を参照してください。

詳細については、以下のセクションを参照してください。

- 『SQL リレーショナル・データベースとシステム用語』
- 4 ページの『SQL ステートメントのタイプ』
- 6 ページの『SQL 通信域 (SQLCA)』

SQL リレーショナル・データベースとシステム用語

データのリレーショナル・モデルでは、すべてのデータは表内に存在するものとして認識されます。DB2 UDB for iSeries オブジェクトは、システム・オブジェクトとして作成され、保守されます。 次の表に、システム用語と SQL リレーショナル・データベース用語の間の関係を示します。 データベース・プログラミングの詳

細については、データベース・プログラミングを参照してください。

表 1. システム用語と SQL 用語の関係

システム用語	SQL 用語
ライブラリー。関連のあるオブジェクトをグループ化し、ユーザーがオブジェクトを名前で見つけることができるようにします。	**スキーマ**。ライブラリー、ジャーナル、ジャーナル・レシーバー、SQL カタログ、およびオプションであるデータ・ディクショナリーから成ります。スキーマは関連のあるオブジェクトをグループ化し、ユーザーがオブジェクトを名前で見つけることができるようにします。
物理ファイル。レコードのセット。	**表**。列と行のセット。
レコード。フィールドのセット。	**行**。一連の行を含む表の水平部分。
フィールド。1 つのデータ・タイプの関連情報の 1 つまたは複数の文字。	**列**。1 データ・タイプの表の垂直部分。
論理ファイル。1 つまたは複数の物理ファイルのレコードおよびフィールドのサブセット。	**視点**。1 つまたは複数の表の列と行のサブセット。
SQL パッケージ。SQL ステートメントを実行するために使用されるオブジェクト・タイプ。	**パッケージ**。SQL ステートメントを実行するために使用されるオブジェクト・タイプ。
ユーザー・プロファイル	**権限名または権限 ID**

以下も参照してください。

- 『SQL およびシステム命名規則』

SQL およびシステム命名規則

DB2 UDB for iSeries プログラミングで使用できる命名規則には、システム (*SYS) と SQL (*SQL) の 2 つがあります。使用される命名規則は、ファイルおよび表名の修飾方式と、対話式 SQL 画面で使用される用語に影響を及ぼします。使用される命名規則は、SQL コマンドのパラメーターによって選択されるか、あるいは REXX の場合は、SET OPTION ステートメントを介して選択されます。 詳細は、「SQL 解説書」にある『修飾されていないオブジェクト名の修飾』を参照してください。

システム命名規則 (*SYS): システム命名規則では、表、および SQL ステートメント内のその他のオブジェクトは、次の形式でスキーマ名によって修飾されます。

```
schema/table
```

SQL 命名規則 (*SQL): SQL 命名規則では、表、および SQL ステートメント内のその他のオブジェクトは、次の形式でスキーマ名によって修飾されます。

```
schema.table
```

SQL ステートメントのタイプ

SQL ステートメントには、以下の 4 つの基本タイプがあります。

- データ定義言語 (DDL) ステートメント
- データ操作言語 (DML) ステートメント
- 動的 SQL ステートメント
- その他のステートメント

SQL ステートメントは、SQL によって作成されたオブジェクトのほかに、外部記述物理ファイルと単一様式論理ファイル (それらが SQL スキーマに置かれているかどうかに関係なく) を操作することができます。 プログラム記述ファイルの IDDU ディクショナリー定義は参照の対象となりません。 プログラム記述ファイルは、1 つの列が入った表の形になります。

SQL DDL ステートメント	**SQL DML ステートメント**
ALTER TABLE	CLOSE
COMMENT ON	COMMIT
CREATE ALIAS	DECLARE CURSOR
CREATE DISTINCT TYPE	DELETE
CREATE FUNCTION	FETCH
CREATE INDEX	INSERT
CREATE PROCEDURE	LOCK TABLE
CREATE SCHEMA	OPEN
CREATE TABLE	RELEASE SAVEPOINT
CREATE TRIGGER	ROLLBACK
CREATE VIEW	SAVEPOINT
DECLARE GLOBAL TEMPORARY TABLE	SELECT INTO
DROP ALIAS	SET 変数
DROP DISTINCT TYPE	UPDATE
DROP FUNCTION	VALUES INTO
DROP INDEX	
DROP PACKAGE	
DROP PROCEDURE	
DROP SCHEMA	
DROP TABLE	
DROP TRIGGER	
DROP VIEW	
GRANT DISTINCT TYPE	
GRANT FUNCTION	
GRANT PACKAGE	
GRANT PROCEDURE	
GRANT TABLE	
LABEL ON	
RENAME	
REVOKE DISTINCT TYPE	
REVOKE FUNCTION	
REVOKE PACKAGE	
REVOKE PROCEDURE	
REVOKE TABLE	

動的 SQL ステートメント	その他のステートメント
DESCRIBE	BEGIN DECLARE SECTION
EXECUTE	CALL
EXECUTE IMMEDIATE	CONNECT
PREPARE	DECLARE PROCEDURE
	DECLARE STATEMENT
	DECLARE VARIABLE
	DESCRIBE TABLE
	DISCONNECT
	END DECLARE SECTION
	FREE LOCATOR
	HOLD LOCATOR
	INCLUDE
	RELEASE
	SET CONNECTION
	SET OPTION
	SET PATH
	SET RESULT SETS
	SET SCHEMA
	SET TRANSACTION
	WHENEVER

SQL DDL ステートメントは、51 ページの『第 4 章 データ定義言語 (DDL)』で説明されています。 SQL DML ステートメントは、67 ページの『第 5 章 SELECT ステートメントを使用したデータの検索』 および 101 ページの『第 6 章 SQL 挿入、更新、および削除』 で説明されています。これらのステートメントに関する詳細な説明は、SQL 解説書にあります。

SQL 通信域 (SQLCA)

SQLCA は、各 SQL ステートメントの実行の終了時に更新される変数のセットです。実行可能 SQL ステートメントを持っているプログラムは、SQLCA を 1 つだけ提供する必要があります (代わりに、スタンドアロン SQLCODE またはスタンドアロン SQLSTATE 変数が使用されていない限り)。詳しくは、iSeries Information Center の中の *SQL 解説書* の SQL 通信域を参照してください。

SQL オブジェクト

SQL オブジェクトとは、スキーマ、データ・ディクショナリー、ジャーナル、カタログ、表、別名、視点、索引、制約、トリガー、ストアード・プロシージャー、ユーザー定義関数、ユーザー定義タイプ、および、SQL パッケージのことです。SQL は、これらのオブジェクトをシステム・オブジェクトとして作成し、管理します。これらのオブジェクトについて、以下に、簡単に説明します。

- 8 ページの『別名』
- 8 ページの『視点』
- 8 ページの『索引』
- 9 ページの『制約』
- 9 ページの『トリガー』
- 10 ページの『ストアード・プロシージャー』
- 10 ページの『ユーザー定義関数』
- 10 ページの『ユーザー定義タイプ』
- 10 ページの『SQL パッケージ』

スキーマ

スキーマとは、SQL オブジェクトを論理的にグループ化したものです。**スキーマ**は、ライブラリー、ジャーナル、ジャーナル・レシーバー、カタログ、および、オプションとして、データ・ディクショナリーから構成されます。表、視点、およびシステム・オブジェクト (プログラムなど) は、どのシステム・ライブラリーにも作成、移動、あるいは復元することができます。SQL スキーマにデータ・ディクショナリーが入っていない場合は、すべてのシステム・ファイルを SQL スキーマ内に作成または移動することができます。SQL スキーマにデータ・ディクショナリーが入っている場合には、以下のようになります。

- 1 つのメンバーから成るソース物理ファイルまたは非ソース物理ファイルは、SQL スキーマ内に作成、移動、または復元することができます。
- 論理ファイルは、データ・ディクショナリーで記述できないため、SQL スキーマに置くことはできません。

ユーザーは多数のスキーマを作成し、所有することができます。 スキーマ の同義語としてコレクション という用語が使われる場合があります。

データ・ディクショナリー

スキーマがバージョン 3 リリース 1 より前に作成された場合、あるいは CREATE SCHEMA ステートメントで WITH DATA DICTIONARY 文節が指定された場合には、スキーマにデータ・ディクショナリーが入ります。**データ・ディクショナリー**とは、オブジェクト定義が入っている一連の表のことです。SQL でディクショナリーが作成されると、そのディクショナリーはシステムによって自動的に保守されます。ユーザーは、OS/400 プログラムの一部である対話式データ定義ユーティリティー (IDDU) を使用することによって、データ・ディクショナリーを処理することができます。IDDU について詳しくは、IDDU Use  を参照してください。

ジャーナルおよびジャーナル・レシーバー

ジャーナルおよび**ジャーナル・レシーバー**は、データベースの表と視点への変更を記録するために使用されます。 その後ジャーナルおよびジャーナル・レシーバーを使用して SQL COMMIT、ROLLBACK、SAVEPOINT、および RELEASE SAVEPOINT ステートメントの処理が行われます。 ジャーナルおよびジャーナル・レシーバーは、監査証跡あるいは順方向または逆方向回復のためにも使用できます。 ジャーナル処理について詳しくは、ジャーナル処理のトピックを参照してください。コミットメント制御について詳しくは、コミットメント制御のトピックを参照してください。

カタログ

SQL **カタログ**は、表、視点、索引、パッケージ、プロシージャー、関数、ファイル、トリガー、および制約を記述する一連の表および視点から構成されます。この情報は、ライブラリー QSYS および QSYS2 内の一連の相互参照表の中に入れられます。各 SQL スキーマには、スキーマ内の表、視点、索引、パッケージ、ファイル、および制約についての情報が入っている、カタログ表に基づいて作成された一連の視点があります。

カタログは、スキーマを作成するときに自動的に作成されます。カタログを除去したり、明示的に変更したりすることはできません。

SQL カタログについて詳しくは、SQL 解説書の中のカタログのトピックを参照してください。

表、行、および列

表は、**行**と**列**から構成されるデータの 2 次元の配列です。行は、1 つまたは複数の列を含む横方向の構成部分です。列は、1 つのデータ・タイプのデータの 1 つまたは複数の行を含む縦方向の構成部分です。1 つの列に含まれるデータはすべて同一タイプでなくてはなりません。SQL の表は、キー付きまたはキーなしの物理ファイルです。データ・タイプの説明については、SQL 解説書の中の、データ・タイプのトピックを参照してください。

表内のデータは、サーバー間に分散させることができます。分散表の詳細については、DB2 マルチシステムを参照してください。

別名

別名とは、表または視点の代替名のことです。既存の表または視点を参照できる場合に、別名を使用して表や視点を参照することができます。さらに、別名を表メンバーと結合することができます。別名について詳しくは、SQL 解説書の中の別名のトピックを参照してください。

視点

視点は、アプリケーション・プログラムにとっては表と同じように見えます。ただし、視点にはデータがありません。視点は 1 つまたは複数の表に基づいて作成されます。 1 つの視点には、特定の表のすべての列またはそれらのサブセットを入れることができ、また、特定の表のすべての行またはそれらのサブセットを入れることができます。視点内での列の配置は、それらの列が入っている元の表での配置と異なるものにすることができます。SQL の視点は、特殊な形式のキーなし論理ファイルです。

視点について詳しくは、iSeries Information Center の中の *SQL 解説書*のビューを参照してください。

索引

SQL の**索引**は、表の列のデータを昇順または降順で論理的に配列したサブセットです。各索引には個別の配列が含まれます。これらの配列は、順序付け (ORDER BY 文節)、グループ化 (GROUP BY 文節)、および結合のために使用されます。SQL の索引はキー付き論理ファイルです。

索引は、データ検索を迅速にするためにシステムによって使用されます。索引の作成はオプションです。索引はいくつでも作成できます。索引の作成または除去はいつでも可能です。索引はシステムで自動的に保守されます。しかし、索引はシステムによって保守されるので、索引の数が多いと、表を変更するアプリケーションのパフォーマンスに悪影響が及ぶ可能性があります。

コーディングに有効な索引について詳しくは、iSeries Information Center の中の、データベース・パフォーマンスおよび *Query 最適化* のラージ・テーブルへのアクセスをスピードアップするための索引の使用のトピックを参照してください。

制約

制約は、データベース・マネージャーによって実施される規則です。DB2 UDB for iSeries は、以下の制約をサポートします。

- 固有制約

  固有制約は、キーの値が固有である場合にのみその値が有効となる規則です。固有制約は、CREATE TABLE および ALTER TABLE ステートメントを用いて作成することができます。 CREATE INDEX では、固有性を保証する固有索引を作成することができますが、そのような索引は制約ではありません。

  固有制約は、INSERT および UPDATE ステートメントの実行時に実施されます。 PRIMARY KEY 制約は UNIQUE 制約の形式の 1 つです。 違いは、PRIMARY KEY にはヌル値可能列を入れられない点です。

- 参照制約

  参照制約は、外部キーの値が次の場合に限り有効となる規則です。

  – 親キーの値として存在するか、または

  – 外部キーの一部のコンポーネントがヌルである場合。

  参照制約は、INSERT、UPDATE、および DELETE ステートメントの実行時に実施されます。

- 検査制約

  検査制約は、列または列のグループの中で使用できる値を制限する規則です。検査制約は、CREATE TABLE および ALTER TABLE ステートメントを用いて追加することができます。検査制約は、INSERT および UPDATE ステートメントの実行時に実施されます。制約を満たすには、挿入または更新されるデータの各行が指定された条件を TRUE または未知 (ヌル値のため) のいずれかにしなければなりません。

制約について詳しくは、『第 10 章 データ保全性』を参照してください。

トリガー

トリガーは、指定されたイベントが指定された基礎となる表に起こるたびに自動的に実行される一連のアクションです。イベントとしては、挿入、更新、削除、または読み取り操作が可能です。 トリガーは、イベントの前後どちらでも実行することができます。 DB2 UDB for iSeries は SQL 挿入トリガー、更新トリガー、削除トリガー、および外部トリガーをサポートします。トリガーについて詳しくは、本書の『第 10 章 データ保全性』を参照するか、データベース・プログラミング の中の、データベース内での自動イベントのトリガーのトピックを参照してください。

ストアード・プロシージャー

ストアード・プロシージャーとは、SQL CALL ステートメントを使用して呼び出すことができるプログラムのことです。DB2 UDB for iSeries は、外部ストアード・プロシージャーおよび SQL プロシージャーをサポートします。外部ストアード・プロシージャーは、どのシステム・プログラムまたは REXX プロシージャーであっても構いません。ただし、システム/36 のプログラムまたはプロシージャー、あるいは、サービス・プログラムであってはなりません。SQL プロシージャーは、全体が SQL で定義され、(SQL 制御ステートメントを含む) SQL ステートメントを含めることができます。ストアード・プロシージャーについて詳しくは、本書の『第 11 章 ストアード・プロシージャー』を参照してください。

ユーザー定義関数

ユーザー定義関数とは、組み込み関数と同じように、呼び出すことができるプログラムのことです。 DB2 UDB for iSeries は、外部関数、SQL 関数、およびソース関数をサポートします。 外部関数は、どのシステム ILE プログラムでもサービス・プログラムでも構いません。 SQL 関数は、全体が SQL で定義され、(SQL 制御ステートメントを含む) SQL ステートメントを含めることができます。 ソース関数は、組み込み関数または既存のユーザー定義関数の上に作成することができます。スカラー関数または表関数を、SQL 関数または外部関数と同じように作成することができます。ユーザー定義関数について詳しくは、 251 ページの『第 13 章 ユーザー定義関数 (UDF) の作成』を参照してください。

ユーザー定義タイプ

ユーザー定義タイプは、データベース管理システムによって提供されるデータ・タイプとは無関係に定義できる、特殊なデータ・タイプです。特殊なデータ・タイプは、既存のデータベース・タイプに対して 1 対 1 でマップされます。ユーザー定義タイプについて詳しくは、 234 ページの『ユーザー定義の特殊タイプ (UDT)』を参照してください。

SQL パッケージ

SQL パッケージとは、アプリケーション・プログラム内の SQL ステートメントが遠隔リレーショナル・データベース管理システム (DBMS) にバインドされるときに作成される制御構造を含むオブジェクトです。DBMS は、制御構造を使用して、アプリケーション・プログラムの実行中に出される SQL ステートメントを処理します。

SQL パッケージは、SQL の作成 (CRTSQLxxx) コマンドでリレーショナル・データベース名 (RDB パラメーター) が指定され、プログラム・オブジェクトが作成されるときに作成されます。パッケージは、CRTSQLPKG コマンドを使用して作成することもできます。パッケージおよび分散リレーショナル・データベース機能について詳しくは、『第 21 章 分散リレーショナル・データベース機能』を参照してください。

SQL パッケージは、QSQPRCED API を使用して作成することもできます。本書において「SQL パッケージ」という場合は、分散型プログラム SQL パッケージのみを意味します。QSQPRCED は SQL パッケージを使用して拡張動的 SQL サポート

を提供します。 QSQPRCED について詳しくは、iSeries Information Center の OS/400 API セクションの QSQPRCED のトピックを参照してください。

注: このコマンドの xxx は、ホスト言語標識 (ILE C 言語の場合は CI、ILE C++ for iSeries 言語の場合は CPPI、COBOL for iSeries 言語の場合は CBL、ILE COBOL言語の場合は CBLI、iSeries PL/I 言語の場合は PLI、RPG for iSeries 言語の場合は RPG、ILE RPG 言語の場合は RPGI) を表します。

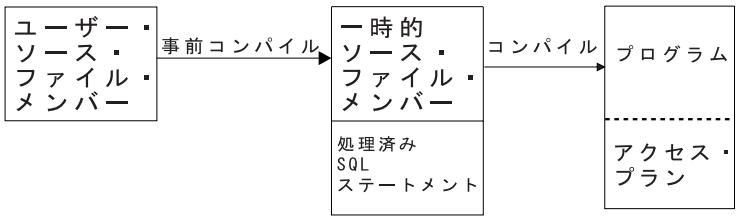
アプリケーション・プログラム・オブジェクト

DB2 UDB for iSeries アプリケーション・プログラムを作成するプロセスでは、いくつかのオブジェクトが作成されます。このセクションでは、DB2 UDB for iSeries アプリケーションの作成のプロセスについて簡単に説明します。DB2 UDB for iSeries は、ILE 以外の事前コンパイラーおよび ILE 事前コンパイラーの両方をサポートします。アプリケーション・プログラムは、分散型でも非分散型でも構いません。DB2 UDB for iSeries アプリケーション・プログラムの作成の詳細については、*ホスト言語での SQL プログラミング* の、SQL ステートメントでのプログラムの準備および実行 に記載されています。

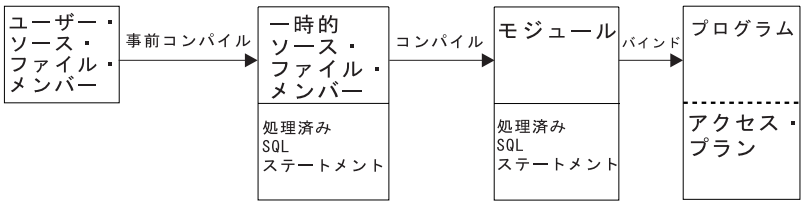
DB2 UDB for iSeries では、以下のオブジェクトを管理することが必要です。

- 元のソース
- ILE プログラムのモジュール・オブジェクト (任意)
- プログラムまたはサービス・プログラム
- 分散型プログラム用の SQL パッケージ

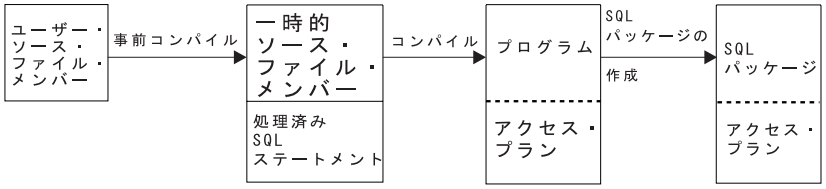
非分散型の非 ILE DB2 UDB for iSeries プログラムでは、管理しなければならないのは、元のソースと結果のプログラムだけです。以下に、非分散型の非 ILE DB2 UDB for iSeries プログラムの事前コンパイルおよびコンパイル・プロセスで生じる、必要なオブジェクトとステップを示します。



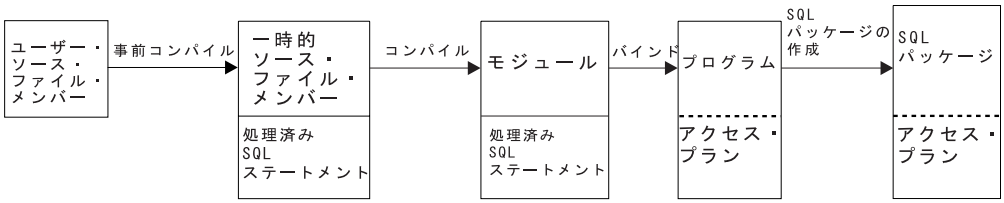
非分散型の ILE DB2 UDB for iSeries プログラムでは、元のソース、モジュール、および結果のプログラムまたはサービス・プログラムを管理する必要があります。以下に、事前コンパイル・コマンドで OBJTYPE(*PGM) が指定された場合に、非分散型の ILE DB2 UDB for iSeries プログラムの事前コンパイルおよびコンパイル・プロセスで生じる、必要なオブジェクトとステップを示します。



分散型の非 ILE DB2 UDB for iSeries プログラムでは、元のソース、結果のプログラム、および結果のパッケージを管理しなければなりません。以下に、分散型の非 ILE DB2 UDB for iSeries プログラムの事前コンパイルおよびコンパイル・プロセスで生じるオブジェクトとステップを示します。



分散型の ILE DB2 UDB for iSeries プログラムでは、元のソース、モジュール・オブジェクト、結果のプログラムまたはサービス・プログラム、および結果のパッケージを管理しなければなりません。 SQL パッケージは、分散型の ILE プログラムまたはサービス・プログラム内の各分散モジュールについて作成することができます。以下に、分散型の ILE DB2 UDB for iSeries プログラムの事前コンパイルおよびコンパイル・プロセスで生じるオブジェクトとステップを示します。



注: DB2 UDB for iSeries 分散プログラム・オブジェクトに関連するアクセス・プランは、プログラムがローカルに実行されるまで作成されません。

詳細については、以下のセクションを参照してください。

- 『ユーザー・ソース・ファイル・メンバー』
- 13 ページの『出力ソース・ファイル・メンバー』
- 13 ページの『プログラム』
- 14 ページの『モジュール』
- 14 ページの『サービス・プログラム』

ユーザー・ソース・ファイル・メンバー

ソース・ファイル・メンバーには、プログラマーが指定したアプリケーション言語および SQL ステートメントが含まれます。ソース・ファイル・メンバーの作成お

よび保守は、IBM WebSphere Development Studio for iSeries ライセンス・プログラムの一部である、原始ステートメント入力ユーティリティー (SEU) を使用して行うことができます。

出力ソース・ファイル・メンバー

SQL 事前コンパイルにより、出力ソース・ファイル・メンバーが作成されます。デフォルトにより、事前コンパイル・プロセスが一時的ソース・ファイル QSQLTxxxxx を QTEMP に作成するか、ユーザーが、事前コンパイル・コマンドで、永続ファイル名としてその出力ソース・ファイルを指定することができます。事前コンパイル・プロセスで QTEMP ライブラリーを使用する場合、ジョブが完了するとシステムは自動的にファイルを削除します。 プログラム名と同じ名前のメンバーが出力ソース・ファイルに追加されます。 このメンバーには、以下の項目が含まれています。

- SQL 実行時サポートへの呼び出し (これは、組み込み SQL ステートメントに代わるものです)
- 解析され構文検査された SQL ステートメント

省略時解釈により、事前コンパイラーはホスト言語コンパイラーを呼び出します。事前コンパイラーについて詳しくは、*ホスト言語での SQL プログラミング* の SQL ステートメントでのプログラムの準備および実行を参照してください。

プログラム

プログラムとは、コンパイル・プロセス (非 ILE コンパイルの場合) またはバインド・プロセス (ILE コンパイルの場合) の結果として作成される、実行可能なオブジェクトです。

アクセス・プランとは、組み込み SQL ステートメントを最も効率的に実行するための方法を SQL に指示する、内部構造と情報のセットです。これはプログラムが正しく作成されたときにだけ作成されます。次のような SQL ステートメント用のアクセス・プランは、プログラムの作成時には作成されません。

- 見つけることができない表または視点を参照するステートメント
- ユーザーが許可されていない表または視点を参照するステートメント

このようなステートメント用のアクセス・プランは、プログラムの実行時に作成されます。その時点でも、表または視点が見つからないか、あるいはユーザーが認可されていない場合は、負の SQLCODE が返されます。アクセス・プランは、非分散型 SQL プログラムではプログラム・オブジェクトに、分散型 SQL プログラムでは SQL パッケージに保管され、維持管理されます。

SQL パッケージ

SQL パッケージには、分散型 SQL プログラムのアクセス・プランが含まれます。

SQL パッケージは、以下のいずれかの場合に作成されるオブジェクトです。

- 分散型 SQL プログラムが、CRTSQLxxx コマンドの RDB パラメーターを使用して正しく作成された場合。
- SQL パッケージの作成 (CRTSQLPKG) コマンドが実行された場合。

分散型 SQL プログラムが作成されるとき、SQL パッケージの名前と内部整合性トークンがプログラム内に保管されます。これらは、SQL パッケージを見つけたり、SQL パッケージがこのプログラムにとって正しいものであることを確認したりするために、実行時に使用されます。SQL パッケージの名前は分散型 SQL プログラムを実行する上で重要であるため、SQL パッケージに対して以下のことを行うことはできません。

- 移動
- 名前変更
- 複製
- 別のライブラリーへの復元

モジュール

モジュールは、CRTxxxMOD コマンド (あるいは、xxx に C、CBL、CPP、または RPG が入る CRTBNDxxx コマンド) を用いてソース・コードをコンパイルすることにより作成される、統合言語環境 (ILE) オブジェクトです。プログラム作成 (CRTPGM) コマンドを使用してモジュールをプログラムにバインドするときにだけ、そのモジュールを実行することができます。通常は、複数のモジュールをまとめてバインドしますが、1 つのモジュールそれ自体をバインドすることもできます。モジュールには、SQL ステートメントに関する情報が含まれます。ただし、SQL アクセス・プランは、モジュールがプログラムまたはサービス・プログラムにバインドされるまで作成されません。プログラム作成 (CRTPGM) について詳しくは、「コマンド言語」のトピックの中のプログラム作成 (CRTPGM) を参照してください。

サービス・プログラム

サービス・プログラムは、外部でサポートされる呼び出し可能ルーチン (関数またはプロシージャー) を別のオブジェクト内にパッケージする手段を提供する統合言語環境 (ILE) オブジェクトです。バインド済みプログラムおよび他のサービス・プログラムは、これらのルーチンのインポートを、サービス・プログラムによって提供されるエクスポートに解決することによって、これらのルーチンにアクセスすることができます。これらのサービスへの接続は、呼び出しプログラムの作成時に行われます。これにより、呼び出しプログラム内にコードを組み込まずに、これらのルーチンへの呼び出しパフォーマンスを改善することができます。

第 2 章 SQL 入門

この章では、対話式 SQL において SQL ステートメントを使用して、スキーマ、表 (テーブル)、および視点 (ビュー) を作成する方法ならびに処理方法について説明します。

この章で使用される各 SQL ステートメントの構文は、SQL 解説書で詳しく説明されています。より複雑な状況で SQL ステートメントおよび文節を使用する方法については、 51 ページの『第 4 章 データ定義言語 (DDL)』、『第 5 章 SELECT ステートメントを使用したデータの検索』および『第 6 章 SQL 挿入、更新、および削除』に記載されています。

この章の例では、SQL ステートメントの実行を示すために対話式 SQL インターフェースを使用しています。各 SQL インターフェースは、SQL ステートメントを使用して表、視点、および他のオブジェクトを定義するための手段、オブジェクトを更新するための手段、およびオブジェクトからデータを読み取るための手段を提供します。

詳細については、以下のトピックを参照してください。

- 『対話式 SQL の開始』
- 16 ページの『スキーマの作成』
- 16 ページの『表 (テーブル) の作成および使用』
- 19 ページの『LABEL ON ステートメントの使用』
- 20 ページの『表への情報の挿入』
- 23 ページの『1 つの表からの情報の取り出し』
- 26 ページの『複数の表からの情報の取り出し』
- 28 ページの『表内の情報の変更』
- 30 ページの『表からの情報の削除』
- 31 ページの『視点 (ビュー) の作成および使用』

注: コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

対話式 SQL の開始

対話式 SQL の使用を開始するには、次のように入力してください。

```
STRSQL NAMING(*SQL)
```

次に実行キーを押します。「SQL ステートメントの入力」画面が表示されたら、SQL ステートメントの入力を開始することができます。対話式 SQL と STRSQL コマンドについて詳しくは、『第 17 章 対話式 SQL の使用』を参照してください。

既存の対話式 SQL セッションを再利用している場合、必ず命名モードを「*SQL 命名 (SQL naming)*」にセットしてください。これは、「F13 (サービス)」パネルのオプション 1 (セッション属性の変更) で指定することができます。

スキーマの作成

スキーマは、表、視点、索引、およびパッケージを入れるための基本オブジェクトです。スキーマを作成することについて詳しくは、SQL CREATE SCHEMA ステートメントを参照してください。

注: スキーマ の同義語として*コレクション* という用語が使われる場合があります。

対話式 SQL を使用してスキーマを作成する例については、『例: スキーマの作成 (SAMPLECOLL)』を参照してください。

例: スキーマの作成 (SAMPLECOLL)

「SQL ステートメントの入力」画面で次の SQL ステートメントを入力して実行キーを押すと、SAMPLECOLL という名前のサンプル・スキーマを作成することができます。

```
                          SQL ステートメントの入力

SQL ステートメントを入力して、実行キーを押してください。
    現在の接続相手はリレーショナル・データベース SYSTEM1 である。
===> CREATE SCHEMA SAMPLECOLL_____________________________________________
     ____________________________________________________________________
     ____________________________________________________________________
     ____________________________________________________________________
                                                                      終わり
F3=終了    F4=プロンプト    F6=行の挿入    F9=コマンドの複写    F10=行のコピー
F12=取消し                  F13=サービス   F24=キーの続き
```

注: このステートメントの実行では、いくつかのオブジェクトが作成されるため、数秒を要します。

スキーマが正しく作成されたら、表、視点、および索引をそこに作成することができます。表、視点、および索引は、スキーマの代わりに、ライブラリーに作成することもできます。

表 (テーブル) の作成および使用

SQL CREATE TABLE ステートメントを使用して、表を作成することができます。CREATE TABLE ステートメントは、表の作成、表内の列の物理属性の定義、および表内で使用できる値を制限するための制約の定義に使用されます。

対話式 SQL を使用して表を作成する例については、17 ページの『例: 表 (INVENTORY_LIST) の作成』を参照してください。

表を作成するときは、ヌル値と省略時値の概念を理解しておく必要があります。ヌル値は、ある行の列値が存在しないことを示します。 ゼロまたはすべてブランクの値とは異なります。それは「未知」であることを意味しています。他のどのような

値ともまったく異なり、他のヌル値とさえも異なります。列にヌル値が認められない場合は、その列に省略時値またはユーザーが指定する値のいずれかを割り当てる必要があります。

省略時値は、表に行が追加され、ある列に値が指定されていない場合に、その列に割り当てられます。列について特定の省略時値が定義されていない場合には、システムの省略時値が使用されます。INSERT によって使用される省略時値の詳細については、101 ページの 『INSERT ステートメントを使用した行の挿入』を参照してください。

例: 表 (INVENTORY_LIST) の作成

ここでは、現在の在庫に関する情報を保守するための表を作成します。この表には、在庫として保管している品目、それらの原価、現在の手持ち数量、最後の注文日、および最後の注文量に関する情報が入っています。 品目番号は必須値となります。ヌル値であってはなりません。 品目名、在庫数量、および注文量には、ユーザーが指定した省略時値が入ります。最後の注文日と注文量は、ヌル値であっても構いません。

「SQL ステートメントの入力」画面で、CREATE TABLE と入力し、F4 (プロンプト) キーを押してください。次の画面が表示されます (入力域にまだ何も入っていない状態で)。

```
                    CREATE TABLE ステートメントの指定

情報を入力して、実行キーを押してください。

テーブル . . . . . .   INVENTORY_LIST______     名前
   コレクション . . . . .    SAMPLECOLL__        名前, リストは F4 キー

ヌル値:  1=NULL, 2=NOT NULL, 3=NOT NULL WITH DEFAULT

列                  FOR 列        タイプ          長さ   スケール NULL
ITEM_NUMBER_______  ____________  CHAR___________  6____  __       2
ITEM_NAME_________  ____________  VARCHAR________  20___  __       3
UNIT_COST_________  ____________  DECIMAL________  8____  2_       3
QUANTITY_ON_HAND__  ____________  SMALLINT_______  _____  __       1
LAST_ORDER_DATE___  ____________  DATE___________  _____  __       1
ORDER_QUANTITY____  ____________  SMALLINT_______  _____  __       1
__________________  ____________  _______________  _____  __       3
                                                                  終わり
  テーブルの制約事項  . . . . . .  N     Y=YES, N=NO
  分散テーブル  . . . . . .  N     Y=YES, N=NO

F3=終了    F4=プロンプト     F5=最新表示   F6=行の挿入    F10=行のコピー
F11=属性の続きの表示         F12=取り消し  F14=行の削除   F24=キーの続き
```

「テーブル」および「コレクション」プロンプトに、作成する表 (テーブル) の表名 (INVENTORY_LIST) とスキーマ名 (SAMPLECOLL) を入力してください。表用に定義する各列は、画面の下の部分にあるリストの項目によって示されています。各列について、列の名前、列のデータ・タイプ、その長さとスケール、およびヌル値属性を入力してください。

F11 キーを押すと、これらの列について指定できるその他の属性が表示されます。ここで、省略時値を指定することができます。

```
                        CREATE TABLE ステートメントの指定

情報を入力して、実行キーを押してください。

テーブル . . . . . .   INVENTORY_LIST______     名前
  コレクション . . . . .    SAMPLECOLL__        名前、リストは F4 キー

データ:  1=BIT, 2=SBCS, 3=MIXED, 4=CCSID

列                  データ 割り振り  CCSID    制約      省略時の値
ITEM NUMBER_______     _     _____   _____     N
ITEM NAME_________     _     _____   _____     N       '***UNKNOWN***'___
UNIT_COST_________     _     _____   _____     N
QUANTITY_ON_HAND__     _     _____   _____     N       NULL______________
LAST_ORDER_DATE___     _     _____   _____     N
ORDER_QUANTITY____     _     _____   _____     N       20________________
__________________     _     _____   _____     _
                                                                  終わり
  テーブルの制約事項  . . . . . .   N       Y=YES, N=NO
  分散テーブル  . . . . . .   N       Y=YES, N=NO

F3=終了    F4=プロンプト     F5=最新表示    F6=行の挿入     F10=行のコピー
F11=属性の続きの表示         F12=取り消し   F14=行の削除    F24=キーの続き
```

注: 列定義を入力する別の方法として、リスト内の列項目の 1 つにカーソルを置いて F4 (プロンプト) キーを押す方法があります。これにより、1 つの列を定義するための属性のすべてを表示する画面が表示されます。

すべての値を入力したら、実行キーを押して表を作成してください。表の作成が完了したことを示すメッセージと共に、「SQL ステートメントの入力」画面がもう一度表示されます。

この CREATE TABLE ステートメントを「SQL ステートメントの入力」画面で次のように直接入力することもできます。

```
CREATE TABLE SAMPLECOLL.INVENTORY_LIST
     (ITEM_NUMBER CHAR(6) NOT NULL,
      ITEM_NAME VARCHAR(20) NOT NULL WITH DEFAULT '***UNKNOWN***',
      UNIT_COST DECIMAL(8,2) NOT NULL WITH DEFAULT,
      QUANTITY_ON_HAND SMALLINT DEFAULT NULL,
      LAST_ORDER_DATE DATE,
      ORDER_QUANTITY SMALLINT DEFAULT 20)
```

供給者表 (SUPPLIERS) の作成

後述する例では、2 つ目の表も必要となります。この表には、在庫品目の供給者、それらの供給者の供給品目、および供給品目の原価に関する情報が入ります。この表を作成するには、「SQL ステートメントの入力」画面に直接入力するか、または対話式 SQL 画面を使用して定義を作成するために F4 (プロンプト) キーを押します。

```
CREATE TABLE SAMPLECOLL.SUPPLIERS
     (SUPPLIER_NUMBER CHAR(4) NOT NULL,
     ITEM_NUMBER CHAR(6) NOT NULL,
     SUPPLIER_COST DECIMAL(8,2))
```

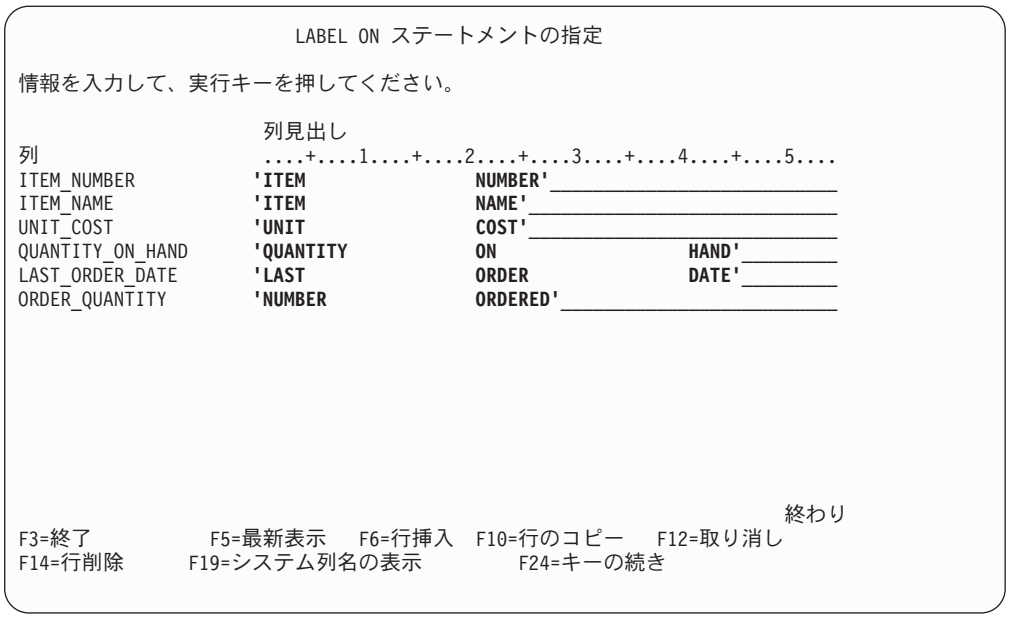
LABEL ON ステートメントの使用

通常、対話式 SQL で SELECT ステートメントの出力を表示する場合、列見出しとして列名を使用します。 LABEL ON ステートメントを使用すると、列名に対してより分かりやすいラベルを作成できます。ここに挙げる例は対話式 SQL で実行されるので、LABEL ON ステートメントを使用して列見出しを変更することにします。これらの列名は明確ですが、列見出しが名前の各部を 1 行で示すともっと読みやすくなります。またこれにより、データに関してより多くの列を 1 つの画面に表示できます。

列のラベルを変更するには、「SQL ステートメントの入力」画面で LABEL ON COLUMN と入力し、F4 (プロンプト) キーを押してください。次の画面が表示されます。

```
                        LABEL ON ステートメントの指定

選択項目を入力して、実行キーを押してください。

  ラベル・オン . .   2                        1= テーブルまたはビュー
                                              2= 列
                                              3= パッケージ
                                              4= 別名

  テーブルまたは . . . . INVENTORY_LIST_____   名前、リストは F4
  ビュー・コレクション .   SAMPLECOLL__        名前、リストは F4

  オプション . . .   1                        1= 列見出し
                                              2= テキスト









F3=終了    F4=プロンプト    F5=最新表示    F12=取り消し
F20=名前全体の表示          F21=ステートメントの表示
```

ラベルを追加したい列が入っている表 (テーブル) とスキーマの名前を入力し、実行キーを押してください。表内の各列を入力するよう指示する次の画面が表示されます。



各列に列見出しを入力してください。列見出しは 20 文字のセクションで定義されます。SELECT ステートメントの出力が表示されるときに、各セクションは別々の行に表示されます。列見出しの入力域の一番上にある罫線を使用すると、各見出しの間隔を簡単に正しくとることができます。見出しを入力したら、実行キーを押してください。

次のメッセージが、LABEL ON ステートメントが正しく完了したことを示します。

```
SAMPLECOLL の INVEN00001 の LABEL ON が完了した。
```

メッセージ内の表名は、この表のシステム表名で、ステートメントに実際に指定した名前ではありません。 DB2 UDB for iSeriesは、10 文字よりも長い名前の表について 2 つの名前を保持します。 システム表名について詳しくは、SQL 解説書の中の「CREATE TABLE ステートメント」を参照してください。

LABEL ON ステートメントは、「SQL ステートメントの入力」画面で次のように直接入力することもできます。

```
LABEL ON SAMPLECOLL.INVENTORY_LIST
(ITEM_NUMBER      IS 'ITEM              NUMBER',
ITEM_NAME         IS 'ITEM              NAME',
UNIT_COST         IS 'UNIT              COST',
QUANTITY_ON_HAND  IS 'QUANTITY          ON                  HAND',
LAST_ORDER_DATE   IS 'LAST              ORDER               DATE',
ORDER_QUANTITY    IS 'NUMBER            ORDERED')
```

表への情報の挿入

表を作成した後で、SQL INSERT ステートメントを使用して、表に情報 (データ) を挿入または追加することができます。

対話式 SQL を使用して表にデータを挿入する例については、 21 ページの『例: 表 (INVENTORY_LIST) への情報の挿入』を参照してください。

例: 表 (INVENTORY_LIST) への情報の挿入

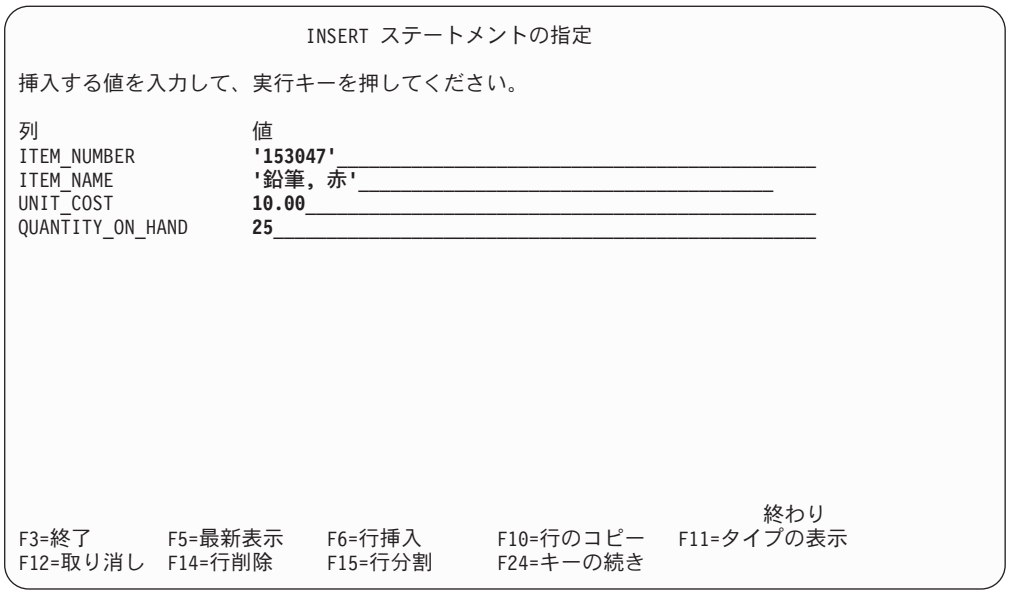
対話式 SQL で作業を行うには、「SQL ステートメントの入力」画面で、INSERT と入力し、F4 (プロンプト) を押してください。「INSERT ステートメントの指定」画面が表示されます。

```
                        INSERT ステートメントの指定

選択項目を入力して、実行キーを押してください。

  INTO テーブル. . . . . .    INVENTORY_LIST_____     名前, リストは F4 キー
    コレクション . . . . .      SAMPLECOLL__          名前, リストは F4 キー

  INTO の挿入列の選択 . . .   Y                       Y=YES, N=NO

  挿入方法  . . . . . .  .   1                       1=VALUES の入力
                                                     2=部分選択

選択項目を入力して、実行キーを押してください。

  WITH 分離レベル . . . . .   1               1=現行レベル, 2=NC (NONE)
                                              3=UR (CHG), 4=CS, 5=RS (ALL)
                                              6=RR



F3=終了    F4=プロンプト    F5=最新表示    F12=取り消し
F20=名前全体の表示          F21=ステートメントの表示
```

示されているように、入力フィールドに、表 (テーブル) 名とスキーマ名を入力してください。「*INTO の挿入欄*」プロンプトを YES に変更してください。実行キーを押すと、値を挿入したい列を選択できる画面が表示されます。

```
                        INSERT ステートメントの指定

選択するためには順序番号 (1 ～ 999) を入力して、実行キーを押してください。

SEQ  列                   タイプ              長さ   スケール
1__  ITEM_NUMBER          CHARACTER            6
2__  ITEM_NAME            VARCHAR             20
3__  UNIT_COST            DECIMAL              8    2
4__  QUANTITY_ON_HAND     SMALLINT             4
___  LAST_ORDER_DATE      DATE
___  ORDER_QUANTITY       SMALLINT             4

                                                                    終わり
F3=終了    F5=最新表示     F12=取り消し  F19=システム列名の表示
F20=名前全体の名前         F21=ステートメントの表示
```

この例では、挿入を行いたい列は 4 つだけです。その他の列には、省略時値が挿入されるようにします。この画面上の順序番号は、これらの列と値が INSERT ステートメントにリストされる順序を示しています。実行キーを押して、選択した列の値を入力できる画面を表示してください。



注: 入力リスト内の各列のデータ・タイプと長さを表示するには、F11 (タイプの表示) キーを押してください。これにより、値の挿入画面の別の視点が表示され、列定義に関する情報が提供されます。

すべての列について挿入する値を入力してから実行キーを押してください。これらの値の入っている行が表に追加されます。指定しなかった列の値には、省略時値が挿入されます。LAST_ORDER_DATE については、省略時値が提供されておらず、ヌル値が許可されるため、ヌル値になります。ORDER_QUANTITY については、20 になります。この値は、CREATE TABLE ステートメントで省略時値として指定されたものです。

この INSERT ステートメントは、「SQL ステートメントの入力」画面で次のように入力することができます。

```
INSERT INTO SAMPLECOLL.INVENTORY_LIST
                  (ITEM_NUMBER,
                   ITEM_NAME,
                   UNIT_COST,
                   QUANTITY_ON_HAND)
          VALUES('153047',
                 ' 鉛筆, 赤',
                 10.00,
                 25)
```

次の行を表に追加するには、「SQL ステートメントの入力」画面で F9 (検索) キーを押してください。これにより、前の INSERT ステートメントが入力域にコピーされます。前の INSERT ステートメントからの値を上書きするか、または F4 (プロンプト) キーを押し、対話式 SQL 画面を使用してデータを入力することができます。

引き続き INSERT ステートメントを使用して次の行を表に追加してください。次の図表に示されていない値は入力しないで、省略時値が使用されるようにしてください。INSERT ステートメントの列リストでは、値を挿入したい列名だけを指定してください。たとえば、3 番目の行を挿入するには、列名に ITEM_NUMBER と UNIT_COST だけを指定し、VALUES リスト内でこれらの列にこの 2 つの値だけ

を指定します。

ITEM_NUMBER	ITEM_NAME	UNIT_COST	QUANTITY_ON_HAND
153047	鉛筆、赤	10.00	25
229740	罫線付き便せん	1.50	120
544931		5.00	
303476	ペーパー・クリップ	2.00	100
559343	封筒、正式	3.00	500
291124	封筒、標準		
775298	いす、秘書用	225.00	6
073956	ペン、黒	20.00	25

次の行を SAMPLECOLL.SUPPLIERS 表に追加してください。

SUPPLIER_NUMBER	ITEM_NUMBER	SUPPLIER_COST
1234	153047	10.00
1234	229740	1.00
1234	303476	3.00
9988	153047	8.00
9988	559343	3.00
2424	153047	9.00
2424	303476	2.50
5546	775298	225.00
3366	303476	1.50
3366	073956	17.00

これでサンプル・スキーマには 2 つの表が含まれ、各表には複数のデータ行が入ります。

1 つの表からの情報の取り出し

すべての情報を表に入力したところで、この情報を再び見ることができるようにする必要があります。SQL では、SELECT ステートメントを使用してこれを行います。SELECT ステートメントは、すべての SQL ステートメントの中で最も複雑なステートメントです。このステートメントは、次の 3 つの主要な文節から構成されます。

1. SELECT 文節。これは、必要なデータを含む列を指定するためのものです。
2. FROM 文節。これは、必要なデータを含む列が入っている 1 つまたは複数の表を指定するためのものです。
3. WHERE 文節。これは、どのデータ行を取り出すのかを判別する条件を提供するためのものです。

この 3 つの主要な文節の他に、戻されるデータの最終的な形式に影響を与える文節がいくつかあります (これらについては、67 ページの 『第 5 章 SELECT ステートメントを使用したデータの検索』 および SQL 解説書で説明されています)。

INVENTORY_LIST 表に挿入した値を表示するには、SELECT と入力し、F4 (プロンプト) キーを押してください。次の画面が表示されます。

```
                          SELECT ステートメントの指定

SELECT ステートメント情報を入力してください。リストの表示は、F4 キーを押して
ください。

  FROM テーブル . . . . . .  SAMPLECOLL.INVENTORY_LIST_____________________
  SELECT 列  . . . . . . . .  *_____________________________________________
  WHERE 条件 . . . . . . . .  _____________________________________________
  GROUP BY 列 . . . . . . .  _____________________________________________
  HAVING 条件 . . . . . . .  _____________________________________________
  ORDER BY 列 . . . . . . .  _____________________________________________
  FOR UPDATE OF 列  . . . . .  _____________________________________________

                                                                   終わり
選択項目を入力して、実行キーを押してください。

  結果テーブル中の DISTINCT 行 . . . . . .  N    Y=YES, N=NO
  別の SELECT との UNION . . . . . .  N    Y=YES, N=NO
  追加オプションの指定 . . . . . .  N    Y=YES, N=NO



F3=終了         F4=プロンプト    F5=最新表示    F6=行挿入      F9=SUBQUERYの指定
F10=行のコピー  F12=取り消し     F14=行削除     F15=行分割     F24=キーの続き
```

画面上の「*FROM* テーブル」フィールドに表 (テーブル) 名を入力してください。表からすべての列を選択するには、画面上の「*SELECT* 列」フィールドに * を入力してください。実行キーを押すと、ステートメントが実行されて、表内のすべての列の全データが選択されます。次の出力が表示されます。

```
                              データの表示
                                          データの幅  . . . . . :     71
行の位置指定  . . . . .                      桁移動  . . . . . .
....+....1....+....2....+....3....+....4....+....5....+....6....+....7.
ITEM    ITEM                      UNIT   QUANTITY  LAST      NUMBER
NUMBER  NAME                      COST   ON        ORDER     ORDERED
                                         HAND      DATE
153047  鉛筆、赤                 10.00       25   -              20
229740  罫線付き便せん            1.50      120   -              20
544931  ***UNKNOWN***             5.00        -   -              20
303476  ペーパー・クリップ        2.00      100   -              20
559343  封筒、正式                3.00      500   -              20
291124  封筒、標準                 .00        -   -              20
775298  いす、秘書用            225.00        6   -              20
073956  ペン、黒                 20.00       25   -              20
********  データの終わり  ********

F3=終了      F12=取り消し      F19=左      F20=右      F21=分割
```

LABEL ON ステートメントを使用して定義した列見出しが表示されます。 3 番目の ITEM_NAME には、CREATE TABLE ステートメントで指定された省略時値が入っています。QUANTITY_ON_HAND 列では、値が挿入されなかった行がヌル値になっています。*LAST_ORDER_DATE* 列は、すべてヌル値になっています。これは、この列がどの INSERT ステートメントにも指定されておらず、この列に省略時値が入るように定義されていないためです。同様に、ORDER_QUANTITY 列では、すべての行に省略時値が入っています。

このステートメントは、「SQL ステートメントの入力」画面で次のように入力することができます。

```
SELECT *
  FROM SAMPLECOLL.INVENTORY_LIST
```

SELECT ステートメントによって戻される列の数を制限するには、表示したい列を指定する必要があります。戻される出力行の数を制限するには、WHERE 文節を使用します。原価が 10 ドルを超える品目だけを表示し、さらに ITEM_NUMBER、UNIT_COST、および ITEM_NAME 列の値だけが戻されるようにするには、SELECT と入力し、F4 (プロンプト) キーを押してください。「SELECT ステートメントの指定」画面が表示されます。

```
                        SELECT ステートメントの指定

SELECT ステートメント情報を入力してください。リストの表示は、F4 キーを押して
ください。

  FROM テーブル . . . . . .   SAMPLECOLL.INVENTORY_LIST_____________________
  SELECT 列  . . . . . .   ITEM_NUMBER, UNIT_COST, ITEM_NAME_____________
  WHERE 条件 . . . . . .   UNIT_COST > 10.00_____________________________
  GROUP BY 列 . . . . . .  _______________________________________________
  HAVING 条件 . . . . . .  _______________________________________________
  ORDER BY 列 . . . . . .  _______________________________________________
  FOR UPDATE OF 列 . . . .  _______________________________________________

                                                                  終わり
選択項目を入力して、実行キーを押してください。

  結果テーブル中の DISTINCT 行 . . . . . .  N    Y=YES, N=NO
  別の SELECT との UNION . . . . . .  N    Y=YES, N=NO
  追加オプションの指定 . . . . . .  N    Y=YES, N=NO



F3=終了         F4=プロンプト   F5=最新表示   F6=行挿入      F9=SUBQUERYの指定
F10=行のコピー  F12=取り消し     F14=行削除     F15=行分割     F24=キーの続き
```

「SELECT ステートメントの指定」画面の各プロンプトに最初に表示されているのは 1 行だけですが、F6 (行の挿入) キーを使用すると、画面上部の任意の入力域に行をさらに追加することができます。これは、「*SELECT 列*」リストにさらに列を入力したり、もっと長く複雑な WHERE 条件が必要な場合に使用することができます。

上記のとおり画面に入力を行ってください。実行キーを押すと、SELECT ステートメントが実行されます。次の出力が表示されます。

```
                              データの表示
                                          データの幅  . . . . . :     41
行の位置指定  . . . . .                        桁移動  . . . . . .
....+....1....+....2....+....3....+....4.
ITEM          UNIT   ITEM
NUMBER        COST   NAME
775298     225.00   いす、秘書用
073956      20.00   ペン、黒
********  データの終わり  ********

F3=終了      F12=取り消し      F19=左      F20=右      F21=分割
```

戻される行は、WHERE 文節で指定した条件に該当するデータ値が入っている行に限られます。さらに、戻されるデータ値は、SELECT 文節で明示的に指定した列からのデータ値だけです。明示的に指定されていない列のデータ値は戻されません。

このステートメントを「SQL ステートメントの入力」画面で入力する場合は、次のようになります。

```
SELECT ITEM_NUMBER, UNIT_COST, ITEM_NAME
  FROM SAMPLECOLL.INVENTORY_LIST
  WHERE UNIT_COST > 10.00
```

複数の表からの情報の取り出し

SQL では、複数の表に入っている列から情報を取り出すことができます。この操作は*結合* 操作と呼ばれます。 (結合操作の詳細については、85 ページの 『複数の表からのデータの結合』を参照してください。) SQL では、結合操作は、結合する表の名前を SELECT ステートメントの同一の FROM 文節に入れることによって指定されます。

たとえば、全供給者と、それらの供給品目の品目番号および品目名のリストを表示したいとします。品目名は、SUPPLIERS 表には入っておらず、INVENTORY_LIST 表に入っています。共通の列 ITEM_NUMBER を使用すると、これらの 3 つの列がすべて 1 つの表から取り出されたかのように、それらを表示することができます。

結合する複数の表に同じ列名が存在する場合は必ず、どの列が実際に参照されているのかを指定するために、列名を表名で修飾する必要があります。この SELECT ステートメントでは、列名 ITEM_NUMBER を両方の表で定義しているので、この列名を表名で修飾する必要があります。これらの列の名前が異なる場合は、混乱もないので修飾の必要もありません。

この結合を実行するには、次の SELECT ステートメントを使用することができます。「SQL ステートメントの入力」画面で直接入力するか、またはプロンプトを用いて入力してください。プロンプトを使用する場合は、両方の表名を「*FROM テーブル*」入力行に入力する必要があります。

```
SELECT SUPPLIER_NUMBER, SAMPLECOLL.INVENTORY_LIST.ITEM_NUMBER, ITEM_NAME
  FROM SAMPLECOLL.SUPPLIERS, SAMPLECOLL.INVENTORY_LIST
  WHERE SAMPLECOLL.SUPPLIERS.ITEM_NUMBER
                    = SAMPLECOLL.INVENTORY_LIST.ITEM_NUMBER
```

同じステートメントを入力するための別の方法として、相関名の使用があります。相関名は、ある表名に、ステートメントで使用するための別の名前を与えます。表名が同じときには相関名を使用しなければなりません。相関名は、FROM リストの各表名の後に指定することができます。上記のステートメントは次のように書き直すことができます。

```
SELECT SUPPLIER_NUMBER, Y.ITEM_NUMBER, ITEM_NAME
  FROM SAMPLECOLL.SUPPLIERS X, SAMPLECOLL.INVENTORY_LIST Y
  WHERE X.ITEM_NUMBER = Y.ITEM_NUMBER
```

この例では、SAMPLECOLL.SUPPLIERS に X の相関名が与えられ、SAMPLECOLL.INVENTORY_LIST に Y の相関名が与えられています。相関名 X および Y は、ITEM_NUMBER 列名を修飾するために使用されています。

列名および相関名について詳しくは、iSeries Information Center の中の *SQL 解説書* の「相関名」を参照してください。

この例を実行すると、次の出力が戻ります。

```
                                データの表示
                                                  データの幅 . . . . . :     45
行の位置指定  . . . . .                             桁移動  . . . . . .
....+....1....+....2....+....3....+....4....+
SUPPLIER_NUMBER  ITEM    ITEM
                 NUMBER  NAME
     1234        153047  鉛筆、赤
     1234        229740  罫線付き便せん
     1234        303476  ペーパー・クリップ
     9988        153047  鉛筆、赤
     9988        559343  封筒、正式
     2424        153047  鉛筆、赤
     2424        303476  ペーパー・クリップ
     5546        775298  いす、秘書用
     3366        303476  ペーパー・クリップ
     3366        073956  ペン、黒
********  データの終わり  ********

F3=終了      F12=取り消し      F19=左      F20=右      F21=分割
```

注: 照会において ORDER BY 文節が指定されなかったため、ユーザーの照会で戻される行の順序は異なる場合があります。

結果表内のデータ値は、INVENTORY_LIST と SUPPLIERS の 2 つの表に含まれるデータ値を組み合わせたものです。この結果表には、SUPPLIER 表からの供給者番号と、INVENTORY_LIST 表からの品目番号および品目名が入っています。SUPPLIER 表に入っていない品目番号は、結果表に表示されません。この結果は、SELECT ステートメントに ORDER BY 文節を指定しない限り、どのような順序になるか保証されません。この例では、SUPPLIER 表の列見出しを変更しなかったので、SUPPLIER_NUMBER 列名が列見出しとして使用されています。

次の例では、行の順序を保証するために ORDER BY を使用します。このステートメントは、まず結果表を SUPPLIER_NUMBER 列に従って配列します。SUPPLIER_NUMBER の値が同じ行は、ITEM_NUMBER に従って配列されます。

```
SELECT SUPPLIER_NUMBER, Y.ITEM_NUMBER, ITEM_NAME
  FROM SAMPLECOLL.SUPPLIERS X, SAMPLECOLL.INVENTORY_LIST Y
  WHERE X.ITEM_NUMBER = Y.ITEM_NUMBER
  ORDER BY SUPPLIER_NUMBER, Y.ITEM_NUMBER
```

上記のステートメントを実行すると、次の出力が生成されます。

```
                                データの表示
                                                  データの幅 . . . . . :     45
行の位置指定  . . . . .                             桁移動  . . . . . .
....+....1....+....2....+....3....+....4....+
SUPPLIER_NUMBER  ITEM    ITEM
                 NUMBER  NAME
     1234        153047  鉛筆、赤
     1234        229740  罫線付き便せん
     1234        303476  ペーパー・クリップ
     2424        153047  鉛筆、赤
     2424        303476  ペーパー・クリップ
     3366        073956  ペン、黒
     3366        303476  ペーパー・クリップ
     5546        775298  いす、秘書用
     9988        153047  鉛筆、赤
     9988        559343  封筒、正式
********  データの終わり  ********

F3=終了      F12=取り消し      F19=左      F20=右      F21=分割
```

表内の情報の変更

SQL UPDATE ステートメントを使用すると、表内の一部または全部の列のデータ値を変更することができます。

対話式 SQL を使用して表内の情報を変更する例については、『例: 表内の情報の変更』を参照してください。

1 回のステートメントの実行の間に変更される行の数を制限したい場合には、UPDATE ステートメントで WHERE 文節を使用します。詳しくは、 106 ページの『UPDATE ステートメントを使用した表内のデータの変更』を参照してください。WHERE 文節を指定しないと、指定した表のすべての行が変更されます。ただし、WHERE 文節を使用した場合、システムは、指定した条件を満たす行だけを変更します。詳細については、 69 ページの『WHERE 文節を使用した検索条件の指定』を参照してください。

例: 表内の情報の変更

対話式 SQL を使用して、たとえば、ペーパー・クリップの注文を今日追加する場合を考えてみましょう。品目番号 303476 の LAST_ORDER_DATE と ORDER_QUANTITY を更新するには、UPDATE と入力し、F4 (プロンプト) キーを押します。「UPDATE ステートメントの指定」画面が表示されます。

```
                        UPDATE ステートメントの指定

選択項目を入力して、実行キーを押してください。

  テーブル . . . . . .    INVENTORY_LIST_____   名前, リストは F4 キー
    コレクション . . . .      SAMPLECOLL__       名前, リストは F4 キー

  相関名 . . . . . .      ___________________   名前

F3=終了    F4=プロンプト    F5=最新表示    F12=取り消し
F20=名前全体の表示          F21=ステートメントの表示
```

表 (テーブル) 名とスキーマ名を入力したら、実行キーを押してください。表内の列のリストが入った画面が表示されます。

```
                          UPDATE ステートメントの指定

選択項目を入力して、実行キーを押してください。

   テーブル . . . . . .  INVENTORY_LIST_____    名前，リストは F4 キー
    コレクション  . . . .   SAMPLECOLL__        名前，リストは F4 キー

  相関名  . . . . . .  ____________________    名前


情報を入力して、実行キーを押してください。

列                    値
ITEM_NUMBER           ____________________________________________________
ITEM_NAME             ____________________________________________________
UNIT_COST             ____________________________________________________
QUANTITY_ON_HAND      ____________________________________________________
LAST_ORDER_DATE       CURRENT DATE________________________________________
ORDER_QUANTITY        50__________________________________________________

                                                                    終わり
F3=終了    F4=プロンプト      F5=最新表示    F6=行挿入       F10=行のコピー
F11=タイプの表示              F12=取り消し   F14=行削除      F24=キーの続き
```

値に CURRENT DATE を指定すると、選択されたすべての行の日付が今日の日付に変更されます。

この表について更新する値を入力したら、実行キーを押して、WHERE 条件を指定できる画面を表示してください。 WHERE 条件を指定しないと、表内のすべての行が前の画面からの値により更新されます。

```
                          UPDATE ステートメントの指定

WHERE 条件を入力して、実行キーを押してください。リストの表示は、F4 キーを押
してください。
  ITEM_NUMBER = '303476'_____________________________________________
  ____________________________________________________________________



                                                                    終わり
選択項目を入力して、実行キーを押してください。

  WITH 分離レベル  . . . . .  1              1=現行レベル，2=NC (NONE)
                                             3=UR (CHG)，4=CS，5=RS (ALL)
                                             6=RR




F3=終了          F4=プロンプト    F5=最新表示    F6=行挿入      F9=SUBQUERYの指定
F10=行のコピー   F12=取り消し     F14=行削除     F15=行分割     F24=キーの続き
```

条件を入力したら、実行キーを押して表の更新を実行してください。この機能が終了したことを示すメッセージが表示されます。

このステートメントを「SQL ステートメントの入力」画面で入力すると、次のようになります。

```
UPDATE SAMPLECOLL.INVENTORY_LIST
  SET LAST_ORDER_DATE = CURRENT DATE,
        ORDER_QUANTITY = 50
  WHERE ITEM_NUMBER = '303476'
```

SELECT ステートメント (SELECT * FROM SAMPLECOLL.INVENTORY_LIST) を実行して表からすべての行を取り出すと、結果は次のようになります。

```
                                データの表示
                                                 データの幅  . . . . . :     71
行の位置指定  . . . . .                                 桁移動  . . . . . .
....+....1....+....2....+....3....+....4....+....5....+....6....+....7.
ITEM    ITEM                          UNIT   QUANTITY  LAST      NUMBER
NUMBER  NAME                          COST   ON        ORDER     ORDERED
                                             HAND      DATE
153047  鉛筆、赤                     10.00       25   -              20
229740  罫線付き便せん                1.50      120   -              20
544931  ***UNKNOWN***                5.00        -   -              20
303476  ペーパー・クリップ            2.00      100   05/30/94       50
559343  封筒、正式                    3.00      500   -              20
291124  封筒、標準                     .00        -   -              20
775298  いす、秘書用                225.00        6   -              20
073956  ペン、黒                     20.00       25   -              20
********  データの終わり  ********
                                                                   終わり
F3=終了      F12=取り消し      F19=左      F20=右      F21=分割
```

ペーパー・クリップ の項目だけが変更されました。LAST_ORDER_DATE は現在の日付に変更されました。この日付は、必ず、更新が実行された日付になります。NUMBER_ORDERED は更新された値を示しています。

表からの情報の削除

SQL DELETE ステートメントを使用すると、表からデータを削除できます。行に入っている情報が不要になった場合は、それらの行全体を表から削除することができます。あるいは、DELETE ステートメントで WHERE 文節を使用すると、1 回のステートメントの実行で削除される行 (複数) を指定することができます。詳細については、111 ページの『DELETE ステートメントを使用した表からの行の除去』を参照してください。

対話式 SQL を使用して表内の情報を削除する例については、『例: 表 (INVENTORY_LIST) からの情報の削除』を参照してください。

例: 表 (INVENTORY_LIST) からの情報の削除

表内の QUANTITY_ON_HAND 列にヌル値が入っているすべての行を削除したい場合は、「SQL ステートメントの入力」画面で次のステートメントを入力することができます。

```
DELETE
  FROM SAMPLECOLL.INVENTORY_LIST
  WHERE QUANTITY_ON_HAND IS NULL
```

列にヌル値があるかどうかをチェックするには、IS NULL 比較を使用します。削除が完了してから別の SELECT ステートメントを実行すると、次の結果表が戻されます。

```
                                データの表示
                                                  データの幅  . . . . . :     71
行の位置指定  . . . . .                             桁移動  . . . . . .
....+....1....+....2....+....3....+....4....+....5....+....6....+....7.
ITEM    ITEM                        UNIT   QUANTITY  LAST      NUMBER
NUMBER  NAME                        COST   ON        ORDER     ORDERED
                                           HAND      DATE
153047  鉛筆、赤                   10.00        25   -              20
229740  罫線付き便せん              1.50       120   -              20
303476  ペーパー・クリップ          2.00       100   05/30/94       50
559343  封筒、正式                  3.00       500   -              20
775298  いす、秘書用              225.00         6   -              20
073956  ペン、黒                   20.00        25   -              20
********  データの終わり  ********
                                                                  終わり
F3=終了      F12=取り消し      F19=左      F20=右      F21=分割
```

QUANTITY_ON_HAND にヌル値が入っている行が削除されました。

視点 (ビュー) の作成および使用

必要なすべての情報が 1 つの表にまとまって入っていない場合もあります。また、表に入っているデータの一部だけしかユーザーのアクセスを認めたくない場合もあります。視点を使用すると、表をいくつかのサブセットに分けて、必要なデータだけを取り扱うことができます。視点は複雑さを軽減すると同時に、アクセスを制限します。

SQL CREATE VIEW ステートメントを使用して、視点を作成することができます。 CREATE VIEW ステートメントを使用すると、ある表に基づいて視点を定義することは、必要な列と行だけを含む新規の表を作成するようなものです。 アプリケーションが視点を使用するとき、その視点に含まれていない表の行または列にアクセスすることはできません。 ただし、SQL WITH CHECK OPTION が使用されない場合は、選択基準に一致しない行が視点を介して挿入される可能性があります。 WITH CHECK OPTION の使用について詳しくは、『第 10 章 データ保全性』を参照してください。

対話式 SQL を使用して視点を作成する例については、以下の例を参照してください。

- 32 ページの『例: 1 つの表に基づく視点の作成』
- 32 ページの『例: 複数の表からのデータを組み合わせた視点の作成』

視点を作成するには、ユーザーに視点の基礎となる表または物理ファイルに対する正当な権限がなければなりません。必要な権限のリストについては、*SQL 参照*内の CREATE VIEW ステートメントを参照してください。

視点定義で列名を指定しないと、列名は視点の基礎となる表のものと同じになります。

視点に含まれる列または行の数が表と同じでなくても、視点を介して表に変更を加えることができます。INSERT の場合、視点にない表の列には、省略時値が入っていなければなりません。

視点は、データが 1 つまたは複数の表に完全に依存していても、それが表であるかのように使用することができます。視点自体はデータを持たないため、データのた

めの記憶域を必要としません。視点は記憶域に存在する表から作られるので、視点のデータを更新するときは、実際は表にあるデータを更新することになります。したがって、視点は、その基礎となる表が更新されると、自動的に更新されます。

追加情報については、 60 ページの『視点の作成および使用』を参照してください。

例: 1 つの表に基づく視点の作成

次の例は、1 つの表に基づいて視点を作成する方法を示しています。 この視点は INVENTORY_LIST 表に基づいて作成されます。 この表には 6 つの列がありますが、視点ではそのうちの 3 つの列 (ITEM_NUMBER、LAST_ORDER_DATE、および QUANTITY_ON_HAND) だけを使用します。 SELECT 文節で指定される列の順序は、それらの列が視点に置かれる順序になります。視点には、ここ 2 週間で注文された品目の行だけが入ります。 CREATE VIEW ステートメントは次のようになります。

```
CREATE VIEW SAMPLECOLL.RECENT_ORDERS AS
    SELECT ITEM_NUMBER, LAST_ORDER_DATE, QUANTITY_ON_HAND
         FROM SAMPLECOLL.INVENTORY_LIST
         WHERE LAST_ORDER_DATE > CURRENT DATE - 14 DAYS
```

上記の例では、視点名の後に列のリストが指定されていないので、視点の列の名前は表の列と同じになります。視点が作成されて入れられるスキーマは、視点の作成の基礎となる表と同じスキーマでなくても構いません。どのスキーマまたはライブラリーでも、使用することができます。以下の画面は、次の SQL ステートメントを実行した結果です。

```
SELECT * FROM SAMPLECOLL.RECENT_ORDERS
```

```
                              データの表示
                                                  データの幅 . . . . . :     26
行の位置指定  . . . . .                                桁移動  . . . . . .
....+....1....+....2....+.
ITEM    LAST      QUANTITY
NUMBER  ORDER     ON
        DATE      HAND
303476  05/30/94      100
********  データの終わり  ********
                                                                      終わり
F3=終了      F12=取り消し      F19=左      F20=右      F21=分割
```

視点によって選択された唯一の行は、現在の日付が入るように更新した行です。表内の他の行の日付には、まだヌル値が入っているので戻されません。

例: 複数の表からのデータを組み合わせた視点の作成

FROM 文節に複数の表の名前を指定することにより、複数の表からのデータを組み合わせた視点を作成することができます。次の例では、INVENTORY_LIST 表に *ITEM_NUMBER* という品目番号の列と、*UNIT_COST* という品目の原価の列が含まれています。これらを、SUPPLIERS 表の *ITEM_NUMBER* 列および *SUPPLIER_COST* 列と結合します。WHERE 文節を使用して、戻される行の数を制限します。 視点には、現在の単価よりも低い原価で品目を供給できる供給者の品目番号だけが入ります。

CREATE VIEW ステートメントは次のようになります。

```
CREATE VIEW SAMPLECOLL.LOWER_COST AS
  SELECT SUPPLIER_NUMBER, A.ITEM_NUMBER, UNIT_COST, SUPPLIER_COST
    FROM SAMPLECOLL.INVENTORY_LIST A, SAMPLECOLL.SUPPLIERS B
    WHERE A.ITEM_NUMBER = B.ITEM_NUMBER
    AND UNIT_COST > SUPPLIER_COST
```

次の SQL ステートメントを実行すると、結果は以下のようになります。

```
SELECT * FROM SAMPLECOLL.LOWER_COST
```

```
                                データの表示
                                                データの幅 . . . . . :      51
行の位置指定  . . . . .                            桁移動  . . . . . .
....+....1....+....2....+....3....+....4....+....5.
SUPPLIER_NUMBER  ITEM          UNIT   SUPPLIER_COST
                 NUMBER        COST
     9988        153047       10.00           8.00
     2424        153047       10.00           9.00
     1234        229740        1.50           1.00
     3366        303476        2.00           1.50
     3366        073956       20.00          17.00
********  データの終わり  ********
                                                                    終わり
F3=終了      F12=取り消し      F19=左      F20=右      F21=分割
```

注: 照会において ORDER BY 文節が指定されなかったため、ユーザーの照会で戻される行の順序は異なる場合があります。

この視点により表示される行は、単価よりも低い供給者原価が入っている行だけです。

対話式 SQL の使用に関する詳細については、 313 ページの『第 17 章 対話式 SQL の使用』を参照してください。

第 3 章 iSeries ナビゲーター・データベース入門

この章では、iSeries ナビゲーターを使用して、ライブラリー (スキーマまたは SQL コレクション)、表 (テーブル)、および視点 (ビュー) を作成する方法ならびに処理方法について説明します。iSeries ナビゲーター・データベースはグラフィカル・インターフェースで、よくご使用になる管理用のデータベース操作を実行するために使用することができます。iSeries ナビゲーターの操作のほとんどは構造化照会言語 (SQL) をベースにしたものですが、この操作を実行するのに、SQL を完全に理解している必要はありません。 295 ページの『第 16 章 iSeries ナビゲーターを使用した拡張データベース機能』には、iSeries ナビゲーターを使用する拡張データベース機能についての説明があります。この章の例では、iSeries ナビゲーターを使用して、一般的なデータベース・タスクの実行を説明します。作成されるオブジェクトは、15 ページの『第 2 章 SQL 入門』で対話式 SQL を使用する例で作成されたオブジェクトと同じものです。

詳細については、以下のトピックを参照してください。

- 『iSeries ナビゲーターの開始』
- 36 ページの『iSeries ナビゲーターを使用してのライブラリーの作成』
- 37 ページの『iSeries ナビゲーターで表示されるライブラリーのリストの編集』
- 38 ページの『iSeries ナビゲーターを使用した表の作成と使用』
- 39 ページの『iSeries ナビゲーターを使用した表の列の定義』
- 41 ページの『iSeries ナビゲーターを使用した列定義のコピー』
- 41 ページの『iSeries ナビゲーターを使用した表への情報の挿入』
- 42 ページの『iSeries ナビゲーターを使用した表の内容の表示』
- 44 ページの『iSeries ナビゲーターを使用した表のコピーと移動』
- 45 ページの『iSeries ナビゲーターを使用した視点の作成および使用』
- 49 ページの『iSeries ナビゲーターを使用したデータベース・オブジェクトの削除』

注: コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

iSeries ナビゲーターの開始

以下の例で使用するためにiSeries ナビゲーターを開始するには、次のようにします。

1. 「**iSeries ナビゲーター**」アイコンをダブルクリックする。
2. 使用したいシステムを展開する。

iSeries ナビゲーターのセットアップについて詳しくは、iSeries ナビゲーター入門を参照してください。

iSeries ナビゲーターを使用してのライブラリーの作成

ライブラリーはデータベース構造体で、そこには、ユーザーの表、視点、およびその他のオブジェクト・タイプが入っています。ライブラリーを使用して、関連したオブジェクトをグループ化し、名前でオブジェクトを検索することができます。また、ライブラリーをスキーマとして作成し、データ・ディクショナリーを指定することもできます。

スキーマ (SQL コレクション) にはカタログ視点も含まれ、カタログ視点には、ライブラリーに作成されているすべての表、視点、索引、ファイル、パッケージ、および制約の説明と情報が入っています。 スキーマに作成されているすべての表には、ジャーナル処理が自動的に実行されています。

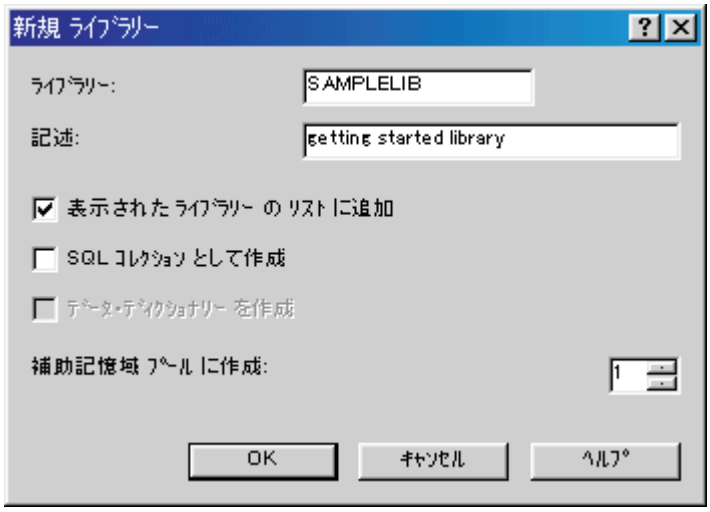
複数のデータベースを処理することもできます。 複数データベースの処理を参照してください。

iSeries ナビゲーターを使用してライブラリー (スキーマ) を作成する方法の例については、『例: iSeries ナビゲーターを使用したライブラリーの作成 (SAMPLELIB)』を参照してください。

例: iSeries ナビゲーターを使用したライブラリーの作成 (SAMPLELIB)

以下のようにして、SAMPLELIB というサンプル・ライブラリーを作成します。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース の順に展開する。
2. 「**ライブラリー**」を右クリックし、「**新規ライブラリー**」を選択する。
3. 「**新規ライブラリー**」ダイアログで、名前フィールドに SAMPLELIB と入力する。
4. 「記述」を入力する (オプション)。
5. 表示されるライブラリーのリストに追加するには、表示されている「**表示されたライブラリーのリストに追加**」を選択する。
6. 「**SQL コレクションとして作成**」を選択すると、SQL コレクションとしてライブラリーを作成できます。「**データ・ディクショナリーを作成**」を選択すると、データ・ディクショナリーを作成できます。 ただし、このサンプル・ライブラリーは、基本ライブラリーとして作成するだけにしてください。
7. ライブラリーを入れるディスク・プールを指定する。1 を選択して、ライブラリーがシステム・ディスク・プールに作成されるようにします。
8. OK をクリックする。



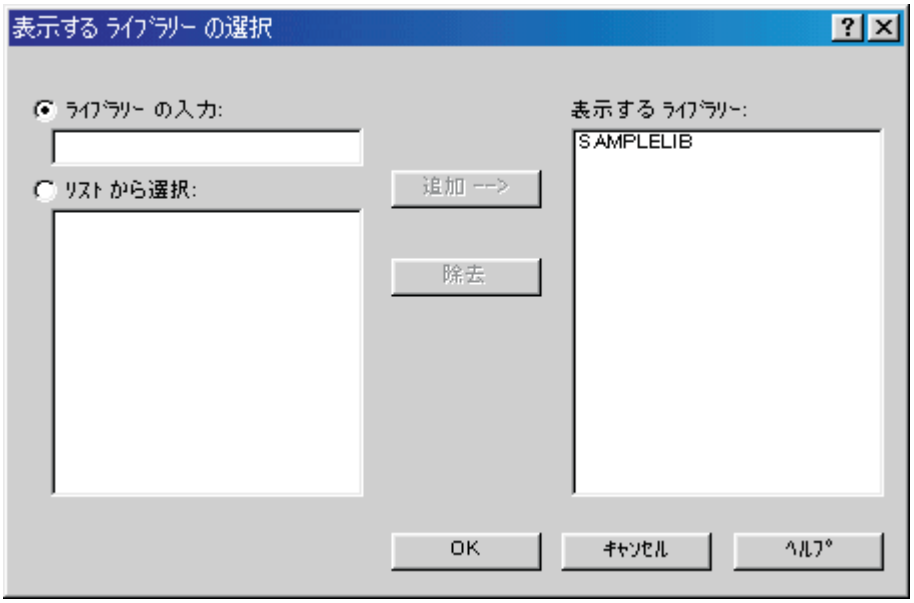
注:

1. SAMPLELIB をスキーマとして作成した場合は、複数のオブジェクトが作成され、このプロセスには数秒かかります。
2. ユーザー・ディスク・プールにライブラリーを作成する詳細については、複数データベースの処理を参照してください。

iSeries ナビゲーターで表示されるライブラリーのリストの編集

ライブラリーが正しく作成されたら、そのライブラリーに、表、視点、索引、ストアード・プロシージャー、ユーザー定義関数、およびユーザー定義タイプを作成することができます。「ライブラリー」をクリックして表示されたライブラリーのリストを編集するには、以下のようにします。

1. **「ライブラリー」**を右クリックして、**「表示するライブラリーの選択」**を選択します。
2. 「**表示するライブラリーの選択**」ダイアログで、ライブラリー名を選択し、「追加」をクリックすることによって、リストを編集することができます。
3. 「表示するライブラリー」のリストからライブラリーを選択し、「**除去**」をクリックすることによって、「表示するライブラリー」のリストからそのライブラリーを除去することができます。



いまは、SAMPLELIB を、表示するライブラリーとして、そのままにしておきます。

iSeries ナビゲーターを使用した表の作成と使用

表とは、情報を保管するのに使用する基本データベース・オブジェクトのことです。表を作成すると、「テーブル・プロパティー」ダイアログを使用して、列 (カラム) を定義し、索引を作成し、トリガーと制約を追加することがでできます。

iSeries ナビゲーターを使用して表を作成する例については、『例: iSeries ナビゲーターを使用した表の作成 (INVENTORY_LIST)』を参照してください。

表を作成するときには、ヌル値と省略時値 (デフォルト値) の概念を理解しておく必要があります。ヌル値は、ある行の列値が存在しないことを示します。 ゼロまたはすべてブランクの値とは異なります。それは「未知」であることを意味しています。 他のどのような値ともまったく異なり、他のヌル値とさえも異なります。列にヌル値が認められない場合は、その列には、なんらかの値を割り当てる必要があります。この値は、省略時値またはユーザー指定値のいずれでもかまいません。

表に行が追加されるときに、ある列に値が指定されていない場合は、その行には省略時値が割り当てられます。特定の省略時値が列に割り当てられていない場合は、その列は、システムの省略時値を使用します。INSERT によって使用される省略時値の詳細については、101 ページの 『INSERT ステートメントを使用した行の挿入』を参照してください。

例: iSeries ナビゲーターを使用した表の作成 (INVENTORY_LIST)

ここでは、現在の在庫に関する情報を保守するための表を作成します。この表には、在庫として保管している品目、それらの原価、現在の手持ち数量、最後の注文日、および最後の注文量に関する情報が入っています。 品目番号は必須値となりま

す。ヌル値であってはなりません。 品目名、在庫数量、および注文量には、ユーザーが指定した省略時値が入ります。最後の注文日と注文量は、ヌル値であっても構いません。

表を作成するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. SAMPLELIB を右クリックし、「**新規作成**」を選択する。
3. 「**テーブル**」を選択する。
4. 「**新規テーブル**」ダイアログで、表名として INVENTORY_LIST を入力する。
5. 「記述」を入力する (オプション)。
6. 「**OK**」をクリックする。

「新規テーブル - INVENTORY_LIST」が表示されます。このダイアログはクローズしないでください。次のステップで必要です。

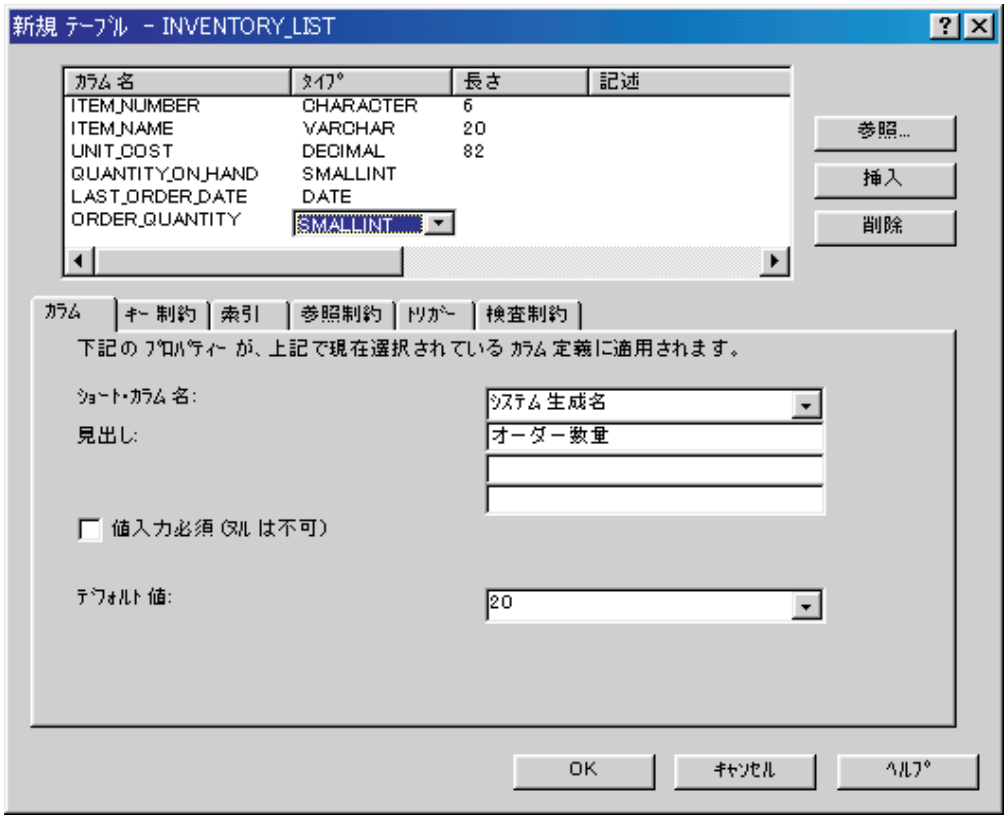
iSeries ナビゲーターを使用した表の列の定義

列は、新規表にも既存の表にも定義することができます。列を既存の表に定義する場合は、「**データベース**」→「**ライブラリー**」→ *SAMPLELIB* (または該当するライブラリー名) のように展開して、表までナビゲートします。詳細ペインで、表「*INVENTORY_LIST*」を右クリックして、「**プロパティー**」を選択します。

1. 「**テーブル・プロパティー**」または「**新規テーブル**」ダイアログで列を定義するには、「**新規**」または「**挿入**」をクリックする。**カラム定義グリッド**に新規の列が表示されます。
2. **カラム定義グリッド**で名前 *ITEM_NUMBER* を入力する。
3. タイプが CHARACTER であることを確かめる。タイプの変更は、いまリストされているタイプをクリックし、下矢印をクリックし、表示されたリストにある別のタイプを選択することによって、行うことができます。
4. この列に、長さ 6 を指定する。サイズがあらかじめ決められているデータ・タイプの場合は、サイズがすでに入っているので、その値を変更できません。
5. この列の「記述」を入力することができます。このステップはオプションです。
6. 「**カラム**」タブの下で、「**ショート・カラム名**」テキスト・ボックスに、短縮名を指定できます。短縮名を指定しない場合は、システムが自動的に名前を生成し

ます。列名が 10 文字以下の場合は、短縮名は列名と同じものになります。どちらの列名を使用しても、照会を実行できます。いまは、このスペースをブランクにしておきます。

7. 各列に列見出しを入力する。
8. 「**値入力必須 (ヌルは不可)**」を選択する。これによって、この列に値が入れられ、行の挿入が正常に行われます。
9. 必ずデフォルト値に「**値**」をセットする。これは、値が入力されなければならない列です。



これで、列 ITEM_NUMBER が定義されました。表 INVENTORY_LIST に、以下の列を追加します。

列名	タイプ	長さ	スケール	NULL	省略時値
ITEM_NAME	VARCHAR	20		ヌルは不可	UNKNOWN
UNIT_COST	DECIMAL	8	2	ヌルは不可	(カラム・データ・タイプのデフォルトに設定)
QUANTITY_ON_HAND	SMALLINT			NULL	NULL
LAST_ORDER_DATE	DATE			NULL	
ORDER_QUANTITY	SMALLINT			NULL	20

以上の列の定義が終了したら、「**OK** 」をクリックして、表を作成します。

iSeries ナビゲーターを使用した供給者表の作成 (SUPPLIERS)

後述する例では、2 つ目の表も必要となります。この表には、在庫品目の供給者、それらの供給者の供給品目、および供給品目の原価に関する情報が入ります。SAMPLELIB に、SUPPLIERS という表を作成します。この表には、列が 3 つ、すなわち、SUPPLIER_NUMBER、ITEM_NUMBER、および SUPPLIER_COST があります。この表には、表 INVENTORY_LIST と共通の列 ITEM_NUMBER があります。新たに ITEM_NUMBER 列を作成するよりも、ITEM_NUMBER に使用した列の定義を、INVENTORY_LIST 表にコピーできます。

iSeries ナビゲーターを使用した列定義のコピー

列定義をコピーするには、以下のようにします。

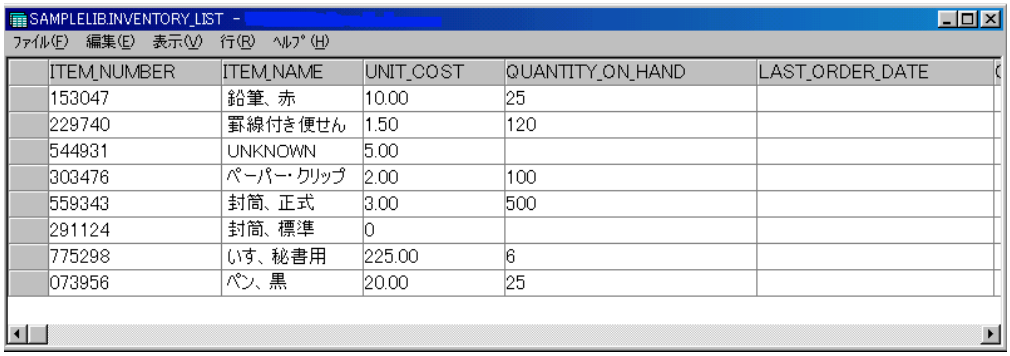
1. SUPPLIER「**テーブル・プロパティー**」または「**新規テーブル**」ダイアログで、「**参照**」をクリックする。
2. 「**テーブルの参照**」ダイアログで、SAMPLELIB を展開する。
3. INVENTORY_LIST をクリックする。その表の中の列が、それぞれのデータ・タイプ、サイズ、記述とともに、リストされます。
4. ITEM_NUMBER を選択する。
5. 「**OK**」をクリックして、この列定義を、表 SUPPLIERS にコピーする。

表 SUPPLIERS の最後の 2 列を、次の値、すなわち、SUPPLIER_NUMBER、CHAR(4)、NOT NULL および SUPPLIER_COST、DECIMAL (8,2) で追加します。

iSeries ナビゲーターを使用した表への情報の挿入

表の中のデータを挿入、編集、削除するには、その表に対して権限を持っていなければなりません。表 INVENTORY_LIST にデータを追加するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. SAMPLELIB をクリックする。
3. INVENTORY_LIST を右クリックして、「**オープン**」を選択する。
4. 「行」メニューから、「**挿入**」を選択する。これにより、新規の行が表示されます。
5. 該当する見出しの下に、以下の情報を入力する。



	ITEM_NUMBER	ITEM_NAME	UNIT_COST	QUANTITY_ON_HAND	LAST_ORDER_DATE
	153047	鉛筆、赤	10.00	25	
	229740	罫線付き便せん	1.50	120	
	544931	UNKNOWN	5.00		
	303476	ペーパー・クリップ	2.00	100	
	559343	封筒、正式	3.00	500	
	291124	封筒、標準	0		
	775298	いす、秘書用	225.00	6	
	073956	ペン、黒	20.00	25	

注: ユーザーが入力する値は、すべての制約を満たし、各列のタイプに合致しなければなりません。 表に対する固有の制約または索引がある場合は、ユーザーが入力する値が固有キーの値を定義しなければなりません。 列に値を入力しない場合は、許可される場合には、省略時値が入力されます。この練習では、省略時値が使用されるように、以下の図表に示されていない値を挿入しないでください。

ITEM_NUMBER	ITEM_NAME	UNIT_COST	QUANTITY_ON_HAND
153047	鉛筆、赤	10.00	25
229740	罫線付き便せん	1.50	120
544931		5.00	
303476	ペーパー・クリップ	2.00	100
559343	封筒、正式	3.00	500
291124	封筒、標準		
775298	いす、秘書用	225.00	6
073956	ペン、黒	20.00	25

「**ファイル**」メニューから、「**保管**」を選択します。

次の行を SAMPLELIB.SUPPLIERS 表に追加してください。

SUPPLIER_NUMBER	ITEM_NUMBER	SUPPLIER_COST
1234	153047	10.00
1234	229740	1.00
1234	303476	3.00
9988	153047	8.00
9988	559343	3.00
2424	153047	9.00
2424	303476	2.50
5546	775298	225.00
3366	303476	1.50
3366	073956	17.00

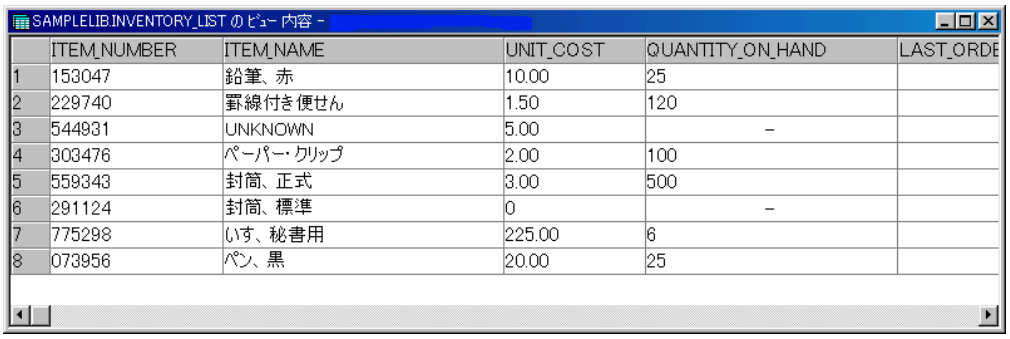
「**ファイル**」メニューから、「**保管**」を選択します。 これでサンプル・ライブラリーには 2 つの表が含まれ、各表には複数のデータ行が入ります。

iSeries ナビゲーターを使用した表の内容の表示

「クイック・ビュー」を使用して、表および視点の内容を表示することができます。内容を表示することだけができます。表を変更する場合は、表をオープンしなければなりません。INVENTORY_LIST の内容を表示するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. SAMPLELIB をクリックする。

3. INVENTORY_LIST を右クリックして、「**クイック・ビュー**」を選択する。



	ITEM_NUMBER	ITEM_NAME	UNIT_COST	QUANTITY_ON_HAND	LAST_ORDE
1	153047	鉛筆、赤	10.00	25	
2	229740	罫線付き便せん	1.50	120	
3	544931	UNKNOWN	5.00	-	
4	303476	ペーパー・クリップ	2.00	100	
5	559343	封筒、正式	3.00	500	
6	291124	封筒、標準	0	-	
7	775298	いす、秘書用	225.00	6	
8	073956	ペン、黒	20.00	25	

注: 「クイック・ビュー」を使用すると、データ・リンク列および関連 URL にアクセスし、URL が選択されたときにブラウザーを立ち上げることもできます。

iSeries ナビゲーターを使用した表内の情報の変更

iSeries ナビゲーターを使用して、表の列の中のデータ値を変更することができます。iSeries ナビゲーターを使用して列を更新し、きょう、ペーパー・クリップの注文を受領したことを示したいとします。入力する値は、その列に有効な値でなければならないことに注意してください。

1. 表 INVENTORY_LIST をダブルクリックしてオープンする。
2. ペーパー・クリップという行の LAST_ORDER_DATE 列に現在日付を入力する。ご使用のシステムでの正しい日付形式を使用してください。
3. ORDER_QUANTITY を 50 に変更する。
4. 変更内容を保管し、「**クイック・ビュー**」を使用して、表の内容を表示する。

ペーパー・クリップの行には、変更内容が反映されています。

iSeries ナビゲーターを使用した表からの情報の削除

iSeries ナビゲーターを使用して、データを表から削除することができます。1 つの行の 1 つの列から情報を削除することも、1 つの行全体を削除することもできます。値が、ある列に必須である場合は、その値は、1 行全体を削除しない限り、削除できないことを覚えておいてください。

1. 表 INVENTORY_LIST をダブルクリックしてオープンする。
2. 封筒、標準の行の ORDER_QUANTITY 列の値を削除する。これは、ヌル値が可能な列であるので、値を削除できます。
3. 罫線付き便せんの行の UNIT_COST の列値を削除する。この列にはヌル値が許されないので、削除は行えません。

複数の列の値を一時に 1 つずつ削除する代わりに、1 つの行全体を削除することもできます。

1. 表 INVENTORY_LIST をダブルクリックしてオープンする。
2. UNKNOWN の行の左にあるグレーのセルをクリックする。これにより、1 つの行全体が強調表示されます。
3. 「行」メニューにある「**削除**」を選択するか、キーボードの Delete (削除) キーを押す。UNKNOWN 行が削除されます。

4. 表 INVENTORY_LIST の中で、QUANTITY_ON_HAND 列に値が入っていない行をすべて削除する。
5. 変更内容を保管し、「クイック・ビュー」を使用して、内容を表示する。以下のデータが入っている表が作成されます。

ITEM_ NUMBER	ITEM_ NAME	UNIT_ COST	QUANTITY_ ON_ HAND	LAST_ ORDER_ DATE	ORDER_ QUANTITY
153047	鉛筆、赤	10.00	25		20
229740	罫線付き便せん	1.50	120		20
303476	ペーパー・クリップ	2.00	100	2000-10-02	50
559343	封筒、正式	3.00	500		20
775298	いす、秘書用	225.00	6		20
073956	ペン、黒	20.00	25		20

iSeries ナビゲーターを使用した表のコピーと移動

iSeries ナビゲーターを使用して、1 つのライブラリーまたはシステムから別のライブラリーまたはシステムに、表をコピーまたは移動することができます。表をコピーすると、表のインスタンスが複数個作成されます。移動は、表を別の位置に転送し、同時に、前の位置からそのインスタンスを除去します。

LIBRARY1 という新規ライブラリーを作成し、これを、表示されるライブラリーのリストに追加します。この新規ライブラリーを作成したら、INVENTORY_LIST を LIBRARY1 にコピーします。表をコピーするには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. SAMPLELIB をクリックする。
3. INVENTORY_LIST を右クリックして、「**コピー**」を選択する。
4. LIBRARY1 を右クリックして、「**貼り付け**」を選択する。

これで、表 INVENTORY_LIST を LIBRARY1 にコピーしたので、表 SUPPLIERS を LIBRARY1 に移動します。表を移動するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. SAMPLELIB をクリックする。
3. SUPPIERS を右クリックして、「**切り取り**」を選択する。
4. LIBRARY1 を右クリックして、「**貼り付け**」を選択する。

注: 表を新規ライブラリーの上にドラッグ・アンド・ドロップすることにより、移動することができます。 表を新規位置に移動することが、常に起動システムから除去することであるとは限りません。 たとえば、ソース表に対して読み取り権限はあるが、削除権限がない場合は、表を受動システムに移動できます。 ただし、表を起動システムから削除できないので、表の 2 つのインスタンスが存在することになります。

iSeries ナビゲーターを使用した視点の作成および使用

必要なすべての情報が 1 つの表にまとまって入っていない場合もあります。また、表に入っているデータの一部だけしかユーザーのアクセスを認めたくない場合もあります。視点を使用すると、表をいくつかのサブセットに分けて、必要なデータだけを取り扱うことができます。視点は複雑さを軽減すると同時に、アクセスを制限します。

視点を作成するには、ユーザーに視点の基礎となる表または物理ファイルに対する正当な権限がなければなりません。必要な権限のリストについては、SQL 参照内の CREATE VIEW ステートメントを参照してください。

視点定義で列名を指定しないと、列名は視点の基礎となる表のものと同じになります。

視点に含まれる列または行の数が表と同じでなくても、視点を介して表に変更を加えることができます。INSERT の場合、視点にない表の列には、省略時値が設定されていなければなりません。

視点は、データの 1 つまたは複数の表に完全に従属していても、それが表であるかのように使用することができます。視点自体はデータを持たないため、データのための記憶域を必要としません。視点は記憶域に存在する表から作られるので、視点のデータを更新するときは、実際は表にあるデータを更新することになります。したがって、視点は、その基礎となる表が更新されると、自動的に更新されます。

追加情報については、 60 ページの『視点の作成および使用』を参照してください。

iSeries ナビゲーターを使用して視点を作成する例については、以下の例を参照してください。

- 『iSeries ナビゲーターを使用した 1 つの表に基づく視点の作成』
- 47 ページの『iSeries ナビゲーターを使用した、複数の表にあるデータを結合した視点の作成』

iSeries ナビゲーターを使用した 1 つの表に基づく視点の作成

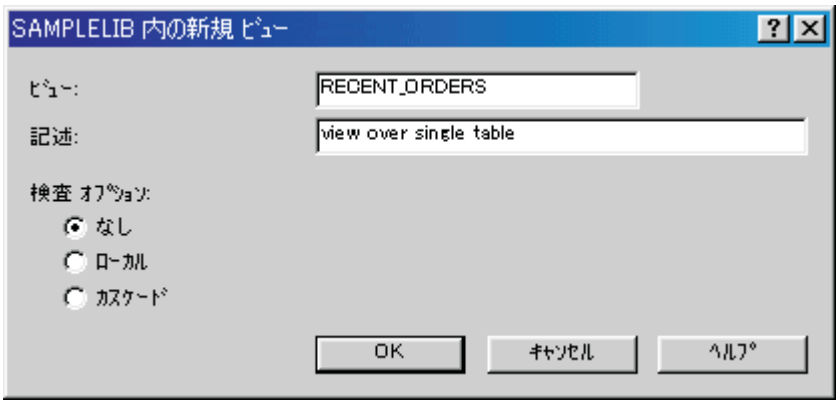
次の例は、1 つの表に基づいて視点を作成する方法を示しています。 この視点は INVENTORY_LIST 表に基づいて作成されます。 この表には 6 つの列がありますが、視点ではそのうちの 3 つの列 (ITEM_NUMBER、LAST_ORDER_DATE、および QUANTITY_ON_HAND) だけを使用します。

1 つの表に基づいて視点を作成するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. SAMPLELIB を右クリックし、「**新規作成**」を、次に「**ビュー**」を選択する。
3. 「**新規ビュー**」ダイアログで、「**ビュー**」フィールドに、RECENT_ORDERS と入力する。
4. オプションで、「記述」を入力することができます。
5. さらに、このダイアログで、「検査オプション」を選択する。視点で検査オプションを選択すると、行に挿入または更新される値が視点の条件に準拠していなけ

ればならないことを指定します。検査オプションについて詳しくは、 155 ページの『視点に関する WITH CHECK OPTION』を参照してください。この視点では、「**なし**」を選択します。

6. 「**OK**」をクリックする。

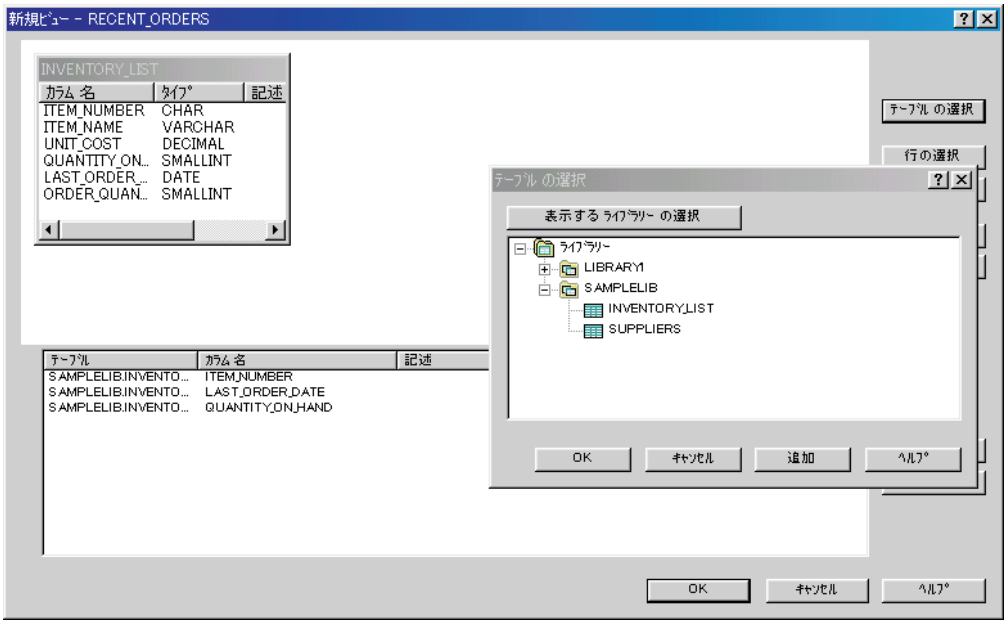


「**新規ビュー**」ダイアログで、以下のことを行います。

1. 「**テーブルの選択**」を選択する。
2. 「**テーブルの選択**」ダイアログで、SAMPLELIB を展開し、次に、INVENTORY_LIST を選択する。
3. 「**追加**」をクリックする。
4. 「**OK**」をクリックする。 INVENTORY_LIST が、「新規ビュー」ダイアログ上の作業域に入ります。
5. 新規視点に入れたい列を選択するには、選択された表内の選択したい列をクリックし、ダイアログの下部にある選択グリッドにドラッグ・アンド・ドロップする。ITEM_NUMBER、LAST_ORDER_DATE、および QUANTITY_ON_HAND を選択します。
6. 列が選択グリッドに入っている順序が、それらの列が視点に表示される順序になります。順序を変更するには、列を選択して、新規の位置にドラッグします。列を、ITEM_NUMBER LAST_ORDER_DATE QUANTITY_ON_HAND の順序で入れます。

これで、視点が基本的には完成しましたが、最近の 2 週間に注文された品目だけを表示したいと考えます。この情報を指定するには、WHERE 文節を作成する必要があります。

1. 「**行の選択**」をクリックする。
2. 「**行の選択**」ダイアログで、WHERE LAST_ORDER_DATE > CURRENT DATE - 14 DAYS と入力する。表示されたオプションから選択して、この WHERE 文節を構成するエレメントを選択することができます。
3. 「**OK**」をクリックする。
4. この視点を生成するのに使用した SQL を表示するには、「**SQL の表示**」をクリックする。
5. 「**OK**」をクリックして、視点を作成する。



RECENT_ORDERS の内容を表示するには、RECENT_ORDERS を右クリックし、「**クイック・ビュー**」を選択します。以下の情報が表示されます。

ITEM_NUMBER	LAST_ORDER_DATE	QUANTITY_ON_HAND
303476	2000-10-02	100

上記の例では、視点の中の列は、表の中の列と同じ名前を持っていますが、これは、新規の名前を指定しなかったからです。視点が作成されて入れられるスキーマは、視点の作成の基礎となる表と同じスキーマでなくても構いません。 どのスキーマまたはライブラリーでも使用することができます。

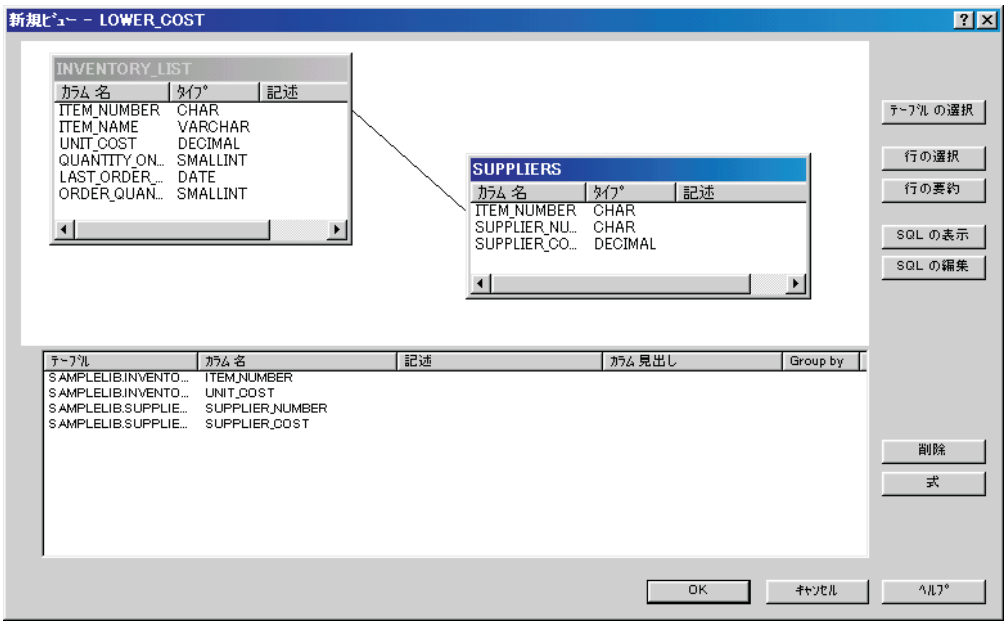
iSeries ナビゲーターを使用した、複数の表にあるデータを結合した視点の作成

「新規ビュー」ダイアログの作業域で複数の表を選択することにより、複数の表にある情報を結合して視点を作成することができます。さまざまな表から、組み込みたい列を選択して OK をクリックすることにより、複数の表から単純な視点を作成することができます。ただし、この例で示すのは、2 つの異なる表にある情報を結合し、表示したい行だけを戻す視点を作成する方法で、WHERE 文節を使用する方法によく似ています。

現行の単価よりも低いコストでサプライできる供給者の品目番号だけが入った視点を作成します。これを作成するには、INVENTORY_LIST 表から ITEM_NUMBER と UNIT_COST を選択し、これらを、SUPPLIERS 表にある SUPPLIER_NUMBER と SUPPLIER_COST に結合することが必要です。WHERE 文節を使用して、戻される行の数を制限します。

1. LOWER_COST という視点を作成する。
2. 「**新規ビュー**」ダイアログで、「**テーブルの選択**」をクリックする。

3. SAMPLELIB から INVENTORY_LIST を選択し、LIBRARY1 から SUPPLIERS を選択する。
4. 「**OK**」をクリックする。 両方の表が、ダイアログの作業域に表示されます。
5. INVENTORY_LIST から、ITEM_NUMBER と UNIT_COST を選択する。
6. SUPPLIERS から、SUPPLIER_NUMBER と SUPPLIER_COST を選択する。
7. 結合を定義するために、INVENTORY_LIST から ITEM_NUMBER を選択し、SUPPLIERS の中の ITEM_NUMBER にドラッグする。 一方の列から他方の列に線が引かれ、「**結合 (Join)**」ダイアログがオープンします。
8. 「**結合**」ダイアログで、「**内部結合**」を選択する。結合について詳しくは、85 ページの『複数の表からのデータの結合』を参照してください。
9. 「**OK**」をクリックします。
10. 前回と同様に、「**SQL の表示**」を選択して、この視点を作成するときに使用される SQL を表示することができます。 「**SQL の編集**」を選択して、SQL を編集することもできます。「**SQL の編集**」は、ユーザーが SQL ステートメントの編集を行える「**SQL スクリプトの実行 (Run SQL Scripts)** 」を立ち上げます。ただし、SQL を変更する場合は、「**新規ビュー**」ダイアログに戻らずに、そのステートメントを「**SQL スクリプトの実行 (Run SQL Scripts)**」から実行する必要があることに注意してください。 「**新規ビュー**」ダイアログに戻る場合は、変更内容は保管されません。
11. 「**行の選択**」をクリックして、この視点の WHERE 文節を作成します。 SUPPLIER_COST をダブルクリックし、次に < 演算子をダブルクリックし、最後に UNIT_COST をダブルクリックします。 品目は、クリックした通りにダイアログに表示されます。これは直接入力することもできます。
12. 「**OK**」をクリックして、視点 LOWER_COST を作成する。



この新規視点の内容を表示するには、LOWER_COST を右クリックし、「**クイック・ビュー**」を選択します。この視点を使用して表示された行は、単価よりも低い供給者原価が入っている行だけです。

SUPPLIER_NUMBER	ITEM_NUMBER	UNIT_COST	SUPPLIER_COST
9988	153047	10.00	8.00
2424	153047	10.00	9.00
1234	229740	1.50	1.00
3366	303476	2.00	1.50
3366	073956	20.00	17.00

iSeries ナビゲーターを使用したデータベース・オブジェクトの削除

これらのオブジェクトをシステム上に作成した後で、これらを除去してシステム・リソースを節約しなければならない場合があります。このタスクを実行するには、削除権限が必要です。

注: これらの表の中の情報を保存したい場合は、第三のライブラリーを作成し、表と視点をそのライブラリーにコピーします。まず、LIBRARY1 から、INVENTORY_LIST 表を除去します。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. LIBRARY1 を展開する。
3. INVENTORY_LIST を右クリックし、「**削除**」を選択するか、Delete (削除) キーを押す。
4. 「**オブジェクトの削除の確認** 」ダイアログで、「**削除**」を選択する。INVENTORY_LIST 表が除去されます。

次に、LIBRARY1 から SUPPLIERS を削除します。

1. SUPPLIERS を右クリックし、「**削除**」を選択するか、Delete (削除) キーを押す。
2. 「オブジェクトの削除の確認」ダイアログで、「**削除**」を選択する。
3. 別のダイアログがオープンし、視点 LOWER_COST が SUPPLIERS に従属しているが、これも削除すべきかどうかが示されます。「**はい**」をクリックする。

SUPPLIERS と LOWER_COST が削除されます。これで LIBRARY1 が空になったので、それを右クリックし、「削除」を選択して削除します。「オブジェクト削除の確認」ダイアログで、「削除」を選択します。LIBRARY1 が削除されます。

最後に、SAMPLELIB を削除します。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. SAMPLELIB を右クリックし、「**削除**」を選択する。
3. 「オブジェクト削除の確認」ダイアログで、「**削除**」を選択する。
4. 別のダイアログがオープンし、表 INVENTORY_LIST および視点 RECENT_ORDERS が INVENTORY_LIST に従属しているが、これも削除すべきかどうかが示されます。「**はい**」をクリックする。

SAMPLELIB、INVENTORY_LIST、および RECENT_ORDERS が削除されます。

iSeries ナビゲーターの使用について詳しくは、 295 ページの『第 16 章 iSeries ナビゲーターを使用した拡張データベース機能』を参照してください。

第 4 章 データ定義言語 (DDL)

データ定義言語 (DDL) は、データベース・オブジェクトの作成、変更、および破棄を行えるようにする、SQL の部分です。このデータベース・オブジェクトには、スキーマ、表、視点、索引、および別名が含まれます。

詳細については、以下のセクションを参照してください。

- 『スキーマの作成』
- 52 ページの『表の作成』
- 53 ページの『LIKE を使用した表の作成』
- 53 ページの『AS を使用した表の作成』
- 54 ページの『グローバル一時表の宣言』
- 54 ページの『識別列の作成および変更』
- 55 ページの『ROWID』
- 55 ページの『LABEL ON ステートメントを使用した記述ラベルの作成』
- 56 ページの『COMMENT ON を使用した SQL オブジェクトの記述』
- 57 ページの『表の定義の変更』
- 60 ページの『ALIAS 名の作成と使用』
- 60 ページの『視点の作成および使用』
- 63 ページの『索引の追加』
- 63 ページの『データベース設計でのカタログ』
- 65 ページの『データベース・オブジェクトの除去』

スキーマの作成

スキーマとは、SQL オブジェクトを論理的にグループ化したものです。スキーマは、ライブラリー、ジャーナル、ジャーナル・レシーバー、カタログ、および、オプションとして、データ・ディクショナリーから構成されます。 表、視点、およびシステム・オブジェクト (プログラムなど) は、どのシステム・ライブラリーにも作成、移動、あるいは復元することができます。 SQL スキーマにデータ・ディクショナリーが入っていない場合は、すべてのシステム・ファイルを SQL スキーマ内に作成または移動することができます。 SQL スキーマにデータ・ディクショナリーが入っている場合には、以下のようになります。

- 1 つのメンバーから成るソース物理ファイルまたは非ソース物理ファイルは、SQL スキーマ内に作成、移動、または復元することができます。
- 論理ファイルは、データ・ディクショナリーで記述できないため、SQL スキーマに置くことはできません。

ユーザーは多数のスキーマを作成し、所有することができます。

スキーマは、CREATE SCHEMA ステートメントを使用して作成されます。たとえば、次の通りです。

DBTEMP というスキーマを作成します。

```
CREATE SCHEMA DBTEMP
```

スキーマ の同義語としてコレクション という用語を使うこともできます。CREATE SCHEMA ステートメントについて詳しくは、*SQL 解説書* の CREATE SCHEMA を参照してください。

表の作成

表は、行と列から構成されるデータの 2 次元の配列として理解することができます。行は、1 つまたは複数の列を含む横方向の構成部分です。 列は、1 つのデータ・タイプのデータの 1 つまたは複数の行を含む縦方向の構成部分です。 1 つの列に含まれるデータはすべて同一タイプでなくてはなりません。SQL の表は、キー付きまたはキーなしの物理ファイルです。 データ・タイプの説明については、「SQL 解説書」の中の、データ・タイプのトピックを参照してください。

表は、CREATE TABLE ステートメントを使用して作成されます。 定義には、表の名前、列の名前および属性が含まれている必要があります。定義には、基本キーなど、表に関するその他の属性を含めることができます。

例 1: 管理権限を与えられているものとして、INVENTORY という名前で、次の列を持つ表を作成します。

- 部品番号: 1 ～ 9 999 の間の整数で、ヌルは許されない
- 記述: 長さ 0 ～ 24 の文字
- 在庫数量: 0 ～ 100000 の間の整数

基本キーは PARTNO です。

```
CREATE TABLE INVENTORY
                (PARTNO         SMALLINT     NOT NULL,
                 DESCR          VARCHAR(24 ),
                 QONHAND        INT,
                 PRIMARY KEY(PARTNO))
```

表への制約の追加

新規の表、または既存の表に、制約を追加することができます。固有キーまたは基本キー、参照制約、あるいは検査制約を、 CREATE TABLE ステートメントまたは ALTER TABLE ステートメントの *ADD 制約* 文節を使用して追加することができます。たとえば、基本キーを新規の表または既存の表に追加するとします。次の例では、ALTER TABLE ステートメントを使用して既存の表に基本キーを追加します。

```
ALTER TABLE CORPDATA.DEPARTMENT
  ADD PRIMARY KEY (DEPTNO)
```

このキーを固有キーにするには、キーワード PRIMARY を UNIQUE に置き換えるだけです。詳細については、 145 ページの『第 10 章 データ保全性』を参照してください。

LIKE を使用した表の作成

別の表と同様な表を作成することができます。つまり、既存の表からすべての列定義を組み込んだ表を、作成できるということです。コピーされる定義は、次のとおりです。

- 列名 (およびシステム列名)
- データ・タイプ、精度、長さ、およびスケール
- CCSID
- 列テキスト (LABEL ON)
- 列見出し (LABEL ON)

LIKE 文節が表名の直後に続き、しかも括弧で囲まれない場合は、以下の属性も組み込まれます。

- 省略時値
- ヌル可/不可

指定された表または視点が識別列を含んでいる場合に、新規の表にも識別列を存在させたいのであれば、CREATE TABLE ステートメントにおいて必ず INCLUDING IDENTITY を指定する必要があります。 CREATE TABLE の省略時の動作は、EXCLUDING IDENTITY です。指定される表または視点が、SQL 以外で作成された物理ファイルまたは論理ファイルの場合、すべての非 SQL 属性は除去されます。

EMPLOYEE にあるすべての列を含む表 EMPLOYEE2 を、作成します。

```
CREATE TABLE EMPLOYEE2 LIKE EMPLOYEE
```

CREATE TABLE LIKE についてより詳しくは、*SQL 解説書* のCREATE TABLE を参照してください。

AS を使用した表の作成

CREATE TABLE AS は、SELECT ステートメントの結果から表を作成します。SELECT ステートメントで使用できるすべての式を、CREATE TABLE AS ステートメントで使用することができます。選択の対象となる表 (単数または複数) からのすべてのデータを、組み込むこともできます。

たとえば、EMPLOYEE から DEPTNO = D11 であるすべての列定義を組み込んで、EMPLOYEE3 という名前の表を作成します。

```
CREATE TABLE EMPLOYEE3 AS
   (SELECT PROJNO, PROJNAME, DEPTNO
    FROM EMPLOYEE
    WHERE DEPTNO = 'D11') WITH NO DATA
```

指定された表または視点が識別列を含んでいる場合に、新規の表にも識別列を存在させたいのであれば、CREATE TABLE ステートメントにおいて必ず INCLUDING IDENTITY を指定する必要があります。 CREATE TABLE の省略時の動作は、EXCLUDING IDENTITY です。 WITH NO DATA 文節は、列定義をデータ抜きでコピーするという意味です。新規の表 EMPLOYEE3 にデータを入れたい場合は、WITH DATA を指定します。 SELECT の使用の詳細については、67 ページの『第

5 章 SELECT ステートメントを使用したデータの検索』を参照してください。 指定される照会が、SQL 以外で作成された物理ファイルまたは論理ファイルの場合、すべての非 SQL 結果属性は除去されます。 CREATE TABLE AS についてより詳しくは、*SQL 解説書* のCREATE TABLE を参照してください。

グローバル一時表の宣言

DECLARE GLOBAL TEMPORARY TABLE ステートメントを使用して、ユーザーの現行セッションで使用する一時表を作成することができます。この一時表はシステム・カタログには表示されず、他のセッションと共用することもできません。ユーザーがセッションを終了すると、一時表の行は削除され、表が除去されます。

このステートメントの構文は、LIKE および AS 文節も含め、CREATE TABLE と同じです。

たとえば、一時表 ORDERS は次のように作成します。

```
DECLARE GLOBAL TEMPORARY TABLE ORDERS
               (PARTNO   SMALLINT NOT NULL,
                DESCR    VARCHAR(24),
                QONHAND  INT)
      ON COMMIT DELETE ROWS
```

この表は QTEMP に作成されます。スキーマ名を使用してこの表を参照するには、SESSION または QTEMP のどちらかを使用します。この表に対して、他の表の場合とまったく同じように、SELECT、INSERT、UPDATE、および DELETE の各ステートメントを出すことができます。この表は、次のように DROP TABLE ステートメントを出して除去することができます。

```
DROP TABLE ORDERS
```

詳細については、*SQL 解説書* の DECLARE GLOBAL TEMPORARY TABLE を参照してください。

識別列の作成および変更

識別列を使用して表に新しい行を追加するたびに、新しい行の識別列値がシステムによって増分 (または減分) されます。 識別列として作成できるのは、タイプ SMALLINT、INTEGER、BIGINT、DECIMAL、または NUMERIC の列だけです。識別列は、1 つの表につき 1 つだけ許可されます。表の定義を変更するときには、追加される列だけを識別列として指定できます。既存の列は識別列には指定できません。

表を作成するときは、桁の中の 1 つの列を識別列として定義することができます。たとえば、ORDERNO、SHIPPED_TO、ORDER_DATE という名前の 3 つの列を持つ表 ORDERS を作成します。ORDERNO を識別列として定義します。

```
CREATE TABLE ORDERS
   (ORDERNO SMALLINT NOT NULL
   GENERATED ALWAYS AS IDENTITY
   (START WITH 500
   INCREMENT BY 1
   CYCLE),
   SHIPPED_TO VARCHAR (36) ,
   ORDER_DATE DATE)
```

この列は、開始値 500 で定義され、新しい行が挿入されるたびに 1 ずつ増分され、最大値に達した場合はリサイクルされます。この例では、識別列の最大値は、データ・タイプにおける最大値です。データ・タイプが SMALLINT として定義されているため、ORDERNO に割り当てることのできる値の範囲は 500 ～ 32767 です。この列の値が 32767 に達したら、もう一度 500 から再開されます。ある列に 500 がまだ割り当てられたままで、しかもこの識別列に固有キーの指定がされている場合は、重複キー・エラーが戻されます。次の挿入は、501 を使用するように試みられます。識別列に固有キーの指定がされていない場合は、表の中に 500 が何度現れようと、500 が再度使用されます。

値の範囲がもっと大きい場合は、列に対して INTEGER あるいは BIGINT も指定することができます。識別列の値を減分させたい場合は、INCREMENT オプションに負の値を指定します。 MINVALUE と MAXVALUE を使用して、数値の正確な範囲を指定することもできます。

ALTER TABLE ステートメントを使用して、既存の識別列の属性を変更することができます。たとえば、識別列を新規の値にして再開したい場合は、次のようにします。

```
ALTER TABLE ORDER
  ALTER COLUMN ORDERNO
  RESTART WITH 1
```

列から識別属性を除去することもできます。

```
ALTER TABLE ORDER
  ALTER COLUMN ORDERNO
  DROP IDENTITY
```

ORDERNO は SMALLINT 列のままですが、識別属性は除去されます。システムは、この列の値をもはや生成しなくなります。

ROWID

ROWID の使用は、表の中の列に固有値を割り当てるもう 1 つの方法です。ROWID は識別列と同じですが、数値列の属性にはならず、異なるデータ・タイプです。識別列の例と同じ表を作成するには、以下のようにします。

```
CREATE TABLE ORDERS
   (ORDERNO ROWID
   GENERATED ALWAYS,
   SHIPPED_TO VARCHAR (36) ,
   ORDER_DATE DATE)
```

LABEL ON ステートメントを使用した記述ラベルの作成

対話式画面に表を表示するとき、表名、列名、視点名、別名、または SQL パッケージ名では、データの定義が明確ではない場合があります。 LABEL ON ステートメントを使用すると、表名、列名、視点名、別名または SQL パッケージ名に対して、より内容の分かりやすいラベルを作成することもできます。これらのラベルは、SQL カタログ内の LABEL 列で見ることができます。

LABEL ON ステートメントは、次のようになります。

```
LABEL ON
  TABLE CORPDATA.DEPARTMENT IS 'Department Structure Table'

LABEL ON
  COLUMN CORPDATA.DEPARTMENT.ADMRDEPT IS 'Reports to Dept.'
```

これらのステートメントが実行されると、DEPARTMENT という名前の表には *Department Structure Table* というテキスト記述が表示され、*ADMRDEPT* という名前の列には *Reports to Dept* という見出しが表示されます。表、視点、SQL パッケージ、および列テキストのラベルの長さは最大 50 桁であり、列見出しのラベルの長さは最大 60 桁です (ブランクを含む)。以下に、列見出しの LABEL ON ステートメントの例を示します。

この LABEL ON ステートメントは、列見出し 1 と列見出し 2 を提供します。

```
*...+....1....+....2....+....3....+....4....+....5....+....6..*
LABEL ON COLUMN CORPDATA.EMPLOYEE.EMPNO IS
          'Employee            Number'
```

この LABEL ON ステートメントは、SALARY 列用の 3 つのレベルの列見出しを提供します。

```
*...+....1....+....2....+....3....+....4....+....5....+....6..*
LABEL ON COLUMN CORPDATA.EMPLOYEE.SALARY IS
           'Yearly              Salary              (in dollars)'
```

この LABEL ON ステートメントは、SALARY の列見出しを削除します。

```
*...+....1....+....2....+....3....+....4....+....5....+....6..*
LABEL ON COLUMN CORPDATA.EMPLOYEE.SALARY IS ''
```

次の例は、2 つのレベルを指定した DBCS 列見出しです。

```
*...+....1....+....2....+....3....+....4....+....5....+....6..*
LABEL ON COLUMN CORPDATA.EMPLOYEE.SALARY IS
           '<AABBCCDD>          <EEFFGG>'
```

この LABEL ON ステートメントは、EDLEVEL 列用の列テキストを提供します。

```
*...+....1....+....2....+....3....+....4....+....5....+....6..*
LABEL ON COLUMN CORPDATA.EMPLOYEE.EDLEVEL TEXT IS
   'Number of years of formal education'
```

LABEL ON ステートメントの詳細については、SQL 解説書のLABEL ON ステートメントを参照してください。

COMMENT ON を使用した SQL オブジェクトの記述

表、視点、索引、パッケージ、プロシージャー、パラメーター、ユーザー定義タイプ、関数、またはトリガーのような SQL オブジェクトの作成後、以後の参照用として、ユーザーは、そのオブジェクトの目的、使用者、および特殊性あるいは固有性などの情報を提供することができます。表または視点の各列についても、同様の情報を組み込むことができます。注釈は 2000 バイト以下でなければなりません。COMMENT ON ステートメントの詳細については、*SQL 解説書* の COMMENT ON を参照してください。

名前だけでは列またはオブジェクトの内容を明確に表せない場合には、注釈が特に役立ちます。その場合には、注釈を用いて、列またはオブジェクトの特定の内容を記述します。

COMMENT ON の使用例を次に示します。

```
COMMENT ON TABLE CORPDATA.EMPLOYEE IS
   'Employee table.  Each row in this table represents
    one employee of the company.'
```

COMMENT ON ステートメント実行後の注釈の取り出し

表用に COMMENT ON ステートメントを実行すると、注釈は SYSTABLES の *LONG_COMMENT* 列に格納されます。 その他のオブジェクトの注釈は、該当するカタログ表の LONG_COMMENT 列に格納されます。 (指示した行にすでに注釈が含まれていた場合には、古い注釈は新しい注釈により置換されます。) 次の例では、前の例で COMMENT ON ステートメントにより追加された注釈を取り出します。

```
SELECT LONG_COMMENT
  FROM CORPDATA.SYSTABLES
  WHERE NAME = 'EMPLOYEE'
```

表の定義の変更

表の定義を変更することにより、新しい列の追加、既存の列定義の変更 (その長さおよび省略時値などの変更)、既存の列の削除、および制約の追加と除去を行うことができます。 表の定義は、SQL ALTER TABLE ステートメントを使用して変更します。

列の追加、変更または削除、および、制約の追加または削除は、すべて 1 つの ALTER TABLE ステートメントで行うことができます。ただし、ADD COLUMN、ALTER COLUMN、および DROP COLUMN 文節では、1 つの列は 1 回だけしか参照できません。すなわち、同じ ALTER TABLE ステートメントの中で、ある列を追加してから、その列を更新することはできません。

詳細については、以下のトピックを参照してください。

- 『列の追加』
- 58 ページの『列の変更』
- 58 ページの『可能な変換』
- 59 ページの『列の削除』
- 59 ページの『ALTER TABLE ステートメントの操作の順序』

列の追加

SQL ALTER TABLE ステートメントの ADD COLUMN 文節を使用して、表に列を追加することができます。

新しい列が表に追加されると、その列は既存のすべての行について省略時値で初期設定されます。NOT NULL を指定する場合は、省略時値も指定しなければなりません。

更新された表は最大 8000 列まで構成することができます。列のバイト・カウントの合計は、32766 より大きくてはならず、また VARCHAR または VARGRAPHIC

列が指定される場合は、32740 より大きくてはなりません。LOB 列が指定された場合は、列のレコード・データ・バイト・カウントの合計が 15 728 640 より大きな数になってはなりません。

列の変更

ALTER TABLE ステートメントの ALTER COLUMN 文節を使用して、表の列定義を変更することができます。既存の列のデータ・タイプを変更する場合は、旧属性と新属性に互換性がなければなりません。『可能な変換』には、属性に互換性のある変換が示されています。

より長い長さのデータ・タイプに変換するとき、データは適切な埋め込み文字で埋められます。より短い長さのデータ・タイプに変換するときは、切り捨てによってデータが失われる場合があります。照会メッセージにより、要求を確認するよう指示されます。

ヌル値を認めない列があり、その列を、ヌル値を認めるように変更したい場合は、DROP NOT NULL 文節を使用します。ヌル値を認める列があり、ヌル値の使用を防ぎたい場合は、SET NOT NULL 文節を使用します。その列中の既存の値のいずれかがヌル値の場合、ALTER TABLE は実行されず、結果的に SQLCODE が -190 となります。

可能な変換

表 2. 可能な変換

FROM データ・タイプ	**TO データ・タイプ**
10 進数	数値
10 進数	大きい整数、整数、小さい整数
10 進数	浮動
数値	10 進数
数値	大きい整数、整数、小さい整数
数値	浮動
大きい整数、整数、小さい整数	10 進数
大きい整数、整数、小さい整数	数値
大きい整数、整数、小さい整数	浮動
浮動	数値
浮動	大きい整数、整数、小さい整数
文字	DBCS 混用
文字	UCS-2 グラフィック
DBCS 混用	文字
DBCS 混用	UCS-2 グラフィック
DBCS 択一	文字
DBCS 択一	DBCS 混用
DBCS 択一	UCS-2 グラフィック
DBCS 専用	DBCS 混用
DBCS 専用	DBCS グラフィック

表 2. 可能な変換 (続き)

FROM データ・タイプ	TO データ・タイプ
DBCS 専用	UCS-2 グラフィック
DBCS グラフィック	UCS-2 グラフィック
UCS-2 グラフィック	文字
UCS-2 グラフィック	DBCS 混用
UCS-2 グラフィック	DBCS グラフィック
特殊タイプ	ソース・タイプ
ソース・タイプ	特殊タイプ

既存の列を変更するときは、指定する属性のみが変更されます。他のすべての属性は変更されません。たとえば、次の表の定義があるとします。

```
CREATE TABLE EX1 (COL1 CHAR(10) DEFAULT 'COL1',
                  COL2 VARCHAR(20) ALLOCATE(10) CCSID 937,
                  COL3 VARGRAPHIC(20) ALLOCATE(10)
                       NOT NULL WITH DEFAULT)
```

次の ALTER TABLE ステートメントが実行された後、

```
ALTER TABLE EX1 ALTER COLUMN COL2 SET DATA TYPE VARCHAR(30)
                ALTER COLUMN COL3 DROP NOT NULL
```

COL2 は依然として割り振られた長さ 10 および CCSID 937 を持ち、COL3 は依然として割り振られた長さ 10 を持ちます。

列の削除

ALTER TABLE ステートメントの DROP COLUMN 文節を使用して、列を削除することができます。

列を除去すると、表の定義からその列が削除されます。CASCADE が指定された場合、その列に従属するすべての視点、索引、および制約も除去されます。RESTRICT が指定された場合、その列に従属する視点、索引、または制約があると、その列は除去されず、SQLCODE -196 が発行されます。

```
ALTER TABLE DEPT
  DROP COLUMN NUMDEPT
```

ALTER TABLE ステートメントの操作の順序

ALTER TABLE ステートメントは、以下のように一連のステップとして実行されます。

1. 制約の除去。
2. RESTRICT オプションが指定されている列の除去。
3. 列定義の更新 (これには、CASCADE オプションが指定されている列の追加および除去が含まれます)。
4. 制約の追加。

これらの各ステップでは、ユーザーが文節を指定する順序がそれらのステップが実行される順序になりますが、例外が 1 つあります。列のいずれかが除去される場

合、その操作は、レコード長が ALTER TABLE ステートメントの結果として増加される場合に備えて、列定義の追加または更新が行われる前に論理的に実行されます。

ALIAS 名の作成と使用

既存の表または視点を参照する場合、または複数のメンバーで構成されている物理ファイルを参照する場合は、別名を作成することにより、ファイル一時変更の使用を避けることができます。 これは、SQL CREATE ALIAS ステートメントを使用して行うことができます。

以下のものに別名を作成することができます。

- 表または視点
- 表のメンバー

表の別名は、特定のメンバー名を含むファイル名を定義します。SQL ステートメントの中で、この別名を、表名を使用するのと同じように使用することができます。一時変更とは異なり、別名は除去されるまで存在するオブジェクトであるといえます。

たとえば、MBR1 と MBR2 というメンバーを含む複数のメンバー・ファイル MYLIB.MYFILE がある場合、SQL で簡単に参照できるように、2 番目のメンバーについて別名を作成することができます。

```
CREATE ALIAS MYLIB.MYMBR2_ALIAS FOR MYLIB.MYFILE (MBR2)
```

次の INSERT ステートメントに別名 MYLIB.MYMBR2_ALIAS を指定すると、その値は MYLIB.MYFILE のメンバー MBR2 に挿入されます。

```
INSERT INTO MYLIB.MYMBR2_ALIAS VALUES('ABC', 6)
```

別名は、DDL ステートメントでも指定することができます。 MYLIB.MYALIAS という別名があり、これは表 MYLIB.MYTABLE の別名であるとします。次の DROP ステートメントにより、表 MYLIB.MYTABLE が除去されます。

```
DROP TABLE MYLIB.MYALIAS
```

表ではなく別名を除去したい場合、その DROP ステートメントに ALIAS キーワードを指定してください。

```
DROP ALIAS MYLIB.MYALIAS
```

視点の作成および使用

視点を使用すると、1 つまたは複数の表内のデータにアクセスできます。これは、SELECT ステートメントを使用することにより行います。 SELECT 文節の使用について詳しくは、67 ページの『第 5 章 SELECT ステートメントを使用したデータの検索』を参照してください。視点の場合、ORDER BY 文節は使用できません。

たとえば、すべての管理者の姓と所属部門だけを選択する視点を作成するには、次のように指定します。

```
CREATE VIEW CORPDATA.EMP_MANAGERS AS
   SELECT LASTNAME, WORKDEPT FROM  CORPDATA.EMPLOYEE
   WHERE JOB = 'MANAGER'
```

選択リストに列以外の要素 (式、関数、定数、または特殊レジスターなど) が含まれていて、しかも列に名前を付けるために AS 文節が使用されていない場合には、視点の列リストを指定する必要があります。次の例では、視点の列は LASTNAME と YEARSOFSERVICE です。

```
CREATE VIEW CORPDATA.EMP_YEARSOFSERVICE
    (LASTNAME, YEARSOFSERVICE) AS
   SELECT LASTNAME, YEARS (CURRENT DATE - HIREDATE)
   FROM CORPDATA.EMPLOYEE
```

前述の視点は、選択リスト内で AS 文節を使用して視点の列に名前を付けることによって定義することもできます。たとえば、次の通りです。

```
CREATE VIEW CORPDATA.EMP_YEARSOFSERVICE AS
    SELECT LASTNAME,
           YEARS (CURRENT_DATE - HIREDATE) AS YEARSOFSERVICE
    FROM CORPDATA.EMPLOYEE
```

視点を作成したら、表名と同じように SQL ステートメントの中で使用することができます。また、基礎となる表のデータを変更することもできます。

視点を作成する際には、次の制約事項を考慮する必要があります。

- 読み取り専用視点を使用して、データを変更、挿入、または削除することはできません。 視点は、次のいずれかを含んでいる場合は読み取り専用になります。
  - 複数の表 (結合) を識別する最初の FROM 文節
  - 読み取り専用視点を識別する最初の FROM 文節
  - ユーザー定義表関数を識別する最初の FROM 文節
  - SQL 列関数 (SUM、MAX、MIN、AVG、COUNT、STDDEV、COUNT_BIG、または VAR) のどれかを含む最初の SELECT 文節
  - キーワード DISTINCT を指定する最初の SELECT 文節
  - GROUP BY 文節または HAVING 文節を含む外部副選択
  - UNION 文節を含む外部副選択
  - 最外部副選択の基礎となるオブジェクトと副照会の表とが同じ表であるような副照会

    これらの条件に該当する場合は、SQL の SELECT ステートメントを用いて視点からデータを取り出すことはできますが、INSERT、UPDATE、または DELETE などのステートメントは使用できません。
- 次の場合には、行を視点に挿入することはできません。
  - 視点の基礎となっている表に、省略時値がなく、ヌル値が許可されず、かつ視点に入っていない列がある場合。
  - 式、定数、関数、または特殊レジスターの結果が入る列が視点に含まれていて、その列が INSERT の列リストに指定されている場合。
  - 視点が作成されたときに WITH CHECK OPTION が指定されていて、行が選択基準に一致しない場合。
- 視点内の、式、定数、関数、または特殊レジスターの結果である列を更新することはできません。
- 視点の定義では、FOR UPDATE OF、FOR READ ONLY、FETCH FIRST *n* ROWS、ORDER BY、OPTIMIZE FOR n ROWS、または分離文節を使用することはできません。

視点は、CREATE VIEW ステートメントの実行時に有効な分類順序を使用して作成されます。分類順序は、CREATE VIEW ステートメントの副選択におけるすべての文字および UCS-2 グラフィック比較に適用されます。分類順序の詳細については、123 ページの 『第 8 章 SQL での分類順序』を参照してください。

視点は、UNION 演算子を使用して作成することができます。詳しくは、『UNION を使用した視点の作成』を参照してください。

視点を介してデータを挿入または更新するときに行わなければならない検査レベルを指定するために、WITH CHECK OPTION を使用して視点を作成することもできます。詳細については、155 ページの 『視点に関する WITH CHECK OPTION』を参照してください。

UNION を使用した視点の作成

UNION キーワードを使用すると、2 つ以上の副選択を結合して 1 つの視点にすることができます。 たとえば、次の通りです。

```
CREATE VIEW D11_EMPS_PROJECTS AS
  (SELECT EMPNO
     FROM CORPDATA.EMPLOYEE
     WHERE WORKDEPT = 'D11'
   UNION
   SELECT EMPNO
     FROM CORPDATA.EMPPROJACT
     WHERE PROJNO = 'MA2112' OR
        PROJNO = 'MA2113' OR
        PROJNO = 'AD3111')
```

この結果は、次のようなデータを持つ視点です。

表 3. UNION の結果としての視点の作成

EMPNO
000060
000150
000160
000170
000180
000190
000200
000210
000220
000230
000240
200170
200220

UNION の詳細については、93 ページの『副選択を結合するための UNION キーワードの使用』を参照してください。

索引の追加

索引を使用して、データのソートと選択ができます。さらに、索引を使用すると、システムはデータをより速く取り出すことができ、照会のパフォーマンスが向上します。

索引を作成するには、CREATE INDEX ステートメントを使用します。次の例では、CORPDATA.EMPLOYEE 表の *LASTNAME* 列に対する索引を作成します。

```
CREATE INDEX CORPDATA.INX1 ON CORPDATA.EMPLOYEE (LASTNAME)
```

索引はいくつでも作成できます。ただし、索引はシステムによって保守されるため、索引の数が多いとパフォーマンスが低下することもあります。索引および照会のパフォーマンスの詳細については、データベース・パフォーマンスおよび *Query 最適化* のSQL 索引の効果的使用法を参照してください。

コード化ベクトル索引というタイプの索引を使用すると、並列処理が簡単に行え、より高速のスキャンが可能になります。コード化ベクトル索引は SQL CREATE INDX ステートメントを使用して作成します。コード化ベクトル索引を使用して照会をスピードアップする方法について詳しくは、  「DB2 for iSeries の Web ページ」を参照してください。

既存の索引とまったく同じ属性を持つ索引を作成する場合、新規の索引は既存の索引のバイナリー・ツリーを共用します。そうでない場合は、別のバイナリー・ツリーが作成されます。新規索引の列の数が少ないという点を除き、新規索引の属性が別の索引と全く同じである場合でも、別のバイナリー・ツリーは作成されます。 別のバイナリー・ツリーが作成される理由は、余分の列があると、それらの列を更新するカーソルまたは UPDATE ステートメントが索引を使用できないようになるためです。

索引は、CREATE INDEX ステートメントの実行時に有効な分類順序を使用して作成されます。分類順序は、索引のすべての SBCS 文字フィールドと UCS-2 グラフィック・フィールドに適用されます。分類順序の詳細については、123 ページの『第 8 章 SQL での分類順序』を参照してください。

データベース設計でのカタログ

カタログは、スキーマを作成するときに自動的に作成されます。さらに、常に QSYS2 ライブラリーに存在するシステム全体のカタログもあります。 SQL オブジェクトをスキーマに作成すると、システム・カタログ表とスキーマのカタログ表の両方に情報が追加されます。 SQL オブジェクトをライブラリー内で作成すると、QSYS2 カタログだけが更新されます。 DECLARE GLOBAL TEMPORARY TABLE を使用して作成された表は、カタログには追加されません。カタログの詳細については、SQL 解説書 を参照してください。

以下の例で示されているとおり、カタログ情報は表示することができます。カタログ情報の INSERT、DELETE、または UPDATE を行うことはできません。以下の例を実行するためには、カタログ視点に対する SELECT 特権が必要です。

表に関するカタログ情報の入手

SYSTABLES には、SQL スキーマ内の各表および視点につき 1 つの行が入っています。その行を表示すれば、オブジェクトが表または視点のいずれかであるか、オブジェクト名、オブジェクトの所有者、オブジェクトが入っている SQL スキーマなどの情報を知ることができます。

次のサンプル・ステートメントでは、CORPDATA.DEPARTMENT 表に関する情報が表示されます。

```
SELECT *
  FROM CORPDATA.SYSTABLES
  WHERE NAME = 'DEPARTMENT'
```

列に関するカタログ情報の入手

SYSCOLUMNS には、スキーマ内のすべての表および視点の各列につき 1 つの行が入っています。

次のサンプル・ステートメントでは、CORPDATA.DEPARTMENT 表内のすべての列の名前が表示されます。

```
SELECT *
  FROM CORPDATA.SYSCOLUMNS
  WHERE TBNAME = 'DEPARTMENT'
```

このサンプル・ステートメントを実行すると、表内の各列につき 1 行の情報が表示されます。情報の長さが表示画面より長いために、情報の一部が見えないことがあります。

各列の詳しい情報を表示するには、選択ステートメントを次のように指定してください。

```
SELECT NAME, TBNAME, COLTYPE, LENGTH, DEFAULT
  FROM CORPDATA.SYSCOLUMNS
  WHERE TBNAME = 'DEPARTMENT'
```

この選択ステートメントでは、各列の列名のほかに次の情報が表示されます。

- 列が入っている表の名前
- 列のデータ・タイプ
- 列の長さ属性
- 列が省略時値を認めるかどうか

結果は次のようになります。

NAME	TBNAME	COLTYPE	LENGTH	DEFAULT
DEPTNO	DEPARTMENT	CHAR	3	N
DEPTNAME	DEPARTMENT	VARCHAR	29	N
MGRNO	DEPARTMENT	CHAR	6	Y
ADMRDEPT	DEPARTMENT	CHAR	3	N

データベース・オブジェクトの除去

DROP ステートメントは、オブジェクトを削除します。該当オブジェクトに直接または間接に従属するオブジェクトは、要求されるアクションにより、同時に削除される場合もあり、除去されない場合もあります。たとえば、表を除去すると、その表に関連するすべての別名、制約、トリガー、視点、または索引も、同時に除去されます。オブジェクトが削除されるときは必ず、そのオブジェクトについての記述がカタログから削除されます。

たとえば、表 EMPLOYEE を除去するには、次のステートメントを出します。

```
DROP TABLE EMPLOYEE RESTRICT
```

より詳しくは、「SQL 解説書」の DROP ステートメントを参照してください。

第 5 章 SELECT ステートメントを使用したデータの検索

SELECT ステートメントを使用して、データベースからデータを検索することができます。本章では、次のセクションを説明します。

- 『SELECT ステートメントを使用したデータの照会』

SQL ステートメントは、1 行に書くことも、複数行に書くこともできます。事前コンパイル済みプログラムの場合、行を継続するときの規則は、ホスト言語 (プログラムを作成する言語) の規則と同じです。

注:

1. このセクションで説明する SQL ステートメントは、SQL 表および視点と、データベースの物理ファイルおよび論理ファイルに対して実行することができます。表、視点、およびファイルは、スキーマまたはライブラリーのどちらにあっても構いません。
2. SQL ステートメントで指定する文字ストリング (たとえば、WHERE または VALUES 文節で使用するもの) は、大文字と小文字が区別されます。すなわち、大文字は大文字で、小文字は小文字で入力しなければなりません。

   ```
   WHERE ADMRDEPT='a00'      (結果を戻しません。)

   WHERE ADMRDEPT='A00'      (有効な部門番号を戻します。)
   ```

   大文字と小文字が同じ文字として扱われる共用重み分類順序が使用されると、比較で大文字と小文字が区別されない場合があります。

SELECT ステートメントを使用したデータの照会

さまざまなステートメントや文節を使用して、データの照会を行うことができます。プログラム内で SELECT ステートメントを使用すると、特定の行 (たとえば、ある社員の行) を取り出すことができます。 さらに、この例では、ある特定の方法でデータを集めるのに、さまざまな文節が使用されています。 SQL は、ユーザーが特定の方法でデータを集めるために照会を調整するいくつかのメソッドを提供しています。 メソッドには、以下のものがあります。

- 69 ページの『WHERE 文節を使用した検索条件の指定』
- 71 ページの『GROUP BY 文節』
- 74 ページの『HAVING 文節』
- 75 ページの『ORDER BY 文節』

以下に示す形式と構文は非常に基本的なものです。 SELECT ステートメントは、この章で示されている例よりも多様化できます。 SELECT ステートメントには、次の項目を指定することができます。

1. 組み込みたい各列の名前
2. データが入っている表または視点の名前
3. 必要な情報が入っている行を固有に識別する検索条件
4. データをグループ分けするために使用される各列の名前

5. 必要な情報が入っているグループを固有に識別する検索条件
6. 重複する行の中の特定の行が返されるようにするための、結果の順序

SELECT ステートメントは次のようになります。

```
SELECT 列名
  FROM 表名または視点名
  WHERE 検索条件
  GROUP BY 列名
  HAVING 検索条件
  ORDER BY 列名
```

SELECT 文節と FROM 文節は、必ず指定する必要があります。その他の文節はオプションです。

SELECT 文節では、取り出したい各列の名前を指定します。 たとえば、次の通りです。

```
SELECT EMPNO, LASTNAME, WORKDEPT
  ⋮
```

1 つまたは複数 (最高 8000 個まで) を取り出すことを指定できます。指定した各列の値は、SELECT 文節で指定した順序で取り出されます。

すべての列を (表の定義に入っているのと同じ順序で) 取り出す場合には、列名を指定する代わりに、アスタリスク (*) を使用してください。

```
SELECT *
  ⋮
```

FROM 文節は、データを選択する元となる *(from)* 表を指定します。複数の表からの列を選択することができます。 SELECT を出す場合、FROM 文節を指定する必要があります。以下のステートメントを出します。

```
SELECT *
       FROM EMPLOYEE
```

結果は、表 EMPLOYEE からのすべての列および行となります。

SELECT リストには、式 (定数、特殊レジスター、およびスカラー副選択を含む) を含めることができます。結果の列に名前を与えるために、AS 文節を使用することもできます。たとえば、以下のようにステートメントを出します。

```
SELECT LASTNAME, SALARY * .05 AS RAISE
       FROM EMPLOYEE
       WHERE EMPNO = '200140'
```

このステートメントの結果は以下のようになります。

表 4. 照会の結果

LASTNAME	RAISE
NATZ	1421

SQL が検索条件を満たす行を見つけることができない場合は、+100 の SQLCODE が返されます。

SQL が選択ステートメントの実行中にエラーを検出すると、負の SQLCODE が返されます。SQL が結果より多いホスト変数を検出したときは、+326 が返されます。

戻されるデータを修飾する方法の詳細については、『WHERE 文節を使用した検索条件の指定』を参照してください。

WHERE 文節を使用した検索条件の指定

WHERE 文節では、取り出し、更新、または削除対象の 1 つまたは複数の行を識別する検索条件を指定します。したがって、1 つの SQL ステートメントで処理する行の数は、WHERE 文節の**検索条件**を満たす行数によって決まります。検索条件は、1 つまたは複数の**述部**によって構成されます。述部は、特定の行または表の複数の行に対して SQL で実行したいテストを指定するものです。 述部の詳細については、82 ページの『複雑な検索条件の実行』を参照してください。

次の例では、WORKDEPT = 'C01' が述部で、WORKDEPT および 'C01' が式、等号 (=) が比較演算子です。文字値はアポストロフィ (') で囲み、数値は囲まないことに注意してください。これは、SQL ステートメントの中でコーディングするすべての定数値に当てはまります。たとえば、部門番号が C01 である行を処理の対象にしたいときは、次のように指定します。

```
... WHERE WORKDEPT = 'C01'
```

この場合、検索条件は 1 つの述部 (WORKDEPT = 'C01') から構成されています。

WHERE をさらに説明するために、SELECT ステートメントに入れます。CORPDATA.DEPARTMENT 表にリストされている各部門が固有の部門番号を持っているとします。 部門 C01 について、CORPDATA.DEPARTMENT 表から部門名と管理者番号を取り出したいとします。 以下のステートメントを出します。

```
SELECT DEPTNAME, MGRNO
       FROM CORPDATA.DEPARTMENT
       WHERE DEPTNO = 'C01'
```

このステートメントが実行されると、次の 1 行が取り出されます。

表 5. 結果表

DEPTNAME	MGRNO
情報センター	000030

検索条件に文字または UCS-2 グラフィック列述部が含まれる場合、それらの述部には照会の実行時に有効な分類順序が適用されます。 分類順序および選択について詳しくは、123 ページの 『第 8 章 SQL での分類順序』を参照してください。 分類順序が使用されない場合は、比較対象となる列または式に一致するように大文字または小文字で文字定数を指定することが必要です。

WHERE の使用について詳しくは、次のセクションを参照してください。

- 70 ページの『WHERE 文節での式』
- 71 ページの『比較演算子』
- 71 ページの『NOT キーワード』

WHERE 文節での式

WHERE 文節における式には、あるものと比較する対象となるものを指定します。それぞれの式は、SQL によって評価されると、文字ストリング、日付/時刻/タイム・スタンプ、または数値となります。指定できる式には、次のものがあります。

- **列名** は、列の名前を指定します。たとえば、次の通りです。

  ```
  ... WHERE EMPNO = '000200'
  ```

  EMPNO は 6 バイトの文字値として定義された列名です。文字データに対しては、同等比較 (すなわち、X = Y または X <> Y) を実行することができます。文字データに関しては、他のタイプの比較を評価することもできます。

  ただし、文字ストリングを数値と比較することはできません。また、文字データに対して算術演算を実行することもできません (これは、EMPNO が数値の形を取る文字ストリングであっても同じです)。キャスト関数を使用して、文字および数値データを比較可能な値に変換することができます。日付/時刻値および期間は、加算および減算できます。

- **式** は、加算 (+)、減算 (-)、乗算 (*)、除算 (/)、指数演算 (**)、または連結 (CONCAT または ||) の結果として 1 つの値を得るために使用される 2 つの値を指定します。式のオペランドには、次のものがあります。

  定数
  列
  ホスト変数
  関数から戻される値
  特殊レジスター
  副照会
  別の式

  たとえば、次の通りです。

  ```
  ... WHERE INTEGER(PRENDATE - PRSTDATE) > 100
  ```

  計算の順序を括弧で指定しないと、式は次の順序で評価されます。

  1. 接頭演算子
  2. 指数演算
  3. 乗算、除算、および連結
  4. 加算および減算

  優先順位が同じである演算子は、左から右へ計算されます。

- **定数**は、式のリテラル値を指定します。たとえば、次の通りです。

  ```
  ... WHERE 40000 < SALARY
  ```

  SALARY は、9 桁のパック 10 進数値 (DECIMAL(9,2)) として定義された列名です。これが数値定数 40000 と比較されます。

- **ホスト変数**は、アプリケーション・プログラム内の変数を識別します。たとえば、次の通りです。

  ```
  ... WHERE EMPNO = :EMP
  ```

- **特殊レジスター**は、データベース・マネージャーによって定義される特殊値を識別します。 たとえば、次の通りです。

  ```
  ... WHERE LASTNAME = USER
  ```

- **NULL** 値は、未知の値を持っている条件を指定します。

  ```
  ... WHERE DUE_DATE IS NULL
  ```

- 副照会。 副照会の使用の詳細については、113 ページの『第 7 章 副照会の使用』を参照してください。

検索条件は、算術演算子や比較演算子で区切られた 2 つの列名または定数に限定する必要はありません。 AND や OR で区切った複数の述部を指定する、より複雑な検索条件を作成することができます。検索条件の複雑度に関係なく、それが行に対して実行されると、TRUE または FALSE のどちらかの値をもたらします。真理値には、偽と同じ働きをする *unknown (未知)* という値もあります。すなわち、ある行の値が空である場合、このヌル値は検索条件で指定された値より小さくも、等しくも、大きくもないので、検索の結果として返されません。より複雑な検索条件と述部については、82 ページの『複雑な検索条件の実行』で説明されています。

WHERE 文節を十分に理解するためには、SQL がどのように検索条件と述部を評価し、式の値を比較するかを理解していなければなりません。このトピックについては、SQL 解説書で説明されています。

比較演算子

SQL では、次の比較演算子がサポートされています。

=	等しい
<> または ¬= または !=	等しくない
<	より小さい
>	より大きい
<= または ¬> または !>	より小さいか等しい (より大きくない)
> = または ¬< または !<	より大きいか等しい (より小さくない)

NOT キーワード

述部の前に NOT キーワードを付けると、その述部の値と反対の値が得られます (すなわち、述部が偽なら真で、真なら偽の値が得られます)。 NOT が適用される述部は、それが前置きされる述部だけで、WHERE 文節内のすべての述部に適用されるわけではありません。たとえば、部門 C01 に所属する社員を除くすべての社員を対象とすることを示すには、次のように指定することができます。

```
... WHERE NOT WORKDEPT = 'C01'
```

これは、次のように指定することもできます。

```
... WHERE WORKDEPT <> 'C01'
```

GROUP BY 文節

GROUP BY 文節がない場合、SQL 列関数が実行されると、*1* つ の行が戻されます。GROUP BY を使用すると、関数は各 グループに適用されるので、グループの数と同数の行が戻されます。

GROUP BY 文節を使用すると、個々の行ではなく、行のグループの特性を調べることができます。GROUP BY 文節の指定があると、SQL は、選択された行をグループに分けて、各グループの行が 1 つまたは複数の列または式で合致した値を持つようにします。次に、SQL は各グループを処理して、各グループの結果が 1 行になるようにします。GROUP BY 文節で 1 つまたは複数の列または式を指定して、行をグループ分けすることができます。SELECT ステートメントで指定する項目は行の各グループの特性であって、表または視点の個々の行の特性ではありません。

たとえば、CORPDATA.EMPLOYEE 表に行のセットがいくつかあり、各セットは特定の部門の社員を記述した行から構成されているとします。各部門の社員の平均給与を知りたいときは、次のステートメントを発行することができます。

```
SELECT WORKDEPT, DECIMAL (AVG(SALARY),5,0)
       FROM CORPDATA.EMPLOYEE
       GROUP BY WORKDEPT
```

結果として、各部門につき 1 行ずつ、複数の行が得られます。

WORKDEPT	AVG-SALARY
A00	40850
B01	41250
C01	29722
D11	25147
D21	25668
E01	40175
E11	21020
E21	24086

注:

1. 行をグループ分けすることは、行を順序付けすることではありません。グループ分けを行うと、選択された各行はグループに入れられ、さらに、SQL がそのグループを処理して、グループの特性を導出します。 行を順序付けすると、すべての行が昇順または降順の照合順序で結果表に入れられます。(これを行う方法は、75 ページの 『ORDER BY 文節』で説明されています)。データベース・マネージャーによって選択されるインプリメンテーションによって、結果のグループが順序付けされて出力される場合もあります。
2. GROUP BY 文節で指定した列にヌル値が入っているときは、ヌル値がある行のデータについては 1 行の結果が得られます。
3. 文字または UCS-2 グラフィック列に対してグループ分けが行われる場合、そのグループ分けには照会の実行時に有効な分類順序が適用されます。分類順序および選択について詳しくは、123 ページの 『第 8 章 SQL での分類順序』を参照してください。

GROUP BY を使用する場合、行をグループ分けするために SQL が使用するようにしてほしい列または式をリストします。たとえば、CORPDATA.PROJECT 表で記述されている各主要プロジェクトに従事している社員の数のリストを入手したいとします。次のステートメントを発行することができます。

```
SELECT SUM(PRSTAFF), MAJPROJ
       FROM CORPDATA.PROJECT
       GROUP BY MAJPROJ
```

結果として、会社の現在の主要プロジェクトとそのプロジェクトに従事する社員の数のリストが得られます。

SUM(PRSTAFF)	MAJPROJ
6	AD3100
5	AD3110
10	MA2100
8	MA2110
5	OP1000
4	OP2000
3	OP2010
32.5	?

複数の列または式に基づいて行をグループ分けすることを指定することもできます。たとえば、CORPDATA.EMPLOYEE 表を使用して選択ステートメントを発行すれば、各部門の男性社員と女性社員の平均給与を知ることができます。これを行うには、次のステートメントを発行することができます。

```
SELECT WORKDEPT, SEX, DECIMAL(AVG(SALARY),5,0) AS AVG_WAGES
       FROM CORPDATA.EMPLOYEE
       GROUP BY WORKDEPT, SEX
```

結果は、以下のようになります。

WORKDEPT	SEX	AVG_WAGES
A00	F	49625
A00	M	35000
B01	M	41250
C01	F	29722
D11	F	25817
D11	M	24764
D21	F	26933
D21	M	24720
E01	M	40175
E11	F	22810
E11	M	16545
E21	F	25370
E21	M	23830

この例では WHERE 文節が含まれていないので、SQL は CORPDATA.EMPLOYEE 表のすべての行を調べて処理します。 SQL が各グループの平均 SALARY 値を算出する前に、行がまず部門番号別にグループ分けされ、次に (各部門内で) 性別にグループ分けされます。

HAVING 文節

HAVING 文節を使用すると、GROUP BY 文節に基づいて選択されるグループの検索条件を指定することができます。 HAVING 文節は、その文節の条件を満たすグループだけ を取り出したいことを指定するものです。したがって、HAVING 文節に指定する検索条件は、グループ内の個々の行の特性ではなくて、各グループの特性をテストするものでなければなりません。

HAVING 文節は、GROUP BY 文節に続けて指定し、WHERE 文節に指定できる検索条件と同種の検索条件を含めることができます。さらに、HAVING 文節には列関数も指定することができます。たとえば、各部門の女性の平均給与を取り出したいとします。そのためには、AVG 列関数を使用し、WORKDEPT 別に結果の行をグループ分けし、WHERE 文節に SEX = 'F' を指定します。

選択された部門のすべての女性社員の学歴が 16 (大学卒) 以上の場合に限りこのデータが得られるように指定したい場合は、HAVING 文節を使用します。HAVING 文節は、グループの特性をテストします。この例の場合には、テストはグループの特性である MIN(EDLEVEL) について行われます。

```
SELECT WORKDEPT, DECIMAL(AVG(SALARY),5,0) AS AVG_WAGES, MIN(EDLEVEL) AS MIN_EDUC
       FROM CORPDATA.EMPLOYEE
       WHERE SEX='F'
       GROUP BY WORKDEPT
       HAVING MIN(EDLEVEL)>=16
```

結果は、以下のようになります。

WORKDEPT	AVG_WAGES	MIN_EDUC
A00	49625	18
C01	29722	16
D11	25817	17

HAVING 文節では、AND および OR で結合することにより複数の述部を使用できます。また、検索条件の任意の述部に NOT を使用できます。

注: 列の更新または行の削除を行いたい場合には、DECLARE CURSOR ステートメント内の SELECT ステートメントに GROUP BY 文節や HAVING 文節を含めることはできません。(DECLARE CURSOR ステートメントについては、129 ページの 『第 9 章 カーソルの使用』で説明されています。) これらの文節を使用すると、読み取り専用カーソルとなります。

引き数が列関数でない述部は、WHERE 文節または HAVING 文節のどちらでも指定できます。通常、選択基準は WHERE 文節の中で指定した方が効率的です。これは、WHERE 文節が照会処理の早期の段階で処理されるためです。HAVING の選択は、結果表の事後処理で行われます。

検索条件に文字または UCS-2 グラフィック列を伴う述部が含まれる場合、それらの述部には照会の実行時に有効な分類順序が適用されます。分類順序および選択について詳しくは、123 ページの 『第 8 章 SQL での分類順序』を参照してください。

ORDER BY 文節

ORDER BY 文節を使用すると、選択された行を特定の順序で戻したり、ある列の値または式の値の昇順または降順にソートすることを指定できます。 ORDER BY 文節の使い方は、GROUP BY 文節の場合と同じです。すなわち、行を照合順序で取り出すときに SQL が使用する列または式を指定します。

たとえば、女性社員の名前と部門番号を部門番号のアルファベット順にリストしたいときは、次の選択ステートメントを使用できます。

```
SELECT LASTNAME,WORKDEPT
       FROM CORPDATA.EMPLOYEE
       WHERE SEX='F'
       ORDER BY WORKDEPT
```

結果は、以下のようになります。

LASTNAME	WORKDEPT
HAAS	A00
HEMMINGER	A00
KWAN	C01
QUINTANA	C01
NICHOLLS	C01
NATZ	C01
PIANKA	D11
SCOUTTEN	D11
LUTZ	D11
JOHN	D11
PULASKI	D21
JOHNSON	D21
PEREZ	D21
HENDERSON	E11
SCHNEIDER	E11
SETRIGHT	D11
SCHWARTZ	E11
SPRINGER	E11
WONG	E21

注: ヌル値は最も高い値として順序付けられます。

ORDER BY 文節で指定される列は、SELECT 文節に組み込む必要はありません。たとえば次のステートメントでは、女性社員全員が、給与が最も高額な者を先頭にして戻されます。

```
SELECT LASTNAME,FIRSTNME
       FROM CORPDATA.EMPLOYEE
       WHERE SEX='F'
       ORDER BY SALARY DESC
```

選択リストにおける結果の列に名前を付けるために AS 文節を指定する場合、その名前を ORDER BY 文節に指定することができます。 AS 文節で指定される名前は、選択リストの中で固有でなければなりません。たとえば、アルファベット順にリストしてある社員のフルネームを取り出すには、次の選択ステートメントを使用できます。

```
SELECT LASTNAME CONCAT FIRSTNME AS FULLNAME
  FROM CORPDATA.EMPLOYEE
  ORDER BY FULLNAME
```

この選択ステートメントはオプションであり、以下のように書くこともできます。

```
SELECT LASTNAME CONCAT FIRSTNME
  FROM CORPDATA.EMPLOYEE
  ORDER BY LASTNAME CONCAT FIRSTNME
```

結果を順序付けするための列名を指定する代わりに、番号を使用することもできます。たとえば、ORDER BY 3 は、選択リストで指定された結果表の *3 番目* の列に基づいて結果が順序付けられることを指定します。 順序付け値が名前の付いた列でない場合は、番号を使用して結果表の行を順序付けしてください。

SQL に行を昇順 (ASC) と降順 (DESC) のどちらで照合させるかも指定できます。昇順の照合順序が省略時値です。前述した選択ステートメントでは、SQL は、*FULLNAME* 式が (アルファベット順と数字順で) 最も小さい行を最初に戻し、以下、フルネームがだんだん大きくなるように行を戻します。行をこの名前に基づいて降順の照合順序で順序付けするには、次のように指定します。

```
... ORDER BY FULLNAME DESC
```

GROUP BY と同じように、1 次配列順のほかに、2 次配列順 (または複数レベルの配列順) も指定できます。前の例では、行を最初に部門番号順に配列し、さらに各部門内で社員名順に配列することができます。 これを行うには、次のように指定します。

```
... ORDER BY WORKDEPT, FULLNAME
```

ORDER BY 文節で文字列または UCS-2 グラフィック列を使用すると、これらの列の順序付けは、照会の実行時に有効な分類順序に基づいて行われます。分類順序と順序付けへの影響について詳しくは、123 ページの 『第 8 章 SQL での分類順序』を参照してください。

静的 SELECT ステートメント

静的 SELECT ステートメント (SQL プログラムに組み込まれた SELECT ステートメント) の場合、INTO 文節を FROM 文節より前に指定する必要があります。INTO 文節には、ホスト変数 (取り出された列値を入れておくためにプログラム内で使用される変数) の名前を指定します。 SELECT 文節で最初に指定した列の値は、INTO 文節で最初に指定したホスト変数に入ります。 2 番目の列の値は 2 番目のホスト変数に入り、以下同様です。

SELECT INTO の結果表には、1 行しか入りません。たとえば、CORPDATA.EMPLOYEE 表の各行は固有の EMPNO (社員番号) 列を持っています。したがって、WHERE 文節に EMPNO 列への同等比較が入っている場合、この表に対する SELECT INTO ステートメントの実行結果は、1 行だけ (または 0 行)

になります。複数の行が見つかったときは、エラーとなりますが、1 行が返されます。このエラー条件のもとでどの行が返されるかは、ORDER BY 文節を指定することによって制御できます。ORDER BY 文節を使用すると、結果表内の最初の行が返されます。

SELECT INTO ステートメントの実行結果として複数の行が返されるようにしたい場合は、DECLARE CURSOR ステートメントを使用して行を選択した後、FETCH ステートメントを使用して、一度に 1 つまたは複数の行単位で列値をホスト変数に移動してください。カーソルの使い方は、 129 ページの『第 9 章 カーソルの使用』で説明されています。

選択ステートメントをアプリケーション・プログラムの中で使用するときは、プログラムにおけるデータの独立性を高めるために、列名をリストするようにしてください。これには 2 つの理由があります。

1. ソース・コード・ステートメントを見るときに、SELECT 文節内の列名と INTO 文節に指定されたホスト変数との 1 対 1 の対応を簡単に確認することができます。
2. ユーザーがアクセスする表または視点に列が追加され、『SELECT * ...,』 を使用し、さらにソースからプログラムを再び作成する場合、INTO 文節には、その新しい列のために指定された対応するホスト変数がありません。余分な列があると、SQLCA で警告 (エラーではない) を受け取ります (SQLWARN4 には『W』 が入ります)。

行内の列値が存在しないことを示すためのヌル値

NULL 値 とは、ある行の列値が存在しないことを意味します。 ヌル値は、ゼロまたはすべてブランクの値とは異なります。ヌル値は、『未知』 と同じです。ヌル値は、WHERE および HAVING 文節の中で条件として使用しても、数学的引き数として使用しても構いません。たとえば、WHERE 文節では、ある行についてヌル値を含んでいる列を指定することができます。通常、ヌル値が入った列を使用する比較述部は、その列にヌル値のある行を選択しません。これは、ヌル値が条件に指定された値より小さくもなく、等しくもなく、大きくもないためです。管理者番号にヌル値の入っているすべての行について値を選択するには、次のように指定してください。

```
SELECT DEPTNO, DEPTNAME, ADMRDEPT
  FROM CORPDATA.DEPARTMENT
  WHERE MGRNO IS NULL
```

結果は次のようになります。

DEPTNO	DEPTNAME	ADMRDEPT
D01	開発センター	A00
F22	BRANCH OFFICE F2	E01
G22	BRANCH OFFICE G2	E01
H22	BRANCH OFFICE H2	E01
I22	BRANCH OFFICE I2	E01
J22	BRANCH OFFICE J2	E01

管理者番号にヌル値の入っていない行を取り出すには、WHERE 文節を次のように変更できます。

```
WHERE MGRNO IS NOT NULL
```

ヌル値の使用に関する詳細については、SQL 解説書を参照してください。

SQL ステートメント内の特殊レジスター

一部の 『特殊レジスター』 は SQL ステートメントの中に指定できます。*ローカル* で実行される SQL ステートメントの場合、特殊レジスターとその内容は、次の表に示すとおりです。

特殊レジスター	内容
CURRENT DATE CURRENT_DATE	現在の日付。
CURRENT PATH CURRENT_PATH CURRENT FUNCTION PATH	動的準備済み SQL ステートメントにおける、非修飾のデータ・タイプ名、プロシージャー名、および関数名を解決するのに使用される SQL パス。
CURRENT SCHEMA	動的準備済み SQL ステートメントにおいて適用できる、非修飾のデータベース・オブジェクト参照を修飾するために使用される、スキーマ名。
CURRENT SERVER CURRENT_SERVER	現在使用中のリレーショナル・データベースの名前。
CURRENT TIME CURRENT_TIME	現在の時刻。
CURRENT TIMESTAMP CURRENT_TIMESTAMP	時刻スタンプ形式の現在の日付と時刻。
CURRENT TIMEZONE CURRENT_TIMEZONE	次の式を用いて現地時間を世界標準時 (UTC) に結び付ける時間の長さ。 **現地時間 -** `CURRENT TIMEZONE = UTC` これは、システム値 QUTCOFFSET から取られます。
USER	ジョブの実行時権限 ID (ユーザー・プロファイル)。

1 つのステートメントに CURRENT DATE、CURRENT TIME、または CURRENT TIMESTAMP 特殊レジスター、あるいは CURDATE、CURTIME、または NOW スカラー関数への複数の参照が含まれる場合、すべての値は単一の時刻機構の読み取り値に基づきます。

遠隔 で行われる SQL ステートメントの場合、特殊レジスターとその内容は、次の表に示すとおりです。

特殊レジスター	内容
CURRENT DATE CURRENT_DATE CURRENT TIME CURRENT_TIME CURRENT TIMESTAMP CURRENT_TIMESTAMP	ローカル・システム側ではなく、遠隔システム側の現在の日付と時刻。
CURRENT TIMEZONE CURRENT_TIMEZONE	遠隔システム側の時刻を UTC に結び付ける時間の長さ。
CURRENT SERVER CURRENT_SERVER	現在使用中のリレーショナル・データベースの名前。
CURRENT SCHEMA	遠隔システムの現行スキーマ値。
USER	遠隔システム側のサーバー・ジョブの実行時権限 ID。
CURRENT PATH CURRENT_PATH CURRENT FUNCTION PATH	遠隔システムの現行パス値。

分散表に対する照会で特殊レジスターが参照されると、照会を要求したシステムの特殊レジスターの内容が使用されます。分散表の詳細については、DB2 マルチシステムを参照してください。

データ・タイプのキャスト

データ・タイプのタイプ を、異なるデータ・タイプに、あるいは同じデータ・タイプでも異なる長さ、精度、スケールを持つように、キャストまたは変更する必要がある場合があります。たとえば文字と整数など、タイプの異なる 2 つの列を比較したい場合、文字を正数に変更するかまたは整数を文字に変更して、比較を行えるようにすることができます。別のデータ・タイプに変更できるデータ・タイプは、ソース・データ・タイプからターゲット・データ・タイプへとキャスト*可能* です。

キャスト関数または CAST 仕様を使用して、データ・タイプを別のデータ・タイプに明示的にキャストすることができます。たとえば、DATE として定義された日付の列 (BIRTHDATE) があり、この列のデータ・タイプを固定長 10 桁の CHARACTER にキャストしたい場合、次のように入力します。

```
SELECT CHAR (BIRTHDATE,USA)
   FROM CORPDATA.EMPLOYEE
```

CAST 関数を使用して、データ・タイプを直接キャストすることもできます。

```
SELECT CAST(BIRTHDATE AS CHAR(10))
   FROM CORPDATA.EMPLOYEE
```

データ・タイプのキャストについて詳しくは、「SQL 解説書」のデータ・タイプ間のキャストを参照してください。

日付、時刻、および時刻スタンプのデータ・タイプ

日付、時刻、および時刻スタンプは、内部形式で表されたデータ・タイプであるため、SQL ユーザーには見えません。日付、時刻、および時刻スタンプは、文字ストリング値で表現して、文字ストリング変数に割り当てることができます。データベース・マネージャーは、次のものを日付、時刻、および時刻スタンプとして認識します。

- DATE、TIME、または TIMESTAMP スカラー関数によって返された値。
- CURRENT DATE、CURRENT TIME、または CURRENT TIMESTAMP 特殊レジスターによって返された値。
- 算術式または比較の一方のオペランドであり、かつ 他方のオペランドが日付、時刻、または時刻スタンプであるときの文字ストリング。たとえば、述部が次のようになっている場合、

  ```
  ... WHERE HIREDATE < '1950-01-01'
  ```

  HIREDATE が日付列ならば、文字ストリング '1950-01-01' は日付と解釈されます。
- UPDATE ステートメントの SET 文節または INSERT ステートメントの VALUES 文節で日付、時刻、または時刻スタンプ列を設定するために使用された文字ストリング変数または定数。

文字ストリング形式の日付値、時刻値、および時刻スタンプ値の詳細については、SQL 解説書の「日時値」を参照してください。

以下のトピックも参照してください。

- 『現在日付値および現在時刻値の指定』
- 81 ページの『日付/時刻演算』

現在日付値および現在時刻値の指定

現在の日付、時刻、または時刻スタンプは、CURRENT DATE、CURRENT TIME、または CURRENT TIMESTAMP の 3 つの特殊レジスターの 1 つを指定することによって、式の中で指定できます。各特殊レジスターの値は、ステートメントの実行時の時刻機構の読み取り値から得られます。同じ SQL ステートメントの中で CURRENT DATE、CURRENT TIME、または CURRENT TIMESTAMP を複数参照すると、同じ値が使用されます。次のステートメントを実行すると、そのステートメントの実行時の EMPLOYEE 表に入っている各社員の年齢 (年単位) が返されます。

```
SELECT YEAR(CURRENT DATE - BIRTHDATE)
  FROM CORPDATA.EMPLOYEE
```

CURRENT TIMEZONE 特殊レジスターを使用すると、現地時間を世界標準時 (UTC) に変換することができます。 たとえば、DATETIME という名前の表に、STARTT という名前の時刻タイプの列が含まれていて、STARTT を UTC に変換したい場合には、次のステートメントが使用できます。

```
SELECT STARTT - CURRENT TIMEZONE
  FROM DATETIME
```

日付/時刻演算

日付、時刻、および時刻スタンプには、加算と減算の算術演算子だけが適用されます。日付、時刻、および時刻スタンプは、期間単位で増分または減分することができます。日付から日付を、時刻から時刻を、時刻スタンプから時刻スタンプを減算することもできます。日付および時刻演算の詳細については、SQL 解説書の日時の演算を参照してください。

重複行の防止

SQL によって選択ステートメントが評価されると、その選択ステートメントの検索条件を満たす複数の行が結果表に入る資格を得ることがあります。結果表の一部の行が重複する可能性もあります。DISTINCT キーワードの後に列名のリストを付けて使用すると、行の重複が起こらないように指定することができます。

```
SELECT DISTINCT JOB, SEX
...
```

DISTINCT は、固有の行だけ選択することを意味します。選択された行が結果表内の別の行と重複している場合には、重複する行は無視されます (これは結果表には入りません)。たとえば、社員の職種コードのリストが必要であるとします。どの社員がどの職種コードを持っているかは知らなくてもよいとします。部門の何人かの社員が同じ職種コードを持っている可能性があるので、DISTINCT を使用すれば、結果表に固有の値だけが入るようにすることができます。

次の例は、その方法を示しています。

```
SELECT DISTINCT JOB
    FROM CORPDATA.EMPLOYEE
    WHERE WORKDEPT = 'D11'
```

結果は以下の 2 行になります。

JOB
DESIGNER
MANAGER

SELECT 文節に DISTINCT がないと、結果に重複行が含まれることがあります。これは、検索条件を満たすすべての行から *JOB* 列の値が返されるためです。 ヌル値は、DISTINCT では重複行として扱われます。

SELECT 文節で DISTINCT とともに共用重み分類順序を使用すれば、返される値をより少なくすることができます。この分類順序を使用すれば、同じ文字を含んでいる値は、大文字小文字にかかわらず同等と見なされます。 'MGR'、'Mgr'、および 'mgr' がすべて同じ表にある場合、これらの値の 1 つだけが返されます。 分類順序および選択について詳しくは、123 ページの 『第 8 章 SQL での分類順序』を参照してください。

複雑な検索条件の実行

検索条件には、基本的な比較述部 (=、>、< など) に加え、
BETWEEN、IN、EXISTS、IS NULL、および LIKE のどのキーワードでも含めることができます。 検索条件には副照会を含めることもできます。詳細について、および例については、 113 ページの『第 7 章 副照会の使用』を参照してください。

文字および UCS-2 グラフィック列述部の場合、BETWEEN、IN、EXISTS、および LIKE 文節の述部が評価される前に、オペランドに分類順序が適用されます。選択とともに分類順序を使用する方法の詳細については、123 ページの 『第 8 章 SQL での分類順序』を参照してください。

複数の検索条件を実行することもできます。詳しくは、 84 ページの『WHERE 文節内の複数の検索条件』を参照してください。

- **BETWEEN ... AND ...** は、2 つの値の間にある値 (両端の値も含む) によって満たされる検索条件を指定するために使用します。たとえば、1987 年に雇用されたすべての社員を調べるには、次のようにすることができます。

  ```
  ...   WHERE HIREDATE BETWEEN '1987-01-01' AND '1987-12-31'
  ```

  BETWEEN キーワードには、両端の値が含まれます。同じ結果をもたらす次の検索条件は、より複雑ではあるものの、明確に条件を指定します。

  ```
  ... WHERE HIREDATE >= '1987-01-01' AND HIREDATE <= '1987-12-31'
  ```

- **IN** は、指定された式の値がリストされた値のどれかに該当する行が処理の対象となることを指定します。たとえば、部門 A00、C01、および E21 のすべての社員の名前を調べるには、次のように指定することができます。

  ```
  ... WHERE WORKDEPT IN ('A00', 'C01', 'E21')
  ```

- **EXISTS** は、ある行が存在するかどうかをテストしたいときに指定します。たとえば、給与が 60000 より大きい社員がいるかどうかを調べたい場合には、次のように指定することができます。

  ```
  EXISTS (SELECT * FROM EMPLOYEE WHERE SALARY > 60000)
  ```

- **IS NULL** は、ヌル値をテストしたいときに指定します。 たとえば、電話番号がない社員がいるかどうかを調べたい場合には、次のように指定することができます。

  ```
  ... WHERE EMPLOYEE.PHONE IS NULL
  ```

- **LIKE** は、ある列値がユーザー指定の値に類似している行が処理の対象となることを指定します。 LIKE を使用すると、SQL は、指定した文字ストリングに類似した文字ストリングを検索します。類似の度合いは、検索条件に含めたストリングに使用されている 2 つの特殊文字により決まります。

  _ 下線は、任意の 1 文字を表します。

  % パーセント記号は、0 文字、1 文字、または 2 文字以上の未知のストリングを表します。検索ストリングの前にパーセント記号を付ける場合、SQL は、列内で検索ストリングと一致する値の前に 1 つ以上の文字が入っていても (あるいは入っていなくても)、その値を一致と見なします。そうでない場合は、検索ストリングが列の 1 桁目から始まっていなければ、一致とは見なされません。

注: MIXED データを処理する場合には、次の相違があります。SBCS の下線文字は 1 つの SBCS 文字を参照します。パーセント記号にはこのような制限はありません。すなわち、パーセント記号は任意の数の SBCS または DBCS 文字を参照します。LIKE 述部と MIXED データの詳細については、SQL 解説書を参照してください。

下線文字またはパーセント記号は、列の値の一部の文字しか知らない場合、あるいは他の文字は関係ない場合に使用してください。たとえば、Minneapolis (ミネアポリス) に住んでいる社員を調べたい場合には、次のように指定することができます。

```
... WHERE ADDRESS LIKE '%MINNEAPOLIS%'
```

SQL は、ADDRESS 列に MINNEAPOLIS というストリングが含まれていれば、そのストリングがどこに置かれているかに関係なく、当該の行をすべて戻します。

別の例として、名前が 'SAN' で始まる町のリストを入手するには、次のように指定できます。

```
... WHERE TOWN LIKE 'SAN%'
```

アドレスを探したいが、マスターのストリート名リストにそのストリート名がない場合は、LIKE 式で式を使用することができます。次の例では、表内の STREET 列は大文字と想定されます。

```
... WHERE UCASE (:address_variable) NOT LIKE '%'||STREET||'%'
```

下線またはパーセント文字のいずれかが含まれる文字ストリングを検索したい場合は、ESCAPE 文節を使用してエスケープ文字を指定してください。たとえば、名前にパーセントの入っている業務を表示するには、次のように指定することができます。

```
... WHERE BUSINESS_NAME LIKE '%@%%' ESCAPE '@'
```

最初と最後のパーセント文字は通常通り解釈されます。'@%' の組み合わせは、パーセント文字として解釈されます。 詳細については、『LIKE に関する特殊な考慮事項』を参照してください。

すべての述部のリストについては、「SQL 解説書」の中の述部を参照してください。

LIKE に関する特殊な考慮事項

- 検索パターンでストリング定数の代わりにホスト変数を使用する場合は、可変長ホスト変数の使用を検討する必要があります。このようなホスト変数を使用すると、次のことが可能になります。
  - 以前に使用されたストリング定数を、変更を加えずにホスト変数に割り当てる。
  - ストリング定数を使用した場合と同じ選択基準および結果を得る。
- 検索パターンでストリング定数の代わりに固定長ホスト変数を使用する場合は、ホスト変数に指定された値が、以前にストリング定数で使用されたパターンと一致していることを確認する必要があります。 ホスト変数内の文字のうち、値が割り当てられていないものは、すべてブランクで初期設定されます。

たとえば、***可変長***ホスト変数の中のストリング・パターン 'ABC%' を用いて検索を行うとすると、返される値としては次のようなものが考えられます。

```
'ABCD      ' 'ABCDE'    'ABCxxx'      'ABC '
```

しかし、*固定長*が 10 のホスト変数に含まれる検索パターン 'ABC%' を用いて検索を行うとすると、列の長さが 12 だと想定した上で返される値としては次のようなものが考えられます。

```
'ABCDE       ' 'ABCD        ' 'ABCxxx      ' 'ABC         '
```

返される値のすべては 'ABC' で始まり、最低 6 つのブランクで終わることに注意してください。 これは、ホスト変数の中の最後の 6 文字に特定の値が割り当てられていなかったためにブランクが使用されたものです。

固定長ホスト変数で検索を行いたい場合に最後の 7 文字は何でも構わないのであれば、'ABC%%%%%%%' を検索することになります。 この場合、返される値としては次のようなものが考えられます。

```
'ABCDEFGHIJ'   'ABCXXXXXXX'  'ABCDE'      'ABCDD'
```

WHERE 文節内の複数の検索条件

69 ページの『WHERE 文節を使用した検索条件の指定』では、単一の検索条件の使用法を示しました。いくつかの述部を含む検索条件をコーディングすると、要求をさらに限定することができます。 指定する検索条件には、任意の比較演算子、または BETWEEN、IN、LIKE、EXISTS、IS NULL、および IS NOT NULL のどのキーワードでも含めることができます。

2 つの述部を AND と OR の結合子を使って結合することができます。 さらに、NOT キーワードを使用すると、必要とする検索条件を、指定した検索条件の否定値にすることを指定できます。WHERE 文節には、必要なだけの述部が指定できます。

- **AND** は、条件を限定したい行について、その行が検索条件の両方の述部を満たさなければならないことを示します。たとえば、部門 D21 の社員のうちで、1987 年 12 月 31 日以後に雇用された社員を調べるには、次のように指定します。

  ```
  ...
    WHERE WORKDEPT = 'D21' AND HIREDATE > '1987-12-31'
  ```

- **OR** は、条件を限定したい行について、その行が検索条件の述部の一方または両方で設定された条件を満たさなければならないことを示しますたとえば、部門 C01 または D11 のいずれかに所属する社員を調べたい場合には、次のように指定することができます。

  ```
  ...
    WHERE WORKDEPT = 'C01' OR WORKDEPT = 'D11'
  ```

  注: IN を用いて、この要求を次のように指定することもできます。 WHERE WORKDEPT IN ('C01', 'D11')

- **NOT** は、条件を限定したい行について、その行が NOT の後に置かれた検索条件または述部で設定された基準を満たしてはならないことを示します。たとえば、部門 E11 の社員のうちで、職種コードがアナリストの社員を除くすべての社員を調べるには、次のように指定できます。

```
...
  WHERE WORKDEPT = 'E11' AND NOT JOB = 'ANALYST'
```

これらの結合子を含む検索条件を評価するとき、SQL は特定の順序でその評価を行います。SQL は最初に NOT 文節を評価し、次に AND 文節を評価し、次に OR 文節を評価します。

評価の順序は括弧を使用することで変更できます。括弧で囲んだ検索条件が最初に評価されます。たとえば、部門 E11 と E21 の社員のうちで、学歴が 12 より上のすべての社員を選択するには、次のように指定できます。

```
...
  WHERE EDLEVEL > 12 AND
     (WORKDEPT = 'E11' OR WORKDEPT = 'E21')
```

括弧は、検索条件の意味を決定します。この例では、次の条件に一致するすべての行を必要としています。

WORKDEPT (部門番号) の値が E11 または E21 であり、かつ

EDLEVEL (学歴) の値が 12 より大きい。

括弧を使用しないで、次のように指定すると、

```
...
  WHERE EDLEVEL > 12 AND WORKDEPT = 'E11'
    OR WORKDEPT = 'E21'
```

異なった結果が得られます。選択される行は、次の条件を満たす行です。

WORKDEPT = E11 で、かつ EDLEVEL > 12、または

WORKDEPT = E21 で、EDLEVEL の値は任意

複数の表からのデータの結合

見たい情報が 1 つの表だけに入っていない場合もあります。ある表からいくつかの列値を取り出し、他の表からいくつかの列値を取り出して、結果表の 1 つの行を形成したい場合もあります。2 つ以上の表から列値を取り出し、1 つの行に結合することができます。

DB2 UDB for iSeries では、さまざまなタイプの結合、すなわち、内部結合、左方外部結合、右方外部結合、左方例外結合、右方例外結合、およびクロス結合がサポートされます。

- 86 ページの『内部結合』は、各表から、結合列内に一致する値が入っている行のみを返します。表間の一致がない行は、結果表には入れられません。
- 87 ページの『左方外部結合』は、最初の表 (左にある表) からのすべての行の値と、2 番目の表からの一致する行の値を返します。 2 番目の表内の一致がない行は、2 番目の表からのすべての列に対してヌル値を返します。
- 88 ページの『右方外部結合』は、2 番目の表 (右にある表) からのすべての行の値と、最初の表からの一致する行の値を返します。最初の表内の一致がない行は、最初の表からのすべての列に対してヌル値を返します。
- 左方例外結合は、右側の表に一致を持たない、左側の表からの行のみを返します。結果表内の、右側の表からの列はヌル値になります。

- 右方例外結合は、左側の表に一致を持たない、右側の表からの行のみを返します。結果表内の、左側の表からの列はヌル値になります。
- 89 ページの『クロス結合』は、結合される各表からの各行の組み合わせにつき 1 つの行を結果表に返します (カルテシアン積)。

左方外部結合と右方例外結合を使用して、全外部結合をシミュレートすることができます。詳細については、 90 ページの『全外部結合のシミュレート』を参照してください。 さらに、単一のステートメントに複数の結合タイプを使用することができます。詳細については、 90 ページの『1 つのステートメントでの複数の結合タイプ』を参照してください。

結合に関する注意事項

2 つ以上の表を結合するときは、次の点に注意してください。

- 共通の列名があるときには、各共通名を表名 (または相関名) で修飾しなくてはなりません。固有の列名を修飾する必要はありません。
- 必要な列名をリストしないで、SELECT * を使用した場合は、最初の表のすべての列から成る行、2 番目の表のすべての列から成る行 (以下同様) という順序で、SQL から行が返されます。
- FROM 文節 に指定する各表または視点から行を選択するためには、その権限が必要です。
- 分類順序は、結合されるすべての文字および UCS-2 グラフィック列に適用されます。

内部結合

内部結合では、表のある行からの列値が、別の (または同じ) 表の別の行からの列値と組み合わされて、1 行のデータが形成されます。SQL は、結合用に指定された両方の表を調べて、結合の検索条件に合致するすべての行からデータを取り出します。内部結合を指定する方法は 2 つあります。すなわち、JOIN 構文の使用と、WHERE 文節の使用です。

あるプロジェクトを担当しているすべての社員の社員番号、名前、およびプロジェクト番号を取り出したいとします。言い換えれば、CORPDATA.EMPLOYEE 表の *EMPNO* および *LASTNAME* 列と、CORPDATA.PROJECT 表の *PROJNO* 列が必要です。 ここでは、姓が 'S' または 'S' 以降のアルファベットで始まる社員のみを考慮したいとします。 この情報を見つけるには、2 つの表を結合する必要があります。

JOIN 構文を使用する内部結合

内部結合構文を使用するには、表に適用される結合条件とともに、結合しようとする両方の表を FROM 文節で指定します。結合条件は、ON キーワードの後ろに指定され、結合結果を生むために 2 つの表が相互に比較される方法を判別します。条件は、どの比較演算子でも可能であり、等号演算子である必要はありません。AND キーワードで区切れば、ON 文節で複数の結合条件を指定することができます。実際の結合とは関連しないその他の条件は、WHERE 文節内または ON 文節内の実際の結合の一部として指定されます。

```
SELECT EMPNO, LASTNAME, PROJNO
  FROM CORPDATA.EMPLOYEE INNER JOIN CORPDATA.PROJECT
       ON EMPNO = RESPEMP
  WHERE LASTNAME > 'S'
```

この例では、結合は 2 つの表に基づき、これらの表からの EMPNO および RESPEMP 列を用いて行われます。姓が少なくとも 'S' で始まる社員のみが戻されるので、この追加条件は WHERE 文節で提供されます。

この照会では、次の出力が戻されます。

EMPNO	LASTNAME	PROJNO
000250	SMITH	AD3112
000060	STERN	MA2110
000100	SPENSER	OP2010
000020	THOMPSON	PL2100

WHERE 文節を使用する内部結合

WHERE 文節を使用してこの同じ結合を実行するには、WHERE 文節で結合条件と追加の選択条件の両方を指定します。結合される表は、FROM 文節でリストされ、コンマによって区切られます。

```
SELECT EMPNO, LASTNAME, PROJNO
  FROM CORPDATA.EMPLOYEE, CORPDATA.PROJECT
  WHERE EMPNO = RESPEMP
  AND LASTNAME > 'S'
```

この照会では、前の例と同じ出力が戻されます。

左方外部結合

左方外部結合は、内部結合で戻されるすべての行に加えて、他の各行 (2 番目の表で値を持たない行と最初の表の行が結合された行) を戻します。

すべての社員と、その社員らが現在担当しているプロジェクトを調べたいとします。現在プロジェクトを担当していない社員も調べたいとします。次の照会では、'S' 以降のアルファベットで始まる名前を持つすべての社員のリストが、その社員らに割り当てられたプロジェクト番号とともに戻されます。

```
SELECT EMPNO, LASTNAME, PROJNO
  FROM CORPDATA.EMPLOYEE LEFT OUTER JOIN CORPDATA.PROJECT
       ON EMPNO = RESPEMP
  WHERE LASTNAME > 'S'
```

この照会の結果には、プロジェクト番号を持たない社員も含まれます。それらの社員は照会でリストされますが、そのプロジェクト番号についてはヌル値が戻されます。

EMPNO	LASTNAME	PROJNO
000020	THOMPSON	PL2100
000060	STERN	MA2110
000100	SPENSER	OP2010
000170	YOSHIMURA	-

EMPNO	LASTNAME	PROJNO
000180	SCOUTTEN	-
000190	WALKER	-
000250	SMITH	AD3112
000280	SCHNEIDER	-
000300	SMITH	-
000310	SETRIGHT	-
200170	YAMAMOTO	-
200280	SCHWARTZ	-
200310	SPRINGER	-
200330	WONG	-

注

左方外部結合または例外結合で、右側の表内の列に相対レコード番号を返すために RRN スカラー関数を使用すると、不一致行については 0 の値が返されます。

右方外部結合

右方外部結合は、内部結合で戻されるすべての行に加えて、最初の表に一致が見付からなかった、2 番目の表の他行の各行を戻します。これは、逆の順序で指定された表を持つ左方外部結合と同じです。

左方外部結合の例として使用された照会は、次の例のように、右方外部結合として作成し直すことができます。

```
SELECT EMPNO, LASTNAME, PROJNO
  FROM CORPDATA.PROJECT RIGHT OUTER JOIN CORPDATA.EMPLOYEE
       ON EMPNO = RESPEMP
  WHERE LASTNAME > 'S'
```

この照会の結果は、左方外部結合の照会の結果と同じです。

例外結合

左方例外結合は、2 番目の表に一致を持たない、最初の表からの行だけを返します。 前の例と同じ表を使用すると、どのプロジェクトも担当していない社員が戻されます。

```
SELECT EMPNO, LASTNAME, PROJNO
  FROM CORPDATA.EMPLOYEE EXCEPTION JOIN CORPDATA.PROJECT
       ON EMPNO = RESPEMP
  WHERE LASTNAME > 'S'
```

この結合では、次の出力が戻されます。

EMPNO	LASTNAME	PROJNO
000170	YOSHIMURA	-
000180	SCOUTTEN	-
000190	WALKER	-
000280	SCHNEIDER	-
000300	SMITH	-

EMPNO	LASTNAME	PROJNO
000310	SETRIGHT	-
200170	YAMAMOTO	-
200280	SCHWARTZ	-
200310	SPRINGER	-
200330	WONG	-

例外結合は、NOT EXISTS 述部を使用して、副照会として書くこともできます。前の照会は、以下のように書き換えることもできます。

```
SELECT EMPNO, LASTNAME
  FROM CORPDATA.EMPLOYEE
  WHERE LASTNAME > 'S'
    AND NOT EXISTS
       (SELECT * FROM CORPDATA.PROJECT
          WHERE EMPNO = RESPEMP)
```

この照会における唯一の違いは、PROJECT 表からの値を戻すことができないという点です。

左方例外結合とまったく同じように働く (ただし、逆になった表で) 右方例外結合もあります。

クロス結合

クロス結合 (あるいはカルテシアン積結合) は、最初の表からの各行が 2 番目の表からの各行と組み合わされる結果表を戻します。結果表内の行の数は、各表の行の数の積です。関与している表が大きい場合、この結合には非常に長い時間がかかります。

クロス結合は、2 つの方法で指定することができます。すなわち、JOIN 構文を使用する方法と、結合基準を指定するために WHERE 文節を使用せずに、FROM 文節でコンマによって区切った表をリストする方法です。

以下の表が存在すると仮定します。

表 6. 表 A

ACOL1	ACOL2
A1	AA1
A2	AA2
A3	AA3

表 7. 表 B

BCOL1	BCOL2
B1	BB1
B2	BB2

以下の 2 つの選択ステートメントは同じ結果をもたらします。

```
SELECT * FROM A CROSS JOIN B
```

```
SELECT * FROM A, B
```

これらの選択ステートメントの結果表は、どちらも以下のようになります。

ACOL1	ACOL2	BCOL1	BCOL2
A1	AA1	B1	BB1
A1	AA1	B2	BB2
A2	AA2	B1	BB1
A2	AA2	B2	BB2
A3	AA3	B1	BB1
A3	AA3	B2	BB2

全外部結合のシミュレート

左方外部結合および右方外部結合と同様に、全外部結合は双方の表から一致する行を戻します。ただし全外部結合では、不一致行も、左方の表と右方の表の双方から戻します。 DB2 UDB for iSeries は全外部結合構文をサポートしませんが、左方外部結合と右方例外結合を使用することにより、全外部結合をシミュレートすることができます。すべての社員、すべてのプロジェクトを、調べたいとします。現在プロジェクトを担当していない社員も調べたいとします。次の照会では、'S' 以降のアルファベットで始まる名前を持つすべての社員のリストが、その社員らに割り当てられたプロジェクト番号とともに戻されます。

```
SELECT EMPNO, LASTNAME, PROJNO
     FROM CORPDATA.EMPLOYEE LEFT OUTER JOIN CORPDATA.PROJECT
           ON EMPNO = RESPEMP
     WHERE LASTNAME > 'S'
  UNION
  (SELECT EMPNO, LASTNAME, PROJNO
     FROM CORPDATA.PROJECT EXCEPTION JOIN CORPDATA.EMPLOYEE
           ON EMPNO = RESPEMP
     WHERE LASTNAME > 'S');
```

1 つのステートメントでの複数の結合タイプ

希望する結果を得るために 3 つ以上の表を結合することが必要な場合があります。全社員、社員の部門名、および社員が担当しているプロジェクト (もしあれば) を戻したい場合には、情報を得るために EMPLOYEE 表、DEPARTMENT 表、および PROJECT 表をすべて結合することが必要です。以下に、照会と結果の例を示します。

```
SELECT EMPNO, LASTNAME, DEPTNAME, PROJNO
  FROM CORPDATA.EMPLOYEE INNER JOIN CORPDATA.DEPARTMENT
        ON WORKDEPT = DEPTNO
      LEFT OUTER JOIN CORPDATA.PROJECT
        ON EMPNO = RESPEMP
  WHERE LASTNAME > 'S'
```

この照会の結果は以下のようになります。

EMPNO	LASTNAME	DEPTNAME	PROJNO
000020	THOMPSON	計画	PL2100
000060	STERN	製造システム	MA2110
000100	SPENSER	ソフトウェア・サポート	OP2010

EMPNO	LASTNAME	DEPTNAME	PROJNO
000170	YOSHIMURA	製造システム	-
000180	SCOUTTEN	製造システム	-
000190	WALKER	製造システム	-
000250	SMITH	管理システム	AD3112
000280	SCHNEIDER	業務部	-
000300	SMITH	業務部	-
000310	SETRIGHT	業務部	-

結合の詳細については、SQL 解説書 を参照してください。

表式の使用

表式を使用すれば、中間結果表を指定することができます。表式は、視点の代わりに表式を使用して、視点の一般使用が不要なときに視点の作成を避けることができます。 表式は、ネストされた表式 (派生表式とも呼ばれる) および共通表式から成っています。

ネストされた表式は、FROM 文節の括弧内に指定されます。たとえば、管理者番号、部門番号、各部門の最高給与を示す結果表を求めるものとします。管理者番号は DEPARTMENT 表にあり、部門番号は DEPARTMENT 表と EMPLOYEE 表の両方にあり、給与は EMPLOYEE 表にあるとします。表式を FROM 文節内で使用すれば、各部門の最高給与を選択することができます。 ネストされた表式のあとに相関名 T2 を追加し、得られた表に名前を付けることもできます。 次に、外部選択は T2 を使用して、得られた表から選択された列 (このケースでは MAXSAL および WORKDEPT) を修飾します。 ネストされた表式の中で選択された MAX(SALARY) 列は、外部選択の中で参照されるように名前を付けねばならないことに注意してください。 これは、AS 文節を使用して行うことができます。

```
SELECT MGRNO, T1.DEPTNO, MAXSAL
FROM CORPDATA.DEPARTMENT T1,
      (SELECT MAX(SALARY) AS MAXSAL, WORKDEPT
       FROM CORPDATA.EMPLOYEE E1
       GROUP BY WORKDEPT) T2
WHERE T1.DEPTNO = T2.WORKDEPT
ORDER BY DEPTNO
```

照会の結果は以下のようになります。

MGRNO	DEPTNO	MAXSAL
000010	A00	52750.00
000020	B01	41250.00
000030	C01	38250.00
000060	D11	32250.00
000070	D21	36170.00
000050	E01	40175.00
000090	E11	29750.00
000100	E21	26150.00

共通表式は、SELECT ステートメント、INSERT ステートメント、または、CREATE VIEW ステートメント内の全選択の前に指定できます。共通表式は、同じ結果表を全選択内で共用する必要があるときに使用できます。共通表式の前には、キーワード WITH が付きます。

たとえば、ある一組の部門の平均給与の最低と最高を示す表が必要であるとします。部門番号の最初の文字が意味を持ち、文字 'D' で始まる部門と文字 'E' で始まる部門の最低値と最高値を求めたいものとします。共通表式を使用すれば、各部門の平均給与が選択できます。ここでも同様に、得られた表に名前を付ける必要があります。このケースでは、名前は DT です。次に、WHERE 文節を使用して SELECT ステートメントを指定すれば、ある文字で始まる部門のみに選択を制限することができます。得られた表 DT の列 AVGSAL の最低値と最高値を指定してください。UNION を指定すれば、文字 'E' の結果および文字 'D' の結果が得られます。

```
WITH DT AS (SELECT E.WORKDEPT AS DEPTNO, AVG(SALARY) AS AVGSAL
            FROM CORPDATA.DEPARTMENT D , CORPDATA.EMPLOYEE E
            WHERE D.DEPTNO = E.WORKDEPT
            GROUP BY E.WORKDEPT)
 SELECT 'E', MAX(AVGSAL), MIN(AVGSAL) FROM DT
 WHERE DEPTNO LIKE 'E%'
 UNION
 SELECT 'D', MAX(AVGSAL), MIN(AVGSAL) FROM DT
 WHERE DEPTNO LIKE 'D%'
```

照会の結果は以下のようになります。

	MAX(AVGSAL)	MIN(AVGSAL)
E	40175.00	21020.00
D	25668.57	25147.27

品目 'XXX' を合わせて注文した顧客からの、最新の 1000 件の受注における、合計受注数量の上位 5 品目を戻す照会を、受注データベースに対して作成したいとします。

```
WITH X AS (SELECT ORDER_ID, CUST_ID
                  FROM ORDERS
                  ORDER BY ORD_DATE DESC
                  FETCH FIRST 1000 ROWS ONLY),
     Y AS (SELECT CUST_ID, LINE_ID, ORDER_QTY
                  FROM X, ORDERLINE
                  WHERE X.ORDER_ID = ORDERLINE.ORDER_ID)
SELECT LINE_ID
     FROM (SELECT LINE_ID
                  FROM Y
                  WHERE Y.CUST_ID IN (SELECT DISTINCT CUST_ID
                                             FROM Y
                                             WHERE LINE.ID = 'XXX' )
                  GROUP BY LINE_ID
                  ORDER BY SUM(ORDER_QTY) DESC)
   FETCH FIRST 5 ROWS ONLY
```

最初の共通表式 (X) は、最新の 1000 件の受注番号を戻します。結果は日付の降順に並べられ、次に、並べられた行の最初の 1000 件のみが、結果表として戻されます。

2 番目の共通表式 (Y) は、最新の 1000 件の受注を品目表と結合し、1000 の受注のそれぞれに対し、その受注の顧客、品目、および品目の数量を戻します。

主選択ステートメントで得られた表は、品目 XXX を注文し最新の 1000 件の受注の中にある顧客に対する品目を戻します。 XXX を注文したすべての顧客の結果は、品目によってグループ分けされ、それらのグループが品目の合計数量順に順序付けられます。

最後に外部選択は、得られた表から戻された順序付きリストから、最初の 5 列だけを選択します。

副選択を結合するための UNION キーワードの使用

UNION キーワードを使用すると、2 つ以上の副選択を結合して全選択にすることができます。 SQL が UNION キーワードを見つけると、各副選択を処理して中間結果表を作り、次に、各副選択の中間結果表を結合し、重複する行を削除して結合結果表を作成します。 UNION を使用すると、2 つ以上の表からの値のリストを組み合わせることができます。 選択ステートメントをコーディングする際には、これまでに学んだ文節と技法をどれでも使用することができます。

複数の表から取り出した値のリストを組み合わせるとき、UNION を使用すると、重複を取り除くことができます。たとえば、次のような社員の社員番号の結合リストを入手することができます。

- 部門 D11 の社員
- プロジェクト MA2112、MA2113、および AD3111 が割り当てられている社員

この結合リストは 2 つの表から導出され、重複する行を含みません。これを行うには、次のように指定します。

```
SELECT EMPNO
   FROM CORPDATA.EMPLOYEE
   WHERE WORKDEPT = 'D11'
 UNION
 SELECT EMPNO
   FROM CORPDATA.EMPPROJACT
   WHERE PROJNO = 'MA2112' OR
      PROJNO = 'MA2113' OR
      PROJNO = 'AD3111'
 ORDER BY EMPNO
```

共通表式、ネストされた表式、および視点の作成の際も、UNION を使用できます。詳細については、62 ページの『UNION を使用した視点の作成』を参照してください。

これらの SQL ステートメントからどのような結果が得られるかを分かりやすく示すために、SQL の処理過程を次に示します。

ステップ 1. SQL は最初の SELECT ステートメントを処理します。

```
SELECT EMPNO
     FROM CORPDATA.EMPLOYEE
     WHERE WORKDEPT = 'D11'
```

この結果、中間結果表が作成されます。

EMPNO from CORPDATA.EMPLOYEE
000060
000150
000160
000170
000180
000190
000200
000210
000220
200170
200220

ステップ 2. SQL は 2 番目の SELECT ステートメントを処理します。

```
SELECT EMPNO
     FROM CORPDATA.EMPPROJACT
     WHERE PROJNO='MA2112' OR
                 PROJNO= 'MA2113' OR
                 PROJNO= 'AD3111'
```

この結果、もう 1 つの中間結果表が作成されます。

EMPNO from CORPDATA.EMPPROJACT
000230
000230
000240
000230
000230
000240
000230
000150
000170
000190
000170
000190
000150
000160
000180
000170
000210
000210

ステップ 3. SQL は、2 つの中間結果表を結合し、重複行を除去し、結果を順序付けます。

```
SELECT EMPNO
     FROM CORPDATA.EMPLOYEE
     WHERE WORKDEPT = 'D11'
UNION
SELECT EMPNO
     FROM CORPDATA.EMPPROJACT
     WHERE PROJNO='MA2112' OR
                 PROJNO= 'MA2113' OR
                 PROJNO= 'AD3111'
ORDER BY EMPNO
```

この結果、結合された結果表が作成され、値が昇順に並びます。

EMPNO
000060
000150
000160
000170
000180
000190
000200
000210
000220
000230
000240
200170
200220

UNION を使用する場合は、次の点に注意してください。

- どの ORDER BY 文節も、UNION の一部の最後の副選択の後に置かなければなりません。 上の例では、結果は最初に選択された列 (*EMPNO*) を基準にして順序付けられています。 ORDER BY 文節は、結合結果表を照合順序にするように指定しています。 ORDER BY は、視点においては許されません。
- 結果の列に名前が付いている場合は、ORDER BY 文節で名前を指定できます。結合された各選択ステートメントに対応する列に同じ名前が付いている場合は、結果の列に名前が付けられます。AS 文節を使用すると、選択リスト内の列に名前を割り当てることができます。

  ```
  SELECT A + B AS X ...
  UNION
  SELECT X ... ORDER BY X
  ```

  結果の列に名前が付いていない場合は、正整数を使用して結果の列を順序付けしてください。 番号は、副選択に含める式のリストにおける式の位置を表します。

  ```
  SELECT A + B ...
  UNION
  SELECT X ... ORDER BY 1
  ```

各行がどの副選択からのものであるかを識別するために、UNION の各副選択の選択リストの最後に定数を含めることができます。 SQL から結果が返されるときに、最後の列にその行の取り出し元である副選択を示す定数が入ります。たとえば、次のように指定することができます。

```
SELECT A, B, 'A1' ...
UNION
SELECT X, Y, 'B2'...
```

行が戻されたときに、その行には、その行の値の取り出し元である表を示す値 (A1 か B2 のどちらか) が入っています。 結合する列名が異なっている場合には、SQL は、対話式 SQL で結果を表示または印刷するとき、あるいは SQL の DESCRIBE ステートメントの処理結果を SQLDA で戻すとき、最初の副選択に指定されている列名のセットを使用します。

UNION における列の長さおよびデータ・タイプの互換性に関する詳細については、SQL 解説書 の結果のデータ・タイプに関する規則を参照してください。

注: 分類順序は、UNION の各コンポーネント間のフィールドの互換性が確立された後で適用されます。分類順序は、UNION の処理中に暗黙に生じる個別の処理で使用されます。 分類順序の詳細については、123 ページの『第 8 章 SQL での分類順序』を参照してください。

UNION ALL の指定

UNION の結果で重複を残したいときは、UNION ではなく UNION ALL を指定してください。使用方法は UNION のステップおよび例と同じです。

ステップ 3. SQL は、2 つの中間結果表を結合します。

```
SELECT EMPNO
     FROM CORPDATA.EMPLOYEE
     WHERE WORKDEPT = 'D11'
 UNION ALL
SELECT EMPNO
     FROM CORPDATA.EMPPROJACT
     WHERE PROJNO='MA2112' OR
                 PROJNO= 'MA2113' OR
                 PROJNO= 'AD3111'
   ORDER BY EMPNO
```

重複行が含まれ、順序付けられた結果表が作成されます。

EMPNO
000060
000150
000150
000150
000160
000160
000170
000170
000170
000170

EMPNO
000180
000180
000190
000190
000190
000200
000210
000210
000210
000220
000230
000230
000230
000230
000230
000240
000240
200170
200220

UNION ALL 演算は結合型です。たとえば、次のステートメントは、

```
(SELECT PROJNO FROM CORPDATA.PROJECT
UNION ALL
SELECT PROJNO FROM CORPDATA.PROJECT)
UNION ALL
SELECT PROJNO FROM CORPDATA.EMPPROJACT
```

このステートメントは、以下のように書くこともできます。

```
SELECT PROJNO FROM CORPDATA.PROJECT
UNION ALL
(SELECT PROJNO FROM CORPDATA.PROJECT
UNION ALL
SELECT PROJNO FROM CORPDATA.EMPPROJACT)
```

ただし、UNION 演算子と同じ SQL ステートメントの中に UNION ALL を含めたときは、演算の結果は評価の順序によって異なります。括弧が付いていないときは、評価は左から右に向かって行われます。括弧が付いているときは、括弧で囲まれた副選択が先に評価され、次にステートメントの他の部分が左から右に向かって評価されます。

データ取り出しエラー

SQL は、取り出された文字またはグラフィック列が長すぎてホスト変数に収まらないことを検出すると、次のことを行います。

- 値をホスト変数に割り当てるときにデータを切り捨てる。
- SQLCA の SQLWARN0 および SQLWARN1 を値 'W' に設定する。

- 標識変数 (提供されている場合) を切り捨て前の値の長さに設定する。

SQL がステートメントの実行中にデータ・マッピング・エラーを検出すると、次の 2 つのどちらかが行われます。

- エラーが SELECT リスト内の式で起こり、エラーを起こした式のために標識変数が提供されている場合。
  - SQL は、エラーを起こした式に対応する標識変数に -2 を返します。
  - SQL は、その行についてのすべての有効なデータを返します。
  - SQL は、正の SQLCODE を返します。
- 標識変数が提供されていない場合、SQL は、対応する負の SQLCODE を SQLCA に入れて返します。

データ・マッピング・エラーには、次のものがあります。

- +138 - サブストリング関数の引き数が有効でない。
- +180 - 日付、時刻、またはタイム・スタンプのストリング表記の構文が有効でない。
- +181 - 日付、時刻、またはタイム・スタンプのストリング表記が有効な値でない。
- +183 - 日付/時刻式からの結果が正しくない。結果の日付またはタイム・スタンプが日付またはタイム・スタンプの有効な範囲内にない。
- +191 - MIXED データが正しい形式になっていない。
- +304 - 数値変換エラー (たとえば、オーバーフロー、アンダーフロー、またはゼロによる除算)。
- +331 - 文字が変換できない。
- +420 - CAST 引き数内の文字が有効でない。
- +802 - データ変換エラーまたはデータ・マッピング・エラー。

データ・マッピング・エラーの場合、SQLCA は最後に検出されたエラーだけを報告します。エラーのある結果列のそれぞれに対応する標識変数は、-2 に設定されます。

全選択の選択リストに DISTINCT が含まれていて、選択リスト内のある列に無効な数値データが入っている場合、そのデータはヌル値と同等であると見なされます (照会が分類として完了する場合)。既存の索引を使用する場合、データはヌル値と同等であるとは見なされません。

ORDER BY 文節に関するデータ・マッピング・エラーの影響は、状況によって異なります。

- SELECT INTO ステートメントまたは FETCH ステートメントでホスト変数にデータが割り当てられているときにデータ・マッピング・エラーが発生し、その同じ式が ORDER BY 文節で使用されている場合には、結果のレコードは式の値に基づいて順序付けられます。レコードは、それがヌル値 (他のどの値よりも大きい) であるかのようには順序付けられません。これは、ホスト変数への割り当てが試みられる前に式が評価されたためです。
- 選択リスト内の式を評価しているときにデータ・マッピング・エラーが発生し、その同じ式が ORDER BY 文節で使用されている場合には、結果の列は、通常、

それがヌル値 (他のどの値よりも大きい) であるかのように順序付けられます。ORDER BY 文節が分類を用いて導入される場合、結果の列は、それがヌル値であるかのように順序付けられます。ORDER BY 文節が、以下のケースのように既存の索引を用いて導入される場合、結果の列は、索引内の式の実際の値に基づいて順序付けられます。

- 式が日付列 (日付形式 *MDY、*DMY、*YMD、または *JUL) であり、日付が日付の有効範囲内にないために日付変換エラーが起こった場合。
- 式が文字列であり、文字が変換できなかった場合。
- 式が 10 進数列であり、有効でない数値が検出された場合。

第 6 章 SQL 挿入、更新、および削除

このセクションでは、表および視点内のデータの更新、削除、または挿入を行うための基本的な SQL ステートメントおよび文節について説明します。 使用する SQL ステートメントは、UPDATE、DELETE、および INSERT です。 SQL アプリケーションの作成に役立つように、上記の SQL ステートメントの使用例が提供されています。 SQL ステートメントの詳しい構文とパラメーターの説明は、SQL 解説書に記載されています。

注:

1. このセクションで説明する SQL ステートメントは、SQL 表および視点と、データベースの物理ファイルおよび論理ファイルに対して実行することができます。表、視点、およびファイルは、スキーマまたはライブラリーのどちらにあっても構いません。
2. SQL ステートメントで指定する文字ストリング (たとえば、WHERE または VALUES 文節で使用するもの) は、大文字と小文字が区別されます。すなわち、大文字は大文字で、小文字は小文字で入力しなければなりません。

   ```
   WHERE ADMRDEPT='a00'     (結果を戻しません。)

   WHERE ADMRDEPT='A00'     (有効な部門番号を戻します。)
   ```

   大文字と小文字が同じ文字として扱われる共用重み分類順序が使用されると、文字ストリングで大文字と小文字が区別されない場合があります。 分類順序の詳細については、123 ページの 『第 8 章 SQL での分類順序』を参照してください。

この章には次のセクションがあります。

INSERT ステートメントを使用した行の挿入

次のいずれかの方法で、INSERT ステートメントを使用して表または視点に新規の行を追加することができます。

- 追加される 1 つの行の列について、INSERT ステートメントで値を指定する。
- INSERT ステートメントに選択ステートメントを組み込んで、別の表または視点内のどのデータを新しい行に入れるかを SQL に指示する。 104 ページの『選択ステートメントによる表への行の挿入』では、INSERT ステートメント内で選択ステートメントを使用して、表に 0 行、1 行、または 2 行以上を追加する方法が説明されています。
- INSERT ステートメントのブロック化形式を指定して、複数の行を追加する。INSERT ステートメントのブロック化形式を使って表に複数の行を追加する方法については、 105 ページの『ブロック化 INSERT ステートメントを使用した複数行の表への挿入』で説明します。

注: 視点は表を基礎として作成され、実際にはデータが入っていないので、視点を用いる作業に関して混乱が生じる可能性もあります。視点を使用してデータを挿入する方法について詳しくは、 60 ページの『視点の作成および使用』を参照してください。

INSERT の詳細な説明については、「SQL 解説書」の INSERT ステートメントを参照してください。

挿入する各行について、NOT NULL 属性が定義されている各列に値を提供しなければなりません (その列に省略時値がない場合)。表または視点に行を追加するための INSERT ステートメントは、次のようになります。

```
INSERT INTO 表名
   (列 1, 列 2, ... )
VALUES (列 1 の値, 列 2 の値, ... )
```

INTO 文節には、値を指定する列の名前を指定します。 VALUES 文節には、INTO 文節に指定した各列の値を指定します。 値には、次のいずれかを指定できます。

定数。 VALUES 文節で提供される値を挿入します。

ヌル値。 キーワード NULL を使用して、ヌル値を挿入します。列は、ヌル値可能として定義されていなければなりません。そうしなければ、エラーが発生します。

ホスト変数。 ホスト変数の内容を挿入します。

特殊レジスター。 特殊レジスター値、たとえば USER を挿入します。

式。 式から得られた値を挿入します。

副照会は、実行中の選択ステートメントの結果の値を挿入します。

DEFAULT キーワード。列のデフォルト値を挿入します。列は、その列用に定義された省略時値を持つか、またはヌル値可能でなければなりません。そうでなければ、エラーが発生します。

INSERT ステートメントの列リストに名前を指定した各列について、VALUES 文節で値を指定しなければなりません。表内のすべての列が、VALUES 文節で指定する値を持つ場合には、列名リストを省略することができます。ある列に省略時値が入る場合には、VALUES 文節内の値としてキーワード DEFAULT を使用することができます。 これにより、その列に省略時値が入れられます。

値を挿入しようとするすべての列の名前を指定することをお勧めします。理由は以下のとおりです。

- INSERT ステートメントが分かりやすくなる。
- 列名に基づいた正しい順序で値を指定していることを確認できる。
- データの独立性が高まる。表内で列が定義されている順序は、INSERT ステートメントには影響しません。

列がヌル値を認めるか、または省略時値を持つように定義されている場合は、列名リストでその名前を指定したり、その値を指定したりする必要はありません。この場合は、省略時値が使用されます。列が省略時値を持つように定義されている場合は、列に省略時値が入ります。明示の省略時値を持たない列定義について DEFAULT が指定されると、SQL はそのデータ・タイプについての省略時値を列に

入れます。列用に定義された省略時値がないが、その列がヌル値を認めるように定義されている (列定義で NOT NULL が指定されていない) 場合には、SQL はその列にヌル値を入れます。

- 数値列の場合、省略時値は 0 です。
- 固定長文字またはグラフィック列の場合、省略時値はブランクです。
- 可変長文字またはグラフィック列または LOB 列の場合、省略時値は長さゼロのストリングです。
- 日付、時刻、およびタイム・スタンプ列の場合、省略時値は現在の日付、時刻、またはタイム・スタンプです。レコードのブロックが挿入される場合には、省略時値の日付/時刻値はブロックの書き込み時にシステムから取り出されます。これは、ブロック内の各行でこの列に同じ省略時値が割り当てられることを意味します。
- データ・リンク列の場合、省略時値は DLVALUE(’’,’URL’,’’) に対応する値になります。
- 特殊タイプ列の場合、省略時値は対応するソース・タイプの省略時値になります。
- ROWID 列、または AS IDENTITY で定義された列の場合、データベース・マネージャーが省略時値を生成します。 105 ページの『識別列への挿入』を参照してください。

表内の既存の行と重複する行をプログラムが挿入しようとすると、エラーが起こることがあります。複数のヌル値は、索引の作成時に使用されたオプションによって、重複する値と見なされる場合と見なされない場合があります。

- 表が基本キー、固有キー、または固有索引を備えているときは、その行は挿入されません。その代わりに、SQL から SQLCODE -803 が戻されます。
- 表が基本キー、固有キー、または固有索引を備えていないときは、その行は正常に挿入され、エラーは起こりません。

SQL が INSERT ステートメントの実行中にエラーを検出すると、データの挿入を中止します。 COMMIT(*ALL)、COMMIT(*CS)、COMMIT(*CHG)、または COMMIT(*RR) が指定されていると、行の挿入は行われません。選択ステートメントを伴う INSERT またはブロック化挿入の場合、このステートメントによってすでに挿入されている行は削除されます。COMMIT(*NONE) が指定されている場合は、すでに挿入された行があっても、削除されません。

SQL によって作成される表は、*YES の削除済みレコード再利用パラメーターを用いて作成されます。これにより、データベース・マネージャーは、表内の削除済みとしてマークされた行を再利用することができます。CHGPF コマンドを使用すると、属性を *NO に変更できます。これを行うと、INSERT では常に表の終わりに行が追加されるようになります。

行を挿入した順序で、それらの行が取り出されるとは限りません。

行がエラーなしで挿入されると、SQLCA の SQLERRD(3) フィールドに 1 が入ります。

注: ブロック化 INSERT の場合、または選択ステートメントを伴う INSERT の場合は、複数の行が挿入される可能性があります。挿入された行の数は SQLERRD(3) に反映されます。

選択ステートメントによる表への行の挿入

INSERT ステートメントの中で選択ステートメントを使用すると、指定する表または視点から選択された 0 行、1 行、または 2 行以上を別の表に挿入することができます。行を選択する表は、挿入先の表と同じにすることができます。それらが同じ表である場合には、SQL は、選択された行を含む一時結果表を作成し、その後一時表からターゲット表へ挿入します。

この種の INSERT ステートメントの使用法の 1 つとして、要約データ用に作成した表にデータを移す場合があります。たとえば、プロジェクトに対する各社員の従事時間を示す表が必要であるとします。 *EMPNUMBER* 列、*PROJNUMBER* 列、*STARTDATE* 列、および *ENDDATE* 列が入った EMPTIME という名前の表を作成し、次の INSERT ステートメントを用いて表にデータを入れることができます。

```
INSERT INTO CORPDATA.EMPTIME
   (EMPNUMBER, PROJNUMBER, STARTDATE, ENDDATE)
SELECT EMPNO, PROJNO, EMSTDATE, EMENDATE
  FROM CORPDATA.EMPPROJACT
```

INSERT ステートメントに組み込む選択ステートメントは、データの取り出しに使用する選択ステートメントと変わりありません。 FOR READ ONLY、FOR UPDATE、または OPTIMIZE 文節を除き、データの取り出しに用いられるすべてのキーワード、列関数、および技法を使用することができます。 SQL は、検索条件を満たすすべての行を指定の表に挿入します。 1 つの表から別の表に行を挿入しても、ソース表の既存の行にもターゲット表の既存の行にも影響はありません。

複数の行の挿入に関する注意事項

表に複数の行を挿入する場合には、次の点に注意してください。

- INSERT ステートメントに暗黙にまたは明示的にリストされた列の数は、選択ステートメントにリストされた列の数と同じでなければなりません。
- 選択ステートメントを指定した INSERT を使用する場合、選択する列のデータは、挿入先の列と互換性がなければなりません。
- INSERT に組み込まれた選択ステートメントから行が戻されない場合には、行が挿入されなかったことをユーザーに警告するために SQLCODE 100 が戻されます。行の挿入が正常に行われた場合には、SQLCA の SQLERRD(3) フィールドに、SQL が実際に挿入した行の数を示す整数が入ります。
- SQL が INSERT ステートメントの実行中にエラーを検出すると、処理を中止します。COMMIT (*CHG)、COMMIT(*CS)、COMMIT (*ALL)、または COMMIT(*RR) の指定があるときは、表には何も挿入されず、負の SQLCODE が返されます。 COMMIT(*NONE) の指定があるときは、エラーの前に挿入された行は表に残っています。
- INSERT ステートメントで選択ステートメントを指定すると、2 つ以上の表を結合することができます。この方法でロードされた表は、行が物理的に保管された行として表に存在するので、UPDATE、DELETE、および INSERT ステートメントを使用して処理することができます。

ブロック化 INSERT ステートメントを使用した複数行の表への挿入

ブロック化 INSERT を使用すると、1 つのステートメントで複数の行を表に挿入することができます。このブロック化 INSERT ステートメントは、REXX を除くすべての言語でサポートされています。表に挿入されるデータは、ホスト構造配列になっていなければなりません。ブロック化 INSERT で標識変数を使用する場合は、その標識変数もホスト構造配列になっていなければなりません。特定の言語のホスト構造配列については、ホスト言語でのSQLプログラミングの該当する言語に関する章を参照してください。

たとえば、10 人の社員を CORPDATA.EMPLOYEE 表に追加する場合は、次のようになります。

```
INSERT INTO CORPDATA.EMPLOYEE
                  (EMPNO,FIRSTNME,MIDINIT,LASTNAME,WORKDEPT)
            10 ROWS VALUES(:DSTRUCT:ISTRUCT)
```

DSTRUCT は、プログラムで宣言された、5 つの要素を持つホスト構造配列です。この 5 つの要素は、EMPNO、FIRSTNME、MIDINIT、LASTNAME、および WORKDEPT に対応しています。 DSTRUCT は、10 行の挿入を受け入れるために、少なくとも 10 のディメンションを持ちます。ISTRUCT は、プログラムで宣言されたホスト構造配列です。ISTRUCT は、少なくとも標識用の小整数フィールド 10 個分のディメンションを持ちます。

ブロック化 INSERT ステートメントは、非分散 SQL アプリケーションについて、および、アプリケーション・サーバーとアプリケーション・リクエスターの両方が iSeries システムである分散アプリケーションについてサポートされます。

識別列への挿入

ユーザーが識別列に値を挿入したり、あるいはユーザーに代わりシステムに値を挿入させることができます。たとえば、 54 ページの『識別列の作成および変更』で作成された表では、ORDERNO (識別列)、SHIPPED_TO (VARCHAR(36))、および ORDER_DATE (日付) という列があります。この表に、次のステートメントを出すことにより、行を挿入することができます。

```
INSERT INTO ORDERS (SHIPPED_TO, ORDER_DATE)
    VALUES ('BME TOOL', 2002-02-04)
```

この例では、識別列に入れる値をシステムが自動的に生成します。このステートメントを、DEFAULT キーワードを使用して書くこともできます。

```
INSERT INTO ORDERS (SHIPPED_TO, ORDER_DATE, ORDERNO)
    VALUES ('BME TOOL', 2002-02-04, DEFAULT)
```

挿入後、IDENTITY_VAL_LOCAL 関数を使用して、システムが列に割り当てた値を知ることができます。詳細および例については、「SQL 解説書」にある IDENTITY_VAL_LOCAL 関数を参照してください。

次の、SELECT を使用した INSERT ステートメントなど、ユーザーが識別列の値を指定する場合もあります。

```
INSERT INTO ORDERS OVERRIDING USER VALUE
   (SELECT * FROM TODAYS_ORDER)
```

この例では、OVERRIDING USER VALUE は、システムに対し、SELECT から識別列に与えられる値を無視して識別列に新規の値を生成するよう指定します。識別列が GENERATED ALWAYS 文節を指定して作成された場合には、OVERRIDING USER VALUE が使用されることが必要です。GENERATED BY DEFAULT の場合にはオプションです。 GENERATED BY DEFAULT 識別列に OVERRIDING USER VALUE が指定されない場合は、SELECT でその列に提供される値が挿入されます。

OVERRIDING SYSTEM VALUE を指定することにより、システムに、SELECT からの値を GENERATED ALWAYS 識別列に強制的に使用させることができます。たとえば、以下のようにステートメントを出します。

```
INSERT INTO ORDERS OVERRIDING SYSTEM VALUE
   (SELECT * FROM TODAYS_ORDER)
```

この INSERT ステートメントでは SELECT からの値を使用し、識別列用の新規の値は生成しません。 GENERATED ALWAYS を使用して作成された識別列の場合は、OVERRIDING SYSTEM VALUE 文節を使用しなければ値を提供することはできません。

UPDATE ステートメントを使用した表内のデータの変更

表内のデータを変更するには、UPDATE ステートメントを使用します。 UPDATE ステートメントを使用すると、WHERE 文節の検索条件を満たしている各行の 1 つまたは複数の列の値を変更することができます。 UPDATE ステートメントを実行すると、WHERE 文節に指定された検索条件を満たす行の数に応じて、表の 0 個以上の行の 1 つまたは複数の列値が変更されます。UPDATE ステートメントの形式は次のとおりです。

```
UPDATE 表名
  SET 列 1 = 値 1,
      列 2 = 値 2, ...
  WHERE 検索条件 ...
```

たとえば、ある社員が異動になったとします。この異動が反映されるように、CORPDATA.EMPLOYEE 表に入っているその社員のデータのいくつかの項目を更新するためには、次のように指定します。

```
UPDATE CORPDATA.EMPLOYEE
  SET JOB = :PGM-CODE,
      PHONENO = :PGM-PHONE
  WHERE EMPNO = :PGM-SERIAL
```

SET 文節は、更新したい各列についての新しい値を指定するために使用します。SET 文節には、更新したい列名と、変更後の値を指定します。値には、次のいずれかを指定できます。

列名。列の現行値を同じ行内の別の列の内容で置換します。

定数。 列の現行値を、SET 文節に指定した値で置換します。

ヌル値。 キーワード NULL を使用して、列の現行値をヌル値で置換します。列は、表の作成時にヌル値可能として定義されていなければなりません。そうでなければ、エラーが発生します。

ホスト変数。 列の現行値をホスト変数の内容で置換します。

特殊レジスター。 列の現行値を特殊レジスターの値 (たとえば、USER) で置換します。

式。 列の現行値を式の結果の値で置換します。

スカラー副選択。 列の現行値を、副照会で戻した値で置換します。

DEFAULT キーワード。列の現行値を列の省略時値で置換します。列は、その列用に定義された省略時値を持つか、またはヌル値可能でなければなりません。そうでなければ、エラーが発生します。

以下に、多数のさまざまな値を使用するステートメントの例を示します。

```
UPDATE WORKTABLE
   SET COL1 = 'ASC',
       COL2 = NULL,
       COL3 = :FIELD3,
       COL4 = CURRENT TIME,
       COL5 = AMT - 6.00,
       COL6 = COL7
    WHERE EMPNO = :PGM-SERIAL
```

更新される行を識別するには、WHERE 文節を使用します。

- 1 つの行を更新するには、1 つの行だけを選択する WHERE 文節を使用してください。
- 複数の行を更新するには、更新したい行だけを選択する WHERE 文節を使用してください。

WHERE 文節は省略することができます。省略すると、表または視点の各行が、指定した値で更新されます。

データベース・マネージャーが UPDATE ステートメントの実行中にエラーを検出すると、更新を中止して負の SQLCODE を戻します。COMMIT(*ALL)、COMMIT(*CS)、COMMIT(*CHG)、または COMMIT(*RR) が指定されていると、表内のど行も変更されません (このステートメントによってすでに変更された行があれば、以前の値に復元されます)。COMMIT(*NONE) が指定されている場合は、すでに変更された行があっても、以前の値に復元されません。

データベース・マネージャーが検索条件を満たす行を見つけることができない場合は、+100 の SQLCODE が返されます。

注: UPDATE ステートメントでは、複数の行が更新される可能性があります。更新された行の数は、SQLCA の SQLERRD(3) に反映されます。

さまざまな方法で UPDATE ステートメントの SET 文節を使用し、更新中の各行で設定する実際の値を決定することができます。次の例では、各列とそれに対応する値を示します。

```
UPDATE EMPLOYEE
  SET WORKDEPT = 'D11',
      PHONENO = '7213',
      JOB = 'DESIGNER'
  WHERE EMPNO = '000270'
```

すべての列を指定してからすべての値を指定することにより、直前の更新を書き込むこともできます。

```
UPDATE EMPLOYEE
  SET (WORKDEPT, PHONENO, JOB)
      = ('D11', '7213', 'DESIGNER')
  WHERE EMPNO = '000270'
```

表の中のデータを更新する方法の詳細は、次のセクションを参照してください。

- 『スカラー副選択を使用する表の更新』
- 『別の表からの行を使用する表の更新』
- 109 ページの『表のデータの検索と更新』

UPDATE ステートメントの詳細な説明については、「SQL 解説書」の UPDATE を参照してください。

スカラー副選択を使用する表の更新

更新する値 (複数可) を選択する別の方法は、スカラー副選択を使用することです。スカラー副選択を使用すると、1 つ以上の列に、別の表から選択した 1 つ以上の値を設定して、それらの列を更新できるようになります。次の例では、ある社員が別の部門へ異動しますが、同じプロジェクトで作業をします。 社員表はすでに更新され、新しい部門番号が入っています。 ここで、プロジェクト表を更新して、この社員 (社員番号は '000030') の新しい部門番号を反映させる必要があります。

```
UPDATE PROJECT
  SET DEPTNO =
        (SELECT WORKDEPT FROM EMPLOYEE
           WHERE PROJECT.RESPEMP = EMPLOYEE.EMPNO)
  WHERE RESPEMP='000030'
```

この同じ技法を使用し、1 つの選択で返された複数の値を使って列のリストを更新することができます。

別の表からの行を使用する表の更新

さらに、別の表の行からの値を用いて、ある表の行全体を更新することもできます。 マスター・クラス・スケジュール表があり、その表のコピーに加えられた変更によって更新しなければならないとします。変更は作業コピーに加えられ、毎晩マスター表にマージされます。 この 2 つの表には同じ列があり、その 1 つである CLASS_CODE は固有キー列です。

```
UPDATE CL_SCHED
  SET ROW =
       (SELECT * FROM MYCOPY
          WHERE CL_SCHED.CLASS_CODE = MYCOPY.CLASS_CODE)
```

この更新により、MYCOPY からの値で、CL_SCHED 内のすべての行が更新されます。

識別列の更新

識別列の値を、指定した値に更新したり、あるいはシステムに新規の値を生成させることができます。たとえば、 54 ページの『識別列の作成および変更』で作成された、ORDERNO (識別列)、SHIPPED_TO (VARCHAR(36))、および ORDER_DATE (日付) という列を持つ表を使用して、識別列の値を変更することができます。次のステートメントを出します。

```
UPDATE ORDERS
  SET (ORDERNO, ORDER_DATE)=
       (DEFAULT, 2002-02-05)
  WHERE SHIPPED_TO = 'BME TOOL'
```

識別列に入れる値は、システムが自動的に生成します。 OVERRIDING SYSTEM VALUE 文節の使用により、システムに値を生成させることを指定変更できます。

```
UPDATE ORDERS OVERRIDING SYSTEM VALUE
  SET (ORDERNO, ORDER_DATE)=
       (553, '2002-02-05')
  WHERE SHIPPED_TO = 'BME TOOL'
```

表のデータの検索と更新

行のデータは、カーソルを使用することにより、検索と同時に更新することができます。 カーソルの詳細については、 129 ページの『第 9 章 カーソルの使用』を参照してください。選択ステートメントで、FOR UPDATE OF を指定し、その後に更新可能な列のリストを続けます。そして、カーソルにより制御される UPDATE ステートメントを使用してください。更新したい行を指し示すカーソルの名前は、WHERE CURRENT OF 文節で指定します。 FOR UPDATE OF、ORDER BY、FOR READ ONLY、または DYNAMIC 文節を持たない SCROLL 文節を指定しなかった場合は、すべての列が更新できます。

複数行用 FETCH ステートメントを指定して実行した場合には、カーソルはそのブロックの最後の行に置かれています。したがって、UPDATE ステートメントに WHERE CURRENT OF 文節の指定があると、ブロックの最後の行が更新されます。ブロック内のある行を更新しなければならない場合には、プログラムではまずカーソルをその行に移動する必要があります。その後、UPDATE WHERE CURRENT OF を指定することができます。次の例を見てください。

表 8. 表の更新

スクロール可能カーソル用の SQL ステートメント	注釈
EXEC SQL **DECLARE** THISEMP **DYNAMIC SCROLL CURSOR FOR** **SELECT** EMPNO, WORKDEPT, BONUS **FROM** CORPDATA.EMPLOYEE **WHERE** WORKDEPT = 'D11' **FOR UPDATE OF** BONUS END-EXEC.	
EXEC SQL **OPEN** THISEMP END-EXEC.	
EXEC SQL **WHENEVER NOT FOUND** GO TO CLOSE-THISEMP END-EXEC.	
EXEC SQL **FETCH NEXT FROM** THISEMP **FOR** 5 **ROWS** **INTO** :DEPTINFO :IND-ARRAY END-EXEC.	DEPTINFO と IND-ARRAY は、プログラムの中ではホスト構造配列および標識配列として宣言されます。

表 8. 表の更新 (続き)

スクロール可能カーソル用の SQL ステートメント	注釈
... 部門 D11 の中で受け取った賞与の金額が $500.00 未満の社員がいるかどうかを判別する。もしいれば、そのレコードを新たな最低金額 $500.00 として更新する。	
EXEC SQL **FETCH** RELATIVE :NUMBACK **FROM** THISEMP END-EXEC.	... ブロック内の該当レコードに移動し、逆順の検索によって更新する。

EXEC SQL
  UPDATE CORPDATA.EMPLOYEE
  SET BONUS = 500
  WHERE CURRENT OF THISEMP
END-EXEC.
	... 部門 D11 の社員の中で新たな最低金額 $500.00 未満のものの賞与を更新する。

EXEC SQL
  FETCH RELATIVE :NUMBACK **FROM** THISEMP
  FOR 5 **ROWS**
  INTO :DEPTINFO :IND-ARRAY
END-EXEC.
	... すでに検索を行った同じブロックの先頭に移動し、もう一度ブロックを検索する。(NUMBACK -(5 - NUMBACK - 1))

... ブランチで戻り、そのブロック内に賞与が $500.00 未満の社員が他にもいるかどうか判別する。 ... ブランチで戻り、次の行ブロックを検索して処理する。	
CLOSE-THISEMP. EXEC SQL **CLOSE** THISEMP END-EXEC.	

制約事項

下記のいずれかの要素を含んでいる選択ステートメントで FOR UPDATE OF を使用することはできません。

- 最初の FROM 文節で 2 つ以上の表または視点を指定している。
- 最初の FROM 文節で読み取り専用視点を指定している。
- 最初の SELECT 文節でキーワード DISTINCT を指定している。
- 最初の FROM 文節でユーザー定義表関数を指定している。
- 外部副選択が GROUP BY 文節を含んでいる。
- 外部副選択が HAVING 文節を含んでいる。
- 最初の SELECT 文節に列関数が含まれている。
- 選択ステートメントに UNION または UNION ALL 演算子が含まれている。
- 選択ステートメントに ORDER BY 文節を含まれており、FOR UPDATE OF 文節と DYNAMIC SCROLL が指定されていない。
- 選択ステートメントに FOR FETCH ONLY 文節が組み込まれている。
- DYNAMIC を指定せずに、SCROLL キーワードが指定されている。
- 選択リストに データ・リンク列が組み込まれ、FOR UPDATE OF 文節が指定されていない。
- 最初の副選択に一時結果表が必要である。

- 選択ステートメントに *n* ROWS ONLY が組み込まれている。

FOR UPDATE OF 文節を指定した場合には、その FOR UPDATE OF 文節内で指定されていない列を更新することはできません。ただし、次の例のように、SELECT リストに含まれていない列名を FOR UPDATE OF 文節の中で指定することはできます。

```
SELECT A, B, C FROM TABLE
FOR UPDATE OF A,E
```

FOR UPDATE OF 文節では、必要以上の列を指定することのないようにしてください。表にアクセスする場合にはこれらの列の索引は使用されません。

DELETE ステートメントを使用した表からの行の除去

表から行を除去するときは、DELETE ステートメントを使用します。行の DELETE では、その行全体が除去されます。DELETE は、行から特定の列を取り除くためのものではありません。 DELETE ステートメントが実行されると、WHERE 文節で指定された検索条件を満たす行の数に応じて、表の 0 個以上の行が削除されます。DELETE ステートメントで WHERE 文節の指定を省略すると、SQL は表のすべての行を削除します。DELETE ステートメントは次のようになります。

```
DELETE FROM 表名
  WHERE 検索条件 ...
```

たとえば、部門 D11 が別の場所に移転したとします。この場合には、次のように CORPDATA.EMPLOYEE 表内の WORKDEPT に D11 という値が入っているすべての行を削除する必要があります。

```
DELETE FROM CORPDATA.EMPLOYEE
  WHERE WORKDEPT = 'D11'
```

WHERE 文節は、表から削除する行を SQL に指示します。SQL は、検索条件を満たすすべての行を、基礎となる表から削除します。視点から行を削除すると、基礎となる表からの行が削除されます。 WHERE 文節は省略できますが、WHERE 文節のない DELETE ステートメントでは表または視点のすべての行が削除されるため、WHERE 文節を組み込むことが推奨されます。 表の内容とともに表定義を削除するには、DROP ステートメントを実行してください。 DROP ステートメントについて詳しくは、*SQL 解説書* の DROP ステートメントのトピックを参照してください。

SQL が DELETE ステートメントの実行中にエラーを検出すると、データの削除を中止して負の SQLCODE を戻します。COMMIT(*ALL)、COMMIT(*CS)、COMMIT(*CHG)、または COMMIT(*RR) が指定されていると、表内のどの行も削除されません (このステートメントによってすでに削除された行があれば、以前の値に戻されます)。COMMIT(*NONE) が指定されている場合は、すでに削除された行があっても、以前の値に復元されません。

SQL が検索条件を満たす行を見つけることができない場合は、+100 の SQLCODE が返されます。

注: DELETE ステートメントでは、複数の行が削除される可能性があります。 削除された行の数は SQLERRD(3) に反映されます。

DELETE ステートメントについて詳しくは、*SQL 解説書* の DELETE ステートメントのトピックを参照してください。

第 7 章 副照会の使用

データを選択するもう 1 つの方法として、検索条件の中で副照会を使用することができます。副照会は、式、選択リスト、および ORDER BY 文節と GROUP BY 文節の中で、使用することができます。詳細については、以下のセクションを参照してください。

SELECT ステートメントの副照会

単純な WHERE 文節および HAVING 文節では、リテラル値、列名、式、または特殊レジスターを使用して検索条件を指定することができます。これらの検索条件では、ある特定の値を探しますが、場合によっては表から別のデータを取り出さなければ、探している値を提供できないことがあります。 たとえば、特定のプロジェクト (ここではプロジェクト番号 MA2100 とします) に従事しているすべての社員の社員番号、氏名、および職種コードのリストが必要であるとします。ステートメントの最初の部分は、次のように簡単に書くことができます。

```
SELECT EMPNO, LASTNAME, JOB
FROM CORPDATA.EMPLOYEE
WHERE EMPNO ...
```

しかし、CORPDATA.EMPLOYEE 表にプロジェクト番号のデータが含まれていないので、ここから先に進むことができません。CORPDATA.EMP_ACT 表に対して別の SELECT ステートメントを発行しなければ、プロジェクト MA2100 に従事している社員は分かりません。

SQL では、ある SELECT ステートメントを別の SELECT ステートメントの中にネストできるので、この問題は解決します。内部 SELECT ステートメントを**副照会**と呼びます。副照会を囲んでいる SELECT ステートメントを**外部レベル SELECT** と呼びます。副照会を使用すると、SQL ステートメントを 1 つ発行するだけで、プロジェクト MA2100 に従事している社員の社員番号、氏名、および職種コードを取り出すことができます。

```
SELECT EMPNO, LASTNAME, JOB
FROM CORPDATA.EMPLOYEE
WHERE EMPNO IN
   (SELECT EMPNO
      FROM CORPDATA.EMPPROJACT
      WHERE PROJNO = 'MA2100')
```

この SQL ステートメントからどのような結果が得られるかを分かりやすく示すために、SQL の処理過程を次に示します。

ステップ 1: SQL は SUBQUERY を評価して EMPNO 値のリストを入手します。

```
(SELECT EMPNO
     FROM CORPDATA.EMPPROJACT
     WHERE PROJNO= 'MA2100')
```

この結果、中間結果表が作成されます。

CORPDATA.EMPPROJACT からの EMPNO
000010
000110

ステップ 2: そして、この中間結果表が、外部レベル SELECT の検索条件のリストとなります。本質的には、この選択ステートメントこそ実行されるステートメントです。

```
SELECT EMPNO, LASTNAME, JOB
     FROM CORPDATA.EMPLOYEE
     WHERE EMPNO IN
            ('000010', '000110')
```

最終結果は次のようになります。

EMPNO	LASTNAME	JOB
000010	HAAS	PRES
000110	LUCCHESSI	SALESREP

詳細については、以下のセクションを参照してください。

- 『相関』
- 『副照会と検索条件』
- 115 ページの『副照会の使用方法』
- 117 ページの『副照会の使用に関する注意事項』
- 117 ページの『相関副照会』
- 121 ページの『UPDATE ステートメントでの相関副照会の使用』
- 122 ページの『DELETE ステートメントでの相関副照会の使用』

相関

副照会の目的は、単一の行 (WHERE 文節) または行のグループ (HAVING 文節) の述部の評価に必要な情報を、提供することです。 これは、副照会により生成される結果表を通して行われます。 概念的には、副照会は新しい行または行のグループの処理が必要になるたびに評価されます。実際には、副照会がどの行またはグループについても同じものならば、副照会は 1 回しか評価されません。 このような副照会を**非相関**副照会と呼びます。

行ごとに、あるいはグループごとに、異なる値を戻す副照会もあります。このような変化を可能にするメカニズムを**相関**と呼び、このような副照会を**相関**副照会と呼びます。 相関副照会の詳しい説明は、117 ページの『相関副照会』にあります。

副照会と検索条件

副照会は、検索条件の一部となることができます。 検索条件は、*オペランド 演算子 オペランド* という形式です。 副照会はどちらのオペランドでも可能です。次の

例では、第 1 の**オペランド**が EMPNO で、**演算子**が IN です。 検索条件は WHERE 文節または HAVING 文節の一部にすることができます。 この文節には、副照会を含む複数の検索条件を組み込むことができます。 副照会を含む検索条件は、他の検索条件と同様に、括弧で囲んだり、NOT キーワードを前に付けたり、AND キーワードや OR キーワードを用いて別の検索条件にリンクしたりすることができます。 たとえば、照会を含む WHERE 文節は次のようになります。

```
WHERE  (subquery1) = X AND (Y > SOME (subquery2) OR Z = 100)
```

副照会は、他の副照会の検索条件の中に置くこともできます。このような副照会は、あるネスト・レベルで**ネストされた**副照会と呼びます。たとえば、外部レベル SELECT 内の副照会の中の副照会は、ネスト・レベル 2 でネストされていることになります。SQL では、ネスト・レベル 32 までネストできます。

副照会の使用方法

WHERE 文節または HAVING 文節に副照会を組み込む方法はいくつかあります。

- 『基本比較』
- 『限量化比較 (ALL、ANY、および SOME)』
- 116 ページの『IN キーワード』
- 116 ページの『EXISTS キーワード』

基本比較

いずれかの比較演算子の直前または直後に副照会を使用することができます。副照会から返される値は多くても 1 つです。 この値は、列関数または算術式の結果であることもあります。 SQL は、副照会から得た結果の値を、比較演算子の他方の側にある値と比較します。 たとえば、会社全体の平均学歴より高い学歴を持つ社員の社員番号、氏名、および給与を調べたいとします。

```
SELECT EMPNO, LASTNAME, SALARY
FROM CORPDATA.EMPLOYEE
WHERE EDLEVEL >
   (SELECT AVG(EDLEVEL)
      FROM CORPDATA.EMPLOYEE)
```

SQL はまず副照会を実行し、次に、その結果を SELECT ステートメントの WHERE 文節に代入します。この例では、結果は全社の平均学歴です。 副照会は、1 つの値を返すほかに、まったく値を返さないこともあります。 その場合、比較の結果は未知になります。

限量化比較 (ALL、ANY、および SOME)

ALL、ANY、または SOME キーワードが続く比較演算子の後で副照会を使用することができます。副照会をこのように使用する場合、返される値は 0 個、1 個、または複数個であり、この中にはヌル値も含まれます。ALL、ANY、および SOME の使い方は、次のとおりです。

- ALL は、指定した値を、指定した方法で、副照会から返された**すべての値**と比較する必要がある場合に使用します。たとえば、「より大」比較演算子を ALL とともに使用する場合は、次のようになります。

  ```
  ... WHERE 式 > ALL (副照会)
  ```

この WHERE 文節を満たすには、式の値が副照会から返されるすべての値より大きい (すなわち、最大値より大きい) ことが必要です。副照会が空のセットを返した場合 (すなわち、値が 1 つも選択されなかった場合) にも、条件は満たされます。

- ANY または SOME は、指定した値を、指定した方法で、副照会から返される値のうち少なくとも *1* つ と比較する必要がある場合に使用します。たとえば、「より大」比較演算子を **ANY** とともに使用する場合は、次のようになります。

  ```
  ... WHERE 式 > ANY (副照会)
  ```

  この WHERE 文節を満たすには、式の値が副照会から返される値の少なくとも 1 つより大きい (すなわち、最小値より大きい) ことが必要です。副照会によって空のセットが返された場合、条件は満たされません。

注: 副照会から 1 つまたは複数のヌル値が返される場合の結果は、その論理に慣れていないと混乱を招くことがあります。詳細については、SQL 解説書 に記載されている比較述部の説明を参照してください。

IN キーワード

IN を使用すると、式の値が副照会から返される値の範囲に入っていなければならないことを指定できます。 IN を使用することは、*=ANY* または *= SOME* を使用することと同等です。 ANY と SOME の使用法はすでに説明しました。また、値が副照会から返される値の範囲に入っていないときに行を選択するために、IN キーワードを NOT キーワードと併用することもきます。たとえば、次のように指定することができます。

```
... WHERE WORKDEPT NOT IN (SELECT ...)
```

EXISTS キーワード

これまでに説明した副照会では、SQL が副照会を評価し、その結果を外部レベル SELECT の WHERE 文節の一部として使用しています。これとは対照的に、キーワード EXISTS を使用すると、SQL は、副照会が 1 つまたは複数の行を返したかどうかを検査するだけです。返していれば、条件は満たされます。行が 1 つも返されていない場合は、条件は満たされません。 たとえば、次の通りです。

```
SELECT EMPNO,LASTNAME
FROM CORPDATA.EMPLOYEE
WHERE EXISTS
   (SELECT *
      FROM CORPDATA.PROJECT
      WHERE PRSTDATE > '1982-01-01');
```

この例では、CORPDATA.PROJECT 表内に予定開始日が 1982 年 1 月 1 日より遅いプロジェクトがあれば、検索条件が真になります。 この例では、EXISTS の威力が完全には示されていないことに注意してください。これは、外部レベル SELECT について検査されるすべての行で結果が常に同じであるためです。結果として、すべての行が結果に入れられるか、あるいは 1 つも入れられないかのどちらかです。さらに威力を発揮する例では、副照会自体が相関になり、行ごとに変化します。相関副照会の詳細については、117 ページの『相関副照会』を参照してください。

この例で示されているように、EXISTS 文節の副照会の選択リストには列名を指定する必要がありません。 その代わりに、SELECT * をコーディングするべきです。

また、指定したデータまたは条件が存在しないときに行を選択するために、EXISTS キーワードを NOT キーワードと併用することもできます。次のように使用できます。

```
... WHERE NOT EXISTS (SELECT ...)
```

副照会の使用に関する注意事項

1. SELECT ステートメントをネストする場合、要件を満たすのに必要なだけの副照会 (1 ～ 31 個) を使用することができます。ただし、副照会の数が増えるほどパフォーマンスは低下します。
2. 外部ステートメントが SELECT ステートメントの場合には (ネストのレベルに関係なく)、次の点に注意してください。
   - 副照会は、外部ステートメントと同じ表またはビューを基礎にすることも、別の表または視点を基礎にすることもできます。
   - 外部レベル SELECT が DECLARE CURSOR、CREATE TABLE、CREATE VIEW、または INSERT ステートメントの一部であるときも、副照会を外部レベル SELECT の WHERE 文節の中で使用することができます。
   - SELECT ステートメントの HAVING 文節の中で副照会を使用することができます。その場合、SQL は副照会を評価し、それを使用して各グループを限定します。
3. ステートメントが UPDATE または DELETE ステートメントのとき、副照会は、UPDATE ステートメントまたは DELETE ステートメントの WHERE 文節の中で使用することができます。UPDATE ステートメントの SET 文節の中で副照会を使用することもできます。
4. 副照会が UPDATE ステートメントの SET 文節で使用されているとき、副選択の結果表には、対応する更新列のリストと同じ数の値が含まれていなければなりません。その他のすべての場合、副照会を EXISTS キーワードとともに使用する場合を除き、副照会の結果表は 1 つの列で構成されなければなりません。ALL、SOME、または EXISTS キーワードを使用する述部の場合、副照会から戻される行数は 0 から多数までさまざまです。その他のすべての副照会の場合は、戻される行数は必ず 0 または 1 です。
5. 副照会には、ORDER BY、UNION、UNION ALL、FOR READ ONLY、FETCH FIRST *n* ROWS、UPDATE、または OPTIMIZE 文節を含めることはできません。

相関副照会

これまでに説明した副照会では、SQL は副照会を 1 回評価し、副照会の結果を検索条件に代入し、検索条件の値に基づいて外部レベル SELECT を評価しました。外部レベル SELECT で新しい行の検査 (WHERE 文節) または行グループの検査 (HAVING 文節) に移るたびに、SQL が評価し直すような副照会を作成することもできます。 このような副照会を**相関副照会**と呼びます。

相関名と相関参照

相関参照は、副照会の検索条件に置くことができます。この参照は常に X.C の形式です。X は相関名であり、C は X が表す表の列名です。

FROM 文節に現れる各表名ごとに、相関名を定義することができます。相関名は、照会の中で固有な、表の名前を与えます。照会、およびそのネストされた副選択の中で、同一の表名を何度も使用することができます。表を参照するごとに異なる相関名を指定すれば、ある列がどの表を参照しているか、一意的に指定できます。

相関名は、照会の FROM 文節で定義します。 この照会は、外部レベル SELECT であっても、参照が入っている副照会を含む副照会であっても構いません。 たとえば、ある照会に副照会 A、B、および C が含まれており、A に B が、B に C が含まれているとします。この場合、C で使用される相関名は、B、A、または外部レベル SELECT で定義することができます。 相関名を定義するには、単に表名の後に相関名を入れるだけです。表名とその相関名との間に 1 つまたは複数のブランクを置き、さらに別の表名を指定する場合には、その相関名の後にコンマを入れてください。 次の FROM 文節は、TABLEA と TABLEB に対してはそれぞれ相関名 TA と TB を定義していますが、表 TABLEC に対しては相関名を定義していません。

```
FROM TABLEA TA, TABLEC, TABLEB TB
```

相関参照は副照会にいくつでも置くことができます。たとえば、ある検索条件の 1 つの相関名は外部レベル SELECT で指定し、別の相関名はその副照会を内包している副照会で定義することができます。

副照会が実行される前に、参照される列からの値が必ず相関参照に代入されます。

例: WHERE 文節の相関副照会

それぞれの所属部門の平均教育レベルより高い教育レベルを持つすべての社員のリストが必要であるとします。この情報を得るには、SQL は CORPDATA.EMPLOYEE 表を検索しなければなりません。 表内の各社員について、SQL は、その社員の教育レベルをその社員が所属する部門の平均教育レベルと比較する必要があります。副照会では、現在行の部門番号についての平均教育レベルを計算するように SQL に指示します。たとえば、次の通りです。

```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
 FROM CORPDATA.EMPLOYEE X
 WHERE EDLEVEL >
    (SELECT AVG(EDLEVEL)
       FROM CORPDATA.EMPLOYEE
       WHERE WORKDEPT = X.WORKDEPT)
```

相関副照会は、1 つまたは複数の相関参照がある点を除けば、非相関副照会と類似しています。上記の例では、副選択の FROM 文節に X.WORKDEPT が現れていることが 1 つの相関参照です。ここで、修飾子 *X* は、外部 SELECT ステートメントの FROM 文節で定義された相関名です。その FROM 文節では、*X* は表 CORPDATA.EMPLOYEE の相関名として取り入れられています。

CORPDATA.EMPLOYEE のある行に対してこの副照会が実行されるとどうなるかについて、次に検討します。副照会が実行される前に、X.WORKDEPT がある位置は該当の行の WORKDEPT 列の値に置き換えられます。たとえば、該当行が、CHRISTINE I HAAS の行だとします。この社員の所属部門は A00 であり、これがこの行の WORKDEPT の値です。この行に対して実行する副照会は次のとおりです。

```
(SELECT AVG(EDLEVEL)
   FROM CORPDATA.EMPLOYEE
   WHERE WORKDEPT = 'A00')
```

このように、検討している行について副照会を実行すると、Christine の所属部門の平均教育レベルが得られます。これが、外側のステートメントで Christine 自身の教育レベルと比較されます。WORKDEPT の値が異なる別の行の場合は、この値は副照会の A00 の個所に置かれます。たとえば、MICHAEL L THOMPSON の行の場合は、この値は B01 となり、この行についての副照会からは、部門 B01 の平均教育レベルが得られます。

この照会から得られる結果表には、次のような値が入ることになります。

表 9. 前述の照会の結果のセット

EMPNO	LASTNAME	WORKDEPT	EDLEVEL
000010	HAAS	A00	18
000030	KWAN	C01	20
000070	PULASKI	D21	16
000090	HENDERSON	E11	16
000110	LUCCHESSI	A00	19
000160	PIANKA	D11	17
000180	SCOUTTEN	D11	17
000210	JONES	D11	17
000220	LUTZ	D11	18
000240	MARINO	D21	17
000260	JOHNSON	D21	16
000280	SCHNEIDER	E11	17
000320	MEHTA	E21	16
000340	GOUNOT	E21	16
200010	HEMMINGER	A00	18
200220	JOHN	D11	18
200240	MONTEVERDE	D21	17
200280	SCHWARTZ	E11	17
200340	ALONZO	E21	16

例: HAVING 文節の相関副照会

部門の平均給与がその業務分野の平均給与より高い部門をすべてリストしたいと想定します (WORKDEPT が同じ文字で始まる部門はすべて同じ分野に属するものとします)。この情報を得るには、SQL は CORPDATA.EMPLOYEE 表を検索しなければなりません。 表の各部門について、SQL はその部門の平均給与を当該部門の平均給与とします。副照会では、SQL は現行グループの部門の所属分野について平均給与を計算します。たとえば、次の通りです。

```
SELECT WORKDEPT, DECIMAL(AVG(SALARY),8,2)
 FROM CORPDATA.EMPLOYEE X
 GROUP BY WORKDEPT
 HAVING AVG(SALARY) >
```

```
(SELECT AVG(SALARY)
   FROM CORPDATA.EMPLOYEE
   WHERE SUBSTR(X.WORKDEPT,1,1) = SUBSTR(WORKDEPT,1,1))
```

CORPDATA.EMPLOYEE のある部門に対してこの副照会が実行されると、どうなるかについて検討します。副照会が実行される前に、X.WORKDEPT がある位置は該当のグループの WORKDEPT 列の値に置き換えられます。 たとえば、選択された最初のグループの WORKDEPT の値が A00 とします。このグループに対して実行される副照会は次のようになります。

```
(SELECT AVG(SALARY)
   FROM CORPDATA.EMPLOYEE
   WHERE SUBSTR('A00',1,1) = SUBSTR(WORKDEPT,1,1))
```

このように、検討しているグループについて、副照会からその分野の平均給与が得られます。この値が外部ステートメントで部門 'A00' の平均給与と比較されます。WORKDEPT が 'B01' の他のグループの場合は、副照会から部門 B01 の所属分野の平均給与が得られます。

この照会から得られる結果表には、次のような値が入ることになります。

WORKDEPT	AVG SALARY
D21	25668.57
E01	40175.00
E21	24086.66

例: 選択リストの相関副照会

部門名、部門番号、および管理者の名前を組み込んだ、すべての部門のリストが必要であるとします。部門名、部門番号は CORPDATA.DEPARTMENT 表にありますが、DEPARTMENT は管理者の番号しか持っておらず、管理者の名前はありません。各部門の管理者の名前を知るには、DEPARTMENT 表の管理者番号と一致する社員番号を EMPLOYEE 表から検索して、一致する行にある名前を戻す必要があります。現在管理者が割り当てられている部門だけが戻されることになります。次のように実行します。

```
SELECT DEPTNO, DEPTNAME,
    (SELECT FIRSTNME CONCAT ' ' CONCAT
         MIDINIT CONCAT ' ' CONCAT LASTNAME
       FROM EMPLOYEE X
       WHERE X.EMPNO = Y.MGRNO) AS MANAGER_NAME
   FROM DEPARTMENT Y
   WHERE MGRNO IS NOT NULL
```

DEPTNO および DEPTNAME に戻される各行ごとに、システムは EMPNO = MGRNO となるものを探してその管理者名を戻します。この照会から得られる結果表には、次のような値が入ることになります。

表 10.

DEPTNO	DEPTNAME	MANAGER_NAME
A00	SPIFFY コンピューター・サービス事業部	CHRISTINE I HAAS
B01	計画	MICHAEL L THOMPSON

表 10. (続き)

DEPTNO	DEPTNAME	MANAGER_NAME
C01	情報センター	SALLY A KWAN
D11	製造システム	IRVING F STERN
D21	管理システム	EVA D PULASKI
E01	サポート・サービス	JOHN B GEYER
E11	オペレーション	EILEEN W HENDERSON
E21	ソフトウェア・サポート	THEODORE Q SPENSER

UPDATE ステートメントでの相関副照会の使用

UPDATE ステートメントの中で相関副照会を使用するときには、相関名は更新対象の行を表します。たとえば、あるプロジェクトのすべての活動が 1983 年 9 月の前に完了しなければならないときに、部門ではそのプロジェクトを優先プロジェクトと見なすとします。次に示す SQL ステートメントを使用すると、CORPDATA.PROJECT 表内のプロジェクトを評価し、各優先プロジェクトの PRIORITY 列 (この目的のために CORPDATA.PROJECT に追加した列) に 1 (PRIORITY を示すフラグ) を書き込むことができます。

```
UPDATE CORPDATA.PROJECT X
  SET PRIORITY = 1
  WHERE '1983-09-01' >
     (SELECT MAX(EMENDATE)
         FROM CORPDATA.EMPPROJACT
         WHERE PROJNO = X.PROJNO)
```

SQL は、CORPDATA.EMPPROJACT 表の各行を調べるときに、プロジェクト (CORPDATA.PROJECT 表にある) のすべての活動について、最大の活動終了日 (EMENDATE) を判別します。プロジェクトに関連する各活動の終了日が 1983 年 9 月よりも前であれば、CORPDATA.PROJECT 表の現在行は優先プロジェクトに該当し、更新されます。

受注数量に変更があればそれに従ってマスター受注表を更新します。受注表の数量が設定されていない場合 (NULL 値の場合) は、マスター受注表にある値のままにします。

```
UPDATE MASTER_ORDERS X
   SET QTY=(SELECT COALESCE (Y.QTY, X.QTY)
               FROM ORDERS Y
               WHERE X.ORDER_NUM = Y.ORDER_NUM)
   WHERE X.ORDER_NUM IN (SELECT ORDER_NUM
               FROM ORDERS)
```

この例では、MASTER_ORDERS 表の各行は、対応する ORDERS 表の行があるかどうか、検査されます。 ORDERS 表に一致する行があれば、COALESCE 関数が使用されて QTY 列の値が戻されます。 ORDERS 表の QTY が非ヌル値である場合は、その値が MASTER_ORDERS 表の QTY 列の更新に使用されます。 ORDERS 表の QTY が NULL である場合は、MASTER_ORDERS QTY 列自身が持っていた値が使用されて更新されます。

DELETE ステートメントでの相関副照会の使用

DELETE ステートメントの中で相関副照会を使用するときは、その相関名は削除対象の行を表します。 SQL は、DELETE ステートメントに指定された表の各行ごとに一度ずつ相関副照会を評価し、その行を削除するかどうかを判断します。

CORPDATA.PROJECT 表内のある行が削除されたと想定します。この場合、削除されたプロジェクトに関連する CORPDATA.EMPPROJACT 表内の行も削除しなければなりません。これには、次のステートメントを使用できます。

```
DELETE FROM CORPDATA.EMPPROJACT X
  WHERE NOT EXISTS
     (SELECT *
        FROM CORPDATA.PROJECT
        WHERE PROJNO = X.PROJNO)
```

SQL は、CORPDATA.EMP_ACT 表内の各行について、CORPDATA.PROJECT 表内に同じプロジェクト番号を持つ行が存在するかどうかを判別します。それが存在しなければ、CORPDATA.EMP_ACT の行が削除されます。

第 8 章 SQL での分類順序

分類順序は、ある文字セット内の文字が比較または順序付けされるときの、それらの相互関係を定義します。分類順序について詳しくは、SQL 解説書の分類順序のセクションを参照してください。

分類順序は、SQL ステートメントで実行されるすべての文字および UCS-2 グラフィック比較に使用されます。 1 バイトと 2 バイトの文字データ用の分類順序表があります。それぞれの 1 バイトの分類順序表には、対応する 2 バイトの分類順序表があり、逆も同様です。2 つの表の間の変換は、照会を行う必要があるときに実行されます。さらに、CREATE INDEX ステートメントには、索引の中で参照される文字列に適用される分類順序 (ステートメントの実行時に有効になる) があります。

詳細については、以下のセクションを参照してください。

- 『ORDER BY および行選択で使用される分類順序』
- 124 ページの『分類順序と ORDER BY』
- 125 ページの『行選択』
- 126 ページの『分類順序と視点』
- 127 ページの『分類順序と CREATE INDEX ステートメント』
- 127 ページの『分類順序と制約』

ORDER BY および行選択で使用される分類順序

分類順序の使い方を見るために、このセクションでは、以下の表に示した STAFF 表に対してステートメントの例を実行します。 JOB 列の値は、大文字小文字混合であることに注意してください。 'Mgr'、'MGR'、および 'mgr' などの値があります。

表 11. STAFF 表

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	0
20	Pernal	20	Sales	8	18171.25	612.45
30	Merenghi	38	MGR	5	17506.75	0
40	OBrien	38	Sales	6	18006.00	846.55
50	Hanes	15	Mgr	10	20659.80	0
60	Quigley	38	SALES	0	16808.30	650.25
70	Rothman	15	Sales	7	16502.83	1152.00
80	James	20	Clerk	0	13504.60	128.20
90	Koonitz	42	sales	6	18001.75	1386.70
100	Plotz	42	mgr	6	18352.80	0

以下の例では、次の分類順序を使用する各ステートメントの結果を示します。

- *HEX 分類順序

- 言語 ID ENU を使用する共用重み分類順序
- 言語 ID ENU を使用する固有分類順序

注: ENU を言語 ID として選択するには、CRTSQLxxx、STRSQL、または RUNSQLSTM コマンドで SRTSEQ(*LANGIDUNQ)、または SRTSEQ(*LANGIDSHR) と LANGID(ENU) を指定するか、あるいは SET OPTION ステートメントを使用します。

分類順序と ORDER BY

次の SQL ステートメントでは、結果表が JOB 列の値を用いて分類されます。

```
SELECT * FROM STAFF ORDER BY JOB
```

表 12 は、*HEX 分類順序を使用した場合の結果表を示しています。行は、JOB 列内の EBCDIC 値に基づいて分類されています。この場合、小文字はすべて大文字より前に分類されています。

*表 12. "SELECT * FROM STAFF ORDER BY JOB" *HEX 分類順序を使用した場合*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
100	Plotz	42	mgr	6	18352.80	0
90	Koonitz	42	sales	6	18001.75	1386.70
80	James	20	Clerk	0	13504.60	128.20
10	Sanders	20	Mgr	7	18357.50	0
50	Hanes	15	Mgr	10	20659.80	0
30	Merenghi	38	MGR	5	17506.75	0
20	Pernal	20	Sales	8	18171.25	612.45
40	OBrien	38	Sales	6	18006.00	846.55
70	Rothman	15	Sales	7	16502.83	1152.00
60	Quigley	38	SALES	0	16808.30	650.25

表 13 は、固有分類順序を使用した場合に分類がどのように行われるかを示しています。JOB 列内の値に分類順序が適用された後で、行が分類されています。分類後、小文字が同じ文字の大文字の前に置かれ、'mgr'、'Mgr'、および 'MGR' の値が互いに隣接していることに注目してください。

*表 13. "SELECT * FROM STAFF ORDER BY JOB" ENU 言語 ID の固有分類順序を使用した場合*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
80	James	20	Clerk	0	13504.60	128.20
100	Plotz	42	mgr	6	18352.80	0
10	Sanders	20	Mgr	7	18357.50	0
50	Hanes	15	Mgr	10	20659.80	0
30	Merenghi	38	MGR	5	17506.75	0
90	Koonitz	42	sales	6	18001.75	1386.70
20	Pernal	20	Sales	8	18171.25	612.45
40	OBrien	38	Sales	6	18006.00	846.55

*表 13. "SELECT * FROM STAFF ORDER BY JOB" ENU 言語 ID の固有分類順序を使用した場合 (続き)*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
70	Rothman	15	Sales	7	16502.83	1152.00
60	Quigley	38	SALES	0	16808.30	650.25

表 14 は、共用重み分類順序を使用した場合に分類がどのように行われるかを示しています。JOB 列内の値に分類順序が適用された後で、行が分類されています。この分類比較では、各小文字は、対応する大文字と同等として扱われています。 表 14 では、'MGR'、'mgr' および 'Mgr' のすべての値が混在していることに注目してください。

*表 14. "SELECT * FROM STAFF ORDER BY JOB" ENU 言語 ID の共用重み分類順序を使用した場合*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
80	James	20	Clerk	0	13504.60	128.20
10	Sanders	20	Mgr	7	18357.50	0
30	Merenghi	38	MGR	5	17506.75	0
50	Hanes	15	Mgr	10	20659.80	0
100	Plotz	42	mgr	6	18352.80	0
20	Pernal	20	Sales	8	18171.25	612.45
40	OBrien	38	Sales	6	18006.00	846.55
60	Quigley	38	SALES	0	16808.30	650.25
70	Rothman	15	Sales	7	16502.83	1152.00
90	Koonitz	42	sales	6	18001.75	1386.70

行選択

次の SQL ステートメントは、JOB 列に値 'MGR' がある行を選択します。

```
SELECT * FROM STAFF WHERE JOB='MGR'
```

表 15 は、*HEX 分類順序を使用した場合に行選択がどのように行われるかを示しています。 表 15 では、選択ステートメントで指定された列 'JOB' に関する行選択基準と正確に一致する行が選択されています。 すなわち、大文字の 'MGR' のみが選択されています。

*表 15. "SELECT * FROM STAFF WHERE JOB='MGR' " *HEX 分類順序を使用した場合*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
30	Merenghi	38	MGR	5	17506.75	0

126 ページの表 16 は、固有分類順序を使用した場合に行選択がどのように行われるかを示しています。 126 ページの表 16 では、小文字と大文字は別個の文字として扱われています。小文字の 'mgr' と大文字の 'MGR' は同じであるとは見なされません。したがって、小文字の 'mgr' は選択されていません。

*表 16. "SELECT * FROM STAFF WHERE JOB = 'MGR' " ENU 言語 ID の固有分類順序を使用した場合*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
30	Merenghi	38	MGR	5	17506.75	0

表 17 は、共用重み分類順序を使用した場合に行選択がどのように行われるかを示しています。 表 17 では、大文字と小文字を同等として扱うことによって、列 'JOB' に関する行選択基準と一致する行が選択されています。 表 17 では、'mgr'、'Mgr'、および 'MGR' のすべての値が選択されていることに注目してください。

*表 17. "SELECT * FROM STAFF WHERE JOB = 'MGR' " ENU 言語 ID の共用重み分類順序を使用した場合*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	0
30	Merenghi	38	MGR	5	17506.75	0
50	Hanes	15	Mgr	10	20659.80	0
100	Plotz	42	mgr	6	18352.80	0

分類順序と視点

視点は、CREATE VIEW ステートメントの実行時に有効であった分類順序を使用して作成されます。 FROM 文節で視点が参照されると、CREATE VIEW の副選択での文字比較にこの分類順序が使用されます。その時点で、視点の副選択から中間結果表が作成されます。その後、照会で指定されたすべての文字および UCS-2 グラフィック比較 (文字または UCS-2 グラフィックへの暗黙の変換を伴う比較を含む) には、照会の実行時に有効な分類順序が適用されます。

以下の SQL ステートメントと表では、視点と分類順序が機能する仕組みを示します。以下の例で使用される視点 V1 は、SRTSEQ(*LANGIDSHR) および LANGID(ENU) の共用重み分類順序を使用して作成されています。CREATE VIEW ステートメントは次のようになります。

```
CREATE VIEW V1 AS SELECT *
  FROM STAFF
  WHERE JOB = 'MGR' AND ID < 100
```

表 18 は、視点から作成された結果表を示しています。

*表 18. "SELECT * FROM V1"*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	0
30	Merenghi	38	MGR	5	17506.75	0
50	Hanes	15	Mgr	10	20659.80	0

視点 V1 に対して実行されるすべての照会は、表 18 に示した結果表に対して実行されます。以下の照会は、SRTSEQ(*LANGIDUNQ) および LANGID(ENU) の分類順序を用いて実行されています。

*表 19. "SELECT * FROM V1 WHERE JOB = 'MGR'" 言語 ID ENU の固有分類順序を使用した場合*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
30	Merenghi	38	MGR	5	17506.75	0

分類順序と CREATE INDEX ステートメント

索引は、CREATE INDEX ステートメントの実行時に有効であった分類順序を使用して作成されます。索引が定義されている表への挿入が行われるたびに、索引に項目が追加されます。索引項目には、文字キーおよび UCS-2 グラフィック・キー列についての重み付きの値が入ります。システムは、索引の分類順序に基づいてキー値を変換することによって、重み付きの値を入手します。

その分類順序とその索引を用いて選択が行われる場合、文字または UCS-2 グラフィック・キーは、比較の前に変換する必要がありません。これによって、照会のパフォーマンスが改善されます。有効な索引と分類順序の作成について詳しくは、データベース・パフォーマンスおよび *Query 最適化* のラージ・テーブルへのアクセスをスピードアップするための索引の使用を参照してください。

分類順序と制約

固有制約は、索引と一緒に組み込まれます。固有制約の付加された表が、ある分類順序を用いて定義された場合、索引はその同じ分類順序を用いて作成されます。

参照制約を定義する場合、親と従属表の間の分類順序は一致する必要があります。分類順序および制約について詳しくは、iSeries Information Center の中のデータベース・プログラミングの参照制約を使ったデータ保全性の確保のトピックを参照してください。

検査制約の定義時に使用される分類順序は、INSERT または UPDATE の実行時にその制約への順守性を検証するためにシステムによって使用されるのと同じ分類順序です。

第 9 章 カーソルの使用

SQL が選択ステートメントを実行する場合、その結果として生成された行が結果表を構成します。カーソルは、結果表にアクセスするための手段となります。カーソルは、結果表内の位置を保持するために SQL プログラムの中で使用されます。SQL は、カーソルを使用して結果表の行を処理し、それらをユーザーのプログラムが利用できるようにします。ユーザーのプログラムは複数のカーソルを持つことができますが、それぞれが固有の名前を持たなければなりません。

カーソルの使用に関連するステートメントは、次のとおりです。

- カーソルの定義とその名前の指定を行い、組み込まれている選択ステートメントによって取り出される行を指定する DECLARE CURSOR ステートメント。
- カーソルをプログラム内で使用するためにオープンしたりクローズしたりするための OPEN および CLOSE ステートメント。カーソルをオープンしてからでなければ、行を取り出すことはできません。
- カーソルの結果表から行を取り出したり、カーソルを別の行に移動したりするための FETCH ステートメント。
- カーソル現在行を更新するための UPDATE ... WHERE CURRENT OF ステートメント。
- カーソルの現在行を削除するための DELETE ... WHERE CURRENT OF ステートメント。

これらのステートメントの詳しい説明については、SQL 解説書を参照してください。

カーソルについての詳細は、以下のトピックを参照してください。

- 『カーソルのタイプ』
- 131 ページの『カーソルの使用例』
- 138 ページの『複数行用 FETCH ステートメントの使用』
- 143 ページの『作業単位とオープン・カーソル』

注: コーディング例に関する詳細は、x ページの『コードに関する特記事項』を参照してください。

カーソルのタイプ

SQL では、シリアル・カーソルとスクロール可能カーソルがサポートされます。カーソルのタイプによって、カーソルで使用できる位置決め方式が決まります。詳細については、以下を参照してください。

- 130 ページの『シリアル・カーソル』
- 130 ページの『スクロール可能カーソル』

シリアル・カーソル

シリアル・カーソルは、SCROLL キーワードを指定しないで定義するタイプのカーソルです。

シリアル・カーソルの場合、カーソルのオープンごとに結果表の各行を 1 回しか取り出すことができません。カーソルをオープンすると、カーソルは結果表の最初の行の前に置かれます。 FETCH が発行されると、カーソルは結果表内の次の行に移動します。その行が現在行となります。ホスト変数の指定がある場合 (FETCH ステートメントの INTO 文節で)、SQL は現在行の内容をプログラムのホスト変数に移動します。

この手順は、FETCH ステートメントが発行されるたびに、データの終わり (SQLCODE = 100) に達するまで繰り返し実行されます。データの終わりに達したら、カーソルをクローズしてください。データの終わりに達した後は、結果表内の行にアクセスすることはできません。シリアル・カーソルを再び使用するには、まずカーソルをクローズしてから OPEN ステートメントを再発行する必要があります。 シリアル・カーソルの使用をバックアップすることは、決してできません。

スクロール可能カーソル

スクロール可能カーソルの場合、結果表内の行は何度でも取り出すことができます。カーソルは、FETCH ステートメントに指定された位置オプションに基づいて、結果表内を移動します。カーソルをオープンすると、カーソルは結果表の最初の行の前に置かれます。 FETCH が発行されると、カーソルは、位置オプションによって指定された結果表内の行に移動します。その行が現在行となります。ホスト変数の指定がある場合 (FETCH ステートメントの INTO 文節で)、SQL は現在行の内容をプログラムのホスト変数に移動します。 位置オプションが BEFORE または AFTERの場合には、ホスト変数を指定することはできません。

この手順は、FETCH ステートメントが発行されるたびに、繰り返し実行されます。データの終わり条件またはデータの先頭条件が発生しても、カーソルをクローズする必要はありません。位置オプションが指定してあるので、プログラムは引き続き表から行を取り出すことができます。

次のスクロール・オプションは、FETCH ステートメントの発行時にカーソルの位置決めに使用されます。これらの位置は、結果表内の現在のカーソル位置に対する相対的なものです。

NEXT	カーソルを次の行に置きます。位置を指定しない場合は、これが省略時値です。
PRIOR	カーソルを前の行に置きます。
FIRST	カーソルを最初の行に置きます。
LAST	カーソルを最後の行に置きます。
BEFORE	カーソルを最初の行の前に置きます。
AFTER	カーソルを最後の行の後に置きます。
CURRENT	カーソル位置を変更しません。

RELATIVE n	カーソルの現在位置との相対的な関係でホスト変数または整数 *n* を評価します。たとえば、*n* が -1 の場合には、カーソルは結果表の現在行の 1 つ前の行に置かれます。*n* が +3 の場合には、カーソルは現在行の 3 つ後の行に置かれます。

スクロール可能なカーソルの場合、表の終わりは次のステートメントで判別することができます。

```
FETCH AFTER FROM C1
```

カーソルが表の終わりに置かれると、プログラムは PRIOR または RELATIVE スクロール・オプションを使用して、表の終わりから始まるデータを見つけて取り出すことができます。

カーソルの使用例

プログラムで部門 D11 の社員についてのデータを調べるとします。次の例では、シリアル・カーソルとスクロール可能カーソルを定義して使用するためにプログラムに組み込む SQL ステートメントを示します。このようなカーソルを使用すると、CORPDATA.EMPLOYEE 表からその部門に関する情報を取り出すことができます。

シリアル・カーソルの例では、プログラムは表から取り出したすべての行を処理して、部門 D11 の全社員の職種を更新し、他の部門からの社員のレコードを削除します。

表 20. シリアル・カーソルの例

シリアル・カーソル用の SQL ステートメント	参照ページ
EXEC SQL **DECLARE** THISEMP **CURSOR FOR** **SELECT** EMPNO, LASTNAME, WORKDEPT, JOB **FROM** CORPDATA.EMPLOYEE **FOR UPDATE OF** JOB END-EXEC.	133 ページの『ステップ 1: カーソルを定義する』
EXEC SQL **OPEN** THISEMP END-EXEC.	134 ページの『ステップ 2: カーソルをオープンする』
EXEC SQL **WHENEVER NOT FOUND** **GO TO** CLOSE-THISEMP END-EXEC.	135 ページの『ステップ 3: データの終わりに達したときの処置を指定する』
EXEC SQL **FETCH** THISEMP **INTO** :EMP-NUM, :NAME2, :DEPT, :JOB-CODE END-EXEC.	135 ページの『ステップ 4: カーソルを用いて行を取り出す』

表 20. シリアル・カーソルの例 (続き)

シリアル・カーソル用の SQL ステートメント	参照ページ
... 部門 11 のすべての 社員についての JOB の値を 更新する。 EXEC SQL **UPDATE** CORPDATA.EMPLOYEE **SET** JOB = :NEW-CODE **WHERE CURRENT OF** THISEMP END-EXEC. ... 次にその行を印刷する。	136 ページの『ステップ 5a: 現在行を更新する』
... 他の社員については、 行を削除する。 EXEC SQL **DELETE FROM** CORPDATA.EMPLOYEE **WHERE CURRENT OF** THISEMP END-EXEC.	136 ページの『ステップ 5b: 現在行を削除する』
FETCH に戻り、次の行を処理する。	
CLOSE-THISEMP. EXEC SQL **CLOSE** THISEMP END-EXEC.	137 ページの『ステップ 6: カーソルをクローズする』

スクロール可能カーソルの例では、プログラムは RELATIVE 位置オプションを使用して、部門 D11 の代表的な給与例を取り出します。

表 21. スクロール可能カーソルの例

スクロール可能カーソル用の SQL ステートメント	参照ページ
EXEC SQL **DECLARE** THISEMP **DYNAMIC** **SCROLL CURSOR FOR** **SELECT** EMPNO, LASTNAME, SALARY **FROM** CORPDATA.EMPLOYEE **WHERE** WORKDEPT = 'D11' END-EXEC.	133 ページの『ステップ 1: カーソルを定義する』
EXEC SQL **OPEN** THISEMP END-EXEC.	134 ページの『ステップ 2: カーソルをオープンする』
EXEC SQL **WHENEVER NOT FOUND** **GO TO** CLOSE-THISEMP END-EXEC.	135 ページの『ステップ 3: データの終わりに達したときの処置を指定する』

表 21. スクロール可能カーソルの例 (続き)

スクロール可能カーソル用の SQL ステートメント	参照ページ
...プログラムの合計給与変数を 初期設定する。 EXEC SQL **FETCH RELATIVE** 3 **FROM** THISEMP **INTO** :EMP-NUM, :NAME2, :JOB-CODE END-EXEC. ...現行給与をプログラムの 合計給与に加算する。 ...FETCH に戻り、 次の行を処理する。	135 ページの『ステップ 4: カーソルを用いて行を取り出す』
...平均給与を 計算する。	
CLOSE-THISEMP. EXEC SQL **CLOSE** THISEMP END-EXEC.	137 ページの『ステップ 6: カーソルをクローズする』

ステップ 1: カーソルを定義する

カーソルでアクセスする結果表を定義するには、DECLARE CURSOR ステートメントを使用します。

DECLARE CURSOR ステートメントでは、カーソルの名前を指定し、選択ステートメントを指定します。選択ステートメントでは、結果表を (概念上) 構成する行のセットを定義します。シリアル・カーソルの場合、ステートメントは次のようになります (FOR UPDATE OF 文節は任意指定です)。

```
EXEC SQL
 DECLARE カーソル名 CURSOR FOR
 SELECT 列 1, 列 2 ,...
   FROM 表名 , ...
   FOR UPDATE OF 列 2 ,...
END-EXEC.
```

スクロール可能カーソルの場合、ステートメントは次のようになります (WHERE 文節は任意指定です)。

```
EXEC SQL
 DECLARE カーソル名 DYNAMIC SCROLL CURSOR FOR
 SELECT 列 1, 列 2 ,...
   FROM 表名 ,...
   WHERE 列 1 = 式 ...
END-EXEC.
```

ここに示した選択ステートメントはどちらかと言えば単純なものです。しかし、シリアル・カーソルまたはスクロール可能カーソル用の DECLARE CURSOR ステートメントの選択ステートメントには、他のタイプの文節をコーディングすることができます。

識別されている表 (FROM 文節に指定した表) のいずれかの行またはすべての行の中の列を更新しようとしている場合には、FOR UPDATE OF 文節を含めてください。この文節には、更新しようとする各列名を指定します。列の名前を指定しないで、ORDER BY 文節または FOR READ ONLY 文節を指定すると、更新を試みたときに負の SQLCODE が返されます。 FOR UPDATE OF 文節、FOR READ ONLY 文節、または ORDER BY 文節を指定しないで、しかも結果表が読み取り専用でない場合は、指定した表のどの列でも更新することができます。

指定した表の列は、それが結果表の一部でなくても、更新することができます。この場合には、SELECT ステートメントで列名を指定する必要はありません。更新したい列値が入っている行をカーソルで (FETCH を用いて) 取り出すと、UPDATE ... WHERE CURRENT OF を使用してその行を更新できます。

たとえば、結果表の各行に、CORPDATA.EMPLOYEE 表から取り出された *EMPNO* 列、*LASTNAME* 列、および *WORKDEPT* 列が含まれているとします。 *JOB* 列 (CORPDATA.EMPLOYEE 表の各行の列の 1 つ) を更新したい場合には、SELECT ステートメントに JOB を指定しなくても、DECLARE CURSOR ステートメントに FOR UPDATE OF JOB ... を組み込む必要があります。

次のいずれかに該当する場合は、結果表とカーソルは*読み取り専用* になります。

- 最初の FROM 文節で 2 つ以上の表または視点を指定している。
- 最初の FROM 文節で読み取り専用視点を指定している。
- 最初の FROM 文節でユーザー定義表関数を指定している。
- 最初の SELECT 文節でキーワード DISTINCT を指定している。
- 外部副選択が GROUP BY 文節を含んでいる。
- 外部副選択が HAVING 文節を含んでいる。
- 最初の SELECT 文節に列関数が含まれている。
- 選択ステートメントに、外部副選択の基本オブジェクトと副照会の基本オブジェクトが同一の表であるような副照会が、含まれている。
- 選択ステートメントに UNION または UNION ALL 演算子が含まれている。
- 選択ステートメントに ORDER BY 文節が含まれており、FOR UPDATE OF 文節と DYNAMIC SCROLL が指定されていない。
- 選択ステートメントに FOR READ ONLY 文節が含まれている。
- DYNAMIC を指定せずに、SCROLL キーワードが指定されている。
- 選択リストに データ・リンク列が組み込まれ、FOR UPDATE OF 文節が指定されていない。
- 最初の副選択に一時結果表が必要である。
- 選択ステートメントに *n* ROWS ONLY が組み込まれている。

ステップ 2: カーソルをオープンする

結果表内の行の処理を開始するには、OPEN ステートメントを使用します。プログラムで OPEN ステートメントが実行されると、SQL は DECLARE CURSOR ステートメント内の選択ステートメントを処理し、選択ステートメントに指定されているすべてのホスト変数の現行値を用いて、結果表と呼ばれる一連の行を識別しま

す。結果表には、検索条件を満たす行の数に応じて、0 行、1 行、または複数行を入れることができます。OPEN ステートメントは次のようになります。

```
EXEC SQL
 OPEN カーソル名
END-EXEC.
```

ステップ 3: データの終わりに達したときの処置を指定する

結果表の終わりに達したことを調べるには、SQLCODE フィールドの値が 100 であるかをテストするか、または SQLSTATE フィールドの値が '02000' (すなわち、データの終わり) であるかをテストしてください。 この条件は、FETCH ステートメントが結果表の最後の行を取り出した後で、さらにプログラムにより FETCH が発行された場合に起こります。たとえば、次の通りです。

```
...
IF SQLCODE =100 GO TO DATA-NOT-FOUND.
```

あるいは

```
IF SQLSTATE ='02000' GO TO DATA-NOT-FOUND.
```

これに代わる方法として、WHENEVER ステートメントのコーディングがあります。 WHENEVER NOT FOUND を使用すると、分岐で CLOSE ステートメントを発行するプログラムの別の部分に復帰できます。WHENEVER ステートメントは次のようになります。

```
EXEC SQL
 WHENEVER NOT FOUND GO TO 記号アドレス
END-EXEC.
```

プログラムでは、カーソルを用いて行を取り出すときには、常にデータの終わり条件を予期し、その条件が起こったときの処理の準備をしておくべきです。

シリアル・カーソルを使用しているときにデータの終わりに達すると、後続のすべての FETCH ステートメントがデータの終わり条件を返します。すでに処理が終わっている行にカーソルを置くことはできません。このカーソルに対して実行できる操作は CLOSE ステートメントのみです。

スクロール可能カーソルを使用しているときにデータの終わりに達しても、結果表ではまだ追加のデータを処理することができます。位置オプションを組み合わせて使用すれば、結果表のどこにでもカーソルを移動することができます。データの終わりに達しても、カーソルを CLOSE する必要はありません。

ステップ 4: カーソルを用いて行を取り出す

選択された行の内容をプログラムのホスト変数の中に移動するには、FETCH ステートメントを使用します。DECLARE CURSOR ステートメントの中の SELECT ステートメントでは、プログラムに必要な列値が入っている行を識別します。しかし、SQL は、FETCH ステートメントが発行されるまでは、アプリケーション・プログラムのためにデータを取り出しません。

プログラムから FETCH ステートメントが発行されると、SQL は現在のカーソル位置を開始点として使用し、結果表内の要求された行を見つけます。これにより、そ

の行が**現在行**になります。 INTO 文節の指定があるときは、SQL は現在行の内容をプログラムのホスト変数に移動します。この手順は、FETCH ステートメントが発行されるたびに繰り返し実行されます。

SQL は、カーソルに対する次の FETCH ステートメントが発行されるまで、現在行の位置を保持します (すなわち、カーソルは現在行を指しています)。 UPDATE ステートメントでは結果表内の現在行の位置は変わりませんが、DELETE ステートメントでは変わります。

シリアル・カーソルの FETCH ステートメントは、次のようになります。

```
EXEC SQL
 FETCH カーソル名
   INTO :ホスト変数 1[, :ホスト変数 2] ...
END-EXEC.
```

スクロール可能カーソルの FETCH ステートメントは次のようになります。

```
EXEC SQL
 FETCH RELATIVE 整数
   FROM カーソル名
   INTO :ホスト変数 1[, :ホスト変数 2] ...
END-EXEC.
```

ステップ 5a: 現在行を更新する

プログラムによってカーソルが行に位置付けられたら、WHERE CURRENT OF 文節を指定した UPDATE ステートメントを使用してそのデータを更新することができます。WHERE CURRENT OF 文節には、更新したい行を指し示すカーソルを指定します。UPDATE ... WHERE CURRENT OF ステートメントは次のようになります。

```
EXEC SQL
 UPDATE 表名
   SET 列 1 = 値 [, 列 2 = 値] ...
   WHERE CURRENT OF カーソル名
END-EXEC.
```

UPDATE ステートメントをカーソルと一緒に使用すると、次の働きをします。

- 1 行だけ (すなわち、現在行) を更新します。
- 更新すべき行を指し示すカーソルを識別します。
- ORDER BY 文節も指定された場合は、DECLARE CURSOR ステートメントの FOR UPDATE OF 文節で、更新される列に前もって名前を付けておく必要があります。

ある行を更新した後も、次の行に対する FETCH ステートメントが発行されるまでは、カーソルの位置はその行にとどまっています (すなわち、カーソルの現在行は変わりません)。

ステップ 5b: 現在行を削除する

プログラムによって現在行が取り出されたら、DELETE ステートメントを使用してその行を削除することができます。これを行うには、カーソルと一緒に使用することを目的とした DELETE ステートメントを使用します。 WHERE CURRENT OF 文節には、削除したい行を指し示すカーソルを指定します。DELETE ... WHERE CURRENT OF ステートメントは次のようになります。

```
EXEC SQL
 DELETE FROM 表名
   WHERE CURRENT OF カーソル名
END-EXEC.
```

DELETE ステートメントをカーソルと一緒に使用すると、次の働きをします。

- 1 行だけ (すなわち、現在行) を削除します。
- WHERE CURRENT OF 文節を用いて、削除すべき行を指し示すカーソルを識別します。

行を削除した後は、FETCH ステートメントを発行してカーソルを位置付けるまでは、そのカーソルを用いて別の行を更新または削除することはできません。

111 ページの『DELETE ステートメントを使用した表からの行の除去』では、DELETE ステートメントを使用して、特定の検索条件を満たすすべての行を削除する方法が説明されています。行のコピーを取り出し、それを検査してから削除したい場合には、FETCH および DELETE ... WHERE CURRENT OF ステートメントを使用することもできます。

ステップ 6: カーソルをクローズする

シリアル・カーソルで結果表の行の処理を終えて、カーソルを再び使用したい場合は、CLOSE ステートメントを発行してカーソルをクローズしてからもう一度オープンしてください。

```
EXEC SQL
 CLOSE カーソル名
END-EXEC.
```

結果表の行の処理を終えて、カーソルを再び使用する必要がないときは、そのままにしておけば、システムにカーソルをクローズさせることができます。システムが自動的にカーソルをクローズするのは、次の場合です。

- HOLD の指定がない COMMIT ステートメントが発行され、カーソルが WITH HOLD 文節により宣言されていない場合。
- HOLD の指定がない ROLLBACK ステートメントが発行された場合。
- ジョブが終了した場合。
- 活動化グループが終了し、事前コンパイル時に CLOSQLCSR(*ENDACTGRP) が指定された場合。
- 呼び出しスタック内の最初の SQL プログラムが終了し、プログラムの事前コンパイル時に CLOSQLCSR(*ENDJOB) も CLOSQLCSR(*ENDACTGRP) も指定されなかった場合。
- DISCONNECT ステートメントにより、アプリケーション・サーバーへの接続が終了された場合。
- アプリケーション・サーバーへの接続が解放され、COMMIT が正常に行われた場合。
- *RUW CONNECT が起こった場合。

  オープン・カーソルは参照された表または視点に対するロックを保持し続けるため、オープン・カーソルが不要になったら、できるだけ早く明示的にクローズするべきです。

複数行用 FETCH ステートメントの使用

複数行用 FETCH ステートメントを使用すると、1 回の FETCH で表または視点から複数の行を取り出すことができます。行のブロック化は、FETCH ステートメントで要求された行の数に応じてプログラムが制御します (OVRDBF は何の影響ももたらしません)。1 回の FETCH 呼び出しで要求できる最大行数は 32767 行です。データが取り出されると、カーソルは取り出された最後の行に置かれます。

取り出した行が置かれる記憶域を定義する方法は、2 つあります。すなわち、ホスト構造配列と、関連記述子を持つ行記憶域です。どちらの方法も、SQL 事前コンパイラーによりサポートされるすべての言語でコーディングすることができます。ただし、REXX でのホスト構造配列は例外です。プログラミング言語の詳細については、ホスト言語での SQL プログラミングを参照してください。複数行用 FETCH ステートメントのどちらの形式を使用しても、アプリケーションで個別の標識配列をコーディングすることができます。この標識配列には、ヌル値可能の各ホスト変数につき 1 つの標識が入ります。

複数行用 FETCH ステートメントは、シリアルとスクロール可能のどちらのカーソルとでも一緒に使用できます。複数行用 FETCH のためにカーソルを定義、オープン、およびクローズするための操作は変わりません。 FETCH ステートメントで、取り出す行の数と行が置かれる記憶域を指定する点だけが異なります。

それぞれの複数行用 FETCH を実行した後で、SQLCA を介してプログラムに情報が返されます。 SQLCODE フィールドと SQLSTATE フィールドのほかに、SQLERRD は次の情報を提供します。

- SQLERRD3 には、複数行用 FETCH ステートメントで取り出された行の数が入ります。SQLERRD3 が要求した行数より少ない場合は、エラーまたはデータの終わり条件が発生しています。
- SQLERRD4 には、取り出された各行の長さが入ります。
- SQLERRD5 には、表の最後の行が取り出されたことを示す標識が入ります。これは、カーソルが更新を即時に感知しないときに、取り出しが行われている表のデータの終わり条件を検出するために使用できます。更新を即時に感知するカーソルは、SQLCODE +100 を受け取ってデータの終わり条件を検出するまで取り出しを継続します。

ホスト構造配列を使用した複数行 FETCH

複数行用 FETCH をホスト構造配列とともに使用するには、アプリケーションで、SQL により使用できるホスト構造配列を定義しなければなりません。それぞれの言語には、ホスト構造配列を定義するための独自の規則があります。ホスト構造配列は、変数宣言を使用するか、または外部記述ファイルを取り出すためのコンパイラー指示 (COBOL の COPY 指示など) を使用することによって定義できます。

ホスト構造配列は、構造の配列から構成されます。それぞれの構造は、結果表の 1 つの行に対応します。すなわち、配列の最初の構造は最初の行に対応し、配列の 2 番目の構造は 2 番目の行に対応し、以下同様です。ホスト構造配列内の基本項目の属性は、ホスト構造配列の宣言に基づいて SQL が決定します。パフォーマンスを最高にするには、ホスト構造配列を構成する各項目の属性が、取り出される列の属性と一致していなければなりません。

次の COBOL 例を見てください。

```
      EXEC SQL INCLUDE SQLCA
      END-EXEC.

...

      01 TABLE-1.
          02 DEPT  OCCURS 10 TIMES.
             05  EMPNO   PIC   X(6).
             05 LASTNAME.
                49 LASTNAME-LEN  PIC S9(4) BINARY.
                49 LASTNAME-TEXT PIC X(15).
             05  WORKDEPT PIC  X(3).
             05  JOB      PIC  X(8).
      01 TABLE-2.
         02 IND-ARRAY   OCCURS 10 TIMES.
            05 INDS PIC  S9(4) BINARY OCCURS 4 TIMES.

...
      EXEC SQL
      DECLARE D11 CURSOR FOR
      SELECT EMPNO, LASTNAME, WORKDEPT, JOB
       FROM CORPDATA.EMPLOYEE
        WHERE WORKDEPT = "D11"
      END-EXEC.

...

      EXEC SQL
       OPEN D11
      END-EXEC.
      PERFORM FETCH-PARA UNTIL SQLCODE NOT EQUAL TO ZERO.
  ALL-DONE.
        EXEC SQL CLOSE D11 END-EXEC.

...

      FETCH-PARA.
         EXEC SQL WHENEVER NOT FOUND GO TO ALL-DONE END-EXEC.
      EXEC SQL FETCH D11 FOR 10 ROWS INTO :DEPT :IND-ARRAY
      END-EXEC.

...
```

この例では、カーソルは、CORPDATA.EMPLOYEE 表で WORKDEPT 列が 'D11' に等しいすべての行を選択するように定義されています。結果表には 8 行が入ります。DECLARE CURSOR ステートメントと OPEN ステートメントには、複数行用 FETCH ステートメントとともに使用する場合の特殊な構文はありません。同じカーソルに対して 1 つの行を返す別の FETCH ステートメントをプログラム内のほかの場所にコーディングすることができます。複数行用 FETCH ステートメントは、結果表のすべての行を取り出すために使用されます。FETCH の実行後、カーソルは取り出された最後の行に置かれたままになります。

アプリケーションでは、ホスト構造配列 DEPT とそれに関連する標識配列 IND-ARRAY が定義されています。どちらの配列もディメンションは 10 です。標識配列には、結果表の各列につき 1 つの項目が入ります。

DEPT ホスト構造配列の基本項目のタイプおよび長さの属性は、取り出される列と一致しています。

複数行用 FETCH ステートメントが正しく完了すると、ホスト構造配列には 8 行分すべてのデータが入ります。標識配列 IND_ARRAY には、各行のすべての列にゼロが入ります。これは、NULL 値が返されなかったためです。

アプリケーションに返される SQLCA には、次の情報が入ります。

- SQLCODE には、0 が入ります。
- SQLSTATE には、'00000' が入ります。
- SQLERRD3 には、取り出された行の数を示す 8 が入ります。
- SQLERRD4 には、各行の長さを示す 34 が入ります。
- SQLERRD5 には、結果表の最後の行がブロックに入っていることを示す +100 が入ります。

SQLCA の詳細については、SQL 解説書 の付録 B を参照してください。

行記憶域を使用した複数行 FETCH

アプリケーションで複数行用 FETCH を行記憶域とともに使用するには、行記憶域および関連する記述子域を定義しておかなければなりません。行記憶域とは、アプリケーション・プログラムで定義されるホスト変数のことです。この行記憶域には、複数行用 FETCH の結果が入ります。行記憶域は、複数行用 FETCH で要求された行をすべて収容できるだけの十分なバイト数を持つ文字変数でも構いません。

行記憶域形式の 複数行用 FETCH で使用される関連記述子では、返される各列の SQLTYPE と SQLLEN が入る SQLDA を定義します。記述子に指定する情報によって、データベースから行記憶域へのデータ・マッピングが判別されます。パフォーマンスを最高にするには、記述子に指定されている属性情報が、取り出される列の属性と一致していなければなりません。

SQLDA の詳細については、SQL 解説書 の付録 C を参照してください。

次の PL/I の例を検討してください。

```
       *....+....1....+....2....+....3....+....4....+....5....+....6....+....7...*
     EXEC SQL INCLUDE SQLCA;
     EXEC SQL INCLUDE SQLDA;

...

    DCL DEPTPTR PTR;
    DCL 1 DEPT(20) BASED(DEPTPTR),
        3 EMPNO CHAR(6),
        3 LASTNAME CHAR(15) VARYING,
        3 WORKDEPT CHAR(3),
        3 JOB CHAR(8);
    DCL I BIN(31) FIXED;
    DEC J BIN(31) FIXED;
    DCL ROWAREA CHAR(2000);

...

    ALLOCATE SQLDA SET(SQLDAPTR);
    EXEC SQL
     DECLARE D11 CURSOR FOR
     SELECT EMPNO, LASTNAME, WORKDEPT, JOB
     FROM CORPDATA.EMPLOYEE
     WHERE WORKDEPT = 'D11';
```

図 1. 行記憶域を使用する複数行用 FETCH の例 (1/2)

```
...
    EXEC SQL
      OPEN D11;
    /* SET UP THE DESCRIPTOR FOR THE MULTIPLE-ROW FETCH */
    /* 4 COLUMNS ARE BEING FETCHED                 */
    SQLD = 4;
    SQLN = 4;
    SQLDABC = 366;
    SQLTYPE(1) = 452; /* FIXED LENGTH CHARACTER -  */
                      /* NOT NULLABLE              */
    SQLLEN(1) = 6;
    SQLTYPE(2) = 456; /*VARYING LENGTH CHARACTER   */
                      /* NOT NULLABLE              */
    SQLLEN(2) = 15;
    SQLTYPE(3) = 452; /* FIXED LENGTH CHARACTER -  */
    SQLLEN(3) = 3;
    SQLTYPE(4) = 452; /* FIXED LENGTH CHARACTER -  */
                      /* NOT NULLABLE              */
    SQLLEN(4) = 8;
    /*ISSUE THE MULTIPLE-ROW FETCH STATEMENT TO RETRIEVE*/
    /*THE DATA INTO THE DEPT ROW STORAGE AREA      */
    /*USE A HOST VARIABLE TO CONTAIN THE COUNT OF  */
    /*ROWS TO BE RETURNED ON THE MULTIPLE-ROW FETCH     */

    J = 20;        /*REQUESTS 20 ROWS ON THE FETCH */
   ...
    EXEC SQL
      WHENEVER NOT FOUND
      GOTO FINISHED;
    EXEC SQL
      WHENEVER SQLERROR
      GOTO FINISHED;
    EXEC SQL
      FETCH D11 FOR :J ROWS
      USING DESCRIPTOR :SQLDA INTO :ROWAREA;
    /* ADDRESS THE ROWS RETURNED                   */
    DEPTPTR = ADDR(ROWAREA);
    /*PROCESS EACH ROW RETURNED IN THE ROW STORAGE */
    /*AREA BASED ON THE COUNT OF RECORDS RETURNED  */
    /*IN SQLERRD3.                                 */
    DO I = 1 TO SQLERRD(3);
     IF EMPNO(I) = '000170'  THEN
       DO;
      :
       END;
    END;
    IF SQLERRD(5) = 100 THEN
      DO;
        /* PROCESS END OF FILE  */
      END;
    FINISHED:
```

図 1. 行記憶域を使用する複数行用 FETCH の例 (2/2)

この例では、カーソルは、CORPDATA.EMPLOYEE 表で WORKDEPT 列が 'D11' に等しいすべての行を選択するように定義されています。『付録 A. DB2 UDB for iSeries サンプル表』の EMPLOYEE 表の例では、結果表に複数行が入ることが示されています。 DECLARE CURSOR ステートメントと OPEN ステートメントには、複数行用 FETCH ステートメントとともに使用する場合の特殊な構文はありません。 同じカーソルに対して 1 つの行を返す別の FETCH ステートメントをプログラム内のほかの場所にコーディングすることができます。 複数行用 FETCH ステー

トメントは、結果表のすべての行が取り出すために使用されます。 FETCH の実行後、カーソルはブロック内の最終行に置かれたままになります。

行記憶域 ROWAREA は、文字配列として定義されています。結果表から取り出されたデータは、ホスト変数に入ります。この例では、ROWAREA のアドレスにポインター変数が割り当てられています。返される行内の各項目が検査され、基底付き構造 DEPT で使われます。

記述子の項目の属性 (タイプと長さ) は、取り出される列と一致しています。このケースでは、標識区域は指定されていません。

FETCH ステートメントが完了すると、ROWAREA には 'D11' と等しいすべての行 (この例では 11 行) が入ります。 アプリケーションに返される SQLCA には、次の情報が入ります。

- SQLCODE には、0 が入ります。
- SQLSTATE には、'00000' が入ります。
- SQLERRD3 には、返される行数を示す 11 が入ります。
- SQLERRD4 には、取り出される行の長さを示す 34 が入ります。
- SQLERRD5 には、結果表の最後の行が取り出されたことを示す +100 が入ります。

この例では、アプリケーションは、ファイルの終わりに達したことを示す標識が SQLERRD5 に入るということを利用しています。その結果、このアプリケーションでは、さらに行の取り出しを試みるためにもう一度 SQL を呼び出す必要がありません。カーソルが挿入を即時に感知する場合は、レコードが追加されたときに SQL を呼び出す必要があります。カーソルが即時感知性を持つのは、コミットメント制御レベルが *RR 以外である場合です。

作業単位とオープン・カーソル

プログラムでは、1 つの作業単位を完了したときに、それまでに行われた変更をコミットまたはロールバックする必要があります。COMMIT ステートメントまたは ROLLBACK ステートメントに HOLD の指定がなければ、オープン・カーソルはすべて SQL によって自動的にクローズされます。WITH HOLD 文節を用いて宣言されたカーソルは、COMMIT 時に自動的にクローズされません。このようなカーソルは、ROLLBACK 時には自動的にクローズされます (DECLARE CURSOR ステートメントに指定された WITH HOLD 文節は無視されます)。

COMMIT または ROLLBACK の後で現在のカーソル位置から処理を継続したいときは、COMMIT HOLD または ROLLBACK HOLD を指定しなければなりません。HOLD の指定があると、オープン・カーソルはすべてオープンされたままになり、処理が再開できるようにそのカーソル位置を保持しています。COMMIT ステートメントでは、カーソル位置は維持されます。 ROLLBACK ステートメントでは、カーソル位置は、直前の作業単位で最後に取り出された行の直後に復元されます。レコード・ロックはすべて解除されます。

HOLD の指定なしで COMMIT または ROLLBACK ステートメントを発行すると、すべてのロックが解除され、すべてのカーソルがクローズされます。カーソルは再びオープンできますが、処理は結果表の最初の行から始まります。

注: CRTSQLxxx コマンドで ALWBLK(*ALLREAD) パラメーターを指定すると、読み取り専用カーソルのカーソル位置の復元を変更することができます。CRTSQLxxx コマンドで ALWBLK パラメーターおよびその他のパフォーマンス関連オプションを使用する方法の詳細については、『第 14 章 動的 SQL アプリケーション』を参照してください。

コミットメント制御と作業単位について詳しくは、コミットメント制御のトピックを参照してください。

第 10 章 データ保全性

データ保全性とは、スキーマの表間のデータ値が、業務にとって意味がある状態に保持できるようにする規範のことです。 たとえば、ある銀行の表 A に顧客リストが入っており、表 B に顧客の口座リストが入っている場合、新規の口座を表 B に追加できても、対応する顧客が表 A に存在しなければ、何の意味もありません。

この章では、システムがこの種の関係を自動的に適用するためのさまざまな方法について説明します。参照保全、検査制約、およびトリガーはすべて、データ保全性を確保するための手段です。さらに、CREATE VIEW の WITH CHECK OPTION 文節は、視点を介するデータの挿入または更新を制限します。詳細については、以下のトピックを参照してください。

- 『検査制約の追加および使用』
- 146 ページの『参照保全』
- 155 ページの『視点に関する WITH CHECK OPTION』
- 158 ページの『DB2 UDB for iSeries トリガー・サポート』

データ保全性に関する包括的な情報については、データベース・プログラミングを参照してください。

検査制約の追加および使用

検査制約 は、列または列のグループの中で使用できる値を制限することにより、挿入および更新中のデータの妥当性を保証します。 SQL の CREATE TABLE ステートメントまたは ALTER TABLE ステートメントを使用して、検査制約を追加または削除することができます。

以下の例では、次のステートメントにより、3 つの列を持つ表が作成され、COL2 には、その列で使用できる値を正の整数に制限する検査制約が作成されます。

```
CREATE TABLE T1 (COL1 INT, COL2 INT CHECK (COL2>0), COL3 INT)
```

この表の場合、次のステートメントは、

```
INSERT INTO T1 VALUES (-1, -1, -1)
```

COL2 に挿入される値が検査制約に適合しない (すなわち、-1 は 0 より大きくない) ため、失敗します。

次のステートメントは成功します。

```
INSERT INTO T1 VALUES (1, 1, 1)
```

その行が挿入された後は、次のステートメントは失敗します。

```
ALTER TABLE T1 ADD CONSTRAINT C1 CHECK (COL1=1 AND COL1<COL2)
```

この ALTER TABLE ステートメントは、COL1 で使用できる値を 1 に制限し、かつ、COL2 の値が 1 より大きいことを規定する 2 番目の検査制約の追加を試みる

ものです。この制約は、制約の 2 番目の部分が既存のデータに適合しない (COL2 の '1' の値が COL1 の '1' の値より大きくない) ため、許可されません。

参照保全

参照保全とは、1 つの表から別の表へのあらゆる参照が有効であるデータベースの中の一組の表の状態のことをいいます。

以下の例を考えて見ましょう (これらのサンプル表は『付録 A. DB2 UDB for iSeries サンプル表』にあります)。

- CORPDATA.EMPLOYEE は、社員のマスター・リストです。
- CORPDATA.DEPARTMENT は、有効なすべての部門番号のマスター・リストです。
- CORPDATA.EMP_ACT は、プロジェクトごとに行われる活動のマスター・リストです。

他の表では、上記の表で記述されているのと同じエンティティーを参照しています。表に、マスター・リストのあるデータが含まれる場合は、そのデータが実際にマスター・リストに入っていなければなりません。そうでなければ、その参照は無効となります。マスター・リストの入っている表が*親表* で、それを参照している表が*従属表* です。従属表から親表への参照が有効である場合、それらの一組の表の状態が*参照保全* と呼ばれます。

言いかえれば、参照保全とは、すべての外部キーのすべての値が有効であるデータベースの状態のことです。外部キーの各値は親キーの中にもなければならないか、ヌルでなければなりません。参照保全のこの定義では、次の用語を理解している必要があります。

- *固有キー* は、行を固有に識別する、表内の列または列のセットです。1 つの表が複数の固有キーを持つことができますが、表内の 2 つの行が同じ固有キーの値を持つことはできません。
- *基本キー* は、ヌル値を認めない固有キーです。1 つの表が複数の基本キーを持つことはできません。
- *親キー* は、参照制約で参照される固有キーまたは基本キーのいずれかです。
- *外部キー* は、値が親キーの値と一致しなければならない列または列のセットです。外部キーの作成に使用されるいずれかの列値がヌルである場合には、この規則は当てはまりません。
- *親表* は、親キーが入っている表です。
- *従属表* は、外部キーが入っている表です。
- *下層表* は、従属表または従属表の下層の表です。

参照保全を実施することにより、ヌルでないすべての外部キーに対応する親キーがなければならないという規則への違反が防止されます。

参照保全について詳しくは、以下のトピックを参照してください。

- 147 ページの『参照制約の追加または削除』
- 148 ページの『参照制約の除去』
- 149 ページの『参照制約付き表への挿入』

- 150 ページの『参照制約付きの表の更新』
- 151 ページの『参照制約付き表からの削除』
- 154 ページの『検査保留』

SQL は、CREATE TABLE ステートメントおよび ALTER TABLE ステートメントにより参照保全の概念をサポートします。これらのコマンドの詳細については、SQL 解説書を参照してください。 表の作成時に制約の使用を追加することもできますし、あるいは、iSeries ナビゲーターを使用して既存の表に制約を追加することもできます。

参照制約の追加または削除

制約は、1 つの表 (従属表) から別の表 (親表) のデータへの参照が必ず有効となるようにするための規則です。 参照保全を確実にするために、参照制約を使用します。

SQL の CREATE TABLE ステートメントまたは ALTER TABLE ステートメントを使用して、検査制約を追加または変更することができます。

参照制約を使用すると、外部キーの値が親キーの値としても使用されている場合にのみ、外部キーのヌルでない値が有効になります。参照制約を定義する際には、次のものを指定してください。

- 基本キーまたは固有キー
- 外部キー
- 親行が削除または更新されるときに従属行に関してとられる処置を指定する削除規則と更新規則

オプションで、制約の名前を指定することができます。名前を指定しないと、自動的に生成されます。

参照制約が定義されると、システムは、SQL またはその他のインターフェース (iSeries ナビゲーター、CL コマンド、ユーティリティー、または高水準言語ステートメントなど) を使用して実行されるすべての INSERT、DELETE、および UPDATE 操作にこの制約を適用します。

例: 参照制約の追加

サンプル社員表内のすべての部門番号が部門表に入っていなければならないという規則は、参照制約です。この制約により、すべての社員が既存の部門に所属することが保証されます。次の SQL ステートメントでは、このような制約関係が定義された CORPDATA.DEPARTMENT 表および CORPDATA.EMPLOYEE 表を作成します。

```
CREATE TABLE CORPDATA.DEPARTMENT
      (DEPTNO    CHAR(3)     NOT NULL PRIMARY KEY,
       DEPTNAME  VARCHAR(29) NOT NULL,
       MGRNO     CHAR(6),
       ADMRDEPT  CHAR(3)     NOT NULL
                             CONSTRAINT REPORTS_TO_EXISTS
                               REFERENCES CORPDATA.DEPARTMENT (DEPTNO)
                               ON DELETE CASCADE)

CREATE TABLE CORPDATA.EMPLOYEE
      (EMPNO     CHAR(6)     NOT NULL PRIMARY KEY,
```

```
FIRSTNAME VARCHAR(12) NOT NULL,
MIDINIT   CHAR(1)     NOT NULL,
LASTNAME  VARCHAR(15) NOT NULL,
WORKDEPT  CHAR(3)     CONSTRAINT WORKDEPT_EXISTS
                        REFERENCES CORPDATA.DEPARTMENT (DEPTNO)
                        ON DELETE SET NULL ON UPDATE RESTRICT,
PHONENO   CHAR(4),
HIREDATE  DATE,
JOB       CHAR(8),
EDLEVEL   SMALLINT    NOT NULL,
SEX       CHAR(1),
BIRTHDATE DATE,
SALARY    DECIMAL(9,2),
BONUS     DECIMAL(9,2),
COMM      DECIMAL(9,2),
CONSTRAINT  UNIQUE_LNAME_IN_DEPT UNIQUE (WORKDEPT, LASTNAME))
```

この場合、DEPARTMENT 表は、基本キーの役割を果たす固有の部門番号 (DEPTNO) の列を持ち、次の 2 つの制約関係における親表になります。

REPORTS_TO_EXISTS

自己参照制約。ここでは、DEPARTMENT 表が同じ関係における親表と従属表の両方になっています。ADMRDEPT のすべての非ヌル値は、DEPTNO の値と一致しなければなりません。ある部門は、データベース内の既存の部門の監督下に置かれなければなりません。 DELETE CASCADE 規則は、DEPTNO の値が *n* である行が削除された場合に、ADMRDEPT が *n* であるすべての行も表から削除されることを示しています。

WORKDEPT_EXISTS

EMPLOYEE 表を従属表として設定し、社員の部門割り当て (WORKDEPT) の列を外部キーとして設定します。したがって、WORKDEPT のすべての値は、DEPTNO と一致しなければなりません。 DELETE SET NULL 規則は、DEPTNO の値が *n* である行が DEPARTMENT から削除される場合に、EMPLOYEE 内の WORKDEPT が、その値が *n* であったすべての行でヌル値に設定されることを指定しています。 UPDATE RESTRICT 規則は、EMPLOYEE 内の WORKDEPT に現行の DEPTNO 値と一致する値がある場合に、DEPARTMENT 内の DEPTNO の値が更新できないことを指定しています。

EMPLOYEE 表内の制約 UNIQUE_LNAME_IN_DEPT は、社員の姓が部門内で固有になるようにします。この制約はほとんど用いられませんが、複数の列から構成される制約を表レベルで定義する方法を示しています。

参照制約の除去

ALTER TABLE ステートメントを使用すると、表についての制約を一度に 1 つ追加または除去することができます。除去する制約が参照制約関係における親キーである場合は、この親ファイルとすべての従属ファイルの間の制約も除去されます。

DROP TABLE および DROP SCHEMA ステートメントでも、除去される表またはスキーマに関する制約が除去されます。

例: 制約の除去

次の例では、表 DEPARTMENT 内の DEPTNO 列に対する基本キーを除去します。DEPARTMENT 表と EMPLOYEE 表に関して定義された REPORTS_TO_EXISTS と

WORKDEPT_EXISTS の各制約も除去されます。これは、除去される基本キーがそれらの制約関係における親キーであるためです。

```
ALTER TABLE CORPDATA.EMPLOYEE DROP PRIMARY KEY
```

次の例のように、名前で制約を除去することもできます。

```
ALTER TABLE CORPDATA.DEPARTMENT
            DROP CONSTRAINT UNIQUE_LNAME_IN_DEPT
```

参照制約付き表への挿入

参照制約付きの表にデータを挿入する際に覚えておく必要のある重要なことがいくつかあります。親キーが入っている親表にデータを挿入する場合、SQL は次のものを認めません。

- 重複する親キーの値
- 親キーが基本キーである場合、基本キーの列に入っているヌル値

外部キーが入っている従属表にデータ挿入する場合は、次のとおりです。

- 外部キー列に挿入するそれぞれの非ヌル値は、親表の対応する親キーの値と等しくなければなりません。
- 外部キーのいずれかの列がヌル値である場合は、その外部キー全体がヌル値と見なされます。その列を含むすべての外部キーがヌル値である場合は、INSERT が成功します (固有索引の違反がない限り)。

例: 制約付きのデータ挿入

サンプルの業務プロジェクト表 (PROJECT) を変更して、次の 2 つの外部キーを定義します。

- 部門表を参照する部門番号 (DEPTNO) に対する外部キー
- 社員表を参照する社員番号 (RESPEMP) に対する外部キー

```
ALTER TABLE CORPDATA.PROJECT ADD CONSTRAINT RESP_DEPT_EXISTS
             FOREIGN KEY (DEPTNO)
             REFERENCES CORPDATA.DEPARTMENT
             ON DELETE RESTRICT

ALTER TABLE CORPDATA.PROJECT ADD CONSTRAINT RESP_EMP_EXISTS
             FOREIGN KEY (RESPEMP)
             REFERENCES CORPDATA.EMPLOYEE
             ON DELETE RESTRICT
```

REFERENCES 文節に親表の列が指定されていないことに注意してください。参照される表に親キーとして使用できる基本キーまたは適格な固有キーがある限り、これらの列を指定する必要はありません。

PROJECT 表に挿入されるすべての行には、部門表内の DEPTNO の値と等しい DEPTNO の値が入っていなければなりません。(プロジェクト表の DEPTNO を NOT NULL として定義してあるので、ヌル値は認められません。) さらに、行には、社員表内の EMPNO の値と等しいか、またはヌルの RESPEMP の値が入っていなければなりません。

『付録 A. DB2 UDB for iSeries サンプル表』に記載されているサンプル・データを伴う表は、これらの制約に準拠しています。次の INSERT ステートメントは、DEPARTMENT 表の中に一致する DEPTNO 値 ('A01') がないために失敗します。

```
INSERT INTO  CORPDATA.PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP)
        VALUES  ('AD3120', 'BENEFITS ADMIN', 'A01', '000010')
```

同様に、次の INSERT ステートメントは、EMPLOYEE 表に EMPNO 値 '000011' がないために失敗します。

```
INSERT INTO CORPDATA.PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP)
        VALUES ('AD3130', 'BILLING', 'D21', '000011')
```

次の INSERT ステートメントは、DEPARTMENT 表の中に一致する DEPTNO 値 'E01' があり、EMPLOYEE 表の中に一致する EMPNO 値 '000010' があるため、正常に完了します。

```
INSERT INTO CORPDATA.PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP)
        VALUES ('AD3120', 'BENEFITS ADMIN', 'E01', '000010')
```

参照制約付きの表の更新

親 表を更新する場合は、従属行が存在する基本キーを変更することはできません。このキーを変更すると、従属表の参照制約に違反し、一部の行の親がなくなります。また、基本キーのどの部分にもヌル値を与えることはできません。

更新規則

親表に対して UPDATE が実行されるときに従属表に対して取られる処置は、参照制約に関して指定されている更新規則によって異なります。参照制約に関して更新規則が定義されていない場合には、UPDATE NO ACTION 規則が使用されます。

UPDATE NO ACTION

親表内の行は、その行に他の行が従属していない場合に限り更新できることを指定します。 関係の中に従属行が存在する場合は、UPDATE は失敗します。 従属行の検査はステートメントの終わりに実行されます。

UPDATE RESTRICT

親表内の行は、その行に他の行が従属していない場合に限り更新できることを指定します。 関係の中に従属行が存在する場合は、UPDATE は失敗します。 従属行の検査は直ちに実行されます。

RESTRICT 規則と NO ACTION 規則とのわずかな違いは、トリガーと参照制約の対話を見るとよく分かります。 トリガーは、操作 (この場合は UPDATE ステートメント) の前か後のいずれかに起動するよう定義することができます。 *前トリガー* は UPDATE が実行される前、したがって制約の検査が行われる前に起動します。 *後トリガー* は、UPDATE の実行後、RESTRICT の参照規則 (検査が直ちに実行される) の後で、ただし NO ACTION の制約規則 (検査がステートメントの終わりに実行される) の前に起動します。 トリガーと規則は、次の順序で実施されます。

1. *前トリガー* は、UPDATE の前で、しかも RESTRICT または NO ACTION の制約規則の前に起動します。
2. *後トリガー* は、RESTRICT の制約規則の後で、しかも NO ACTION 規則の前に起動します。

従属 表を更新する場合、変更するヌルでないすべての外部キー値は、その表が従属しているそれぞれの関係についての基本キーと一致しなければなりません。たとえ

ば、社員表内の部門番号は、部門表内の部門番号に従属しています。ある社員に部門なし (ヌル値) を割り当てることはできますが、存在しない部門に割り当てることはできません。

参照制約付きの表に対する UPDATE が失敗すると、更新操作中に行ったすべての変更は無効になります。制約を用いる作業時のコミットメント制御およびジャーナル処理の意味について詳しくは、339 ページの 『ジャーナル処理』および 340 ページの 『コミットメント制御』を参照してください。

例: UPDATE 規則

たとえば、部門表内のある部門番号に、プロジェクト表内の従属行で記述されているプロジェクトに対する責任がある場合は、その部門番号を更新することはできません。

次の UPDATE は、PROJECT 表に、値 'D01' (WHERE ステートメントの対象となっている行) の DEPARTMENT.DEPTNO に従属している行があるため、失敗します。 この UPDATE が許可されると、PROJECT 表と DEPARTMENT 表の間の参照制約が壊れます。

```
UPDATE CORPDATA.DEPARTMENT
        SET DEPTNO = 'D99'
        WHERE DEPTNAME = 'DEVELOPMENT CENTER'
```

次のステートメントは、DEPARTMENT 内の基本キー DEPTNO と PROJECT 内の外部キー DEPTNO の間に存在する参照制約に違反するため、失敗します。

```
UPDATE CORPDATA.PROJECT
        SET DEPTNO = 'D00'
        WHERE DEPTNO = 'D01';
```

このステートメントでは、D01 のすべての部門番号を D00 に変更しようとしています。D00 は DEPARTMENT 内の基本キー DEPTNO の値ではないので、このステートメントは失敗します。

参照制約付き表からの削除

表に基本キーがあって従属関係がない場合は、DELETE は参照制約がない場合と同じように機能します。表に外部キーだけがあって基本キーがない場合も同様です。表に基本キーと従属表がある場合、DELETE は、指定された削除規則に従って行の削除または更新を行います。削除操作が正常に完了するためには、影響されるすべての関係のすべての削除規則が満たされる必要があります。参照制約に違反すると、DELETE が失敗します。

親表に対して DELETE が実行されるときに従属表に対して取られる処置は、参照制約に関して指定されている削除規則によって異なります。削除規則が定義されていない場合には、DELETE NO ACTION 規則が使用されます。

DELETE NO ACTION
親表にある行が、それに依存する他の行がない場合に、削除できることを指定します。 関係の中に従属行が存在する場合は、DELETE は失敗します。従属行の検査はステートメントの終わりに実行されます。

DELETE RESTRICT
親表にある行が、それに依存する他の行がない場合に、削除できることを指

定します。 関係の中に従属行が存在する場合は、DELETE は失敗します。従属行の検査は、即時に実行されます。

たとえば、部門表内のある部門番号に、プロジェクト表内の従属行で記述されているプロジェクトに対する責任がある場合は、その部門番号を削除することはできません。

DELETE CASCADE

親表の指定された行が、最初に削除されることを指定します。その後で、従属行が削除されます。

たとえば、部門表内のある部門の行を削除することにより、その部門を削除することができます。 部門表からその行を削除すると、次のものも削除されます。

- その部門の監督下にあるすべての部門の行
- それらの部門の監督下にあるすべての部門 (以下同様)

DELETE SET NULL

各従属行における外部キーの各ヌル値可能列を省略時値に設定することを指定します。 これは、その列が、削除される行を参照する外部キーのメンバーである場合に限り省略時値に設定されることを意味しています。 影響を受けるのは、すぐ下の従属行だけです。

DELETE SET DEFAULT

各従属行における外部キーの各列をその省略時値に設定することを指示します。 これは、その列が、削除される行を参照する外部キーのメンバーである場合に限り省略時値に設定されることを意味しています。 影響を受けるのは、すぐ下の従属行だけです。

たとえば、ある社員がどこかの部門を管理している場合でも、その社員を社員表 (EMPLOYEE) から削除することができます。 この場合、その管理者の監督下にある各社員の MGRNO の値は、部門表 (DEPARTMENT) 内でブランクに設定されます。 表の作成に関して他の省略時値が指定された場合には、その値が使用されます。

これは、部門表に関して定義された REPORTS_TO_EXISTS 制約によるものです。

下層表に RESTRICT または NO ACTION の削除規則があって、下層の行を削除できないような行が見つかった場合には、DELETE 全体が失敗します。

プログラムでこのステートメントを実行すると、削除された行の数が SQLCA 内の SQLERRD(3) で返されます。この数に含まれるのは、DELETE ステートメントで指定した表内で削除された行の数だけです。CASCADE 規則に従って削除された行は含まれません。 SQLCA 内の SQLERRD(5) には、すべての表内で参照制約による影響を受けた行の数が入ります。

RESTRICT 規則と NO ACTION 規則とのわずかな違いは、トリガーと参照制約の対話を見るとよく分かります。トリガーは、操作 (この場合は DELETE ステートメント) の前か後のいずれかに起動するよう定義することができます。*前トリガー* は DELETE が実行される前、したがって制約の検査が行われる前に起動します。*後トリガー* は、DELETE の実行後、RESTRICT の制約規則 (検査が直ちに実行される)

の後で、ただし NO ACTION の制約規則 (検査がステートメントの終わりに実行される) の前に起動します。トリガーと規則は、次の順序で実施されます。

1. *前トリガー* は、DELETE の前で、しかも RESTRICT または NO ACTION の制約規則の前に起動します。
2. *後トリガー* は、RESTRICT の制約規則の後で、しかも NO ACTION 規則の前に起動します。

例: DELETE カスケード規則

DEPARTMENT 表からある部門を削除すると、その部門に割り当てられているすべての社員について (EMPLOYEE 表内の) WORKDEPT がヌル値に設定されます。次の DELETE ステートメントについて検討してください。

```
DELETE FROM CORPDATA.DEPARTMENT
       WHERE DEPTNO = 'E11'
```

『付録 A. DB2 UDB for iSeries サンプル表』に記載されている表とデータがあるとすれば、DEPARTMENT 表から 1 行が削除されます。さらに、EMPLOYEE 表が更新されて、値が 'E11' になっている WORKDEPT がヌル値に設定されます。以下のサンプル・データ内の疑問符 ('?') は、ヌル値を表しています。結果は、次のようになります。

表 22. DEPARTMENT 表: DELETE ステートメント完了後の表の内容

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT
A00	SPIFFY コンピューター・サービス事業部	000010	A00
B01	計画	000020	A00
C01	情報センター	000030	A00
D01	開発センター	?	A00
D11	製造システム	000060	D01
D21	管理システム	000070	D01
E01	サポート・サービス	000050	A00
E21	ソフトウェア・サポート	000100	E01
F22	BRANCH OFFICE F2	?	E01
G22	BRANCH OFFICE G2	?	E01
H22	BRANCH OFFICE H2	?	E01
I22	BRANCH OFFICE I2	?	E01
J22	BRANCH OFFICE J2	?	E01

部門 'E11' の監督下にある部門はないため、DEPARTMENT 表にはカスケードされた削除がないことに注意してください。

以下の 2 つの表は、DELETE ステートメントの完了前と完了後の EMPLOYEE 表の、1 つの影響を受ける部分のスナップショットです。

表 23. 部分的な EMPLOYEE 表: DELETE ステートメントの前の内容の一部

EMPNO	FIRSTNME	MI	LASTNAME	WORKDEPT	PHONENO	HIREDATE
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05

表 23. 部分的な EMPLOYEE 表 (続き): DELETE ステートメントの前の内容の一部

EMPNO	FIRSTNME	MI	LASTNAME	WORKDEPT	PHONENO	HIREDATE
000250	DANIEL	S	SMITH	D21	0961	1960-10-30
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11
000270	MARIA	L	PEREZ	D21	9001	1980-09-30
000280	ETHEL	R	SCHNEIDER	E11	0997	1967-03-24
000290	JOHN	R	PARKER	E11	4502	1980-05-30
000300	PHILIP	X	SMITH	E11	2095	1972-06-19
000310	MAUDE	F	SETRIGHT	E11	3332	1964-09-12
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07
000330	WING		LEE	E21	2103	1976-02-23
000340	JASON	R	GOUNOT	E21	5696	1947-05-05

表 24. 部分的な EMPLOYEE 表: DELETE ステートメントの後の内容の一部

EMPNO	FIRSTNME	MI	LASTNAME	WORKDEPT	PHONENO	HIREDATE
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05
000250	DANIEL	S	SMITH	D21	0961	1960-10-30
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11
000270	MARIA	L	PEREZ	D21	9001	1980-09-30
000280	ETHEL	R	SCHNEIDER	?	0997	1967-03-24
000290	JOHN	R	PARKER	?	4502	1980-05-30
000300	PHILIP	X	SMITH	?	2095	1972-06-19
000310	MAUDE	F	SETRIGHT	?	3332	1964-09-12
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07
000330	WING		LEE	E21	2103	1976-02-23
000340	JASON	R	GOUNOT	E21	5696	1947-05-05

検査保留

参照制約および検査制約は、潜在的な制約違反が存在する検査保留と呼ばれる状態にすることができます。参照制約の場合、親キーと外部キーの間に潜在的な不一致が存在するときに、違反が生じます。検査制約の場合、検査制約によって制限されている潜在的な値が列に存在するときに、違反が生じます。システムが、制約が違反されている可能性があると判別すると (復元操作の後などに)、制約は検査保留としてマークされます。この場合、制約に関係する表の使用について制限が課せられます。参照制約の場合は、次の制限が適用されます。

- 従属ファイルでは、入出力操作が許可されません。
- 親ファイルでは、読み取り操作と挿入操作しか許可されません。

検査制約が検査保留状態にあるときは、次の制限が適用されます。

- ファイルに対する読み取り操作が許可されません。
- 挿入と更新が許可され、制約が適用されます。

制約を検査保留状態から解除するには、次のことを行う必要があります。

1. 物理ファイル制約の変更 (CHGPFCST) CL コマンドを用いて関係を使用不能にします。
2. 参照制約のキー (外部または親、あるいはその両方) データ、または検査制約の列データを訂正します。
3. CHGPFCST CL コマンドを用いて制約を再び使用可能にします。

検査保留制約の表示 (DSPCPCST) CL コマンドを使用すると、制約違反の対象となっている行を識別することができます。

検査保留状態の表を用いる作業の詳細については、データベース・プログラミングを参照してください。

視点に関する WITH CHECK OPTION

WITH CHECK OPTION は、CREATE VIEW ステートメントの任意指定の文節であり、視点を介するデータの挿入または更新時に行われる検査のレベルを指定します。このオプションを指定する場合、視点を介して挿入または更新されるすべての行は、視点の定義に従う必要があります。

WITH CHECK OPTION は、視点が読み取り専用である場合には指定できません。視点の定義には、副照会を組み込んではなりません。

WITH CHECK OPTION 文節を指定しないで視点を作成する場合、視点に対して実行される挿入および更新操作は、視点の定義に準拠しているかどうかについて検査されません。それでも、視点が WITH CHECK OPTION を含む別の視点に直接または間接的に従属している場合は、何らかの検査が行われる可能性があります。視点の定義が使用されないため、視点の定義に準拠していない行が視点を介して挿入または更新される可能性があります。これは、その視点を使用してこれらの行を再び選択できないことを意味します。

検査は、『WITH CASCADED CHECK OPTION』または 156 ページの『WITH LOCAL CHECK OPTION』のどちらを使用してもかまいません。 WITH CHECK OPTION の詳細については、SQL 解説書の CREATE VIEW のトピックを参照してください。

WITH CASCADED CHECK OPTION

WITH CASCADED CHECK OPTION は、視点を通して挿入または更新されるすべての行が視点の定義に準拠していなければならないことを指定します。また、すべての従属視点の検索条件は、行の挿入または更新時に検査されます。行が視点の定義に準拠していない場合は、視点を使用してその行を取り出すことはできません。

たとえば、次の更新可能な視点について検討してください。

```
CREATE VIEW V1 AS SELECT COL1
       FROM T1 WHERE COL1 > 10
```

WITH CHECK OPTION が指定されていないため、次の INSERT ステートメントは、挿入される値が視点の検索条件に適合しない場合でも成功します。

```
INSERT INTO V1 VALUES (5)
```

V1 に基づいて、WITH CASCADED CHECK OPTION を指定して別の視点を作成します。

```
CREATE VIEW V2 AS SELECT COL1
          FROM V1 WITH CASCADED CHECK OPTION
```

次の INSERT ステートメントは、V2 の定義に準拠しない行をもたらすために失敗します。

```
INSERT INTO V2 VALUES (5)
```

V2 に基づいて作成するもう 1 つの視点について検討してください。

```
CREATE VIEW V3 AS SELECT COL1
          FROM V2 WHERE COL1 < 100
```

次の INSERT ステートメントは、V3 が V2 に従属していて、V2 に WITH CASCADED CHECK OPTION が組み込まれているために失敗します。

```
INSERT INTO V3 VALUES (5)
```

ただし、次の INSERT ステートメントは、V2 の定義に準拠しているために成功します。V3 には WITH CASCADED CHECK OPTION が組み込まれていないため、ステートメントが V3 の定義に適合していないことは関係ありません。

```
INSERT INTO V3 VALUES (200)
```

WITH LOCAL CHECK OPTION

WITH LOCAL CHECK OPTION は、ある行を更新して、これ以上視点を通してその行を取り出せないようにすることができる点を除けば、WITH CASCADED CHECK と同じです。これは、視点が WITH CHECK OPTION 文節を指定しないで定義された視点に、直接または間接的に依存している場合にだけ起きます。

たとえば、前の例で使用したのと同じ更新可能な視点について考えてみます。

```
CREATE VIEW V1 AS SELECT COL1
          FROM T1 WHERE COL1 > 10
```

V1 に基づいて 2 番目の視点を作成し、今回は WITH LOCAL CHECK OPTION を指定しましょう。

```
CREATE VIEW V2 AS SELECT COL1
          FROM V1 WITH LOCAL CHECK OPTION
```

前の CASCADED CHECK OPTION の例で失敗した同じ INSERT が、今回は成功します。これは、V2 に検索条件がない上に、V1 に検査オプションが指定されていないので V1 の検索条件を検査する必要がないためです。

```
INSERT INTO V2 VALUES (5)
```

ここで再び、V2 に基づいて作成したもう 1 つの視点について考えてみましょう。

```
CREATE VIEW V3 AS SELECT COL1
     FROM V2 WHERE COL1 < 100
```

次の INSERT もやはり成功します。これは、V2 について WITH LOCAL CHECK OPTION が指定されているため、前の例の WITH CASCADED CHECK OPTION の場合とは異なり、V1 の検索条件が検査されないためです。

```
INSERT INTO V3 VALUES (5)
```

LOCAL CHECK OPTION と CASCADED CHECK OPTION の違いは、行の挿入または更新時に検査される従属視点の検索条件の数にあります。

- WITH LOCAL CHECK OPTION は、行の挿入または更新時に、WITH LOCAL CHECK OPTION または WITH CASCADED CHECK OPTION がある従属視点の検索条件だけを検査するよう指定します。
- WITH CASCADED CHECK OPTION は、行の挿入または更新時に、すべての従属視点の検索条件を検査するよう指定します。

例: カスケード検査オプション

次の表と視点を使用します。

```
CREATE TABLE T1 (COL1 CHAR(10))

CREATE VIEW  V1 AS SELECT COL1
      FROM T1 WHERE COL1 LIKE 'A%'

CREATE VIEW V2 AS SELECT COL1
      FROM V1 WHERE COL1 LIKE '%Z'
                           WITH LOCAL CHECK OPTION

CREATE VIEW V3 AS SELECT COL1
      FROM V2 WHERE COL1 LIKE 'AB%'

CREATE VIEW V4 AS SELECT COL1
      FROM V3 WHERE COL1 LIKE '%YZ'
                            WITH CASCADED CHECK OPTION

CREATE VIEW V5 AS SELECT COL1
      FROM V4 WHERE COL1 LIKE 'ABC%'
```

INSERT または UPDATE でどの視点が操作されるかによって、検査される検索条件が異なります。

- V1 が操作される場合は、V1 に WITH CHECK OPTION が指定されていないため、条件は検査されません。
- V2 が操作される場合は、次のとおりです。
  - COL1 は、文字 Z で終わらなければなりませんが、文字 A で始まる必要はありません。これは、検査オプションが LOCAL であり、視点 V1 に検査オプションが指定されていないためです。
- V3 が操作される場合は、次のとおりです。
  - COL1 は、文字 Z で終わらなければなりませんが、文字 A で始まる必要はありません。 V3 には検査オプションが指定されていないため、その独自の検索条件が満たされる必要はありません。ただし、V3 は V2 に基づいて定義されており、V2 には検査オプションがあるので、V2 の検索条件は検査される必要があります。
- V4 が操作される場合は、次のとおりです。
  - COL1 は、'AB' で始まって 'YZ' で終わらなければなりません。V4 には WITH CASCADED CHECK OPTION が指定されているため、V4 が依存しているすべての視点に関するすべての検索条件が検査される必要があります。
- V5 が操作される場合は、次のとおりです。
  - COL1 は、'AB' で始まらなければなりませんが、必ずしも 'ABC' である必要はありません。これは、V5 に検査オプションが指定されていないので、その独自の検索条件が検査される必要がないためです。ただし、V5 は V4 に基づ

いて定義されており、V4 にはカスケード検査オプションがあるため、V4、V3、V2、および V1 に関するすべての検索条件が検査されることが必要です。すなわち、COL1 は、'AB' で始まって 'YZ' で終わらなければなりません。

V5 を WITH LOCAL CHECK OPTION を指定して作成した場合、V5 を操作することは、COL1 が 'ABC' で始まり、'YZ' で終わる必要があることを意味します。LOCAL CHECK OPTION は、3 番目の文字が 'C' でなければならないという新たな要件を追加します。

DB2 UDB for iSeries トリガー・サポート

トリガー は、指定した表に対して指定した変更操作が実行されるときに自動的に実行される一連のアクションです。変更操作は、SQL の INSERT、UPDATE、または DELETE ステートメント、あるいは、アプリケーション・プログラム内の高水準言語の挿入、更新、または削除ステートメントのどちらであってもかまいません。トリガーは、業務に関する規則の適用、入力データの妥当性検査、および監査証跡の保管などの作業に役立ちます。

トリガーは、以下の 2 つの方法で定義できます。

- 159 ページの『SQL トリガー』
- 163 ページの『外部トリガー』

外部トリガーの場合は、CRTPFTRG CL コマンドが使用されます。 一連のトリガー・アクションを含むプログラムは、サポートされているどの高水準言語ででも定義できます。外部トリガーは、挿入トリガー、更新トリガー、削除トリガー、または読み取りトリガーになることができます。

SQL トリガーの場合は、CREATE TRIGGER ステートメントが使用されます。トリガー・プログラムはすべて、SQL を使用して定義されます。 SQL トリガーは、挿入トリガー、更新トリガー、または削除トリガーになることができます。

トリガーが表に関連付けられると、表、あるいは、その表に基づいて作成されたすべての論理ファイルまたは視点に対して変更操作が開始されるたびに、トリガー・サポートによりトリガー・プログラムが呼び出されます。SQL トリガーと外部トリガーは、同じ表に定義できます。 1 つの表に、最大 200 のトリガーが定義できます。

それぞれの変更操作では、変更操作の前または後にトリガーを呼び出すことができます。さらに、表がアクセスされるたびに呼び出される *読み取り* トリガーを追加することができます。したがって、1 つの表は、以下の多くのタイプのトリガーに関連付けることができます。

- 削除前トリガー
- 挿入前トリガー
- 更新前トリガー
- 削除後トリガー
- 挿入後トリガー
- 更新後トリガー

- 読み取り専用トリガー (外部トリガーのみ)

トリガーの制限についての説明、すなわち、1 つの SQL 表に定義できるトリガーの数、トリガーの最大ネスティング・レベル、トリガーをコーディングするときの推奨事項と注意事項などの説明については、「データベース・プログラミング」のデータベース内での自動イベントのトリガーの章を参照してください。

SQL トリガー

SQL CREATE TRIGGER ステートメントは、データベース管理システムが、挿入、更新、または削除操作が実行されるたびに、表のグループを制御、モニター、管理する方法を提供します。SQL の挿入、更新、または、削除操作が実行されるたびに、SQL トリガーに指定されたステートメントが実行されます。トリガーが実行されるときに、SQL トリガーは、ストアード・プロシージャーまたはユーザー定義関数を呼び出して、追加処理を実行します。

ストアード・プロシージャーとは異なり、SQL トリガーは、アプリケーションから直接呼び出すことはできません。代わりに、SQL トリガーは、挿入、更新、削除操作のトリガーが実行されると、データベース管理システムによって呼び出されます。SQL トリガーの定義はデータベース管理システムに保管されており、トリガーが定義されている SQL 表が変更されると、データベース管理システムによって呼び出されます。

SQL トリガーの作成について詳しくは、『SQL トリガーの作成』を参照してください。

CREATE TRIGGER ステートメントの使用についての詳細な説明は、「SQL 解説書」のCREATE TRIGGER ステートメントを参照してください。

SQL トリガーの作成

SQL トリガーは、CREATE TRIGGER SQL ステートメントを指定して作成することができます。 SQL トリガーのルーチン本体の中のステートメントは、SQL によってプログラム・オブジェクト (*PGM) に変換されます。 プログラムは、トリガー名修飾子で指定されるスキーマに作成されます。 指定されたトリガーは、SYSTRIGGERS、SYSTRIGDEP、SYSTRIGCOL、および SYSTRIGUPD の各 SQL カタログに登録されます。 SQL トリガーの中の変数制御ステートメントの使用方法、および SQL ステートメント・レベルでの SQL トリガーのデバッグ方法の詳細については、「SQL 解説書」の SQL プロシージャー、関数、およびトリガーの章を参照してください。

SQL トリガーの作成の例と考慮事項については、以下のトピックを参照してください。

- 160 ページの『BEFORE SQL トリガー』
- 161 ページの『AFTER SQL トリガー』
- 162 ページの『SQL トリガーのハンドラー』
- 163 ページの『SQL トリガーの遷移表』

BEFORE SQL トリガー

BEFORE トリガーは、表を変更することはできませんが、入力列の値を検査したり、表に挿入または更新された列値を変更するのに使用することができます。 以下の例では、トリガーが、行をターゲット表に挿入する前に、会社の会計上の四半期 (fiscal quarter) をセットするのに使用されています。

```
CREATE TABLE TransactionTable (DateOfTransaction DATE, FiscalQuarter SMALLINT)

CREATE TRIGGER TransactionBeforeTrigger BEFORE INSERT ON TransactionTable
REFERENCING NEW AS new_row
FOR EACH ROW MODE DB2ROW
BEGIN
  DECLARE newmonth SMALLINT;
SET newmonth = MONTH(new_row.DateOfTransaction);
  IF newmonth < 4 THEN
    SET new_row.FiscalQuarter=3;
  ELSEIF newmonth < 7 THEN
    SET new_row.FiscalQuarter=4;
  ELSEIF newmonth < 10 THEN
    SET new_row.FiscalQuarter=1;
  ELSE
    SET new_row.FiscalQuarter=2;
  END IF;
END
```

次の SQL 挿入ステートメントでは、現在日付が November 14, 2000 であれば、″FiscalQuarter″ 列は 2 にセットされます。

```
INSERT INTO TransactionTable(DateOfTransaction)
          VALUES(CURRENT DATE)
```

SQL トリガーは、User-defined Distinct Types (UDT) (ユーザー定義特殊タイプ) およびストアード・プロシージャーにアクセスし、使用することができます。次の例では、SQL トリガーはストアード・プロシージャーを呼び出して、事前に定義されたビジネス・ロジック、このケースでは、業務の事前定義値に列をセットするロジックを実行します。

```
CREATE DISTINCT TYPE enginesize AS DECIMAL(5,2) WITH COMPARISONS

CREATE DISTINCT TYPE engineclass AS VARCHAR(25) WITH COMPARISONS

CREATE PROCEDURE SetEngineClass(IN SizeInLiters enginesize,
                                OUT CLASS engineclass)
LANGUAGE SQL CONTAINS SQL
BEGIN
  IF SizeInLiters<2.0 THEN
    SET CLASS = 'Mouse';
  ELSEIF SizeInLiters<3.1 THEN
    SET CLASS ='Economy Class';
  ELSEIF SizeInLiters<4.0 THEN
    SET CLASS ='Most Common Class';
  ELSEIF SizeInLiters<4.6 THEN
    SET CLASS = 'Getting Expensive';
  ELSE
    SET CLASS ='Stop Often for Fillups';
  END IF;
END

CREATE TABLE EngineRatings (VariousSizes enginesize, ClassRating engineclass)

CREATE TRIGGER SetEngineClassTrigger BEFORE INSERT ON EngineRatings
```

```
REFERENCING NEW AS new_row
FOR EACH ROW MODE DB2ROW
  CALL SetEngineClass(new_row.VariousSizes, new_row.ClassRating)
```

次の SQL 挿入ステートメントでは、″VariousSizes″ 列の値が 3.0 の場合、″ClassRating″ 列は ″Economy Class″ にセットされます。

```
INSERT INTO EngineRatings(VariousSizes) VALUES(3.0)
```

SQL では、SQL トリガーを作成する前に、すべての表、ユーザー定義関数、プロシージャー、および、ユーザー定義タイプが存在していることが必要です。上の例では、トリガーが作成される前に、表、ストアード・プロシージャー、およびユーザー定義タイプがすべて定義されています。

AFTER SQL トリガー

SQL トリガーの中で WHEN 条件を使用して、条件を指定することができます。条件の評価の結果が true (真) の場合は、SQL トリガーのルーチン本体の中の SQL ステートメントが実行されます。条件が false (偽) の場合は、SQL トリガーのルーチン本体の中の SQL ステートメントは実行されず、制御はデータベース・システムに戻されます。次の例では、照会が評価され、トリガーが活動化されたときに、トリガー・ルーチン本体内のステートメントを実行するべきかどうかが決められます。

```
CREATE TABLE TodaysRecords(TodaysMaxBarometricPressure FLOAT,
  TodaysMinBarometricPressure FLOAT)

CREATE TABLE OurCitysRecords(RecordMaxBarometricPressure FLOAT,
  RecordMinBarometricPressure FLOAT)

CREATE TRIGGER UpdateMaxPressureTrigger
AFTER UPDATE OF TodaysMaxBarometricPressure ON TodaysRecords
REFERENCING NEW AS new_row
FOR EACH ROW MODE DB2ROW
WHEN (new_row.TodaysMaxBarometricPressure>
     (SELECT MAX(RecordMaxBarometricPressure) FROM
     OurCitysRecords))
  UPDATE OurCitysRecords
         SET RecordMaxBarometricPressure =
             new_row.TodaysMaxBarometricPressure

CREATE TRIGGER UpdateMinPressureTrigger
AFTER UPDATE OF TodaysMinBarometricPressure
ON TodaysRecords
REFERENCING NEW AS new_row
FOR EACH ROW MODE DB2ROW
WHEN(new_row.TodaysMinBarometricPressure<
     (SELECT MIN(RecordMinBarometricPressure) FROM
     OurCitysRecords))
  UPDATE OurCitysRecords
         SET RecordMinBarometricPressure =
             new_row.TodaysMinBarometricPressure
```

まず、表の現行値が初期設定されます。

```
INSERT INTO TodaysRecords VALUES(0.0,0.0)
INSERT INTO OurCitysRecords VALUES(0.0,0.0)
```

次の SQL 更新ステートメントでは、OurCitysRecords の中の RecordMaxBarometricPressure が、UpdateMaxPressureTrigger によって更新されます。

```
UPDATE TodaysRecords SET TodaysMaxBarometricPressure = 29.95
```

しかし、翌日、TodaysMaxBarometricPressure が 29.91 でしかなかった場合は、RecordMaxBarometricPressure は更新されません。

```
UPDATE TodaysRecords SET TodaysMaxBarometricPressure = 29.91
```

SQL では、1 つのトリガー・アクションに複数のトリガーを定義することができます。上の例では、2 つの AFTER UPDATE トリガー、すなわち、UpdateMaxPressureTrigger と UpdateMinPressureTrigger があります。これらのトリガーが活動化されるのは、表 TodaysRecords の特定の列が更新されたときだけです。

AFTER トリガーは表を変更することができます。上の例では、2 番目の表に UPDATE 操作が行われています。挿入と更新の操作を繰り返し行うことは避けてください。トリガーのネスティング・レベルが最大に達すると、データベース管理システムは操作を終了します。挿入と更新の反復操作は、最大ネスティング・レベルになる前に挿入/更新操作を終了するように条件ロジックを追加することによって、避けることができます。トリガーのネットワークでカスケードが反復して起こるようなトリガー・ネットワークでは、同様の状態を避ける必要があります。

SQL トリガーのハンドラー

SQL トリガーの中のハンドラーによって、SQL トリガーは、トリガー・ルーチン本体の SQL ステートメントの実行中に生じたエラーまたはエラーに関するログ情報から回復する機能が持てるようになります。

次の例には、2 つのハンドラーが定義されています。1 つはオーバーフロー条件を処理し、2 番目のハンドラーは SQL 例外を処理するものです。

```
CREATE TABLE ExcessInventory(Description VARCHAR(50), ItemWeight SMALLINT)

CREATE TABLE YearToDateTotals(TotalWeight SMALLINT)

CREATE TABLE FailureLog(Item VARCHAR(50), ErrorMessage VARCHAR(50), ErrorCode INT)

CREATE TRIGGER InventoryDeleteTrigger
AFTER DELETE ON ExcessInventory
REFERENCING OLD AS old_row
FOR EACH ROW MODE DB2ROW
BEGIN
  DECLARE sqlcode INT;
  DECLARE invalid_number condition FOR '22003';
  DECLARE exit handler FOR invalid_number
  INSERT INTO FailureLog VALUES(old_row.Description,
        'Overflow occurred in YearToDateTotals', sqlcode);
  DECLARE exit handler FOR sqlexception
  INSERT INTO FailureLog VALUES(old_row.Description,
        'SQL Error occurred in InventoryDeleteTrigger', sqlcode);
  UPDATE YearToDateTotals SET TotalWeight=TotalWeight +
        old_row.itemWeight;
END
```

まず、表の現行値が初期設定されます。

```
INSERT INTO ExcessInventory VALUES('Desks',32500)
INSERT INTO ExcessInventory VALUES('Chairs',500)
INSERT INTO YearToDateTotals VALUES(0)
```

次の例で、最初の SQL 削除ステートメントが実行されると、品目 ″Desks″ の ItemWeight が、表 YearToDateTotals の TotalWeight の列合計に加算されます。 2 番目の SQL 削除ステートメントが実行されるとき、品目 ″Chairs″ の ItemWeight

が、TotalWeight の列合計に加算されるとオーバーフローが起こります。これは、この列が 32767 までの値しか処理しないからです。オーバーフローが起こると、invalid_number 出口ハンドラーが実行され、1 つの行が FailureLog 表に書き込まれます。YearToDateTotals 表が不慮の事故によって削除された場合は、たとえば sqlexception 出口ハンドラーが実行されます。この例では、後で問題を診断できるように、ハンドラーを使用してログが書き込まれます。

```
DELETE FROM ExcessInventory WHERE Description='Desks'
DELETE FROM ExcessInventory WHERE Description='Chairs'
```

SQL トリガーの遷移表

SQL トリガーは、SQL 挿入、更新、または削除操作の際に、影響を受けた行のすべてを参照する必要があります。これは、たとえば、トリガーが、影響を受けた行の特定の列に、MIN または MAX のような集合関数を適用しなければならない場合に当てはまります。OLD_TABLE および NEW_TABLE 遷移表が、この目的のために使用できます。次の例では、トリガーは、集合関数 MAX を、表 StudentProfiles の、影響を受けた行のすべてに適用します。

```
CREATE TABLE StudentProfiles(StudentsName VARCHAR(125),
      StudentsYearInSchool SMALLINT, StudentsGPA DECIMAL(5,2))

CREATE TABLE CollegeBoundStudentsProfile
      (YearInSchoolMin SMALLINT, YearInSchoolMax SMALLINT, StudentGPAMin
      DECIMAL(5,2), StudentGPAMax DECIMAL(5,2))

CREATE TRIGGER UpdateCollegeBoundStudentsProfileTrigger
AFTER UPDATE ON StudentProfiles
REFERENCING NEW_TABLE AS ntable
FOR EACH STATEMENT MODE DB2SQL
BEGIN
  DECLARE maxStudentYearInSchool SMALLINT;
  SET maxStudentYearInSchool =
       (SELECT MAX(StudentsYearInSchool) FROM ntable);
  IF maxStudentYearInSchool >
       (SELECT MAX (YearInSchoolMax) FROM
          CollegeBoundStudentsProfile) THEN
    UPDATE CollegeBoundStudentsProfile SET YearInSchoolMax =
         maxStudentYearInSchool;
  END IF;
END
```

上の例では、更新ステートメントのトリガーの実行の後で、トリガーが一回実行されます。これは、このトリガーが、FOR EACH STATEMENT トリガーとして定義されているからです。遷移表を参照するトリガーを定義するときには、データベース管理システムが遷移表にデータを満たしていくのにかかる処理オーバーヘッドについて考慮する必要があります。

外部トリガー

外部トリガーでは、一連のトリガー・アクションを含むプログラムは、*PGM オブジェクトを作成する、サポートされているどの高水準言語ででも定義できます。トリガー・プログラムには、SQL を組み込むことができます。外部トリガーを定義するには、トリガー・プログラムを作成して ADDPFTRG CL コマンドで表に追加するか、または iSeries ナビゲーターを使用して追加します。トリガーを表に追加するには、次のことを行う必要があります。

- 表を識別する

- 操作の種類を識別する
- 必要なアクションを実行するプログラムを識別する

外部トリガーの例については、『外部トリガーのプログラム例』を参照してください。

外部トリガーのプログラム例

以下に、外部トリガー・サンプル・プログラムを示します。 このプログラムは、ILE C で書かれており、組み込み SQL を含みます。

DB2 UDB for iSeries での外部トリガーの使用の詳細な説明と例については、「データベース・プログラミング」のデータベース内での自動イベントのトリガーの章を参照してください。

注: コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

```
#include "string.h"
#include "stdlib.h"
#include "stdio.h"
#include <recio.h>
#include <xxcvt.h>
#include "qsysinc/h/trgbuf"       /* Trigger input parameter         */
#include "lib1/csrc/msghand1"     /* User defined message handler    */
/*********************************************************************/
/* This is a trigger program which is called whenever there is an    */
/* update to the EMPLOYEE table. If the employee's commission is     */
/* greater than the maximum commission, this trigger program will    */
/* increase the employee's salary by 1.04 percent and insert into    */
/* the RAISE table.                                                  */
/*                                                                   */
/* The EMPLOYEE record information is passed from the input parameter*/
/* to this trigger program.                                          */
/*********************************************************************/

Qdb_Trigger_Buffer_t *hstruct;
char *datapt;

   /*****************************************************/
   /* Structure of the EMPLOYEE record which is used to   */
   /* store the old or the new record that is passed to   */
   /* this trigger program.                               */
   /*                                                     */
   /* Note : You must ensure that all the numeric fields  */
   /*        are aligned at 4 byte boundary in C.         */
   /*        Used either Packed struct or filler to reach */
   /*        the byte boundary alignment.                 */
   /*****************************************************/

_Packed struct rec{
         char  empn[6];
  _Packed struct { short fstlen ;
                   char  fstnam[12];
                 } fstname;
         char  minit[1];
_Packed struct { short lstlen;
                 char lstnam[15];
               } lstname;
       char  dept[3];
       char  phone[4];
       char  hdate[10];
       char  jobn[8];
       short edclvl;
       char  sex1[1];
       char  bdate[10];
       decimal(9,2) salary1;
       decimal(9,2) bonus1;
       decimal(9,2) comm1;
       } oldbuf, newbuf;
EXEC SQL INCLUDE SQLCA;
```

図 2. サンプル・トリガー・プログラム (1/5)

```
main(int argc, char **argv)
{
int i;
int obufoff;                        /* old buffer offset            */
int nuloff;                         /* old null byte map offset     */
int nbufoff;                        /* new buffer offset            */
int nul2off;                        /* new null byte map offset     */
short work_days  = 253;             /* work days during in one year */
decimal(9,2) commission = 2000.00; /* cutoff to qualify for        */
decimal(9,2) percentage = 1.04;    /* raised salary as percentage  */
char raise_date[12] = "1982-06-01";/* effective raise date         */


struct {
       char  empno[6];
       char  name[30];
       decimal(9,2) salary;
       decimal(9,2) new_salary;
       } rpt1;

   /*****************************************************/
   /* Start to monitor any exception.                   */
   /*****************************************************/

  _FEEDBACK fc;
  _HDLR_ENTRY hdlr = main_handler;
                            /*************************************/
                            /* Make the exception handler active.   */
                            /*************************************/
  CEEHDLR(&hdlr, NULL, &fc);
                            /*************************************/
                            /* Ensure exception handler OK          */
                            /*************************************/
  if (fc.MsgNo != CEE0000)
    {
      printf("Failed to register exception handler.\n");
      exit(99);
    };

   /*****************************************************/
   /* Move the data from the trigger buffer to the local  */
   /* structure for reference.                          */
   /*****************************************************/

hstruct = (Qdb_Trigger_Buffer_t *)argv[1];
datapt  = (char *) hstruct;

obufoff = hstruct ->Old_Record_Offset;       /* old buffer   */
memcpy(&oldbuf,datapt+obufoff,; hstruct->Old_Record_Len);

nbufoff = hstruct ->New_Record_Offset;       /* new buffer   */
memcpy(&newbuf,datapt+nbufoff,; hstruct->New_Record_Len);
```

図 2. サンプル・トリガー・プログラム (2/5)

```
EXEC SQL WHENEVER SQLERROR  GO TO ERR_EXIT;

   /*******************************************************/
   /* Set the transaction isolation level to the same as  */
   /* the application based on the input parameter in the */
   /* trigger buffer.                                     */
   /*******************************************************/

if(strcmp(hstruct->Commit_Lock_Level,"0") == 0)
     EXEC SQL SET TRANSACTION ISOLATION LEVEL NONE;
else{
  if(strcmp(hstruct->Commit_Lock_Level,"1") == 0)
       EXEC SQL SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED, READ
                 WRITE;
  else {
     if(strcmp(hstruct->Commit_Lock_Level,"2") == 0)
       EXEC SQL SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
  else
     if(strcmp(hstruct->Commit_Lock_Level,"3") == 0)
       EXEC SQL SET TRANSACTION ISOLATION LEVEL ALL;
  }
}

 /*******************************************************/
 /* If the employee's commission is greater than maximum */
 /* commission, then increase the employee's salary      */
 /* by 1.04 percent and insert into the RAISE table.     */
 /*******************************************************/

if (newbuf.comm1 >= commission)
{
  EXEC SQL  SELECT EMPNO, EMPNAME, SALARY
          INTO :rpt1.empno, :rpt1.name, :rpt1.salary
          FROM TRGPERF/EMP_ACT
          WHERE EMP_ACT.EMPNO=:newbuf.empn ;

  if (sqlca.sqlcode == 0) then
  {
     rpt1.new_salary = salary * percentage;
     EXEC SQL INSERT INTO TRGPERF/RAISE VALUES(:rpt1);
  }
  goto finished;
 }
 err_exit:
   exit(1);

  /* All done  */
 finished:
  return;
}  /* end of main line  */
```

図 2. サンプル・トリガー・プログラム (3/5)

```
/*****************************************************************/
/*   INCLUDE NAME : MSGHAND1                                     */
/*                                                               */
/*   DESCRIPTION  : Message handler to signal an exception to    */
/*                  the application to inform that an            */
/*                  error occured in the trigger program.        */
/*                                                               */
/*   NOTE : This message handler is a user defined routine.      */
/*                                                               */
/*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>
#include <leawi.h>

#pragma linkage (QMHSNDPM, OS)
void QMHSNDPM(char *,             /* Message identifier             */
              void *,             /* Qualified message file name    */
              void *,             /* Message data or text           */
              int,                /* Length of message data or text */
              char *,             /* Message type                   */
              char *,             /* Call message queue             */
              int,                /* Call stack counter             */
              void *,             /* Message key                    */
              void *,             /* Error code                     */
              ...);               /* Optionals:
                                       length of call message queue
                                        name
                                       Call stack entry qualification
                                       display external messages
                                        screen wait time           */
/*********************************************************************/
/********  This is the start of the exception handler function.      */
/*********************************************************************/
  void main_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc,
                    _FEEDBACK *new)
   {
                           /***************************************/
                           /* Initialize variables for call to    */
                           /* QMHSNDPM.                           */
                           /* User must create a message file and  */
                           /* define a message ID to match the    */
                           /* following data.                     */
                           /***************************************/
  char      message_id[7] = "TRG9999";
  char      message_file[20] = "MSGF      LIB1      ";
  char      message_data[50] = "Trigger error                " ;
  int       message_len = 30;
  char      message_type[10] = "*ESCAPE   ";
  char      message_q[10] = "_C_pep    ";
  int       pgm_stack_cnt = 1;
  char      message_key[4];
```

図 2. サンプル・トリガー・プログラム (4/5)

```
                          /****************************************/
                          /* Declare error code structure for     */
                          /* QMHSNDPM.                            */
                          /****************************************/
struct error_code {
  int bytes_provided;
  int bytes_available;
  char message_id[7];
} error_code;

error_code.bytes_provided = 15;
                          /****************************************/
                          /* Set the error handler to resume and  */
                          /* mark the last escape message as      */
                          /* handled.                             */
                          /****************************************/
*rc = CEE_HDLR_RESUME;
                          /****************************************/
                          /* Send my own *ESCAPE message.         */
                          /****************************************/
QMHSNDPM(message_id,
        &message_file,
        &message_data,
         message_len,
         message_type,
         message_q,
         pgm_stack_cnt,
        &message_key,
        &error_code );
                          /****************************************/
                          /* Check that the call to QMHSNDPM      */
                          /* finished correctly.                  */
                          /****************************************/
if (error_code.bytes_available != 0)
    {
      printf("Error in QMHOVPM : %s\n", error_code.message_id);
    }
}
```

図 2. サンプル・トリガー・プログラム (5/5)

第 11 章 ストアード・プロシージャー

プロシージャー (しばしば、ストアード・プロシージャーと呼ばれる) とは、操作を実行するために呼び出すことができるプログラムのことで、ホスト言語ステートメントおよび SQL ステートメントの両方を含みます。SQL のプロシージャーの場合も、ホスト言語のプロシージャーの場合と同じ利点があります。

DB2 SQL for iSeries のストアード・プロシージャー・サポートは、SQL アプリケーションが SQL ステートメントを使用してプロシージャーを定義し、呼び出すための手段を提供します。ストアード・プロシージャーは、分散型と非分散型の両方の DB2 SQL for iSeries アプリケーションで使用することができます。ストアード・プロシージャーを使用することの大きな利点の 1 つは、分散アプリケーションの場合に、アプリケーション・リクエスター (すなわちクライアント) での CALL ステートメントの 1回の実行で、アプリケーション・サーバー上で多量の作業を実行できることです。

プロシージャーは、SQL プロシージャーまたは外部プロシージャーとして定義することができます。外部プロシージャーとしては、サポートされる任意の高水準言語プログラム (システム/36 のプログラムおよびプロシージャーを除く) または REXX プロシージャーが可能です。このプロシージャーは、SQL ステートメントを含む必要はありませんが、SQL ステートメントを含むことができます。SQL プロシージャーは、全体が SQL で定義され、SQL ステートメント (SQL 制御ステートメントを含む) を含むことができます。

ストアード・プロシージャーをコーディングするには、次のことについて理解している必要があります。

- CREATE PROCEDURE ステートメントによるストアード・プロシージャーの定義
- CALL ステートメントによるストアード・プロシージャーの呼び出し
- パラメーターの引き渡し規則
- プロシージャーを呼び出しているプログラムに完了状況を戻す方法

ストアード・プロシージャーは、CREATE PROCEDURE ステートメントを使用して定義できます。 CREATE PROCEDURE ステートメントは、プロシージャーおよびパラメーター定義をカタログ表 SYSROUTINES および SYSPARMS に追加します。その後、これらの定義は、システムにおける任意の SQL CALL ステートメントによりアクセスできるようになります。

外部プロシージャーまたは SQL プロシージャーを作成するには、SQL CREATE PROCEDURE ステートメントを使用します。または、iSeries ナビゲーターを使用して、プロシージャーを定義することができます。

以下の節では、ストアード・プロシージャーの定義と呼び出しに使用される SQL ステートメント、ストアード・プロシージャーにパラメーターを渡す方法、およびストアード・プロシージャーの使用例について説明します。

• 172 ページの『外部プロシージャーの定義』

- 173 ページの『SQL プロシージャーの定義』
- 179 ページの『ストアード・プロシージャーのデバッグ』
- 180 ページの『ストアード・プロシージャーの呼び出し』
- 185 ページの『ストアード・プロシージャーおよび UDF 用のパラメーターの引き渡し規則』
- 190 ページの『標識変数とストアード・プロシージャー』
- 193 ページの『呼び出しプログラムへの完了状況の戻し』
- 194 ページの『CALL ステートメントの例』

Java でコーディングされたストアード・プロシージャーの説明については、「IBM Developer Kit for Java」の Java SQL ルーチンを参照してください。

DRDA でストアード・プロシージャーを使用することの説明については、383 ページの『DRDA ストアード・プロシージャーに関する考慮事項』を参照してください。

注: コーディング例に関する詳細は、x ページの『コードに関する特記事項』を参照してください。

外部プロシージャーの定義

外部プロシージャーの CREATE PROCEDURE ステートメントでは、次のことを行います。

- プロシージャーに名前を付ける。
- パラメーターとそれらの属性を定義する。
- プロシージャーに関するその他の情報 (システムがプロシージャーを呼び出すときに使用する) を指定する。

次の例を検討してください。

```
CREATE PROCEDURE P1
        (INOUT PARM1 CHAR(10))
        EXTERNAL NAME MYLIB.PROC1
        LANGUAGE C
        GENERAL WITH NULLS;
```

この CREATE PROCEDURE ステートメントは、次のことを行います。

- プロシージャーに P1 という名前を付ける。
- 入力パラメーターと出力パラメーターの両方として使用される 1 つのパラメーターを定義する。このパラメーターは、長さ 10 の文字フィールドです。パラメーターのタイプは、IN、OUT、または INOUT として定義できます。パラメーターのタイプにより、パラメーターの値がプロシージャーとの間で受け渡しされるときが決まります。
- プロシージャーに対応するプログラムの名前 (MYLIB 内の PROC1) を定義する。 MYLIB.PROC1 は、CALL ステートメントでプロシージャーを呼び出したときに呼び出されるプログラムです。
- プロシージャー P1 (プログラム MYLIB.PROC1) が C で作成されていることを示す。言語は、渡すことのできるパラメーターのタイプに影響するため重要です。言語は、パラメーターがプロシージャーに渡される方法にも影響します (た

とえば、ILE C プロシージャーの場合、文字、グラフィック、日付、時刻、およびタイム・スタンプ・パラメーターで NUL 終了文字が渡されます)。

- CALL タイプを GENERAL WITH NULLS として定義する。これは、プロシージャーへのパラメーターが NULL 値を含むことができるため、CALL ステートメントでプロシージャーに追加の引き数を渡したいことを示します。追加の引き数は、N 個の短整数の配列です。ここで、N は CREATE PROCEDURE ステートメントで宣言されたパラメーターの数です。この例では、パラメーターが 1 つしかないため、配列には要素が 1 つしか含まれません。

覚えておくべき重要な点は、プロシージャーを呼び出すためにそれを定義する必要はないということです。ただし、前の CREATE PROCEDURE から、あるいはこのプログラム内の DECLARE PROCEDURE からプロシージャー定義が検出されない場合は、CALL ステートメントでプロシージャーが呼び出されるときに、特定の制限および仮定が行われます。たとえば、NULL 標識引き数を渡すことはできません。対応するプロシージャー定義のない CALL ステートメントの例については、181 ページの『プロシージャー定義が存在しない組み込み CALL ステートメントの使用』を参照してください。

SQL プロシージャーの定義

SQL プロシージャーの CREATE PROCEDURE ステートメントでは、次のことを行います。

- プロシージャーに名前を付ける。
- パラメーターとそれらの属性を定義する。
- プロシージャーに関するその他の情報 (プロシージャーの呼び出し時に使用される) を指定する。
- プロシージャー本体を定義する。 プロシージャー本体は、プロシージャーの実行可能部分であり、単一の SQL ステートメントです。

以下に、入力として社員番号と歩合を受け取り、社員の給与を更新する単純な例を示します。

```
CREATE PROCEDURE UPDATE_SALARY_1
  (IN EMPLOYEE_NUMBER CHAR(10),
   IN RATE DECIMAL(6,2))
   LANGUAGE SQL MODIFIES SQL DATA
   UPDATE CORPDATA.EMPLOYEE
     SET SALARY = SALARY * RATE
     WHERE EMPNO = EMPLOYEE_NUMBER;
```

この CREATE PROCEDURE ステートメントは、次のことを行います。

- プロシージャーに UPDATE_SALARY_1 という名前を付ける。
- パラメーター EMPLOYEE_NUMBER (長さが 6 で、文字データ・タイプの入力パラメーター) および RATE (10 進数データ・タイプの入力パラメーター) を定義する。
- プロシージャーが SQL データを変更する SQL プロシージャーであることを示す。
- プロシージャー本体を単一の UPDATE ステートメントとして定義する。このプロシージャーが呼び出されると、EMPLOYEE_NUMBER および RATE に渡された値を使用して UPDATE ステートメントが実行されます。

SQL 制御ステートメントを使用すると、SQL プロシージャーに単一の UPDATE ステートメントではなく、論理を追加することができます。SQL 制御ステートメントは、次のものから構成されます。

- 割り当てステートメント
- CALL ステートメント
- CASE ステートメント
- 複合ステートメント
- FOR ステートメント
- GET DIAGNOSTICS ステートメント
- GOTO ステートメント
- IF ステートメント
- ITERATE ステートメント
- LEAVE ステートメント
- LOOP ステートメント
- REPEAT ステートメント
- RESIGNAL ステートメント
- RETURN ステートメント
- SIGNAL ステートメント
- WHILE ステートメント

次の例では、入力として社員番号と、最後の評価で受け取られた等級を使用します。このプロシージャーでは、CASE ステートメントを使用して、更新用の適切な増加額と賞与を判別します。

```
CREATE PROCEDURE UPDATE_SALARY_2
  (IN EMPLOYEE_NUMBER CHAR(6),
  IN RATING INT)
  LANGUAGE SQL MODIFIES SQL DATA
     CASE RATING
        WHEN 1 THEN
          UPDATE CORPDATA.EMPLOYEE
            SET SALARY = SALARY * 1.10,
             BONUS = 1000
            WHERE EMPNO = EMPLOYEE_NUMBER;
        WHEN 2 THEN
          UPDATE CORPDATA.EMPLOYEE
            SET SALARY = SALARY * 1.05,
            BONUS = 500
            WHERE EMPNO = EMPLOYEE_NUMBER;
      ELSE
          UPDATE CORPDATA.EMPLOYEE
            SET SALARY = SALARY * 1.03,
            BONUS = 0
            WHERE EMPNO = EMPLOYEE_NUMBER;
     END CASE;
```

この CREATE PROCEDURE ステートメントは、次のことを行います。

- プロシージャーに UPDATE_SALARY_2 という名前を付ける。
- パラメーター EMPLOYEE_NUMBER (長さが 6 で、文字データ・タイプの入力パラメーター) および RATING (整数データ・タイプの入力パラメーター) を定義する。

- プロシージャーが SQL データを変更する SQL プロシージャーであることを示す。
- プロシージャー本体を定義する。 このプロシージャーが呼び出されると、入力パラメーター RATINGがチェックされ、適切な更新ステートメントが実行されます。

複合ステートメントを追加することによって、プロシージャーに複数のステートメントを追加することができます。複合ステートメントでは、任意の数の SQL ステートメントを指定することができます。さらに、SQL 変数、カーソル、およびハンドラーを宣言することができます。

次の例では、入力として部門番号を使用します。このプロシージャーは、その部門内のすべての社員の合計給与と、その部門内の賞与を受ける社員の数を戻します。

```
CREATE PROCEDURERETURN_DEPT_SALARY
         (IN DEPT_NUMBER CHAR(3),
          OUT DEPT_SALARY DECIMAL(15,2),
          OUT DEPT_BONUS_CNT INT)
         LANGUAGE SQL READS SQL DATA
         P1: BEGIN
          DECLARE EMPLOYEE_SALARY DECIMAL(9,2);
          DECLARE EMPLOYEE_BONUS DECIMAL(9,2);
          DECLARE TOTAL_SALARY DECIMAL(15,2)DEFAULT 0
          DECLARE BONUS_CNT INT DEFAULT 0;
          DECLARE END_TABLE INT DEFAULT 0;
          DECLARE C1 CURSOR FOR
           SELECT SALARY, BONUS FROM CORPDATA.EMPLOYEE
            WHERE WORKDEPT = DEPT_NUMBER;
          DECLARE CONTINUE HANDLER FOR NOT FOUND
            SET END_TABLE = 1;
          DECLARE EXIT HANDLER FOR SQLEXCEPTION
            SET DEPT_SALARY = NULL;
          OPEN C1;
          FETCH C1 INTO EMPLOYEE_SALARY, EMPLOYEE_BONUS;
          WHILE END_TABLE = 0 DO
             SET TOTAL_SALARY = TOTAL_SALARY + EMPLOYEE_SALARY + EMPLOYEE_BONUS;
             IF EMPLOYEE_BONUS > 0 THEN
               SET BONUS_CNT = BONUS_CNT + 1;
             END IF;
             FETCH C1 INTO EMPLOYEE_SALARY, EMPLOYEE_BONUS;
           END WHILE;
           CLOSE C1;
           SET DEPT_SALARY = TOTAL_SALARY;
           SET DEPT_BONUS_CNT = BONUS_CNT;
         END P1;
```

この CREATE PROCEDURE ステートメントは、次のことを行います。

- プロシージャーに RETURN_DEPT_SALARY という名前を付ける。
- パラメーター DEPT_NUMBER (長さが 3 で、文字データ・タイプの入力パラメーター)、DEPT_SALARY (10 進数データ・タイプの出力パラメーター)、および DEPT_BONUS_CNT (整数データ・タイプの出力パラメーター) を定義する。
- プロシージャーが SQL データを読み取る SQL プロシージャーであることを示す。
- プロシージャー本体を定義する。
  - SQL 変数 EMPLOYEE_SALARY および TOTAL_SALARY を 10 進数フィールドとして宣言する。

- SQL 変数 BONUS_CNT および END_TABLE を整数として宣言し、0 に初期設定する。
- 社員表から列を選択するカーソル C1 を宣言する。
- 呼び出されたときに変数 END_TABLE を 1 に設定する、NOT FOUND 用の継続ハンドラーを宣言する。このハンドラーは、FETCH でこれ以上戻す行がないときに呼び出されます。このハンドラーが呼び出されると、SQLCODE および SQLSTATE が 0 に再初期設定されます。
- SQLEXCEPTION 用の終了ハンドラーを宣言する。このハンドラーが呼び出されると、DEPT_SALARY が NULL に設定され、複合ステートメントの処理が終了されます。このハンドラーは、エラーが発生した場合、すなわち、SQLSTATE クラスが '00'、'01'、または '02' でない場合に呼び出されます。SQL プロシージャーには必ず標識が渡されるため、DEPT_SALARY の標識値はプロシージャーの戻り時に -1 になります。このハンドラーが呼び出されると、SQLCODE および SQLSTATE が 0 に再初期設定されます。

  SQLEXCEPTION 用のハンドラーが指定されていない場合に、別のハンドラーで処理されないエラーが発生すると、複合ステートメントの実行が終了され、SQLCA でエラーが戻されます。標識と同様に、SQLCA は SQL プロシージャーから必ず戻されます。
- カーソル C1 の OPEN、FETCH、および CLOSE を組み込む。カーソルの CLOSE が指定されない場合、カーソルは複合ステートメントの終了時にクローズされます。これは、CREATE PROCEDURE ステートメントで SET RESULT SETS が指定されていないためです。
- 最後のレコードが取り出されるまでループする WHILE ステートメントを組み込む。取り出されたそれぞれの行について、TOTAL_SALARY が増加され、さらに、社員の賞与が 0 より大きい場合は、BONUS_CNT が増加されます。
- 出力パラメーターとして DEPT_SALARY および DEPT_BONUS_CNT を戻す。

複合ステートメントをアトミックにすると、予期しないエラーが生じた場合に、アトミック・ステートメント内のステートメントがロールバックされるようにすることができます。アトミック複合ステートメントは、SAVEPOINTS を使用してインプリメントされます。その複合ステートメントが成功すると、トランザクションがコミットされます。 SAVEPOINTS の使用の詳細については、344 ページの『保管ポイント』を参照してください。

次の例では、入力として部門番号を使用します。EMPLOYEE_BONUS 表が存在することを確認し、部門内の賞与を受けるすべての社員の名前を挿入します。このプロシージャーでは、賞与を受ける社員の合計数が戻されます。

```
CREATE PROCEDURE CREATE_BONUS_TABLE
        (IN DEPT_NUMBER CHAR(3),
         INOUT CNT INT)
         LANGUAGE SQL MODIFIES SQL DATA
          CS1: BEGIN ATOMIC
          DECLARE NAME VARCHAR(30) DEFAULT NULL;
          DECLARE CONTINUE HANDLER FOR SQLSTATE '42710'
             SELECT COUNT(*) INTO CNT
             FROM DATALIB.EMPLOYEE_BONUS;
          DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
             SET  CNT = CNT - 1;
          DECLARE UNDO HANDLER FOR SQLEXCEPTION
```

```
   SET CNT = NULL;
IF DEPT_NUMBER IS NOT NULL THEN
  CREATE TABLE DATALIB.EMPLOYEE_BONUS
       (FULLNAME VARCHAR(30),
        BONUS DECIMAL(10,2),
       PRIMARY KEY (FULLNAME));
FOR_1:FOR V1 AS C1 CURSOR FOR
    SELECT FIRSTNME, MIDINIT, LASTNAME, BONUS
       FROM CORPDATA.EMPLOYEE
       WHERE WORKDEPT = CREATE_BONUS_TABLE.DEPT_NUMBER
    DO
    IF BONUS > 0 THEN
      SET NAME = FIRSTNME CONCAT ' ' CONCAT
                 MIDINIT  CONCAT ' 'CONCAT LASTNAME;
      INSERT INTO DATALIB.EMPLOYEE_BONUS
         VALUES(CS1.NAME, FOR_1.BONUS);
      SET CNT = CNT + 1;
    END IF;
  END FOR FOR_1;
END IF;
END CS1
```

この CREATE PROCEDURE ステートメントは、次のことを行います。

- プロシージャーに CREATE_BONUS_TABLE という名前を付ける。
- パラメーター DEPT_NUMBER (長さが 3 で、文字データ・タイプの入力パラメーター) および CNT (整数データ・タイプの入出力パラメーター) を定義する。
- プロシージャーが SQL データを変更する SQL プロシージャーであることを示す。
- プロシージャー本体を定義する。
  - SQL 変数 NAME を可変長文字として宣言する。
  - SQLSTATE 42710 (表がすでに存在する場合) 用の継続ハンドラーを宣言する。EMPLOYEE_BONUS 表がすでに存在する場合は、このハンドラーが呼び出され、表内のレコードの数を取り出します。 SQLCODE および SQLSTATE が 0 にリセットされ、FOR ステートメントから処理が継続されます。
  - SQLSTATE 23505 (重複キー) 用の継続ハンドラーを宣言する。プロシージャーにより表内にすでに存在する名前の挿入が試みられると、このハンドラーが呼び出され、CNT を減らします。INSERT ステートメントの後で、SET ステートメントから処理が継続されます。
  - SQLEXCEPTION 用の UNDO ハンドラーを宣言する。このハンドラーが呼び出されると、これまでのステートメントがロールバックされ、CNT が 0 に設定され、複合ステートメントの後から処理が継続されます。この場合は、複合ステートメントに続くステートメントがないため、プロシージャーが戻ります。
  - FOR ステートメントを使用して、EMPLOYEE 表からレコードを読み取るためのカーソル C1 を宣言する。FOR ステートメントの中では、選択リストからの列名が、取り出された行からのデータが入る SQL 変数として使用されています。それぞれの行について、列 FIRSTNME、MIDINIT、および LASTNAME からのデータが間にブランクを 1 つ入れて連結され、結果が SQL 変数 NAME に入れられます。SQL 変数 NAME および BONUS は、EMPLOYEE_BONUS 表に挿入されます。選択リスト項目のデータ・タイプが

プロシージャーの作成時に判明していなければならないため、FOR ステートメントで指定される表は、プロシージャーの作成時に存在していなければなりません。

SQL 変数名は、それが定義される FOR ステートメントまたは複合ステートメントのラベル名で修飾することができます。 たとえば、FOR_1.BONUS は、選択されたそれぞれの行の列 BONUS の値が入る SQL 変数を参照しています。 CS1.NAME は、開始ラベル CS1 の複合ステートメントで定義されている変数 NAME です。また、パラメーター名をプロシージャー名で修飾することもできます。CREATE_BONUS_TABLE.DEPT_NUMBER は、プロシージャー CREATE_BONUS_TABLE の DEPT_NUMBER パラメーターです。列名も認められる SQL ステートメントで非修飾 SQL 変数名が使用され、変数名が列名と同じである場合、その名前は列を参照するために使用されます。

SQL プロシージャーで動的 SQL を使用することもできます。次の例では、特定の部門内の全社員を含む表を作成します。部門番号が入力データとしてプロシージャーに渡され、表名に連結されます。

```
CREATE PROCEDURE CREATE_DEPT_TABLE (IN P_DEPT CHAR(3))
        LANGUAGE SQL
   BEGIN
     DECLARE STMT CHAR(1000);
     DECLARE MESSAGE CHAR(20);
     DECLARE TABLE_NAME CHAR(30);
     DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
        SET MESSAGE = 'ok';
     SET TABLE_NAME = 'CORPDATA.DEPT_' CONCAT P_DEPT CONCAT '_T';
     SET STMT = 'DROP TABLE ' CONCAT TABLE_NAME;
     PREPARE S1 FROM STMT;
     EXECUTE S1;
        SET STMT = 'CREATE TABLE ' CONCAT TABLE_NAME CONCAT
       '( EMPNO CHAR(6) NOT NULL,
          FIRSTNME VARCHAR(12) NOT NULL,
          MIDINIT CHAR(1) NOT NULL,
          LASTNAME CHAR(15) NOT NULL,
          SALARY DECIMAL(9,2))';
     PREPARE S2 FROM STMT;
     EXECUTE S2;
     SET STMT = 'INSERT INTO ' CONCAT TABLE_NAME CONCAT
       'SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY
          FROM CORPDATA.EMPLOYEE
          WHERE  WORKDEPT = ?';
     PREPARE S3 FROM STMT;
     EXECUTE S3 USING P_DEPT;
END;
```

この CREATE PROCEDURE ステートメントは、次のことを行います。

- プロシージャーに CREATE_DEPT_TABLE という名前を付ける。
- パラメーター P_DEPT (長さが 3 で、文字データ・タイプの入力パラメーター) を定義する。
- プロシージャーが SQL プロシージャーであることを示す。
- プロシージャー本体を定義する。
  - SQL 変数 STMT および SQL 変数 TABLE_NAME を文字として宣言する。
  - CONTINUE ハンドラーを宣言する。このプロシージャーは、表がすでに存在している場合、表の DROP を試行します。表が存在していない場合、最初の EXECUTE は失敗します。このハンドラーにより、処理は続行します。

– 変数 TABLE_NAME を 'DEPT_' に設定し、その後にパラメーター P_DEPT で渡された文字と '_T' を続ける。
– DROP ステートメントに変数 STMT を設定し、そのステートメントを準備して実行する。
– CREATE ステートメントに変数 STMT を設定し、そのステートメントを準備して実行する。
– INSERT ステートメントに変数 STMT を設定し、そのステートメントを準備して実行する。WHERE 文節にパラメーター・マーカーが指定されます。ステートメントを実行すると、変数 P_DEPT が USING 文節に渡されます。

プロシージャーが呼び出されて部門の値 'D21' が渡されると、表 DEPT_D21_T が作成され、部門 'D21' のすべての社員によって初期設定されます。

ストアード・プロシージャーのデバッグ

SQL プロシージャー作成ステートメント、SQL 関数作成ステートメント、またはトリガー作成ステートメントにおいて、SET OPTION DBGVIEW = *SOURCE を指定することにより、生成されたプログラムまたはモジュールを SQL ステートメント・レベルでデバッグすることができます。 RUNSQLSTM コマンドのパラメーターとして DBGVIEW(*SOURCE) を指定することもできます。
DBGVIEW(*SOURCE) は、RUNSQLSTM の中のすべてのルーチンに適用されます。

ソース視点は、システムが、ユーザーのオリジナルのルーチン本体を元にして QTEMP/QSQDSRC に作成します。ソース視点は、プログラムまたはサービス・プログラムと一緒に保管はされません。ソース視点は、ユーザーがデバッグで停止できる場所に対応する行へと分割されます。テキスト (パラメーター名および変数名を含む) は、大文字に変換されます。

すべての変数およびパラメーターは、構造の一部として生成されます。デバッグで変数を評価するときは、構造名を使用する必要があります。変数は、現行ラベル名で修飾されます。パラメーターは、プロシージャーまたは関数で修飾されます。トリガー内の遷移変数は、該当する相関名で修飾されます。それぞれの複合ステートメントまたは FOR ステートメントごとにラベル名を指定することを、強くお勧めします。ラベル名を指定しなければ、システムが代わってそれを生成します。 こうなると、変数を評価することはほとんど不可能になります。すべての変数およびパラメーターは大文字の名前として評価されなければならないことを、記憶してください。構造の名前を評価することもできます。こうすると、構造内のすべての変数が示されます。変数またはパラメーターがヌル可能である場合、構造内のその変数またはパラメーターの直後に対応する標識が続きます。

SQL ルーチンは C で生成されるため、C における制約事項の一部が SQL ソース・デバッグにも影響を及ぼします。 SQL ルーチン本体で指定された、区切り文字で区切った名前は、C では指定できません。これらの名前に対して名前が生成され、デバッグや評価を行うことがさらに難しくなります。任意の文字変数の内容を評価するには、変数の名前の前に * を指定します。

システムは、変数名およびパラメーター名のほとんどに対して標識を生成するため、ある変数が SQL ヌル値であるかどうかを知るのに直接の検査方法はありません。変数を評価すると、たとえ標識はヌル値を示すようにセットされている場合でも、必ずある値が表示されます。

ハンドラーが呼び出されているかを判別するために、ハンドラーの中の最初のステートメントにブレークポイントを設定します。ハンドラーの中の複合ステートメントまたは FOR ステートメントで宣言されている変数は、評価することができます。

ストアード・プロシージャーの呼び出し

SQL CALL ステートメントは、ストアード・プロシージャーを呼び出します。CALL ステートメントでは、ストアード・プロシージャーの名前および任意の引き数を指定します。引き数としては、定数、特殊レジスター、またはホスト変数が可能です。 CALL ステートメントで指定する外部ストアード・プロシージャーには、対応する CREATE PROCEDURE ステートメントがなくても構いません。 SQL プロシージャーによって作成されたプログラムは、CREATE PROCEDURE ステートメントで指定されたプロシージャー名を呼び出すことによってのみ、呼び出すことができます。

プロシージャーはシステム・プログラム・オブジェクトですが、CALL CL コマンドを使用してもプロシージャーを呼び出すことはできません。 CALL CL コマンドは、プロシージャー定義を使用して入出力パラメーターをマップすることも、プロシージャーのパラメーター・スタイルを使用してプログラムにパラメーターを渡すこともしません。

CALL ステートメントには 3 つのタイプがあり、DB2 SQL for iSeries ではタイプごとにルールが異なるので、以下に、それぞれのタイプについて説明します。それらは次のとおりです。

- プロシージャー定義が存在する組み込みまたは動的 CALL ステートメント
- プロシージャー定義が存在しない組み込み CALL ステートメント
- CREATE PROCEDURE が存在しない動的 CALL ステートメント

注: この場合の動的とは、次のものを指しています。

- 動的に準備され実行される CALL ステートメント
- 対話式環境で発行される CALL ステートメント (たとえば、STRSQL または Query Manager を介して)
- EXECUTE IMMEDIATE ステートメントで実行される CALL ステートメント

以下に、それぞれのタイプについて説明します。

- 181 ページの『プロシージャー定義が存在する CALL ステートメントの使用』
- 181 ページの『プロシージャー定義が存在しない組み込み CALL ステートメントの使用』
- 182 ページの『SQLDA を伴う組み込み CALL ステートメントの使用』

- 183 ページの『CREATE PROCEDURE が存在しない動的 CALL ステートメントの使用』

プロシージャー定義が存在する CALL ステートメントの使用

このタイプの CALL ステートメントは、プロシージャーおよび引き数属性に関するすべての情報を CREATE PROCEDURE カタログ定義から読み取ります。次の PL/I の例では、示されている CREATE PROCEDURE ステートメントに対応する CALL ステートメントが示されています。

```
DCL HV1 CHAR(10);
DCL IND1 FIXED BIN(15);
   :
EXEC SQL CREATE P1 PROCEDURE
          (INOUT PARM1 CHAR(10))
          EXTERNAL NAME MYLIB.PROC1
          LANGUAGE C
          GENERAL WITH NULLS;
   :
EXEC SQL CALL P1 (:HV1 :IND1);
   :
```

この CALL ステートメントを呼び出すと、プログラム MYLIB/PROC1 への呼び出しが行われ、2 つの引き数が渡されます。プログラムの言語が ILE C であるため、最初の引き数は、長さ 11 文字の C NUL 終了ストリングで、ホスト変数 HV1 の内容が含まれます。ILE C プロシージャーへの呼び出し時に、DB2 SQL for iSeries は、パラメーターが文字、グラフィック、日付、時刻、または時刻スタンプ変数として宣言されている場合は、パラメーター宣言に 1 文字を追加することに注意してください。2 番目の引き数は標識配列です。この例では、CREATE PROCEDURE ステートメントにパラメーターが 1 つしかないため、これは 1 つの短整数になります。この引き数には、プロシージャーに入るときの標識変数 IND1 の内容が含まれます。

最初のパラメーターが INOUT として宣言されているため、SQL は、ユーザー・プログラムに戻る前に、ホスト変数 HV1 と標識変数 IND1 を MYLIB.PROC1 から戻された値で更新します。

注:

1. CREATE PROCEDURE ステートメントと CALL ステートメントで指定されたプロシージャー名が正確に一致していなければ、プログラムの SQL 事前コンパイル時にこれらの間のリンクは行われません。
2. CREATE PROCEDURE と DECLARE PROCEDURE ステートメントの両方が存在する組み込み CALL ステートメントの場合には、DECLARE PROCEDURE ステートメントが使用されます。

プロシージャー定義が存在しない組み込み CALL ステートメントの使用

対応する CREATE PROCEDURE ステートメントが存在しない静的 CALL ステートメントは、次の規則を用いて処理されます。

- すべてのホスト変数引き数は、INOUT タイプ・パラメーターとして扱われます。
- CALL タイプは GENERAL です (標識引き数は渡されません)。

- 呼び出すプログラムは、CALL で指定されたプロシージャー名と、命名規則 (必要であれば) に基づいて判別されます。
- 呼び出すプログラムの言語は、プログラムに関してシステムから取り出した情報に基づいて判別されます。

例: プロシージャー定義が存在しない組み込み CALL ステートメント

注: コーディング例に関する詳細は、x ページの『コードに関する特記事項』を参照してください。

以下に、プロシージャー定義が存在しない組み込み CALL ステートメント (PL/I) の例を示します。

```
DCL HV2 CHAR(10);
   :
EXEC SQL CALL P2 (:HV2);
   :
```

この CALL ステートメントが呼び出されると、DB2 SQL for iSeries は、標準の SQL 命名規則に基づいてプログラムを検出しようとします。上の例では、*SYS (システム命名) の命名オプションが使用されることと、CRTSQLPLI コマンドで DFTRDBCOL パラメーターが指定されていないことが想定されています。この場合、P2 という名前のプログラム名を見つけるためにライブラリー・リストが探索されます。呼び出しタイプが GENERAL であるため、標識変数用の追加の引き数はプログラムに渡されません。

注: CALL ステートメントで標識変数が指定され、CALL ステートメントの実行時にその値がゼロより小さい場合は、標識をプロシージャーに渡す方法がないためにエラーが起こります。

プログラム P2 がライブラリー・リスト内で検出されるとすれば、CALL 時にホスト変数 HV2 の内容がプログラムに渡され、P2 の実行完了後に、P2 から戻された引き数がホスト変数に再びマップされます。

SQLDA を伴う組み込み CALL ステートメントの使用

どちらのタイプの組み込み CALL でも (プロシージャー定義が存在する場合でも、存在しない場合でも)、次の C の例で示されているように、パラメーター・リストの代わりに SQLDA を渡すことができます。この例では、ストアード・プロシージャーが 2 つのパラメーター (1 つ目はタイプ SHORT INT で、2 つ目は長さ 4 のタイプ CHAR) を予期していると想定しています。

```
#define SQLDA_HV_ENTRIES 2
#define SHORTINT 500
#define NUL_TERM_CHAR 460

exec sql include sqlca;
exec sql include sqlda;
...
typedef struct sqlda Sqlda;
typedef struct sqlda* Sqldap;
...
main()
{
 Sqldap dap;
 short col1;
```

```
 char col2[4];
 int bc;
 dap = (Sqldap) malloc(bc=SQLDASIZE(SQLDA_HV_ENTRIES));
        /* SQLDASIZE is a macro defined in the sqlda include */
 col1 = 431;
 strcpy(col2,"abc");
 strncpy(dap->sqldaid,"SQLDA   ",8);
 dap->sqldabc = bc;         /* bc set in the malloc statement above */
 dap->sqln = SQLDA_HV_ENTRIES;
 dap->sqld = SQLDA_HV_ENTRIES;
 dap->sqlvar[0].sqltype = SHORTINT;
 dap->sqlvar[0].sqllen  = 2;
 dap->sqlvar[0].sqldata = (char*) &col1;
 dap->sqlvar[0].sqlname.length = 0;
 dap->sqlvar[1].sqltype = NUL_TERM_CHAR;
 dap->sqlvar[1].sqllen  = 4;
 dap->sqlvar[1].sqldata = col2;
 ...
 EXEC SQL CALL P1 USING DESCRIPTOR :*dap;
 ...
}
```

呼び出されるプロシージャーの名前をホスト変数に格納し、CALL ステートメントで、そのホスト変数を、ハードコーディングされたプロシージャー名の代わりに使用することもできます。 たとえば、次の通りです。

```
...
main()
{
 char proc_name[15];
 ...
 strcpy (proc_name, "MYLIB.P3");
 ...
 EXEC SQL CALL :proc_name ...;
 ...
}
```

上の例で、MYLIB.P3 がパラメーターを予期する場合は、パラメーター・リスト、または USING DESCRIPTOR 文節で渡される SQLDA (前の例で示されている) を使用することができます。

CALL ステートメントでプロシージャー名が入っているホスト変数が使用され、CREATE PROCEDURE カタログ定義が存在する場合は、それが使用されます。プロシージャー名は、パラメーター・マーカーとして指定することはできません。

ストアード・プロシージャーの呼び出しの例が、さらに、この章の後半にあります。

CREATE PROCEDURE が存在しない動的 CALL ステートメントの使用

次の規則は、CREATE PROCEDURE 定義が存在しない場合の動的 CALL ステートメントの処理に関係します。

- すべての引き数は、IN タイプ・パラメーターとして扱われます。
- CALL タイプは GENERAL です (標識引き数は渡されません)。
- 呼び出すプログラムは、CALL で指定されたプロシージャー名と命名規則に基づいて判別されます。

- 呼び出すプログラムの言語は、プログラムに関してシステムから取り出した情報に基づいて判別されます。

例: CREATE PROCEDURE が存在しない動的 CALL ステートメント

注: コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

以下に、C の動的 CALL ステートメントの例を示します。

```
char hv3[10],string[100];
    :
strcpy(string,"CALL MYLIB.P3 ('P3 TEST')");
EXEC SQL EXECUTE IMMEDIATE :string;
    :
```

この例は、EXECUTE IMMEDIATE ステートメントを介して実行される動的 CALL ステートメントを示しています。'P3 TEST' を含む文字変数として渡された 1 つのパラメーターを使用して、プログラム MYLIB.P3 への呼び出しが行われます。

前の例にあるように、CALL ステートメントを実行して定数を渡すときには、プログラム内の予測される引き数の長さを念頭に置いておく必要があります。プログラム MYLIB.P3 で 5 文字の引き数しか予期していないと、例で指定されている定数の最後の 2 文字がプログラムから失われることになります。

注: こうした理由により、CALL ステートメントではホスト変数を使用する方が安全です。そうすれば、プロシージャーの属性を完全に一致させることができ、しかも文字が失われることもなくなります。動的 SQL の場合、CALL ステートメントの引き数としてホスト変数を指定できますが、それを処理するために PREPARE および EXECUTE ステートメントを使用しなければなりません。

CALL ステートメントに渡される数値定数には、次の規則が適用されます。

- すべての整数定数がフルワード 2 進整数として渡されます。
- すべての 10 進定数がパック 10 進値として渡されます。精度および位取りは定数値に基づいて判別されます。たとえば、値 123.45 は、パック 10 進数 (5,2) として渡されます。同様に、値 001.01 も、精度 5、位取り 2 として渡されます。
- すべての浮動小数点定数は、倍精度浮動小数点数として渡されます。

動的 CALL ステートメントで指定された特殊レジスターは、次のように渡されます。

CURRENT DATE
ISO 形式の 10 バイトの文字ストリングとして渡されます。

CURRENT TIME
ISO 形式の 8 バイトの文字ストリングとして渡されます。

CURRENT TIMESTAMP
IBM SQL 形式の 26 バイトの文字ストリングとして渡されます。

CURRENT TIMEZONE
精度 6、位取り 0 のパック 10 進数として渡されます。

CURRENT SCHEMA
128 バイトの可変長文字ストリングとして渡されます。

CURRENT SERVER
18 バイトの可変長文字ストリングとして渡されます。

USER
18 バイトの可変長文字ストリングとして渡されます。

CURRENT PATH
3483 バイトの可変長文字ストリングとして渡されます。

ストアード・プロシージャーおよび UDF 用のパラメーターの引き渡し規則

CALL ステートメントでは、サポートされるすべてのホスト言語で書かれているプログラムおよび REXX プロシージャーに引き数を渡すことができます。各言語は、それに合わせて調整されたさまざまなデータ・タイプをサポートします。 データ・タイプは、次の表の左端の列に入っています。その行の他の欄には、そのデータ・タイプが特定の言語のパラメーター・タイプとしてサポートされるかどうかを示す標識が入っています。欄にダッシュ (-) が入っている場合、そのデータ・タイプはその言語のパラメーター・タイプとしてサポートされません。ホスト変数宣言は、DB2 SQL for iSeries が、このデータ・タイプをこの言語のパラメーターとしてサポートすることを示します。この宣言では、ホスト変数がプロシージャーによって正しく受け取られ、設定されるために必要な宣言を行います。 SQL プロシージャーの呼び出し時には、すべての SQL データ・タイプがサポートされるため、表の中には欄が設けられていません。

詳細は、ホスト言語での SQL プログラミング および「IBM Developer Kit for Java」の Java SQL ルーチン のセクションを参照してください。

表 25. パラメーターのデータ・タイプ

SQL データ・タイプ	C および C++	CL	COBOL for iSeries および ILE COBOL for iSeries
SMALLINT	short	-	PIC S9(4) BINARY
INTEGER	long	-	PIC S9(9) BINARY
BIGINT	long long	-	PIC S9(18) BINARY **注:** ILE COBOL for iSeries でのみサポートされます。
DECIMAL (p,s)	decimal(p,s)	TYPE(*DEC) LEN(p s)	PIC S9(p-s)V9(s) PACKED-DECIMAL **注:** 精度が 18 を超えてはなりません。
NUMERIC(p,s)	-	-	PIC S9(p-s)V9(s) DISPLAY SIGN LEADING SEPARATE **注:** 精度が 18 を超えてはなりません。
REAL または FLOAT(p)	float	-	COMP-1 **注:** ILE COBOL for iSeries でのみサポートされます。

表 25. パラメーターのデータ・タイプ (続き)

SQL データ・タイプ	C および C++	CL	COBOL for iSeries および ILE COBOL for iSeries
DOUBLE PRECISION または FLOAT または FLOAT(p)	double	-	COMP-2 **注:** ILE COBOL for iSeries でのみサポートされます。
CHARACTER(n)	char ... [n+1]	TYPE(*CHAR) LEN(n)	PIC X(n)
VARCHAR(n)	char ... [n+1]	-	可変長文字ストリング (ホスト言語での SQL プログラミング の「COBOL」の章参照)。
VARCHAR(n) FOR BIT DATA	VARCHAR 構造化形式 (ホスト言語での SQL プログラミング の「C」の章参照)	-	可変長文字ストリング (ホスト言語での SQL プログラミング の「COBOL」の章参照)。
GRAPHIC(n)	wchar_t ... [n+1]	-	PIC G(n) DISPLAY-1 または PIC N(n) **注:** ILE COBOL for iSeries でのみサポートされます。
VARGRAPHIC(n)	VARGRAPHIC 構造化形式 (C の章を参照)	-	可変長グラフィック・ストリング (ホスト言語での SQL プログラミング の「COBOL」の章参照)。 **注:** ILE COBOL for iSeries でのみサポートされます。
DATE	char ... [11]	TYPE(*CHAR) LEN(10)	PIC X(10) ILE COBOL for iSeries の場合のみ、FORMAT DATE。
TIME	char ... [9]	TYPE(*CHAR) LEN(8)	PIC X(8) ILE COBOL for iSeries の場合のみ、FORMAT TIME。
TIMESTAMP	char ... [27]	TYPE(*CHAR) LEN(26)	PIC X(26) ILE COBOL for iSeries の場合のみ、FORMAT TIMESTAMP。
CLOB	CLOB 構造化形式 (ホスト言語での SQL プログラミング の「C」の章参照)	-	CLOB 構造化形式 (ホスト言語での SQL プログラミング の「COBOL」の章参照)。 **注:** ILE COBOL for iSeries でのみサポートされます。

表 25. パラメーターのデータ・タイプ (続き)

SQL データ・タイプ	C および C++	CL	COBOL for iSeries および ILE COBOL for iSeries
BLOB	BLOB 構造化形式 (ホスト言語での SQL プログラミング の「C」の章参照)	-	BLOB 構造化形式 (ホスト言語での SQL プログラミング の「COBOL」の章参照)。**注:** ILE COBOL for iSeries でのみサポートされます。
DBCLOB	DBCLOB 構造化形式 (ホスト言語での SQL プログラミング の「C」の章参照)	-	DBCLOB 構造化形式 (ホスト言語での SQL プログラミング の「COBOL」の章参照)。**注:** ILE COBOL for iSeries でのみサポートされます。
ROWID	ROWID 構造化形式 (ホスト言語での SQL プログラミング の「C」の章参照)	-	ROWID 構造化形式 (ホスト言語での SQL プログラミング の「COBOL」の章参照)。
データ・リンク	-	-	-
標識変数	short	-	PIC S9(4) BINARY

表 26. パラメーターのデータ・タイプ

SQL データ・タイプ	FORTRAN	Java パラメーター・スタイル JAVA	Java パラメーター・スタイル DB2GENERAL	PL/I
SMALLINT	INTEGER*2	short	short	FIXED BIN(15)
INTEGER	INTEGER*4	int	int	FIXED BIN(31)
BIGINT	-	long	long	-
DECIMAL (p,s)	-	BigDecimal	BigDecimal	FIXED DEC(p,s)
NUMERIC(p,s)	-	BigDecimal	BigDecimal	-
REAL または FLOAT(p)	REAL*4	float	float	FLOAT BIN(p)
DOUBLE PRECISION または FLOAT または FLOAT(p)	REAL*8	double	double	FLOAT BIN(p)
CHARACTER(n)	CHARACTER*n	文字列	文字列	CHAR(n)
VARCHAR(n)	-	文字列	文字列	CHAR(n) VAR
VARCHAR(n) FOR BIT DATA	-	-	com.ibm.db2.app.Blob	CHAR(n) VAR
GRAPHIC(n)	-	文字列	文字列	-
VARGRAPHIC(n)	-	文字列	文字列	-
DATE	CHARACTER*10	日付	文字列	CHAR(10)
TIME	CHARACTER*8	時間	文字列	CHAR(8)
TIMESTAMP	CHARACTER*26	タイム・スタンプ	文字列	CHAR(26)

表 26. パラメーターのデータ・タイプ (続き)

SQL データ・タイプ	FORTRAN	Java パラメーター・スタイル JAVA	Java パラメーター・スタイル DB2GENERAL	PL/I
CLOB	-	-	com.ibm.db2.app.Clob	CLOB 構造化形式 (ホスト言語での SQL プログラミング の「PL/I」の章参照)
BLOB	-	-	com.ibm.db2.app.Blob	BLOB 構造化形式 (ホスト言語での SQL プログラミング の「PL/I」の章参照)
DBCLOB	-	-	com.ibm.db2.app.Clob	DBCLOB 構造化形式 (ホスト言語での SQL プログラミング の「PL/I」の章参照)
ROWID	-	-	-	ROWID 構造化形式 (ホスト言語での SQL プログラミング の「PL/I」の章参照)
データ・リンク	-	-	-	-
標識変数	INTEGER*2	-	-	FIXED BIN(15)

表 27. パラメーターのデータ・タイプ

SQL データ・タイプ	REXX	RPG	ILE RPG
SMALLINT	-	1 つのサブフィールドを含むデータ構造。サブフィールド指定の 43 桁目は ***B***、長さは 2、52 桁目は 0 とする。	データ指定。サブフィールド指定の 40 桁目は ***B***、長さ <= 4、41 ～ 42 桁目は 00 とする。 あるいは データ指定。サブフィールド指定の 40 桁目は ***I***、長さは 5 で、41 ～ 42 桁目は 00とする。
INTEGER	小数部 (およびオプションの先行符号) を伴わない数値ストリング	1 つのサブフィールドを含むデータ構造。サブフィールド指定の 43 桁目は ***B***、長さは 4、52 桁目は 0 とする。	データ指定。サブフィールド指定の 40 桁目は ***B***、長さ <=09 かつ >=05、41 ～ 42 桁目は 00 とする。 あるいは データ指定。サブフィールド指定の 40 桁目は ***I***、長さは 10 で、41 ～ 42 桁目は 00 とする。
BIGINT	-	-	データ指定。サブフィールド指定の 40 桁目は ***I***、長さは 20 で、41 ～ 42 桁目は 00 とする。

表 27. パラメーターのデータ・タイプ (続き)

SQL データ・タイプ	REXX	RPG	ILE RPG
DECIMAL (p,s)	小数部 (およびオプションの先行符号) を伴う数値ストリング	1 つのサブフィールドを含むデータ構造。サブフィールド指定の 43 桁目は***P***、52 桁目は 0 ～ 9 とする。あるいは、数値入力フィールドまたは計算結果フィールド。	データ指定。サブフィールド指定の 40 桁目は ***P***、40 ～ 42 桁目は 00 ～ 31 とする。
NUMERIC(p,s)	-	1 つのサブフィールドを含むデータ構造。サブフィールド指定の 43 桁目は ***ブランク***、52 桁目は 0 ～ 9 とする。	データ指定。サブフィールド指定の 40 桁目は ***S*** とする。あるいは、40 桁目を***ブランク***、41 ～ 42 桁目を 00 ～ 31 とする。
REAL または FLOAT(p)	数字の次に E が入り、(次に任意指定の先行符号)、次に数字が入ったストリング	-	データ指定。40 桁目は ***F***、長さは 4 とする。
DOUBLE PRECISION または FLOAT または FLOAT(p)	数字の次に E が入り、(次にオプションの先行符号)、次に数字が入ったストリング	-	データ指定。40 桁目は ***F***、長さは 8 とする。
CHARACTER(n)	2 つのアポストロフィの間に n 個の文字が入ったストリング	サブフィールドを含まないデータ構造または 1 つのサブフィールドを含むデータ構造。サブフィールド指定の 43 桁目と 52 桁目は ***ブランク*** とする。あるいは、文字入力フィールドまたは計算結果フィールド。	データ指定。サブフィールド指定の 40 桁目は ***A*** とする。あるいは、40 桁目と 41 ～ 42 桁目を ***ブランク*** とする。
VARCHAR(n)	2 つのアポストロフィの間に n 個の文字が入ったストリング	-	データ指定。サブフィールド指定の 40 桁目は ***A*** とする。あるいは、40 桁目と 41 ～ 42 桁目を ***ブランク*** とし、44 ～ 80 桁目にキーワード VARYING を置く。
VARCHAR(n) FOR BIT DATA	2 つのアポストロフィの間に n 個の文字が入ったストリング	-	データ指定。サブフィールド指定の 40 桁目は ***A*** とする。あるいは、40 桁目と 41 ～ 42 桁目を ***ブランク*** とし、44 ～ 80 桁目にキーワード VARYING を置く。
GRAPHIC(n)	G' で始まり、n 個の 2 バイト文字の後に ' が入ったストリング	-	データ指定。サブフィールド指定の 40 桁目は ***G*** とする。
VARGRAPHIC(n)	G' で始まり、n 個の 2 バイト文字の後に ' が入ったストリング	-	データ指定。サブフィールド指定の 40 桁目は ***G*** とし、44 ～ 80 桁目にキーワード VARYING を置く。

表 27. パラメーターのデータ・タイプ (続き)

SQL データ・タイプ	REXX	RPG	ILE RPG
DATE	2 つのアポストロフィの間に 10 文字が入ったストリング	サブフィールドを含まないデータ構造または 1 つのサブフィールドを含むデータ構造。サブフィールド指定の 43 桁目と 52 桁目は **ブランク** とする。長さは 10 とする。あるいは、文字入力フィールドまたは計算結果フィールド。	データ指定。サブフィールド指定の 40 桁目は ***D*** とする。44 ～ 80 桁目は DATFMT(*ISO) とする。
TIME	2 つのアポストロフィの間に 8 文字が入ったストリング	サブフィールドを含まないデータ構造または 1 つのサブフィールドを含むデータ構造。サブフィールド指定の 43 桁目と 52 桁目は **ブランク** とする。長さは 8 とする。あるいは、文字入力フィールドまたは計算結果フィールド。	データ指定。サブフィールド指定の 40 桁目は ***T*** とする。44 ～ 80 桁目は TIMFMT(*ISO) とする。
TIMESTAMP	2 つのアポストロフィの間に 26 文字が入ったストリング	サブフィールドを含まないデータ構造または 1 つのサブフィールドを含むデータ構造。サブフィールド指定の 43 桁目と 52 桁目は **ブランク** とする。長さは 26 とする。あるいは、文字入力フィールドまたは計算結果フィールド。	データ指定。サブフィールド指定の 40 桁目は ***Z*** とする。
CLOB	-	-	CLOB 構造化形式 (ホスト言語での SQL プログラミング の「RPG」の章参照)
BLOB	-	-	BLOB 構造化形式 (ホスト言語での SQL プログラミング の「RPG」の章参照)
DBCLOB	-	-	DBCLOB 構造化形式 (ホスト言語での SQL プログラミング の「RPG」の章参照)
ROWID	-	-	ROWID 構造化形式 (ホスト言語での SQL プログラミング の「RPG」の章参照)
データ・リンク	-	-	-
標識変数	小数部 (およびオプションの先行符号) を伴わない数値ストリング	1 つのサブフィールドを含むデータ構造。 サブフィールド指定の 43 桁目は ***B***、長さは 2、52 桁目は 0 とする。	データ指定。サブフィールド指定の 40 桁目は ***B***、長さ <= 4、41 ～ 42 桁目は 00 とする。

標識変数とストアード・プロシージャー

CALL ステートメントで標識変数を使用すると、プロシージャーとの間で追加の情報を受け渡しすることができます (パラメーターとしてホスト変数が使用されるとすれば)。標識変数は、関連するホスト変数にヌル値が含まれることを示す SQL の標準的な手段であり、これが標識変数の主な使用法です。

関連するホスト変数にヌル値が含まれることを示す場合、2 バイト整数である標識変数は負の値に設定されます。標識変数を伴う CALL ステートメントは、次のように処理されます。

- 標識変数が負の値である場合、これはヌル値を示します。CALL の関連するホスト変数については省略時値が渡され、標識変数は未変更のまま渡されます。
- 標識変数が負の値でない場合、これはホスト変数にヌル値が入っていないことを示します。この場合、ホスト変数と標識変数は未変更のまま渡されます。

これらの処理規則は、プロシージャーへの入力パラメーターと、プロシージャーから戻される出力パラメーターについて同じです。ストアード・プロシージャーとともに標識変数を使用する場合には、関連するホスト変数を使用する前に、まず標識変数の値を検査することが、それらの処理をコーディングするための正しい方法です。

次の例では、CALL ステートメント内の標識変数を処理する方法を示します。関連する変数を使用する前に標識変数の値を検査する論理に注目してください。さらに、標識変数がプロシージャー PROC1 に渡される (2 バイト値の配列から成る 3 番目の引き数として) 方法にも注目してください。

プロシージャーは、次のように定義されているとします。

```
CREATE PROCEDURE PROC1
   (INOUT DECIMALOUT DECIMAL(7,2), INOUT DECOUT2 DECIMAL(7,2))
   EXTERNAL NAME LIB1.PROC1 LANGUAGE RPGLE
   GENERAL WITH NULLS)
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Program CRPG
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    D INOUT1          S              7P 2
    D INOUT1IND       S              4B 0
    D INOUT2          S              7P 2
    D INOUT2IND       S              4B 0
    C                   EVAL      INOUT1 = 1
    C                   EVAL      INOUT1IND  = 0
    C                   EVAL      INOUT2 = 1
    C                   EVAL      INOUT2IND = -2
    C/EXEC SQL CALL PROC1 (:INOUT1 :INOUT1IND , :INOUT2
    C+                         :INOUT2IND)
    C/END-EXEC
    C                   EVAL      INOUT1 = 1
    C                   EVAL      INOUT1IND  = 0
    C                   EVAL      INOUT2 = 1
    C                   EVAL      INOUT2IND = -2
    C/EXEC SQL CALL PROC1 (:INOUT1 :INOUT1IND , :INOUT2
    C+                         :INOUT2IND)
    C/END-EXEC
    C     INOUT1IND     IFLT      0
    C*                   :
    C*                  HANDLE NULL INDICATOR
    C*                   :
    C                   ELSE
    C*                   :
    C*                  INOUT1 CONTAINS VALID DATA
    C*                   :
    C                   ENDIF
    C*                   :
    C*                  HANDLE ALL OTHER PARAMETERS
    C*                  IN A SIMILAR FASHION
    C*                   :
    C                   RETURN
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
End of PROGRAM CRPG
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

図 3. CALL ステートメント内の標識変数の処理 (1/2)

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Program PROC1
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    D INOUTP          S              7P 2
    D INOUTP2         S              7P 2
    D NULLARRAY       S              4B 0 DIM(2)
    C     *ENTRY        PLIST
    C                   PARM                    INOUTP
    C                   PARM                    INOUTP2
    C                   PARM                    NULLARRAY
    C     NULLARRAY(1)  IFLT      0
    C*                   :
    C*                  INOUTP DOES NOT CONTAIN MEANINGFUL DATA
    C*
    C                   ELSE
    C*                   :
    C*                  INOUTP CONTAINS MEANINGFUL DATA
    C*                   :
    C                   ENDIF
    C*                  PROCESS ALL REMAINING VARIABLES
    C*
    C*                  BEFORE RETURNING, SET OUTPUT VALUE FOR FIRST
    C*                  PARAMETER AND SET THE INDICATOR TO A NON-NEGATIV
    C*                  VALUE SO THAT THE DATA IS RETURNED TO THE CALLING
    C*                  PROGRAM
    C*
    C                   EVAL      INOUTP2 = 20.5
    C                   EVAL      NULLARRAY(2) = 0
    C*
    C*                  INDICATE THAT THE SECOND PARAMETER IS TO CONTAIN
    C*                  THE NULL VALUE UPON RETURN. THERE IS NO POINT
    C*                  IN SETTING THE VALUE IN INOUTP SINCE IT WON'T BE
    C*                  PASSED BACK TO THE CALLER.
    C                   EVAL      NULLARRAY(1) = -5
    C                   RETURN
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
End of PROGRAM PROC1
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

図 3. CALL ステートメント内の標識変数の処理 (2/2)

呼び出しプログラムへの完了状況の戻し

SQL プロシージャーでは、プロシージャー内でハンドルされなかったエラーはすべて、SQLCA 内の呼び出し元に戻されます。 SIGNAL 制御ステートメントおよび RESIGNAL 制御ステートメントを使用して、エラー情報を送信することもできます。詳しくは、「SQL 解説書」の SQL プロシージャー、関数、およびトリガーのトピックを参照してください。

外部プロシージャーの場合は、状況情報を戻す 2 つの方法があります。CALL ステートメントを発行している SQL プログラムに状況を戻す方法の 1 つに、追加の INOUT タイプ・パラメーターをコーディングして、それをプロシージャーから戻す前に設定する方法があります。呼び出されているプロシージャーが既存のプログラムである場合、常にこの方法が可能であるとは限りません。

CALL ステートメントを発行している SQL プログラムに状況を戻す別の方法として、プロシージャーを呼び出す呼び出しプログラムにエスケープ・メッセージを送

る方法があります。プロシージャーを呼び出す呼び出しプログラムは、オペレーティング・システム・プログラム QSQCALL です。それぞれの言語には、条件を信号で通知し、メッセージを送るための方法があります。メッセージを送るための適切な方法を判別するには、それぞれの言語解説書を参照してください。メッセージが送られると、QSQCALL はエラーを SQLCODE/SQLSTATE -443/38501 に変えます。

CALL ステートメントの例

以下の例では、いくつかの言語のプロシージャーに CALL ステートメントの引き数を渡す方法が示されています。さらに、プロシージャーのローカル変数に引き数を取り込む方法が示されています。

最初の例では、CREATE PROCEDURE 定義を使用してプロシージャー P1 および P2 を呼び出す、ILE C 呼び出しプログラムを示します。プロシージャー P1 は C で作成されていて、10 個のパラメーターがあります。プロシージャー P2 は PL/I で作成されていて、やはり、10 個のパラメーターがあります。

2 つのプロシージャーは、次のように定義されているとします。

```
EXEC SQL CREATE PROCEDURE P1 (INOUT PARM1 CHAR(10),
                              INOUT PARM2 INTEGER,
                              INOUT PARM3 SMALLINT,
                              INOUT PARM4 FLOAT(22),
                              INOUT PARM5 FLOAT(53),
                              INOUT PARM6 DECIMAL(10,5),
                              INOUT PARM7 VARCHAR(10),
                              INOUT PARM8 DATE,
                              INOUT PARM9 TIME,
                              INOUT PARM10 TIMESTAMP)
          EXTERNAL NAME  TEST12.CALLPROC2
          LANGUAGE C GENERAL WITH NULLS

EXEC SQL CREATE PROCEDURE P2 (INOUT PARM1 CHAR(10),
                              INOUT PARM2 INTEGER,
                              INOUT PARM3 SMALLINT,
                              INOUT PARM4 FLOAT(22),
                              INOUT PARM5 FLOAT(53),
                              INOUT PARM6 DECIMAL(10,5),
                              INOUT PARM7 VARCHAR(10),
                              INOUT PARM8 DATE,
                              INOUT PARM9 TIME,
                              INOUT PARM10 TIMESTAMP)
          EXTERNAL NAME  TEST12.CALLPROC
          LANGUAGE PLI GENERAL WITH NULLS
```

例 1: ILE C アプリケーションから呼び出される ILE C および PL/I プロシージャー

注: コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

```
/*************************************************************/
/*********** START OF SQL C Application **********************/

#include <stdio.h>
#include <string.h>
#include <decimal.h>
main()
{
 EXEC SQL INCLUDE SQLCA;
 char PARM1[10];
 signed long int PARM2;
 signed short int PARM3;
 float PARM4;
 double PARM5;
 decimal(10,5) PARM6;
 struct { signed short int parm7l;
          char parm7c[10];
        } PARM7;
char PARM8[10];        /* FOR DATE */
char PARM9[8];         /* FOR TIME */
char PARM10[26];       /* FOR TIMESTAMP */
```

図 4. CREATE PROCEDURE および CALL の例 (1/2)

```
/*****************************************************/
/* Initialize variables for the call to the procedures */
/*****************************************************/
strcpy(PARM1,"PARM1");
PARM2 = 7000;
PARM3 = -1;
PARM4 = 1.2;
PARM5 = 1.0;
PARM6 = 10.555;
PARM7.parm7l = 5;
strcpy(PARM7.parm7c,"PARM7");
strncpy(PARM8,"1994-12-31",10);         /* FOR DATE      */
strncpy(PARM9,"12.00.00",8);            /* FOR TIME      */
strncpy(PARM10,"1994-12-31-12.00.00.000000",26);
                                        /* FOR TIMESTAMP */
/*********************************************/
/* Call the C procedure                      */
/*                                           */
/*                                           */
/*********************************************/
EXEC SQL CALL P1 (:PARM1, :PARM2, :PARM3,
                  :PARM4, :PARM5, :PARM6,
                  :PARM7, :PARM8, :PARM9,
                  :PARM10 );
if (strncmp(SQLSTATE,"00000",5))
 {
 /* Handle error or warning returned on CALL statement */
 }

/* Process return values from the CALL.                */
:


/*********************************************/
/* Call the PLI procedure                    */
/*                                           */
/*                                           */
/*********************************************/
/* Reset the host variables prior to making the CALL    */
/*                                                      */
:
EXEC SQL CALL P2 (:PARM1, :PARM2, :PARM3,
                  :PARM4, :PARM5, :PARM6,
                  :PARM7, :PARM8, :PARM9,
                  :PARM10 );
if (strncmp(SQLSTATE,"00000",5))
 {
  /* Handle error or warning returned on CALL statement */
}
/* Process return values from the CALL.                */
:
}

/******** END OF C APPLICATION *********************************/
/***************************************************************/
```

図 4. CREATE PROCEDURE および CALL の例 (2/2)

```
/******** START OF C PROCEDURE P1 ******************************/
/*        PROGRAM TEST12/CALLPROC2                              */
/*************************************************************/

#include <stdio.h>
#include <string.h>
#include <decimal.h>
main(argc,argv)
  int argc;
  char *argv[];
  {
    char parm1[11];
    long int parm2;
    short int parm3,i,j,*ind,ind1,ind2,ind3,ind4,ind5,ind6,ind7,
              ind8,ind9,ind10;
    float parm4;
    double parm5;
    decimal(10,5) parm6;
    char parm7[11];
    char parm8[10];
    char parm9[8];
    char parm10[26];
    /* ********************************************************/
    /* Receive the parameters into the local variables -       */
    /* Character, date, time, and timestamp are passed as      */
    /* NUL terminated strings - cast the argument vector to    */
    /* the proper data type for each variable. Note that       */
    /* the argument vector could be used directly instead of   */
    /* copying the parameters into local variables - the copy  */
    /* is done here just to illustrate the method.             */
    /* ********************************************************/

    /* Copy 10 byte character string into local variable      */
    strcpy(parm1,argv[1]);

    /* Copy 4 byte integer into local variable                */
    parm2 = *(int *) argv[2];

    /* Copy 2 byte integer into local variable                */
    parm3 = *(short int *) argv[3];

    /* Copy floating point number into local variable         */
    parm4 = *(float *) argv[4];

    /* Copy double precision number into local variable       */
    parm5 = *(double *) argv[5];

    /* Copy decimal number into local variable                */
    parm6 = *(decimal(10,5) *) argv[6];
```

図 5. サンプル・プロシージャー P1 (1/2)

```
/*********************************************************/
/* Copy NUL terminated string into local variable.         */
/* Note that the parameter in the CREATE PROCEDURE was   */
/* declared as varying length character. For C, varying   */
/* length are passed as NUL terminated strings unless     */
/* FOR BIT DATA is specified in the CREATE PROCEDURE     */
/*********************************************************/
strcpy(parm7,argv[7]);

/*********************************************************/
/* Copy date into local variable.                        */
/* Note that date and time variables are always passed in */
/* ISO format so that the lengths of the strings are      */
/* known. strcpy would work here just as well.            */
/*********************************************************/
strncpy(parm8,argv[8],10);

/* Copy time into local variable                         */
strncpy(parm9,argv[9],8);

/*********************************************************/
/* Copy timestamp into local variable.                   */
/* IBM SQL timestamp format is always passed so the length*/
/* of the string is known.                               */
/*********************************************************/
strncpy(parm10,argv[10],26);

/*********************************************************/
/* The indicator array is passed as an array of short     */
/* integers. There is one entry for each parameter passed */
/* on the CREATE PROCEDURE (10 for this example).        */
/* Below is one way to set each indicator into separate   */
/* variables.                                            */
/*********************************************************/
   ind = (short int *) argv[11];
   ind1 = *(ind++);
   ind2 = *(ind++);
   ind3 = *(ind++);
   ind4 = *(ind++);
   ind5 = *(ind++);
   ind6 = *(ind++);
   ind7 = *(ind++);
   ind8 = *(ind++);
   ind9 = *(ind++);
   ind10 = *(ind++);
  :
/* Perform any additional processing here                */
  :
return;
}
/******** END OF C PROCEDURE P1 *****************************/
```

図 5. サンプル・プロシージャー P1 (2/2)

```
/******** START OF PL/I PROCEDURE P2 *************************/
/******** PROGRAM TEST12/CALLPROC ****************************/
/************************************************************/


CALLPROC :PROC( PARM1,PARM2,PARM3,PARM4,PARM5,PARM6,PARM7,
                      PARM8,PARM9,PARM10,PARM11);
 DCL  SYSPRINT FILE STREAM OUTPUT EXTERNAL;
 OPEN FILE(SYSPRINT);
 DCL PARM1 CHAR(10);
 DCL PARM2 FIXED BIN(31);
 DCL PARM3 FIXED BIN(15);
 DCL PARM4 BIN FLOAT(22);
 DCL PARM5 BIN FLOAT(53);
 DCL PARM6 FIXED DEC(10,5);
 DCL PARM7 CHARACTER(10) VARYING;
 DCL PARM8 CHAR(10);      /* FOR DATE */
 DCL PARM9 CHAR(8);       /* FOR TIME */
 DCL PARM10 CHAR(26);     /* FOR TIMESTAMP */
 DCL PARM11(10) FIXED BIN(15);   /* Indicators */

 /* PERFORM LOGIC - Variables can be set to other values for  */
 /* return to the calling program.                            */

 :

 END CALLPROC;
```

図 6. サンプル・プロシージャー P2

次の例では、ILE C プログラムから呼び出される REXX プロシージャーを示します。

プロシージャーは、次のように定義されているとします。

```
EXEC SQL CREATE PROCEDURE REXXPROC
              (IN PARM1 CHARACTER(20),
               IN PARM2 INTEGER,
               IN PARM3 DECIMAL(10,5),
               IN PARM4 DOUBLE PRECISION,
               IN PARM5 VARCHAR(10),
               IN PARM6 GRAPHIC(4),
               IN PARM7 VARGRAPHIC(10),
               IN PARM8 DATE,
               IN PARM9 TIME,
               IN PARM10 TIMESTAMP)
         EXTERNAL NAME 'TEST.CALLSRC(CALLREXX)'
         LANGUAGE REXX GENERAL WITH NULLS
```

例 2. C アプリケーションから呼び出される REXX プロシージャー

注: コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

```
/************************************************************/
/*********** START OF SQL C Application **********************/

        #include <decimal.h>
        #include <stdio.h>
        #include <string.h>
        #include <wcstr.h>
        /*-----------------------------------------------------------*/
        exec sql include sqlca;
        exec sql include sqlda;
        /* ********************************************************/
        /* Declare host variable for the CALL statement              */
        /* ********************************************************/
        char parm1[20];
        signed long int parm2;
        decimal(10,5) parm3;
        double parm4;
        struct {  short dlen;
                  char dat[10];
                } parm5;
        wchar_t parm6[4] = { 0xC1C1, 0xC2C2, 0xC3C3, 0x0000 };
        struct {  short  dlen;
                  wchar_t dat[10];
                } parm7 = {0x0009, 0xE2E2,0xE3E3,0xE4E4, 0xE5E5, 0xE6E6,
                           0xE7E7, 0xE8E8, 0xE9E9, 0xC1C1, 0x0000 };

        char parm8[10];
        char parm9[8];
        char parm10[26];
        main()
        {
```

図 7. C アプリケーションから呼び出されるサンプル REXX プロシージャー (1/4)

```
        /* ************************************************************/
        /* Call the procedure - on return from the CALL statement the  */
        /* SQLCODE should be 0. If the SQLCODE is non-zero,            */
        /* the procedure detected an error.                            */
        /* ************************************************************/
        strcpy(parm1,"TestingREXX");
        parm2 = 12345;
        parm3 = 5.5;
        parm4 = 3e3;
        parm5.dlen = 5;
        strcpy(parm5.dat,"parm6");
        strcpy(parm8,"1994-01-01");
        strcpy(parm9,"13.01.00");
        strcpy(parm10,"1994-01-01-13.01.00.000000");

         EXEC SQL CALL REXXPROC (:parm1, :parm2,
                                 :parm3,:parm4,
                                 :parm5, :parm6,
                                 :parm7,
                                 :parm8, :parm9,
                                 :parm10);

         if (strncpy(SQLSTATE,"00000",5))
           {
           /* handle error or warning returned on CALL   */
           :
           }
              :
       }

/****** END OF SQL C APPLICATION   ***********************************/
/*********************************************************************/
```

図 7. C アプリケーションから呼び出されるサンプル REXX プロシージャー (2/4)

```
/*********************************************************************/
/****** START OF REXX MEMBER TEST/CALLSRC CALLREXX ********************/
/*********************************************************************/
       /* REXX source member TEST/CALLSRC CALLREXX               */
       /* Note the extra parameter being passed for the indicator*/
       /* array.                                                 */
       /*                                                        */
       /* ACCEPT THE FOLLOWING INPUT VARIABLES SET TO THE        */
       /* SPECIFIED VALUES :                                     */
       /* AR1        CHAR(20)          =  'TestingREXX'          */
       /* AR2        INTEGER           = 12345                   */
       /* AR3        DECIMAL(10,5)     = 5.5                     */
       /* AR4        DOUBLE PRECISION  = 3e3                     */
       /* AR5        VARCHAR(10)       = 'parm6'                 */
       /* AR6        GRAPHIC           = G'C1C1C2C2C3C3'         */
       /* AR7        VARGRAPHIC        =                         */
       /*             G'E2E2E3E3E4E4E5E5E6E6E7E7E8E8E9E9EAEA'   */
       /* AR8        DATE              = '1994-01-01'            */
       /* AR9        TIME              = '13.01.00'              */
       /* AR10       TIMESTAMP         =                         */
       /*                       '1994-01-01-13.01.00.000000'     */
       /* AR11       INDICATOR ARRAY   = +0+0+0+0+0+0+0+0+0+0    */

       /*********************************************************/
       /* Parse the arguments into individual parameters        */
       /*********************************************************/
       parse arg ar1 ar2 ar3 ar4 ar5 ar6 ar7 ar8 ar9 ar10 ar11

       /*********************************************************/
       /* Verify that the values are as expected                */
       /*********************************************************/
       if ar1<>"'TestingREXX'" then signal ar1tag
       if ar2<>12345 then signal ar2tag
       if ar3<>5.5 then signal ar3tag
       if ar4<>3e3 then signal ar4tag
       if ar5<>"'parm6'" then signal ar5tag
       if ar6 <>"G'AABBCC'" then signal ar6tag
       if ar7 <>"G'SSTTUUVVWWXXYYZZAA'" then ,
        signal ar7tag
       if ar8 <> "'1994-01-01'" then signal ar8tag
       if ar9 <> "'13.01.00'" then signal ar9tag
       if ar10 <> "'1994-01-01-13.01.00.000000'" then signal ar10tag
       if ar11 <> "+0+0+0+0+0+0+0+0+0+0" then signal ar11tag
```

図 7. C アプリケーションから呼び出されるサンプル REXX プロシージャー (3/4)

```
        /*************************************************************/
        /* Perform other processing as necessary ..                  */
        /*************************************************************/
           :
        /*************************************************************/
        /* Indicate the call was successful by exiting with a        */
        /* return code of 0                                          */
        /*************************************************************/
        exit(0)

        ar1tag:
        say "ar1 did not match" ar1
        exit(1)
        ar2tag:
        say "ar2 did not match" ar2
        exit(1)
        :
        :

/************ END OF REXX MEMBER   *********************************/
```

図 7. C アプリケーションから呼び出されるサンプル REXX プロシージャー (4/4)

第 12 章 オブジェクト・リレーショナル機能の使用

この章では、DB2 UDB のオブジェクト指向機能について説明します。
- DB2 UDB オブジェクト拡張を使用する理由
- オブジェクトをサポートするための DB2 UDB の方法
- ラージ・オブジェクト (LOB) の使用
- ユーザー定義関数 (UDF)
- ユーザー定義の特殊タイプ (UDT)
- UDT、UDF、および LOB の間の協同

プログラム言語テクノロジーの最近の最も重要な開発の 1 つにオブジェクト・オリエンテーション があります。オブジェクト・オリエンテーションとは、アプリケーション・ドメインの中のエンティティーは、分類によって相互に関連付けられている独立オブジェクトとしてモデル化できるという概念のことです。オブジェクト・オリエンテーションを使用すると、アプリケーション・ドメインの中のオブジェクト間の類似点と相違点を把握し、これらのオブジェクトを関連タイプにグループ化することができます。同じタイプのオブジェクトは、タイプ固有の関数の同じセットを共用するので、その動作が同じになります。これらの関数がアプリケーション・ドメイン内のオブジェクトの動作を表します。

DB2 UDB オブジェクト拡張を使用する理由

DB2 UDB のオブジェクト拡張を使用すると、オブジェクト指向 (OO) の概念と手順をリレーショナル・データベースに組み入れることができます。これは、DB2 UDB を、そのタイプと関数を豊富なセットで拡張することによって行います。この拡張機能の使用により、オブジェクト指向データ・タイプのインスタンスを表の列に格納し、SQL ステートメントの関数を使用してインスタンスを操作することができます。さらに、格納オブジェクトの意味体系の動作を、データベースを介してすべてのアプリケーションで共用できる重要なリソースにすることができます。

オブジェクト・オリエンテーションをリレーショナル・データベース・システムに組み入れるには、ユーザー自身の新しいタイプと関数を定義します。これらの新しいタイプと関数は、アプリケーション・ドメインにあるオブジェクトの意味体系を反映したものでなければなりません。モデル化したいデータ・オブジェクトには、大量で複合した (たとえば、テキスト、音声、イメージ、財務データなど) ものがある場合があります。したがって、ラージ・オブジェクトの格納と操作のためのメカニズムが必要になります。ユーザー定義の特殊タイプ (UDT)、ユーザー定義関数 (UDF)、およびラージ・オブジェクト (LOB) が、DB2 UDB に備わっているメカニズムです。DB2 UDB を使用すれば、ユーザー自身の新しいタイプと関数を定義し、データベースの中で、アプリケーション・オブジェクトを格納し操作できます。

この後の節で述べますが、オブジェクト指向の機能の間には、重要な協同効果があります。複合オブジェクトをアプリケーション・ドメインで UDT としてモデル化するとします。UDT は内部では LOB として表現されます。すると、UDT の動作は、UDF としてインプリメントすることができます。このセクションでは、UDT

と UDF を定義するのに必要なステップとともに、LOB の使用方法について説明します。 UDT、UDF、および LOB がアプリケーション内のオブジェクトをどのように表すか、これらのものがどのように一緒に作動するかについて説明します。

注: DB2 のオブジェクト指向のメカニズム (UDT、UDF、および LOB) の使用は、オブジェクト指向アプリケーションをサポートすることに限定されているわけではありません。 C++ プログラム言語がオブジェクト指向ではないあらゆる種類のアプリケーションを具体化するのに使用できるのと同様に、DB2 UDB が提供しているオブジェクト指向メカニズムも、オブジェクト指向ではないあらゆる種類のアプリケーションをサポートすることができます。 UDT、UDF、および LOB は、すべてのデータベース・アプリケーションをモデル化するのに使用できる汎用のメカニズムです。したがって、DB2 UDB オブジェクト拡張機能は、従来のアプリケーションではないオブジェクト指向アプリケーションをサポートできるだけではなく、従来アプリケーションのサポートの改善にも役立ちます。

オブジェクトをサポートするための DB2 UDB の方法

DB2 UDB のオブジェクト拡張機能によって、リレーショナル・テクノロジーの機能を利用し、オブジェクト・テクノロジーの利点を実現することができます。リレーショナル・システムでは、データ・タイプは、表の列に格納されているデータを記述し、表の列にはデータ・タイプのインスタンスすなわちオブジェクトが格納されます。インスタンスに対する操作は、式が使用できるところであればどこでも呼び出すことができる演算子または関数を使用してサポートされます。

オブジェクト拡張機能をサポートするための DB2 UDB アプローチは、リレーショナル・パラダイム (枠組み) に適合するものです。UDT は、ユーザーが定義するデータ・タイプです。UDT は、組み込まれたタイプと同様に、表の列に格納されているデータを記述するのに使用できます。UDF はユーザーが定義する関数です。UDF は、組み込み関数や演算子と同様に、UDT インスタンスの操作をサポートします。したがって、UDT インスタンスは表の列に格納され、SQL 照会では UDF によって操作されます。UDT は、内部では、さまざまな方法で表されます。LOB は 1 つの例にすぎません。

ラージ・オブジェクト (LOB) の使用

VARCHAR データ・タイプと VARGRAPHIC データ・タイプの記憶域限界は 32K バイトです。 このサイズは小規模ないし中規模のテキスト・データには十分ですが、アプリケーションによっては大きなテキスト文書を格納する必要があります。また、アプリケーションによっては、さまざまな種類のデータ・タイプ (オーディオ、ビデオ、図面、テキストとグラフィックスが混合しているもの、およびイメージなど) を格納する必要がある場合があります。 DB2 は、これらのデータ・オブジェクトを、サイズが最大 2 G バイト (GB) のストリングとして保管するために 3 つのデータ・タイプを提供しています。 3 つのデータ・タイプは、バイナリー・ラージ・オブジェクト (BLOB)、1 バイト文字ラージ・オブジェクト (CLOB)、および 2 バイト文字ラージ・オブジェクト (DBCLOB) です。

ラージ・オブジェクト (LOB) を格納するのに加え、データベース内の LOB のおのおのを参照し、使用し、変更するメソッドが必要になります。 それぞれの DB2

UDB 表には大量の関連 LOB データが入っている場合があります。 1 つまたは複数の LOB 値が入っている単一行は 3.5 メガバイトを超えることはできませんが、1 つの表には、256 ギガバイトに近い LOB データを入れることができます。 特定の行の LOB 列の内容には、どの時点においても、ラージ・オブジェクト*値* が入っています。

ホスト変数を使用すれば、他のすべてのデータ・タイプと同じように、LOB を参照し、操作することができます。ただし、ホスト変数はクライアント・メモリー・バッファーを使用するので、LOB 値を収容するには大きさが足りない場合があります。これらの大きな値を操作するために、他の方法が必要になります。*ロケーター* は、データベース・サーバーでラージ・オブジェクトの値を識別して操作するのに、さらに、LOB 値の各部分を取り出すのに便利です。 *ファイル参照変数* は、クライアントに、またはクライアントから、ラージ・オブジェクト値 (あるいは、その大きな一部) を物理的に移動するのに便利です。

以下のセクションでは、上に述べたトピックについてさらに詳しく説明します。

- 『ラージ・オブジェクトのデータ・タイプ (BLOB、CLOB、DBCLOB) について』
- 208 ページの『ラージ・オブジェクト・ロケーターについて』
- 209 ページの『例: CLOB 値を処理するためのロケーターの使用』
- 212 ページの『標識変数および LOB ロケーター』
- 212 ページの『LOB ファイル参照変数』
- 214 ページの『例: ファイルへの文書の抽出』
- 216 ページの『例: CLOB 列へのデータの挿入』
- 216 ページの『LOB 列のレイアウトの表示』
- 217 ページの『LOB 列のジャーナル・エントリーのレイアウト』

ラージ・オブジェクトのデータ・タイプ (BLOB、CLOB、DBCLOB) について

ラージ・オブジェクトのデータ・タイプは、サイズが 0 バイトから 2 メガバイトまでの範囲のデータを保管します。

以下に、ラージ・オブジェクトの 3 つのデータ・タイプの定義を示します。

- 文字ラージ・オブジェクト (CLOB) - 単一バイト文字からなる文字ストリングで、関連したコード・ページをもっています。このデータ・タイプは、情報量が正規の VARCHAR データ・タイプの限界 (32K バイトが上限) を超えて増える可能性があるテキスト情報を収容するのに最も適しています。情報のコード・ページ変換はサポートされており、他の文字タイプとの互換性も保たれます。
- 2 バイト文字ラージ・オブジェクト (DBCLOB) - 2 バイト文字からなる文字ストリングで、関連したコード・ページをもっています。このデータ・タイプは、2 バイト文字を使用するテキスト情報を保持するのに最も適しています。この場合も、情報のコード・ページ変換はサポートされており、他の 2 バイト文字タイプとの互換性が保たれます。
- バイナリー・ラージ・オブジェクト (BLOB) - バイトを 2 進ストリングで表したもので、関連したコード・ページはありません。このデータ・タイプは、2 進データを格納するのに最も便利です。したがって、これは、ユーザー定義の特殊タ

イプ (UDT) で使用するときに都合のよいソース・タイプです。UDT は、イメージ、音声、グラフィック、および他のタイプのビジネスまたはアプリケーション固有のデータを格納するのに、BLOB をソース・タイプとして使用して作成されます。UDT の詳細については、 234 ページの『ユーザー定義の特殊タイプ (UDT)』を参照してください。

ラージ・オブジェクト・ロケーターについて

LOB ロケーターは、概念的に、簡単に管理できる小さな値を使用してより大きな値を参照するという単純なアイデアを表したものです。具体的には、LOB ロケーターはホスト変数に格納されている 4 バイトの値で、プログラムはこれを使用してデータベース・システムに保持されている LOB 値 (または LOB 式) を参照します。LOB ロケーターを使用すると、プログラムは、正規のホスト変数に格納されている LOB 値と同じように、LOB 値を操作することができます。LOB ロケーターを使用すれば、LOB 値をサーバーからアプリケーションに (また、アプリケーションからサーバーに) トランスポートする必要がなくなります。

LOB ロケーターは、データベース内の行または物理記憶域位置ではなく、LOB 値または LOB 式に関連付けられます。したがって、LOB 値を選択してロケーターに入れた後では、ロケーターが参照する値に影響する元の行または表に対する操作は実行できなくなります。ロケーターに関連付けられた値は、作業単位が終了するか、ロケーターが明示的に解放されるか、そのどちらかが最初に発生するまで、有効です。FREE LOCATOR ステートメントは、ロケーターをその関連値から解放します。同様に、コミットまたはロールバック操作は、トランザクションに関連した LOB ロケーターをすべて解放します。

LOB ロケーターは、DB2 UDB と UDF との間で受け渡しすることもできます。UDF 内では、LOB データを処理する関数を、LOB ロケーターを使用して LOB 値を操作するのに使用できます。

LOB 値を選択するときには、3 つのオプションがあります。

- LOB 値全体を選択してホスト変数に入れる。 LOB 値全体がホスト変数の中にコピーされます。
- LOB 値を選択して LOB ロケーターに入れる。LOB 値はサーバーに残ります。ホスト変数にコピーされません。
- LOB 値全体を選択してファイル参照変数に入れる。LOB 値は、統合ファイル・システム (IFS) ファイルに移動されます。

プログラムの中で使用する LOB 値によって、プログラマーは、どの方式を使うのがよいかを決めることができます。 LOB 値が非常に大きく、後に続く 1 つまたは複数の SQL ステートメントに対する入力値としてのみ必要な場合は、値をロケーターに入れたままにしておきます。

サイズにかかわらず LOB 値全体をプログラムが必要とする場合は、LOB を転送するしかありません。この場合でも、まだオプションが使用できます。 値全体を選択して、正規の、または、ファイル参照のホスト変数に入れることができます。 また、LOB 値を選択してロケーターの中に入れ、LOB 値を 1 つずつロケーターから読み取り、正規のホスト変数に入れることもできます。これを、以下の例 209 ページの『例: CLOB 値を処理するためのロケーターの使用』 で示します。

例: CLOB 値を処理するためのロケーターの使用

この例では、アプリケーション・プログラムは LOB 値のロケーターを取り出し、次に、ロケーターを使用して LOB 値からデータを抜き出します。このメソッドを使用すると、プログラムは、LOB データの一部分を入れるのに必要な記憶域 (この大きさはプログラムが決めます) だけを割り振ります。さらに、プログラムは、カーソルを使用してフェッチ呼び出しを一回出すだけですみます。

注: コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

サンプル LOBLOC プログラムの作動方法

1. **ホスト変数を宣言する。** BEGIN DECLARE SECTION ステートメントおよび END DECLARE SECTION ステートメントによってホスト変数宣言が区切られます。SQL ステートメントで参照される場合は、ホスト変数の前にコロン (:) が付きます。CLOB LOCATOR ホスト変数が宣言されます。
2. **LOB 値をフェッチしてロケーター・ホスト変数に入れる。** CURSOR および FETCH ルーチンを使用してデータベース内の LOB フィールドの位置を入手し、ロケーター・ホスト変数に入れます。
3. **LOB LOCATORS を解放する。**この例で使用されている LOB LOCATORS が解放され、以前に関連していた値からロケーターを解放します。

CHECKERR マクロ/関数はエラー検査のユーティリティーで、プログラムの外部にあります。 このエラー検査ユーティリティーの位置は、使用されるプログラム言語によって異なります。

C check_error は CHECKERR として再定義され、util.c ファイルの中にあります。

C サンプル: LOBLOC.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

#define  CHECKERR(CE_STR)   if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

#ifdef DB2MAC
   char * bufptr;
#endif

   EXEC SQL BEGIN DECLARE SECTION; 1
      char number[7];
      long deptInfoBeginLoc;
      long deptInfoEndLoc;
      SQL TYPE IS CLOB_LOCATOR resume;
      SQL TYPE IS CLOB_LOCATOR deptBuffer;
      short lobind;
      char buffer[1000]="";
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: LOBLOC\n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
  CHECKERR ("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
```

```
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      CHECKERR ("CONNECT TO SAMPLE");
   }
   else {
      printf ("\nUSAGE: lobloc [userid passwd]\n\n");
      return 1;
   } /* endif */

   /* Employee A10030 is not included in the following select, because
      the lobeval program manipulates the record for A10030 so that it is
      not compatible with lobloc */

   EXEC SQL DECLARE c1 CURSOR FOR
            SELECT empno, resume FROM emp_resume WHERE resume_format='ascii'
            AND empno <> 'A00130';

   EXEC SQL OPEN c1;
   CHECKERR ("OPEN CURSOR");

   do {
      EXEC SQL FETCH c1 INTO :number, :resume :lobind;  2
      if (SQLCODE != 0) break;
      if (lobind < 0) {
         printf ("NULL LOB indicated\n");
      } else {
         /* EVALUATE the LOB LOCATOR */
         /* Locate the beginning of "Department Information" section */
         EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
            INTO :deptInfoBeginLoc;
         CHECKERR ("VALUES1");

         /* Locate the beginning of "Education" section (end of "Dept.Info" */
         EXEC SQL VALUES (POSSTR(:resume, 'Education'))
            INTO :deptInfoEndLoc;
         CHECKERR ("VALUES2");

         /* Obtain ONLY the "Department Information" section by using SUBSTR */
         EXEC SQL VALUES(SUBSTR(:resume, :deptInfoBeginLoc,
            :deptInfoEndLoc - :deptInfoBeginLoc)) INTO :deptBuffer;
         CHECKERR ("VALUES3");

         /* Append the "Department Information" section to the :buffer var. */
         EXEC SQL VALUES(:buffer || :deptBuffer) INTO :buffer;
         CHECKERR ("VALUES4");
      } /* endif */
   } while ( 1 );

#ifdef DB2MAC
   /* Need to convert the newline character for the Mac */
   bufptr = &(buffer[0]);
   while ( *bufptr != '\0' ) {
      if ( *bufptr == 0x0A ) *bufptr = 0x0D;
  bufptr++;
   }
#endif

   printf ("%s\n",buffer);

   EXEC SQL FREE LOCATOR :resume, :deptBuffer;  3
   CHECKERR ("FREE LOCATOR");

   EXEC SQL CLOSE c1;
   CHECKERR ("CLOSE CURSOR");

   EXEC SQL CONNECT RESET;
   CHECKERR ("CONNECT RESET");
   return 0;
}
/* end of program : LOBLOC.SQC */
```

COBOL サンプル: LOBLOC.SQB

```
 Identification Division.
 Program-ID. "lobloc".

 Data Division.
 Working-Storage Section.
     copy "sqlenv.cbl".
     copy "sql.cbl".
     copy "sqlca.cbl".

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.  1
 01 userid            pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
```

```
   49 passwd-name     pic x(18).
 01 empnum            pic x(6).
 01 di-begin-loc      pic s9(9) comp-5.
 01 di-end-loc        pic s9(9) comp-5.
 01 resume            USAGE IS SQL TYPE IS CLOB-LOCATOR.
 01 di-buffer         USAGE IS SQL TYPE IS CLOB-LOCATOR.
 01 lobind            pic s9(4) comp-5.
 01 buffer            USAGE IS SQL TYPE IS CLOB(1K).
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc          pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: LOBLOC".

* Get database connection information.
     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR
* format with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

* Employee A10030 is not included in the following select, because
* the lobeval program manipulates the record for A10030 so that it is
* not compatible with lobloc

     EXEC SQL DECLARE c1 CURSOR FOR
              SELECT empno, resume FROM emp_resume
              WHERE resume_format = 'ascii'
              AND empno <> 'A00130' END-EXEC.

     EXEC SQL OPEN c1 END-EXEC.
     move "OPEN CURSOR" to errloc.
     call "checkerr" using SQLCA errloc.

     Move 0 to buffer-length.

     perform Fetch-Loop thru End-Fetch-Loop
        until SQLCODE not equal 0.

* display contents of the buffer.
     display buffer-data(1:buffer-length).

     EXEC SQL FREE LOCATOR :resume, :di-buffer END-EXEC.  3
     move "FREE LOCATOR" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL CLOSE c1 END-EXEC.
     move "CLOSE CURSOR" to errloc.
     call "checkerr" using SQLCA errloc.

     EXEC SQL CONNECT RESET END-EXEC.
     move "CONNECT RESET" to errloc.
     call "checkerr" using SQLCA errloc.
 End-Main.
        go to End-Prog.

 Fetch-Loop Section.
     EXEC SQL FETCH c1 INTO :empnum, :resume :lobind  2
         END-EXEC.

     if SQLCODE not equal 0
        go to End-Fetch-Loop.

* check to see if the host variable indicator returns NULL.
     if lobind less than 0 go to NULL-lob-indicated.

* Value exists.  Evaluate the LOB locator.
* Locate the beginning of "Department Information" section.
     EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
              INTO :di-begin-loc END-EXEC.
     move "VALUES1" to errloc.
     call "checkerr" using SQLCA errloc.
```

```
* Locate the beginning of "Education" section (end of Dept.Info)
     EXEC SQL VALUES (POSSTR(:resume, 'Education'))
               INTO :di-end-loc END-EXEC.
     move "VALUES2" to errloc.
     call "checkerr" using SQLCA errloc.

     subtract di-begin-loc from di-end-loc.

* Obtain ONLY the "Department Information" section by using SUBSTR
     EXEC SQL VALUES (SUBSTR(:resume, :di-begin-loc,
              :di-end-loc))
              INTO :di-buffer END-EXEC.
     move "VALUES3" to errloc.
     call "checkerr" using SQLCA errloc.

* Append the "Department Information" section to the :buffer var
     EXEC SQL VALUES (:buffer || :di-buffer) INTO :buffer
              END-EXEC.
     move "VALUES4" to errloc.
     call "checkerr" using SQLCA errloc.

     go to End-Fetch-Loop.

 NULL-lob-indicated.
     display "NULL LOB indicated".

 End-Fetch-Loop. exit.

 End-Prog.
           stop run.
```

標識変数および LOB ロケーター

アプリケーション・プログラムの通常のホスト変数の場合、ヌル値を選択してホスト変数に入れると、標識変数に負の値が割り当てられ、値がヌル値であることが示されます。ただし、LOB ロケーターの場合は、標識変数の意味が少し異なります。ロケーター・ホスト変数自身が決してヌル値になることはないため、標識変数の負の値は、LOB ロケーターによって表される LOB 値がヌル値であることを示します。ヌル値の情報は、標識変数値を使用しているクライアントの中に保持され、サーバーが有効なロケーターを使用してヌル値をトラッキングすることはありません。

LOB ファイル参照変数

ファイル参照変数はホスト変数に似ていますが、IFS ファイルにまたは IFS ファイルから (メモリー・バッファーにまたはメモリー・バッファーからではなく) データを転送するのに使用される点が異なります。ファイル参照変数は、ファイルを (含むのではなく) 表します。これは、LOB ロケーターが、LOB 値を (含むのではなく) 表すのに似ています。 データベースの照会、更新、および挿入は、ファイル参照変数を使用して単一の LOB 値を格納したり取り出したりすることができます。

ラージ・オブジェクトの場合は、ファイルが通常のコンテナーになります。多くの場合、ほとんどの LOB は、クライアント上のファイルに格納されたデータとして始まり、その後、サーバーのデータベースに移動されます。ファイル参照変数を使用すると、LOB データの移動が容易に行えます。プログラムはファイル参照変数を使用して、LOB データを IFS ファイルからデータベース・エンジンに直接転送します。LOB データの移動を実行するために、ファイルを読み書きするユーティリティー・ルーチンを、ホスト変数を使用してアプリケーションで作成する必要はありません。

注: ファイル参照変数で参照されるファイルは、プログラムが実行されるシステムからアクセス可能でなければなりません (必ずしもシステムに常駐している必要はありません)。ストアード・プロシージャーの場合、このシステムはサーバーになります。

ファイル参照変数にはデータ・タイプ BLOB、CLOB、または DBCLOB があります。これは、データのソース (入力) またはデータのターゲット (出力) として使用されます。ファイル参照変数は、相対ファイル名またはファイルの完全パス名をもつことができます (後者をお勧めします)。ファイル名の長さはアプリケーション・プログラムの中で指定します。ファイル参照変数のデータ長の部分は入力中は使用されません。出力のときに、データ長が、アプリケーション・リクエスター・コードによって、ファイルに書き込まれる新しいデータの長さに設定されます。

ファイル参照変数を使用するとき、入力と出力の両方に異なるオプションがあります。ファイル参照変数構造の `file_options` フィールドの設定を行って、ファイルに対するアクションを選択する必要があります。フィールドに対して割り当てる、入出力の両方をカバーする値の選択を以下に示します。

入力ファイル参照変数を使用するときの値 (C の場合を示します) とオプションを以下に示します。

- **SQL_FILE_READ** (正規ファイル) - このオプションの値は 2 です。これは、オープン、読み取り、およびクローズができるファイルです。DB2 UDB は、ファイルをオープンするときに、ファイルの中のデータの長さ (バイト数) を判断します。次に、DB2 UDB は、ファイル参照変数構造の `data_length` フィールドを介してこの長さを戻します。(COBOL の場合の値は SQL-FILE-READ です。)

出力ファイル参照変数を使用するときの値とオプションを以下に示します。

- **SQL_FILE_CREATE** (新規ファイル) - このオプションの値は 8 です。このオプションにより、新規ファイルが作成されます。ファイルがすでに存在する場合には、エラー・メッセージが戻されます。(COBOL の場合の値は SQL-FILE-CREATE です。)
- **SQL_FILE_OVERWRITE** (上書きファイル) - このオプションの値は 16 です。新規ファイルがない場合、このオプションにより新規ファイルが作成されます。ファイルがすでに存在する場合は、ファイルの中のデータは新しいデータで上書きされます。(COBOL の場合の値は SQL-FILE-OVERWRITE です。)
- **SQL_FILE_APPEND** (付加ファイル) - このオプションの値は 32 です。このオプションにより、ファイルがある場合には、そのファイルに出力が付加されます。ファイルがない場合は、新しいファイルが作成されます。 (COBOL の場合の値は SQL-FILE-APPEND です。)

注: OPEN ステートメントで LOB ファイル参照変数が使用される場合は、カーソルがクローズされるまで、LOB ファイル参照変数に関連付けられたファイルを削除しないでください。

統合ファイル・システムの詳細については、統合ファイル・システムを参照してください。

例: ファイルへの文書の抽出

次のプログラムの例は、CLOB 要素を表から取り出して外部ファイルに入れる方法を示しています。

サンプル LOBFILE プログラムの処理内容

1. **ホスト変数を宣言する。** BEGIN DECLARE SECTION ステートメントおよび END DECLARE SECTION ステートメントによってホスト変数宣言が区切られます。SQL ステートメントで参照される場合は、ホスト変数の前にコロン (:) が付きます。CLOB FILE REFERENCE ホスト変数が宣言されます。
2. **CLOB FILE REFERENCE ホスト変数がセットアップされる。** FILE REFERENCE の属性がセットアップされます。完全宣言パスをもたないファイル名が、省略時値で、ユーザーの現行ディレクトリーに入れられます。パス名がスラッシュ (/) で始まっていない場合は修飾されていません。
3. **CLOB FILE REFERENCE ホスト変数に選択して入れる。**データが resume フィールドから選択され、ホスト変数によって参照されるファイル名に入れられます。

CHECKERR マクロ/関数はエラー検査のユーティリティーで、プログラムの外部にあります。このエラー検査ユーティリティーの位置は、使用されるプログラム言語によって異なります。

C check_error は CHECKERR として再定義され、util.c ファイルの中にあります。

COBOL CHECKERR は checkerr.cbl という名の外部プログラムです。

注: コーディング例に関する詳細は、x ページの『コードに関する特記事項』を参照してください。

C サンプル: LOBFILE.SQC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sql.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

#define  CHECKERR(CE_STR)   if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

   EXEC SQL BEGIN DECLARE SECTION; 1
      SQL TYPE IS CLOB_FILE resume;
      short lobind;
      char userid[9];
      char passwd[19];
   EXEC SQL END DECLARE SECTION;

   printf( "Sample C program: LOBFILE\n" );

   if (argc == 1) {
      EXEC SQL CONNECT TO sample;
  CHECKERR ("CONNECT TO SAMPLE");
   }
   else if (argc == 3) {
      strcpy (userid, argv[1]);
      strcpy (passwd, argv[2]);
      EXEC SQL CONNECT TO sample USER :userid USING :passwd;
      CHECKERR ("CONNECT TO SAMPLE");
   }
   else {
      printf ("\nUSAGE: lobfile [userid passwd]\n\n");
      return 1;
```

```
   } /* endif */

   strcpy (resume.name, "RESUME.TXT");  2
   resume.name_length = strlen("RESUME.TXT");
   resume.file_options = SQL_FILE_OVERWRITE;

   EXEC SQL SELECT resume INTO :resume :lobind FROM emp_resume  3
      WHERE resume_format='ascii' AND empno='000130';

   if (lobind < 0) {
      printf ("NULL LOB indicated \n");
   } else {
      printf ("Resume for EMPNO 000130 is in file : RESUME.TXT\n");
   } /* endif */

   EXEC SQL CONNECT RESET;
   CHECKERR ("CONNECT RESET");
   return 0;
}
/* end of program : LOBFILE.SQC */
```

COBOL サンプル: LOBFILE.SQB

```
 Identification Division.
 Program-ID. "lobfile".

 Data Division.
 Working-Storage Section.
     copy "sqlenv.cbl".
     copy "sql.cbl".
     copy "sqlca.cbl".

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.  1
 01 userid            pic x(8).
 01 passwd.
   49 passwd-length   pic s9(4) comp-5 value 0.
   49 passwd-name     pic x(18).
 01 resume            USAGE IS SQL TYPE IS CLOB-FILE.
 01 lobind            pic s9(4) comp-5.
     EXEC SQL END DECLARE SECTION END-EXEC.

 77 errloc          pic x(80).

 Procedure Division.
 Main Section.
     display "Sample COBOL program: LOBFILE".

* Get database connection information.
     display "Enter your user id (default none): "
          with no advancing.
     accept userid.

     if userid = spaces
       EXEC SQL CONNECT TO sample END-EXEC
     else
       display "Enter your password : " with no advancing
       accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR
* format with the length of the input string.
     inspect passwd-name tallying passwd-length for characters
        before initial " ".

     EXEC SQL CONNECT TO sample USER :userid USING :passwd
         END-EXEC.
     move "CONNECT TO" to errloc.
     call "checkerr" using SQLCA errloc.

     move "RESUME.TXT" to resume-NAME.                          2
     move 10 to resume-NAME-LENGTH.
     move SQL-FILE-OVERWRITE to resume-FILE-OPTIONS.

     EXEC SQL SELECT resume INTO :resume :lobind  3
              FROM emp_resume
              WHERE resume_format = 'ascii'
              AND empno = '000130' END-EXEC.
     if lobind less than 0 go to NULL-LOB-indicated.

     display "Resume for EMPNO 000130 is in file : RESUME.TXT".
     go to End-Main.

 NULL-LOB-indicated.
     display "NULL LOB indicated".

 End-Main.
     EXEC SQL CONNECT RESET END-EXEC.
```

```
    move "CONNECT RESET" to errloc.
    call "checkerr" using SQLCA errloc.
End-Prog.
          stop run.
```

例: CLOB 列へのデータの挿入

以下の C プログラム・セグメントのパス記述では、
- userid は 1 つのユーザーのディレクトリーを表します。
- dirname は 『userid』 のサブディレクトリーの名前を表します。
- filnam.1 は表に挿入したい部門の 1 つの名前になります。
- clobtab は CLOB データ・タイプをもっている表の名前です。

以下の例は、:hv_text_file で参照されている正規ファイルから CLOB 列にデータを挿入する方法を示しています。

```
strcpy(hv_text_file.name, "/home/userid/dirname/filnam.1");
hv_text_file.name_length = strlen("/home/userid/dirname/filnam.1");
hv_text_file.file_options = SQL_FILE_READ; /* this is a 'regular' file */

EXEC SQL INSERT INTO CLOBTAB
  VALUES(:hv_text_file);
```

LOB 列のレイアウトの表示

CL コマンド (たとえば、物理ファイル・メンバーの表示 (DSPPFM)) を使用して、LOB 列を収容している表の 1 つのデータ行が表示されているときには、その行に格納されている LOB データは表示されません。代わりに、データベースは、LOB 列の特殊値を表示します。この特殊値のレイアウトは以下のようになります。
- 13 ～ 28 バイトの16 進数ゼロ
- *POINTER で始まりブランクが続く 16 バイト

値の最初の部分のバイト数は、値の 2 番目の部分に 16 バイトの境界位置合わせを行うのに必要な数に設定されます。

たとえば、ColumnOne Char(10)、ColumnTwo CLOB(40K)、および ColumnThree BLOB(10M) という 3 つの列を収容している表があるとします。DSPPFM を出してこの表を表示すると、データのそれぞれの行は次のように表示されます。
- ColumnOne: 文字データで埋められた 10 バイト。
- ColumnTwo: 16 進数ゼロおよび '*POINTER' の 16 バイトからなる 22 バイト。
- ColumnThree: 16 進数ゼロおよび '*POINTER' の 16 バイトからなる 16 バイト。

このような方法で LOB 列を表示するコマンドの全セットを以下に示します。
- 物理ファイル・メンバーの表示 (DSPPFM)
- TOFILE キーワードに値 *PRINT が指定されている場合のファイル・コピー (CPYF)
- ジャーナル表示 (DSPJRN)
- ジャーナル・エントリーの検索 (RTVJRNE)
- ENTFMT キーワードに値 *TYPE1、*TYPE2、*TYPE3 および *TYPE4 が指定されている場合のジャーナル・エントリーの受け取り (RCVJRNE)

LOB 列のジャーナル・エントリーのレイアウト

以下の 2 つのコマンドは、ジャーナルされた LOB データへのアドレス可能度をユーザーに与えるバッファーを戻します。

- ENTFMT キーワードに値 *TYPEPTR が指定されている場合のジャーナル・エントリーの受け取り (RCVJRNE) CL コマンド。
- ジャーナル・エントリーの検索 (QjoRetrieveJournalEntries) API。

これらのエントリーの中の LOB 列のレイアウトは以下のようになります。

- 0 ～ 15 バイトの16 進数ゼロ
- '00'x に設定された 1 バイトのシステム情報
- ポインターでアドレス指定される LOB データの長さを収容する 4 バイト。
- 8 バイトの 16 進数ゼロ。
- ジャーナル・エントリーに格納されている LOB データへのポインターを収容する 16 バイト。

このレイアウトの最初の部分は、LOB データへのポインターに 16 バイトの境界位置合わせが行えることを目的にしています。このエリアのバイト数は、LOB 列の前にある列の長さによって異なります。この最初の部分の長さの計算方法の例については、LOB 列のレイアウトの表示に関する上記の節を参照してください。

LOB 列のジャーナル処理の詳細については、ジャーナル処理のトピックを参照してください。

ユーザー定義関数 (UDF)

ユーザー定義関数は、SQL に対してユーザー自身の拡張機能を作成するときに使用できるメカニズムです。 DB2 で提供される組み込み関数は便利な関数のセットですが、ユーザーの要件をすべて満たすとは限りません。 したがって、以下のような理由から SQL の関数を拡張したい場合があります。

- **カスタマイズ**

  ユーザーのアプリケーションに特定の関数が DB2 UDB に存在しない場合があります。その関数が単純な変換、普通の計算、あるいは複雑な多変量解析を行うものであっても、多くの場合、UDF を使用することができます。

- **柔軟性**

  DB2 UDB の組み込み関数では、ユーザーがアプリケーションに組み込みたい変量を扱えるとは限りません。

- **標準化**

  ユーザーのサイトで多くのプログラムが同じような基本的な関数のセットをインプリメントしている場合でも、その使用法にはこまかな相違点があります。 したがって、得られる結果に整合性があるとは限りません。 これらの関数を UDF を使用して一度正しく設定すれば、これらのプログラムですべて同じ使用法が SQL で直接適用でき、整合性のある結果が得られるようになります。

- **オブジェクト・リレーショナル・サポート**

234 ページの『ユーザー定義の特殊タイプ (UDT)』で説明するように、UDT は、DB2 UDB の機能を拡張しその安全性を向上させるのに非常に便利な方式です。 UDF は、動作を提供し、タイプをカプセル化することによって、UDT のための方式 として機能します。

さらに、SQL UDF は、ラージ・オブジェクトおよびデータ・リンクのタイプを操作するためのサポートを提供します。データベースには、これらのデータ・タイプを処理するのに便利な組み込み関数がいくつか用意されていますが、SQL UDF には、ユーザーがデータベースの機能をこの分野で (必要な専門化に至るまで) さらに操作し拡張するための方法が用意されています。

詳細については、以下のセクションを参照してください。

- 『UDF を使用する理由』
- 221 ページの『UDF の概念』
- 223 ページの『UDF の使用法』
- 224 ページの『UDF の登録』
- 224 ページの『保管と復元の考慮事項』
- 224 ページの『例: UDF の登録』
- 229 ページの『UDF の使用』

UDF を使用する理由

DB2 UDB アプリケーションを作成する際、必要な処置または操作をインプリメントするのに以下のことを選択することができます。

- UDF として
- アプリケーションのサブルーチンまたは関数として

新しい操作をアプリケーションのサブルーチンまたは関数として設定する方が一見簡単のように思えますが、以下のことをぜひ考慮してください。

- **再利用**

  新しい操作が、ご使用のサイトの他のユーザーまたはプログラムも利用できるようなものである場合には、 UDF を使用することにより、その再利用が可能になります。さらに、データベースのどのユーザーでも式を使用することができる場合には、この関数は SQL から直接呼び出すことができます。データベースは、この関数の引き数の各種のデータ・タイプのプロモーションを自動的に管理することができます。たとえば、DECIMAL から DOUBLE を使用すると、データベースによって、この関数が、異なっているが互換性のあるデータ・タイプに適用されます。

  新しい関数を通常の関数として設定するほうが簡単なように見えます。 (関数を DB2 UDB に定義する必要がありません。) このようにした場合は、関心のある他のアプリケーション開発者すべてにこの旨を通知し、開発者が効率よく使用できるように関数をパッケージ化する必要があります。しかし、このプロセスでは、データベースにアクセスするのに通常コマンド行プロセッサー (CLP) を使用するような対話式ユーザーを無視しています。また、プログラムの中だけで使用するために作成された関数は、関連したプログラムをもたない対話式ユーザーを無視します。これには、ODBC、JDBC などの多くのクライアントのほかに、STRSQL、STRQM、および RUNSQLSTM などのコマンドがあります。 CLP ユ

ーザーは、関数がデータベースの中の UDF でない限り、その関数を使用できません。また、このことは、SQL を使用する他のツール (Visualizer など) すべてに適用され、再コンパイルされることはありません。

- **パフォーマンス**

  あるケースでは、UDF をデータベース・エンジンから (アプリケーションからではなく) 直接呼び出すと、パフォーマンスがかなり向上する場合があります。 この利点は、さらに処理を行うためのデータの修飾に関数を使用できる場合に得られます。 このようなケースは、関数が行選択処理に使用されるときに発生します。

  あるデータを処理する単純なシナリオを考えてみます。関数 SELECTION_CRITERIA() として表されるある選択基準に適合しているとします。アプリケーションで、以下の選択ステートメントを出します。

  ```
  SELECT A, B, C FROM T
  ```

  おのおのの行を受け取ると、SELECTION_CRITERIA がデータに対して実行され、データをさらに処理する必要があるかどうかが決められます。 このステートメントでは、表 T のおのおのの行がアプリケーションに戻されなければなりません。しかし、SELECTION_CRITERIA() が UDF として設定された場合は、アプリケーションで次のステートメントを出すことができます。

  ```
  SELECT C FROM T WHERE SELECTION_CRITERIA(A,B)=1
  ```

  このケースでは、必要な行と列だけが、アプリケーションとデータベースとの間のインターフェースで受け渡されます。

  UDF によってパフォーマンスの利点が得られるもう 1 つのケースは、ラージ・オブジェクト (LOB) を処理する場合です。 LOB の 1 つのタイプの値からある情報を抜き出す関数があるとします。この抽出作業を直接データベース・サーバーで実行して、抽出された値だけをアプリケーションに戻すことができます。 これは、LOB 値全体をアプリケーションに戻し、それから抽出を行うよりも、はるかに効率のよい方法です。 この関数を UDF としてパッケージすることから得られるパフォーマンス上の価値は、特定の状態によっては、非常に大きなものがあります。 (LOB の一部は LOB ロケーターを使用しても取り出すことができることにご注意ください。同じようなシナリオの例については、212 ページの『標識変数および LOB ロケーター』を参照してください。)

- **オブジェクト・オリエンテーション**

  ユーザー定義の特殊タイプ (UDT) (*特殊タイプ* とも呼ばれる) の動作は、UDF を使用して設定することができます。 UDT の詳細については、234 ページの『ユーザー定義の特殊タイプ (UDT)』を参照してください。 UDT についての追加の詳細と、ここで説明するキャスタビリティー についての重要な概念については、SQL 解説書のCREATE DISTINCT TYPE ステートメントを参照してください。 特殊タイプを作成すると、特殊タイプとそのソース・タイプの間にキャスト関数が自動的に提供されます。 また、ソース・タイプによっては、比較演算子 (=、>、< など) が提供されます。 追加の動作は、ユーザー自身で提供する必要があります。 特殊タイプの動作はデータベースに入れておくと、特殊タイプのユーザーすべてがアクセスができるので、最も良い方法です。 したがって、ユーザーは、使用法のメカニズムとして UDF を使用することができます。

たとえば、1 メガバイトの BLOB に定義された BOAT という特殊タイプをもっているとします。 CREATE ステートメントを入力します。

```
CREATE DISTINCT TYPE BOAT AS BLOB(1M)
```

BLOB には船舶のさまざまな仕様といくつかの図が入っています。 船舶のサイズを比較したいとします。しかし、BLOB のソース・タイプに特殊タイプが定義されているので、比較演算は自動的には生成されません。 BOAT_COMPARE 関数を作成して、ユーザーが指定した計量法にもとづいて 2 つの船舶を比較し、どちらが大きいかを判断します。 これらは、BOAT オブジェクトにもとづいて、変位、全体の長さ、メートル法のトン数、その他の計算になります。
BOAT_COMPARE 関数を以下のように作成します。

```
CREATE SQL FUNCTION BOAT_COMPARE (BOAT, BOAT) RETURNS INTEGER ...
```

関数は以下の数を戻します。

- 1 一番目のほうが大きい。
- 2 二番目のほうが大きい。
- 0 2 つが等しい。

この関数を SQL で使用して船舶を比較します。以下の表を作成するとします。

```
CREATE TABLE BOATS_INVENTORY (
  BOAT_ID        CHAR(5),
  BOAT_TYPE      VARCHAR(25),
  DESIGNER       VARCHAR(40),
  OWNER          VARCHAR(40),
  DESIGN_DATE    DATE,
  SPEC           BOAT,
  ...                 )

CREATE TABLE MY_BOATS (
  BOAT_ID        CHAR(5),
  BOAT_TYPE      VARCHAR(25),
  DESIGNER       VARCHAR(40),
  DESIGN_DATE    DATE,
  ACQUIRE_DATE   DATE,
  ACQUIRE_PRICE  CANADIAN_DOLLAR,
  CURR_APPRAISL  CANADIAN_DOLLAR,
  SPEC           BOAT,
  ...                 )
```

以下の SQL SELECT ステートメントを実行することができます。

```
SELECT INV.BOAT_ID, INV.BOAT_TYPE, INV.DESIGNER,
  INV.OWNER, INV.DESIGN_DATE
  FROM BOATS_INVENTORY INV, MY_BOATS MY
  WHERE MY.BOAT_ID = '19GCC'
  AND BOAT_COMPARE(INV.SPEC, MY.SPEC) = 1
  AND INV.DESIGNER = MY.DESIGNER
```

この単純な例では、BOATS_INVENTORY から、MY_BOATS の中の特定の船舶よりも大きい、同じ設計者による船舶がすべて戻されます。この例では、必要な行だけがアプリケーションに戻されることにご注意ください。これは、比較がデータベース・サーバーの中で行われるからです。 実際、データ・タイプ BOAT のいかなる数値も受け渡しされません。 BOAT が 1 メガバイトの BLOB データ・タイプにもとづいているため、記憶域の使用効率とパフォーマンスは大きく向上します。

UDF の概念

以下に、UDF をコーディングする前に理解しておく必要がある重要な概念について説明します。

関数名

- 関数のフル名。

  *SQL 命名を使用した関数のフル名は、<schema-name>.<function-name> になります。

  *SYS 命名での関数のフル名は、<schema-name>/<function-name> になります。DML ステートメントでは、関数名は *SYS 命名を使用して修飾できません。

  以下のフル名は、関数を参照しているところであればどこでも使用できます。たとえば、次の通りです。

  ```
  QGPL.SNOWBLOWER_SIZE    SMITH.FOO    QSYS2.SUBSTR    QSYS2.FLOOR
  ```

  <schema-name>. は省略することもできます。その場合は、ユーザーが参照している関数を DB2 UDB が判別しなければなりません。たとえば、次の通りです。

  ```
  SNOWBLOWER_SIZE    FOO    SUBSTR    FLOOR
  ```

- パス

  パス は、schema-name が指定されないときに発生する*修飾なしの* 参照を DB2 UDB が解決するのに重要な主要概念です。 関数を参照する DDL ステートメントでのパスの使用については、SQL 解説書の中の該当するCREATE FUNCTION ステートメントの説明を参照してください。パスはスキーマ名の順序リストです。 パスは、UDF および UDT への修飾なしの参照を解決するためのスキーマのセットを提供します。 ある関数参照がパス内の複数のスキーマにある関数に一致する場合は、この一致を解決するために、パス内のスキーマの順序が使用されます。 パスは、静的 SQL の事前コンパイル・コマンドおよびバインド・コマンドの SQLPATH オプションを使用して設定されます。 動的 SQL の場合は、パスは SET PATH ステートメントによって設定されます。 活動化グループ内で実行される最初の SQL ステートメントが SQL 命名を使用して実行される場合は、パスには以下のような省略時値が入っています。

  ```
  "QSYS","QSYS2","<ID>"
  ```

  この省略時値は静的 SQL および動的 SQL の両方に適用されます。ここで、<ID> は、現行ステートメントの権限 ID を表します。

  活動化グループ内の最初の SQL ステートメントがシステム命名を使用して実行される場合は、省略時のパスは *LIBL になります。

- 多重定義関数名

  関数名は *多重定義* であっても構いません。多重定義は、複数の関数 (同一のスキーマにあるものでも構いません) が同じ名前をもつことができることを意味します。 ただし、2 つの関数が同じシグニチャー をもつことはできません。関数シグニチャーは、すべての関数パラメーターの定義されたデータ・タイプと (定義された順序で) 連結された、修飾された関数名として定義することができます。 多重定義関数の例については、 226 ページの『例: BLOB 文字列検索』を参照してください。 シグニチャーと関数解決についての詳細は、SQL 解説書の関数のトピックを参照してください。

- 関数解決

  多重定義と関数パスを考慮に入れて、おのおのの関数参照 (修飾されたまたは修飾なしの参照にかかわらず) に対して*最適な適合* を選択するのは *関数解決アルゴリズム* の役割です。関数はすべて (組み込み関数も)、関数選択アルゴリズムを介して処理されます。関数解決アルゴリズムは、関数のタイプは考慮しません。このため、参照の使用がスカラー関数を必要としているにもかかわらず、表関数が*最適な適合* 関数として解決される（あるいはその逆) ことがあります。

  パスの概念、SET PATH ステートメント、および、関数解決アルゴリズムの詳細については、SQL 解説書を参照してください。 SQLPATH 事前コンパイル・オプションの説明は、コマンドの付録にあります。

- 関数のタイプ

  関数には、いくつかのタイプがあります。

  – *組み込み*。 組み込み関数は、データベースと一緒に提供され出荷される関数です。SUBSTR() は 1 つの例です。

  – *システム生成*。 DISTINCT TYPE が作成されるときにデータベース・エンジンによって暗黙的に生成される関数です。これらの関数は、DISTINCT TYPE とその基本タイプの間のキャスト操作を提供します。

  – *ユーザー定義*。 ユーザーによって作成され、データベースに登録される関数です。

  さらに、おのおのの関数は、*スカラー関数*、*列関数*、または*表関数* に分類できます。

  スカラー関数 は、呼び出されるたびに、単一値の応答を戻します。 たとえば、組み込み関数 SUBSTR() は、スカラー関数です。ほとんどの組み込み関数はスカラー関数です。 システム生成関数はすべてスカラー関数です。 スカラー UDF は、外部コード (C などのプログラム言語でコーディングする、または、SQL 関数として SQL でコーディングする) にすることも、ソース・コード (既存関数の設定を使用) にすることもできます。

  列関数 は、類似値のセット (データの列) を受け取り、この値のセットから、単一値の応答を戻します。 これは、DB2 UDB では、*集合関数* とも呼ばれます。組み込み関数のいくつかは、列関数です。 列関数の例に、組み込み関数 AVG() があります。 外部 UDF は、列関数として定義することはできません。 ただし、UDF が組み込み列関数のいずれかのソースになっている場合には、その UDF は列関数として定義できます。 特殊タイプの場合には、後者の定義が便利です。 たとえば、特殊タイプ SHOESIZE が、基本タイプ INTEGER で定義されて存在している場合、UDF AVG(SHOESIZE) を、既存の組み込み列関数 AVG(INTEGER) のソースになっている列関数として定義することができます。

  表関数 は、その関数を参照する SQL ステートメントに表を戻します。表関数は、SELECT の FROM 文節で参照される必要があります。表関数は、SQL 言語のデータ処理能力を DB2 以外のデータに適用したり、あるいは DB2 以外のデータを DB2 の表に変換するために、使用できます。たとえば、あるファイルを入手して表に変換したり、ワールド・ワイド・ウェブ (WWW) からのデータをサンプルとして入手して作表したり、あるいは Lotus Notes データベースにアクセスしてメール・メッセージについての情報 (日付、送信元、およびメッセージ本文など) を戻したりすることができます。これらの情報を、データベース内の他

| の表と結合することができます。表関数は、外部関数または SQL 関数として定
| 義することができますが、ソース関数として定義することはできません。

UDF の使用法

UDF には、ソース、外部、および SQL という 3 つのタイプがあります。それぞれのタイプの使用法はかなり異なります。

- ソース UDF。データベースに単に登録されている関数で、関数自身が別の関数を参照します。これらの関数が、実際に、ソースになっている関数にマップします。したがって、これらの関数を設定するには、CREATE FUNCTION ステートメントを使用してデータベースに登録する以外になにも行う必要はありません。
- 外部関数。これは、高水準言語 (C、COBOL、RPG など) で作成されたプログラムおよびサービス・プログラムへの参照です。関数がデータベースに登録されると、関数が DML ステートメントで参照されるたびに、データベースがプログラムまたはサービス・プログラムを呼び出すことができます。したがって、外部 UDF では、UDF の作成者は、高水準言語およびコーディング方法だけでなく、プログラムとデータベースの間のインターフェースを理解してください。 外部関数の作成の詳細については、 251 ページの『第 13 章 ユーザー定義関数 (UDF) の作成』を参照してください。
- SQL UDF。SQL UDF は、SQL 言語だけで作成されている関数です。SQL UDF の 'コード' は、実際には、CREATE FUNCTION ステートメント自体の中に組み込まれた SQL ステートメントです。SQL UDF には以下のような利点があります。
  - SQL で作成されているため、可搬性があります。
  - データベースと関数の間のインターフェースの定義は SQL 宣言で行うことができ、パラメーターの実際の受け渡しの詳細について心配する必要がありません。
  - ラージ・オブジェクト、データ・リンク、および UDT をパラメーターとして受け渡し、その後の操作は、関数の中で行うことができます。SQL 関数の詳細については、 251 ページの『第 13 章 ユーザー定義関数 (UDF) の作成』を参照してください。

UDF を使用するには、次のステップを実行します。

1. DB2 UDB への UDF の登録。どのタイプの UDF が作成されているかにかかわらず、UDF はすべて、CREATE FUNCTION ステートメントを使用して、データベースに登録する必要があります。ソース関数の場合は、データベースに関数を定義するのに必要な作業がすべて、この登録ステップによって行われます。SQL UDF の場合は、CREATE FUNCTION ステートメントに関数を定義するのに必要なものがすべて入っています。ただし、CREATE ステートメントの構文はもっと複雑です (SQL の実際の実行可能コードが含まれます)。外部 UDF の場合は、CREATE FUNCTION ステートメント**だけ**が、関数をデータベースに登録できます。関数を実際にインプリメントするサポート・コードは、別個に作成する必要があります。詳しくは、 224 ページの『UDF の登録』を参照してください。
2. UDF のデバッグ。 251 ページの『第 13 章 ユーザー定義関数 (UDF) の作成』を参照してください。

これらのステップが正常に完了すると、データ操作言語 (DML) ステートメントまたはデータ定義言語 (DDL) ステートメントで UDF (CREATE VIEW など) を使用できるようになります。

UDF の登録

UDF は、データベースによって関数が認識され使用される前に、データベースに登録しておく必要があります。UDF の登録は、CREATE FUNCTION ステートメントを使用して行います。

このステートメントを使用して、DETERMINISTIC、ALLOW PARALLEL、および RETURNS NULL ON NULL INPUT などのオプションと一緒に、プログラムの言語と名前を指定します。これらのオプションは、関数の意図およびデータベースへの呼び出しを最適化する方法を、データベースに対して具体的に指定するのに役立ちます。

DB2 UDB への外部 UDF の登録は、実際のコードを作成し、十分にテストをしてから行ってください。 実際に作成する前に UDF を定義することも可能です。 しかし、UDF を実行したときの問題を避けるためには、UDF を作成し、十分にテストしてから登録を行うことをお勧めします。 UDF のテストの説明については、251 ページの『第 13 章 ユーザー定義関数 (UDF) の作成』を参照してください。

UDF の登録の例については、『例: UDF の登録』を参照してください。

保管と復元の考慮事項

ILE 外部プログラムまたはサービス・プログラムに関連した外部関数が作成される時、関連したプログラム・オブジェクトまたはサービス・プログラム・オブジェクト内に、関連した関数の属性を保管しようとする試みがなされます。 *PGM オブジェクトまたは *SRVPGM オブジェクトが保管されて同じシステムまたは他のシステムに復元される場合は、これらの情報を用いてカタログが自動的に更新されます。関数の属性が保管できなかった場合、カタログは自動的には更新されず、ユーザーは新しいシステム上で外部関数を作成する必要があります。外部関数の属性は、次の制約に従って保管することができます。

- 外部プログラム・ライブラリーは QSYS または QSYS2 であってはならない。
- 外部プログラムは、CREATE FUNCTION ステートメントが出される時点で存在していなければならない。
- 外部プログラムは、ILE *PGM あるいはまたは *SRVPGM あるいはでなければならない。
- 外部プログラムやサービス・プログラムには、少なくとも 1 つの SQL ステートメントが含まれていなければならない。

オブジェクトが更新できなかった場合でも、関数は作成されます。

例: UDF の登録

以下の例では、UDF を登録するときの、代表的なさまざまな状態について説明します。 例には以下のものがあります。

- 例: 指数
- 例: 文字列検索
- 例: UDT での文字列検索

- 例: UDT パラメーターを使用した外部関数
- 例: UDT での AVG
- 例: カウンティング
- 例: 文書 ID を戻す表関数

注: コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

例: 指数

浮動小数点値の指数を求める外部 UDF を作成し、その UDF を MATH スキーマに登録したいとします。

```
CREATE FUNCTION MATH.EXPON (DOUBLE, DOUBLE)
  RETURNS DOUBLE
  EXTERNAL NAME 'MYLIB/MYPGM(MYENTRY)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  RETURNS NULL ON NULL INPUT
  ALLOW PARALLEL
```

この例では、システムは、RETURNS NULL ON NULL INPUT という省略時値を使用します。どちらかの引き数がヌルの場合に、結果値をヌルにする必要があるので、これはお勧めできる方法です。 ユーザーはスクラッチパッドを必要とせず、最終呼び出しも不要なので、NO SCRATCHPAD および NO FINAL CALL 省略時値が使用されます。 EXPON が並列であってはならない理由はないので、ALLOW PARALLEL 値が指定されています。

例: 文字列検索

同僚の Willie が、ある短い文字列 (引き数として渡される) が、ある CLOB 値 (これも引き数として渡される) の中にあるかどうかを調べる UDF を作成したとします。 UDF は、文字列があった場合には、その文字列の CLOB 内の位置を戻し、なかった場合はゼロを戻します。

さらに、Willie は、浮動小数点 (FLOAT) の結果を戻す関数を作成したとします。しかし、この関数が SQL で使用された場合は、必ず整数 (INTEGER) を戻さなければならないとします。 以下の関数を作成するとします。

```
CREATE FUNCTION FINDSTRING (CLOB(500K), VARCHAR(200))
  RETURNS INTEGER
  CAST FROM FLOAT
  SPECIFIC "willie_find_feb95"
  EXTERNAL NAME 'MYLIB/MYPGM(FINDSTR)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  RETURNS NULL ON NULL INPUT
```

UDF 本体が実際には FLOAT 値を戻すことを指定するために CAST FROM 文節が使用されていますが、UDF を使用したステートメントにこの値を戻す前に、この値を INTEGER にキャストしていることにご注意ください。 SQL 解説書に説明があるように、INTEGER 組み込み関数は、このキャストをユーザーに代わって実行します。また、この関数にユーザーの特定の名前を付け、後で、これを DDL で参照し

たいとします (『例: UDT での文字列検索』を参照)。 この UDF はヌル値を処理するようには作成されていないので、RETURNS NULL ON NULL INPUT を使用します。また、スクラッチパッドがないので、NO SCRATCHPAD および NO FINAL CALL 省略時値を使用します。 FINDSTRING が並列であってはならない理由はないので、ALLOW PARALLEL 省略時値が使用されています。

例: BLOB 文字列検索

この関数を BLOB でも CLOB でもはたらくようにしたいので、BLOB を最初のパラメーターに指定してもう 1 つの FINDSTRING を定義します。

```
CREATE FUNCTION FINDSTRING (BLOB(500K), VARCHAR(200))
  RETURNS INTEGER
  CAST FROM FLOAT
  SPECIFIC "willie_fblob_feb95"
  EXTERNAL NAME 'MYLIB/MYPGM(FINDSTR)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
```

この例は UDF 名の多重定義を示しており、複数の UDF が同一の本体を共用できることが示されています。 BLOB は CLOB に割り当てられないが、同じソース・コードを使用することはできます。上の例では、BLOB と CLOB に対するプログラミング・インターフェースと、DB2 UDB と UDF の間のプログラミング・インターフェースが同じなので、プログラミング上の問題はありません。長さの後にデータが続きます。 DB2 は、ある関数本体を使用する UDF が、同じ本体を使用する他の UDF と整合性があるかどうかはチェックしません。

例: UDT での文字列検索

この例は、前の例からの続きです。『例: BLOB 文字列検索』の FINDSTRING 関数の目的を達したあと、次に、ソース・タイプ BLOB を使用して特殊タイプ BOAT を定義したとします。また、FINDSTRING がデータ・タイプ BOAT をもつ値でもはたらくようにしたいので、もう 1 つの FINDSTRING 関数を作成します。この関数は、『例: BLOB 文字列検索』の中の BLOB 値に対してはたらく FINDSTRING の上にソース化されます。次の例の、FINDSTRING のなお一層の多重定義にご注意ください。

```
CREATE FUNCTION FINDSTRING (BOAT, VARCHAR(200))
  RETURNS INT
  SPECIFIC "slick_fboat_mar95"
  SOURCE SPECIFIC "willie_fblob_feb95"
```

この FINDSTRING 関数は 『例: BLOB 文字列検索』 の FINDSTRING 関数とは異なるシグニチャーをもっているので、名前が多重定義になる問題はありません。ユーザー自身の特定の名前を付けて、後で、DDL で参照できるようにします。SOURCE 文節を使用しているので、EXTERNAL NAME 文節、または、関数属性を指定する関連キーワードのいずれかを使用することはできません。これらの属性は、ソース関数からとられます。最後に、ソース関数を指定するときに、『例: BLOB 文字列検索』の中で明示的に使用されている特定の関数名を使用するようにしてください。 これは修飾なしの参照であるので、このソース関数が常駐するスキーマは関数パスの中になければならず、関数パスにない場合は、参照は解決されません。

例: UDT パラメーターを使用した外部関数

1 つの BOAT を取り出し、その設計属性を調べ、カナダ・ドルでその船舶のコストを計算するもう 1 つの UDF を作成したとします。内部では、労務費がドイツ・マルク、日本円、または米国ドルで表されていても、この関数では、船舶を作るコストを必要な通貨 (カナダ・ドル) で計算することが必要です。 これは、現行の為替レート情報を exchange_rate ファイル (DB2 UDB の外で管理されている) から入手しなければならないことを意味し、したがって、その応答は、このファイルの内容によって異なります。このため、この関数は NOT DETERMINISTIC になります。

```
CREATE FUNCTION BOAT_COST (BOAT)
  RETURNS INTEGER
  EXTERNAL NAME 'MYLIB/COSTS(BOATCOST)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC
  NO EXTERNAL ACTION
```

CAST FROM および SPECIFIC は指定されていないこと、しかし、NOT DETERMINISTIC は指定されていることにご注意ください。

例: UDT での AVG

この例は、CANADIAN_DOLLAR 特殊タイプについて AVG 列関数をインプリメントします。 CANADIAN_DOLLAR の定義については、236 ページの『例: 通貨』を参照してください。固いタイプを指定すると、組み込まれた AVG 関数を特殊タイプで使用することができなくなります。 CANADIAN_DOLLAR のソース・タイプが DECIMAL であることがわかったので、CANADIAN_DOLLAR を AVG(DECIMAL) 組み込み関数の上にソース化することによって AVG をインプリメントします。これが行えるのは、DECIMAL から CANADIAN_DOLLAR に、また、その逆の方向にキャストできるかどうかによって決まりますが、DECIMAL が CANADIAN_DOLLAR のソース・タイプであるので、このキャストがはたらくことがわかります。

```
CREATE FUNCTION AVG (CANADIAN_DOLLAR)
  RETURNS CANADIAN_DOLLAR
  SOURCE "QSYS2".AVG(DECIMAL(9,2))
```

他の AVG 関数が SQL パスに入り込んでくる可能性もあるので、SOURCE 文節で関数名が修飾されていることにご注意ください。

例: カウンティング

次の単純なカウンティング関数では、一回目に 1 を戻し、呼び出されるたびに結果を 1 ずつ増分します。この関数は SQL 引き数をとらず、その応答が呼び出しごとに変わるので、定義上、NOT DETERMINISTIC 関数です。戻された最後の値を保管するために SCRATCHPAD が使用されます。この関数が呼び出されるたびにこの値が増分され、その結果が戻されます。

```
CREATE FUNCTION COUNTER ()
  RETURNS INT
  EXTERNAL NAME 'MYLIB/MYFUNCS(CTR)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
```

```
NOT DETERMINISTIC
NOT FENCED
SCRATCHPAD 4
DISALLOW PARALLEL
```

パラメーター定義は指定されておらず、空の括弧があるだけです。上の関数では SCRATCHPAD が指定されており、NO FINAL CALL という省略時指定が使用されています。 このケースでは、スクラッチパッドのサイズが 4 バイト (カウンターでは十分なサイズ) に設定されています。COUNTER 関数では 1 つのスクラッチパッドを使用して正しく動作する必要があるため、DISALLOW PARALLEL を追加して、DB2 UDB が並列に動かないようにしています。

例: 文書 ID を戻す表関数

テキスト管理システムにおいて、所定のサブジェクト・エリア (1 番目のパラメーター) と一致し、かつ所定のストリング (2 番目のパラメーター) を含む既知の文書ごとに、単数の文書 ID 列で構成される行を戻す、新しい表関数を作成したとします。この UDF は、次のようにテキスト管理システムの関数を使用して迅速に文書を識別します。

```
CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
  RETURNS TABLE (DOC_ID CHAR(16))
  EXTERNAL NAME 'DOCFUNCS/UDFMATCH(udfmatch)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  NOT FENCED
  SCRATCHPAD
  NO FINAL CALL
  DISALLOW PARALLEL
  CARDINALITY 20
```

単一セッションのコンテキストの中では、この関数は常に同じ表を戻すため、DETERMINISTIC として定義されます。 DOCMATCH からの出力を定義する RETURNS 文節 (列名 DOC_ID を含む) に注意してください。それぞれの表関数ごとに FINAL CALL を指定する必要はありません。さらに、表関数が並列に動作できないように DISALLOW PARALLEL キーワードが追加されます。 DOCMATCH からの出力のサイズが変化の大きい変数であっても、CARDINALITY 20 が代表値であり、DB2 最適化プログラムが良い判断を行える援助として指定されます。

一般に、この表関数は、次のように文書テキストを含む表の結合において使用されます。

```
SELECT T.AUTHOR, T.DOCTEXT
  FROM DOCS AS T, TABLE(DOCMATCH('MATHEMATICS', 'ZORN''S LEMMA')) AS F
  WHERE T.DOCID = F.DOC_ID
```

FROM 文節の、表関数を指定する特殊構文 (TABLE キーワード) に注意してください。この呼び出しにおいて、 DOCMATCH() 表関数は、ZORN'S LEMMA を参照する MATHEMATICS 文書ごとに 1 つの列 DOC_ID を含む行を戻します。 DOC_ID 値が原本表に結合され、著者名と文書テキストを検索します。

UDF の使用

スカラーおよび列 UDF は、式が有効であれば SQL ステートメントのどこからでも呼び出すことができます。表 UDF は、SELECT の FROM 文節で呼び出すことができます。ただし、UDF の使用には、いくつかの制約事項があります。

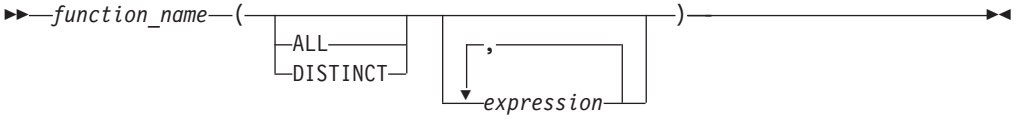
- UDF とシステム生成関数は検査制約に指定することはできません。 また、検査制約には、組み込み関数 DLVALUE、DLURLPATH、DLURLPATHONLY、DLURLSCHEME、DLURLCOMPLETE、DLURLSERVER、ATAN2、DIFFERENCE、RADIANS、RAND、SOUNDEX、NOW、CURDATE、および CURTIME への参照を含めることはできません。
- 外部 UDF、SQL UDF、および組み込み関数 DLVALUE、DLURLPATH、DLURLPATHONLY、DLURLSCHEME、DLURLCOMPLETE、および DLURLSERVER は、ORDER BY または GROUP BY 文節で参照することはできません。ただし、SQL ステートメントが読み取り専用で、一時処理 (ALWCPYDTA(*YES) または (*OPTIMIZE)) が行える場合を除きます。

SQL 解説書には、以下のコンテキストの詳細が記載されています。このセクションで使用されている説明と例は、比較的単純な SELECT ステートメントのコンテキストに焦点を置いていますが、この使用方法はこれらのコンテキストに限られていないことにご注意ください。

パス および *関数解決* アルゴリズムの使用方法と重要性については、221 ページの『UDF の概念』を参照してください。この両方の概念の詳細については、SQL 解説書を参照してください。 関数へのデータ操作言語 (DML) 参照の解決には関数解決アルゴリズムが使用され、したがって、この参照がどのようにはたらくかを理解しておくことが重要になります。

関数への参照

関数への参照は、UDF または組み込み関数にかかわらず、以下の構文をもっています。



上記の場合、function_name は、修飾なしの関数名でも修飾された関数名でも構いません。 *SYS 命名規則を使用する場合は、関数は修飾できないことにご注意ください。引き数の数は 0 ～ 90、および、以下のものを持つ式です。

- 列名 (修飾されたもの、または修飾なし)
- 定数
- 式
- 関数
- ホスト変数
- CAST 関数をもつパラメーター・マーカー
- スカラー副選択
- 特殊レジスター

引き数の位置は重要で、意味体系が正しくなるためには、関数定義に従う必要があります。 引き数の位置および関数定義の両方とも関数本体に従う必要があります。

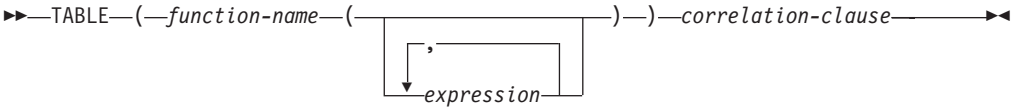
DB2 UDB は、関数定義によりよく一致するように引き数を組み替えたり、また、個々の関数パラメーターの意味体系を判別することはしません。

関数呼び出しの例

関数呼び出しの有効な例を以下に示します。

```
AVG(FLOAT_COLUMN)
BLOOP(COLUMN1)
BLOOP(FLOAT_COLUMN + CAST(? AS INTEGER))
BLOOP(:hostvar :indicvar)
BRIAN.PARSE(CONCAT(CHAR_COLUMN,USER), 1, 0, 0, 1)
CTR()
FLOOR(FLOAT_COLUMN)
PABLO.BLOOP(A+B)
PABLO.BLOOP(:hostvar)
"search_schema"(CURRENT PATH, 'GENE')
SUBSTR(COLUMN2,8,3)
QSYS2.FLOOR(AVG(EMP.SALARY))
QSYS2.AVG(QSYS2.FLOOR(EMP.SALARY))
QSYS2.SUBSTR(COLUMN2,11,LENGTH(COLUMN3))
```

表関数への参照



上記の図において、function_name は、修飾なしの関数名でも修飾された関数名でも構いません。ユーザー定義表関数は、*SYS 命名規則と *SQL 命名規則のいずれかを使用している場合、修飾することができます。

表関数呼び出しの有効な例を以下に示します。

```
TABLE(TABFUNC()) X
TABLE(BLOOP_TAB(:hostvar, COL1+COL2))
NEW_TABLE TABLE(SCHEMA1.BLOOP_TAB2()) X
```

関数でのパラメーター・マーカーまたは NULL の使用

重要な制約事項にパラメーター・マーカーとヌル値があります。以下のようなコーディングは行えません。

```
BLOOP(?)
```

あるいは

```
BLOOP(NULL)
```

関数解決では、引き数がどのようなデータ・タイプになるかがわからないため、参照を解決することができません。 CAST 指定を使用して、関数解決が使用できる、パラメーター・マーカーまたはヌル値のあるタイプ (たとえば、INTEGER) を指定できます。たとえば、以下のようにします。

```
BLOOP(CAST(? AS INTEGER))
```

あるいは

```
BLOOP(CAST(NULL AS INTEGER))
```

修飾された関数参照の使用

修飾された関数参照を使用する場合は、一致する関数を探す DB2 UDB の検索をそのスキーマに制限します。たとえば、以下のステートメントがあるとします。

```
SELECT PABLO.BLOOP(COLUMN1) FROM T
```

スキーマ PABLO の中の BLOOP 関数だけが考慮されます。ユーザー SERGE が BLOOP 関数を定義していても、組み込み関数 BLOOP があってもなくても構いません。次に、ユーザー PABLO が、2 つの BLOOP 関数を彼のスキーマに定義したとします。

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS ...
CREATE FUNCTION BLOOP (DOUBLE) RETURNS ...
```

BLOOP は PABLO スキーマ内で多重定義され、関数選択アルゴリズムは、引き数 column1 のデータ・タイプによって、最良の BLOOP を選択します。このケースでは、PABLO.BLOOP の両方とも数値の引き数をとり、column1 が数値タイプのいずれでもない場合は、ステートメントは失敗します。他方、column1 が SMALLINT または INTEGER のいずれかである場合は、関数選択は最初の BLOOP に解決され、column1 が DECIMAL または DOUBLE のいずれかである場合は、2 番目の BLOOP が選択されます。

この例について、以下に、いくつかの点を述べます。

1. この例は引き数のプロモーションを示しています。最初の BLOOP は INTEGER パラメーターで定義されていますが、これを SMALLINT 引き数に渡すことができます。関数選択アルゴリズムは、組み込みデータ・タイプの間のプロモーションをサポートしており (詳細は、SQL 解説書を参照)、DB2 UDB は該当するデータ値変換を実行します。
2. なんらかの理由で 2 番目の BLOOP を SMALLINT または INTEGER 引き数を使用して呼び出す必要がある場合は、ステートメントの中で、以下のように、明示的に処置をとる必要があります。

   ```
   SELECT PABLO.BLOOP(DOUBLE(COLUMN1)) FROM T
   ```

3. この代わりに、1 番目の BLOOP を DECIMAL または DOUBLE 引き数を使用して呼び出す必要がある場合は、意図に応じて、以下の明示的な処置のどちらかをとることができます。

   ```
   SELECT PABLO.BLOOP(INTEGER(COLUMN1)) FROM T
   SELECT PABLO.BLOOP(FLOOR(COLUMN1)) FROM T
   ```

   他の関数については、SQL 解説書を参照して調べてください。INTEGER 関数は、QSYS2 スキーマの組み込み関数です。

修飾なしの関数参照の使用

修飾された関数参照の代わりに修飾なしの関数参照を使用する場合は、一致する関数を探す DB2 の検索が、通常、関数パスを使用して参照を修飾します。DROP FUNCTION または COMMENT ON FUNCTION 関数のケースでは、これらの関数が *SQL 命名用に、または *SYS 命名のための *LIBL 用に修飾されていない場合は、参照は現行の権限 ID を使用して修飾されます。したがって、*関数パスが何であり、現行の関数パスのスキーマに対立する関数があればそれが何であるかを理解しておくことが重要になります。* たとえば、ユーザーが PABLO で、以下のような静的 SQL ステートメントがあるとします (ここで、COLUMN1 はデータ・タイプ INTEGER です)。

```
SELECT BLOOP(COLUMN1) FROM T
```

231 ページの『修飾された関数参照の使用』にあるように、2 つの BLOOP 関数が作成されており、そのうちの 1 つを選択したいとします。以下の省略時関数パスが使用されている場合、QSYS または QSYS2 に対立する BLOOP がなければ、1 番目の BLOOP が選択されます (column1 が INTEGER であるため)。

```
"QSYS","QSYS2","PABLO"
```

しかし、以前に別の目的のために書いたスクリプトを、いま事前コンパイルとバインディングのために使っていることを忘れたとします。このスクリプトでは、現行の作業には関係のない別の目的で以下に示す関数パスを指定するために、明示的に SQLPATH パラメーターがコーディングされています。

```
"KATHY","QSYS","QSYS2","PABLO"
```

Kathy が自分自身の目的のために BLOOP 関数を作成している場合は、関数選択は適切に Kathy の関数に解決し、ユーザーのステートメントはエラーなしで実行されます。DB2 UDB は、ユーザーが正しく作業を行っていると想定しているので、通知は出されません。ユーザーのステートメントからの間違った出力を識別し、必要な訂正を行うのはユーザーの責任になります。

関数参照の要約

修飾された関数参照および修飾なしの関数参照について、関数選択アルゴリズムは、以下の内容をもつ、適用できる関数 (組み込み関数とユーザー定義関数の両方) をすべて調べます。

- 名前
- 関数参照にある引き数と同数の定義済みパラメーター
- 対応する引き数と同一の、または、対応する引き数からプロモーションできる各パラメーター

(*適用できる関数* は、修飾された参照用の*名前付きスキーマにある関数* または、修飾なしの参照用の*関数パスのスキーマにある関数* を意味します。) アルゴリズムは、正確な一致を探し、正確な一致がない場合には、これらの関数の中での最適な一致を探します。修飾なしの参照のケースに限り、現行関数パスが、異なるスキーマに 2 つの同じ一致があった場合の決め手の要素として使用されます。アルゴリズムの詳細については、SQL 解説書を参照してください。

231 ページの『修飾された関数参照の使用』 の最後にある例で示されている興味深い特徴は、同一関数への参照も含めて、*関数参照はネストすることができるという事実* です。これは、組み込み関数にも UDF にも一般的にあてはまります。しかし、列関数が関係する場合には、いくつかの制約事項があります。

先の例を書き直すと、以下のようになります。

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS INTEGER ...
CREATE FUNCTION BLOOP (DOUBLE) RETURNS INTEGER ...
```

次に、以下の DML ステートメントを考えてみます。

```
SELECT BLOOP( BLOOP(COLUMN1)) FROM T
```

column1 が DECIMAL または DOUBLE 列である場合、内側の BLOOP 参照は、上に定義された 2 番目の BLOOP にまで解決されます。この BLOOP は INTEGER を戻すので、外側の BLOOP は 1 番目の BLOOP にまで解決されます。

また、column1 が SMALLINT または INTEGER 列である場合、内側の BLOOP 参照は、上に定義された 1 番目の BLOOP にまで解決されます。この BLOOP は INTEGERを戻すので、外側の BLOOP も 1 番目の BLOOP にまで解決されます。この場合、同じ関数に対してネストされた参照になります。

以下に、関数参照の重要な追加点について説明します。

- SQL 演算子のいずれかの名前を使用して関数を定義することができます。たとえば、特殊タイプ BOAT をもつ値の "+" 演算子にある意味をもたせたいとします。以下のような UDF を定義できます。

  ```
  CREATE FUNCTION "+" (BOAT, BOAT) RETURNS ...
  ```

  次に、以下の SQL ステートメントを書くことができます。

  ```
  SELECT "+"(BOAT_COL1, BOAT_COL2)
  FROM BIG_BOATS
  WHERE BOAT_OWNER = 'Nelson Mattos'
  ```

  ただし、組み込み条件演算子 (>、=、LIKE、IN など) は、この方法で多重定義できないことにご注意ください。

- 関数選択アルゴリズムは、特定の関数に解決するときに、参照のコンテキストを考慮しません。以下の BLOOP 関数 (前の例を少し変更しています) をご覧ください。

  ```
  CREATE FUNCTION BLOOP (INTEGER) RETURNS INTEGER ...
  CREATE FUNCTION BLOOP (DOUBLE) RETURNS CHAR(10)...
  ```

  次に、以下の SELECT ステートメントを書いたとします。

  ```
  SELECT 'ABCDEFG' CONCAT BLOOP(SMALLINT_COL) FROM T
  ```

  最適な一致 (SMALLINT 引き数を使用して解決される) が上で定義されている 1 番目の BLOOP なので、CONCAT の 2 番目のオペランドはデータ・タイプ INTEGER に解決されます。 CONCAT がストリングの引き数を要求しているので、このステートメントは失敗します。最初の BLOOP がない場合は別の BLOOP が選択され、ステートメントの実行は成功します。

- UDF は、LOB タイプ BLOB、CLOB、または DBCLOB のいずれかをもつパラメーターまたは結果を使用して定義することができます。DB2 UDB は、LOB 値全体を記憶域に入れてから (値のソースが *LOB ロケーター・ホスト変数* の場合でも)、ユーザー定義関数を呼び出します。たとえば、以下のような C 言語アプリケーションの一部を考えてみましょう。

  ```
  EXEC SQL BEGIN DECLARE SECTION;
    SQL TYPE IS CLOB(150K) clob150K ;        /* LOB host var */
    SQL TYPE IS CLOB_LOCATOR clob_locator1;  /* LOB locator host var */
    char                    string[40];      /* string host var */
  EXEC SQL END DECLARE SECTION;
  ```

  ホスト変数 :clob150K または :clob_locator1 はどちらも、対応するパラメーターが CLOB(500K) として定義されている関数の引き数として有効です。 したがっ

て、 225 ページの『例: 文字列検索』 で定義した FINDSTRING については、以下のステートメントの両方ともプログラムの中で有効になります。

```
... SELECT FINDSTRING (:clob150K, :string) FROM ...
... SELECT FINDSTRING (:clob_locator1, :string) FROM ...
```

- LOB タイプのいずれかをもつ SQL 以外の UDF パラメーターまたは結果は、AS LOCATOR 修飾子を使用して作成することができます。この場合、呼び出しの前に LOB 値全体が記憶域に入れられることはありません。代わりに、LOB ロケーターが UDF に渡されます。

  この機能は、LOB にもとづいた特殊タイプをもつ UDF パラメーターまたは結果にも使用できます。ただし、この機能は、SQL 以外の UDF に限られます。そのような関数 (UDF) への引き数は、定義されたタイプのどのような LOB 値でも構いません。LOCATOR タイプのいずれかとして定義されたホスト変数である必要はありません。引き数としてホスト変数ロケーターを使用することは、UDF パラメーターと結果の定義での AS LOCATOR の使用にはまったく関係がありません。

- UDF は、パラメーターまたは結果としての特殊タイプを使用して定義することができます。(これは、前の例で説明しました。) DB2 UDB は、特殊タイプのソース・データ・タイプの形式で UDF に値を渡します。

  ホスト変数にその元があり、対応するパラメーターが特殊タイプとして定義されている UDF への引き数として使用される特殊タイプ値は、**ユーザーによって、明示的に、特殊タイプにキャストされなければなりません。** 特殊タイプのホスト言語タイプはありません。DB2 UDB の固いタイプの指定は、これを必要とします。そうしない場合は、結果があいまいになります。 そこで、BLOB に定義された BOAT 特殊タイプについて考えてみましょう。また、タイプ BOAT のオブジェクトをその引き数としてとる 227 ページの『例: UDT パラメーターを使用した外部関数』 の BOAT_COST UDF について考えてみましょう。 以下に示す C 言語アプリケーションの一部では、ホスト変数 :ship が、BOAT_COST 関数に渡される BLOB 値を保持しています。

  ```
  EXEC SQL BEGIN DECLARE SECTION;
    SQL TYPE IS BLOB(150K) ship;
  EXEC SQL END DECLARE SECTION;
  ```

  以下のステートメントは両方とも BOAT_COST 関数に正しく解決します。これは、両方とも :ship ホスト変数をタイプ BOAT にキャストするからです。

  ```
  ... SELECT BOAT_COST (BOAT(:ship)) FROM ...
  ... SELECT BOAT_COST (CAST(:ship AS BOAT)) FROM ...
  ```

  データベースに複数の BOAT 特殊タイプがある場合、または、他のスキーマに BOAT UDF がある場合は、関数パスの使用には注意が必要です。 そうしない場合は、結果があいまいになります。

ユーザー定義の特殊タイプ (UDT)

ユーザー定義の特殊タイプは、使用できる組み込みデータ・タイプ以上に DB2 UDB の機能を拡張できるようになるメカニズムです。ユーザー定義の特殊タイプによりユーザーは、DB2 UDB に対して新しいデータ・タイプを定義できるようになります。したがって、ユーザーのビジネスをモデル化したり、ユーザーのデータの意味体系を取り込むうえで、システムが提供する組み込みデータ・タイプを使用することに限定されないため、ユーザーは、少なからぬパワーを得ることになりま

す。特殊なデータ・タイプを使用すると、ユーザーは、既存のデータベース・タイプに対して 1 対 1 でマップできるようになります。

以下のトピックでは、UDT についてより詳しく説明しています。

- 『UDT を使用する理由』
- 『UDT の定義』
- 237 ページの『UDT を使用した表の定義』
- 237 ページの『UDT の操作』
- 243 ページの『UDT、UDF、および LOB の間の協同』

UDT を使用する理由

UDT に関連するいくつかの利点について以下に説明します。

1. **拡張性**

   新しいタイプを定義することにより、DB2 UDB が提供しているタイプのセットを大幅に増やして、アプリケーションをサポートできるようにすることができます。

2. **柔軟性**

   ユーザー定義関数 (UDF) を使用して、システムで使用できるタイプの多様性を増やすことにより、ユーザーの新しいタイプにさまざまな意味体系と動作を指定することができます。

3. **一貫性のある動作**

   固いタイプを指定することにより、UDT が正しく動作できるようになります。これにより、UDT に定義された関数だけが UDT のインスタンスに適用できるようになります。

4. **カプセル化**

   UDT の動作は、UDT に適用できる関数と演算子によって制約を受けます。これにより、使用法の柔軟性がもたらされます。これは、アプリケーションの実行が、ユーザーのタイプ用に選択した内部表現に依存しなくてすむからです。

5. **拡張可能な動作**

   タイプに関するユーザー定義関数の定義により、いつでも、UDT を操作するために提供されている機能性を拡大することができます。(217 ページの『ユーザー定義関数 (UDF)』を参照)

6. **オブジェクト指向拡張機能の基礎**

   UDT は、ほとんどのオブジェクト指向機能の基礎になります。UDT は、オブジェクト指向拡張機能に向けての最も重要なステップを表します。

UDT の定義

UDT は、他のオブジェクト (表、索引、UDF など) と同じように、CREATE ステートメントを使用して定義します。

新しい UDT を定義するには、CREATE DISTINCT TYPE ステートメントを使用します。ステートメント構文およびそれらすべてのオプションについての詳細な説明は、SQL 解説書の CREATE DISTINCT TYPE を参照してください。

CREATE DISTINCT TYPE ステートメントについては、以下のことにご注意ください。

1. 新しい UDT の名前は、修飾された名前でも修飾なしの名前でも構いません。
2. UDT のソース・タイプは、UDT を内部で表現するために DB2 UDB が使用しているタイプです。このため、UDT のソース・タイプは、組み込みデータ・タイプでなければなりません。以前に定義された UDT は、他の UDT のソース・タイプとして使用することはできません。

UDT 定義の一部として、DB2 UDB は、以下のことを行うために、必ずキャスト関数を生成します。

- ソース・タイプの標準名を使用した、UDT からソース・タイプへのキャスト。たとえば、FLOAT にもとづいて特殊タイプを作成すると、DOUBLE というキャスト関数が作成されます。
- ソース・タイプから UDT へのキャスト。UDT への追加キャストがいつ生成されるかの説明については、SQL 解説書を参照してください。

これらの関数は、照会での UDT の操作に重要です。

修飾なしの UDT の解決

修飾なしのタイプ名または関数への参照を解決するためには、関数パスが使用されます。ただし、タイプ名または関数が以下の操作をうけるときを除きます。

- 作成される
- 削除される
- 注釈が付けられる

修飾なしの関数参照が解決される方法については、231 ページの『修飾された関数参照の使用』 を参照してください。

例: CREATE DISTINCT TYPE の使用

以下に、CREATE DISTINCT TYPE を使用した例を示します。

- 例: 通貨
- 例: 履歴書

注: コーディング例に関する詳細は、x ページの『コードに関する特記事項』を参照してください。

例: 通貨

異なる通貨を処理する必要があるアプリケーションを作成しているとします。しかし、DB2 UDB で、異なる通貨が照会の中で互いに直接比較されたり操作されたりしないようにしたいとします。 異なる通貨の値を比較するときには、変換が必要になります。したがって、必要な数 (1 つの通貨に 1 つ) の UDT を定義します。

```
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL (9,2)
CREATE DISTINCT TYPE CANADIAN_DOLLAR AS DECIMAL (9,2)
CREATE DISTINCT TYPE GERMAN_MARK AS DECIMAL (9,2)
```

例: 履歴書

会社への応募者が記入した応募書式を DB2 UDB 表で維持管理し、関数を使用してこれらの書式から必要な情報を取り出したいとします。 これらの関数は、正規の文字ストリングに適用できない (文字ストリングでは戻す情報を見つけることができない) ので、記入済みの書式を表すのに UDT を定義します。

```
CREATE DISTINCT TYPE PERSONAL.APPLICATION_FORM AS CLOB(32K)
```

UDT を使用した表の定義

いくつかの UDT を定義した後、タイプが UDT である列をもつ表の定義を始めることができます。以下に、CREATE TABLE を使用した例を示します。

- 例: 売上高
- 例: 応募用紙

注: コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

例: 売上高

それぞれの国の売上高を保持する表を以下のように定義するとします。

```
CREATE TABLE US_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1985),
   TOTAL         US_DOLLAR)

CREATE TABLE CANADIAN_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1985),
   TOTAL         CANADIAN_DOLLAR)

CREATE TABLE GERMAN_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1985),
   TOTAL         GERMAN_MARK)
```

上の例の UDT は、 236 ページの『例: 通貨』 で使用したものと同じ CREATE DISTINCT TYPE ステートメントを使用して作成されています。上の例では、検査制約が使用されていることにご注意ください。検査制約について詳しくは、 145 ページの『検査制約の追加および使用』を参照してください。

例: 応募用紙

応募者が記入した書式を保持する表を、以下のように定義したいとします。

```
CREATE TABLE APPLICATIONS
  (ID               INTEGER,
   NAME             VARCHAR (30),
   APPLICATION_DATE DATE,
   FORM             PERSONAL.APPLICATION_FORM)
```

ここでは、UDT 名は完全に修飾されています。これは、修飾子が権限 ID と同じものではなく、また、省略時の関数パスが変更されていないからです。タイプ名および関数名が完全に修飾されていない場合は、必ず、DB2 UDB は、現行関数パスにリストされているスキーマを検索し、指定された修飾なしの名前に一致するタイプ名または関数名を探すということをおぼえておいてください。

UDT の操作

UDT に関連する最も重要な概念の 1 つに*固いタイプの指定* があります。強いタイプの指定によって、UDT に定義された関数と演算子だけがそのインスタンスに確実に適用されるようになります。

強いタイプの指定は、UDT のインスタンスが正しいことを確かめるために重要です。たとえば、現行の為替レートにしたがって米国ドルをカナダ・ドルに変換する関数を定義した場合、この同じ関数を使用して、ドイツ・マルクをカナダ・ドルに変換することはしません。間違った答えが得られることが明らかだからです。

強いタイプの指定によって、DB2 UDB では、たとえば、UDT のインスタンスを UDT のソース・タイプのインスタンスと比較する照会を作成することはできなくなります。 同じ理由により、DB2 UDB では、他のタイプで定義された関数を UDT に適用することはできません。 UDT のインスタンスと別のタイプのインスタンスとを比較したい場合は、あるタイプまたは別のタイプのインスタンスをキャストする必要があります。 同じ意味で、この関数を UDT インスタンスに適用したい場合は、UDT インスタンスを、UDT に定義されていない関数のパラメーターのタイプにキャストする必要があります。

UDT の登録の例については、『UDT の操作の例』を参照してください。

UDT の操作の例

以下に、UDT を操作する例を示します。
- 例: UDT と定数との比較
- 例: UDT 間のキャスト
- 例: UDT が関係する比較
- 例: UDT が関係する、ソース化された UDF
- 例: UDT が関係する割り当て
- 例: 動的 SQL での割り当て
- 例: さまざまな UDT が関係する割り当て
- 例: UNION での UDT の使用

注: コーディング例に関する詳細は、x ページの『コードに関する特記事項』を参照してください。

例: UDT と定数との比較

米国で、1992 年 7 月 (7/92) に、どの製品が $100 000.00 を超えて売れたかを知りたいとします。

```
SELECT PRODUCT_ITEM
  FROM   US_SALES
  WHERE  TOTAL > US_DOLLAR (100000)
  AND    month = 7
  AND    year  = 1992
```

米国ドルを米国ドルのソース・タイプのインスタンス (すなわち DECIMAL) と直接比較することができないので、DB2 UDB で提供される DECIMAL から米国ドルにキャストするキャスト関数を使用します。 また、DB2 UDB で提供される別のキャスト関数 (すなわち、米国ドルから DECIMAL にキャストする) を使用して、列の合計を DECIMAL にキャストすることもできます。 UDT からまたは UDT に、どちらの方向にキャストする場合でも、キャスト指定表記または関数表記のどちらを使用しても、キャストを実行することができます。 すなわち、上の照会は、以下のように書くこともできます。

```
SELECT PRODUCT_ITEM
  FROM   US_SALES
  WHERE  TOTAL > CAST (100000 AS us_dollar)
  AND    MONTH = 7
  AND    YEAR  = 1992
```

例: UDT 間のキャスト

カナダ・ドルを米国ドルに変換する UDF を定義したいとします。現行の為替レートは、DB2 の外で管理されているファイルから入手できるとします。 カナダ・ドルでの値を入手し、為替レート・ファイルにアクセスし、対応する米国ドルでの値を戻す UDF を定義します。

一見、このような UDF を書くのは簡単なように見えます。 しかし、すべての C コンパイラーが DECIMAL 値をサポートしているとはかぎりません。さまざまな通貨を表す UDT が DECIMAL として定義されています。ユーザーが定義する UDF は DOUBLE 値を受け取り、戻す必要があります。これは、DOUBLE 値が、10 進数の精度を失わずに DECIMAL 値が表現できる、C が提供する唯一のデータ・タイプであるからです。 したがって、UDF は以下のように定義されます。

```
CREATE FUNCTION CDN_TO_US_DOUBLE(DOUBLE) RETURNS DOUBLE
  EXTERNAL NAME 'MYLIB/CURRENCIES(C_CDN_US)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC
```

カナダ・ドルと米国ドルの間の為替レートは、2 つの UDF 呼び出しの間に変動する可能性があるので、NOT DETERMINISTIC として宣言します。

さて問題は、どのようにしてカナダ・ドルをこの UDF に渡し、そこから米国ドルを入手するかです。カナダ・ドルは DECIMAL 値にキャストする必要があります。DECIMAL 値は DOUBLE にキャストしなければなりません。さらに、戻された DOUBLE 値を DECIMAL にキャストし、DECIMAL 値を米国ドルにキャストしなければなりません。

このようなキャストは、ソース化された UDF を定義したときにはいつでも DB2 UDB によって自動的に実行されます。しかし、UDF のパラメーターと戻りタイプは、ソース関数のパラメーターとリターン・タイプに正確にマッチしません。 したがって、ソース化された UDF を 2 つ定義する必要があります。 1 番目の UDF は、DOUBLE 値を DECIMAL 表現にします。 2 番目の UDF は、DECIMAL 値を UDT に渡します。 以下のように定義します。

```
CREATE FUNCTION CDN_TO_US_DEC (DECIMAL(9,2)) RETURNS DECIMAL(9,2)
  SOURCE CDN_TO_US_DOUBLE (DOUBLE)

CREATE FUNCTION US_DOLLAR (CANADIAN_DOLLAR) RETURNS US_DOLLAR
  SOURCE CDN_TO_US_DEC (DECIMAL())
```

US_DOLLAR 関数を US_DOLLAR(C1) のようにして呼び出す (ここで、C1 はタイプがカナダ・ドルの列です) と、以下の関数を呼び出した場合と同じ結果になることにご注意ください。

```
US_DOLLAR (DECIMAL(CDN_TO_US_DOUBLE (DOUBLE (DECIMAL (C1)))))
```

すなわち、C1 (カナダ・ドルの値) は DECIMAL にキャストされ、次に DECIMAL が DOUBLE 値にキャストされ、DOUBLE 値が CDN_TO_US_DOUBLE 関数に渡されま

す。 この関数は為替レート・ファイルにアクセスし、DOUBLE 値 (米国ドルの値を表す) を戻し、DOUBLE 値は DECIMAL に、次に米国ドルにキャストされます。

以下に示すドイツ・マルクを米国ドルに変換する関数は、上の例に似ています。

```
CREATE FUNCTION GERMAN_TO_US_DOUBLE(DOUBLE)
  RETURNS DOUBLE
  EXTERNAL NAME 'MYLIB/CURRENCIES(C_GER_US)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC

CREATE FUNCTION GERMAN_TO_US_DEC (DECIMAL(9,2))
  RETURNS DECIMAL(9,2)
  SOURCE GERMAN_TO_US_DOUBLE(DOUBLE)

CREATE FUNCTION US_DOLLAR(GERMAN_MARK) RETURNS US_DOLLAR
  SOURCE GERMAN_TO_US_DEC (DECIMAL())
```

例: UDT が関係する比較

1989 年 3 月 (3/89) に、カナダやドイツよりも米国でより多く売れた製品は何かを知りたいとします。

```
SELECT US.PRODUCT_ITEM, US.TOTAL
  FROM US_SALES AS US, CANADIAN_SALES AS CDN, GERMAN_SALES AS GERMAN
  WHERE US.PRODUCT_ITEM = CDN.PRODUCT_ITEM
  AND US.PRODUCT_ITEM = GERMAN.PRODUCT_ITEM
  AND US.TOTAL > US_DOLLAR (CDN.TOTAL)
  AND US.TOTAL > US_DOLLAR (GERMAN.TOTAL)
  AND US.MONTH = 3
  AND US.YEAR  = 1989
  AND CDN.MONTH = 3
  AND CDN.YEAR  = 1989
  AND GERMAN.MONTH = 3
  AND GERMAN.YEAR  = 1989
```

米国ドルを直接カナダ・ドルやドイツ・マルクと比較できないので、カナダ・ドルの値を米国ドルにキャストする UDF と、ドイツ・マルクの値を米国ドルにキャストする UDF を使用します。 これらの値をすべて DECIMAL にキャストし、変換された DECIMAL 値を比較することはできません。通貨が同じでないため、金額を金銭的に比較できないからです。

例: UDT が関係する、ソース化された UDF

組み込み SUM 関数上にソース化された UDF を定義して、ドイツ・マルクで SUM をサポートしたいとします。

```
CREATE FUNCTION SUM (GERMAN_MARKS)
  RETURNS GERMAN_MARKS
  SOURCE SYSIBM.SUM (DECIMAL())
```

1994 年のドイツでの、おのおのの製品の売上高の合計を知りたいとします。 売上高の合計は米国ドルで入手したいとします。

```
SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
  FROM GERMAN_SALES
  WHERE YEAR = 1994
  GROUP BY PRODUCT_ITEM
```

上の例と似た方法で米国ドルについて SUM 関数を定義していない場合は、**SUM (us_dollar (total))** を作成することはできません。

例: UDT が関係する割り当て

新しい応募者が記入した書式をデータベースに格納したいとします。 記入された書式を表すのに使用される文字ストリングの値が入ったホスト変数を定義したとします。

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE IS CLOB(32K) hv_form;
EXEC SQL END DECLARE SECTION;

/* Code to fill hv_form */

INSERT INTO APPLICATIONS
  VALUES (134523, 'Peter Holland', CURRENT DATE, :hv_form)
```

ユーザーが明示的にキャスト関数を呼び出して、文字ストリングを UDT personal.application_form に変換することはありません。 これは、DB2 UDB を使用することによって、ユーザーが UDT のソース・タイプのインスタンスを、その UDT をもつターゲットに割り当てることができるからです。

例: 動的 SQL での割り当て

『例: UDT が関係する割り当て』にある同じステートメントを動的 SQL で使用したい場合は、以下のように、パラメーター・マーカーを使用することができます。

```
EXEC SQL BEGIN DECLARE SECTION;
  long id;
  char name[30];
  SQL TYPE IS CLOB(32K) form;
  char command[80];
EXEC SQL END DECLARE SECTION;

/* Code to fill host variables */

strcpy(command,"INSERT INTO APPLICATIONS VALUES");
strcat(command,"(?, ?, CURRENT DATE, ?)");

EXEC SQL PREPARE APP_INSERT FROM :command;
EXEC SQL EXECUTE APP_INSERT USING :id, :name, :form;
```

この例では、DB2 UDB のキャスト指定を使用して、DB2 UDB に、パラメーター・マーカーのタイプが CLOB(32K) (UDT に割り当てることができるタイプ) であることを通知しています。 ホスト言語が UDT をサポートしていないので、UDT タイプのホスト変数を宣言することができないことをおぼえておいてください。 したがって、パラメーター・マーカーのタイプが UDT であると指定することはできません。

例: さまざまな UDT が関係する割り当て

240 ページの『例: UDT が関係する、ソース化された UDF』 の、ドイツ・マルクについてソース化された UDF と同様に、組み込み SUM 関数上に、ソース化された UDF を 2 つ定義して、米国ドルとカナダ・ドルについて SUM をサポートしたいとします。

```
CREATE FUNCTION SUM (CANADIAN_DOLLAR)
       RETURNS CANADIAN_DOLLAR
       SOURCE SYSIBM.SUM (DECIMAL())
```

```
CREATE FUNCTION SUM (US_DOLLAR)
  RETURNS US_DOLLAR
  SOURCE SYSIBM.SUM (DECIMAL())
```

ここで、上司の要求により、各製品の年間総売上高を、国別に、米国ドルで、別々の表に維持管理する必要があるとします。

```
CREATE TABLE US_SALES_94
       (PRODUCT_ITEM  INTEGER,
        TOTAL         US_DOLLAR)

     CREATE TABLE GERMAN_SALES_94
       (PRODUCT_ITEM  INTEGER,
        TOTAL         US_DOLLAR)

     CREATE TABLE CANADIAN_SALES_94
       (PRODUCT_ITEM  INTEGER,
        TOTAL         US_DOLLAR)

     INSERT INTO US_SALES_94
       SELECT PRODUCT_ITEM, SUM (TOTAL)
       FROM US_SALES
       WHERE YEAR = 1994
       GROUP BY PRODUCT_ITEM

     INSERT INTO GERMAN_SALES_94
       SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
       FROM GERMAN_SALES
       WHERE YEAR = 1994
       GROUP BY PRODUCT_ITEM

     INSERT INTO CANADIAN_SALES_94
       SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
       FROM CANADIAN_SALES
       WHERE YEAR = 1994
       GROUP BY PRODUCT_ITEM
```

カナダ・ドルとドイツ・マルクの値を米国ドルに明示的にキャストします。これは、異なる UDT が、相互に直接割り当てることができないからです。 UDT はそれ自身のソース・タイプにしかキャストすることができないので、キャスト指定構文を使用することはできません。

例: UNION での UDT の使用

会社の各製品のすべての売上高を示す照会を米国のユーザーに提供したいとします。

```
       SELECT PRODUCT_ITEM, MONTH, YEAR, TOTAL
       FROM US_SALES
       UNION
       SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR (TOTAL)
       FROM CANADIAN_SALES
       UNION
       SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR (TOTAL)
       FROM GERMAN_SALES
```

カナダ・ドルを米国ドルに、ドイツ・マルクを米国ドルにキャストします。これは、UDT が共用体 (UNION) 互換性があるのは、同じ UDT に対してだけだからです。 関数表記を使用して UDT 間のキャストを行う必要があります。キャスト指定を使用すると、UDT とそのタイプの間でだけキャストを行うからです。

UDT、UDF、および LOB の間の協同

前の節では、個々の DB2 UDB オブジェクト拡張機能 (UDT、UDF、および LOB) の定義方法と使用方法について説明しました。 ここでは、これらの 3 つのオブジェクト拡張機能の間にある協同について説明します。

詳細については、以下のセクションを参照してください。

- 『UDT、UDF、および LOB の結合』
- 『複合アプリケーションの例』

UDT、UDF、および LOB の結合

オブジェクト・オリエンテーションの概念によれば、アプリケーション・ドメインの中の類似オブジェクトは関連タイプにグループ化することができます。 これらのタイプのおのおのは、名前、内部表現、および動作をもっています。UDT を使用することにより、新しいタイプの名前と、その内部表現の方法を DB2 UDB に伝えることができます。 LOB は、新しいタイプに使用できる内部表現の 1 つで、大きなサイズの複合構造に最も適した表現です。 UDF を使用すると、新しいタイプの動作を定義できます。 したがって、UDT、UDF、および LOB の間には重要な協同があります。 複雑なデータ構造と動作をもったアプリケーション・タイプは UDT としてモデル化され、UDT は内部では LOB として表現され、その動作は UDF によってインプリメントされます。 アプリケーション・タイプの意味体系上の整合性の統制をとる規則は、制約とトリガーとして表されます。 相互に関連する UDT と UDF のコントロールと編成をより良く行うためには、これらを同じスキーマに保持しておくことをお勧めします。

複合アプリケーションの例

以下に示す例で、複合アプリケーションで UDT、UDF、および LOB をどのように一緒に使用するかについて説明します。

例: UDT と UDF の定義
例: データベースにデータを入れるための LOB 関数の使用
例: UDT のインスタンスを照会するための UDF の使用
例: UDT インスタンスを操作するための LOB ロケーターの使用

注: コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

例: UDT と UDF の定義

会社に送られてきた電子メール (e-mail) を DB2 UDB 表に保持しておきたいとします。プライバシーの問題はここでは無視するとして、このような電子メールに対して照会を作成して、件名や、顧客からの注文の受信に電子メール・サービスが使用されている頻度などを調べることを計画します。電子メールは非常に大きく、複雑な内部構造 (送信者、受信者、件名、日付、電子メールの内容) をもっています。したがって、ソース・タイプがラージ・オブジェクトである UDT を使用して電子メールを保持することに決めます。電子メールのタイプについて、たとえば、電子メールの件名、送信者、日付などを取り出す関数をもつ UDF のセットを定義します。また、電子メールの内容について検索を実行する関数も定義します。これらのことは、以下に示すように、CREATE ステートメントを使用して行います。

```
CREATE DISTINCT TYPE E_MAIL AS BLOB (1M)

CREATE FUNCTION SUBJECT (E_MAIL)
  RETURNS VARCHAR (200)
  EXTERNAL NAME 'LIB/PGM(SUBJECT)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION

CREATE FUNCTION SENDER (E_MAIL)
  RETURNS VARCHAR (200)
  EXTERNAL NAME 'LIB/PGM(SENDER)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION

CREATE FUNCTION RECEIVER (E_MAIL)
  RETURNS VARCHAR (200)
  EXTERNAL NAME 'LIB/PGM(RECEIVER)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION

CREATE FUNCTION SENDING_DATE (E_MAIL)
  RETURNS DATE CAST FROM VARCHAR(10)
  EXTERNAL NAME 'LIB/PGM(SENDING_DATE)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION

CREATE FUNCTION CONTENTS (E_MAIL)
  RETURNS BLOB (1M)
  EXTERNAL NAME 'LIB/PGM(CONTENTS)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION

CREATE FUNCTION CONTAINS (E_MAIL, VARCHAR (200))
  RETURNS INTEGER
  EXTERNAL NAME 'LIB/PGM(CONTAINS)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION

CREATE TABLE ELECTRONIC_MAIL
  (ARRIVAL_TIMESTAMP TIMESTAMP,
   MESSAGE E_MAIL)
```

例: データベースにデータを入れるための LOB 関数の使用

ファイルに保持されている電子メールを DB2 UDB に転送することによって表にデータを入れていくとします。 すべての電子メールが DB2 UDB に格納されるまで、さまざまな値の HV_EMAIL_FILE を使用して、以下の INSERT ステートメントを複数回実行します。

```
EXEC SQL BEGIN DECLARE SECTION
  SQL TYPE IS BLOB_FILE HV_EMAIL_FILE;

EXEC SQL END DECLARE SECTION
  strcpy (HV_EMAIL_FILE.NAME, "/u/mail/email/mbox");
  HV_EMAIL_FILE.NAME_LENGTH = strlen(HV_EMAIL_FILE.NAME);
  HV_EMAIL_FILE.FILE_OPTIONS = 2;

EXEC SQL INSERT INTO ELECTRONIC_MAIL
  VALUES (CURRENT TIMESTAMP, :hv_email_file);
```

DB2 LOB サポートによって提供されているすべての関数は、ソース・タイプが LOB である UDT に適用することができます。 したがって、LOB ファイル参照変数を使用してファイルの内容を UDT 列に割り当てています。 BLOB タイプの値を電子メール・タイプに変換するために、キャスト関数は使用していません。 これは、DB2 UDB を使用すると、特殊タイプのソース・タイプの値を、特殊タイプのターゲットに割り当てることができるからです。

例: UDT のインスタンスを照会するための UDF の使用

注文について、特定の顧客から送られた電子メールの数を調べたいとします。顧客の電子メール・アドレスは顧客表にあるとします。

```
SELECT COUNT (*)
  FROM ELECTRONIC_MAIL AS EMAIL, CUSTOMERS
  WHERE SUBJECT (EMAIL.MESSAGE) = 'customer order'
  AND CUSTOMERS.EMAIL_ADDRESS = SENDER (EMAIL.MESSAGE)
  AND CUSTOMERS.NAME = 'Customer X'
```

この SQL では、UDT に定義されている UDF (複数) を使用しています。これは、これらの UDF が UDT を操作する唯一の手段だからです。この意味で、UDT 電子メールは完全にカプセル化されています。その内部表現および構造は隠れていて、定義された UDF によってのみ操作することができます。 これらの UDF は、データの内部表現を露出することなく、そのデータをどのように解釈するかがわかっています。

1994 年に受信した、市場における会社の製品の売上実績に関するすべての電子メールの詳細を知りたいとします。

```
SELECT SENDER (MESSAGE), SENDING_DATE (MESSAGE), SUBJECT (MESSAGE)
  FROM ELECTRONIC_MAIL
  WHERE CONTAINS (MESSAGE,
  '"performance" AND "products" AND "marketplace"') = 1
```

メッセージの内容を、関係のあるキーワードまたは同義語をサーチして分析することができる contains UDF が使用されています。

例: UDT インスタンスを操作するための LOB ロケーターの使用

アプリケーション・プログラムの中のホスト変数に電子メール全体を転送せずに、ある特定の電子メールに関する情報を入手したいとします。 (電子メールはきわめて大きなものであることを思い出してください。) UDT が LOB について定義されているので、この目的のためには、LOB ロケーターを使用することができます。

```
EXEC SQL BEGIN DECLARE SECTION
  long hv_len;
  char hv_subject[200];
  char hv_sender[200];
  char hv_buf[4096];
  char hv_current_time[26];
```

```
  SQL TYPE IS BLOB_LOCATOR hv_email_locator;
EXEC SQL END DECLARE SECTION

EXEC SQL SELECT MESSAGE
  INTO :hv_email_locator
  FROM ELECTRONIC MAIL
  WHERE ARRIVAL_TIMESTAMP = :hv_current_time;

EXEC SQL VALUES (SUBJECT (E_MAIL(:hv_email_locator))
  INTO :hv_subject;
.... code that checks if the subject of the e_mail is relevant ....
.... if the e_mail is relevant, then.............................

EXEC SQL VALUES (SENDER (CAST (:hv_email_locator AS E_MAIL)))
  INTO :hv_sender;
```

ホスト変数のタイプが BLOB ロケーター (UDT のソース・タイプ) であるので、BLOB ロケーターが UDT について定義された UDF の引き数として使用されるたびに、明示的に UDT に変換しています。

データ・リンクの使用

データ・リンクというデータ・タイプは、データベース・ファイルに格納できるデータのタイプを拡張するための 1 つの基本構築ブロックです。データ・リンクという考え方は、列に格納される実際のデータは、オブジェクトを指し示すポインターにすぎないということです。このオブジェクトは、イメージ・ファイル、音声記録、テキスト・ファイルなど、なんでも構いません。オブジェクトに解決するために使用される方式は、URL を格納することです。このことは、表の中の行が従来のデータ・タイプのオブジェクトに関する情報を入れるのに使用され、オブジェクト自身は、データ・リンクというデータ・タイプを使用して参照されることを意味しています。ユーザーは、新しい SQL スカラー関数を使用して、パスを、オブジェクトおよびそのオブジェクトが格納されているサーバーに戻すことができます。データ・リンクというデータ・タイプを使用すると、行とオブジェクトの間に緩い関係ができます。たとえば、行を削除すると、データ・リンクによって参照されているオブジェクトへの関係を切断しますが、オブジェクト自身が削除されることはありません。

データ・リンク列を使用して作成された SQL 表を使用すると、オブジェクトに関する情報を保持することができますが、オブジェクト自体を持たずにすみます。この概念により、SQL 表を使用して管理できるデータのタイプの柔軟性が増します。たとえば、ユーザーが、多数のビデオ・クリップをサーバーの統合ファイル・システムに保管している場合、これらのビデオ・クリップについての情報を、SQL 表を使用して入れたい場合があります。しかし、ユーザーは、すでにディレクトリーに格納されたオブジェクトをもっているので、SQL 表で実際に大量の記憶域をもつのではなく、オブジェクトへの参照だけをもつようにしたいと考えます。この解決策は、データ・リンクを使用することです。SQL 表には、従来の SQL データ・タイプを使用して、おのおののクリップに関する情報 (タイトル、長さ、日付など) を入れますが、クリップ自体は、データ・リンク列を使用して参照できるようにします。表の各行にはオブジェクトの URL と、オプションでコメントを入れます。このようにすると、クリップを処理するアプリケーションは SQL インターフェースを使用して URL を検索し、次に、ブラウザーまたはその他のプレーバック・ソフトウェアを使用して URL を処理し、ビデオ・クリップを表示することができます。

この手法を使用すると、以下のような利点があります。

- 統合ファイル・システムがあらゆるタイプのストリーム・ファイルを格納できます。
- 統合ファイル・システムが、文字列または LOB 列に収まらないような非常に大きなサイズのラージ・オブジェクトを格納できます。
- 統合ファイル・システムの階層的な構造は、ストリーム・ファイル・オブジェクトを編成し、処理するのに非常に適しています。
- 大量のオブジェクトを、データベースの外に置き、統合ファイル・システムの中に入れることによって、SQL ランタイム・エンジンが照会と報告書を処理し、ファイル・システムがビデオのストリーミング、イメージやテキストの表示などを処理するので、アプリケーションのパフォーマンスが向上します。

データ・リンクを使用すると、オブジェクトが「リンクされている」状況にあるときにコントロールがしやすくなります。データ・リンク列は、SQL 表の中にオブジェクトを参照する行がある間は、参照されているそのオブジェクトが削除、移動、名前変更されないように作成することができます。このオブジェクトは「リンクされている」と考えます。その参照をもつ行が削除されると、オブジェクトはリンク解除されます。この概念をよく理解するには、データ・リンク列を作成するときに指定できるコントロールのレベルについて理解する必要があります。データ・リンク列を作成するときに使用する正確な構文については、SQL 解説書を参照してください。

データ・リンクの詳細については、以下のセクションを参照してください。

- 『NO LINK CONTROL』
- 『FILE LINK CONTROL (ファイル・システム許可を使用した)』
- 248 ページの『FILE LINK CONTROL (データベース許可を使用した)』
- 248 ページの『データ・リンクを処理するために使用するコマンド』

NO LINK CONTROL

NO LINK CONTROL を使用して列を作成すると、行が SQL 表に追加されるときにリンクが行われません。URL は構文的に正しいかどうかが検査されますが、サーバーがアクセスできるか、または、ファイルが存在するかはチェックされません。

FILE LINK CONTROL (ファイル・システム許可を使用した)

データ・リンク列が、ファイル・システム (FS) 許可を使用して FILE LINK CONTROL として作成されると、システムは、すべてのデータ・リンク値が有効な URL で、有効なサーバー名とファイル名をもっていることを確かめます。ファイルは、行が SQL 表に挿入されるときに存在していなければなりません。オブジェクトが見つかると、「リンクされている」とマークされます。すなわち、オブジェクトがリンクされている間は、移動、削除、名前変更ができないことになります。また、オブジェクトがリンクできるのは一回だけです。URL のサーバー名の部分がリモート・システムを指定している場合は、そのシステムはアクセス可能でなければなりません。データ・リンク値をもっている行が削除されると、オブジェクトはリンク解除されます。データ・リンク値が別の値に更新されると、古いオブジェクトはリンク解除され、新しいオブジェクトがリンクされます。

統合ファイル・システムは、まだ、リンクされたオブジェクトの許可を管理する役割をもっています。リンクまたはリンク解除処理中は、許可が変更されることはありません。このオプションにより、オブジェクトがリンクされている間は、オブジェクトの存在をコントロールすることができます。

FILE LINK CONTROL (データベース許可を使用した)

データ・リンク列が、データベース許可を使用して、FILE LINK CONTROL として作成されると、URL が検査され、オブジェクトに対する現存する許可がすべて削除されます。オブジェクトの所有権は、特別のシステム提供ユーザー・プロファイルに変更されます。オブジェクトがリンクされている間は、オブジェクトへのアクセスは、オブジェクトがリンクされている SQL 表から URL を入手することによってのみ行うことができます。これは、SQL から戻される URL に付加されている特別なアクセス・トークンを使用して処理されます。アクセス・トークンがない場合は、オブジェクトへのアクセスはすべて、権限違反になり、失敗します。アクセス・トークンをもつ URL が、通常の手段 (FETCH、SELECT INTO など) で SQL 表から検索された場合は、ファイル・システム・フィルターがアクセス・トークンを検査し、オブジェクトへのアクセスを許可します。

このオプションの使用により、直接的な手段でアクセスしようとするユーザーが、リンクされたオブジェクトを更新するのを防ぐことができます。オブジェクトへのアクセスは、SQL 操作からアクセス・トークンを入手することによってのみ行うことができるので、管理者は、データ・リンクをもつ SQL 表へのデータベース許可を使用して、リンクされたオブジェクトへのアクセスを効果的にコントロールすることができます。

データ・リンクを処理するために使用するコマンド

データ・リンクというデータ・タイプのサポートは、3 つの異なるコンポーネントに分けられます。

1. DB2 UDB のデータベース・サポートは、DATALINK と呼ばれるデータ・タイプをもっています。これは、CREATE TABLE および ALTER TABLE などの SQL ステートメントで指定します。列には、NULL 以外の省略時値をもつことはできません。データへのアクセスは、SQL インターフェースを使用しなければなりません。これは、DATALINK 自身が、どのホスト変数タイプとも互換性がないからです。SQL スカラー関数は、DATALINK 値を文字形式で検索するのに使用できます。SQL には DLVALUE スカラー関数があり、列に値を INSERT または UPDATE するときに使用します。
2. データ・リンク・ファイル・マネージャー (DLFM) は、サーバー上のファイルのリンク状況を維持管理し、各ファイルのメタデータを把握するコンポーネントです。このコードは、リンク、リンク解除、コミットメント・コントロールなどの問題を処理します。データ・リンクの 1 つの重要な側面は、DLFM が、データ・リンク列をもつ SQL 表と同じ物理システムになくてもよいということです。したがって、SQL 表は、同じシステムの統合ファイル・システムにあるオブジェクトでも、リモート・サーバーの統合ファイル・システムにあるオブジェクトでも、リンクすることができます。
3. データ・リンク・フィルターは、ファイル・システムが、リンクされているオブジェクトを入れるものとして指定されたディレクトリーにあるファイルに対して操作を試行するときに実行されなければなりません。このコンポーネントは、フ

ファイルがリンクされているか、また、任意指定で、ユーザーがファイルにアクセスすることを許可されているかを判断します。ファイル名にアクセス・トークンが組み込まれている場合は、トークンが検査されます。このフィルター・プロセスは、オーバーヘッドがかかるので、アクセスされたオブジェクトがデータ・リンクの「プレフィックス」内のディレクトリーのいずれかにあるときにのみ実行されます。プレフィックスについては、以下の説明を参照してください。

データ・リンクを処理するときには、システムを正しく構成するためにとらなければならない以下のステップがあります。

- データ・リンクを処理するときに使用されるどのシステムにも、TCP/IP を構成する必要があります。これらのシステムには、データ・リンク列をもつ SQL 表が作成されるシステム、および、リンクされるオブジェクトが入るシステムが含まれます。ほとんどの場合、これは同じシステムです。オブジェクトを参照するのに使用される URL には TCP/IP サーバー名が入っているので、この名前は、データ・リンクをもつ予定のシステムによって認識されなければなりません。コマンド CFGTCP は、TCP/IP 名を構成するために、または、TCP/IP ネーム・サーバーを登録するために使用できます。
- SQL 表をもつシステムは、ローカル・データベース・システムおよび任意指定のリモート・システムを反映するように更新されたリレーショナル・データベース・ディレクトリーをもっていなければなりません。コマンド WRKRDBDIRE は、このディレクトリーに情報を追加または変更するために使用します。整合性をもたせるために、TCP/IP サーバー名およびリレーショナル・データベース名に同じ名前を使用することをお勧めします。
- DLFM サーバーは、リンクされるオブジェクトを入れる予定のすべてのシステムで開始する必要があります。DLFM サーバーを開始するには、コマンド STRTCPSVR *DLFM を使用します。DLFM サーバーは、CL コマンド ENDTCPSVR *DLFM を使用して終了します。

DLFM を開始した後で、いくつかのステップを使用して、DLFM を構成する必要があります。これらの DLFM 関数は、QShell インターフェースから入ることができる実行可能スクリプトを介して使用することができます。対話式シェル・インターフェースに行くには、CL コマンド QSH を使用します。これにより、コマンド入力画面が表示され、ここから、DLFM スクリプト・コマンドを入力することができます。スクリプト・コマンド dfmadmin -help を使用して、ヘルプ・テキストと構文図を表示することができます。よく使用される関数については、CL コマンドも提供されています。CL コマンドを使用すると、ほとんどあるいはすべての DLFM 構成作業は、スクリプト・インターフェースを使用せずに実行できます。ユーザーの好みに応じて、QSH コマンド入力画面のスクリプト・コマンドを使用するか、あるいは、CL コマンド入力画面の CL コマンドを使用するかのどちらでも選択することができます。

これらの関数はすべて、システム管理者用またはデータベース管理者用のものなので、特殊権限 *IOSYSCFG が必要です。

プレフィックスの追加 - プレフィックスは、リンクされるオブジェクトを入れる予定のパスまたはディレクトリーです。システム上に DLFM をセットアップするときは、管理者は、データ・リンクに使用されるプレフィックスを追加する必要があります。プレフィックスを追加するには、スクリプト・コマンド dfmadmin -add_prefix を使用します。プレフィックスを追加する CL コマンドは ADDPFXDLFM です。

たとえば、サーバー TESTSYS1 に、リンクされるオブジェクトをもつ /mydir/datalinks/ というディレクトリーがあるとします。管理者は、コマンド ADDPFXDLFM PREFIX(('/mydir/datalinks/')) を使用してプレフィックスを追加します。これにより、以下のような URL のリンク、

```
http://TESTSYS1/mydir/datalinks/videos/file1.mpg
```

あるいは

```
file://TESTSYS1/mydir/datalinks/text/story1.txt
```

は、パスが有効なプレフィックスで始まっているので、有効です。

また、スクリプト・コマンド dfmadmin -del_prefix を使用してプレフィックスを削除することもできます。これは、よく使用される関数ではありません。この関数が使用できるのは、プレフィックス名内にあるディレクトリー構造のどこにもリンクされているオブジェクトがない場合に限られるからです。

ホスト・データベースの追加 - ホスト・データベースは、そこからリンク要求が出されるリレーショナル・データベース・システムです。DLFM が、データ・リンクをもつ予定の SQL 表と同じシステムにある場合は、ローカル・データベース名だけを追加するだけですみます。DLFM が、リモート・システムから出されるリンク要求を受け取る場合は、リモート・システムのすべての名前を DLFM に登録する必要があります。ホスト・データベースを追加するスクリプト・コマンドは dfmadmin -add_db で、CL コマンドは ADDHDBDLFM です。この関数では、SQL 表が入っているライブラリーも登録する必要があります。

たとえば、サーバー TESTSYS1 で、ユーザーがすでに /mydir/datalinks/ プレフィックスを追加している場合、ローカル・システムのライブラリー TESTDB または PRODDB の中の SQL 表が、このサーバーにあるオブジェクトをリンクするようにしたいとします。コマンド **ADDHDBDLFM HOSTDBLIB((TESTDB) (PRODDB)) HOSTDB(TESTSYS1)** を使用します。

DLFM が開始され、プレフィックスとホスト・データベース名が登録されると、ファイル・システムにあるオブジェクトのリンクを始めることができます。

第 13 章 ユーザー定義関数 (UDF) の作成

ユーザー定義関数 (UDF) は、ソース、外部、および SQL という 3 つのタイプからなります。ソース関数 UDF は、他の関数を呼び出して操作を実行します。SQL および外部関数 UDF は、ユーザーがコードを作成して実行する必要があります。この章では、SQL および外部関数の作成について説明します。外部関数および SQL 関数を作成するには、以下のことを行う必要があります。

- 『UDF 実行時環境』を理解する。
- データベースに認知されるようにUDF を登録する。
- 関数コードを作成し、関数を実行して、適切なパラメーターを渡す。
- 関数をデバッグし、テストする。

関数を作成する例については、以下を参照してください。

- 263 ページの『例: 数の平方を求める UDF』
- 265 ページの『例: カウンター』

UDF 実行時環境

UDF が実行される環境とその環境の制約について、考慮する点がいくつかあります。UDF 用に複雑な関数コードを作成することを計画している場合は、以下の事項を注意深く考慮する必要があります。

考慮すべき要素には次のものがあります。

- 『UDF が実行される時間の長さ』

- 252 ページの『スレッドについての考慮事項』
- 252 ページの『並列処理』

UDF が実行される時間の長さ

UDF は SQL ステートメントの実行の中で呼び出されます。SQL ステートメントは、通常、表の中の何千もの行に対して実行される照会操作です。このため、UDF は低レベルのデータベースから呼び出す必要があります。

低レベルから呼び出される結果として、UDF が呼び出される時点、あるいは、UDF の実行中に、ある種のリソースを保留 (ロックして占有する) します。これらのリソースは主に、UDF を呼び出している SQL ステートメントで必要な表や索引に関係するロックです。保留されたこれらのリソースがあるために何分あるいは何時間もかかるような操作を、UDF が実行しないことが重要になります。リソースを長時間保留することは重大なことなので、データベースは、一定の時間だけ、UDF が完了するまで待ちます。UDF が与えられた時間内に終わらない場合は、SQL ステートメントは失敗します。これは、エンド・ユーザーにとっては腹立たしいことです。

データベースで使用している UDF の省略時の待ち時間は、通常の UDF が完了するのに十分な時間のはずです。しかし、長い実行時間がかかる UDF があり、その待ち時間を増やしたい場合は、照会 INI ファイルで UDF_TIME_OUT オプション

を使用してこれを行うことができます。INI ファイルの詳細については、データベース・パフォーマンスおよび *Query 最適化* の Query オプション・ファイル QAQQINI を参照してください。ただし、UDF_TIME_OUT で指定した値に関係なく、データベースが超えられない最大制限時間があることにご注意ください。

UDF が実行される間はリソースが保留されるので、元の SQL ステートメントに割り振られている表や索引と同じ表や索引で UDF が操作しないこと、あるいは、同じ表や索引で UDF が操作する場合は、SQL ステートメントで実行されている操作と対立する操作を UDF が実行しないことが重要になります。具体的には、UDF は、そのような表では、行を挿入、更新、または削除する操作を実行しないようにしなければなりません。

スレッドについての考慮事項

FENCED として定義された UDF は、その UDF を呼び出した SQL ステートメントと同じジョブの中で実行されます。 ただし、UDFは、SQL ステートメントを実行しているスレッドとは別のシステム・スレッドの中で実行されます。 スレッドの詳細については、Information Center の「プログラミング」のカテゴリーのマルチスレッド・プログラミングのためのデータベースに関する考慮事項を参照してください。

UDF は、SQL ステートメントと同じジョブの中で実行されるので、UDF は SQL ステートメントと同じ環境の多くを共用します。ただし、UDF が別のスレッドのもとで実行されるので、スレッドに関する以下の考慮事項が必要になります。

- UDF は、SQL ステートメントのスレッドによって保持されているスレッド・レベルのリソースと対立します。主に、これは、上記の表リソースです。
- UDF は、SQL ステートメントが呼び出されたときに活動状態だったプログラム借用権限を継承しません。UDF の権限は、UDF プログラム自体に関連している権限、または、SQL ステートメントを実行しているユーザーの権限から得られます。
- UDF は、2 次スレッドで実行がブロックされている操作は行えません。
- UDF プログラムは、名前付き活動化グループのもと、あるいは、その呼び出し元 (ACTGRP パラメーター) の活動化グループの中で実行されるように作成する必要があります。ACTGRP(*NEW) を指定するプログラムは、UDF として実行することはできません。

関数を UNFENCED として定義することについての詳細は、263 ページの『関数を隔離または隔離解除にする』を参照してください。

並列処理

並列処理が行えるように 1 つの UDF を定義することができます。これは、同じ UDF プログラムが同時に複数のスレッドで実行できることを意味します。したがって、ALLOW PARALLEL が UDF に指定された場合は、スレッド・セーフになるようにしてください。スレッドの詳細については、iSeries Information Center の「プログラミング」カテゴリーのマルチスレッド・プログラミングのためのデータベースに関する考慮事項を参照してください。

ユーザー定義表関数は並列に実行できません。したがって、ユーザー定義表関数を作成するときは必ず DISALLOW PARALLEL を指定する必要があります。

関数コードの作成

関数コードを作成するには、その関数を実行するための SQL または外部関数の作成方法について理解していなければなりません。また、関数を正確に定義するために、データベースと関数コードの間のインターフェースについての理解、および、実行可能プログラムを作成するときのパッケージ・オプションの決定が必要になります。

UDF を SQL 関数または外部関数として作成することができます。詳細については、以下を参照してください。

- 『SQL 関数としての UDF の作成』
- 254 ページの『外部関数としての UDF の作成』

261 ページの『表関数の考慮事項』 も参照してください。

SQL 関数としての UDF の作成

SQL 関数とは、CREATE FUNCTION ステートメントを使用して、ユーザーが定義し、作成し、登録した UDF のことです。 したがって、SQL 関数は、SQL 言語だけを使用して作成され、その定義は、1 つの (ただし、可能性として大きな) CREATE FUNCTION ステートメントの中に完全に含まれます。 SQL 関数の作成によって、UDF の登録が行われ、関数の実行可能コードが生成され、パラメーターが実際に渡される方法の詳細がデータベースに定義されます。 したがって、これらの関数の作成には複雑さはなく、関数にエラーが持ち込まれるチャンスは少ないといえます。

スカラー UDF

SQL スカラー関数の CREATE FUNCTION ステートメントは、以下のような一般フローにしたがいます。

```
CREATE FUNCTION function-name(parameters) RETURNS return-value
LANGUAGE SQL
BEGIN
   sql-statements
END
```

たとえば、日付にもとづいて優先順位を戻す関数は、以下のように作成されます。

```
CREATE FUNCTION PRIORITY(indate DATE) RETURNS CHAR(7)
LANGUAGE SQL
BEGIN
RETURN(
      CASE   WHEN indate>CURRENT DATE-3 DAYS THEN 'HIGH'
         WHEN indate>CURRENT DATE-7 DAYS THEN 'MEDIUM'
         ELSE 'LOW'
      END
      );
END
```

この関数は、以下のようにして呼び出されます。

```
SELECT ORDERNBR, PRIORITY(ORDERDUEDATE) FROM ORDERS
```

表 UDF

SQL 表関数の CREATE FUNCTION ステートメントは、以下のような一般フローにしたがいます。

```
CREATE FUNCTION function-name(parameters) RETURNS TABLE return-columns
LANGUAGE SQL
BEGIN
   sql-statements
RETURN
   select-statement
END
```

たとえば、日付にもとづいてデータを戻す関数は、以下のように作成されます。

```
CREATE FUNCTION PROJFUNC(indate DATE)
  RETURNS TABLE (PROJNO CHAR(6), ACTNO SMALLINT, ACTSTAFF DECIMAL(5,2),
           ACSTDATE DATE, ACENDATE DATE)
  LANGUAGE SQL
  BEGIN
   RETURN SELECT * FROM PROJACT
     WHERE ACSTDATE<=indate;
  END
```

この関数は、以下のようにして呼び出されます。

```
SELECT * FROM TABLE(PROJFUNC(:datehv)) X
```

SQL 表関数には、ただ 1 つの RETURN ステートメントが必要です。

外部関数としての UDF の作成

UDF の実行可能コードは、SQL 以外の言語で作成することもできます。この方法は SQL 関数よりも若干煩わしいのですが、ユーザーにとって最も有効な言語が使用できる柔軟性があります。実行可能コードは、プログラムまたはサービス・プログラムのどちらにでも入れることができます。

外部関数は Java で作成することもできます。パラメーターの説明については、「IBM Developer Kit for Java」の Java SQL ルーチンを参照してください。

DB2 UDB から外部関数への引き数の受け渡し

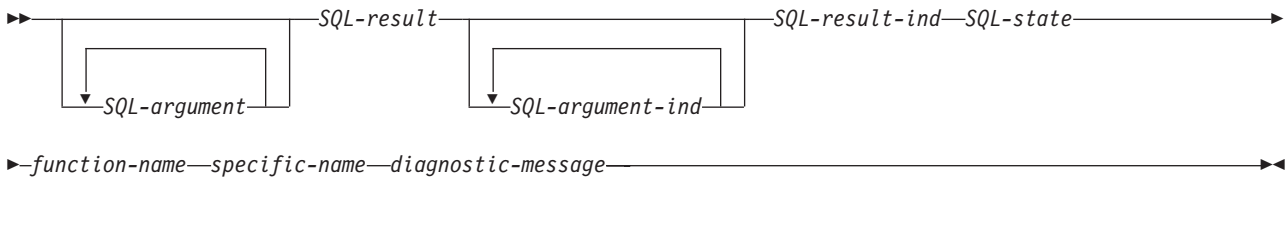
DB2 UDB は、UDF に渡されるすべてのパラメーターに記憶域を提供します。したがって、パラメーターは外部関数にアドレスで渡されます。これは、プログラムでパラメーターを渡す通常の方法です。サービス・プログラムの場合は、関数コードの中でパラメーターが正しく定義されるようにしなければなりません。

UDF の中でパラメーターを定義し、使用する場合は、あるパラメーター用に定義されたよりも多くの記憶域が、そのパラメーターに対して参照されないように十分な注意が必要です。パラメーターはすべて同じ記憶スペースに格納されるので、あるパラメーターの記憶スペースを超えると、別のパラメーターの値を上書きすることになります。このことは、無効な入力データを関数が使用する原因になったり、データベースに戻される値が無効になる原因になります。

外部 UDF が使用できるいくつかのパラメーター・スタイルがサポートされています。それぞれのスタイルの相違点は、外部プログラムまたはサービス・プログラムに渡されるパラメーターの数の相違です。

パラメーター・スタイル SQL: パラメーター・スタイル SQL は、業界標準の構造化照会言語 (SQL) に準拠しています。 このパラメーター・スタイルは、スカラ

ー UDF でのみ、使用できます。パラメーター・スタイル SQL を使用すると、パラメーターは、外部プログラムに、以下のように (指定された順序で) 渡されます。



SQL-argument
この引き数は、UDF を呼び出す前に、DB2 UDB によって設定されます。この値は *n* 回繰り返されます。ここで、*n* は、関数参照で指定された引き数の数です。これらの引き数のおのおのの値は、関数呼び出しで指定された式からとられます。これは、CREATE FUNCTION ステートメントの定義されたパラメーターのデータ・タイプの中で表されます。注: これらのパラメーターは入力としてのみ扱われます。UDF によって行われたパラメーター値の変更は、DB2 UDB によって無視されます。

SQL-result
この引き数は、DB2 UDB に戻る前に、UDF によって設定されます。データベースは、戻り値用の記憶域を提供します。パラメーターはアドレスで渡されるので、アドレスは、戻り値が入る記憶域のアドレスになります。データベースは、CREATE FUNCTION ステートメントで定義された戻り値に必要とされるだけの記憶域を提供します。 CREATE FUNCTION ステートメントで CAST FROM 文節が使用されている場合は、DB2 UDB は、UDF が CAST FROM 文節で定義された値を戻すと想定し、CAST FROM 文節が使用されていない場合は、DB2 UDB は、UDF が RETURNS 文節で定義された値を戻すと想定します。

SQL-argument-ind
この引き数は、UDF を呼び出す前に、DB2 UDB によって設定されます。これは、対応する *SQL-argument* がヌルかどうかを判断するために、UDF によって使用されます。前に述べたように、*n* 番目の *SQL-argument-ind* は、*n* 番目の *SQL-argument* に対応します。おのおのの標識は、2 バイトの符号付き整数として定義されます。次の値のいずれかに設定されます。

0 引き数が存在し、ヌルではない。

-1 引き数がヌルである。

関数が RETURNS NULL ON NULL INPUT を使用して定義されている場合は、UDF はヌル値かどうかをチェックする必要はありません。ただし、関数が CALLS ON NULL INPUT を使用して定義されている場合は、どの引き数も NULL にできるため、UDF ではヌルの入力があるかどうかをチェックする必要があります。注: これらのパラメーターは入力としてのみ扱われます。UDF によって行われたパラメーター値の変更は、DB2 UDB によって無視されます。

SQL-result-ind
この引き数は、DB2 UDB に戻る前に、UDF によって設定されます。データベースは、戻り値用の記憶域を提供します。この引き数は、2 バイトの符

号付き整数として定義されます。負の値に設定された場合は、データベースは、関数の結果をヌルとして解釈します。ゼロまたは正の値に設定された場合は、データベースは、*SQL-result* に戻された値を使用します。データベースは、戻り値標識用の記憶域を提供します。パラメーターはアドレスで渡されるので、アドレスは、標識の値が入る記憶域のアドレスになります。

SQL-state
この引き数は、SQLSTATE を表す CHAR(5) の値です。

このパラメーターは、'00000' に設定されたデータベースから渡され、関数の結果状態として関数によって設定されます。 通常、SQLSTATE は関数によって設定されませんが、以下のように、データベースにエラーまたは警告を送るのに使用することができます。

01Hxx 関数コードが警告状態を検出しました。これは SQL 警告になります。ここで、*xx* は、可能な文字ストリングのいずれかです。

38xxx 関数コードがエラー状態を検出しました。これは SQL エラーになります。ここで、*xxx* は、可能ないくつかのストリングのいずれかです。

関数が使用する有効な SQLSTATE の詳細については、SQL メッセージおよびコードを参照してください。

function-name
この引き数は、UDF を呼び出す前に、DB2 UDB によって設定されます。引き数は、関数の名前が入っている VARCHAR(139) 値です。関数コードがこの関数のために呼び出されます。

渡される関数名の形式は、以下のようになります。

```
<schema-name>.<function-name>
```

関数コードが複数の UDF 定義で使用され、どの定義が呼び出されるかをそのコードによって識別するときに、このパラメーターが役立ちます。注: このパラメーターは入力としてのみ扱われます。UDF によって行われたパラメーター値の変更は、DB2 UDB によって無視されます。

specific-name
この引き数は、UDF を呼び出す前に、DB2 UDB によって設定されます。引き数は、関数の特定の名前が入っている VARCHAR(128) 値です。関数コードがこの関数のために呼び出されます。

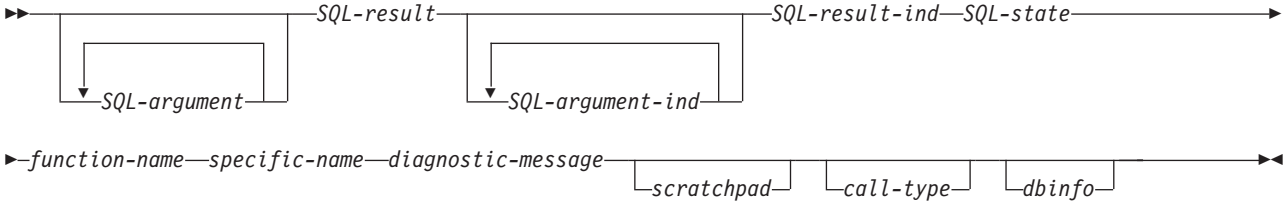
function-name と同様に、関数コードが複数の UDF 定義で使用され、どの定義が呼び出されるかをそのコードによって識別するときに、このパラメーターが役立ちます。 *specific-name* の詳細については、CREATE FUNCTION ステートメントを参照してください。 注: このパラメーターは入力としてのみ扱われます。UDF によって行われたパラメーター値の変更は、DB2 UDB によって無視されます。

diagnostic-message
この引き数は、UDF を呼び出す前に、DB2 UDB によって設定されます。この引き数は、SQLSTATE 警告またはエラーが UDF によって送られたときに、メッセージ・テキストを戻すのに UDF が使用する VARCHAR(70) 値です。

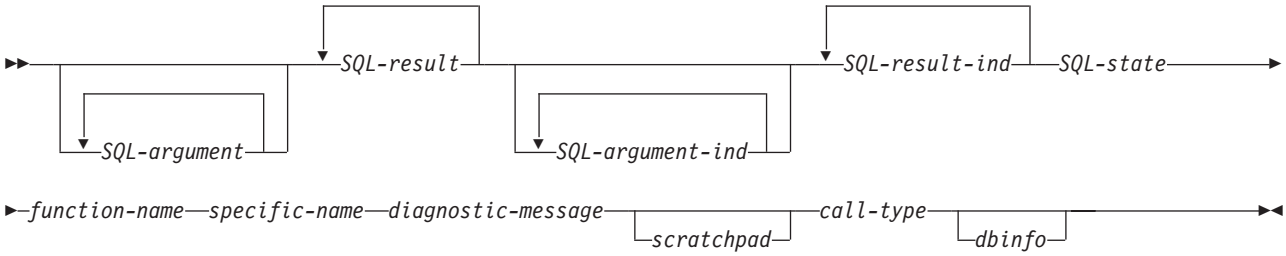
これは、UDF に入力があったときにデータベースによって初期設定され、記述情報が UDF によって設定されます。SQL-state パラメーターが UDF によって設定されていないかぎり、メッセージ・テキストは DB2 UDB によって無視されます。

パラメーター・スタイル DB2SQL: DB2SQL パラメーター・スタイルを使用すると、パラメーター・スタイル SQL の場合に渡されるのと同じように、同じパラメーターが同じ順序で、外部プログラムまたはサービス・プログラムに渡されます。ただし、DB2SQL では、追加のオプション・パラメーターも渡すことができます。UDF 定義に以下のオプション・パラメーターが複数指定された場合は、以下に定義された順序で UDF に渡されます。 共通のパラメーターについては、パラメーター・スタイル SQL を参照してください。 このパラメーター・スタイルは、スカラー UDF と表 UDF の両方で使用できます。

スカラー関数の場合、次のようになります。



表関数の場合、次のようになります。



scratchpad

この引き数は、UDF を呼び出す前に、DB2 UDB によって設定されます。これは、UDF の CREATE FUNCTION ステートメントが SCRATCHPAD キーワードを指定したときにのみ存在します。この引き数は、以下のエレメントをもつ構造になっています。

- スクラッチパッドの長さが入っている INTEGER。
- UDF への最初の呼び出しの前に、DB2 UDB によってすべて 2 進法の 0 に初期設定されたスクラッチパッド。

スクラッチパッドは、作業記憶域または永続記憶域として UDF によって使用されます。スクラッチパッドは、複数の UDF 呼び出しの間中、維持管理されるからです。

表関数の場合、CREATE FUNCTION で FINAL CALL が指定されていれば、スクラッチパッドは上記のように UDF に対する FIRST 呼び出しの前に初期設定されます。この呼び出しの後、スクラッチパッドの内容は全体として表関数の制御下になります。以降、DB2 はスクラッチパッドの内容を

調べたり変更したりすることはありません。スクラッチパッドは、呼び出しのたびに関数に渡されます。関数は再入可能にすることができ、DB2 は関数の状態情報をスクラッチパッドに保存します。

表関数に NO FINAL CALL が指定されたかまたは省略値とされた場合、スクラッチパッドは前述のように OPEN 呼び出しのたびごとに初期設定され、OPEN 呼び出しと OPEN 呼び出しの間、スクラッチパッドの内容は完全に表関数の制御下になります。これは、結合または副照会で使用される表関数の場合、非常に重要になります。複数の OPEN 呼び出しをまたがってスクラッチパッドの内容を維持管理する必要があるときは、CREATE FUNCTION ステートメントで FINAL CALL を指定する必要があります。FINAL CALL を指定すれば、通常の OPEN、FETCH、および CLOSE 呼び出しに加えて、スクラッチパッドの維持管理とリソースの解放のために、表関数が FIRST 呼び出しと FINAL 呼び出しも受け取ることになります。

call-type

この引き数は、UDF を呼び出す前に、DB2 UDB によって設定されます。スカラー関数の場合は、これは、UDF の CREATE FUNCTION ステートメントが FINAL CALL キーワードを指定したときにのみ存在しますが、表関数の場合は常に 存在します。これは*scratchpad* 引き数の後に続きます。スクラッチパッド引き数が存在しない場合は、*diagnostic-message* 引き数の後に続きます。この引き数は、INTEGER 値の形式をとります。

スカラー関数の場合、次のようになります。

-1 このステートメントの UDF への *最初の呼び出し* を示します。最初の呼び出しは、すべての SQL 引き数値が渡されるという意味で *通常の呼び出し* です。

0 *通常の呼び出し* を示します。(通常の入力引き数値のすべてが渡されます。)

1 *最終呼び出し* を示します。 *SQL-argument* または *SQL-argument-ind* の値は渡されません。 UDF は、SQL-result、SQL-result-ind 引き数、SQL-state、または diagnostic-message 引き数を使用して応答を戻すことはありません。これらの引き数はいずれも、UDF から戻るときに DB2 UDB によって無視されます。

表関数の場合、次のようになります。

-2 このステートメントの UDF への *最初の呼び出し* を示します。最初の呼び出しは、すべての SQL 引き数値が渡されるという意味で *通常の呼び出し* です。

-1 このステートメントの UDF への *OPEN 呼び出し* を示します。スクラッチパッドは、NO FINAL CALL が指定されている場合は初期設定されますが、それ以外の場合には必要ありません。すべての SQL 引き数値が渡されます。

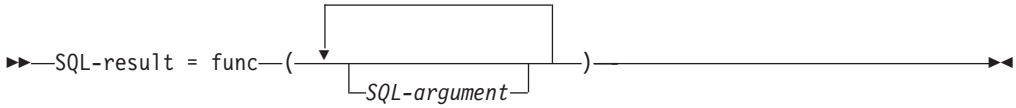
0 *FETCH 呼び出し* を示します。 DB2 は、表関数が、戻り値のセットを含む行か、または SQLSTATE 値 '02000' で示される「表の終わり」条件か、いずれかを戻すことを予期しています。

1 *CLOSE 呼び出し* を示します。 この呼び出しは OPEN 呼び出しと対応しており、外部 CLOSE 処理とリソース解放を実行するために使用できます。

2 *最終呼び出し* を示します。 *SQL-argument* または *SQL-argument-ind* の値は渡されません。 UDF は、SQL-result、SQL-result-ind 引き数、SQL-state、または diagnostic-message 引き数を使用して応答を戻すことはありません。これらの引き数はいずれも、UDF から戻るときに DB2 UDB によって無視されます。

dbinfo この引き数は、UDF を呼び出す前に、DB2 UDB によって設定されます。これは、UDF の CREATE FUNCTION ステートメントが DBINFO キーワードを指定したときにのみ存在します。この引き数は、その定義が `sqludf` 組み込みに入っている構造です。

パラメーター・スタイル GENERAL (または SIMPLE CALL): パラメーター・スタイル GENERAL を使用すると、パラメーターは、CREATE FUNCTION ステートメントで指定されたのと同じように、外部サービス・プログラムに渡されます。 このパラメーター・スタイルは、スカラー UDF でのみ、使用できます。 形式は以下のようになります。



SQL-argument

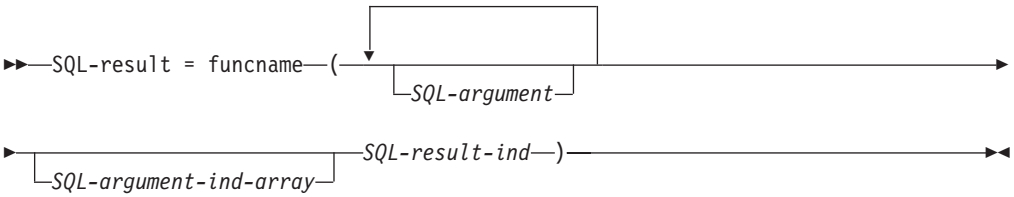
この引き数は、UDF を呼び出す前に、DB2 UDB によって設定されます。この値は *n* 回繰り返されます。ここで、*n* は、関数参照で指定された引き数の数です。これらの引き数のおのおのの値は、関数呼び出しで指定された式からとられます。これは、CREATE FUNCTION ステートメントの定義されたパラメーターのデータ・タイプの中で表されます。注: これらのパラメーターは入力としてのみ扱われます。UDF によって行われたパラメーター値の変更は、DB2 UDB によって無視されます。

SQL-result

この値は UDF によって戻されます。DB2 UDB は値をデータベース記憶域にコピーします。値を正しく戻すためには、関数コードは値を戻す関数でなければなりません。データベースは、CREATE FUNCTION ステートメントで指定された戻り値として定義された値だけをコピーします。CREATE FUNCTION ステートメントで CAST FROM 文節が使用されている場合は、DB2 UDB は、UDF が CAST FROM 文節で定義された値を戻すと想定し、CAST FROM 文節が使用されていない場合は、DB2 UDB は、UDF が RETURNS 文節で定義された値を戻すと想定します。

関数コードは値を戻す関数でなければならないという要件があるので、パラメーター・スタイル GENERAL で使用される関数コードはすべて、サービス・プログラムの中に組み込まなければなりません。

パラメーター・スタイル GENERAL WITH NULLS: パラメーター・スタイル GENERAL WITH NULLS は、スカラー UDF でのみ、使用できます。このパラメーター・スタイルを使用すると、パラメーターは、サービス・プログラムに、以下のように (指定された順序で) 渡されます。



SQL-argument

この引き数は、UDF を呼び出す前に、DB2 UDB によって設定されます。この値は *n* 回繰り返されます。ここで、*n* は、関数参照で指定された引き数の数です。これらの引き数のおのおのの値は、関数呼び出しで指定された式からとられます。これは、CREATE FUNCTION ステートメントの定義されたパラメーターのデータ・タイプの中で表されます。注: これらのパラメーターは入力としてのみ扱われます。UDF によって行われたパラメーター値の変更は、DB2 UDB によって無視されます。

SQL-argument-ind-array

この引き数は、UDF を呼び出す前に、DB2 UDB によって設定されます。これは、1 つまたは複数の *SQL-argument* がヌルかどうかを判断するために、UDF によって使用されます。 これは、2 バイトの符号付き整数 (標識) の配列です。 *n* 番目の配列引き数は、*n* 番目の *SQL-argument* に対応します。 配列の各項目は、以下の値のいずれかに設定されます。

0 引き数が存在し、ヌルではない。

-1 引き数がヌルである。

UDF は、ヌルの入力があるかどうかチェックします。注: このパラメーターは入力としてのみ扱われます。UDF によって行われたパラメーター値の変更は、DB2 UDB によって無視されます。

SQL-result-ind

この引き数は、DB2 UDB に戻る前に、UDF によって設定されます。データベースは、戻り値用の記憶域を提供します。この引き数は、2 バイトの符号付き整数として定義されます。負の値に設定された場合は、データベースは、関数の結果をヌルとして解釈します。ゼロまたは正の値に設定された場合は、データベースは、*SQL-result* に戻された値を使用します。データベースは、戻り値標識用の記憶域を提供します。パラメーターはアドレスで渡されるので、アドレスは、標識の値が入る記憶域のアドレスになります。

SQL-result

この値は UDF によって戻されます。DB2 UDB は値をデータベース記憶域にコピーします。値を正しく戻すためには、関数コードは値を戻す関数でなければなりません。データベースは、CREATE FUNCTION ステートメントで指定された戻り値として定義された値だけをコピーします。CREATE FUNCTION ステートメントで CAST FROM 文節が使用されている場合は、DB2 UDB は、UDF が CAST FROM 文節で定義された値を戻すと想定し、CAST FROM 文節が使用されていない場合は、DB2 UDB は、UDF が RETURNS 文節で定義された値を戻すと想定します。

関数コードは値を戻す関数でなければならないという要件があるので、パラメーター・スタイル GENERAL WITH NULLS で使用される関数コードはすべて、サービス・プログラムの中に組み込まなければなりません。

注:

1. CREATE FUNCTION ステートメントで指定された外部名は、引用符付きでも引用符なしでも指定することができます。名前が引用符付きでない場合は、それが保管される前に英大文字になり、引用符付きの場合は、指定されたとおりに保管されます。これは実際のプログラムの名前を付けるときに重要になります。なぜなら、データベースは、関数定義を使用して保管された名前に正確に一致する名前を持つプログラムを検索するからです。たとえば、次のように関数が作成された場合、

   ```
   CREATE FUNCTION X(INT) RETURNS INT
     LANGUAGE C
     EXTERNAL NAME 'MYLIB/MYPGM(MYENTRY)'
   ```

   そして、プログラムのソースが次のようなとき、

   ```
   void myentry(
          int*in
          int*out,
          .
          .
          . .
   ```

   データベースはこのエントリーを見つけることができません。なぜなら、このエントリーは英小文字の *myentry* であり、データベースは英大文字の *MYENTRY* を見つけるように指示されたからです。
2. C++ モジュールを持つサービス・プログラムについては、C++ ソース・コードで、*extern "C"* がプログラム機能定義の前にあることを確認してください。ない場合は、C++ コンパイラーは関数名の「ネーム・マングリング」を行い、データベースはその関数名を見つけることができません。

パラメーター・スタイル DB2GENERAL: パラメーター・スタイル DB2GENERAL は Java UDF によって使用されます。このパラメーター・スタイルの説明については、「IBM Developer Kit for Java」の Java SQL ルーチンを参照してください。

パラメーター・スタイル Java: Java パラメーター・スタイルは、SQLJ 第一部: SQL ルーチン標準で指定されているスタイルです。 このパラメーター・スタイルの説明については、「IBM Developer Kit for Java」の Java SQL ルーチンを参照してください。

表関数の考慮事項

外部表関数は、その関数が参照された SQL に対して表を引き渡す UDF です。表関数は、SELECT の FROM 文節においてのみ、有効です。表関数を使用する際には、以下の点にご注意ください。

- 表関数が表を引き渡す場合でも、DB2 と UDF との間の物理インターフェースは 1 度に 1 行ずつです。表関数に対してなされる呼び出しには、OPEN、FETCH、CLOSE、FIRST、および FINAL の 5 つのタイプがあります。 FIRST 呼び出しおよび FINAL 呼び出しが存在するかどうかは、ユーザーが UDF をどのように

定義するかによります。スカラー関数に対して使用できるのと同じ *call-type* メカニズムが、これらの呼び出しを識別するのに使用されます。

- DB2 とユーザー定義スカラー関数との間で使用される標準インターフェースは、表関数に合わせて拡張されます。 *SQL-result* 引き数は、表関数の場合は繰り返されます。各インスタンスが列に対応し、CREATE FUNCTION ステートメントの RETURNS TABLE 文節で定義された通りに戻されます。 *SQL-result-ind* 引き数も同様に繰り返され、各インスタンスは対応する *SQL-result* インスタンスに関連づけられます。
- 表関数の CREATE FUNCTION ステートメントの RETURNS 文節で定義される結果の列すべてが、戻される必要があるわけではありません。 CREATE FUNCTION の DBINFO キーワード、および対応する *dbinfo* 引き数により、特定の表関数参照に必要な列だけが戻されるような最適化が使用可能になります。
- 戻される個々の列値は、スカラー関数によって戻される値の形式に従います。
- 表関数の CREATE FUNCTION ステートメントには、CARDINALITY *n* 指定があります。この指定により、定義者は、DB2 最適化プログラムに結果の概算サイズを知らせることができ、関数が参照される時に最適化プログラムがより良い判断を行うことができます。表関数の CARDINALITY がどのように指定されていようと、無限基数を持つ関数 (すなわち、FETCH 呼び出しで必ず 1 つの行を戻す関数) の作成に対しては十分な注意を払ってください。 DB2 は、*end-of-table* 条件を、照会処理におけるきっかけとして予期しています。このため、「表の終わり」条件 (SQL-state の値 '02000') を戻すことのない表関数は、無限の処理ループを引き起こします。

表関数のエラー処理

表関数呼び出しのエラー処理は次のようになります。

1. FIRST 呼び出しが失敗した場合、もう呼び出しは行われません。
2. FIRST 呼び出しが成功した場合、ネストされた OPEN 呼び出し、FETCH 呼び出し、および CLOSE 呼び出しが行われ、そして必ず FINAL 呼び出しが行われます。
3. OPEN 呼び出しが失敗した場合、FETCH 呼び出しまたは CLOSE 呼び出しは行われません。
4. OPEN 呼び出しが成功した場合に、FETCH 呼び出しおよび CLOSE 呼び出しが行われます。
5. FETCH 呼び出しが失敗した場合、もう FETCH 呼び出しは行われませんが、CLOSE 呼び出しは行われます。

注: このモデルは、表 UDF の場合の通常のエラー処理を説明したものです。システム障害や通信の問題のイベントがある場合は、上記のエラー処理モデルで示した呼び出しが行われない可能性があります。

スカラー関数のエラー処理

FINAL CALL 指定を指定して定義されたスカラー UDF のエラー処理モデルは、次のようになります。

1. FIRST 呼び出しが失敗した場合、もう呼び出しは行われません。

2. FIRST 呼び出しが成功した場合、ステートメントの処理によって保証されたものとして NORMAL 呼び出しが行われ、そして必ず FINAL 呼び出しが行われます。
3. NORMAL 呼び出しが失敗した場合、もう NORMAL 呼び出しは行われませんが、FINAL 呼び出しは行われます (FINAL CALL をユーザーが指定した場合)。これは、FIRST 呼び出しでエラーが戻されたときは、FINAL 呼び出しが行われないため、UDF が戻りの前に終結処理をしなければならないことを意味します。

注: このモデルは、スカラー UDF の場合の通常のエラー処理を説明したものです。 システム障害や通信の問題のイベントがある場合は、上記のエラー処理モデルで示した呼び出しが行われない可能性があります。

関数を隔離または隔離解除にする

ユーザー定義関数 (UDF) を作成する時には、その UDF を隔離解除 UDF にするかどうか検討します。省略時では、UDF は隔離される UDF として作成されます。「隔離される」とは、データベースがその UDF を別のスレッドで実行しなければならないことを示します。複雑な UDF の場合、この分離は、固有の SQL カーソル名の生成などの問題が起きる可能性を避けるために意味があります。リソースの競合について留意する必要がないということは、省略時値に従って UDF を隔離される UDF として作成する、1 つの理由です。「NOT FENCED」オプションを指定して作成された UDF は、ユーザーが要求しているデータベースに対して、この UDF は、UDF を開始したスレッドと同じスレッドの中で実行できるということを示します。「隔離されない」はデータベースに対する提案であり、この指定がされてもデータベースは UDF を隔離される UDF と同じ方法で実行するよう決めることもできます。

```
CREATE FUNCTION QGPL.FENCED (parameter1 INTEGER)
RETURNS INTEGER LANGUAGE SQL
BEGIN
RETURN parameter1 * 3;
END;

CREATE FUNCTION QGPL.UNFENCED1 (parameter1 INTEGER)
RETURNS INTEGER LANGUAGE SQL NOT FENCED
-- UDF を作成し、NOT FENCED オプションを解してより高速な実行を要求します。
BEGIN
RETURN parameter1 * 3;
END;
```

UDF コードの例

以下の例は、SQL 関数と外部関数を使用して UDF コードをインプリメントする方法を示しています。

- 『例: 数の平方を求める UDF』
- 265 ページの『例: カウンター』
- 266 ページの『例: 天気の表関数』

例: 数の平方を求める UDF

注: コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

ある数の平方を戻す関数を作成するとします。照会ステートメントは以下のようになります。

```
SELECT SQUARE(myint) FROM mytable
```

以下の例は、UDF を定義するいくつかの方法を示しています。

- **SQL 関数を使用する**

  ```
  CREATE FUNCTION SQUARE( inval INT) RETURNS INT
  LANGUAGE SQL
  BEGIN
  RETURN(inval*inval);
  END
  ```

- **外部関数、パラメーター・スタイル SQL を使用する**

  CREATE FUNCTION ステートメント:

  ```
  CREATE FUNCTION SQUARE(INT) RETURNS INT CAST FROM FLOAT
  LANGUAGE C
  EXTERNAL NAME 'MYLIB/MATH(SQUARE)'
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  PARAMETER STYLE SQL
  ALLOW PARALLEL
  ```

  コード:

  ```
  void SQUARE(int *inval,
  double *outval,
  short *inind,
  short *outind,
  char *sqlstate,
  char *funcname,
  char *specname,
  char *msgtext)
  {
  if (*inind<0)
    *outind=-1;
  else
    {
     *outval=*inval;
     *outval=(*outval)*(*outval);
     *outind=0;
    }
  return;
  }
  ```

  デバッグできるように、外部サービス・プログラムを作成する:

  ```
  CRTCMOD MODULE(mylib/square) DBGVIEW(*SOURCE)
  CRTSRVPGM SRVPGM(mylib/math) MODULE(mylib/square)
  EXPORT(*ALL) ACTGRP(*CALLER)
  ```

- **外部関数、パラメーター・スタイル GENERAL を使用する**

  CREATE FUNCTION ステートメント:

  ```
  CREATE FUNCTION SQUARE(INT) RETURNS INT CAST FROM FLOAT
  LANGUAGE C
  EXTERNAL NAME 'MYLIB/MATH(SQUARE)'
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  PARAMETER STYLE GENERAL
  ALLOW PARALLEL
  ```

コード:

```
double SQUARE(int *inval)
{
  double outval;
  outval=*inval;
  outval=outval*outval;
return(outval);
}
```

デバッグできるように、外部サービス・プログラムを作成する:

```
CRTCMOD MODULE(mylib/square) DBGVIEW(*SOURCE)

   CRTSRVPGM SRVPGM(mylib/math) MODULE(mylib/square)
EXPORT(*ALL) ACTGRP(*CALLER)
```

例: カウンター

注: コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

SELECT ステートメントで、複数の行に単に番号を付けていきたいとします。したがって、数を増分しカウンターを戻す UDF を作成します。以下の例は、DB2 UDB の SQL パラメーター・スタイルとスクラッチパッドを使った外部関数を使用しています。

```
CREATE FUNCTION COUNTER()
       RETURNS INT
       SCRATCHPAD
       NOT DETERMINISTIC
       NO SQL
       NO EXTERNAL ACTION
       LANGUAGE C
       PARAMETER STYLE DB2SQL
       EXTERNAL NAME 'MYLIB/MATH(ctr)'
       DISALLOW PARALLEL

/* structure scr defines the passed scratchpad for the function "ctr" */
struct scr {
   long len;
   long countr;
   char not_used[96];
};

void ctr (
   long *out,                          /* output answer (counter) */
   short *outnull,                     /* output NULL indicator */
   char *sqlstate,                     /* SQL STATE */
   char *funcname,                     /* function name */
   char *specname,                     /* specific function name */
   char *mesgtext,                     /* message text insert */
   struct scr *scratchptr) {           /* scratch pad */

   *out = ++scratchptr->countr;      /* increment counter & copy out */
   *outnull = 0;
   return;
}
/* end of UDF : ctr */
```

この UDF の場合、以下のことにご注意ください。

- UDF は定義された入力 SQL 引き数がありませんが、値を戻します。
- UDF は、標準の 4 つの末尾引き数すなわち *SQL-state*、*function-name*、*specific-name*、および、 *message-text* の後に、スクラッチパッド入力引き数を付加します。
- 渡されるスクラッチパッドをマップするために、構造定義を組み込んでいます。

- 入力パラメーターは定義されていません。これは、コードに一致します。
- SCRATCHPAD のコードが作成されているので、DB2 UDB は、スクラッチパッド引き数の割り振り、適切な初期設定、および、受け渡しを行うことができます。
- この UDF は、SQL 入力引き数以外の引き数に依存するので (このケースにはありませんが)、NOT DETERMINISTIC を指定しています。
- DISALLOW PARALLEL を指定しているのは正しいやりかたです。この UDF が正しく働くためには、1 つのスクラッチパッドが必要だからです。

例: 天気の表関数

注: コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

以下は、米国内のさまざまな都市の天気情報を戻す表関数の例です。これらの都市の天気のデータは、プログラム例に含まれているコメントで示されている通り、外部ファイルから読み取られます。データには、都市名に続いてその都市の天気情報が入っています。このパターンが、他の都市についても繰り返されます。

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h> /* for use in compiling User Defined Function */


#define   SQL_NOTNULL   0   /* Nulls Allowed - Value is not Null */
#define   SQL_ISNULL   -1   /* Nulls Allowed - Value is Null */
#define   SQL_TYP_VARCHAR 448
#define   SQL_TYP_INTEGER 496
#define   SQL_TYP_FLOAT 480

/* Short and long city name structure */
typedef struct {
  char * city_short ;
  char * city_long ;
} city_area ;

/* Scratchpad data */ 1
/* Preserve information from one function call to the next call */
typedef struct {
  /* FILE * file_ptr; if you use weather data text file */
  int file_pos ;  /* if you use a weather data buffer */
} scratch_area ;

/* Field descriptor structure */
typedef struct {
  char fld_field[31] ;                    /* Field data */
  int  fld_ind ;            /* Field null indicator data */
  int  fld_type ;                         /* Field type */
  int  fld_length ;  /* Field length in the weather data */
  int  fld_offset ;  /* Field offset in the weather data */
} fld_desc ;

/* Short and long city name data */
city_area cities[] = {
  { "alb", "Albany, NY"                 },
  { "atl", "Atlanta, GA"                },
  .
  .
```

```
  .
  { "wbc", "Washington DC, DC"           },
  /* You may want to add more cities here */

  /* Do not forget a null termination */
  { ( char * ) 0, ( char * ) 0           }
} ;

/* Field descriptor data */
fld_desc fields[] = {
  { "", SQL_ISNULL, SQL_TYP_VARCHAR, 30,  0 }, /* city          */
  { "", SQL_ISNULL, SQL_TYP_INTEGER,  3,  2 }, /* temp_in_f     */
  { "", SQL_ISNULL, SQL_TYP_INTEGER,  3,  7 }, /* humidity      */
  { "", SQL_ISNULL, SQL_TYP_VARCHAR,  5, 13 }, /* wind          */
  { "", SQL_ISNULL, SQL_TYP_INTEGER,  3, 19 }, /* wind_velocity */
  { "", SQL_ISNULL, SQL_TYP_FLOAT,    5, 24 }, /* barometer     */
  { "", SQL_ISNULL, SQL_TYP_VARCHAR, 25, 30 }, /* forecast      */
  /* You may want to add more fields here */

  /* Do not forget a null termination */
  { ( char ) 0, 0, 0, 0, 0 }
} ;

/* Following is the weather data buffer for this example. You */
/* may want to keep the weather data in a separate text file. */
/* Uncomment the following fopen() statement.  Note that you  */
/* have to specify the full path name for this file.          */
char * weather_data[] = {
   "alb.forecast",
   "   34   28%    wnw   3  30.53 clear",
   "atl.forecast",
   "   46   89%   east  11  30.03 fog",
   .
   .
   .
   "wbc.forecast",
   "   38   96%    ene  16  30.31 light rain",
   /* You may want to add more weather data here */

   /* Do not forget a null termination */
   ( char * ) 0
} ;

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Find a full city name using a short name */
int get_name( char * short_name, char * long_name ) {

    int name_pos = 0 ;

    while ( cities[name_pos].city_short != ( char * ) 0 ) {
       if (strcmp(short_name, cities[name_pos].city_short) == 0) {
          strcpy( long_name, cities[name_pos].city_long ) ;
          /* A full city name found */
          return( 0 ) ;
       }
       name_pos++ ;
    }
    /* Could not find such city in the city data */
    strcpy( long_name, "Unknown City" ) ;
    return( -1 ) ;

}

#ifdef __cplusplus
```

```
extern "C"
#endif
/* This is a subroutine. */
/* Clean all field data and field null indicator data */
int clean_fields( int field_pos ) {

    while (fields[field_pos].fld_length !=0 ) {
       memset( fields[field_pos].fld_field, '\0', 31 ) ;
       fields[field_pos].fld_ind = SQL_ISNULL ;
       field_pos++ ;
    }
    return( 0 ) ;

}

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Fills all field data and field null indicator data ... */
/* ... from text weather data */
int get_value( char * value, int field_pos ) {

    fld_desc * field ;
    char field_buf[31] ;
    double * double_ptr ;
    int * int_ptr, buf_pos ;

    while ( fields[field_pos].fld_length != 0 ) {
       field = &fields[field_pos] ;
       memset( field_buf, '\0', 31 ) ;
       memcpy( field_buf,
               ( value + field->fld_offset ),
               field->fld_length ) ;
       buf_pos = field->fld_length ;
       while ( ( buf_pos > 0 ) &&
               ( field_buf[buf_pos] == ' ' ) )
          field_buf[buf_pos--] = '\0' ;
       buf_pos = 0 ;
       while ( ( buf_pos < field->fld_length ) &&
               ( field_buf[buf_pos] == ' ' ) )
          buf_pos++ ;
       if ( strlen( ( char * ) ( field_buf + buf_pos ) ) > 0 ||
            strcmp( ( char * ) ( field_buf + buf_pos ), "n/a") != 0 ) {
          field->fld_ind = SQL_NOTNULL ;

          /* Text to SQL type conversion */
          switch( field->fld_type ) {
            case SQL_TYP_VARCHAR:
                 strcpy( field->fld_field,
                         ( char * ) ( field_buf + buf_pos ) ) ;
                 break ;
            case SQL_TYP_INTEGER:
                 int_ptr = ( int * ) field->fld_field ;
                 *int_ptr = atoi( ( char * ) ( field_buf + buf_pos ) ) ;
                 break ;
            case SQL_TYP_FLOAT:
                 double_ptr = ( double * ) field->fld_field ;
                 *double_ptr = atof( ( char * ) ( field_buf + buf_pos ) ) ;
                 break ;
            /* You may want to add more text to SQL type conversion here */
          }

       }
       field_pos++ ;
    }
    return( 0 ) ;
```

```
}

#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN weather( /* Return row fields */
              SQLUDF_VARCHAR * city,
              SQLUDF_INTEGER * temp_in_f,
              SQLUDF_INTEGER * humidity,
              SQLUDF_VARCHAR * wind,
              SQLUDF_INTEGER * wind_velocity,
              SQLUDF_DOUBLE  * barometer,
              SQLUDF_VARCHAR * forecast,
              /* You may want to add more fields here */

              /* Return row field null indicators */
              SQLUDF_NULLIND * city_ind,
              SQLUDF_NULLIND * temp_in_f_ind,
              SQLUDF_NULLIND * humidity_ind,
              SQLUDF_NULLIND * wind_ind,
              SQLUDF_NULLIND * wind_velocity_ind,
              SQLUDF_NULLIND * barometer_ind,
              SQLUDF_NULLIND * forecast_ind,
              /* You may want to add more field indicators here */

              /* UDF always-present (trailing) input arguments */
              SQLUDF_TRAIL_ARGS_ALL
            ) {

  scratch_area * save_area ;
  char line_buf[81] ;
  int line_buf_pos ;

  /* SQLUDF_SCRAT is part of SQLUDF_TRAIL_ARGS_ALL */
  /* Preserve information from one function call to the next call */
  save_area = ( scratch_area * ) ( SQLUDF_SCRAT->data ) ;

  /* SQLUDF_CALLT is part of SQLUDF_TRAIL_ARGS_ALL */
  switch( SQLUDF_CALLT ) {

    /* First call UDF: Open table and fetch first row */
    case SQL_TF_OPEN:
         /* If you use a weather data text file specify full path */
         /* save_area->file_ptr = fopen("tblsrv.dat","r"); */
         save_area->file_pos = 0 ;
         break ;

    /* Normal call UDF: Fetch next row */ 2
    case SQL_TF_FETCH:
         /* If you use a weather data text file */
         /* memset(line_buf, '\0', 81); */
         /* if (fgets(line_buf, 80, save_area->file_ptr) == NULL) { */
         if ( weather_data[save_area->file_pos] == ( char * ) 0 ) {

            /* SQLUDF_STATE is part of SQLUDF_TRAIL_ARGS_ALL */
            strcpy( SQLUDF_STATE, "02000" ) ;

            break ;
         }
         memset( line_buf, '\0', 81 ) ;
         strcpy( line_buf, weather_data[save_area->file_pos] ) ;
         line_buf[3] = '\0' ;

         /* Clean all field data and field null indicator data */
         clean_fields( 0 ) ;
```

```
                        /* Fills city field null indicator data */
                        fields[0].fld_ind = SQL_NOTNULL ;

                        /* Find a full city name using a short name */
                        /* Fills city field data */
                        if ( get_name( line_buf, fields[0].fld_field ) == 0 ) {
                           save_area->file_pos++ ;
                           /* If you use a weather data text file */
                           /* memset(line_buf, '\0', 81); */
                           /* if (fgets(line_buf, 80, save_area->file_ptr) == NULL) { */
                           if ( weather_data[save_area->file_pos] == ( char * ) 0 ) {
                              /* SQLUDF_STATE is part of SQLUDF_TRAIL_ARGS_ALL */
                              strcpy( SQLUDF_STATE, "02000" ) ;
                              break ;
                           }
                           memset( line_buf, '\0', 81 ) ;
                           strcpy( line_buf, weather_data[save_area->file_pos] ) ;
                           line_buf_pos = strlen( line_buf ) ;
                           while ( line_buf_pos > 0 ) {
                              if ( line_buf[line_buf_pos] >= ' ' )
                                 line_buf_pos = 0 ;
                              else {
                                 line_buf[line_buf_pos] = '\0' ;
                                 line_buf_pos-- ;
                              }
                           }
                        }

                        /* Fills field data and field null indicator data ... */
                        /* ... for selected city from text weather data */
                        get_value( line_buf, 1 ) ;  /* Skips city field */

                        /* Builds return row fields */
                        strcpy( city, fields[0].fld_field ) ;
                        memcpy( (void *) temp_in_f,
                                fields[1].fld_field,
                                sizeof( SQLUDF_INTEGER ) ) ;
                        memcpy( (void *) humidity,
                                fields[2].fld_field,
                                sizeof( SQLUDF_INTEGER ) ) ;
                        strcpy( wind, fields[3].fld_field ) ;
                        memcpy( (void *) wind_velocity,
                                fields[4].fld_field,
                                sizeof( SQLUDF_INTEGER ) ) ;
                        memcpy( (void *) barometer,
                                fields[5].fld_field,
                                sizeof( SQLUDF_DOUBLE ) ) ;
                        strcpy( forecast, fields[6].fld_field ) ;

                        /* Builds return row field null indicators */
                        memcpy( (void *) city_ind,
                                &(fields[0].fld_ind),
                                sizeof( SQLUDF_NULLIND ) ) ;
                        memcpy( (void *) temp_in_f_ind,
                                &(fields[1].fld_ind),
                                sizeof( SQLUDF_NULLIND ) ) ;
                        memcpy( (void *) humidity_ind,
                                &(fields[2].fld_ind),
                                sizeof( SQLUDF_NULLIND ) ) ;
                        memcpy( (void *) wind_ind,
                                &(fields[3].fld_ind),
                                sizeof( SQLUDF_NULLIND ) ) ;
                        memcpy( (void *) wind_velocity_ind,
                                &(fields[4].fld_ind),
                                sizeof( SQLUDF_NULLIND ) ) ;
                        memcpy( (void *) barometer_ind,
                                &(fields[5].fld_ind),
```

```
                sizeof( SQLUDF_NULLIND ) ) ;
        memcpy( (void *) forecast_ind,
                &(fields[6].fld_ind),
                sizeof( SQLUDF_NULLIND ) ) ;

        /* Next city weather data */
        save_area->file_pos++ ;

        break ;

    /* Special last call UDF for cleanup (no real args!): Close table */ 3
    case SQL_TF_CLOSE:
        /* If you use a weather data text file */
        /* fclose(save_area->file_ptr); */
        /* save_area->file_ptr = NULL; */
        save_area->file_pos = 0 ;
        break ;

  }

}
```

この UDF コードに組み込まれた番号を参照しながら、以下のことにご注意ください。

1. スクラッチパッドが定義されます。 row 変数が OPEN 呼び出しの時に初期設定され、iptr 配列と nbr_rows 変数はオープン時に *mystery* 関数によって入力されます。
2. FETCH は行を指標として使用して iptr 配列をトラバースし、iptr の現行要素から関係のある値を out_c1、out_c2、および out_c3 の各結果ポインターによって示される位置に移動します。
3. 最後に、CLOSE が、OPEN によって獲得された記憶域およびスクラッチパッドがアンカーしていた記憶域を解放します。

以下に示すのは、この UDF の CREATE FUNCTION ステートメントです。

```
CREATE FUNCTION tfweather_u()
  RETURNS TABLE (CITY VARCHAR(25),
                 TEMP_IN_F INTEGER,
                 HUMIDITY INTEGER,
                 WIND VARCHAR(5),
                 WIND_VELOCITY INTEGER,
                 BAROMETER FLOAT,
                 FORECAST VARCHAR(25))
  SPECIFIC tfweather_u
  DISALLOW PARALLEL
  NOT FENCED
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  SCRATCHPAD
  NO FINAL CALL
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  EXTERNAL NAME 'LIB1/WEATHER(weather)';
```

このステートメントを参照しながら、以下のことにご注意ください。

- このステートメントはいかなる入力も取らず、7 桁の出力列を戻します。
- SCRATCHPAD が指定されているため、DB2 はスクラッチパッド引き数を割り振り、適切に初期設定し、渡します。

- NO FINAL CALL が指定されています。
- この関数は、SQL 入力引き数以外の引き数に依存するので、NOT DETERMINISTIC として指定されています。 つまり、この関数は場合によって異なり、内容は実行するたびに変わると想定されます。
- 表関数の場合は DISALLOW PARALLEL が必要です。
- CARDINALITY 100 は、戻されることが予期される行数の見積もりであり、DB2 最適化プログラムに対して提供されます。
- DBINFO は使用されず、この関数を参照している特定のステートメントが必要とする列だけが戻されるという最適化は、行われません。
- NOT NULL CALL が指定されているため、この UDF は、入力 SQL 引き数のいずれかが NULL の場合には呼び出されず、つまりこの条件を検査する必要はありません。

この表関数によって生成される行をすべて選択するには、次の照会を使用します。

```
SELECT *
    FROM TABLE (tfweather_u())x
```

第 14 章 動的 SQL アプリケーション

アプリケーションで動的 SQL を使用すると、プログラムの実行時に SQL ステートメントを定義して、実行させることができます。動的 SQL を使用するアプリケーションは、文字ストリングの形で SQL ステートメントを入力として受け取ります (または作成します)。アプリケーションは、どのようなタイプの SQL ステートメントが実行されるかを知る必要はありません。アプリケーションは次のことを行います。

- SQL ステートメントを作成し、あるいは入力として受け取る。
- その SQL ステートメントの実行の準備をする。
- そのステートメントを実行する。
- SQL 戻りコードを処理する。

対話式 SQL (『第 17 章 対話式 SQL の使用』を参照) は、動的 SQL プログラムの一例です。SQL ステートメントは、対話式 SQL により動的に処理されます。

注:

1. 実行時のオーバーヘッドは、動的 SQL を使用して処理されるステートメントの場合の方が静的 SQL ステートメントの場合よりも大きくなります。プログラムを単に実行するだけでなく、事前コンパイル、バインド、そして実行といった余分の処理が必要になるためです。したがって、動的 SQL の使用は、動的 SQL の融通性を必要とするアプリケーションだけに限定すべきです。その他のアプリケーションの場合には、通常の (静的) SQL ステートメントを使用してデータベースからデータをアクセスするようにしてください。
2. EXECUTE ステートメントまたは EXECUTE IMMEDIATE ステートメントを含んでいるプログラムが FOR READ ONLY 文節を使用してカーソルを読み取り専用にすると、カーソルの行を検索するためにブロック化が使用されるので、パフォーマンスが向上します。

   ALWBLK(*ALLREAD) CRTSQLxxx オプションを使用すると、FOR UPDATE OF を明示的にコーディングしていない、あるいはそのカーソルを参照する位置指定の削除または更新を持たないすべてのカーソルに対して FOR READ ONLY が暗黙に宣言されます。暗黙の FOR READ ONLY を持つカーソルは、上記 2 に説明した利点が得られます。

動的 SQL ステートメントの中には、アドレス変数の使用を必要とするものがあります。RPG for iSeries プログラムは、アドレス変数の処理のために PL/I、COBOL、C、または ILE RPG for iSeries の各プログラムの助けが必要です。

次の表は、DB2 UDB for iSeries でサポートされるすべてのステートメントと、これらのステートメントが動的アプリケーションで使用できるかどうかを示しています。

注: 次の表において、*動的 SQL* 列の番号は、次のページの注の番号に対応しています。

表 28. 動的アプリケーションの中で使用できる SQL ステートメントの一覧

SQL ステートメント	静的 SQL	動的 SQL
ALTER TABLE	Y	Y
BEGIN DECLARE SECTION	Y	N
CALL	Y	Y
CLOSE	Y	N
COMMENT ON	Y	Y
COMMIT	Y	Y
CONNECT	Y	N
CREATE ALIAS	Y	Y
CREATE DISTINCT TYPE	Y	Y
CREATE FUNCTION	Y	Y
CREATE INDEX	Y	Y
CREATE PROCEDURE	Y	Y
CREATE SCHEMA	Y	Y
CREATE TABLE	Y	Y
CREATE TRIGGER	Y	Y
CREATE VIEW	Y	Y
DECLARE CURSOR	Y	注 1 を参照
DECLARE GLOBAL TEMPORARY TABLE	Y	Y
DECLARE PROCEDURE	Y	N
DECLARE STATEMENT	Y	N
DECLARE VARIABLE	Y	N
DELETE	Y	Y
DESCRIBE	Y	注 2 を参照
DESCRIBE TABLE	Y	N
DISCONNECT	Y	N
DROP	Y	Y
END DECLARE SECTION	Y	N
EXECUTE	Y	注 3 を参照
EXECUTE IMMEDIATE	Y	注 4 を参照
FETCH	Y	N
FREE LOCATOR	Y	Y
GRANT	Y	Y
HOLD LOCATOR	Y	Y
INCLUDE	Y	N
INSERT	Y	Y
LABEL ON	Y	Y
LOCK TABLE	Y	Y
OPEN	Y	N
PREPARE	Y	注 5 を参照
RELEASE	Y	N

表 28. 動的アプリケーションの中で使用できる SQL ステートメントの一覧 (続き)

SQL ステートメント	静的 SQL	動的 SQL
RELEASE SAVEPOINT	Y	Y
RENAME	Y	Y
REVOKE	Y	Y
ROLLBACK	Y	Y
SAVEPOINT	Y	Y
SELECT INTO	Y	注 6 を参照
SELECT ステートメント	Y	注 7 を参照
SET CONNECTION	Y	N
SET OPTION	Y	注 8 を参照
SET PATH	Y	Y
SET RESULT SETS	Y	N
SET SCHEMA	Y	Y
SET TRANSACTION	Y	Y
SET 変数	Y	N
UPDATE	Y	Y
VALUES INTO	Y	N
WHENEVER	Y	N

注:

1. 準備の対象とすることはできませんが、実行に先立って関連する動的 SELECT ステートメントに対してカーソルを定義するために使用されます。
2. 準備の対象にすることはできませんが、準備されたステートメントの記述を戻すのに使用されます。
3. 準備の対象にすることはできませんが、準備された SQL ステートメントを実行するのに使用されます。 SQL ステートメントは、EXECUTE ステートメントを使用する前に PREPARE ステートメントによって準備されていなければなりません。 277 ページの『PREPARE と EXECUTE ステートメントの使用法』の項の PREPARE の例を参照してください。
4. 準備の対象にすることはできませんが、? パラメーター・マーカーを持たない動的ステートメント・ストリングとともに使用されます。EXECUTE IMMEDIATE ステートメントを使用すると、プログラムの実行時にステートメント・ストリングが動的に準備され、実行されます。 276 ページの『非 SELECT ステートメントの処理』 の項の EXECUTE IMMEDIATE の例を参照してください。
5. 準備の対象とすることはできませんが、実行に先立って動的 SELECT ステートメントの解析、最適化、およびセットアップのために使用されます。 276 ページの『非 SELECT ステートメントの処理』の項の PREPARE の例を参照してください。
6. SELECT INTO ステートメントは準備の対象にすることも、EXECUTE IMMEDIATE で使用することもできません。
7. EXECUTE または EXECUTE IMMEDIATE で使用することはできませんが、OPEN 用に作成し、使用することができます。

8. REXX プロシージャーを実行しているとき、または事前コンパイルされたプログラムでのみ使用することができます。

注: コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

動的 SQL アプリケーションの設計と実行

動的 SQL ステートメントは、事前コンパイル時に準備されないので、実行時に準備しなければなりません。そのために、動的 SQL ステートメントを出すためには、そのステートメントを EXECUTE ステートメントまたは EXECUTE IMMEDIATE ステートメントで使用しなければなりません。EXECUTE IMMEDIATE ステートメントを使用すると、その SQL ステートメントはプログラムの実行時に動的に準備され、実行されます。

動的 SQL ステートメントには、基本タイプが 2 つあります。それは SELECT ステートメントと非 SELECT ステートメントです。非 SELECT ステートメントとしては、DELETE、INSERT、および UPDATE などのステートメントがあります。

ODBC などのインターフェースを使用するクライアント・サーバー・アプリケーションは、通常、動的 SQL を使用してデータベースにアクセスします。iSeries Access を使用するクライアント・サーバー・アプリケーションの開発の詳細については、Programming for iSeries Access Express を参照してください。

非 SELECT ステートメントの処理

動的 SQL の非 SELECT ステートメントを作成するには、以下のステップに従ってください。

1. 作成したい SQL ステートメントが、動的に実行できるものであるかどうかを確認します (274 ページの表 28 を参照のこと)。
2. その SQL ステートメントを作成します。(対話式 SQL を使用すると、簡単な方法で SQL ステートメントを作成し、確認し、実行することができます。詳しい説明については、『第 17 章 対話式 SQL の使用』を参照してください。)

動的 SQL の非 SELECT ステートメントを実行するには、以下のステップに従ってください。

1. EXECUTE IMMEDIATE を使用して SQL ステートメントを実行するか、あるいは SQL ステートメントを PREPARE を用いて準備し、準備されたステートメントを EXECUTE を用いて実行します。
2. SQL 戻りコードが戻されたら、それを処理します。

動的 SQL の非 SELECT ステートメント (stmtstrg) を実行するアプリケーションの例を以下に示します。

```
EXEC SQL
 EXECUTE IMMEDIATE :stmtstrg;
```

動的 SQL ステートメントの CCSID

SQL ステートメントは通常ホスト変数です。ホスト変数の CCSID が、ステートメント・テキストの CCSID として使用されます。PL/I では、SQL ステートメントは

ストリング式でも構いません。この場合には、そのジョブの CCSID はステートメント・テキストの CCSID として使用されます。

動的 SQL ステートメントは、ステートメント・テキストの CCSID を使用して処理されます。この影響を最も受けるのは、可変文字です。たとえば、NOT 記号 (¬) は CCSID 500 では 'BA'X に置かれています。これは、ステートメント・テキストの CCSID が 500 である場合に、SQL が NOT 記号 (¬) が 'BA'X に置かれるよう要求したことを意味します。

これは、ステートメント・テキストの CCSID が 65535 である場合には、SQL は CCSID が 37 であるかのように可変文字を処理します。すなわち、SQL は NOT 記号 (¬) が '5F'X にあるものとして探します。

PREPARE と EXECUTE ステートメントの使用法

非 SELECT ステートメント にパラメーター・マーカーが含まれていなければ、EXECUTE IMMEDIATE ステートメントを使用して動的に実行することができます。しかし、非 SELECT ステートメントにパラメーター・マーカーが含まれている場合は、PREPARE と EXECUTE を使用して実行しなければなりません。

PREPARE ステートメントは非 SELECT ステートメント (たとえば、DELETE ステートメント) を作成し、ユーザーが選択した名前を与えます。 CRTSQLxxx コマンドで DLYPRP (*YES) の指定があるときは、PREPARE ステートメントで USING 文節の指定がある場合以外は、ステートメントが初めて EXECUTE または DESCRIBE ステートメントの中で使用されるときまでは準備は行われません。この例では、ステートメントは S1 の名前になっています。準備の終わったステートメントは、同じプログラムの中でパラメーター・マーカーに種々の値を与えて何回でも実行することができます。 次の例では、準備の終わったステートメントが複数回にわたって実行できます。

```
DSTRING = 'DELETE FROM CORPDATA.EMPLOYEE WHERE EMPNO = ?';

/* ? はパラメーター・マーカーであり、この値が
  ステートメントを実行するたびに代入される
  ホスト変数であることを示しています。*/

EXEC SQL PREPARE S1 FROM :DSTRING;

/*DSTRING は、PREPARE ステートメントで S1 の名前を付ける
  削除ステートメントです。*/

 DO UNTIL (EMP =0);
/* アプリケーション・プログラムは、表示装置から EMP の値を
  読み取ります。*/
   EXEC SQL
     EXECUTE S1 USING :EMP;


END;
```

上記の例のようなルーチンでは、ユーザーはパラメーター・マーカーの数とそのデータ・タイプを知っていなければなりません。これは、入力データを提供するホスト変数はプログラムの作成時に宣言されるからです。

注: アプリケーション・サーバーに対応する準備されたすべてのステートメントは、アプリケーション・サーバーへの接続が終るたびに消滅します。 接続は、

CONNECT (タイプ 1) ステートメント、DISCONNECT ステートメント、または正常な COMMIT が後続する RELEASE により終了します。

SELECT ステートメントの処理および SQLDA の使用

SELECT ステートメントには、基本タイプが 2 つあります。**固定リスト**と**可変リスト**です。

固定リスト SELECT ステートメントを処理するときは、SQLDA は必要ありません。

可変リスト SELECT ステートメントを処理するときは、最初に SQLDA 構造を宣言しなければなりません。 SQLDA は、アプリケーション・プログラムから SQL にホスト変数の入力値を渡したり、SQL から出力値を受け取るのに使用される制御ブロックです。さらに、SELECT リスト式に関する情報を、PREPARE または DESCRIBE ステートメントに入れて返すことができます。

固定リスト SELECT ステートメント

動的 SQL では、固定リスト SELECT ステートメントは、予測可能な数とタイプのデータを検索するために設計されたステートメントです。このようなステートメントを使用する場合には、検索したデータを入れるホスト変数を予測して定義することができるので、SQLDA は必要ありません。連続した複数の FETCH のどれからでも、直前の FETCH と同数の値が返され、これらの値は直前の返されたものと同じデータ形式になっています。ホスト変数の指定の仕方は、SQL アプリケーションの場合と同じです。

SQL がサポートする任意のアプリケーション・プログラムで、固定リスト動的 SELECT ステートメントを使用することができます。

固定リスト SELECT ステートメントを動的に実行するときは、アプリケーションは以下の処置を行う必要があります。

1. 入力 SQL ステートメントをホスト変数に入れる。
2. PREPARE ステートメントで動的 SQL ステートメントの妥当性を検査し、それを実行可能な形式に変換する。 CRTSQLxxx コマンドで DLYPRP (*YES) の指定があるときは、PREPARE ステートメントで USING 文節の指定がある場合以外は、ステートメントが初めて EXECUTE または DESCRIBE ステートメントの中で使用されるときまでは準備は行われません。
3. そのステートメント名に対しカーソルを宣言する。
4. カーソルをオープンする。
5. 変数の固定リストに行を FETCH する (次の節 可変リスト選択ステートメントで説明する可変リスト SELECT ステートメントを使用する場合のように記述域に FETCH するのではなく)。
6. データの終わりが現れたとき、カーソルをクローズする。
7. SQL 戻りコードが戻されたときは、それを処理する。

たとえば、次の通りです。

```
MOVE 'SELECT EMPNO, LASTNAME FROM CORPDATA.EMPLOYEE WHERE EMPNO>?'
TO DSTRING.
EXEC SQL
 PREPARE S2 FROM :DSTRING END-EXEC.

EXEC SQL
 DECLARE C2 CURSOR FOR S2 END-EXEC.

EXEC SQL
 OPEN C2 USING :EMP END-EXEC.

PERFORM FETCH-ROW UNTIL SQLCODE NOT=0.

EXEC SQL
 CLOSE C2 END-EXEC.
STOP-RUN.
FETCH-ROW.
EXEC SQL
 FETCH C2 INTO :EMP, :EMPNAME END-EXEC.
```

注: このような場合、SELECT ステートメントは常に、前に実行した固定リスト SELECT ステートメントと同じ数およびタイプのデータ項目を戻すので、SQL 記述域 (SQLDA) を使用する必要がないことに注意してください。

可変リスト選択ステートメント

動的 SQL では、可変リスト SELECT ステートメントは、返される実行結果の列の数および様式が予測できないステートメントです。すなわち、必要とする変数がいくつあるか、またどのようなデータ・タイプであるか分かっていない場合です。したがって、戻される結果列に入れるホスト変数を前もって定義することはできません。

注: REXX には、5.b、6、および 7 の各ステップは適用できません。

可変リスト SELECT ステートメントを受け入れるアプリケーションの場合には、そのプログラムの中で次のことを行わなければなりません。

1. 入力 SQL ステートメントをホスト変数に入れる。
2. PREPARE ステートメントで動的 SQL ステートメントの妥当性を検査し、それを実行可能な形式に変換する。 CRTSQLxxx コマンドで DLYPRP (*YES) の指定があるときは、PREPARE ステートメントで USING 文節の指定がある場合以外は、ステートメントが初めて EXECUTE または DESCRIBE ステートメントの中で使用されるときまでは準備は行われません。
3. そのステートメント名に対しカーソルを宣言する。
4. 動的 SELECT ステートメントの名前を含んでいるカーソル (ステップ 3 で宣言した) をオープンする。
5. DESCRIBE ステートメントを出して、結果の表の各列のタイプおよびサイズに関する情報を SQL に要求する。

   注:

   a. PREPARE ステートメントに INTO 文節を付けてコーディングすると、1 つのステートメントで PREPARE と DESCRIBE の機能を実行することもできます。

b. 検索された各列の記述を収容するのに十分なスペースが SQLDA にないときは、プログラムは、必要なスペース量を判別し、そのスペース量に見合う記憶域を獲得し、新しい SQLDA を作成して、DESCRIBE ステートメントを出し直さなければなりません。

6. 検索されたデータの 1 行分を収めるのに必要なだけの記憶域量を割り振る。
7. 検索された各データ項目をどこに入れるかを SQL に指示するために、記憶域アドレスを SQLDA (SQL 記述域) に入れる。
8. FETCH で行を検索する。
9. データの終わりが現れたとき、カーソルをクローズする。
10. SQL 戻りコードが戻されたら、それを処理する。

| これらのステップを実行する詳細については、285 ページの『SQLDA の記憶域を
| 割り振るための選択ステートメントの例』を参照してください。

SQL 記述域 (SQLDA)

SQLDA を使用すると、SQL ステートメントに関する情報を SQL とアプリケーションとの間で受け渡しすることができます。 SQLDA は、DESCRIBE ステートメントおよび DESCRIBE TABLE ステートメントを実行するのに必要で、また、PREPARE、OPEN、FETCH、CALL、および EXECUTE の各ステートメントでも使用することができます。

SQLDA の情報の意味はその用途によって異なります。 PREPARE と DESCRIBE では、SQLDA は準備されたステートメントに関する情報をアプリケーション・プログラムに与えます。 DESCRIBE TABLE では、SQLDA は表または視点の中の列に関する情報をアプリケーション・プログラムに提供します。OPEN、EXECUTE、CALL、および FETCH では、SQLDA はホスト変数に関する情報を提供します。たとえば、DESCRIBE ステートメントを使用して SQLDA に値を読み取り、その値を、ホスト変数のアドレスに基づいて変更し、そして FETCH ステートメントで再使用します。

同時に複数のカーソルのオープンができるアプリケーションの場合、各動的 SELECT ステートメントごとに 1 つずつ、SQLDA をコーディングすることができます。 詳しくは、「SQL 解説書」の SQLDA および SQLCA を参照してください。

SQLDA は、C、C++、COBOL、PL/I、REXX、および RPG で使用することができます。 RPG for iSeries には、ポインターを設定する手段がないので、SQLDA を RPG for iSeries プログラムの外部で PL/I、C、COBOL、または ILE RPG for iSeries の各プログラムによって設定しなければなりません。 そのプログラムが次に RPG for iSeries プログラムを呼び出す必要があります。

SQLDA の形式

SQLDA は、4 つの変数の後に 6 つの変数 (まとめて SQLVAR と呼びます) を任意の回数だけ繰り返す形をとります。

注: REXX の SQLDA は異なります。詳しくは*ホスト言語での SQL プログラミング* のREXX アプリケーションにおける SQL ステートメントのコーディングを参照してください。

SQLDA が OPEN、FETCH、CALL、および EXECUTE で使用されるときは、1 つの SQLVAR により 1 つのホスト変数が記述されます。

SQLDA のフィールドは次の通りです。

SQLDAID

SQLDAID は記憶ダンプ用の「目印」として使用されます。 これは、PREPARE または DESCRIBE ステートメントで SQLDA が使用された後で、'SQLDA' の値をとる 8 文字のストリングです。 この変数は、FETCH、OPEN、CALL または EXECUTE では使用されません。

7 番目のバイトは、各列に複数の SQLVAR 記入項目が必要かどうかを判断するために使用されます。 LOB 列または特殊タイプの列がある場合は、複数の SQLVAR 項目が必要です。 LOB も特殊タイプもない場合は、このフラグはブランクに設定されます。

SQLDAID は、REXXには適用できません。

SQLDABC

SQLDABC は、SQLDA の長さを示します。これは PREPARE または DESCRIBE ステートメントで SQLDA が使用された後で、SQLN*LENGTH(SQLVAR) + 16 の値をとる 4 バイトの整数です。FETCH、OPEN、CALL、または EXECUTE で使用される前の SQLDABC の値は、SQLN*LENGTH(SQLVAR) + 16 に等しいか、それより大きくなければなりません。

SQLABC は、REXX には適用できません。

SQLN SQLN は、SQLVAR が現れる総数を示す 2 バイトの整数です。これは、SQL ステートメントで使用される前に、0 かそれより大きい値にセットしておかなければなりません。

SQLN は REXX には適用できません。

SQLD SQLD は、SQLVAR が現れる回数 (すなわち SQLDA によって記述されるホスト変数または列の数) を示す、2 バイトの整数です。このフィールドは、SQL によって DESCRIBE または PREPARE ステートメントでセットされます。他のステートメントでは、このフィールドは、使用される前に 0 かそれより大きい値でしかも SQLN かそれより小さい値にセットしておかなければなりません。

SQLVAR

値のこのグループは、ホスト変数または列ごとに 1 回ずつ繰り返されます。これらの変数は、SQL によって DESCRIBE または PREPARE ステートメントでセットされます。 他のステートメントでは、これらの変数は使用される前にセットしなければなりません。 これらの変数の定義は下記のとおりです。

SQLTYPE

SQLTYPE は、 283 ページの表 29 に示すように、ホスト変数または列のデータ・タイプを示す 2 バイトの整数です。 SQLTYPE の値が奇数のときは、ホスト変数に SQLIND によってアドレス指定される関連する標識変数があることを示します。

SQLLEN

SQLLEN は、図10-2 に示すホスト変数または列の長さ属性を示す 2 バイトの整変数です。

SQLRES

SQLRES は、境界合わせの目的で予約された 12 バイトの区域です。OS/400 では、ポインターは 4 ワード境界上になければならない ことに注意してください。

SQLRES は、REXX には適用できません。

SQLDATA

SQLDATA は、OPEN、FETCH、CALL、および EXECUTE で SQLDA が使用されたときホスト変数のアドレスを示す 16 バイトのポインター変数です。

SQLDA が PREPARE および DESCRIBE で使用されるときは、この区域は次の情報でオーバーレイされます。

文字フィールドまたはグラフィック・フィールドの CCSID が、SQLDATA の第 3 バイトと第 4 バイトに入ります。 BIT データの場合、CCSID は 65535 です。 REXX では、変数 SQLCCSID に CCSID が戻されます。

SQLIND

SQLIND は、OPEN、FETCH、CALL、および EXECUTE で SQLDA が使用されるときヌルか非ヌルかを示すために使用される短精度整数のホスト変数のアドレスを示す 16 バイトのポインターです。値が負のときは、ヌルを示し、値が負でないときは、ヌルでないことを示します。このポインターは、SQLTYPE の値が奇数のときだけ使用されます。

SQLDA が PREPARE および DESCRIBE で使用されるときは、この区域は将来の使用に備えて予約されます。

SQLNAME

SQLNAME は、最大長が 30 の可変長文字変数です。 PREPARE または DESCRIBE の実行後、この変数には、選択された列の名前、ラベル、またはシステム列名が入ります。OPEN、FETCH、EXECUTE、または CALL では、この変数は文字ストリングの CCSID を渡すために使用できます。 CCSID は文字ホスト変数およびグラフィック・ホスト変数の場合に、渡すことができます。

入力 SQLDA の SQLVAR 配列項目の中の SQLNAME フィールドは、次のようにセットして CCSID を指定することができます。

データ・タイプ	サブタイプ	SQLNAME の長さ	SQLNAME バイト 1 & 2	SQLNAME バイト 3 & 4
文字	SBCS	8	X'0000'	CCSID
文字	MIXED	8	X'0000'	CCSID
文字	BIT	8	X'0000'	X'FFFF'
GRAPHIC	適用外	8	X'0000'	CCSID
その他のデータ・タイプ	適用外	適用外	適用外	適用外

注: SQLNAME フィールドは CCSID を置き換えることだけを目的としているので、十分に注意してください。省略時値を使用するアプリケーションは、CCSID 情報を渡す必要はありません。CCSID を渡さないときは、ジョブの省略時値の CCSID が使用されます。

グラフィック・ホスト変数の省略時値は、ジョブの CCSID に対応する 2 バイトの CCSID です。対応する 2 バイトの CCSID がないときは、65535 が使用されます。

表 29. PREPARE、DESCRIBE、FETCH、OPEN、CALL、または EXECUTE の場合の SQLTYPE 値と SQLLEN 値

	PREPARE と DESCRIBE の場合		FETCH、OPEN、CALL、および EXECUTE の場合	
SQLTYPE	**列のデータ・タイプ**	**SQLLEN**	**ホスト変数の DATA TYPE**	**SQLLEN**
384/385	日付	10	日付の固定長文字ストリング表現	ホスト変数の長さ属性
388/389	時間	8	時刻の固定長文字ストリング表現	ホスト変数の長さ属性
392/393	タイム・スタンプ	26	タイム・スタンプの固定長文字ストリング表現	ホスト変数の長さ属性
396/397	データ・リンク (注 1 を参照)	列の長さ属性	適用外	適用外
400/401	適用外	適用外	NUL 終了グラフィック・ストリング	ホスト変数の長さ属性
392/393	タイム・スタンプ	26	タイム・スタンプの固定長文字ストリング表現	ホスト変数の長さ属性
404/405	BLOB	0 (注 2 を参照)	BLOB	使用されません。(注 2 を参照)
408/409	CLOB	0 (注 2 を参照)	CLOB	使用されません。(注 2 を参照)
412/413	DBCLOB	0 (注 2 を参照)	DBCLOB	使用されません。(注 2 を参照)
452/453	固定長文字ストリング	列の長さ属性	固定長文字ストリング	ホスト変数の長さ属性
456/457	長い可変長文字ストリング	列の長さ属性	長い可変長文字ストリング	ホスト変数の長さ属性
460/461	適用外	適用外	NUL 終了文字ストリング	ホスト変数の長さ属性
464/465	可変長グラフィック・ストリング	列の長さ属性	可変長グラフィック・ストリング	ホスト変数の長さ属性
468/469	固定長グラフィック・ストリング	列の長さ属性	固定長グラフィック・ストリング	ホスト変数の長さ属性
472/473	長い可変長グラフィック・ストリング	列の長さ属性	長い可変長グラフィック・ストリング	ホスト変数の長さ属性
476/477	適用外	適用外	PASCAL L ストリング	ホスト変数の長さ属性

表 29. PREPARE、DESCRIBE、FETCH、OPEN、CALL、または EXECUTE の場合の SQLTYPE 値と SQLLEN 値 (続き)

SQLTYPE	PREPARE と DESCRIBE の場合		FETCH、OPEN、CALL、および EXECUTE の場合	
	列のデータ・タイプ	SQLLEN	ホスト変数の DATA TYPE	SQLLEN
480/481	浮動小数点	単精度では 4、倍精度では 8	浮動小数点	単精度では 4、倍精度では 8
484/485	パック 10 進数	バイト 1 には精度、バイト 2 には位取り	パック 10 進数	バイト 1 には精度、バイト 2 には位取り
488/489	ゾーン 10 進数	バイト 1 には精度、バイト 2 には位取り	ゾーン 10 進数	バイト 1 には精度、バイト 2 には位取り
492/493	大きい整数	8	大きい整数	8
496/497	長精度整数	4 (注 3 を参照)	長精度整数	4
500/501	短精度整数	2 (注 3 を参照)	短精度整数	2
504/505	適用外	適用外	DISPLAY SIGN LEADING SEPARATE	バイト 1 には精度、バイト 2 には位取り
904/905	適用外	適用外	ROWID	40
916/917	適用外	適用外	BLOB ファイル参照変数	267
920/921	適用外	適用外	CLOB ファイル参照変数	267
924/925	適用外	適用外	DBCLOB ファイル参照変数	267
960/961	適用外	適用外	BLOB ロケーター	4
964/965	適用外	適用外	CLOB ロケーター	4
968/969	適用外	適用外	DBCLOB ロケーター	4

注:

1. データ・リンク・データ・タイプは DESCRIBE TABLE にのみ戻されます。
2. 2 次 SQLVAR 内の len.sqllonglen フィールドには、列の長さ属性が入ります。
3. 長精度と短精度の 2 進数は、長さを 2 または 4 として SQL の記述子域 (SQLDA) に表すことができます。また、バイト 1 を精度に、バイト 2 をスケールにして表すこともできます。最初のバイトが X'00' より大きい場合は、精度とスケールを表します。 大きい整数では精度とスケールは使えません。 SQLDA で長さを 8 と定義します。

SQLVAR2

これは拡張 SQLVAR 構造で、3 つのフィールドがあります。拡張 SQLVAR は、結果に特殊タイプ列または LOB 列がある場合に、結果のすべての列に必要です。特殊タイプの場合は、特殊タイプの名前が入ります。LOB の場合は、ホスト変数の長さ属性および実際の長さが入っているバッファーへのポインターが入ります。 ロケーターが LOB を表すために使用されている場合は、これらの記入項目は必要ありません。 必要な拡張 SQLVAR の発生の数は、SQLDA が用意されているステートメント、およ

び、記述される列またはパラメーターのデータ・タイプに依存します。 SQLDAID の 7 番目のバイトは、必ず、必要な SQLVAR のセットの数に設定されます。

SQLD が SQLVAR の十分な発生数に設定されていない場合は、

- SQLD は、すべてのセットに必要な SQLVAR 発生の合計数に設定されます。
- 少なくとも基本 SQLVAR 記入項目用に十分な数が指定された場合は、+237 警告が SQLCA の SQLCODE フィールドに戻されます。基本 SQLVAR 記入項目は戻されますが、拡張 SQLVAR は戻されません。
- 基本 SQLVAR 記入項目用にさえも十分な SQLVAR が指定されなかった場合は、+239 警告が SQLCA の SQLCODE フィールドに戻されます。 SQLVAR 記入項目は戻されません。

SQLLONGLEN

SQLLONGLEN は、LOB (BLOB、CLOB、または DBCLOB) ホスト変数または列の長さ属性を示す 4 バイトの整変数です。

SQLDATALEN

SQLDATALEN は 16 バイトのポインター変数で、ホスト変数の長さのアドレスを指定します。 この変数は、LOB (BLOB、CLOB、および DBCLOB) ホスト変数にのみ使用されます。 これは、DESCRIBE または PREPARE では使用されません。

このフィールドがヌルの場合は、データの実際の長さがデータの始まりの直前の 4 バイトに保管され、SQLDATA は、フィールド長の最初のバイトを指します。 長さは、BLOB または CLOB のバイト数、および、DBCLOB の文字数を示します。

このフィールドがヌルでない場合は、ここには、一致する基本 SQLVAR の中の SQLDATA フィールドで指し示されるバッファーにあるデータの実際の長さのバイト数 (DBCLOB の場合も含む) が入っている 4 バイトの長さのバッファーを指すポインターが入ります。

SQLDATATYPE_NAME

SQLDATATYPE_NAME は、最大長が 30 の可変長文字変数です。これは、DESCRIBE または PREPARE でのみ、使用されます。 この変数は、以下のいずれかに設定されます。

- 特殊タイプ列の場合は、データベース・マネージャーがこれを完全修飾特殊タイプ名に設定します。 修飾名が 30 バイトより長い場合は、切り捨てられます。
- ラベルの場合は、データベース・マネージャーは、ここに、ラベルの最初の 20 バイトを設定します。
- 列名の場合は、データベース・マネージャーは、ここに、列名を設定します。

SQLDA の記憶域を割り振るための選択ステートメントの例

アプリケーションが、動的 SELECT ステートメント (1 度使用して次に使用するまでに変更される) をハンドルできることが必要だとします。このステートメントは

表示装置から読み取られ、別のアプリケーションから渡されるか、またはユーザーのアプリケーションの進行中に作成されます。すなわち、このステートメントが毎回戻そうとする内容は、正確にはわかりません。アプリケーションは、実行前には未知であるデータ・タイプを持つ、さまざまな数の結果の列をハンドルできることが必要です。

たとえば、以下のようなステートメントを処理しなければなりません。

```
SELECT WORKDEPT, PHONENO
  FROM CORPDATA.EMPLOYEE
  WHERE LASTNAME = 'PARKER'
```

注: この SELECT ステートメントには INTO 文節がありません。 動的 SELECT ステートメントには、唯一の行を戻す場合でも、INTO 文節を入れてはなりません。

ステートメントは、ホスト変数に割り当てられます。そしてそのホスト変数 (この例では DSTRING という名前) が、次に示すように PREPARE ステートメントを使用して処理されます。

```
EXEC SQL
PREPARE S1 FROM :DSTRING;
```

次に、結果の列の数とデータ・タイプを判別する必要があります。これを行うには、SQLDA が必要です。

SQLDA を定義する最初のステップは、SQLDA 用に記憶域を割り振ることです。(REXX では、記憶域の割り振りは必要ありません。) 記憶域獲得の方法は言語によって異なります。 SQLDA は 16 バイト境界上に割り振られなければなりません。SQLDA は、長さが 16 バイトの固定長の見出しから成ります。 この見出しの後に、可変長の配列セクション n (SQLVAR) が続き、配列の各要素は 80 バイトの長さになっています。

割り振りを必要とする記憶域量は、SQLVAR 配列に入れたい要素の数によって決まります。 選択する各列は、SQLVAR 配列要素が対応していなければなりません。したがって、割り振りを必要とする SQLVAR 配列要素の数は、SELECT ステートメントにリストされる列の数によって決まります。 しかし、SELECT ステートメントは実行時に指定されるので、アクセスされる列の数を前もって知ることはできません。 このため、列の数を見積もる必要があります。 この例で、単一の SELECT ステートメントによってアクセスされる列数は、最高 20 個までとします。 この例では、SQLVAR 配列は、選択リストの各項目が SQLVAR 内に対応する項目を必ず持つように、ディメンション 20 でなければなりません。これにより、SQLDA の合計サイズは 20 x 80、または 1600 に合計用の 16 バイトを加えた 1616 バイトになります。

SQLDA 用に見積もった十分なスペースを割り振った後、SQLDA の SQLN フィールドを SQLVAR 配列要素と同じ数にセットする必要があります (この例では 20)。

記憶域の割り振りとサイズの初期設定が終わったら、次に DESCRIBE ステートメントを出すことができます。

```
EXEC SQL
DESCRIBE S1 INTO :SQLDA;
```

DESCRIBE ステートメントが実行されると、SQL はユーザーのステートメントの選択リストに関する情報を示す値を SQLDA に入れます。次の表は、DESCRIBE の実行後の SQLDA の内容を示しています。この文脈において意味のある項目のみを示します。

SQLDA ヘッダーには次の値が入ります。

表 30. SQLDA ヘッダー

記述	値
SQLAID	'SQLDA'
SQLDABC	1616
SQLN	20
SQLD	2

SQLDAID は、DESCRIBE の実行時に SQL によって初期設定される識別子フィールドです。 SQLDABC はバイト数、すなわち、SQLDA のサイズです。 SQLDA ヘッダーには、記述される SELECT ステートメントの結果表にある各列ごとに 1 つずつ、2 つの SELECT 構造が続きます。

表 31. SQLVAR 要素 1

記述	値
SQLTYPE	453
SQLLEN	3
SQLDATA (3:4)	37
SQLNAME	8 WORKDEPT

表 32. SQLVAR 要素 2

記述	値
SQLTYPE	453
SQLLEN	4
SQLDATA(3:4)	37
SQLNAME	7 PHONENO

SQLDA が、記述された SQLVAR 要素を収容するだけの大きさでないときは、ユーザーのプログラムで SQLN の値を変更する必要が起こることがあります。たとえば、見積もりの最大数である 20 列ではなく、SELECT ステートメントが 27 を戻したとします。 SQLVAR は割り振りスペースが許容するよりも多くの要素を必要とするため、SQL はこの選択リストを記述することができません。代わりに、SQL は、SELECT ステートメントに指定されている実際の列の数に SQLD をセットし、構造の残り部分は無視されます。 したがって、DESCRIBE の実行後は、SQLN 値を SQLD 値と比較するようにしてください。 SQLD の値が SQLN の値より大きければ、次の例のように、SQLD の値に基づいて SQLDA をもっと大きく割り振ってから、DESCRIBE を再度実行してください。

```
EXEC SQL
    DESCRIBE S1 INTO :SQLDA;
IF SQLN <= SQLD THEN
DO;
```

```
/*SQLD の値を使用して、より大きな SQLDA を割り振ります。*/
/*SQLN をより大きな値にリセットします。*/

EXEC SQL
    DESCRIBE S1 INTO :SQLDA;
END;
```

非 SELECT ステートメントで DESCRIBE を使用する場合は、SQL は SQLD を 0 にセットします。したがって、SELECT ステートメントと非 SELECT ステートメントの両方を処理するプログラムを設計するときは、SELECT ステートメントかどうかを示すために各ステートメントについて、作成後に記述することができます。この例は、SELECT ステートメントだけを処理するように設計されています。したがって SQLD 値は検査されません。

次に、ユーザー・プログラムは、成功した DESCRIBE から戻された SQLVAR の要素を分析しなければなりません。選択リストの最初の項目は WORKDEPT です。DESCRIBE は、式のデータ・タイプと、ヌルが使用可能かどうかを示す値を、SQLTYPE フィールドに戻します (283 ページの表 29を参照)。

この例では、SQL は、SQLVAR 要素 1 の SQLTYPE を 453 にセットします。これは、WORKDEPT が固定長文字ストリングの結果の列であり、その列にヌル値が許されることを指定しています。

SQL は、SQLLEN を列の長さにセットします。 WORKDEPT のデータ・タイプは CHAR であるので、SQL は SQLLEN を文字列の長さに等しくなるようセットします。 WORKDEPT の場合は、その長さは 3 です。したがって、SELECT ステートメントを後で実行させるときは、CHAR(3) ストリングが収容できるだけの大きさの記憶域が必要になります。

WORKDEPT のデータ・タイプは CHAR FOR SBCS DATA であるので、SQLDATA の最初の 4 バイトは、文字列の CCSID がセットされています。

SQLVAR 要素の最後のフィールドは、SQLNAME と名付けた可変長文字ストリングです。 SQLNAME の最初の 2 バイトには、文字データの長さが入ります。通常、この文字データ自体が、SELECT ステートメントで使用される列の名前です (この例では WORKDEPT)。これに対する例外は、関数 (たとえば、SUM(SALARY))、式 (たとえば、A+B-C)、および定数のように、名前のない選択リスト項目です。 このような場合には、SQLNAME は空のストリングになります。SQLNAME には、名前ではなくラベルを含めることもできます。 PREPARE および DESCRIBE ステートメントに関連して使用されるパラメーターの 1 つに、USING 文節があります。これは次のように指定できます。

```
EXEC SQL
    DESCRIBE S1 INTO:SQLDA
        USING LABELS;
```

指定の意味は次のようになります。

NAMES (あるいは USING パラメーターを完全に省略する場合)
列名だけが SQLNAME フィールドに入ります。

SYSTEM NAMES
システム列名だけが SQLNAME フィールドに入ります。

LABELS
SQL ステートメントでリストした列に付けられたラベルだけがここに入ります。

ANY ラベルが付いた列について SQLNAME フィールドにラベルが入ります。その他の列については、列名が入ります。

BOTH 名前とラベルの両方が、それぞれの長さでこのフィールドに入ります。 要素の数が 2 倍になるので、SQLVAR 配列のサイズも 2 倍にすることを忘れないでください。

ALL 列名、システム、およびシステムの列名がそれぞれの長さでこのフィールドに入ります。 SQLVAR 配列のサイズを 3 倍にすることを忘れないでください。

USING オプションについて詳しくは、*SQL 解説書* の DESCRIBE ステートメントおよび SQLDA セクションを参照してください。

この例では、2 番目の SQLVAR 要素には、選択で使用されている 2 番目の列、すなわち、PHONENO に関する情報が入ります。 SQLTYPE の 453 というコードは、PHONENO が CHAR 列であることを示しています。SQLLEN は 4 にセットされます。

ここで、SELECT ステートメントを実行する時に値を検索するために、SQLDA を使用するようセットアップする必要があります。

DESCRIBE の結果を分析した後で、ユーザーは SELECT ステートメントの結果を入れる変数の記憶域を割り振ることができます。 WORKDEPT の場合は、長さが 3 の文字フィールドを割り振らなければなりません。PHONENO の場合は、長さが 4 の文字フィールドを割り振らなければなりません。 これらの結果の両方がヌル値である可能性があるため、フィールドごとに標識変数も割り振る必要があります。

記憶域の割り振りが終わったら、次に、SQLDATA と SQLIND を割り振られた記憶域を指すようにセットしなければなりません。 SQLVAR 配列の各要素について、SQLDATA はその結果の値を入れるべき場所を指し示します。 SQLIND はヌル標識の値を入れるべき場所を指し示します。次の表は、この時点で構造がどのようになるかを示します。この文脈において意味のある項目のみを示します。

表 33. SQLDA ヘッダー

記述	値
SQLAID	'SQLDA'
SQLDABC	1616
SQLN	20
SQLD	2

表 34. SQLVAR 要素 1

記述	値
SQLTYPE	453
SQLLEN	3
SQLDATA	CHAR(3) の結果用の区域へのポインター

表 34. SQLVAR 要素 1 (続き)

記述	値
SQLIND	結果の列用の 2 バイトの整数標識へのポインター

表 35. SQLVAR 要素 2

記述	値
SQLTYPE	453
SQLLEN	4
SQLDATA	CHAR(4) の結果用の区域へのポインター
SQLIND	結果の列用の 2 バイトの整数標識へのポインター

これまでの処理によって、SELECT ステートメントの結果を取り出す準備ができました。動的に定義される SELECT ステートメントには、INTO ステートメントを入れることはできません。したがって、動的に定義される SELECT ステートメントはすべて、カーソルを使用しなければなりません。動的に定義される SELECT ステートメントでは、特別な形式の DECLARE、OPEN、および FETCH が使用されます。

例示したステートメントのための DECLARE ステートメントは次のようになります。

```
EXEC SQL DECLARE C1 CURSOR FOR S1;
```

この場合の唯一の相違は、SELECT ステートメント自体の代わりに準備された SELECT ステートメントの名前 (S1) が使用されていることです。結果の行の実際の検索は、次のようにして行われます。

```
EXEC SQL
    OPEN C1;
EXEC SQL
    FETCH C1 USING DESCRIPTOR :SQLDA;
DO WHILE (SQLCODE = 0);
/*SQLDATA によって指し示される結果を処理します*/
EXEC SQL
    FETCH C1 USING DESCRIPTOR :SQLDA;
END;
EXEC SQL
    CLOSE C1;
```

カーソルがオープンされます。次に SELECT からの結果の列が、FETCH ステートメントを使用して 1 度に 1 列ずつ戻されます。この FETCH ステートメントには、出力ホスト変数のリストはありません。 代わりに、この FETCH ステートメントは、SQLDA によって記述されている区域に結果を返すように、SQL に指示します。 結果は、SQLVAR 要素の SQLDATA フィールドと SQLIND フィールドによって指し示される記憶域に返されます。 FETCH ステートメントが処理された後、WORKDEPT の SQLDATA ポインターは、'E11' にセットされた参照値を持ちます。非ヌル値が戻されたため、対応する標識の値は 0 です。 PHONENO の SQLDATA ポインターは、'4502' にセットされた参照値を持ちます。非ヌル値が戻されたため、PHONENO に対応する標識の値も 0 です。

パラメーター・マーカー

前記の例で、動的に実行された SELECT ステートメントの WHERE 文節には定数値がありました。この例では、WHERE 文節は次のようになっています。

```
WHERE LASTNAME = 'PARKER'
```

LASTNAME に異なる値を使用して、SELECT ステートメントを複数回実行したいときは、SQL ステートメントを以下のように使用することができます。

```
SELECT WORKDEPT, PHONENO
    FROM CORPDATA.EMPLOYEE
    WHERE LASTNAME = ?
```

パラメーターが予測不可能であるときは、実行時までは、アプリケーションでパラメーターの数とタイプを知ることはできません。この情報は、アプリケーションの実行時に受け取るようにすることができます。また、OPEN ステートメントで USING DESCRIPTOR を使用することによって、SELECT ステートメントの WHERE 文節に入っているパラメーター・マーカーを、特定のホスト変数に入っている値で置き換えることができます。

このようなプログラムをコーディングするには、USING DESCRIPTOR 文節を伴う OPEN ステートメントを使用する必要があります。この SQL ステートメントには、単にカーソルをオープンするだけでなく、各パラメーター・マーカーを対応するホスト変数の値で置き換える働きもあります。このステートメントで指定する記述子名は、これらのホスト変数に関する有効な記述が入っている SQLDA を識別するものでなければなりません。この SQLDA は、これまでに説明したものとは異なり、SELECT リストの一部となっているデータ項目に関する情報を返すためには使用されません。すなわち、DESCRIBE ステートメントからの出力として使用されるのではなく、OPEN ステートメントへの入力として使用されます。 この SQLDA からは、SELECT ステートメントの WHERE 文節のパラメーター・マーカーと置き変わるホスト変数に関する情報が得られます。この情報は、アプリケーションから SQLDA に渡されるので、アプリケーションは、SQLDA の必要なフィールドに適切な値を入れるように設計されていなければなりません。そうすれば、SQLDA は、ホスト変数のデータによるパラメーター・マーカーの置き換え処理において、SQL への情報源として使用できるようになります。

USING DESCRIPTOR 文節を伴う OPEN ステートメントへの入力として SQLDA を使用するときは、その SQLDA のすべてのフィールドを埋める必要はありません。具体的には、SQLDAID、SQLRES、および SQLNAME はブランクにしておくことができます (特定の CCSID が必要ならば、SQLNAME をセットできます)。したがって、この方法でパラメーター・マーカーをホスト変数に置き換えるときは、次の各事項を判別する必要があります。

- パラメーター・マーカーがいくつあるか
- これらのパラメーター・マーカーのデータ・タイプと属性はなにか (SQLTYPE, SQLLEN, および SQLNAME)。
- 標識変数を必要とするか。

さらに、SELECT ステートメントと非 SELECT ステートメントの両方を取り扱うルーチンの場合には、ステートメントがどのカテゴリーに属するかを判別することが必要な場合もあります。 (別の方法として、SELECT キーワードを探すためのコードを書くこともできます。)

パラメーター・マーカーを使用するアプリケーションの場合には、そのプログラムで次のことを行う必要があります。

1. ステートメントを読み取って、DSTRING 可変長文字ストリング・ホスト変数に入れる。
2. パラメーター・マーカーの数を判別する。
3. 該当するサイズの SQLDA を割り振る。

   これは REXX には適用されません。
4. SQLN と SQLD を ? パラメーター・マーカーの数にセットする。

   SQLN は REXX には適用できません。
5. SQLDABC を SQLN*LENGTH(SQLVAR) + 16 にセットする。

   これは REXX には適用されません。
6. パラメーター・マーカーのおのおのについて次のことを行う。
   a. データ・タイプ、長さ、および標識を判別する。
   b. SQLTYPE と SQLLEN をセットする。
   c. 入力値 (? の値) を保持するためのストレージを割り振る。
   d. 上記の入力値をセットする。
   e. SQLDATA と SQLIND (適用される場合) を各パラメーター・マーカーごとにセットする。
   f. 文字変数が使用され、それらがジョブの省略時の CCSID 以外の CCSID にある場合は、SQLNAME (REXX では SQLCCSID) にその CCSID をセットする。
   g. グラフィック変数が使用され、それらがジョブの CCSID に対応する DBCS CCSID 以外の CCSID を持っている場合は、SQLNAME (REXX では SQLCCSID) にその CCSID をセットする。
   h. カーソルをオープンし、各パラメーター・マーカーをホスト変数の値で置き換えるために、USING DESCRIPTOR 文節を伴う OPEN ステートメントを出す。

これで、ステートメントを正常に処理できるようになります。

第 15 章 クライアント・インターフェースを介した動的 SQL の使用

サーバーのクライアント・インターフェースを介して DB2 UDB for iSeries データにアクセスできます。必要な操作を開始するには、次を参照してください。

- 『Java プログラムからのデータ・アクセス』
- 『ドミノを使用したデータのアクセス』
- 『オープン・データベース・コネクティビティー (ODBC) を使用したデータのアクセス』
- 294 ページの『ポータブル・アプリケーション・ソリューション環境 (PASE) を使用したデータへのアクセス』

Java プログラムからのデータ・アクセス

Developer Kit for Java JDBC (データベース・コネクティビティー) ドライバーを使って、Java プログラムの DB2 UDB for iSeries データへアクセスできます。 このドライバーで次の操作が実行できます。

- データベース・ファイルへのアクセス
- Java 用の組み込み構造化照会言語 (SQL) を使用した JDBC データベース機能へのアクセス
- SQL ステートメントの実行および結果の処理

JDBC ドライバーの使用方法の詳細については、iSeries Information Center のIBM Developer Kit for Java JDBC driver を使用するためのセットアップを参照してください。

ドミノを使用したデータのアクセス

ロータス ドミノ iSeries 対応版 はドミノ・サーバーの製品であり、これを使用すれば DB2 UDB for iSeries データベースとドミノ・データベースのデータを双方向より統合できます。 この統合を利用するためには、この 2 種類のデータベース間での許可の仕組みを理解し、管理する必要があります。 詳細については、iSeries Information Center の ロータス ドミノ iSeries 対応版 カテゴリーを参照してください。

オープン・データベース・コネクティビティー (ODBC) を使用したデータのアクセス

iSeries Access for Windows ODBC ドライバーを使用すると、ODBC クライアント・アプリケーションが、お互いに、また、サーバーとの間で、データを効果的に共有できるようになります。iSeries Information Center の iSeries Access for Windows カテゴリーのODBC 管理を参照してください。

Linux を使用した接続についての情報は、iSeries ODBC Driver for Linux のトピックにあります。このトピックでは、Linux の iSeries 論理区画へのインストール、および iSeries ODBC Driver for Linux を使用した iSeries データベースにアクセスについて、説明しています。

ポータブル・アプリケーション・ソリューション環境 (PASE) を使用したデータへのアクセス

ポータブル・アプリケーション・ソリューション環境 (PASE) は、iSeries システムで実行される AIX (または UNIX などの) アプリケーションのための統合実行時環境です。 詳しくは、iSeries Information Center の統合操作環境カテゴリーの OS/400 PASE を参照してください。

第 16 章 iSeries ナビゲーターを使用した拡張データベース機能

この章では、iSeries ナビゲーターを使用して実行できる、データベース用の拡張機能について説明します。概説とヒントが記載されています。 35 ページの『第 3 章 iSeries ナビゲーター・データベース入門』には、基本機能が説明されています。 この章で取り上げているトピックは、次のとおりです。

- 『データベース・ナビゲーターを使用したデータベースのマッピング』
- 298 ページの『「SQL スクリプトの実行」を使用したデータベースの照会』
- 302 ページの『SQL の生成を使用した SQL ステートメントの再構成』
- 303 ページの『iSeries ナビゲーターを使用した拡張表関数』
- 308 ページの『iSeries ナビゲーターを使用した SQL オブジェクトの定義』
- 310 ページの『SQL パッケージの作成』

iSeries ナビゲーターで実行できる、SQL を基にしていない他の機能については、データベース・プログラミングの以下のトピックを参照してください。

- 表、視点、索引記述の表示
- 表の再編成
- ロックされた行の表示

パフォーマンス関連の作業については、データベース・パフォーマンスおよび Query 最適化を参照してください。

データベース・ナビゲーターを使用したデータベースのマッピング

データベース・ナビゲーターを使用すると、システム上のデータベース・オブジェクト間の関係を視覚的に記述できます。データベース用にユーザーが作成したビジュアルな記述をデータベース・ナビゲーター・マップと呼びます。要約すると、データベース・ナビゲーター・マップとは、データベースと、マップ内のすべてのオブジェクト間に存在する関係のスナップショットのことです。

データベース・ナビゲーターを使用すると、データベース内の表、表相互間の関係、表に付随する索引と制約を示す図形表記を使用して、データベース・オブジェクトの複雑な関係を調べることができます。システムに接続した後で、データベース・ナビゲーターを使用して、以下のことを行えます。

- データベース・ナビゲーター・マップの作成
- マップへの新規オブジェクトの追加
- マップに組み込むオブジェクトの変更
- ユーザー定義関連の作成

データベース・ナビゲーターの主なワークスペースは、いくつかのメイン・エリアに分けられたウィンドウです。これらのエリアを使用すると、マップに組み込むオブジェクトを見つけ、マップ内の項目を表示したり隠したり、マップを表示し、保留されているマップの変更状況をチェックすることができます。以下に、データベース・ナビゲーター・ウィンドウの主なエリアについて説明します。

ロケーター・ペイン

ロケーター・ペインは、データベース・ナビゲーター・ウィンドウの左側にあり、新規マップに組み込みたいオブジェクトを見つけるため、または、オープン・マップの一部であるオブジェクトを探し出すために使用されます。ロケーター・ペインの上部は検索機能で、マップに組み込みたいオブジェクトの名前、タイプ、およびライブラリーを指定するために使用できます。検索結果は、「ライブラリー・ツリー」および「ライブラリー・テーブル」のタブの下の、ロケーター・ペインの下部に表示されます。これらのタブの下に結果が表示されると、オブジェクトを右クリックして「マップへ追加」を選択するか、オブジェクト名をダブルクリックすることによって、オブジェクトをマップに追加することができます。次に、マップが作成されたときに、「マップ中のオブジェクト」タブをクリックすることによって、マップ内のオブジェクトのリストを表示することができます。

マップ・ペイン

マップ・ペインはデータベース・ナビゲーター・ウィンドウの右側にあり、データベース・オブジェクトとその関係を図形で表示します。マップ・ペインでは、以下のことが行えます。

- システム上に存在するが、マップの現行インスタンスには含まれていなかった表や視点を追加する。
- マップからオブジェクトを除去する。
- オブジェクトの配置を変更する。
- オブジェクトをズームインまたはズームアウトする。
- マップ内のオブジェクトに変更を加える。
- マップ内のすべてのオブジェクトに SQL を生成する。

ステータス・バー

オブジェクト・ステータス・バー

オブジェクト・ステータス・バーはデータベース・ナビゲーター・ウィンドウの左方下部にあり、マップ内の可視および有資格のオブジェクトの数を表示します。

アクション・ステータス・バー

アクション・ステータス・バーはデータベース・ナビゲーター・ウィンドウの下部中央にあり、マップ内でとられた処置、および、変更が保留かどうかのクリアな説明を示します。

変更ステータス・バー

変更ステータス・バーは、変更が行われたか保留になっているかを示します。

データベース・ナビゲーターを使用するときのヒント

- ウィンドウのどちらかのサイドのサイズを変えるには、2 つのサイドを分けているバー (スプリッター) をドラッグします。
- ウィンドウの左方および右方にあるオブジェクトを右クリックしてください。右クリックによるメニューにより、よく使用する機能に即時にアクセスできます。
- ライブラリーを即時にオープンして、その中のオブジェクトを表示するには、ライブラリーをダブルクリックします。

- 各種のデータベース・ナビゲーター・コマンドにアクセスするには、**メニュー・バー**または**ツールバー**を使用します。

データベース・ナビゲーター・マップの作成

データベースについてユーザーが作成するビジュアルな記述は、データベース・ナビゲーター・マップと呼ばれます。マップは、実際には、マップを作成するまたは既存のマップをリフレッシュすることを決めたときのデータベース・データのスナップショットです。データベースのビジュアルな表現は、ユーザーが、複雑な現存データベースを理解し、新規データベースを作成し、データベースについてコミュニケートでき、データベースのオブジェクトを管理できるようにすることを目的としています。データベース・ナビゲーター・マップを作成するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**データベース・ナビゲーター**」の順に展開する。
2. 「**データベース・ナビゲーター**」を右クリックし、「**新規**」を選択する。
3. 「**データベース・ナビゲーター・ロケーター**」ペインで、「**ライブラリー・ツリー**」タブをクリックし、次に、マップを作成するのに使用したい表、視点、および索引が入っているライブラリーを選択する。
4. オブジェクト・タイプ (表、索引、または視点) を展開する。
5. マップを作成したいオブジェクトを右クリックして「**マップへ追加**」を選択するか、またはこのオブジェクトをダブルクリックする。

   注: 「**iSeries ナビゲーター**」ウィンドウの下部にあるタスク・パッド上の「**データベースのマップ**」タスクをクリックして、マップを作成することもできます。

「**ファイル**」メニューから「**終了**」を選択して、このマップを保管することができます。次に、変更が保留になっている場合は、「**変更を保管**」ダイアログで「**はい**」を選択します。このマップは、後で、再オープンできます。

データベースについて、このマップを作成すると、以下のことが行えます。

- マップへの新規オブジェクトの追加
- マップに組み込むオブジェクトの変更
- ユーザー定義関連の作成

マップへの新規オブジェクトの追加

データベース・ナビゲーターを使用すると、マップに追加する新規の SQL オブジェクトを作成することができます。作成できるオブジェクトには、以下のものがあります。

- 表
- ジャーナル
- 視点

表示される新規 SQL オブジェクトをマップに作成するには、以下のようにします。

1. データベース・ナビゲーター・マップを作成するかオープンする。

2. マップ・ペインの中で右クリックし、「**作成**」を選択する。
3. 作成したいオブジェクトのタイプを選択する。

マップに組み込むオブジェクトの変更

デフォルトでは、データベース・ナビゲーターは、すべてのオブジェクトを検索し、マップに組み込みます。検索されるオブジェクトの数を制限するには、ユーザー設定を変更します。

マップに組み込むオブジェクトを変更するには、以下のようにします。

1. データベース・ナビゲーター・マップを作成するかオープンする。
2. 「**オプション**」メニューから、「**ユーザー・プリファレンス**」を選択する。
3. 「**ユーザー・プリファレンス**」ダイアログの、「**オブジェクトのマップへの追加時に関連オブジェクトを検出**」グループ・ボックスで、組み込みたいオブジェクトを選択するか、組み込みたくないオブジェクトを選択解除する。
4. 「**OK**」をクリックする。
5. 新規設定でマップをリフレッシュしたい場合は、情報ボックスの「**はい**」をクリックする。

ユーザー定義関連の作成

ユーザーのプログラムで定義されている関連を持っているときは、関連がマップ内に表示できるように、ユーザー定義関連をデータベース・ナビゲーターに作成することができます。このような関連の 1 つの例として、2 つの間の重要な結合をプログラマーに思い出させるためのユーザー定義関連を作成することが挙げられます。

ユーザー定義関連をユーザーのマップに追加するには、以下のようにします。

1. データベース・ナビゲーター・マップを作成するかオープンする。
2. マップを右クリックし、「**作成**」を選択する。
3. 「**ユーザー定義の関連**」を選択する。
4. ユーザー定義関連の「**名前**」と「**記述**」を指定する。説明がオプションであるようないくつかの iSeries ナビゲーター機能とは異なり、ユーザー定義関連の分かりやすい説明を指定することは重要です。これが、ユーザー定義関連が何を表すかを示す唯一の手段であるからです。
5. 関連に組み込みたいオブジェクトをオブジェクトのリストから選択する。
6. オブジェクトに使いたい形状とカラーを選択する。

「SQL スクリプトの実行」を使用したデータベースの照会

iSeries ナビゲーターの「**SQL スクリプトの実行**」ウィンドウを使用すると、SQL ステートメントのスクリプトを作成、編集、実行、および、トラブルシューティングすることができます。スクリプトの処理が終了したら、スクリプトをユーザーの PC に保管できます。 「SQL スクリプトの実行」を使用して、以下のことを実行できます。

- SQL スクリプトを作成する。
- SQL スクリプトを実行する。
- ストアード・プロシージャーの結果のセットを表示する。

- ジョブ・ログを表示する。
- SQL スクリプトを実行するオプションを変更する。
- データベース・オブジェクト用の SQL を生成する。

「**SQL スクリプトの実行**」ウィンドウには、いくつかの重要なエリアがあります。

入力

「**SQL スクリプトの実行**」ウィンドウの上部には、**入力**ペインがあります。このエリアは、実行したい SQL ステートメントを作成し、編集するために使用します。ステートメントは手作業で作成することも、「**例**」リストから選択することもできます。また、**SQL の生成**機能を使用して、生成された SQL を現行カーソル位置に挿入することもできます。

例 「**例**」リスト・ボックスは、SQL ステートメントと制御言語 (CL) コマンドの例をリストします。ステートメントを選択し、挿入をクリックして、選択した例を入力ペインの現行カーソル位置に入れます。

注: ステートメントのそれぞれは、セミコロンで分離する必要があります。

出力

「**SQL スクリプトの実行**」ウィンドウの下部には、**出力**ペインがあります。これは、「**メッセージ**」タブと、実行された SQL ステートメントの出力を表示する追加のタブで構成されています。「**メッセージ**」タブは、実行された SQL ステートメントの結果に基づくフィードバックを提供します。

「SQL スクリプトの実行」ウィンドウについてのヒント

- SQL スクリプト機能は、iSeries ナビゲーターを開始せずに使用することができます。スクリプト・ファイルを保管した後で、iSeries ナビゲーターを開始せずに、スクリプト・ファイルをダブルクリックして、使用することができます。
- ステートメントを区切るためには、セミコロン (;) を使用します。
- 行の残りの部分をコメントとするためには、2 つのダッシュ (--) を使用します。
- 長いコメントを作るには、コメントを ″/*″ で開始し ″*/″ で終了します。 この方法で作成したコメントには、長さの制限はありません。
- カーソルをステートメントの中に置くと、実行時にそのステートメントが自動的に選択されます。
- 制御言語 (CL) コマンドは、ステートメントの先頭に CL: を置くことによって実行できます。バッチ・ジョブからサブミットできる CL コマンドであれば、どれでも使用できます。

SQL スクリプトの作成

SQL スクリプトを作成するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベースの順に展開する。
2. 処理したいデータベースを右クリックして、「**SQL スクリプトの実行**」を選択する。
3. 「**SQL スクリプトの実行**」から、「**新規** 」を選択する。
4. 手作業でステートメントを作成し、「**例**」リストにある例を挿入するか、SQL の生成機能を使用して、現存オブジェクト用の SQL を取り出す。

5. ステートメントの作成を終了した後で、「**実行**」メニューから「**構文検査**」を選択して、ステートメントの構文を検査することができます。

構文検査が完了したら、「**ファイル**」メニューから「**保管**」を選択して、スクリプトを保管することができます。スクリプトの位置および名前を入力するようプロンプトが出されます。

スクリプトを作成した後では、以下のことが行えます。

- SQL スクリプトを実行する。
- ジョブ・ログを表示する。
- SQL スクリプトを実行するオプションを変更する。
- データベース・オブジェクト用の SQL を生成する。

SQL スクリプトの実行

SQL スクリプトを実行するには、「**実行**」メニューから、以下のオプションのいずれかを選択します。

- **すべて** - SQL スクリプトを先頭から終了まで実行します。エラーが起こり、「**エラー時に停止**」オプションがオンになっている場合は、プログラムは停止し、エラーが起こったステートメントは選択されたままになります。
- **始めを選択** - SQL スクリプトを、選択された最初のステートメントまたは現行カーソル位置から開始し、スクリプトの終了まで継続します。
- **選択済み** - 選択されたステートメントを実行します。

結果は、「**メッセージ**」タブの終わりに追加されます。「**オプション**」メニューの「**スマート・ステートメント選択**」オプションにチェックが付いていない場合は、選択されたテキストが単一ステートメントとして実行されます。

SQL スクリプトを実行するオプションの変更

SQL スクリプトを実行するオプションを変更するには、「**オプション**」メニューから、以下のオプションのいずれかを選択します。

エラー時に停止
エラーが起こったときの停止を、オンまたはオフにします。このオプションが選択されていた場合にエラーが起こると、SQL スクリプトは実行を停止し、エラーが起こったステートメントは選択されたままになります。

スマート・ステートメント選択
スマート・ステートメント選択このオプションが選択され、「**実行**」メニューの**選択済み**コマンドが選択されると、強調表示されているすべてのステートメントが順番に実行されます。このオプションが選択されなかった場合は、**選択済み**コマンドは、強調表示されたテキストを単一 SQL ステートメントとして実行します。また、**スマート・ステートメント選択**を選択すると、1 つまたは複数のステートメントが部分的に強調表示されている場合でも、すべてのステートメントが必ず実行されます。

別のウィンドウに結果を表示
選択ステートメントの実行の結果が、「**出力**」ペインではなく、別のウィンドウに表示されるようになります。

ジョブ・ログにデバッグ・メッセージを組み込む
「**SQL スクリプトの実行**」ウィンドウで実行されたステートメントのデバッグをオンまたはオフにします。このオプションをオンにし、ステートメントを実行し、「**ジョブ・ログ**」ウィンドウをリフレッシュすることによって、照会最適化プログラムおよび他のデータベース・デバッグ・メッセージを表示できます。

ストアード・プロシージャーの結果のセットの表示

CALL ステートメントにおいて SQL ステートメントと同じようにタイプ入力することにより、「SQL スクリプトの実行」でストアード・プロシージャーの結果のセットを表示することができます。結果は、通常の照会とまったく同様に結果のウィンドウで見ることができます。複数の結果のセットがある場合は、それぞれ追加の結果タブによって表されます。さらに、**メッセージ**タブで出力パラメーターを表示することができます。 CALL の使用について詳しくは、「SQL 解説書」の中の CALL を参照してください。

ジョブ・ログの表示

ジョブ・ログ は、ユーザーのジョブに関連したメッセージを表示します。照会最適化プログラムのメッセージおよび他のデータベース・デバッグのメッセージを表示するには、「**オプション**」メニューにある「**ジョブ・ログにデバッグ・メッセージを組み込む**」を選択し、ステートメントを再度実行します。 これを行うときに「**ジョブ・ログ**」ダイアログがオープンになっている場合は、表示をリフレッシュして、新規メッセージを表示します。ジョブ・ログを表示するには、以下のようにします。

「**表示**」メニューから、「**ジョブ・ログ**」を選択します。

注: ジョブ・ログは、「**実行ヒストリーの消去**」が使用されたときにクリアされません。したがって、ジョブ・ログを使用して、もう「**出力**」ペインにないメッセージを表示することができます。

SQL スクリプトの実行を停止または取り消すには、「実行」メニューから以下のオプションのいずれかを選択します。

現行の後で停止
現在実行されているステートメントが終了した後で SQL スクリプトの実行を停止します。

要求のキャンセル
現行 SQL ステートメントをシステムがキャンセルすることを要求します。ただし、すべての SQL ステートメントをキャンセルできるわけではないので、このオプションが使用された後でも、SQL ステートメントが完了まで継続される場合があります。「**要求のキャンセル**」が押される前にホスト処理を完了した SQL ステートメントも、完了まで継続されます。たとえば、照会処理をすでに完了しているが、その結果をクライアントにまだ戻していない選択ステートメントは、通常、キャンセルできません。

SQL の生成を使用した SQL ステートメントの再構成

「SQL の生成」を使用すると、現存するデータベース・オブジェクトを作成するのに使用した SQL を再構成できます。このプロセスは、よく、リバース・エンジニアリングと呼ばれます。 SQL は、スキーマ、表、タイプ、視点、プロシージャー、関数、別名、および索引用に生成できます。 さらに、関連した制約またはトリガーを持つ表の SQL を生成する場合には、制約やトリガーにも SQL が生成されます。一時に、1 つのオブジェクトまたは多数のオブジェクトに SQL を生成することができます。 生成した SQL を実行または編集するために「SQL スクリプトの実行」ウィンドウに送るオプションを使用するか、SQL を直接データベースまたは PC ファイルに作成することもできます。

SQL の生成の使用の詳細については、以下のトピックを参照してください。

- データベース・オブジェクト用の SQL の生成
- オブジェクトのリストの編集

データベース・オブジェクト用の SQL の生成

既存のデータベース・オブジェクトを作成するのに使用した SQL を生成するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. ライブラリーの SQL を生成するには、そのライブラリーを右クリックして**「SQL の生成」**を選択する。
3. ライブラリーに入っているオブジェクトの SQL を生成するには、SQL を生成したいオブジェクトが入っているライブラリーをクリックする。
4. SQL を生成したいオブジェクトを右クリックして、「**SQL の生成**」を選択する。

SQL を生成するためのデフォルト・オプションを変更するには、タブ上の値を変更します。「**出力**」タブで、生成された SQL の宛先を選択できます。生成された SQL を「**SQL スクリプトの実行**」に送るか、または直接、ファイル、PC、またはシステムに保管することができます。「**オプション**」タブで、生成された SQL に準拠させたい「**標準**」を選択し、さらに、通知メッセージや除去メッセージを組み込み、ラベルを生成し、読み易さのためにフォーマットを設定することを選択できます。「**フォーマット**」タブで、フォーマット・オプションを選択できます。これらのオプションは、「**標準**」オプションに準拠しなければなりません。

SQL を生成するオブジェクトのリストの編集

SQL を生成するオブジェクトのリストを編集することができます。オブジェクトを追加するには、以下のようにします。

1. 「**追加**」をクリックする。
2. 「**オブジェクト・リストの編集**」ダイアログで、組み込みたいオブジェクトまでナビゲートし、「**追加**」を選択する。
3. 「**OK**」をクリックし、「SQL の生成」のメイン・ダイアログに戻る。

リストから、オブジェクトを除去するには、以下のようにします。

1. 「**SQL を生成するオブジェクト**」から、除去したいオブジェクトを選択する。

2. 「**除去**」をクリックする。

iSeries ナビゲーターを使用した拡張表関数

35 ページの『第 3 章 iSeries ナビゲーター・データベース入門』 では、ユーザーが実行できる基本表関数について説明しています。 しかし、ユーザーは iSeries ナビゲーターを使用して、以下のことも実行できます。

- 別名の作成
- 索引の追加
- キー制約の追加
- 検査制約の追加
- 参照制約の追加
- トリガーの追加
- トリガーの使用可/使用不可
- 制約またはトリガーの除去

iSeries ナビゲーターを使用した別名の作成

別名とは、表または視点の代替名のことです。既存の表または視点を参照できる場合に、別名を使用して表や視点を参照することができます。 表または視点の別名、あるいは表または視点のメンバーの別名を作成することができます。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. 新規の別名を作成したいライブラリーを右クリックする。
3. ポップアップ・メニューから、「**新規作成**」を選択し、次に、「**別名**」を選択する。
4. 「**新規別名**」ダイアログの「**別名**」フィールドに、作成する別名の名前を指定する。 この名前は、サーバーにすでに存在している索引、表、視点、ファイル、または別名と同じであってはなりません。
5. 「**記述**」フィールドに、新しい別名の説明を指定する。 この説明は最大 50 文字までの長さとすることができます。 このフィールドはオプションです。
6. 「**テーブル/ビュー**」フィールドで、別名によって示される表 (テーブル) または視点 (ビュー) を指定する。
7. 「**ライブラリー**」フィールドで、別名によって示される表または視点が入っているライブラリーを指定する。
8. 「**拡張**」をクリックする。
9. 表または視点の別名を作成したい場合は、「**拡張**」ダイアログで、「**テーブルまたはビューの別名を作成**」をクリックする。 これが省略時のオプションです。
10. 表または視点のメンバーの別名を作成するには、「**テーブルまたはビューのメンバーの別名を作成** 」をクリックする。コンボ・ボックスに、別名によって示されるメンバーを入力または選択します。
11. 「**OK**」をクリックして、「**新規別名**」ダイアログに戻る。
12. 「**OK**」をクリックして、別名を作成する。

注: 表 (テーブル) または視点 (ビュー) を右クリックし、「**別名の作成**」を選択して、別名を作成することもできます。

別名について詳しくは、 60 ページの『ALIAS 名の作成と使用』を参照してください。

iSeries ナビゲーターを使用した索引の追加

索引を使用して、データのソートと選択ができます。さらに、索引を使用すると、システムはデータをより速く取り出すことができ、照会のパフォーマンスが向上します。

索引は複数作成できます。ただし、索引はシステムによって保守されるため、索引の数が多いとパフォーマンスが低下することもあります。索引および照会のパフォーマンスの詳細については、データベース・パフォーマンスおよび *Query 最適化* の SQL 索引の効果的使用法を参照してください。

コード化ベクトル索引というタイプの索引を使用すると、並列処理が簡単に行え、より高速のスキャンが可能になります。 コード化ベクトル索引を使用して照会をスピードアップする方法について詳しくは、  「DB2 for iSeries の Web ページ」を参照してください。

新規または既存の表に索引を作成することができます。 以下のように iSeries ナビゲーターを使用して、基数索引またはコード化ベクトル索引を作成することができます。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. 索引を追加したい表が入っているライブラリーをクリックする。
3. 詳細ペインで、索引を追加したい表を右クリックし、「**プロパティー**」を選択する。
4. 「**テーブル・プロパティー**」または「**新規テーブル**」ダイアログで、「索引」タブを選択する。
5. 「**索引**」タブで、「**新規**」をクリックする。
6. 「**索引**」フィールドで、新規索引の名前を指定する。
7. 「**ライブラリー**」フィールドで、索引が常駐するライブラリーを選択する。
8. 表のどの列が索引になるかを選択する。列を追加するためにその列をクリックすると、左方に番号が表示されます。この番号が、索引の中の列のキー位置を決めます。列を索引から除去するには、列をもう一度クリックします。
9. キー・フィールドの順序を昇順から降順 (または降順から昇順) に変更するには、2 番目のカラムをクリックする。
10. 「**索引タイプ**」を選択する。
11. コード化ベクトル索引を作成している場合は、特殊値の数を選択する。
12. 「**OK**」をクリックして、索引を作成する。

注: 「**新規テーブルの作成場所**」ダイアログで、索引を新規の表に追加することもできます。

制約は、現行の表の編集セッション中に定義された場合にのみ、変更することができます。「**新規テーブル**」ダイアログ、または「**テーブル・プロパティー**」ダイアログで制約を追加し「**OK**」をクリックした場合は、その制約に対しては読み取り専用アクセスしか持てません。制約特性 (制約プロパティー) を変更したい場合には、その制約を除去し、適当な変更を指定して再作成しなければなりません。

索引の作成について詳しくは、 63 ページの『索引の追加』を参照してください。

iSeries ナビゲーターを使用したキー制約の追加

制約は、データベース・マネージャーによって実施される規則です。DB2 UDB for iSeries は、2 つのタイプのキー制約をサポートします。

- 固有キー制約は、キーの値が固有である場合にのみその値が有効となる規則です。固有制約は、INSERT および UPDATE ステートメントの実行時に実施されます。
- 基本キー制約は固有制約の 1 つの形式です。違いは、基本キーにはヌル可能列を入れられない点です。

キー制約を作成するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. キー制約を追加したい表が入るライブラリーをクリックする。
3. キーを追加したい表を右クリックし、「**プロパティー**」を選択する。
4. 「**テーブル・プロパティー**」ダイアログで、「**キー制約**」タブを選択する。
5. 「**キー制約**」タブで、「**新規**」をクリックする。
6. 「**新規キー制約**」ダイアログで、名前のテキスト・ボックスに、名前を指定する。名前を指定しない場合は、システムが自動的に名前を生成します。
7. キーを追加したい列を選択する。
8. 「**基本**」を選択すれば基本キーが作成され、「**固有**」を選択すれば固有キーが作成される。
9. 「**OK**」をクリックして、「**テーブル・プロパティー**」ダイアログに戻る。
10. 「**OK**」をクリックして、キーを作成する。

注: 「**新規テーブルの作成場所**」ダイアログで、キー制約を新規の表に追加することもできます。

制約は、現行の表の編集セッション中に定義された場合にのみ、変更することができます。 「**新規テーブル**」ダイアログ、または「**テーブル・プロパティー**」ダイアログで制約を追加し「**OK**」をクリックした場合は、その制約に対しては読み取り専用アクセスしか持てません。 制約特性を変更したい場合には、その制約を除去し、適当な変更を指定して再作成しなければなりません。

キー制約の追加について詳しくは、 145 ページの『第 10 章 データ保全性』を参照してください。

iSeries ナビゲーターを使用した検査制約の追加

検査制約は、列または列のグループの中で使用できる値を制限することにより、挿入および更新中のデータの妥当性を保証します。

検査制約を作成するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. 検査制約を追加したい表が入っているライブラリーをクリックする。
3. 検査制約を追加したい表を右クリックし、「**プロパティー**」を選択する。
4. 「**テーブル・プロパティー**」ダイアログで、「**検査制約**」タブをクリックする。
5. 「**検査制約**」タブで、「**新規**」をクリックする。
6. 「**検査制約探索条件**」ダイアログで、名前のテキスト・ボックスに、名前を指定する。名前を指定しない場合は、システムが自動的に名前を生成します。
7. 「**カラム**」リストから、制約を追加したい列 (カラム) をダブルクリックする。その列が、「文節」ボックスに表示されます。
8. リストにある演算子を挿入するには、挿入したい演算子をダブルクリックする。演算子が「文節」ボックスに表示されます。
9. リストにある関数を挿入するには、それをダブルクリックする。ドロップダウン・リストから関数タイプを選択することによって、リストを変更することができます。関数をダブルクリックすると、それが「文節」ボックスに表示されます。
10. 式が正しくなるまで変更する。
11. 「**OK**」をクリックして、「**テーブル・プロパティー**」ダイアログに戻る。
12. 「**OK**」をクリックして、検査制約を作成する。

注: 「**新規テーブルの作成場所**」ダイアログで、検査制約を新規の表に追加することもできます。

制約は、現行の表の編集セッション中に定義された場合にのみ、変更することができます。 「**新規テーブル**」ダイアログ、または「**テーブル・プロパティー**」ダイアログで制約を追加し「**OK**」をクリックした場合は、その制約に対しては読み取り専用アクセスしか持てません。制約特性を変更したい場合には、その制約を除去し、適当な変更を指定して再作成しなければなりません。

検査制約の追加について詳しくは、 145 ページの『検査制約の追加および使用』を参照してください。

iSeries ナビゲーターを使用した参照制約の追加

参照保全とは、1 つの表から別の表へのあらゆる参照が行い得るデータベースの中の一組の表の状態のことをいいます。 データベースの中の参照保全は、参照制約を追加することによって保証できるようになります。

参照制約を作成するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。

2. 従属表が入っているライブラリーをダブルクリックする。
3. 従属表を右クリックし、「**プロパティー**」を選択する。
4. 「**テーブル・プロパティー**」ダイアログで、「**参照制約**」タブを選択する。
5. 「**参照制約**」タブで、「**新規**」をクリックする。
6. 「**新規参照制約**」ダイアログで、制約の名前を指定する。名前を指定しない場合は、システムが自動的に名前を生成します。
7. 親表の中の親キー値に従属関係を持たせたい列を選択する。
8. 親表が入っているライブラリーを選択する。
9. 親キーが入っている表を選択する。
10. 参照する親キーを選択する。
11. 削除アクションを選択する。
12. 更新アクションを選択する (挿入のアクションがデフォルトです) 。
13. 「**OK**」をクリックして、「**テーブル・プロパティー**」ダイアログに戻る。
14. 「**OK**」をクリックして、参照制約を作成する。

注: 「**新規テーブルの作成場所**」ダイアログで、参照制約を新規の表に追加することもできます。

制約は、現行の表の編集セッション中に定義された場合にのみ、変更することができます。 「**新規テーブル**」ダイアログ、または「**テーブル・プロパティー**」ダイアログで制約を追加し「**OK**」をクリックした場合は、その制約に対しては読み取り専用アクセスしか持てません。制約特性を変更したい場合には、その制約を除去し、適当な変更を指定して再作成しなければなりません。

参照制約の追加について詳しくは、 147 ページの『参照制約の追加または削除』を参照してください。

iSeries ナビゲーターを使用したトリガーの追加

トリガーは、指定の物理データベース・ファイルに対して指定の変更操作が実行されるときに自動的に実行されるアクションのセットです。この説明においては、表が物理ファイルに相当します。変更操作としては、アプリケーション・プログラム内の高水準言語の挿入、更新、または削除ステートメント、あるいは SQL の INSERT、UPDATE、または DELETE ステートメントが可能です。トリガーは、業務に関する規則の適用、入力データの妥当性検査、および監査証跡の保管などの作業に役立ちます。

iSeries ナビゲーターを使用して、システム・トリガーおよび SQL トリガーを定義することができます。さらに、トリガーを使用可能にしたり、使用不可にしたりできます。

トリガーを追加するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. トリガーを追加したい表が入っているライブラリーをクリックする。
3. トリガーを追加したい表を右クリックし、「**プロパティー**」を選択する。「**テーブル・プロパティー**」ダイアログで、「**トリガー**」タブをクリックする。

4. 「**システム・トリガーの追加**」を選択して、システム・トリガーを追加する。
5. 「**SQL トリガーの追加**」を選択して、SQL トリガーを追加する。

システム・トリガーについて詳しくは、データベース・プログラミング のデータベース内での自動イベントのトリガーのトピックを参照してください。

SQL トリガーの詳細については、159 ページの『SQL トリガー』を参照してください。

トリガーの使用可能/使用不可

トリガーを実行するには、使用可能にする必要があります。 ただし、トリガーを使用不可にすると、トリガーを実行せずに、表で作業を行うことができます。

トリガーを使用可能/使用不可にするには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. トリガーを使用可能/使用不可にしたい表が入っているライブラリーをクリックする。
3. 表を右クリックし、「**プロパティー**」を選択する。
4. 「**テーブル・プロパティー**」ダイアログで、「**トリガー**」タブをクリックする。
5. 使用可能/使用不可にしたいトリガーを選択し、「**使用可能**」をクリックしてトリガーを使用可能にするか、「**使用不可**」をクリックしてトリガーを使用不可にする。

CHGPFTRG CL コマンドを使用してトリガーを使用可能にする、または使用不可にすることについての説明は、データベース・プログラミング のトリガーの使用可能/使用不可を参照してください。

制約およびトリガーの除去

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. 制約またはトリガーを除去したい表が入っているライブラリーをクリックする。
3. 制約またはトリガーを除去したい表を右クリックし、「**プロパティー**」を選択する。
4. 「**テーブル・プロパティー**」ダイアログで、除去したい制約またはトリガーのタイプのタブをクリックする。
5. 除去したい制約を選択し、「**削除**」をクリックする。

iSeries ナビゲーターを使用した SQL オブジェクトの定義

iSeries ナビゲーターには、ある種の SQL オブジェクトをシステム上に定義する単純な手段が用意されています。たとえば、以下のものを定義できます。

- プロシージャー。プロシージャー (しばしば、ストアード・プロシージャーと呼ばれる) とは、操作を実行するために呼び出すことができるプログラムのことで、ホスト言語ステートメントおよび SQL ステートメントの両方を含みます。SQL のプロシージャーの場合も、ホスト言語のプロシージャーの場合と同じ利点

があります。 すなわち、共通のコードを一度だけ作成し維持管理しさえすれば、いくつかのプログラムから呼び出すことができるようになります。 プロシージャーの詳細については、 171 ページの『第 11 章 ストアード・プロシージャー』を参照してください。

- ユーザー定義関数。ユーザー定義関数 (UDF) は、ソース、外部、および SQL という 3 つのタイプからなります。 ソース関数 UDF は、他の関数を呼び出して操作を実行します。 SQL および外部関数 UDF は、ユーザーがコードを作成して実行する必要があります。 スカラー関数、列関数、および表関数を作成することができます。関数について詳しくは、 217 ページの『ユーザー定義関数 (UDF)』を参照してください。
- ユーザー定義のタイプ。ユーザー定義の特殊タイプは、使用可能な組み込みデータ・タイプ以上に DB2 の機能を拡張できるようにするメカニズムです。ユーザー定義の特殊タイプによりユーザーは、DB2 に対して新しいデータ・タイプを定義できるようになります。したがって、ユーザーのビジネスをモデル化したり、ユーザーのデータの意味体系を取り込むうえで、システムが提供する組み込みデータ・タイプを使用することに限定されないため、ユーザーは、少なからぬパワーを得ることになります。特殊なデータ・タイプを使用すると、ユーザーは、既存のデータベース・タイプに対して 1 対 1 でマップできるようになります。 タイプについて詳しくは、 234 ページの『ユーザー定義の特殊タイプ (UDT)』を参照してください。

iSeries ナビゲーターを使用した、これらのオブジェクトの定義の詳細については、以下のトピックを参照してください。

- 『iSeries ナビゲーターを使用したストアード・プロシージャーの定義』
- 310 ページの『iSeries ナビゲーターを使用したユーザー定義関数の定義』
- 310 ページの『iSeries ナビゲーターを使用したユーザー定義タイプの定義』

iSeries ナビゲーターを使用したストアード・プロシージャーの定義

プログラムを、SQL プログラムからプロシージャーとして呼び出したい場合は、まず、そのプログラムを外部プロシージャーとして定義する必要があります。プロシージャーとして定義しようとしているプログラムは、そのプロシージャーを定義するときには、存在している必要はありません。

プログラムをプロシージャーとして定義するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. 関数を定義して入れたいライブラリーを右クリックし、「**新規**」を選択する。
3. 「**プロシージャー**」を選択する。
4. 「**外部**」を選択して、外部ストアード・プロシージャーを作成する。
5. 「**SQL**」を選択して、SQL ストアード・プロシージャーを作成する。

外部ストアード・プロシージャーの作成と定義について詳しくは、 172 ページの『外部プロシージャーの定義』を参照してください。

SQL ストアード・プロシージャーの作成と定義について詳しくは、 173 ページの『SQL プロシージャーの定義』を参照してください。

iSeries ナビゲーターを使用したユーザー定義関数の定義

プログラムをユーザー定義関数として定義するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. 関数を定義して入れたいライブラリーを右クリックし、「**新規**」を選択する。
3. 「**関数**」を選択する。
4. 「**外部**」を選択して、外部ユーザー定義関数を作成する。
5. 「**SQL**」を選択して、SQL ユーザー定義関数を作成する。
6. 「**ソース**」を選択して、別の関数を基にした関数を作成する。

外部関数の作成と定義について詳しくは、217 ページの『ユーザー定義関数 (UDF)』を参照してください。

iSeries ナビゲーターを使用したユーザー定義タイプの定義

既存のデータ・タイプを基にして新規のユーザー定義データ・タイプを作成すると、データをよりよくコントロールすることができます。

ユーザー定義タイプを作成するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. タイプを定義して入れたいライブラリーを右クリックし、「**新規作成**」を選択する。
3. 「**タイプ**」を選択する。
4. 「**新規タイプ**」ダイアログで、新規タイプに付けたい名前を「タイプ」フィールドに指定する。
5. 新規タイプが基にしているデータ・タイプを、「**ソース・データ・タイプ**」セクションの「**タイプ**」フィールドに指定する。
6. 「**OK**」をクリックする。

ユーザー定義タイプの作成と定義について詳しくは、234 ページの『ユーザー定義の特殊タイプ (UDT)』を参照してください。

SQL パッケージの作成

SQL パッケージは、準備済み SQL ステートメントに関連した情報を保管するために使用される永続オブジェクトです。 これらは、データ・ソースで「拡張動的」ボックスにチェックマークを付けると、ODBC サポートによって使用されます。

SQL パッケージを作成するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」の順に展開する。
2. 使用したいデータベースを右クリックして、「**新規 SQL パッケージ**」を選択する。
3. 「**SQL パッケージの作成**」ダイアログで、必要に応じてパラメーターを指定する。

4. 「**OK**」をクリックする。

パラメーターの全リストについては、 407 ページの『CRTSQLPKG (SQL パッケージ作成) コマンド』を参照してください。

第 17 章 対話式 SQL の使用

この章では、対話式 SQL を用いて SQL ステートメントを実行する方法およびプロンプト機能を使用する方法について説明します。概説および対話式 SQL の使用上のヒントを示してあります。 SQL の使用法を学びたいときは、『第 2 章 SQL 入門』を参照してください。 対話式 SQL を遠隔接続で使用する上での特殊な考慮事項については、324 ページの 『対話式 SQL による遠隔データベースのアクセス』で説明しています。

詳細については、『対話式 SQL の基本機能』を参照してください。

注:

1. スキーマ の同義語としてスキーマ という用語が使われます。
2. コーディング例に関する詳細は、x ページの『コードに関する特記事項』を参照してください。

対話式 SQL の基本機能

対話式 SQL を使用すると、プログラマーまたはデータベース管理者は、データの定義、データの更新、データの削除、またはテスト、問題分析、データベース管理のためのデータの検査を迅速かつ簡単に行うことができます。プログラマーは対話式 SQL を使用して、複数の行を 1 つの表に挿入したり、アプリケーション・プログラムの中で SQL ステートメントを実行する前にその SQL ステートメントをテストすることができます。データベース管理者は、対話式 SQL を使用して、特権の認可または取り消し、スキーマ、表、または視点の作成または除去、あるいは、システム・カタログ表からの情報の選択などを行うことができます。

対話式 SQL ステートメントが実行されると、完了メッセージまたはエラー・メッセージが表示されます。さらに、実行時間の長いステートメントの場合は、その途中で状況メッセージが表示されるのが普通です。

メッセージにカーソルを合わせて F1 (ヘルプ) キーを押すと、そのメッセージに関するヘルプ情報が示されます。

対話式 SQL の基本機能は次のとおりです。

- **ステートメント入力**機能を使用すると、次のことを行うことができます。
  - 対話式 SQL ステートメントを入力しそれを実行する。
  - ステートメントを取り出して編集する。
  - SQL ステートメントのプロンプトを出す。
  - 前のステートメントやメッセージに戻るためにページを戻す。
  - セッション・サービスを呼び出す。
  - リスト選択機能を呼び出す。
  - 対話式 SQL を終了する。
- **プロンプト**機能を使用すると、SQL ステートメントを完全な形であるいは部分的に入力し、F4 (プロンプト) キーを押すことにより、そのステートメントの構文

のプロンプトを表示することができます。また F4 を押すと、すべての SQL ステートメントのメニューが表示されます。このメニューからは、あるステートメントを選択して、そのステートメントの構文のプロンプトを表示することができます。

- **リスト選択機能**を使用すると、使用を許可されたリレーショナル・データベース、スキーマ、表、視点、列、制約、または SQL パッケージのリストから選択を行うことができます。

  リストから選択した項目は、SQL ステートメントの中のカーソルが置かれている位置に挿入することができます。
- **セッション・サービス**機能を使用すると、次のことを行うことができます。
  - セッション属性を変更する。
  - 現行セッションを印刷する。
  - 現行セッションからすべての項目を除去する。
  - セッションをソース・ファイルに保管する。

詳細については、以下のセクションを参照してください。

- 『対話式 SQL の開始』
- 316 ページの『ステートメント入力機能の使用』
- 316 ページの『プロンプト』
- 319 ページの『リスト選択機能の使用』
- 322 ページの『セッション・サービスの説明』
- 323 ページの『対話式 SQL の終了』
- 324 ページの『既存の SQL セッションの使用』
- 324 ページの『SQL セッションの回復』
- 324 ページの『対話式 SQL による遠隔データベースのアクセス』

対話式 SQL の開始

対話式 SQL の使用を開始するには、OS/400 コマンド行で STRSQL とタイプ入力します。 このコマンドとそのパラメーターの詳しい説明については、『付録 B. DB2 UDB for iSeries CL コマンドの説明』を参照してください。

これにより、「SQL ステートメントの入力」画面が表示されます。これは、メインの「対話式 SQL」画面です。この画面から SQL ステートメントを入力して、次のキーを使用することができます。

- F4=プロンプト
- F13=セッション・サービス
- F16=コレクションの選択
- F17=表の選択
- F18=列の選択

```
                                SQL ステートメントの入力

 SQL ステートメントを入力して，実行キーを押してください。
    現行の接続相手は、リレーショナル・データベース RDJACQUE である。
===>_____________________________________________________________________________
     _____________________________________________________________________________
     _____________________________________________________________________________
     _____________________________________________________________________________
     _____________________________________________________________________________
     _____________________________________________________________________________
     _____________________________________________________________________________
     _____________________________________________________________________________
     _____________________________________________________________________________
     _____________________________________________________________________________
     _____________________________________________________________________________
     _____________________________________________________________________________
     _____________________________________________________________________________
     _____________________________________________________________________________
     _____________________________________________________________________________
     _____________________________________________________________________________
                                                                            終り
 F3= 終了     F4=プロンプト    F6= 行の挿入     F9=コマンドの複写     F10= 行のコピー
F12=取消し                    F13=サービス     F24=キーの続き
```

F24 (キーの続き) を押すことにより、残りの機能キーが表示されます 。

```
     _____________________________________________________________________________
     _____________________________________________________________________________
     _____________________________________________________________________________
     _____________________________________________________________________________
                                                                            終り

F14=行の削除     F15=行の分割       F16=コレクションの選択 (ライブラリー)
F17=テーブルの選択                  F18=列の選択   F24=キーの続き
          (ファイル)                      (フィールド)
```

注: システム命名規則を使用している場合は、上記の名称の代りに括弧内に示した名称が表示されます。

対話式セッションは、次のもので構成されます。

- STRSQL コマンドで指定したパラメーターの値。
- セッションで入力した SQL ステートメントとともに、各 SQL ステートメントのあとに続く対応メッセージ。
- セッション・サービス機能を使用して変更したすべてのパラメーターの値。
- 行ったリスト選択。

対話式 SQL には、固有のセッション ID が用意されていますが、これはユーザー ID と現行のワークステーション ID とで構成されています。このセッション ID の考え方によると、同じユーザー ID をもつ複数のユーザーが複数のワークステーションから同時に対話式 SQL を使用することができます。また、同じユーザー ID で、同じワークステーションから複数の対話式 SQL セッションを同時に実行することができます。

SQL セッションが存在し、しかも再入力が行われている場合は、STRSQL コマンドで指定したすべてのパラメーターは無視されます。既存の SQL セッションからのパラメーターが使用されます。

ステートメント入力機能の使用

ステートメント入力機能は、対話式 SQL を選択するときにユーザーが最初に使用する機能です。各対話式 SQL ステートメントの処理が終わるたびに、ステートメント入力機能に戻ります。

ステートメント入力機能では、ユーザーは 1 つの SQL ステートメント全体を入力するか、プロンプトを使用して入力した上で、実行キーを押すと、そのステートメントは処理のために送られます。

ステートメントの入力

コマンド行に入力するステートメントは、1 行でも複数でも構いません。対話式 SQL では SQL ステートメントに注釈を入力することはできません。ステートメントが処理されると、そのステートメントと結果のメッセージが画面の上方に移動します。そのあとで、ユーザーは別のステートメントを入力することができます。

ステートメントが SQL によって認識されたが、構文エラーを含んでいると、そのステートメントと結果のテキスト・メッセージ (構文エラー) は画面の上方に移動します。入力域には、ステートメントのコピーが表示され、構文エラーのある部分にカーソルが置かれています。エラーに関する詳しい情報を表示させるには、カーソルをメッセージ上に置いて F1 (ヘルプ) キーを押してください。

前のステートメント、コマンド、およびメッセージを見るために前のページに戻ることができます。前のステートメントにカーソルを置いて F9 (検索) キーを押して、そのステートメントのコピーを入力域に入れます。 SQL ステートメントを入力するためにもっと多くのスペースを必要とする場合は、画面をページ送りしてください。

プロンプト

プロンプト機能を使用すると、使用したいステートメントの構文に関する必要な情報を得ることができます。プロンプト機能は、*RUN、*VLD、および *SYN の 3 つのステートメント処理モードのいずれでも使用できます。

プロンプト機能を使用する場合、次の 2 つのオプションがあります。

- ステートメントの verb を入力してから F4 (プロンプト) キーを押します。

  ステートメントは解析され、完全なものとなった文節がプロンプト画面上に記入されます。

  SELECT を入力して F4 (プロンプト) キーを押すと、次の画面が表示されます。

```
                        SELECT ステートメントの指定

SELECT ステートメント情報を入力してください。リストの表示は、F4 キーを
押してください。

  FROM テーブル . . . . . . . . _____________________________________________
  SELECT 列     . . . . . . . . _____________________________________________
  WHERE 条件    . . . . . . . . _____________________________________________
  GROUP BY 列   . . . . . . . . _____________________________________________
  HAVING 条件 . . . . . . . . . _____________________________________________
  ORDER BY 列   . . . . . . . . _____________________________________________
  FOR UPDATE OF 列  . . . . . . _____________________________________________

                                                                    終わり
選択項目を入力して、実行キーを押してください。

   結果テーブル中の DISTINCT 行 . . . . . . . . . . .  N   Y=YES, N=NO
   別の SELECT との UNION . . . . . . . . . . . . . .  N   Y=YES, N=NO
   追加オプションの指定 . . . . . . . . . . . . . . .  N   Y=YES, N=NO



F3=終了          F4=プロンプト    F5=最新表示    F6=行挿入      F9=SUBQUERYの指定
F10=行のコピー   F12=取り消し     F14=行削除      F15=行分割      F24=キーの続き
```

- 「SQL ステートメントの入力」画面に入力する前に F4 (プロンプト) キーを押します。これにより、ステートメントのリストが表示されます。ステートメントのリストの種類は、現行の対話式 SQL ステートメントの処理モードによって異なります。 *NONE 以外の言語の構文検査モードの場合、リストにはすべての SQL ステートメントが含まれます。実行および妥当性検査モードの場合、対話式 SQL で実行できるステートメントだけが表示されます。使用したいステートメントの番号を選択することができます。システムは、選択したステートメントの入力をプロンプトで要求します。

  何も入力しないで F4 (プロンプト) キーを押すと、次の画面が表示されます。

```
                        SQL ステートメントの選択

次の中から 1 つを選んでください。

     1. ALTER TABLE
     2. CALL
     3. COMMENT ON
     4. COMMIT
     5. CONNECT
     6. CREATE ALIAS
     7. CREATE COLLECTION
     8. CREATE INDEX
     9. CREATE PROCEDURE
    10. CREATE TABLE
    11. CREATE VIEW
    12. DELETE
    13. DISCONNECT
    14. DROP ALIAS
                                                              続く...
選択項目
     __

F3=終了    F12=取消し
```

プロンプト画面で F21 (ステートメントの表示) を押すと、プロンプト機能はフォーマットされた SQL ステートメントを、その時点までに記入されたとおりに表示します。

プロンプト中に実行キーを押すと、プロンプト画面で作成したステートメントがセッションに挿入されます。ステートメント処理モードが *RUN の場合は、そのステートメントが実行されます。エラーを検出した場合は、プロンプト機能に制御権が残ります。

構文検査

SQL ステートメントの構文は、そのステートメントがプロンプト機能に入ったときに検査されます。プロンプト機能は構文が正しくないステートメントを受け付けません。構文を訂正するかまたはステートメントの正しくない部分を削除しないと、プロンプトは使用できません。

ステートメント処理モード

ステートメント処理モードは、「セッション属性変更」画面で選択することができます。 *RUN (実行) モードまたは *VLD (妥当性検査) モードでは、プロンプトを出すことができるのは、対話式 SQL で実行できるステートメントに限ります。*SYN (構文検査) モードでは、すべての SQL ステートメントを使用することができます。ステートメントは、*SYN モードや *VLD モードでは実際には実行されません。構文とオブジェクトの存在だけが検査されます。

副照会

副照会は、WHERE 文節または HAVING 文節が表示されている任意の画面で選択することができます。副照会画面を表示するには、カーソルが WHERE または HAVING の入力行に置かれているときに F9 (副照会の指定) キーを押してください。部分選択情報を入力できる画面が表示されます。 F9 を押したときカーソルが副照会の括弧の中にあったときは、副照会情報は次に表示される画面で記入されます。カーソルが副照会の括弧の外にあるときは、次の画面はブランクになっています。副照会について詳しくは、 113 ページの『SELECT ステートメントの副照会』を参照してください。

CREATE TABLE プロンプト

CREATE TABLE をプロンプトを出すときは、列定義を個別に入力するためのサポートを利用できます。カーソルを画面の列定義セクションに置いて、F4 (プロンプト) キーを押してください。 1 つの列定義のすべての情報を入れるスペースを提供する画面が表示されています。

18 文字を超える列名を入力するには、F20 (名前全体の表示) キーを押してください。30 文字の名前が十分に入るウィンドウが表示されます。

編集キー、F6 (行の挿入) キー、F10 (行のコピー) キー、および F14 (行の削除) キーを使用して、列定義リストの項目の追加および削除を行うことができます。

DBCS データの入力

複数行にわたる DBCS データを処理するときの規則は、「SQL ステートメントの入力」画面および SQL プロンプト機能の場合と同じです。 1 つの行の中では、シフトイン文字とシフトアウト文字の数が必ず同じでなければなりません。入力に複数行を要する DBCS データ・ストリングを処理する場合、余分なシフトイン文字とシフトアウト文字が除去されます。ある行の最後の列にシフトインがあり、その次の行の最初の列にシフトアウト文字がある場合には、その 2 つの行を組み合わせる時点でプロンプト機能によってこのシフトイン文字とシフトアウト文字は除去され

ます。ある行の最後の 2 列にシフトイン文字と 1 バイトのブランクが入っていて、その次の行の最初の列にシフトアウト文字が入っている場合は、その 2 つの行を組み合わせる時点で、シフトイン文字、ブランク、およびシフトアウト文字の文字列は除去されます。この除去により、DBCS 情報は連続する 1 つの文字ストリングとして読み取ることができます。

その一例として、次の WHERE 条件が入力されたと想定します。シフト文字が、この画面の 2 つの各行のストリング・セクションの始まりと終わりに表示されています。

```
                        SELECT ステートメントの指定

SELECT ステートメント情報を入力してください。リストの表示は、F4 キーを
押してください。

  FROM テーブル . . . . . .  TABLE1_____________________________________
  SELECT 列    . . . . . .  *_________________________________________
  WHERE 条件   . . . . . .  COL1 = '<AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQ>
                            <RRSS>'_____________________________________
  GROUP BY 列  . . . . . .  _____________________________________________
  HAVING 条件 . . . . . .  _____________________________________________
  ORDER BY 列  . . . . . .  _____________________________________________
  FOR UPDATE OF 列 . . . .  _____________________________________________
```

実行キーを押すと、文字ストリングが連結され、余分なシフト文字が除去されます。このステートメントは、「SQL ステートメントの入力」画面で次のように表示されます。

```
SELECT * FROM TABLE1 WHERE COL1 = '<AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSS>'
```

リスト選択機能の使用

リスト選択機能を使用可能にするには、特定のプロンプト画面で F4 キーを押すか、または「SQL ステートメントの入力」画面で F16 キー、F17 キー、または F18 キーを押します。これらの機能キーを押すと、許可されたリレーショナル・データベース、スキーマ、表、視点、別名、列、制約、プロシージャー、パラメーター、またはパッケージのリストが表示されるので、そこから選択することができます。表のリストを要求したが、その前にスキーマを選択していないと、スキーマを先に選択するように求められます。

リスト上では、1 つまたは複数の項目を選択し、ステートメントに表示したい順序を番号で指定することができます。リスト機能を終了すると、選択した項目は、前の画面でカーソルが置かれていた個所に挿入されます。

最も関心のあるリストを常に選択してください。たとえば、列のリストが必要で、その列が現在選択している表にないと思われるときは、F18 (列の選択) を押してください。次に、その列リストから、F17 キーを押して表を変更してください。最初に表のリストを選択した場合は、表名がステートメントに挿入されます。列を選択する選択肢はありません。

リストは、「SQL ステートメントの入力」画面から SQL ステートメントを入力する際に、いつでも要求することができます。リストから選択した項目は、「SQL ステートメントの入力」画面に挿入されます。これらは、カーソルが置かれている個所にリスト画面で指定した番号順に挿入されます。選択リスト情報は追加されましたが、ユーザーはステートメントのキーワードを入力する必要があります。

リスト機能は、選択された列、表、および SQL パッケージに必要な修飾を試みます。しかし、リスト機能が SQL ステートメントの意図を判別できないことがあります。したがって、ユーザーは SQL ステートメントを調べて、選択した列、表、および SQL パッケージが正しく修飾されていることを検査する必要があります。

リスト選択機能の使用例

次の例は、リスト機能を使用して SELECT ステートメントを作成する方法を示しています。

次のような想定の下に行います。

- OS/400 コマンド行で STRSQL とタイプ入力して、対話式 SQL に入ったばかりである。
- まだ、リストの選択も入力も行っていない。
- 命名規則として *SQL を選択した。

注: この例は、ユーザーのサーバーにないリストを示しています。これらは、一例として使用されているに過ぎません。

次のようにして SQL ステートメントの使用を開始します。

1. 最初のステートメント入力行に SELECT と入力します。
2. 2 番目のデータ入力行に FROM と入力します。
3. カーソルは FROM の後の位置に置いたままにしておきます。

```
                          SQL ステートメントの入力

SQL ステートメントを入力して，実行キーを押してください。
===> SELECT
     FROM _
```

4. FROM の後には表名を指定したいので、表のリストを得るために F17 (表の選択) キーを押してください。

   希望した表のリストは表示されず、コレクションのリストが表示されます (「コレクションの選択および順序づけ」画面)。SQL セッションに入ったばかりで、処理の対象とするスキーマをまだ選択していないためです。

5. YOURCOLL2 スキーマの横の *SEQ* 列に 1 を入力してください。

```
                      コレクションの選択および順序づけ

コレクションを選択するためには、順序番号 (1-999) を入力して実行キーを押してください。

SEQ   コレクション   タイプ  テキスト
      YOURCOLL1       SYS    会社の利益
1     YOURCOLL2       SYS    社員のパーソナル・データ
      YOURCOLL3       SYS    ジョブの分類/要件
      YOURCOLL4       SYS    会社の保険
```

6. 実行キーを押します。

   「テーブルの選択および順序づけ」画面が表示され、YOURCOLL2 スキーマの中にある表 (テーブル) が表示されます。

7. PEOPLE 表の横の *SEQ* 列に 1 を入力します。

```
                        テーブルの選択および順序づけ

テーブルを選択するためには、順序番号 (1-999) を入力して実行キーを押してください。

SEQ  テーブル                コレクション タイプ テキスト
     EMPLCO                  YOURCOLL2     TAB   社員の会社データ
1    PEOPLE                  YOURCOLL2     TAB   社員の個人データ
     EMPLEXP                 YOURCOLL2     TAB   社員の経歴
     EMPLEVL                 YOURCOLL2     TAB   社員の査定報告書
     EMPLBEN                 YOURCOLL2     TAB   社員の給付金記録
     EMPLMED                 YOURCOLL2     TAB   社員の健康診断記録
     EMPLINVST               YOURCOLL2     TAB   社員の出資記録
```

8. 実行キーを押します。

   「SQL ステートメントの入力」画面が再び現れ、FROM の後 YOURCOLL2.PEOPLE という表名が表示されます。表名は *SQL 命名規則でスキーマ名により修飾されます。

```
                          SQL ステートメントの入力

 SQL ステートメントを入力して，実行キーを押してください。
 ===> SELECT
      FROM YOURCOLL2.PEOPLE _
```

9. SELECT の後にカーソルを置きます。
10. SELECT の後に列名を指定したいので、列のリストを表示するために F18 (列の選択) キーを押します。

    「列の選択および順序づけ」画面が表示され、PEOPLE 表の中にある列が表示されます。

11. *NAME* 列の横の *SEQ* 列に 2 を入力します。
12. *SOCSEC* 列の横の *SEQ* 列に 1 を入力します。

```
                          列の選択および順序づけ

列を選択するためには、順序番号 (1-999) を入力して、実行キーを押してください。

SEQ  列          テーブル          タイプ                長さ      SCALE
2    NAME        PEOPLE          CHARACTER               6
     EMPLNO      PEOPLE          CHARACTER              30
1    SOCSEC      PEOPLE          CHARACTER              11
     STRADDR     PEOPLE          CHARACTER              30
     CITY        PEOPLE          CHARACTER              20
     ZIP         PEOPLE          CHARACTER               9
     PHONE       PEOPLE          CHARACTER              20
```

13. 実行キーを押します。

    「SQL ステートメントの入力」画面が再び現れ、SELECT の後に、SOCSEC、NAME が表示されます。

```
                        SQL ステートメントの入力

SQL ステートメントを入力して，実行キーを押してください。
===> SELECT SOCSEC, NAME
     FROM YOURCOLL2.PEOPLE
```

14. 実行キーを押します。

    作成したステートメントはこれで実行されます。

いったんリスト機能を使用すると、選択した値は、その値を変更するか、あるいは「セッション属性の変更」画面でスキーマのリストを変更するまで、引き続き効力があります。

セッション・サービスの説明

「対話式 SQL セッション・サービス」画面は、「SQL ステートメントの入力」画面で F13 キーを押すと表示されます。

この画面からは、セッション属性を変更し、セッションを印刷、消去、またはソース・ファイルに保管することができます。

オプション 1 (セッション属性の変更) を選択すると、「セッション属性の変更」画面が表示されるので、対話式 SQL セッションで効力を持つ現行値を選択することができます。この画面で表示されているこれらのオプションは、選択したステートメント処理オプションに基づいて変更されます。

次のセッション属性を変更することができます。

- コミットメント制御属性
- ステートメント処理制御
- SELECT 出力装置
- スキーマのリスト
- すべてのシステム・オブジェクトと SQL オブジェクトを選択するのか、あるいはユーザーの SQL オブジェクトだけを選択するのかを指定するリスト・タイプ
- データを表示するときのデータ再表示オプション
- データのコピー許可オプション
- 命名オプション
- プログラム言語
- 日付形式
- 時刻形式
- 日付区切り記号
- 時刻区切り記号
- 小数点表示
- SQL ストリング区切り文字
- 分類順序

- 言語識別コード

オプション 2 (現行セッションの印刷) を選択すると、「印刷装置の変更」画面が表示されるので、現行セッションを即時に印刷して作業を続けることができます。印刷装置に関する情報がプロンプトで要求されます。入力したすべての SQL ステートメントと表示されたすべてのメッセージが、「SQL ステートメントの入力」画面に表示されたとおりに印刷されます。

オプション 3 (現行セッションからのすべての項目の除去) を選択すると、「SQL ステートメントの入力」画面およびセッション活動記録からすべての SQL ステートメントとメッセージを除去することができます。情報を本当に削除してよいかの確認を求めるプロンプトが出されます。

オプション 4 (セッションをソース・ファイルに保管) を選択すると、「ソース・ファイルの変更」画面が表示されるので、セッションをソース・ファイルに保管することができます。ソース・ファイル名を求めるプロンプトが出されます。この機能を使用すると、原始ステートメント入力ユーティリティー (SEU) の使用によってソース・ファイルをホスト言語プログラムに組み込むことができます。

注: オプション 4 を使用すれば、プロトタイプ SQL ステートメントを、SQL を使用している高水準言語 (HLL) プログラムに組み込むことができます。オプション 4 によって作成したソース・ファイルは、SQL ステートメント実行 (RUNSQLSTM) コマンドの入力ソース・ファイルとして編集し使用することができます。

対話式 SQL の終了

「SQLステートメントの入力」画面で F3 (終了) キーを押すと、対話式 SQL 環境から出ることができ、次のいずれかを行うことができます。

1. セッションを保管し終了する。対話式 SQL を終了します。現行セッションは保管され、次に対話式 SQL を開始するときに使用されます。
2. セッションを保管せずに終了する。ユーザーのセッションを保管せずに対話式 SQL を終了します。
3. セッションを再開する。対話式 SQL に入ったまま「SQL ステートメントの入力」画面にします。現行セッション・パラメーターは効力を持続します。
4. ソース・ファイルにセッションを保管する。現行セッションをソース・ファイルに保管します。「ソース・ファイル変更」画面が表示され、セッションをどこに保管するかを選択できるようにします。このセッションを再び対話式 SQL で回復して処理することはできません。

注:

1. オプション 4 を使用すれば、プロトタイプ SQL ステートメントを、SQL を使用している高水準言語 (HLL) プログラムに組み込むことができます。原始ステートメント入力ユーティリティー (SEU) を使用して、これらのステートメントをユーザーのプログラムにコピーします。ソース・ファイルは編集して、SQL ステートメント実行 (RUNSQLSTM) コマンドの入力ソース・ファイルとして使用することもできます。

2. いくつかの行を変更した後で、この作業単位に対して現在、ロックが保持されているときに対話式 SQL を終了しようとすると、警告メッセージが表示されます。

既存の SQL セッションの使用

「対話式 SQL の終了」画面からオプション 1 (セッションを保管して終了) を使用して対話式 SQL セッションを 1 つだけ保管した場合は、そのセッションはどのワークステーションからでも再開することができます。しかし、オプション 1 を使用して別々のワークステーションで 2 つ以上のセッションを保管した場合には、対話式 SQL は使用しているワークステーションに一致するセッションを最初に再開しようとします。一致するセッションが使用可能でない場合、対話式 SQL は検索範囲を広げ、ユーザー ID に所属するすべてのセッションを含めます。ユーザー ID に使用可能なセッションがない場合、システムはユーザー ID および現行のワークステーションに合わせて新しいセッションを作成します。

たとえば、あるセッションをワークステーション 1 に、別のセッションをワークステーション 2 に保管して、現在ワークステーション 1 で作業しているとします。対話式 SQL はまずワークステーション 1 に保管されたセッションを再開しようとします。そのセッションが現在使用中であれば、次に対話式 SQL はワークステーション 2 に保管されたセッションを再開しようと試みます。そのセッションも使用中である場合、システムはワークステーション 1 に 2 つ目のセッションを作成します。

しかし、ワークステーション 3 で作業中でワークステーション 2 に対応付けられる ISQL セッションを使用したい場合があるかもしれません。この場合、まず「対話式 SQL の終了」画面のオプション 2 (セッションを保管しないで終了) を使ってワークステーション 1 からセッションを削除する必要があるかもしれません。

SQL セッションの回復

前の SQL セッションが異常終了した場合は、対話式 SQL は、次のセッションの開始時に (次の STRSQL コマンドが入力されたとき)「SQL セッションの回復」画面を表示します。この画面から次のいずれかを行うことができます。

- オプション 1 (既存の SQL セッションの再開の試行) を選択することにより、前のセッションを回復する。
- オプション 2 (既存の SQL セッションの削除および新規のセッションの開始) を選択することにより、前のセッションを削除し、新規のセッションを開始する。

旧セッションを削除し新しいセッションに移ることを選択した場合には、STRSQL の入力時に指定したパラメーターが使用されます。旧セッションを回復することを選択した場合、あるいは以前に保管したセッションに入る場合は、STRSQL 入力時に指定したパラメーターは無視され、旧セッションからのパラメーターが使用されます。どのパラメーターが、指定した値から旧セッションの値に変更されたかを示すメッセージが返されます。

対話式 SQL による遠隔データベースのアクセス

対話式 SQL では、SQL の CONNECT ステートメントを使用して遠隔リレーショナル・データベースと通信することができます。対話式 SQL は CONNECT ステー

トメントに対して CONNECT (タイプ 2) 意味体系 (分散作業単位) を使用します。対話式 SQL は、SQL セッションを開始するときにローカル RDB に暗黙の接続を行います。 CONNECT ステートメントが完了すると、確立されたリレーショナル・データベース接続を示すメッセージが表示されます。新しいセッションを開始するときに COMMIT (*NONE) が指定されていない場合、または保管されたセッションを復元するときに、一緒に保管されているコミット・レベルが *NONE ではない場合は、接続はコミットメント制御で登録されます。この暗黙接続および予測されるコミットメント制御登録はその後の遠隔データベースとの接続に影響する場合があります。詳細は、372 ページの『接続タイプの決定』を参照してください。遠隔システムに接続する前に、次のことを行ってください。

- 作業の分散単位をサポートしないアプリケーション・サーバーに接続するときは、RELEASE ALL、およびその後に続けて COMMIT を発行し、ローカルとの暗黙接続を含む前の接続を終了するようにしてください。
- 非 DB2 UDB for iSeries アプリケーション・サーバーに接続するときは、RELEASE ALL、およびその後に続けて COMMIT を発行し、ローカルとの暗黙接続を含む前の接続を終了し、コミットメント制御レベルを最終的に *CHG に変更してください。

DB2 UDB for iSeries 以外のアプリケーション・サーバーと接続するときは、一部のセッション属性は、そのアプリケーション・サーバーがサポートしている属性に変更されます。次の表は、変更される属性を示しています。

表 36. セッション属性の値

セッション属性	元の値	新しい値
日付の形式	*YMD *DMY *MDY *JUL	*ISO *EUR *USA *USA
時刻形式	*HMS (区切り記号 (:) 付き) *HMS (他の区切り記号付き)	*JIS *EUR
コミットメント制御	*CHG, *NONE *ALL	*CS 反復可能読み取り
命名規則	*SYS	*SQL
データ・コピー使用可	*NO, *YES	*OPTIMIZE
データ再表示	*ALWAYS	*FORWARD
小数点	*SYSVAL	*PERIOD
分類順序	*HEX 以外の値	*HEX

注:

1. バージョン 2 リリース 3 より前のリリースで稼動しているサーバーに接続している場合には、分類順序の値は *HEX に変更されます。
2. DB2/2 または DB2/6000 アプリケーション・サーバーに接続するときは、指定される日付と時刻の形式は同じでなければなりません。

接続が完了すると、そのセッション属性が変更されたことを知らせるメッセージが返されます。変更されたセッション属性は、「セッション・サービス」画面を使用

すると、表示することができます。対話式 SQL を実行しているときは、省略時活性化グループに対して他の接続を確立することはできません。

対話式 SQL で遠隔システムに接続されるときは、構文専用のステートメント処理モードがステートメントの構文を遠隔システムではなくローカル・システムがサポートしている構文と照らし合わせて検査します。同様に、SQL プロンプト機能とリスト・サポートは、ローカル・システムがサポートするステートメント構文と命名規則を使用します。このステートメントは実行されますが、ただし遠隔システム上で実行されます。 2 つのシステム間の SQL サポート・レベルの違いのために、実行時に構文エラーが遠隔システム上のステートメントで検出される可能性があります。

スキーマと表のリストは、ローカル・リレーショナル・データベースに接続しているときに利用できます。列のリストは、DESCRIBE TABLE ステートメントをサポートするリレーショナル・データベース・マネージャーに接続しているときに限り利用できます。

変更保留中の接続状態または保護会話を使用している接続状態で、対話式 SQL を終了する場合、その接続がそのまま残ります。これらの接続に対してそれ以上の処理を行わなければ、これらの接続は次の COMMIT 操作または ROLLBACK 操作時に終了します。または、対話式 SQL を終了する前に RELEASE ALL および COMMIT を行って接続を終了することもできます。

対話式 SQL を使用して非 DB2 UDB for iSeries アプリケーション・サーバーにアクセスするためには、若干のセットアップが必要になる場合があります。詳細については、分散データベース・プログラミングを参照してください。

注: 通信トレースの出力には、場合によっては 'CREATE TABLE XXX' ステートメントへの参照があります。これは、パッケージの存在を判別するために使用されます。通常の処理の一部なので、無視しても構いません。

第 18 章 SQL ステートメント処理プログラムの使用

このセクションでは、SQL ステートメント処理プログラムについて説明します。この処理プログラムは、SQL ステートメント実行 (RUNSQLSTM) コマンドを使用する場合に、使用可能になります。

SQL ステートメント処理プログラムを使用すると、SQL ステートメントをソース・メンバーから実行することができます。ソース・メンバーの中のステートメントは、ソース・メンバーをコンパイルせずに、繰り返し実行したり、変更したりすることができます。これによって、データベース環境のセットアップは容易になります。SQL ステートメント処理プログラムで使用できるステートメントは、次のとおりです。

- ALTER TABLE
- CALL
- COMMENT ON
- COMMIT
- CREATE ALIAS
- CREATE DISTINCT TYPE
- CREATE FUNCTION
- CREATE INDEX
- CREATE PROCEDURE
- CREATE SCHEMA
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- DECLARE GLOBAL TEMPORARY TABLE
- DELETE
- DROP
- GRANT (機能またはプロシージャー特権)
- GRANT (パッケージ特権)
- GRANT (表特権)
- GRANT (ユーザー定義のタイプ特権)
- INSERT
- LABEL ON
- LOCK TABLE
- RELEASE SAVEPOINT
- RENAME
- REVOKE (機能またはプロシージャー特権)
- REVOKE (パッケージ特権)

- REVOKE (表特権)
- REVOKE (ユーザー定義のタイプ特権)
- ROLLBACK
- SAVEPOINT
- SET PATH
- SET SCHEMA
- SET TRANSACTION
- UPDATE

ソース・メンバーの中で、各ステートメントはセミコロンで終了します。また、**EXEC SQL** では始まりません。ソース・メンバーのレコード長が 80 より長い場合、最初の 80 文字だけが読み取られます。ソース・メンバー中の注釈は、行の注釈またはブロック注釈のどちらでも使用できます。行の注釈は、二重ハイフン (--) で始まり、行の終わりで終わらなければなりません。ブロックの注釈は /* で始まり、次の */ に到達するまで多数の行にわたって継続できます。ブロックの注釈はネストすることができます。ソース・ファイルの中に入れることができるのは、SQL ステートメントと注釈だけです。SQL ステートメントに関する出力リストと処理結果のメッセージは、印刷ファイルに送られます。デフォルトの印刷ファイルは QSYSPRT です。

ソース・メンバー内のすべてのステートメント上で構文検査だけを実行するには、RUNSQLSTM コマンドで PROCESS(*SYN) パラメーターを指定します。

詳細については、以下のセクションを参照してください。

- 『エラーが発生した後のステートメントの実行』
- 『SQL ステートメント処理プログラムでのコミットメント制御』
- 329 ページの『SQL ステートメント処理プログラムのスキーマ』
- 330 ページの『SQL ステートメント処理プログラムのソース・メンバー・リスト』

エラーが発生した後のステートメントの実行

ステートメントが、RUNSQLSTM コマンドのエラー・レベル (ERRLVL) パラメーターに指定した値を超える重大度のエラーを戻した場合は、そのステートメントは失敗しています。ソース・メンバー内の残りのステートメントは解析され、構文エラーが検査されますが、実行はされません。大部分の SQL エラーの重大度は 30 です。 SQL ステートメントが失敗した後も処理を続行したい場合は、RUNSQLSTM コマンドの ERRLVL パラメーターを 30 以上にセットしてください。

SQL ステートメント処理プログラムでのコミットメント制御

コミットメント制御レベルは RUNSQLSTM コマンドで指定します。 *NONE 以外のコミットメント制御レベルを指定すると、SQL ステートメントはコミットメント制御の下で実行されます。すべてのステートメントが正常に実行されると、SQL ステートメント処理プログラムの最後に COMMIT が行われます。それ以外の場合

は、ROLLBACK が行われます。ステートメントは、戻りコード重大度が RUNSQLSTM コマンドの ERRLVL パラメーターに指定した値以下の場合に、正常に実行されたと見なされます。

SET TRANSACTION ステートメントをソース・メンバーの中で使用して、RUNSQLSTM コマンドで指定されたコミットメント制御のレベルを一時変更することができます。

注: コミットメント制御を指定して SQL ステートメント処理プログラムを使用するときは、ジョブは作業単位の境界になければなりません。

SQL ステートメント処理プログラムのスキーマ

SQL ステートメント処理プログラムは、CREATE SCHEMA ステートメントをサポートします。これは、2 つの異なるセクションからなると考えられる複合ステートメントです。最初のセクションは、スキーマのコレクションを定義します。 2 番目のセクションには、コレクションの中のオブジェクトを定義する DDL ステートメントが入ります。

最初のセクションは、以下の 2 つの方法のいずれかで作成することができます。

- CREATE SCHEMA *コレクション名*

  コレクションは指定したコレクション名を使用して作成されます。
- CREATE SCHEMA AUTHORIZATION 許可名

  コレクションは、コレクション名として許可名を使用して作成されます。スキーマを実行するユーザーには、ユーザー・プロファイルに対する権限がなければなりません。この権限の名前を**許可名**といいます。

  ステートメントの許可名が保持する特権には、以下の権限が含まれていなければなりません。
  - CREATE COLLECTION ステートメントを実行する権限
  - CREATE SCHEMA 内の各 SQL ステートメントを実行する権限

CREATE SCHEMA ステートメントの 2 番目のセクションには、ゼロから任意の個数の以下のステートメントを入れることができます。

- COMMENT ON
- CREATE ALIAS
- CREATE DISTINCT TYPE
- CREATE INDEX
- CREATE TABLE
- CREATE VIEW
- GRANT (表特権)
- GRANT (ユーザー定義のタイプ特権)
- LABEL ON

これらのステートメントは、CREATE SCHEMA ステートメントの最初のセクションのすぐ後に続きます。各ステートメントと各セクションはセミコロンで区切り**ま**

せん。他の SQL ステートメントがこのスキーマ定義に続く場合は、スキーマ内の最後のステートメントをセミコロンで終了しなければなりません。

スキーマ・ステートメントの 2 番目の部分で作成または参照するすべてのオブジェクトは、スキーマのために作成済みのコレクションの中になければなりません。すべての非修飾参照は、作成済みのコレクションによって暗黙に修飾されます。すべての修飾参照は、作成済みのコレクションによって修飾しなければなりません。

SQL ステートメント処理プログラムのソース・メンバー・リスト

コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

```
5722ST1 V5R2M0 020719              SQL ステートメントの実行         SCHEMA                        08/06/02 15:35:18   Page   1
ソース・ファイル.....CORPDATA/SRC
メンバー.................SCHEMA
コミット.................*NONE
命名.....................*SYS
生成レベル...............10
日付の形式...............*JOB
日付区切り記号...........*JOB
時刻の形式...............*HMS
時刻区切り記号...........*JOB
省略時のコレクション.....*NONE
IBM SQL フラグづけ.......*NOFLAG
ANS フラグづけ...........*NONE
小数点...................*JOB
ソート順序...............*JOB
言語.....................*JOB
印刷装置ファイル.....*LIBL/QSYSPRT
ソース・ファイルの CCSID..65535
ジョブの CCSID...........0
ステートメント処理.......*RUN
データのコピー可能.......*OPTIMIZE
ブロック化可能...........*READ
04/01/98  11:54:10 にソース・メンバーが変更された。
```

図 8. SQL ステートメント処理プログラムの QSYSPRT リスト (1/3)

```
5722ST1 V5R2M0 020719              SQL ステートメントの実行       SCHEMA                          08/06/02 15:35:18   Page   2
レコード*...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8   SEQNBR  最終変更
    1
    2     DROP COLLECTION DEPT;
    3     DROP COLLECTION MANAGER;
    4
    5     CREATE SCHEMA DEPT
    6          CREATE TABLE EMP (EMPNAME CHAR(50), EMPNBR INT)
    7                           -- EMP will be created in collection DEPT
    8          CREATE INDEX EMPIND ON EMP(EMPNBR)
    9                           -- EMPIND will be created in DEPT
   10          GRANT SELECT ON EMP TO PUBLIC;  -- grant authority
   11
   12     INSERT INTO DEPT/EMP VALUES('JOHN SMITH', 1234);
   13                           /* table must be qualified since no
   14                              longer in the schema */
   15
   16     CREATE SCHEMA AUTHORIZATION MANAGER
   17                            -- this schema will use MANAGER's
   18                            -- user profile
   19           CREATE TABLE EMP_SALARY (EMPNBR INT, SALARY DECIMAL(7,2),
   20                                    LEVEL CHAR(10))
   21           CREATE VIEW LEVEL AS SELECT EMPNBR, LEVEL
   22                FROM EMP_SALARY
   23           CREATE INDEX SALARYIND ON EMP_SALARY(EMPNBR,SALARY)
   24
   25           GRANT ALL ON LEVEL TO JONES GRANT SELECT ON EMP_SALARY TO CLERK
   26                   -- Two statements can be on the same line
* * * * *  ソ ー ス の 終 わ り  * * * * *
```

図 8. SQL ステートメント処理プログラムの QSYSPRT リスト (2/3)

```
5722ST1 V5R2M0 020719              SQL ステートメントの実行       SCHEMA                          08/06/02 15:35:18   Page     3
レコード*...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8   SEQNBR  最終変更
MSG ID  SEV レコード テキスト
SQL7953   0       1  桁 1 QSYS の DEPT の削除が完了した。
SQL7953   0       3  桁 3 QSYS の MANAGER の削除が完了した。
SQL7952   0       5  桁 3 コレクション DEPT が作成された。
SQL7950   0       6  桁 8 テーブル EMP が DEPT に作成された。
SQL7954   0       8  桁 8 DEPT のテーブル EMP に DEPT の索引 EMPIND が
                     作成されました。
SQL7966   0      10  桁 8 DEPT の EMP に対する権限の GRANT が完了した。
SQL7956   0      10  桁 40 DEPT の EMP に 1 行が挿入された。
SQL7952   0      13  桁 28 コレクション MANAGER が作成された。
SQL7950   0      19  桁 9 テーブル EMP_SALARY が MANAGER に作成された。
SQL7951   0      21  桁 9 ビュー LEVEL が MANAGER に作成された。
SQL7954   0      23  桁 9 MANAGER のテーブル EMP_SALARY に MANAGER
                     の索引 SALARYIND が作成された。
SQL7966   0      25  桁 9 MANAGER の LEVEL に対する権限の GRANT が完了した。
SQL7966   0      25  桁 37 MANAGER の EMP_SALARY に対する権限の GRANT
                     が完了した。

メッセージの要約
 合計    通知      警告     エラー    重大度    端末装置
   13     13         0        0         0          0
ソースに 00 レベルの重大度エラーが見つかった。
* * * * *  リ ス ト の 終 わ り  * * * * *
```

図 8. SQL ステートメント処理プログラムの QSYSPRT リスト (3/3)

第 19 章 DB2 UDB for iSeries のデータ保護

この章では、SQL のデータを権限のないユーザーから保護する機密保護の計画、およびデータ保全性を確保する方法について説明します。詳細については、以下のトピックを参照してください。

- 『SQL オブジェクトの機密保護』
- 335 ページの『iSeries ナビゲーターを使用したデータの保護』
- 336 ページの『データ保全性』

コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

SQL オブジェクトの機密保護

SQL オブジェクトも含めて、サーバー上のオブジェクトはすべて、システムの機密保護機能によって管理されます。ユーザーは、SQL GRANT ステートメントと REVOKE ステートメントまたは CL コマンドのオブジェクト権限編集 (EDTOBJAUT)、オブジェクト権限認可 (GRTOBJAUT)、およびオブジェクト権限取り消し (RVKOBJAUT) のいずれかにより SQL オブジェクトを認可することができます。システム機密保護および GRTOBJAUT コマンドと RVKOBJAUT コマンドの使用の詳細については、iSeries 機密保護解説書 を参照してください。

SQL の GRANT ステートメントおよび REVOKE ステートメントは、SQL パッケージ、SQL プロシージャー、表、視点、および表と視点の個々の列を対象に作動します。さらに、SQL GRANT ステートメントおよび REVOKE ステートメントは私用および共通権限しか認可しません。場合によっては、コマンドやプログラムなどの他のオブジェクトを使用する権限をユーザーに認可するために EDTOBJAUT、GRTOBJAUT、および RVKOBJAUT を使用する必要があります。

GRANT および REVOKE ステートメントの詳細については、SQL 解説書を参照してください。

SQL ステートメントに対する権限の検査は、ステートメントが静的であるか、動的であるか、対話方式で実行されるかによって異なります。

静的 SQL ステートメントの場合の検査は次のとおりです。

- USRPRF 値が *USER の場合は、SQL ステートメントをローカルに実行する権限は、プログラムを実行しているユーザーのユーザー・プロファイルを使用して検査されます。 SQL ステートメントを遠隔に実行する権限は、アプリケーション・サーバー側でユーザー・プロファイルを使用して検査されます。*USER はシステム (*SYS) 命名の場合の省略時値です。
- USRPRF 値が *OWNER の場合は、SQL ステートメントをローカルに実行する権限は、プログラムを実行しているユーザーのユーザー・プロファイルとプログラムの所有者のユーザー・プロファイルを使用して検査されます。 SQL ステートメントを遠隔に実行する権限は、アプリケーション・サーバー・ジョブのユー

ザー・プロファイルと SQL パッケージの所有者のユーザー・プロファイルを使用して検査されます。 より高い方の権限が使用されます。SQL (*SQL) 命名の場合、*OWNER が省略時値です。

動的 SQL ステートメントの場合の検査は次のとおりです。

- USRPRF 値が *USER の場合は、SQL ステートメントをローカルに実行する権限は、プログラムを実行しているユーザーのユーザー・プロファイルを使用して検査されます。 SQL ステートメントを遠隔に実行する権限は、アプリケーション・サーバー・ジョブのユーザー・プロファイルを使用して検査されます。
- USRPRF 値が *OWNER で DYNUSRPRF が *USER の場合は、SQL ステートメントをローカルに実行する権限は、プログラムを実行しているユーザーのユーザー・プロファイルを使用して検査されます。 SQL ステートメントを遠隔に実行する権限は、アプリケーション・サーバー・ジョブのユーザー・プロファイルを使用して検査されます。
- USRPRF 値が *OWNER で DYNUSRPRF が *OWNER の場合は、SQL ステートメントをローカルに実行する権限は、プログラムを実行しているユーザーのユーザー・プロファイルとプログラムの所有者のユーザー・プロファイルを使用して検査されます。 SQL ステートメントを遠隔に実行する権限は、アプリケーション・サーバー・ジョブのユーザー・プロファイルと SQL パッケージの所有者のユーザー・プロファイルを使用して検査されます。 最も高い権限が使用されます。機密保護の観点から、DYNUSRPRF のパラメーター値 *OWNER は注意して使用しなければなりません。このオプションは、プログラムまたはパッケージの所有者のアクセス権限をプログラムを実行するユーザーに与えます。

対話式 SQL ステートメントの場合、権限は、ステートメントを処理している人の権限に対して検査されます。借用権限は、対話式 SQL ステートメントには使用されません。

権限 ID

権限 ID は、ユーザーを個々に識別するものであり、サーバー上のユーザー・プロファイル・オブジェクトです。権限 ID は、システムのユーザー・プロファイル作成 (CRTUSRPRF) コマンドを使用して作成することができます。

視点

視点は、権限のないユーザーが機密データにアクセスするのを防止します。アプリケーション・プログラムは、表内の機密またはアクセス制限データに対するアクセス権がなくても、その表内の必要とするデータをアクセスすることができます。視点は、SELECT リストの中に特定の列 (たとえば、社員の給与) を指定しないことにより、その特定の列に対するアクセスを制限することができます。視点は WHERE 文節を指定すること (たとえば、特定の部門番号に関連する行に対してのみアクセスを許容すること) により、表内の特定の行に対するアクセスを制限することができます。

監査

DB2 UDB for iSeries は、米国政府の C2 機密保護レベルに準拠するように設計されています。このレベルの主な機能は、システム上の動作を監査できることです。DB2 UDB for iSeries は、システム機密保護機能が管理する監査機能を使用しま

す。 監査は、オブジェクト・レベル、ユーザー・レベル、またはシステム・レベルで実行することができます。 システム値 QAUDCTL は、監査をオブジェクト・レベルで実行するかユーザー・レベルで実行するかを制御します。 ユーザー監査変更 (CHGUSRAUD) コマンドおよびオブジェクト監査変更 (CHGOBJAUD) コマンドは、どのユーザーおよびオブジェクトを監査するかを指定します。 システム値 QAUDLVL は、どのタイプの動作 (たとえば、権限の失敗、作成、削除、認可、取り消しなど) を監査するかを制御します。 監査の詳細については、iSeries 機密保護解説書  を参照してください。

DB2 UDB for iSeries は、DB2 UDB for iSeries ジャーナル・サポートを使用して、行の変更を監査できます。

場合によっては、監査ジャーナル内の項目が発生順に並べられないこともあります。たとえば、コミットメント制御の下で実行されているジョブが表を削除し、削除した表と同じ名前の新規表を作成してからコミットする場合です。この場合、監査ジャーナルでは作成が先に、そして削除が後に記録されることになります。これは、作成されたオブジェクトがただちにジャーナルされるためです。コミットメント制御の下で削除されたオブジェクトは隠蔽され、コミットが実行されるまで実際には削除されません。コミットが実行されると、そのアクションがジャーナルされます。

iSeries ナビゲーターを使用したデータの保護

iSeries ナビゲーターを使用し、以下を実行することによってデータを保護することができます。

- 『オブジェクトの共通権限の定義』
- 336 ページの『新規オブジェクト用の省略時の共通権限のセットアップ』
- 336 ページの『オブジェクトに対するユーザーまたはグループの許可』

オブジェクトの共通権限の定義

共通権限は、オブジェクトに対して特定のアクセスを持っていないユーザーが使用するアクセスのタイプを記述するために、システム上の各オブジェクトに定義されます。共通権限を定義するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. 許可を編集したいオブジェクトが見えるまでナビゲートする。
3. 許可を追加したいオブジェクトを右クリックし、「**許可**」を選択する。
4. 「**許可**」ダイアログで、グループ・リストから「**共通**」を選択する。
5. 「**詳細**」ボタンをクリックし、詳細許可をインプリメントする。
6. 該当するチェック・ボックスのボックスにチェックを付けて、必要な許可を共通権限に適用する。
7. 「**OK**」をクリックする。

新規オブジェクト用の省略時の共通権限のセットアップ

省略時の共通権限を設定することによって、新規オブジェクトがライブラリーに作成されるときにすべての新規オブジェクトに割り当てられる共通権限を持つことができます。異なるレベルの機密保護が必要なオブジェクトには、個々に、許可を編集することができます。省略時の共通権限をセットするには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. 共通権限をセットしたいライブラリーを右クリックし、「**許可**」を選択する。
3. 「**許可**」ダイアログで、「**新規オブジェクト**」をクリックする。
4. 「**新規オブジェクト**」ダイアログで、省略時の共通権限を選択する。権限リストを割り当てるには、権限リストの名前を入力するか「**表示**」します。権限リストのプロパティーを表示するには、「**オープン**」を選択します。
5. 「**OK**」をクリックする。

オブジェクトに対するユーザーまたはグループの許可

ユーザーによっては、オブジェクトに対して共通権限での許可とは別の権限が必要になることがあります。オブジェクトに対してユーザーまたはグループを許可するには、以下のようにします。

1. 「**iSeries ナビゲーター**」ウィンドウで、ユーザーのサーバー → 「**データベース**」→ 処理したいデータベース → 「**ライブラリー**」の順に展開する。
2. 許可を編集したいオブジェクトが見えるまでナビゲートする。
3. 許可を追加したいオブジェクトを右クリックし、「**許可**」を選択する。
4. 「**許可**」ダイアログで、「**追加**」をクリックする。
5. 「**追加**」ダイアログで、1 つまたは複数のユーザーおよびグループを選択するか、ユーザー名フィールドにユーザー名を、または、グループ名フィールドにグループ名を入力する。
6. 「**OK**」をクリックする。

これで、ユーザーまたはグループが、リストのトップに追加されます。

注: ユーザーまたはグループは、オブジェクトに対する省略時の権限が与えられます。ユーザーの権限は、システムで定義されているタイプのいずれかに変更するか、カスタマイズすることができます。

オブジェクトからユーザーの許可を除去するには、以下のようにします。

1. 除去したいユーザーまたはグループを選択する。
2. 「**除去**」をクリックする。

データ保全性

データ保全性は、無許可の人、システム操作またはハードウェア障害 (ディスクの物理的な損傷など)、プログラミング・エラー、ジョブの完了前の中断 (電源障害など)、あるいはアプリケーションの同時実行による妨害 (逐次化の問題) などによるデータの破壊または変更を防ぎます。データ保全性は次の機能により確実なものにされます。

- 『並行性』
- 339 ページの『ジャーナル処理』
- 340 ページの『コミットメント制御』
- 347 ページの『アトミック・オペレーション』
- 348 ページの『制約』
- 349 ページの『保管/復元』
- 350 ページの『耐損傷性』
- 350 ページの『索引回復』
- 351 ページの『カタログの保全性』
- 352 ページの『ユーザー補助記憶域プール (ASP)』
- 352 ページの『独立補助記憶域プール (IASP)』

コミットメント制御のトピック、ジャーナル管理のトピック、資料「データベース・プログラミング」、資料「バックアップおよび回復の手引き」  に、これらの機能のそれぞれについて詳細な説明があります。

並行性

並行性とは、同じ表または視点内のデータを複数のユーザーが同時にアクセスし変更しても、データの保全性が失われないようにするための機能です。この機能は、DB2 UDB for iSeries データベース・マネージャーによって自動的に提供されています。 並行ユーザーが同じデータを同時に変更することのないように、表または行が暗黙にロックされます。

通常、DB2 UDB for iSeries は、保全性を保つために行をロックします。しかし、ある状態では、DB2 UDB for iSeries は、行ロックではなく、さらに排他的な表レベルのロックが必要な場合があります。詳細については、 340 ページの『コミットメント制御』を参照してください。

たとえば、あるカーソルが現在保持する行の更新 (排他) ロックは、同じプログラム (またはカーソルと関連しない DELETE または UPDATE ステートメント) にある別のカーソルによって獲得されることがあります。これによって、別の FETCH が実行されるまで、最初のカーソルを参照する位置づけられた UPDATE または DELETE ステートメントを防ぎます。あるカーソルが現在保持する行の読み取り (共有、非更新) ロックは、同じプログラム (または DELETE または UPDATE ステートメント) にある別のカーソルが同じ行のロックを獲得するのを妨げません。

省略時およびユーザーが指定できるロック待機タイムアウト値がサポートされています。 DB2 UDB for iSeries は、省略時のレコード待機時間 (60 秒) および省略時のファイル待機時間 (*IMMED) を指定して表、視点、および索引を作成します。このロックの待機時間は、DML ステートメントに使用されます。これらの値は、CL コマンドの物理ファイル変更 (CHGPF)、論理ファイル変更 (CHGLF)、およびデータベース・ファイル一時変更 (OVRDBF) を使用して変更することができます。

すべての DDL ステートメントおよび LOCK TABLE ステートメントに使用されるロック待機時間は、ジョブ省略時待機時間 (DFTWAIT) です。この値は、CL コマンドのジョブの変更 (CHGJOB) またはクラスの変更 (CHGCLS) を使って変更できます。

大きなレコードの待機時間が指定されるイベントでは、デッドロック検出が備わっています。たとえば、あるジョブで行 1 に排他ロックがあり、別のジョブで行 2 に排他ロックがあるとします。最初のジョブが行 2 をロックしようとしても、2 番目のジョブがロックを保持しているので待機することになります。2 番目のジョブが行 1 をロックしようとすると、DB2 UDB for iSeries は 2 つのジョブがデッドロックに入っていることを検出し、2 番目のジョブにはエラーが戻されます。

SQL LOCK TABLE ステートメントを使用すれば、ある表をユーザーが同時に使用するのを明示的に禁止することができます。これについては、SQL 解説書に説明があります。 COMMIT(*RR) を使用しても、作業単位中に他のユーザーが表を使用するのを防ぐすることができます。

パフォーマンスを向上させるために、DB2 UDB for iSeries では頻繁にオープン・データ・パス (ODP) を開いたままにします (詳細は、データベース・パフォーマンスおよび Query 最適化を参照)。このパフォーマンス機能では、さらに ODP が参照する表のロックもそのまま残しておきますが、行のロックを残すことはしません。表に残されたロックにより、別のジョブがその表で操作を実行できないことがあります。しかし、ほとんどの場合、DB2 UDB for iSeries は他のジョブがロックを保持していることを検出し、イベントがそれらのジョブに通知されます。イベントにより、DB2 UDB for iSeries は、その表に関連し、現在パフォーマンスだけの目的で開いている ODP があればクローズ (および表のロックを解放) します。ロックの待機タイムアウトは、イベントがシグナルされて他のジョブが ODP をクローズするのに十分な長さでなければならず、そうでない場合エラーが戻されることに注意してください。

LOCK TABLE ステートメントを使用して表のロックを獲得するか、または COMMIT(*ALL) あるいは COMMIT(*RR) を使用しない限り、1 つのジョブで読み取られたデータが即時に別のジョブで変更される可能性があります。 通常、データは SQL ステートメントの実行時 (たとえば、FETCH 時) に読み取られるので、データはまさに最新であると言えます。ただし、次の場合には、データは SQL ステートメントの実行前 (たとえば、OPEN 時) に読み取られるので、データは最新でない可能性があります。

- ALWCPYDTA(*OPTIMIZE) が指定され、最適化プログラムがデータのコピーを作成する方がコピーを作成しない場合よりも良いと判断した場合。
- 照会によっては、データベース・マネージャーによる一時的な結果表の作成が必要になる場合があります。一時的な結果表のデータは、カーソルのオープン後に行われた変更を反映しません。一時的な結果表が必要になるのは、次の場合です。
  - ORDER BY 文節に指定した列に対する記憶域の合計長が 2000 バイトを超えている場合。
  - ORDER BY 文節と GROUP BY 文節に異なる列を指定するか、あるいは異なる順序で列を指定した場合。
  - UNION 文節または DISTINCT 文節を指定した場合。

- ORDER BY 文節または GROUP BY 文節に指定した列のすべてが同じ表の列でない場合。
- JOINDFT データ定義仕様 (DDS) キーワードによって定義した論理ファイルを別のファイルに結合する場合。
- 複数のデータベース・ファイル・メンバーを基礎とする論理ファイルを結合する場合またはそのファイルについて GROUP BY を指定する場合。
- 照会が結合を含んでおり、その結合されているファイルの中の少なくとも 1 つが GROUP BY 文節を含む視点である場合。
- 照会に、GROUP BY 文節の入っている視点を参照する GROUP BY 文節を含んでいる場合。

• 基本副照会の評価は、照会がオープンされたときに行われます。

ジャーナル処理

DB2 UDB for iSeries ジャーナル・サポートには、監査証跡、正方向回復、および逆方向回復があります。正方向回復を使用すると、ある表の古いバージョンを取り出して、ジャーナルに記録されている変更をその表に適用することができます。逆方向回復を使用すると、ジャーナルに記録されている変更を表から取り除くことができます。

SQL スキーマを 1 つ作成すると、そのスキーマ内にジャーナルおよびジャーナル・レシーバーが作成されます。 SQL がジャーナルおよびジャーナル・レシーバーを作成する場合、ASP 文節を CREATE COLLECTION ステートメントまたは CREATE SCHEMA ステートメントで指定した場合に限り、それらがユーザーの補助記憶域プール (ASP) に作成されます。 ただし、ジャーナル・レシーバーをそれぞれの ASP に置いておくとパフォーマンスが向上するので、ジャーナル管理の担当者は、すべての将来のジャーナル・レシーバーを個別の ASP 上に作成したほうがよい場合もあります。

表が作成され、スキーマに入れられると、その表は、DB2 UDB for iSeries がスキーマ内に作成したジャーナルに自動的にジャーナル処理されます (QSQJRN)。 スキーマ以外に作成された表でも、QSQJRN というジャーナルがそのライブラリーに存在する場合はジャーナル処理が開始されます。 これ以後は、ユーザーがジャーナル機能を使用して、ジャーナルとジャーナル・レシーバー、およびジャーナルの表のジャーナル処理を管理しなければなりません。 たとえば、ある表をあるスキーマに移した場合、ジャーナル処理状況は自動的には変更されません。 表が復元される場合には、ジャーナルに関する通常の規則が適用されます。 すなわち、表が保管時にジャーナル処理されていれば、復元時にも同じジャーナルにジャーナル処理されます。 保管時に表がジャーナル処理されていなければ、復元時にはジャーナル処理されません。

SQL コレクション内に作成されるジャーナルは、通常、SQL の表のすべての変更を記録するために使用されるジャーナルです。しかし、システム・ジャーナル機能を使用すると、SQL の表を別のジャーナルに記録することができます。これは、1 つのスキーマ内の表が別のスキーマ内の表の親である場合に必要となる可能性があります。 親表に更新または削除を行うときに、DB2 UDB for iSeries が、参照制約にある親ファイルと従属ファイルが同じジャーナルにジャーナル処理されることを必要とするからです。

ユーザーは、ジャーナル機能を使用しているどの表についてもジャーナル処理を停止させることができますが、その場合には、アプリケーションをコミットメント制御下で実行することはできなくなります。ジャーナル処理を NO ACTION、CASCADE、SET NULL、または SET DEFAULT の削除規則を指定した参照制約の親表で停止させると、すべての更新操作と削除操作ができなくなります。それ以外は、COMMIT(*NONE) の指定があれば、アプリケーションはまだ機能することができます。しかし、ジャーナル処理およびコミットメント制御から得られるものと同レベルの保全性は得られません。

コミットメント制御

DB2 UDB for iSeries コミットメント制御サポートは、一連のデータベース変更 (更新、挿入、DDL 操作、または削除) を、1 つの作業単位 (トランザクション) として処理する手段を提供します。コミット操作は、これらの一連の操作が完了することを保証します。ロールバック操作は、これらの一連の操作がバックアウトされることを保証します。保管ポイントを使用して、トランザクションを、ロールバック可能なより小さな単位に分けることができます。コミット操作は各種インターフェースを介して出すことができます。 たとえば、次のインターフェースです。

- SQL COMMIT ステートメント
- CL COMMIT コミット
- 言語のコミット・ステートメント (RPG COMMIT ステートメントなど)

ロールバック操作は、いくつかの各種インターフェースを介して出すことができます。たとえば、次のインターフェースです。

- SQL ROLLBACK ステートメント
- CL ROLLBACK コマンド
- 言語のロールバック・ステートメント (RPG ROLBK ステートメント)

次の SQL ステートメントだけは、コミットまたはロールバックすることができません。

- DROP COLLECTION
- GRANT または REVOKE (指定したオブジェクトについて権限保持者が存在する場合)

COMMIT(*NONE) 以外の分離レベルで SQL ステートメントを実行したとき、または RELEASE ステートメントを実行したときに、コミットメント制御がまだ開始されていない場合は、DB2 UDB for iSeries は、CL コマンドのコミットメント制御開始コマンド (STRCMTCTL) を暗黙的に呼び出して、コミットメント制御環境をセットアップします。 DB2 UDB for iSeries は、STRCMTCTLコマンドで、LCKLVL と一緒に NFYOBJ(*NONE) パラメーターおよび CMTSCOPE(*ACTGRP) パラメーターを指定します。 指定した LCKLVL は、CRTSQLxxx、STRSQL、または RUNSQLSTM の各コマンドの COMMIT パラメーター上のロック・レベルです。REXX では、指定した LCKLVL は、SET OPTION ステートメント上のロック・レベルです。 STRCMTCTL コマンドを使用して、異なる CMTSCOPE、NFYOBJ、または LCKLVL を指定できます。 CMTSCOPE(*JOB) を指定してジョブ・レベル・コミットメント定義を開始すると、DB2 UDB for iSeries は、その活動化グループの中のプログラムのジョブ・レベル・コミットメント定義を使用します。

注:

1. コミットメント制御を用いる場合には、アプリケーション・プログラムの中でデータ操作言語ステートメントによって参照される表は、ジャーナル処理されていなければなりません。
2. 指定した LCKLVL は単に省略時のロック・レベルに過ぎないことに注意してください。 コミットメント制御を開始すると、SET TRANSACTION SQL ステートメントと CRTSQLxxx、STRSQL、または RUNSQLSTM の各コマンドの COMMIT パラメーターで指定したロック・レベルとが省略時のロック・レベルを一時変更します。

列関数、GROUP BY、または HAVING を使用するカーソルで、コミットメント制御下で実行しているものについては、ROLLBACK HOLD がカーソルの位置に影響することはありません。さらに、コミットメント制御下では次のことが起きます。

- COMMIT(*CHG) および (ALWBLK(*NO) または ALWBLK(*READ)) をこれらのカーソルの 1 つに対して指定すると、COMMIT(*CHG) が要求されたが許可されなかったことを示すメッセージ (CPI430B) が送られます。
- KEEP LOCKS 文節で COMMIT(*ALL)、COMMIT(*RR)、または COMMIT(*CS) をカーソルの 1 つに対して指定すると、DB2 UDB for iSeries は、参照されるすべての表を共用モード (*SHRNUP) でロックします。このロックは、該当の表での、並行アプリケーション・プロセスの実行 (読み取り専用操作以外) を防止します。 KEEP LOCKS 文節がカーソルの 1 つに指定されている COMMIT(*ALL)、COMMIT(*RR)、または COMMIT(*CS) が要求されたが許可されなかったことを示すメッセージ (SQL7902 または CPI430A) が送られます。メッセージ SQL0595 も送られます。

KEEP LOCKS 文節と一緒に COMMIT(*ALL)、COMMIT(*RR)、または COMMIT(*CS) が指定されており、カタログ・ファイルが使用されているか、または一時結果表が必要なカーソルについては、DB2 UDB for iSeries は、共有モード (*SHRNUP) のすべての参照表をロックします。これによって、表での並行プロセスの実行 (読み取り専用操作以外) を防止します。 COMMIT(*ALL) が要求されたが許可されなかったことを示すメッセージ (SQL7902 または CPI430A) が送られます。メッセージ SQL0595 も送られます。

プログラムの事前コンパイル時に ALWBLK(*ALLREAD) と COMMIT(*CHG) の指定があった場合は、すべての読み取り専用カーソルは行のブロック化を可能にするので、ROLLBACK HOLD はカーソル位置をロールバックしません。

COMMIT(*RR) が要求されると、表は、照会がクローズされるまでロックされます。カーソルが読み取り専用のときは、表は (*SHRNUP で) ロックされます。カーソルが更新モードにある時は、表は (*EXCLRD で) ロックされます。他のユーザーは表から締め出されるため、反復可能読み取りによる実行は、表への並行アクセスを阻止します。

COMMIT(*NONE) 以外の分離レベルが指定され、アプリケーションが ROLLBACK を出すか、または活動化グループが異常終了した場合 (しかも、コミットメント定義が *JOB でない場合) は、作業単位内で行ったすべての更新、挿入、削除、および DDL 操作はバックアウトされます。 アプリケーションが COMMIT を出すか、または活動化グループが通常どおり終了した場合は、作業単位内で行ったすべての更新、挿入、削除、および DDL 操作はコミットされます。

DB2 UDB for iSeries は、行についてのロックを使用して、作業単位が完了する前に、変更したデータに他のジョブがアクセスしないようにします。 COMMIT(*ALL) の指定があるときは、作業単位が完了する前に読み取られたデータを、他のジョブが変更するのを防止するためにも、取り出された行についての読み取りロックが使用されます。しかし、これによって、他のジョブによる未変更の行の読み取りが妨げられることはありません。 これにより、同じ作業単位がある行を再び読み取った場合、同じ結果が得られます。 読み取りロックは、他のジョブが同じ行を取り出すのを妨げることはありません。

コミットメント制御では、1 つの作業単位内で最大 4,000,000 の異なる行変更を処理することができます。 COMMIT(*ALL) または COMMIT(*RR) の指定がある場合は、読み取られる総数行もこの制限に含められます。(1 つの作業単位内で同じ行が複数回読み取られたり変更されたりしても、それは、この制限に対しては 1 回として数えられます。) 多数のロックを保持すると、システム・パフォーマンスが低下するだけでなく、ある作業単位内でロックされている行に対しては、その作業単位が終了するまでは同時に使用しているユーザーがアクセスできなくなります。したがって、1 つの作業単位で処理される行の数をできるだけ少なくすることが、効率化を図るための最も有効な方法です。

コミットメント制御を使用すると、1 つの作業単位内で、最大 512 個のファイルをコミットメント制御下でオープンし、また保留状態の変更をクローズすることができます。

COMMIT HOLD および ROLLBACK HOLD を使用すると、カーソルはオープンされたままになるので、OPEN を出し直さなくても、別の作業単位を開始することができます。HOLD の値は、iSeries システム上にない遠隔データベースに接続されている場合は、利用不能です。ただし、DECLARE CURSOR 上の WITH HOLD オプションを使用すると、COMMIT の後でもカーソルをオープンしたままにしておくことができます。 このタイプのカーソルは、iSeries システム上にない遠隔データベースに接続している場合にサポートされます。 こうしたカーソルはロールバックでクローズされます。

表 37. 行のロック期間

SQL ステートメント	COMMIT パラメーター (注 6 を参照)	行ロックの期間	ロック・タイプ
SELECT INTO SET 変数 VALUES INTO	*NONE *CHG *CS (注 8 を参照) *ALL (注 2 を参照)	ロックなし。 ロックなし。 読み取りおよび解放時に行がロックされる。 読み取りから次の ROLLBACK または COMMIT まで。	 READ READ
FETCH (読み取り専用カーソル)	*NONE *CHG *CS (注 8 を参照) *ALL (注 2 を参照)	ロックなし。 ロックなし。 読み取りから次の FETCH まで。 読み取りから次の ROLLBACK または COMMIT まで。	 READ READ

表 37. 行のロック期間 (続き)

SQL ステートメント	COMMIT パラメーター (注 6 を参照)	行ロックの期間	ロック・タイプ
FETCH (更新または削除が可能なカーソル) (注 1 を参照)	*NONE	行が更新または削除されない場合は、読み取りから次の FETCH まで。 行が更新または削除される場合は、読み取りから次の UPDATE または DELETE まで。	UPDATE
	*CHG	行が更新または削除されない場合は、読み取りから次の FETCH まで。 行が更新または削除される場合は、読み取りから次の COMMIT または ROLLBACK まで。	UPDATE
	*CS	行が更新または削除されない場合は、読み取りから次の FETCH まで。 行が更新または削除される場合は、読み取りから次の COMMIT または ROLLBACK まで。	UPDATE
	*ALL	読み取りから次の ROLLBACK または COMMIT まで。	UPDATE[3]
INSERT (目標表)	*NONE	ロックなし。	
	*CHG	挿入から次の ROLLBACK または COMMIT まで。	UPDATE
	*CS	挿入から次の ROLLBACK または COMMIT まで。	UPDATE
	*ALL	挿入から次の ROLLBACK または COMMIT まで。	UPDATE[4]
INSERT (部分選択の表)	*NONE	ロックなし。	
	*CHG	ロックなし。	
	*CS	各行は読み取りの期間ロックされる。	READ
	*ALL	読み取りから次の ROLLBACK または COMMIT まで。	READ
UPDATE (カーソルなし)	*NONE	各行は更新の期間ロックされる。	UPDATE
	*CHG	読み取りから次の ROLLBACK または COMMIT まで。	UPDATE
	*CS	読み取りから次の ROLLBACK または COMMIT まで。	UPDATE
	*ALL	読み取りから次の ROLLBACK または COMMIT まで。	UPDATE
DELETE (カーソルなし)	*NONE	各行は削除の期間ロックされる。	UPDATE
	*CHG	読み取りから次の ROLLBACK または COMMIT まで。	UPDATE
	*CS	読み取りから次の ROLLBACK または COMMIT まで。	UPDATE
	*ALL	読み取りから次の ROLLBACK または COMMIT まで。	UPDATE
UPDATE (カーソルつき)	*NONE	ロックは行の更新時に解放される。	UPDATE
	*CHG	読み取りから次の ROLLBACK または COMMIT まで。	UPDATE
	*CS	読み取りから次の ROLLBACK または COMMIT まで。	UPDATE
	*ALL	読み取りから次の ROLLBACK または COMMIT まで。	UPDATE

表 37. 行のロック期間 (続き)

SQL ステートメント	COMMIT パラメーター (注 6 を参照)	行ロックの期間	ロック・タイプ
DELETE (カーソルつき)	*NONE	ロックは行の削除時に解放される。	UPDATE
	*CHG	読み取りから次の ROLLBACK または COMMIT まで。	UPDATE
	*CS	読み取りから次の ROLLBACK または COMMIT まで。	UPDATE
	*ALL	読み取りから次の ROLLBACK または COMMIT まで。	UPDATE
副照会 (更新または削除可能カーソルまたは UPDATE または DELETE カーソルなし)	*NONE	読み取りから次の FETCH まで。	READ
	*CHG	読み取りから次の FETCH まで。	READ
	*CS	読み取りから次の FETCH まで。	READ
	*ALL (注 2 を参照)	読み取りから次の ROLLBACK または COMMIT まで。	READ
副照会 (読み取り専用カーソルまたは SELECT INTO)	*NONE	ロックなし。	
	*CHG	ロックなし。	
	*CS	各行は読み取りの期間ロックされる。	READ
	*ALL	読み取りから次の ROLLBACK または COMMIT まで。	READ

注:

1. 結果表が読み取り専用ではなく (SQL 解説書のDECLARE CURSORの説明を参照)、かつ次のうちの 1 つに該当する場合には、カーソルは UPDATE または DELETE の機能によりオープンされます。
   - カーソルが FOR UPDATE 文節により定義されている。
   - カーソルが FOR UPDATE 文節、FOR READ ONLY 文節、または ORDER BY 文節なしで定義されており、プログラムに次のうち少なくとも 1 つが入っている。
     - 同じカーソル名を参照するカーソル UPDATE
     - 同じカーソル名を参照するカーソル DELETE
     - CRTSQLxxx コマンドで指定された EXECUTE または EXECUTE IMMEDIATE ステートメントおよび ALWBLK(*READ) または ALWBLK(*NONE)
2. COMMIT(*ALL) を満足するために表または視点を排他的にロックすることができます。 UNION を含む部分選択が処理される場合、または照会の処理のために一時的な結果の使用が必要である場合は、コミットされていない変更がユーザーに表示されることがないように排他的ロックが取得されます。
3. 行が更新または削除されない場合は、ロックは *READ に緩和されます。
4. 目標表の行に対する UPDATE ロックおよび部分選択表の行に対する READ ロック。
5. 反復可能読み取りを満足するために表または視点を排他的にロックすることができます。その場合でも、行のロックは反復可能読み取りの下で行われます。取得されるロックとその期間は *ALL と同じです。
6. 反復可能読み取り (*RR) 行のロックは、*ALL のために表示されるロックと同じになります。
7. 分離レベルとロックの説明の詳細については、SQL 解説書の中の分離レベルを参照してください。
8. KEEP LOCKS 文節が *CS で指定される場合、読み取りロックがあればカーソルがクローズされるまで、またはCOMMIT か ROLLBACK が実行されるまで保持されます。分離文節と関連付けられるカーソルがない場合、ロックは SQL ステートメントが完了するまで保持されます。

保管ポイント

保管ポイントにより、ユーザーがトランザクションの中にマイルストーンを作成することができます。トランザクションがロールバックする場合、変更は、トランザ

クションの先頭ではなく指定された保管ポイントまで元に戻されます。保管ポイントは、SAVEPOINT SQL ステートメントを使用してセットされます。たとえば、STOP_HERE という名前の保管ポイントは次のように作成します。

```
SAVEPOINT STOP_HERE
ON ROLLBACK RETAIN CURSORS
```

アプリケーションのプログラム・ロジックは、アプリケーションが進行するにつれ、保管ポイント名が再利用されるか、それとも保管ポイント名がアプリケーション内の固有のマイルストーンを表し、再利用は許されないかを示します。

保管ポイントが、別の SAVEPOINT ステートメントを使って移動してはならない固有のマイルストーンを表している場合は、UNIQUE キーワードを指定します。これにより、SAVEPOINT ステートメントの保管ポイント名と同一の名前を使用するストアード・プロシージャーが呼び出されてその名前が不用意に再利用されてしまうことを防ぎます。ただし、SAVEPOINT ステートメントがループの中で使用される場合は、UNIQUE キーワードを使用することはできません。次の SQL ステートメントは、START_OVER という名前の固有の保管ポイントをセットします。

```
SAVEPOINT START_OVER UNIQUE
       ON ROLLBACK RETAIN CURSORS
```

保管ポイントまでロールバックするには、TO SAVEPOINT 文節を指定して ROLLBACK ステートメントを使用します。以下の例は、SAVEPOINT ステートメントおよび ROLLBACK TO SAVEPOINT ステートメントの使用について説明しています。

このアプリケーション・ロジックは、希望する日付の航空券の予約をしてから、ホテルの予約をします。ホテルが予約できない場合は、航空券の予約をロールバックしてから、別の日付の処理を繰り返します。最大 3 つの日付まで、試みることができます。

```
got_reservations =0;
EXEC SQL SAVEPOINT START_OVER UNIQUE ON ROLLBACK RETAIN CURSORS;

if (SQLCODE != 0) return;

for (i=0; i<3 & got_reservations == 0; ++i)
{
  Book_Air(dates(i), ok);
  if (ok)
  {
    Book_Hotel(dates(i), ok);
    if (ok) got_reservations = 1;
    else
    {
      EXEC SQL ROLLBACK TO SAVEPOINT START_OVER;
      if (SQLCODE != 0) return;
    }
  }
}

EXEC SQL RELEASE SAVEPOINT START_OVER;
```

保管ポイントは、RELEASE SAVEPOINT ステートメントを使用して解放されます。 RELEASE SAVEPOINT ステートメントを使用して明示的に保管ポイントを解

放しない場合は、現行の保管ポイント・レベルの終わりかまたはトランザクションの終わりに解放されます。次のステートメントは、保管ポイント START_OVER を解放します。

```
RELEASE SAVEPOINT START_OVER
```

保管ポイントは、トランザクションがコミットまたはロールバックされた時点で解放されます。いったん保管ポイント名が解放されたら、その保管ポイント名へのロールバックはもう行えません。 COMMIT ステートメントまたは ROLLBACK ステートメントは、トランザクション内に設定されているすべての保管ポイント名を解放します。トランザクション内のすべての保管ポイント名が解放されるため、コミットまたはロールバックの後はすべての保管ポイント名を再利用することができます。

保管ポイントのレベル

1 つのステートメントが、明示的あるいは暗黙的に、ユーザー定義関数、トリガー、またはストアード・プロシージャーを呼び出すことができます。これはネスティングと呼ばれます。一部のケースでは、新たなネスティング・レベルが開始される時点で、新たな保管ポイント・レベルも開始されます。新たな保管ポイント・レベルは、呼び出すアプリケーションを、それより低いレベルのルーチンまたはトリガーによる保管ポイント活動から分離します。

保管ポイントは、それらが定義されているのと同一の保管ポイント・レベル (または有効範囲) の中でのみ参照できます。 ROLLBACK TO SAVEPOINT ステートメントを使用して、現行の保管点レベルの外側に設定された保管点までロールバックすることはできません。同様に、RELEASE SAVEPOINT ステートメントを使用して、現行の保管点レベルの外側に設定された保管点を解放することはできません。次の表は、保管ポイント・レベルが開始および終了される時点を要約したものです。

新たな保管ポイントが開始される時点	保管ポイントが終了する時点
新たな作業単位が開始された時	COMMIT または ROLLBACK が出された時
トリガーが起動された時	トリガーが完了した時
ユーザー定義関数が起動された時	ユーザー定義関数が起動元に戻された時
NEW SAVEPOINT LEVEL 文節が指定されて作成されたストアード・プロシージャーが呼び出された時	ストアード・プロシージャーが呼び出し元に戻った時
ATOMIC 複合 SQL ステートメントに BEGIN がある時	ATOMIC 複合ステートメントに END がある時

ある保管ポイント・レベルで設定された保管ポイントは、その保管ポイント・レベルが終了する時点で暗黙的に解放されます。

分散データベース環境における保管ポイントの考慮事項

保管ポイントは単一接続のみを対象とします。保管ポイントは、設定されても、アプリケーションの接続対象となるすべての遠隔データベースに分散されることはありません。保管ポイントは、その保管ポイントが設定された時点でアプリケーションが接続されている現行データベースに対してのみ、適用されます。

アトミック・オペレーション

COMMIT(*CHG)、COMMIT(*CS)、または COMMIT(*ALL) の下で実行している場合、すべての操作がアトミシティを備えることが保証されます。すなわち、これらの操作は必ず完了するか、または開始前の状態に戻ります。これは、機能が (電源障害、異常なジョブ終了、またはジョブ取り消しなどにより) いつどのように終了または中断しても常に同じです。

ただし、COMMIT (*NONE) を指定すると、基礎をなす一部のデータベース・データ定義機能はアトミシティを備えなくなります。次の SQL データ定義ステートメントは、アトミシティを備えることが保証されます。

ALTER TABLE (注 1 を参照)

COMMENT ON (注 2 を参照)

LABEL ON (注 2 を参照)

GRANT (注 3 を参照)

REVOKE (注 3 を参照)

DROP TABLE (注 4 を参照)

DROP VIEW (注 4 を参照)

DROP INDEX

DROP PACKAGE

注:

1. 列定義変更の他に制約の追加または除去が必要な場合は、各操作は一度に 1 つずつ処理され、SQL ステートメント全体はアトミシティをもたなくなります。操作の順序は以下のようになります。
   - 制約の除去。
   - RESTRICT オプションが指定されている列の除去。
   - 他のすべての列定義の変更 (DROP COLUMN CASCADE、ALTER COLUMN、ADD COLUMN)。
   - 制約の追加。
2. COMMENT ON ステートメントまたは LABEL ON ステートメントに複数の列を指定した場合には、それらの列は一度に 1 つずつ処理されます。これにより、その SQL ステートメント全体はアトミシティをもたなくなりますが、それぞれの列またはオブジェクトに対する COMMENT ON または LABEL ON はアトミシティを持ちます。
3. GRANT ステートメントまたは REVOKE ステートメントの対象として複数の表、SQL パッケージ、またはユーザーを指定した場合には、それらの表は一度に 1 つずつ処理されます。すなわち、その SQL ステートメント全体はアトミシティを備えたものではありませんが、それぞれの表に対する GRANT または REVOKE はアトミシティをもちます。
4. DROP TABLE または DROP VIEW の実行時に従属視点の除去が必要な場合は、個々の従属表が一度に 1 つずつ処理されるので、SQL ステートメント全体はアトミシティをもつものとはなりません。

以下のデータ定義ステートメントは、複数の DB2 UDB for iSeries データベース操作を必要とするので、アトミシティを備えていません。

CREATE ALIAS
CREATE DISTINCT TYPE
CREATE FUNCTION
CREATE INDEX
CREATE PROCEDURE
CREATE SCHEMA
CREATE TABLE
CREATE TRIGGER
CREATE VIEW
DROP ALIAS
DROP DISTINCT TYPE
DROP FUNCTION
DROP PROCEDURE
DROP SCHEMA
DROP TRIGGER
RENAME (注 1 を参照)

注:

1. RENAME は、名前またはシステム名が変更されている場合に限り、アトミシティをもちます。両方が変更されている場合、RENAME はアトミシティをもちません。

たとえば、CREATE TABLE は、DB2 UDB for iSeries 物理ファイルが作成されてからメンバーが追加されるまでの間に中断されることがあります。したがって、作成ステートメントの場合には、操作が異常終了すると、オブジェクトを削除し作成し直さなければならないことがあります。また、DROP COLLECTION ステートメントの場合には、コレクションを作成し直すか、または CL コマンドのライブラリー削除 (DLTLIB) を使用してコレクションの残り部分を除去しなければならないことがあります。

制約

DB2 UDB for iSeries は、固有制約、参照制約、および検査制約をサポートします。固有制約は、キーの値が固有であることを保証する 1 つの規則です。参照制約は、従属表内の外部キーのすべての非ヌル値は、親表内に対応する親キーを持っているという 1 つの規則です。検査制約は、列または列のグループで許可される値を制限する規則です。

DB2 UDB for iSeries は、すべての DML (データ操作言語) ステートメントの実行時に制約の妥当性検査を行います。ただし、特定の操作 (従属表の復元など) では、制約の妥当性が分からなくなることがあります。この場合、DB2 UDB for iSeries が制約の妥当性を確かめるまで、DML ステートメントを実行できないようにします。

- 固有制約は、索引と一緒に組み込まれます。固有制約を組み込む索引が無効である場合は、アクセス・パスの再作成編集 (EDTRBDAP) コマンドを使用して、現在再作成を必要とする任意の索引を表示することができます。
- DB2 UDB for iSeries が、参照制約または検査制約が有効かどうか現時点で分からない場合は、この制約は、検査保留状態にあると見なされます。検査保留の制約編集 (EDTCPCST) コマンドを使用すると、現在再作成を必要とする索引を表示することができます。

制約について詳しくは、 145 ページの『第 10 章 データ保全性』 およびデータベース・プログラミングを参照してください。

保管/復元

OS/400 保管/復元機能は、表、視点、索引、ジャーナル、ジャーナル・レシーバー、SQL パッケージ、SQL プロシージャー、SQL トリガー、ユーザー定義関数、ユーザー定義タイプ、およびコレクションを、ディスク (保管ファイル) に、またはある種の外部媒体 (テープまたはディスケット) に保管する場合に使用します。 保管したバージョンは、あとでいつでも、任意の iSeries システムに復元することができます。 保管/復元機能を使用すると、コレクション全体、選択したオブジェクト、または特定の日付および時刻以降に変更されたオブジェクトだけを保管することができます。 オブジェクトを元の状態に復元するために必要なすべての情報が保管されます。 また、この機能を使用すると、表またはコレクション全体の前のバージョンでデータを復元することにより、損傷した表を回復することができます。

SQL プロシージャー用に作成されたプログラムまたは SQL 関数またはソース関数用に作成されたサービス・プログラムは復元時に、そのプロシージャーまたは関数が同じシグニチャーでまだ存在していない限り、SYSROUTINES および SYSPARMS カタログに自動的に追加されます。 QSYS で作成される SQL プログラムは、復元時には SQL プロシージャーとしては作成されません。さらに、CREATE PROCEDURE または CREATE FUNCTION ステートメントで参照された外部プログラムまたはサービス・プログラムには、SYSROUTINES にそのルーチンを登録するために必要な情報が含まれていない可能性があります。必要な情報があり、シグニチャーが固有な場合には、その関数またはプロシージャーも、復元時に SYSROUTINES および SYSPARMS に追加されます。

SQL 表が復元されると、表に定義された SQL トリガーの定義も復元されます。SQL トリガーの定義は、SYSTRIGGERS、SYSTRIGDEP、SYSTRIGCOL、および SYSTRIGUPD の各カタログに自動的に追加されます。 SQL CREATE TRIGGER ステートメントによって作成されたプログラム・オブジェクトも、SQL 表が保管され復元されるときに、保管/復元されなければなりません。プログラム・オブジェクトの保管と復元は、データベース・マネージャーによって自動化されていません。自己参照トリガー用の予防処置策は、SQL 表を新規ライブラリーに復元するときに、検討する必要があります。SQL 解説書の CREATE TRIGGER ステートメントのセクションの注記の中の 作動不能トリガーを参照してください。

ユーザー定義タイプに対して *SQLUDT オブジェクトが復元されると、そのユーザー定義タイプは、SYSTYPES カタログに自動的に追加されます。ユーザー定義タイプとソース・タイプとの間にキャストするために必要な適切な関数が、まだ存在していない場合には、それらのタイプおよび関数も作成されます。

分散 SQL プログラムとその関連 SQL パッケージのどちらも、保管して、任意の数のシステムに復元することができます。これにより、いくつかの異なるシステムが置かれている SQL プログラムの任意の数のコピーから同じアプリケーション・サーバーに置かれている同じ SQL プログラムを利用することができます。また、復元された SQL プログラムが置かれている任意の数のアプリケーション・サーバーに単一の分散 SQL プログラムを接続することもできます (CRTSQLPKG も使用できます)。SQL パッケージを別のライブラリーに復元することはできません。

重要: スキーマを既存のライブラリーまたは別の名前を持つスキーマに復元しても、ジャーナル、ジャーナル・レシーバー、または IDDU ディクショナリー (存在していても) は復元されません。 スキーマは別の名前のスキーマに復元すると、そのスキーマ内のカタログ視点は前のスキーマ内のオブジェクトしか反映しません。ただし、QSYS2 のカタログ視点は、すべてのオブジェクトを適切に反映します。

耐損傷性

サーバーには、ディスク・エラーが原因で生じた損傷を軽減またはなくすための方法がいくつかあります。たとえば、ミラーリング、チェックサム、および RAID ディスクはすべて、ディスク問題の可能性を軽減します。 DB2 UDB for iSeries 機能には、ディスク・エラーまたはシステム・エラーが原因で生じた損傷に対する一定の許容幅もあります。

DROP 操作は、損傷の有無にかかわらず常に正常に完了します。したがって、仮に何らかの損傷が生じても、少なくとも表、視点、SQL パッケージ、索引、プロシージャー、関数、または特殊タイプを削除して復元または再作成することができます。

ディスク・エラーによって表内の行のほんの一部だけが損傷を受けた場合には、DB2 UDB for iSeries のデータベース・マネージャーでは、まだアクセス可能な状態にある行をユーザーが読み取ることができます。

索引回復

DB2 UDB for iSeries には、索引の回復を行うための機能がいくつかあります。

- システム管理による索引保護

  EDTRCYAP CL コマンドを使用すると、ユーザーは DB2 UDB for iSeries に対して、システム障害または電源障害が生じた場合にシステム上のすべての索引を回復するために必要な時間を、指定時間内に確実にとどめるように指示することができます。システムはシステム・ジャーナルに十分な情報を自動的にジャーナル処理して、回復時間を指定した時間内に制限します。
- 索引のジャーナル処理

  DB2 UDB for iSeries には、電源障害またはシステム障害が起きても索引全体の再作成を行わなくて済む索引ジャーナル処理機能があります。索引がジャーナル処理されていれば、システム・データベース・サポートが自動的に表内のデータと索引が同期を保つようにするので、索引を最初から作成し直す必要はありません。SQL の索引は、自動的にジャーナル処理されません。ただし、CL コマンドのアクセス・パス・ジャーナル開始 (STRJRNAP) コマンドを使用すれば、DB2 UDB for iSeries が作成した任意の索引をジャーナル処理することはできます。
- 索引の再作成

システムの索引にはすべて、索引の維持管理をいつ行うかを指定するメインテナンス・オプションがあります。 SQL 索引は、*IMMED インデックスの属性を指定して作成されます。

電源障害またはシステム異常障害が生じた場合で、索引が前に説明した技法の 1 つによって保護されていなかった場合は、その時点で変更処理中であった索引は、実際のデータに適合するようにデータベース・マネージャーによって作成しなければならないことがあります。システムのすべての索引には、必要な場合にいつ索引を再作成するかを指定する回復オプションがあります。UNIQUE 属性を持つ SQL 索引はすべて、*IPL 回復属性を指定して作成されます (これは、これらの索引が OS/400 の始動前に再作成されることを意味します)。その他の SQL 索引は、*AFTIPL 回復オプションを指定して作成されます (これは、これらの索引がオペレーティング・システムの始動後に非同期に再作成されることを意味します)。 IPL 時に、操作員は再作成する必要がある索引とそれぞれの回復オプションを示す画面を見ることができます。この画面から、操作員は回復オプションを一時変更することができます。

- 索引の保管と復元

  保管/復元機能を使用すると、オブジェクト保管 (SAVOBJ) またはライブラリー保管 (SAVLIB) の各 CL コマンドで ACCPTH(*YES) を使用して表を保管するとき、索引を保管することができます。索引も同時に保管されていれば、復元時に索引を再作成する必要はありません。保管されていない索引を復元すると、索引は自動的にかつ非同期にデータベース・マネージャーによって再作成されます。

カタログの保全性

カタログには、スキーマ内の表、視点、SQL パッケージ、索引、プロシージャー、関数、トリガー、およびパラメーターについての情報が入っています。データベース・マネージャーにより、カタログ内の情報が常に正確であることが保証されます。これは、エンド・ユーザーがカタログ内の情報を明示的に変更できないようにすること、および、カタログの中に記述されている表、視点、SQL パッケージ、索引、タイプ、プロシージャー、関数、トリガー、およびパラメーターに対して変更が行われた場合にカタログ内の情報を暗黙に維持管理することにより行われます。

カタログの保全性は、SQL ステートメント、OS/400 CL コマンド、システム/38 環境の CL コマンド、システム/36 環境の機能、または iSeries システムの他のプロダクトまたはユーティリティーによってスキーマ内のオブジェクトが変更された場合でも維持管理されます。 たとえば、表を削除することは、SQL DROP ステートメントを実行すること、OS/400 DLTF CL コマンドを実行すること、システム/38 DLTF CL コマンドを実行すること、または WRKF または WRKOBJ 画面からオプション 4 を入力することにより行うことができます。 表の削除に使用するインターフェースに関係なく、データベース・マネージャーは、削除が行われたときにカタログから表の記述を取り除きます。 次の一覧表は、各機能とそれぞれがカタログに作用する効果を示しています。

表 38. 各機能がカタログに作用する効果

機能	カタログに作用する効果
表への制約の追加	カタログへの情報の追加
表からの制約の除去	カタログからの関連情報の除去
スキーマへのオブジェクトの作成	カタログへの情報の追加

表 38. 各機能がカタログに作用する効果 (続き)

機能	カタログに作用する効果
スキーマからのオブジェクトの削除	カタログからの関連情報の除去
スキーマへのオブジェクトの復元	カタログへの情報の追加
オブジェクトの長注釈の変更	カタログ内での注釈の更新
オブジェクト・ラベル (テキスト) の変更	カタログ内でのラベルの更新
オブジェクト所有者の変更	カタログ内での所有者の更新
スキーマからのオブジェクトの移動	カタログからの関連情報の除去
スキーマへのオブジェクトの移動	カタログへの情報の追加
オブジェクトの名前の付け直し	カタログ内でのオブジェクトの名前の更新

ユーザー補助記憶域プール (ASP)

スキーマは、CREATE COLLECTION ステートメントおよび CREATE SCHEMA ステートメントで ASP 文節を使用することによって ASP に作成することができます。 CRTLIB コマンドを使用しても、ライブラリーをユーザー ASP に作成することができます。 作成したライブラリーは、SQL 表、視点、および索引を入れるために使用できます。 補助記憶域プールについての詳細は、バックアップおよび回復の手引き を参照してください。

独立補助記憶域プール (IASP)

独立ディスク・プールは、iSeries サーバー上でユーザー・データベースをセットアップするために使用されます。 独立ディスク・プールには、1 次、2 次、ユーザー定義ファイル・システム (UDFS) の 3 つのタイプがあります。データベースは、1 次独立ディスク・プールを使用するようセットアップされます。

iSeries サーバーで複数のデータベースを処理することができます。 iSeries サーバーは、システム・データベース (しばしば SYSBAS と呼ばれる) と、1 つまたは複数のユーザー・データベースを処理できる能力を提供します。ユーザー・データベースは独立ディスク・プールの使用により iSeries サーバーに組み込まれ、iSeries ナビゲーターのディスク管理機能でセットアップされます。独立ディスク・プールがセットアップされると、iSeries ナビゲーターの機能の下に別のデータベースとして表示されます。

第 20 章 アプリケーション・プログラム内の SQL ステートメントのテスト

この章では、アプリケーション・プログラム内の SQL ステートメントのテスト環境を確立する方法、およびこのプログラムをデバッグする方法について説明します。

詳細については、以下のセクションを参照してください。

- 『テスト環境の確立』
- 354 ページの『SQL アプリケーション・プログラムのテスト』

テスト環境の確立

プログラムのテストに必要な事項は次のとおりです。

- **権限**。ユーザーは、表および視点を作成する権限、SQL データをアクセスする権限、およびプログラムを作成し実行する権限を持っていなければなりません。
- **テスト・データ構造**。ユーザーのプログラムが表または視点のデータを更新、挿入、または削除する場合には、プログラムの実行検査には必ずテスト・データを使用してください 。ユーザーのプログラムが表または視点のデータを検索するだけの場合には、プログラムのテストにプロダクション・レベルのデータを使用しても差し支えはありません。ただし、プロダクション・レベルのデータが間違って変更されることのないようにするために、CL コマンドのデバッグ開始 (STRDBG) に UPDPROD(*NO) を指定して使用することをお勧めします。デバッグの詳細については、CL プログラミング のテストに関する章を参照してください。
- **テスト入力データ**。テスト時にユーザーのプログラムで使用する入力データは、予測できる限りの入力条件を取り入れた有効なデータでなければなりません。有効な入力データを使用しない限り、出力データが有効であるかどうかを確証することはできません。

  ユーザーのプログラムが入力データの有効性を検査する場合には、有効なデータと無効なデータの両方を用意して、有効なデータが処理され、無効なデータが除外されることを確かめてください。

  後続のテストのためにデータのリフレッシュが必要になることがあります。

プログラムを完全にテストするためには、プログラム全体にわたりできるだけ多くの経路をテストするようにしてください。たとえば、次の通りです。

- プログラムが各ブランチを実行しなければならないような入力データを使用する。
- 結果を検査する。たとえば、プログラムがある行を更新した場合には、その行を選択してそれが正しく更新されているかどうかを確認してください。

- 必ずプログラム・エラー・ルーチンをテストする。この場合も、予測可能なできるだけ多くのエラー状態にプログラムを置くような入力データを使用してください。
- ユーザーのプログラムで使用する編集ルーチンおよび妥当性検査ルーチンをテストする。可能な限り多くの組み合わせの入力データをプログラムに与えて、データの編集または妥当性検査が正しく行われることを確認してください。

詳細については、『テスト・データ構造の設計』を参照してください。

テスト・データ構造の設計

SQL データをアクセスするアプリケーションをテストするためには、テスト用の表および視点を作成しなければならないことがあります。

- **既存の表のテスト視点**。アプリケーションではデータを変更せず、しかもデータが 1 つまたは複数のプロダクション・レベルの表に入っている場合には、既存の表の視点を使用すると効果的です。また、プロダクション・レベルのデータが間違って変更されてしまうのを防ぐために、UPDPROD(*NO) を指定した STRDBG コマンドを使用することをお勧めします。デバッグの詳細については、CL プログラミング のテストに関する章を参照してください。

- **テスト表**。ユーザーのアプリケーションがデータを作成、変更、または削除する場合には、テスト用のデータの入った表を使用してアプリケーションをテストする方が安全です。表と視点の作成方法の説明については、『第 2 章 SQL 入門』を参照してください。

  また、CL コマンドの重複オブジェクト作成 (CRTDUPOBJ) を使用して、既存の表、視点、または索引の複製をテスト用に作成する方法もあります。

権限

表を作成するには、表を作成し、その表を入れるスキーマを使用する権限が許可されていなければなりません。また、テストしたいプログラムを作成し実行する権限も必要です。

既存の表または視点を (直接に、または視点の基礎として) 使用することを予定している場合には、使用する表および視点をアクセスする権限が必要です。

視点を作成したい場合には、視点を作成する権限、およびその視点の基礎として使用する表の権限をもっていなければなりません。特定の SQL ステートメントに必要な特定の権限の詳細については、SQL 解説書を参照してください。

SQL アプリケーション・プログラムのテスト

DB2 UDB for iSeries SQL アプリケーションは、プログラム・デバッグ・フェーズとパフォーマンス検査フェーズの 2 つのフェーズでテストされます。以下に、355 ページの『プログラム・デバッグ・フェーズ』と 355 ページの『パフォーマンス検査フェーズ』について説明します。

プログラム・デバッグ・フェーズ

このテスト・フェーズは、SQL 照会が正しく指定されており、またプログラムが正しい結果を作り出すことを確認するために実行されます。

SQL ステートメントを含むプログラムのデバッグは、SQL ステートメントを含まないプログラムのデバッグの場合とほとんど同じです。ただし、SQL ステートメントをデバッグ・モードのジョブの中で実行させるときは、データベース・マネージャーは、各 SQL ステートメントがどのように実行されたかのメッセージをジョブ・ログに入れます。このメッセージは各 SQL ステートメントに対する SQLCODE を示します。ステートメントが正しく実行されると、SQLCODE の値はゼロであり、完了メッセージが出されます。SQLCODE が負のときは、診断メッセージが出されます。SQLCODE が正のときは通知メッセージが出されます。

メッセージは、4 桁の数字のコードの前に **SQL** が付いているか、5 桁の数字コードの前に **SQ** が付いています。たとえば、SQLCODE が -204 のときは、メッセージは SQL0204 であり、SQLCODE が 30000 のときは、メッセージは SQ30000 です。

SQLCODE に関連するものとして、SQLSTATE があります。 SQLSTATE は追加の戻りコードであり、SQLCA の中に戻され、いくつかの異なる IBM リレーショナル・データベース・プロダクトに共通のエラー条件を示しています。リレーショナル・データベース・プロダクトが異なっていても、エラー条件が同じならば、同じ SQLSTATE が返されます。エラー条件が同じであっても、同じ SQLCODE は返されません。この戻りコードは、DB2 UDB for iSeries 以外のシステムで実行されるリレーショナル・データベース操作から戻されるエラー原因を判別するときに特に役立ちます。

ILE 以外のプログラム・デバッグの場合、デバッグ・モードの高水準言語ステートメント番号に対する参照は、コンパイル・リストからのものを使用しなければなりません。 ILE プログラムのデバッグの場合、DBGVIEW(*SOURCE) を指定してプログラムを事前コンパイルしてからソース・レベル・デバッガーを使用してください。

SQLCODE が負のときと、+100 以外の正のコードのときは、デバッグ・モードかどうかに関係なく、SQL は常にメッセージをジョブ・ログに入れます。

パフォーマンス検査フェーズ

このテスト・フェーズは、該当する索引が用意されており、またデータベース・マネージャーが予定の応答時間内に照会を解決できる方法で照会がコーティングされていることを確認するものです。 SQL アプリケーションのパフォーマンスは、アクセスされる表の属性によって異なります。使用する表が小さければ、照会の応答時間は索引が使用可能かどうかによって左右されません。しかし、大きな表があり、該当する索引が存在しないデータベースで同じ照会を実行したときは、照会の応答時間は非常に長くなる可能性があります。

テスト環境はできる限り実動環境に近いものにしてください。テスト用のスキーマには、本番用スキーマと同じ名前と同じ構成を持つ表を用意してください。両方のスキーマ内の表で同じ索引が使用可能になっていることが必要です。行の数および値の分布も両方の表で類似していなければなりません。

パフォーマンスを検査するツールとコマンドの説明は、データベース・パフォーマンスおよび Query 最適化を参照してください。

第 21 章 分散リレーショナル・データベース機能

分散リレーショナル・データベース は、相互に接続された計算機システムに分散して配置された SQL オブジェクト群から構成されています。これらのリレーショナル・データベースは、タイプが同じである場合 (たとえば、DB2 UDB for iSeries) と、タイプが異なる場合 (DB2 ユニバーサル・データベース (OS/390 版)、DB2 (VSE 版および VM 版)、DB2 ユニバーサル・データベース (UDB)、または、DRDA をサポートする IBM 以外のデータベース管理システム) とがあります。各リレーショナル・データベースは、その環境内の表を管理するリレーショナル・データベース・マネージャーを備えています。データベース・マネージャーは相互に通信し、協力し合って、あるデータベース・マネージャーのアクセスによって別のシステムに置かれたリレーショナル・データベース上の SQL ステートメントを実行できるようにします。

アプリケーション・リクエスターは、接続のアプリケーション側をサポートします。アプリケーション・サーバーは、アプリケーション・リクエスターの接続先であるローカル・データベースまたは遠隔データベースです。 DB2 UDB for iSeries は分散リレーショナル・データベース・アーキテクチャー (DRDA) をサポートして、アプリケーション・リクエスターがアプリケーション・サーバーと通信できるようにします。さらに、DB2 UDB for iSeries は出口プログラムを組み込むことにより、DRDA をサポートしない他のデータベース管理システム上のデータへのアクセスを可能にします。これらの出口プログラムは、アプリケーション・リクエスター・ドライバー (ARD) プログラムと呼ばれます。

DB2 UDB for iSeries は 2 つのレベルの分散リレーショナル・データベースをサポートします。

- 遠隔作業単位 (RUW)

  遠隔作業単位は、SQL ステートメントの準備と実行を、1 つの作業単位内で 1 つのアプリケーション・サーバーでしか行わないことを指しています。 DB2 UDB for iSeries は、APPC または TCP/IP のいずれかを介した RUW をサポートします。
- 分散作業単位 (DUW)

  分散作業単位は、SQL ステートメントの準備と実行が 1 つの作業単位内で複数のアプリケーション・サーバーで行えることを指します。ただし、単一 SQL ステートメントは、単一アプリケーション・サーバーにあるオブジェクトしか参照することができません。 DB2 UDB for iSeries は DUW over APPC をサポートし、V5R1 からは DUW over TCP/IP のサポートを導入しました。

分散リレーショナル・データベースに関する包括的な情報については、分散データベース・プログラミングを参照してください。

詳細については、以下を参照してください。

- 358 ページの『DB2 UDB for iSeries 分散リレーショナル・データベース・サポート』

DB2 UDB for iSeries 分散リレーショナル・データベース・サポート

DB2 UDB Query Manager and SQL Development Kit ライセンス・プログラムは、次の SQL ステートメントで分散データベースへの対話式アクセスをサポートします。

- CONNECT
- SET CONNECTION
- DISCONNECT
- RELEASE
- DROP PACKAGE
- GRANT PACKAGE
- REVOKE PACKAGE

これらのステートメントの詳しい説明については、SQL 解説書を参照してください。

追加のサポートは、SQL 事前コンパイラー・コマンドのパラメーターを通して開発キットから提供されます。

SQL ILE C オブジェクト作成 (CRTSQLCI) コマンド

SQL ILE C++ オブジェクト作成 (CRTSQLCPPI) コマンド

SQL COBOL プログラム作成 (CRTSQLCBL) コマンド

SQL ILE COBOL オブジェクト作成 (CRTSQLCBLI) コマンド

SQL PL/I プログラム作成 (CRTSQLPLI) コマンド

SQL RPG プログラム作成 (CRTSQLRPG) コマンド

SQL ILE RPG オブジェクト作成 (CRTSQLRPGI) コマンド

SQL 事前コンパイラー・コマンドの詳細については、*ホスト言語での SQL プログラミング* の中のSQL ステートメントでのプログラムの準備および実行を参照してください。 SQL パッケージ作成 (CRTSQLPKG) コマンドを使用すると、ユーザーは分散プログラムとして作成された SQL プログラムから SQL パッケージを作成することができます。 CRTSQLPKG コマンドと CRTSQLxxx コマンドの構文とパラメーターの定義は、『付録 B. DB2 UDB for iSeries CL コマンドの説明』にあります。

コーディング例に関する詳細は、 x ページの『コードに関する特記事項』を参照してください。

DB2 UDB for iSeries 用分散リレーショナル・データベースのプログラム例

遠隔作業単位リレーショナル・データベースのサンプル・プログラムは SQL プロダクトとともに出荷されています。 QSQL ライブラリー内に含まれているいくつかのファイルとメンバーは分散 DB2 UDB for iSeries のサンプル・プログラムが実行される環境をセットアップする際に役立ちます。

これらのファイルとメンバーを使用する場合は、ファイル QSQL/QSQSAMP 内に置かれている SETUP バッチ・ジョブを実行する必要があります。この SETUP バッチ・ジョブを実行すると、以下のことを実行するようにプログラム例をカスタマイズすることができます。

- ローカルおよび遠隔ロケーションで QSQSAMP ライブラリーを作成する。
- ローカルおよび遠隔ロケーションでリレーショナル・データベースの登録簿項目をセットアップする。
- ローカル・ロケーションでアプリケーション・パネルを作成する。
- プログラムの事前コンパイル、コンパイル、および実行を行って、サンプルの分散アプリケーションのスキーマ、表、索引、および視点を作成する。
- ローカルおよび遠隔ロケーションにある表にデータをロードする。
- プログラムの事前コンパイルとコンパイルを行う。
- アプリケーション・プログラム用の SQL パッケージを遠隔ロケーションで作成する。
- プログラムの事前コンパイル、コンパイル、および実行を行って、部門表のロケーションの列を更新する。

SETUP を実行する前に、QSQL/QSQSAMP ファイルの SETUP メンバーを編集する必要がある場合があります。編集のための指示は、注釈としてメンバーに組み入れられています。SETUP を実行するには、以下のコマンドをシステム・コマンド行に指定してください。

```
========> SBMDBJOB QSQL/QSQSAMP SETUP
```

バッチ・ジョブが完了するまで待機してください。

サンプル・プログラムを使用する場合は、コマンド行で次のコマンドを指定します。

```
========> ADDLIBLE QSQSAMP
```

最初の画面を呼び出して、そこでサンプル・プログラムのカスタマイズをできるようにするには、コマンド行で次のコマンドを指定します。

```
========> CALL QSQ8HC3
```

これにより、次の画面が表示されます。この画面では、ユーザーのデータベース・サンプル・プログラムをカスタマイズすることができます。

```
DB2 for OS/400 ORGANIZATION APPLICATION


ACTION...........:   _      A  (ADD)                  E (ERASE)
D  (DISPLAY)            U (UPDATE)

OBJECT...........:   __     DE (DEPARTMENT)           EM (EMPLOYEE)
DS (DEPT STRUCTURE)

SEARCH CRITERIA..:   __     DI (DEPARTMENT ID)     MN (MANAGER NAME)
DN (DEPARTMENT NAME)   EI (EMPLOYEE ID)
MI (MANAGER ID)        EN (EMPLOYEE NAME)

LOCATION.........:   ________________  (BLANK IMPLIES LOCAL LOCATION)

DATA.............:   _________________________________




                                                          Bottom

F3=Exit
                                     (C) COPYRIGHT IBM CORP. 1982, 1991
```

SQL パッケージ・サポート

OS/400 プログラムは、SQL パッケージと呼ばれるオブジェクトをサポートします。 (OS/400 オブジェクト・タイプは *SQLPKG です。) この SQL パッケージには、分散プログラムを実行しているときにアプリケーション・サーバーで SQL ステートメントを処理するために必要な制御構造とアクセス・プランが入っています。 SQL パッケージは、次の場合に作成することができます。

- RDB パラメーターが CRTSQLxxx コマンドで指定され、プログラム・オブジェクトが正常に作成されている場合。 SQL パッケージは、RDB パラメーターで指定したシステム上に作成されます。

  コンパイルが失敗したり、またはコンパイルがモジュール・オブジェクトしか作成しない場合は、SQL パッケージは作成されません。
- CRTSQLPKG コマンドを使用した場合。 CRTSQLPKG を使用すると、パッケージが事前コンパイル時に作成されなかったとき、またはパッケージが事前コンパイル・コマンドで指定された RDB 以外の RDB で必要になった場合にパッケージを作成することができます。

SQL パッケージ削除 (DLTSQLPKG) コマンドを使用すると、ローカル・システム上の SQL パッケージを削除することができます。

SQL パッケージの作成に関係する権限 ID が保持している特権の中に、遠隔システム (アプリケーション・サーバー) でパッケージを作成するために必要な権限が含まれていなければ、SQL パッケージは作成されません。プログラムを実行するには、権限 ID に、SQL パッケージに対する EXECUTE 特権が含まれていなければなりません。 iSeries システムでは、EXECUTE 特権には、*OBJOPR と *EXECUTE のシステム権限が含まれています。

SQL パッケージ作成 (CRTSQLPKG) コマンドの構文は『付録 B. DB2 UDB for iSeries CL コマンドの説明』に示します。

SQL パッケージの中の有効な SQL ステートメント

別のサーバーに接続するプログラムは、SQL 解説書に説明されているどの SQL ステートメントも使用できます。ただし、SET TRANSACTION ステートメントは除きます。 DB2 UDB for iSeries を使用してコンパイルされ、DB2 UDB for iSeries でないシステムを参照しているプログラムは、その遠隔システムがサポートしている実行可能な SQL ステートメントを使用することができます。事前コンパイラーは、意味不明なステートメントを見つけると、診断メッセージを出し続けます。このようなステートメントは、SQL パッケージの作成時に遠隔システムに送られます。ステートメントが現行アプリケーション・サーバーで実行できないときは、実行時サポートから -84 または -525 の SQLCODE が返されます。たとえば、複数行用 FETCH、ブロック化 INSERT、およびスクロール可能なカーソル・サポートは、アプリケーション・リクエスターおよびアプリケーション・サーバーがともにバージョン 2 リリース 2 以降の OS/400 であるところでの分散プログラム内でしか使用できません。詳細については、SQL 解説書にある、分散リレーショナル・データベースの使用に関する考慮事項を参照してください。

SQL パッケージ作成時の考慮事項

SQL パッケージを作成するとき考慮する必要のある事項が多数あります。以下にそのいくつかを挙げておきます。

CRTSQLPKG 権限

SQL パッケージを iSeries システム上に作成するときは、使用する権限 ID に CRTSQLPKG コマンドに対する *USE 権限が必要です。

DB2 UDB for iSeries 以外のシステムでのパッケージの作成

DB2 UDB for iSeries 以外のシステム用にプログラムと SQL パッケージを作成して、そのリレーショナル・データベースに固有の SQL ステートメントを使用したいときは、CRTSQLxxx GENLVL パラメーターを 30 にセットしておく必要があります。重大度レベルが 30 より大きくないメッセージが出されたとき、プログラムが作成されます。重大度レベルが 30 より大きいメッセージが出されたとき、そのステートメントはどのリレーショナル・データベースでも無効であると考えられます。たとえば、ホスト変数が未定義であったり、定数が無効であるとき、重大度が 30 より大きなメッセージが出されます。

10 より大きい GENLVL を指定して実行するときは、予期しないメッセージがないか事前コンパイラー・リストを調べてください。 DB2 ユニバーサル・データベース用にパッケージを作成しているときには、GENLVL パラメーターを 20 未満に設定する必要があります。

RDB パラメーターによって DB2 UDB for iSeries システムでないシステムを指定する場合は、次のオプションは CRTSQLxxx コマンドで使用しないでください。

- COMMIT(*NONE)
- OPTION(*SYS)
- DATFMT(*MDY)
- DATFMT(*DMY)
- DATFMT(*JUL)
- DATFMT(*YMD)

- DATFMT(*JOB)
- DYNUSRPRF(*OWNER)
- TIMFMT(*HMS)　TIMSEP(*BLANK) または TIMSEP(',') を指定した場合
- SRTSEQ(*JOBRUN)
- SRTSEQ(*LANGIDUNQ)
- SRTSEQ(*LANGIDSHR)
- SRTSEQ(ライブラリー名/表名)

注: DB2 ユニバーサル・データベース・サーバーに接続する場合、次の追加規則が適用されます。

- 指定された日時形式は同じ形式設定でなければならない。
- TEXT パラメーターに *BLANK の値が使用されていなければならない。
- デフォルトのスキーマ (DFTRDBCOL) がサポートされていない。
- パッケージが作成されているソース・プログラムの CCSID が 65535 であってはならない。 65535 が使用されている場合、空のパッケージが作成されます。

ターゲット・リリース (TGTRLS)

パッケージを作成している間に、どのリリースがその機能をサポートできるか判別するために SQL ステートメントが検査されます。このリリースは、パッケージの復元レベルとしてセットされます。たとえば、パッケージに FOREIGN KEY 制約を追加する CREATE TABLE ステートメントが入っている場合は、パッケージの復元レベルはバージョン 3 リリース 1 になります。なぜなら、これより前のリリースは FOREIGN KEY 制約をサポートしていないからです。TGTRLS パラメーターが *CURRENT のときは、TGTRLS メッセージは抑止されます。

SQL ステートメントのサイズ

SQL パッケージ作成機能は、事前コンパイラーが処理できるものと同サイズの SQL ステートメントを扱えないことがあります。 SQL プログラムの事前コンパイル時に、SQL ステートメントはプログラムの関連スペースに入れられます。そのようなときは、各トークンはブランクで区切られます。さらに、RDB パラメーターの指定があるときは、ソース・ステートメントのホスト変数は 'H' で置き換えられます。SQL パッケージ作成機能からこのステートメントがアプリケーション・サーバーに渡されるとき、そのステートメントのホスト変数も一緒に渡されます。トークンとトークンの間にブランクを追加したり、ホスト変数を置き換えたりすると、ステートメントが最大 SQL ステートメント・サイズを超えることがあります (SQL0101 理由コード 5)。

パッケージを必要としないステートメント

場合によっては、SQL パッケージを作成しようとしたが、SQL パッケージが作成されず、それでもプログラムが実行されることがあります。このようなことは、SQL パッケージを実行させる必要のない SQL ステートメントだけがプログラムに含まれているときに起こります。たとえば、SQL ステートメントの DESCRIBE TABLE だけを含んでいるプログラムは、SQL パッケージの作成時にメッセージ SQL5041 を生成します。SQL パッケージを必要としない SQL ステートメントには、次のものがあります。

- COMMIT

- CONNECT
- DESCRIBE TABLE
- DISCONNECT
- RELEASE
- RELEASE SAVEPOINT
- ROLLBACK
- SAVEPOINT
- SET CONNECTION

オブジェクト・タイプのパッケージ

SQL パッケージは常に非 ILE オブジェクトとして作成され、常に省略時活動化グループで実行されます。

ILE プログラムおよび ILE サービス・プログラム

SQL ステートメントが含まれている複数のモジュールをバインドする ILE プログラムおよび ILE サービス・プログラムの場合、各モジュールごとに別個の SQL パッケージが必要です。

パッケージ作成の接続

パッケージ作成の際に行われる接続のタイプは、RDBCNNMTH パラメーターを使用して要求した接続のタイプに基づいています。RDBCNNMTH(*DUW) を指定した場合は、コミットメント制御が使用され、接続は読み取り専用の接続になる可能性があります。接続が読み取り専用である場合は、パッケージの作成が失敗します。

作業単位

パッケージの作成は暗黙にコミットまたはロールバックを実行するので、コミット定義はパッケージを作成する前に作業単位の境界になければなりません。コミット定義が作業単位の境界にあるためには、次の条件がすべて満たされている必要があります。

- SQL が作業単位の境界にある。
- オープンになっているコミットメント制御を使用する、ローカルまたは DDM ファイルがなく、しかもクローズになっている保留中の変更が指定されているローカル・ファイルまたは DDM ファイルがない。
- 登録されている API 資源がない。
- DRDA または DDM に関連付けられていない LU 6.2 資源がない。

ローカルにパッケージを作成

RDB パラメーターに指定する名前は、ローカル・システムの名前にすることができます。指定した名前がローカル・システムの名前であるときは、SQL パッケージはそのローカル・システムに作成されます。 SQL パッケージは保管して (SAVOBJ コマンド) から、別のサーバーに復元する (RSTOBJ コマンド) ことができます。プログラムを実行させるとき、ローカル・システムに接続されていると、SQL パッケージは使用されません。RDB パラメーターに *LOCAL を指定すると、*SQLPKG オブジェクトは作成されませんが、パッケージ情報は *PGM オブジェクトに保管されます。

ラベル

LABEL ON ステートメントを使用すると、SQL パッケージの記述を作成することができます。

整合性トークン

プログラムとその関連 SQL パッケージに整合性トークンがあり、これは、SQL パッケージの呼び出しを行うときに検査されます。整合性トークンが一致しないとパッケージを使用することはできません。プログラムと SQL パッケージとが整合性がないように見えることがあります。プログラムが iSeries システムにあって、アプリケーション・サーバーが別の iSeries システムであるとします。 この場合、プログラムはセッション A で実行され、セッション B (そこでは、SQL パッケージも再作成される) でも再作成されます。セッション A で実行されるプログラムを次回に呼び出すと、整合性トークン・エラーが起こります。呼び出しごとに SQL パッケージを探すことを避けるために、SQL は各セッションで使用される SQL パッケージのアドレス・リストをもっています。セッション B で SQL パッケージが再作成されると、古い SQL パッケージは QRPLOBJ ライブラリーに移されます。セッション A での SQL パッケージに対するアドレスはまだ無効のままです。 (このようなことは、プログラムを実行しているセッションからプログラムと SQL パッケージを作成するか、あるいは遠隔コマンドを投入して古い SQL パッケージを削除してからプログラムを作成する方法をとれば、避けることができます。)

新しい SQL パッケージを使用するためには、遠隔システムとの接続を終了させなければなりません。セッションをサインオフしてからもう一度サインオンするか、対話式 SQL (STRSQL) コマンドを使用して DISCONNECT を出すか (非保護のネットワーク接続の場合)、RELEASE に引き続き COMMIT を出す (保護接続の場合) ことができます。次に RCLDDMCNV を使用すれば、接続は終了します。もう一度プログラムを呼び出してください。

SQL および再帰

すでに事前コンパイルしている途中でアテンション・キー・プログラムから SQL を呼び出すと、予測不能な結果を受け取ることになります。

CRTSQLxxx、CRTSQLPKG、STRSQL の各コマンドおよび SQL 実行時環境は再帰的ではありません。再帰を試みると、これらのコマンドから予測不能な結果が作成されます。これらのコマンドのいずれかを実行させ (あるいは SQL ステートメントが組み込まれているプログラムを実行させ)、そのコマンドの実行が終わる前にジョブを中断させ、別の SQL 機能を始動させると、再帰が行われます。

SQL 用の CCSID に関する考慮事項

分散アプリケーションを実行する場合にユーザーのシステムのいずれかが iSeries システムでないときは、iSeries サーバー上のジョブ CCSID の値を 65535 にセットすることはできません。

SQL パッケージ作成のアプリケーション・リクエスター側は、リモート・システムが SQL パッケージを作成するよう要求する前に、RDB パラメーターで指定された名前、SQL パッケージ名、ライブラリー名、および SQL パッケージのテキストを、常にジョブの CCSID から CCSID 500 に変換します。これは DRDA で必要に

なるためです。遠隔リレーショナル・データベースが iSeries システムにあるときは、これらの名前は CCSID 500 から CCSID に変換されません。

区切り文字で区切った ID を表、視点、索引、スキーマ、ライブラリー、または SQL パッケージの名前に使用することはお勧めしません。CCSID が異なるシステムの間では名前の変換は行われません。システム A が CCSID 37 で稼動し、システム B が CCSID 500 で稼動している例を考えます。

- システム A に表を作成する ″a¬b|c″ という名前のプログラムを作成します。
- 次に、プログラム ″a¬b|c″ をシステム A に保管してから、それをシステム B に復元します。
- ¬ のコード点は、CCSID 37 の中では x'5F' であるのに対し、CCSID 500 では x'BA' になっています。
- システム B では、名前は ″a[b]c″ と表示されます。″a¬b|c.″ という名前の表を参照するプログラムを作成しても、そのプログラムはこの表を見つけることができません。

SQL オブジェクト名では、単価記号 (@)、ポンド記号 (#)、およびドル記号 ($) は使用しないでください。これらのコード点は使用する CCSID によって異なります。区切り文字で区切った名前やこれら 3 つの記号を使用すると、将来発表されるリリースでネーム・レゾリューション機能が作動しなくなるおそれがあります。

接続管理および活動化グループ

詳細については、以下のトピックを参照してください。

- 『接続および会話』
- 366 ページの『PGM1 のソース・コード』
- 367 ページの『PGM2 のソース・コード』
- 367 ページの『PGM3 のソース・コード』
- 369 ページの『同じリレーショナル・データベースへの多重接続』
- 370 ページの『省略時活動化グループの場合の暗黙の接続管理』
- 370 ページの『非省略時活動化グループの場合の暗黙の接続管理』

接続および会話

DRDA が TCP/IP を使用していないころ、「接続」という用語の意味はあいまいではありませんでした。この用語は、SQL の側から見た接続を指していました。すなわち、接続は、RDB に対して CONNECT TO が行われた時点で開始し、DISCONNECT が行われた時点、または RELEASE ALL に続いて正常な COMMIT が行われた時点で終了しました。APPC 会話が継続されるかどうかは、ジョブの DDMCNV 属性値や、会話が iSeries に対するものであるか他のタイプのシステムに対するものであるかによって決まります。

TCP/IP 用語には、「会話」という用語は含まれていません。ただし、類似の概念は存在します。 DRDA が TCP/IP をサポートするようになったので、説明が APPC の会話に限定されていないかぎり、本書では、「会話」という用語は、より一般的な「接続」という用語に置き換えられます。したがって読者は、上述したタイプの

SQL 接続と、「会話」という用語の代わりに使用される「ネットワーク」接続という、2 つの異なるタイプの接続があることに注意する必要があります。

2 タイプの接続の間で混乱が生じる可能性のある個所では、「SQL」または「ネットワーク」という修飾語により、意図されている意味が理解できるようになっています。

SQL 接続は活動化グループ・レベルで管理されます。ジョブ内の各活動化グループは、それ自身の接続を管理し、これらの接続は活動化グループ間では共有されません。省略時活動化グループで実行されるプログラムの場合、接続は、バージョン 2 リリース 3 より前と同様に管理されます。

複数の活動化グループで実行されるアプリケーションの例を以下に示します。この例では、活動化グループ間の対話、接続管理、およびコミットメント制御を示します。推奨するコーディング・スタイルを示しているわけでは**ありません**。

PGM1 のソース・コード

```
....
EXEC SQL
  CONNECT TO SYSB
END-EXEC.
EXEC SQL
  SELECT ....
END-EXEC.
CALL PGM2.
....
```

図 9. PGM1 のソース・コード

PGM1 用のプログラムと SQL パッケージを作成するコマンドは次のとおりです。

```
CRTSQLCBL PGM(PGM1) COMMIT(*NONE) RDB(SYSB)
```

PGM2 のソース・コード

```
   ...
           EXEC SQL
             CONNECT TO SYSC;
           EXEC SQL
             DECLARE C1 CURSOR FOR
                 SELECT ....;
           EXEC SQL
             OPEN C1;
           do {
                EXEC SQL
                  FETCH C1 INTO :st1;
                EXEC SQL
                  UPDATE ...
                      SET COL1 = COL1+10
                      WHERE CURRENT OF C1;
                PGM3(st1);
           } while SQLCODE == 0;
           EXEC SQL
            CLOSE C1;
           EXEC SQL COMMIT;
....
```

図 10. PGM2 のソース・コード

PGM2 用のプログラムと SQL パッケージを作成するコマンドは次のとおりです。

```
CRTSQLCI OBJ(PGM2) COMMIT(*CHG) RDB(SYSC) OBJTYPE(*PGM)
```

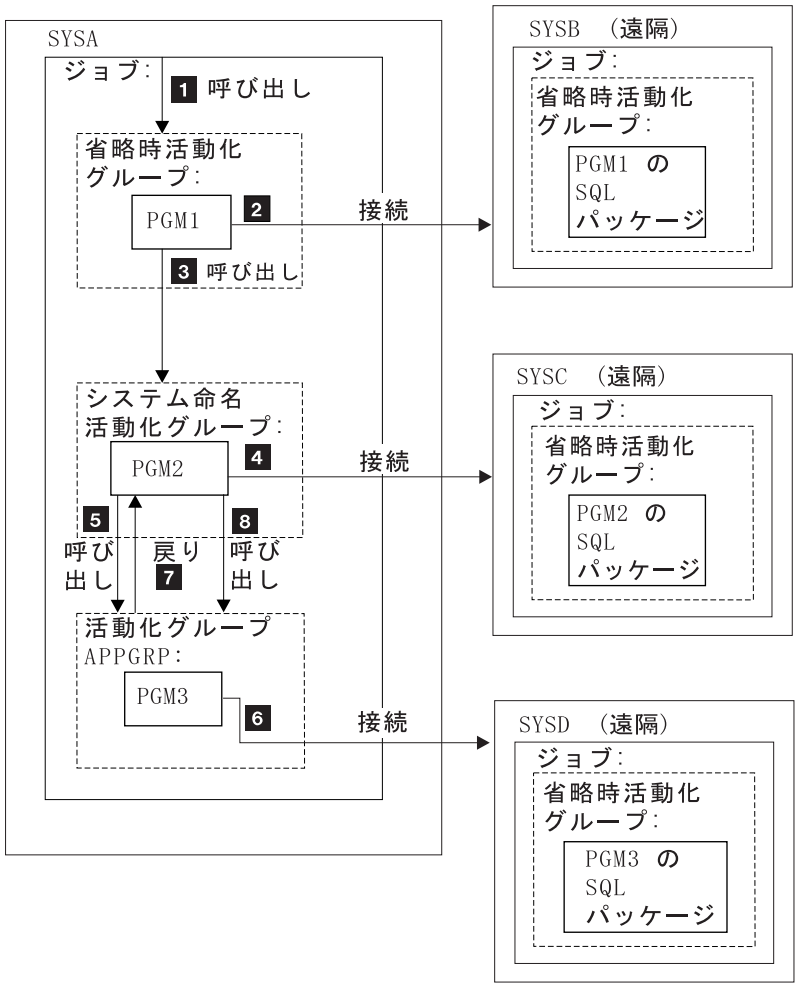
PGM3 のソース・コード

```
   ...
    EXEC SQL
      INSERT INTO TAB VALUES(:st1);
    EXEC SQL COMMIT;
   ....
```

図 11. PGM3 のソース・コード

PGM3 用のプログラムと SQL パッケージを作成するコマンドは、次のとおりです。

```
CRTSQLCI OBJ(PGM3) COMMIT(*CHG) RDB(SYSD) OBJTYPE(*MODULE)
CRTPGM PGM(PGM3) ACTGRP(APPGRP)
CRTSQLPKG PGM(PGM3) RDB(SYSD)
```



上の例で、PGM1 は CRTSQLCBL コマンドを使用して作成された非 ILE プログラムです。このプログラムは、省略時活動化グループで実行されます。 PGM2 は、CRTSQLCI コマンドを使用して作成され、システムが命名した活動化グループ内で実行されます。 PGM3 も CRTSQLCI コマンドを使用して作成されますが、APPGRP という名前の活動化グループ内で実行されます。APPGRP は ACTGRP パラメーターの省略時値ではないので、CRTPGM コマンドは別に出されます。CRTPGM コマンドの後に、SYSD リレーショナル・データベース上に SQL パッケージ・オブジェクトを作成する CRTSQLPKG コマンドが続きます。 上の例で、ユーザーは、ジョブ・レベル・コミットメント定義を明示的に開始していません。SQL はコミットメント制御を暗黙に開始します。

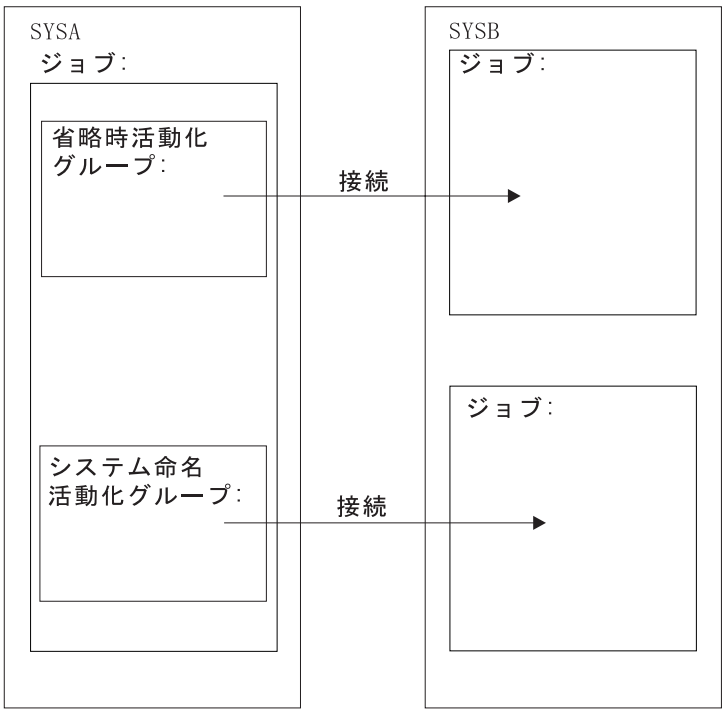
1. PGM1 は、省略時活動化グループで呼び出され実行されます。
2. PGM1 はリレーショナル・データベース SYSB に接続され、SELECT ステートメントを実行します。
3. PGM1 は次に PGM2 を呼び出し、PGM2 はシステム命名活動化グループで実行されます。
4. PGM2 はリレーショナル・データベース SYSC への接続を行います。 PGM1 と PGM2 は別の活動化グループ内にあるので、システム命名活動化グループ内の PGM2 によって開始された接続によって、省略時の活動化グループ内の PGM1 によって開始された接続は切り離されません。したがって、両方の接続が

活動状態になります。PGM2 はカーソルをオープンし、行を取り出して更新します。 PGM2 はコミットメント制御の下で実行中で、作業単位内にあり、接続可能状態ではありません。

5. PGM2 は PGM3 を呼び出し、PGM3 は活動化グループ APPGRP 内で実行されます。
6. INSERT ステートメントは、活動化グループ APPGRP 内で実行される最初のステートメントです。最初の SQL ステートメントによって、リレーショナル・データベース SYSD への暗黙の接続が行われます。 1 つの行が、リレーショナル・データベース SYSD 上の表 TAB に挿入されます。この挿入は、次にコミットされます。コミットメント制御は、活動化グループ内のコミット有効範囲で SQL によって開始されているので、システム命名活動化グループ内で保留中の変更はコミットされません。
7. 次に、PGM3 が終了し、制御は PGM2 に戻ります。PGM2 は別の行を取り出して更新します。
8. 行を挿入するために、PGM3 が再び呼び出されます。PGM3 の最初の呼び出しのときに、暗黙の接続が行われています。 PGM3 の呼び出しと次の呼び出しの間で活動化グループは終了していないので、2 番目以降の呼び出しでは接続は行われません。最後に、すべての行が PGM2 によって処理され、システム命名活動化グループに関連付けられた作業単位がコミットされます。

同じリレーショナル・データベースへの多重接続

別々の活動化グループが同じリレーショナル・データベースに接続されると、各 SQL 接続ごとに、固有のネットワーク接続および固有のアプリケーション・サーバー・ジョブが生じます。コミットメント制御を指定して活動化グループを実行すると、ジョブ・レベル・コミットメント定義を使用しない限り、1 つの活動化グループ内でコミットされた変更は、他の活動化グループではコミットされません。



省略時活動化グループの場合の暗黙の接続管理

アプリケーション・リクエスターはアプリケーション・サーバーと暗黙に接続することができます。暗黙 SQL 接続は、アプリケーション・リクエスターが省略時活動化グループ内の最初の活動状態の SQL プログラムから最初の SQL ステートメントが出されたことを検出し、次の条件を満足しているとき行われます。

- 出された SQL ステートメントがパラメーターを指定した CONNECT ステートメントでない。
- SQL が省略時の活動化グループで活動状態にない。

分散プログラムの場合は、暗黙 SQL 接続は RDB パラメーターに指定されたリレーショナル・データベースに対して行われます。非分散プログラムの場合は、暗黙 SQL 接続はローカル・リレーショナル・データベースに対して行われます。

SQL が活動状態でなくなったときは、SQL は省略時の活動化グループで活動状態にある接続をすべて終了します。SQL は、次の場合に活動状態でなくなります。

- アプリケーション・リクエスターがプロセスの最初の活動状態の SQL プログラムが終了したことを検出した場合で、次のことが当てはまる場合。
  - 保留中の SQL の変更がない
  - 保護接続を使用する接続がない
  - SET TRANSACTION ステートメントが活動状態でない
  - CLOSQLCSR(*ENDJOB) を指定して事前コンパイルされたプログラムが実行されなかった

  保留中の変更、保護接続、または活動状態の SET TRANSACTION ステートメントがある場合、SQL は終了状態に入れられます。 CLOSQLCSR(*ENDJOB) を指定して事前コンパイルされたプログラムが実行された場合、SQL はジョブが終了するまで省略時の活動化グループに対して活動状態のままになります。
- SQL が終了状態にある場合で、作業単位が終了したとき。これは、COMMIT または ROLLBACK コマンドが SQL プログラムの外で出されたときに起きます。
- ジョブが終了したとき。

非省略時活動化グループの場合の暗黙の接続管理

アプリケーション・リクエスターはアプリケーション・サーバーと暗黙に接続することができます。暗黙 SQL 接続は、アプリケーション・リクエスターが活動化グループに対して出された最初の SQL ステートメントを検出し、それがパラメーターを指定した CONNECT ステートメントでない場合に行われます。

分散プログラムの場合は、暗黙 SQL 接続は RDB パラメーターに指定されたリレーショナル・データベースに対して行われます。非分散プログラムの場合は、暗黙 SQL 接続はローカル・リレーショナル・データベースに対して行われます。

暗黙切断は、プロセスの以下の時点で行われます。

- 活動化グループの終了時でコミットメント制御が活動状態でない場合、活動化グループ・レベルのコミットメント制御が活動状態の場合、またはジョブ・レベル・コミットメント定義が作業単位の境界にある場合。

ジョブ・レベルのコミットメント定義が活動状態にあり、作業単位の境界にない場合、SQL は終了状態になります。

- ジョブ・レベルのコミットメント定義がコミットされているか、ロールバックされているときに SQL が終了状態にある場合。
- ジョブが終了したとき。

分散サポート

DB2 UDB for iSeries は 2 つのレベルの分散リレーショナル・データベースをサポートします。

- 遠隔作業単位 (RUW)

  遠隔作業単位は、SQL ステートメントの準備と実行を、1 つの作業単位内で 1 つのアプリケーション・サーバーでしか行わないことを指しています。アプリケーション・リクエスターでアプリケーション処理を指定した活動化グループは、アプリケーション・サーバーに接続でき、しかも 1 つまたは複数の作業単位内で、アプリケーション・サーバー上のオブジェクトを参照する静的または動的 SQL ステートメントをいくつでも実行できます。遠隔作業単位は、DRDA レベル 1 とも呼ばれます。

- 分散作業単位 (DUW)

  分散作業単位は、SQL ステートメントの準備と実行が 1 つの作業単位内で複数のアプリケーション・サーバーで行えることを指します。ただし、単一 SQL ステートメントは、単一アプリケーション・サーバーにあるオブジェクトしか参照することができません。 分散作業単位は、DRDAレベル 2 とも呼ばれます。

  分散作業単位を使用すると、次のことが可能になります。

  – 1 つの論理作業単位内の複数のアプリケーション・サーバーへのアクセスの更新

    あるいは

  – 1 つの論理作業単位内での、複数のアプリケーション・サーバーへの読み取りアクセスによる単一アプリケーション・サーバーへのアクセスの更新

  複数のアプリケーション・サーバーを作業単位内で更新できるかどうかは、アプリケーション・リクエスターでの同期点管理プログラム、アプリケーション・サーバーでの同期点管理プログラム、およびアプリケーション・リクエスターおよびアプリケーション・サーバー間の 2 フェーズ・コミット・プロトコル・サポートがそれぞれ存在するかどうかによって異なります。

  同期点管理プログラムは、2 フェーズ・コミット・プロトコルにおける参加者間でのコミット操作およびロールバック操作を調整するシステムのコンポーネントです。分散された更新を実行している場合、各種のシステム上の同期点管理プログラムが協力して資源が確実に矛盾のない状態になるようにします。同期点管理プログラムが使用するプロトコルおよび流れは、2 フェーズ・コミット・プロトコルとも呼ばれます。

  2 フェーズ・コミット・プロトコルを使用する場合、接続は保護資源です。それ以外の場合、接続は非保護資源です。

  システム間で使用されるデータ転送プロトコルのタイプは、ネットワーク接続が保護接続になるか非保護接続になるかに影響します。V5R1 より前では、TCP/IP

接続は常に無保護接続でした。したがって、限られた方法でしか分散作業単位にかかわることができませんでした。 V5R1 では、TCP/IP 付きの DUW のフル・サポートが追加されました。

たとえば、プログラムからの最初の接続が TCP/IP を介して V5R1 より前のサーバーに対して行われた場合、更新を行うことはできますが、後続の接続は、APPC を介したものであっても、読み取り専用になります。

対話式 SQL を使用する場合、最初の SQL 接続はローカル・システムに対して行われます。 したがって、V5R1 より前の環境では、TCP/IP を使ってリモート・システムを更新するには、RELEASE ALL の後に COMMIT を実行してすべての SQL 接続を終了してから、CONNECT TO remote-tcp-system を行う必要があります。

接続タイプの決定

遠隔接続を確立した場合、非保護接続または保護接続のいずれかが使用されます。コミット可能な更新に関しては、接続を確立したときに更新可能であるかどうかにかかわらず、この SQL 接続が読み取り専用でも、更新可能でも、未知でも構いません。コミット可能な更新は、コミット制御下で実行される任意の挿入、削除、更新、または DDL ステートメントのことです。接続が読み取り専用である場合は、COMMIT(*NONE) を使用している変更は引き続き実行できます。 CONNECT または SET CONNECTION を行うと、SQLCA の SQLERRD(4) が接続のタイプを示します。 SQLERRD(4) は、この接続が非保護ネットワーク接続を使用するか、保護ネットワーク接続を使用するかについても示します。特定の値は、次のとおりです。

1. コミット可能な更新は、この接続で実行できます。接続は保護されていません。これは、次の場合に起こります。
   - 接続が、遠隔作業単位 (RDBCNNMTH(*RUW)) を使用して確立した場合。これには、遠隔作業単位によるローカル接続とアプリケーション・リクエスター・ドライバー (ARD) 接続も含まれます。
   - 分散作業単位 (RDBCNNMTH(*DUW)) を使用して接続を確立し、次のすべてがあてはまる場合。
     - 接続がローカルでない。
     - アプリケーション・サーバーが、分散作業単位をサポートしない。たとえば、バージョン 3 リリース 1 より前の OS/400 のリリースでの DB2 UDB for iSeries アプリケーション・サーバー。
     - 接続の指示を出しているプログラムのコミットメント制御レベルが *NONE ではない。
     - コミット可能な更新を実行できるその他のアプリケーション・サーバー (ローカルを含む) への接続が存在しないか、またはすべての接続が分散作業単位をサポートしないアプリケーション・サーバーへの読み取り専用接続であるかのいずれか。
     - コミットメント制御下でコミットメント定義用にオープンになっている更新可能なローカル・ファイルがない。
     - コミットメント制御下でコミットメント定義用に異なる接続を使用する更新可能なオープン DDM ファイルがない。
     - コミットメント定義用の API コミットメント制御資源がない。
     - コミットメント定義用に登録している保護接続がない。

コミットメント制御により実行する場合、SQL は、遠隔接続の場合は 1 フェーズの更新可能な DRDA 資源、ローカル接続および ARD 接続の場合は 2 フェーズの更新可能な DRDA 資源をそれぞれ登録します。

2. コミット可能な更新はこの接続で行うことはできません。この接続は読み取り専用です。ネットワーク接続は保護されていません。

   これは、遠隔作業単位接続管理 (*RUW) を使ってコンパイルされたアプリケーションでは決して起こりません。

   分散作業単位アプリケーションの場合、接続を確立したときに次があてはまる場合に限り起こります。

   - 接続がローカルでない。
   - アプリケーション・サーバーは、分散作業単位をサポートしない。
   - 少なくとも次のいずれか 1 つがあてはまる場合。
     - 接続の指示を出しているプログラムのコミットメント制御レベルが *NONE になっている。
     - 分散作業単位をサポートしないアプリケーション・サーバーに別の接続が存在し、しかもそのアプリケーション・サーバーがコミット可能な更新を実行できる。
     - 分散作業単位 (ローカルを含む) をサポートするアプリケーション・サーバーに別の接続が存在する。
     - コミットメント制御下でコミットメント定義用に更新可能なオープン・ローカル・ファイルがある。
     - コミットメント制御下でコミットメント定義用に異なる接続を使用する更新可能なオープン DDM ファイルがある。
     - コミットメント定義用の 1 フェーズの API コミットメント制御資源がない。
     - コミットメント定義用に登録している保護接続がある。

   コミットメント制御により実行する場合、SQL は 1 フェーズの読み取り専用資源を登録します。

3. コミット可能な更新を実行できるかどうかは不明です。 この接続は保護されています。

   これは、遠隔作業単位接続管理 (*RUW) を使ってコンパイルされたアプリケーションでは決して起こりません。

   分散作業単位アプリケーションの場合、これは接続が確立したときに次があてはまる場合に起こります。

   - 接続がローカルでない。
   - 接続の指示を出しているプログラムのコミットメント制御レベルが *NONE ではない。
   - アプリケーション・サーバーが、分散作業単位と 2 フェーズ・コミット・プロトコル (保護接続) の両方をサポートしている。

   コミットメント制御で実行している場合、SQL は 2 フェーズの未決定の資源を登録します。

4. コミット可能な更新を実行できるかどうかは不明です。 接続は保護されていません。

   これは、遠隔作業単位接続管理 (*RUW) を使ってコンパイルされたアプリケーションでは決して起こりません。

   分散作業単位接続の場合、これは接続が確立したときに次があてはまる場合に起こります。

   - 接続がローカルでない。
   - アプリケーション・サーバーが、分散作業単位をサポートする。
   - アプリケーション・サーバーが 2 フェーズ・コミット・プロトコル (保護接続) をサポートしないか、または接続の指示を出しているプログラムのコミットメント制御レベルが *NONE になっている。

   コミットメント制御で実行している場合、SQL は 2 フェーズの DRDA 未決定資源を登録します。
5. コミット可能な更新を実行できるかどうか、しかも接続が分散作業単位を使用するローカル接続または分散作業単位を使用する ARD 接続かどうかは不明です。

   コミットメント制御で実行している場合、SQL は 2 フェーズの DRDA 未決定資源を登録します。

2 フェーズおよび 1 フェーズのリソースについて詳しくは、バックアップおよび回復の手引き のコミットメント制御のトピックを参照してください。

次の表は、遠隔分離作業単位接続の場合に行われる接続のタイプを要約しています。 SQLERRD(4) は、正しい CONNECT および SET CONNECTION の各ステートメントで設定されます。

表 39. 接続タイプの要約

コミットメント制御下の接続	アプリケーション・サーバーによる 2 フェーズ・コミットのサポート	アプリケーション・サーバーによる分散作業単位のサポート	その他の更新可能な登録済み 1 フェーズ資源	SQLERRD(4)
いいえ	いいえ	いいえ	いいえ	2
いいえ	いいえ	いいえ	はい	2
いいえ	いいえ	はい	いいえ	4
いいえ	いいえ	はい	はい	4
いいえ	はい	いいえ	いいえ	2
いいえ	はい	いいえ	はい	2
いいえ	はい	はい	いいえ	4
いいえ	はい	はい	はい	4
はい	いいえ	いいえ	いいえ	1
はい	いいえ	いいえ	はい	2
はい	いいえ	はい	いいえ	4
はい	いいえ	はい	はい	4
はい	はい	いいえ	いいえ	N/A *

表 39. 接続タイプの要約 (続き)

コミットメント制御下の接続	アプリケーション・サーバーによる 2 フェーズ・コミットのサポート	アプリケーション・サーバーによる分散作業単位のサポート	その他の更新可能な登録済み 1 フェーズ資源	SQLERRD(4)
はい	はい	いいえ	はい	N/A *
はい	はい	はい	いいえ	3
はい	はい	はい	はい	3

* DRDAは、遠隔作業単位 (DRDA1) だけをサポートするアプリケーション・サーバーに使用される保護接続を許可していません。これには、すべての DB2 for iSeries TCP/IP 接続が含まれます。

接続およびコミットメント制御に関する制約事項

コミットメント制御によりいつ接続できるかについては、いくつかの制約事項があります。これらの制約事項は、COMMIT(*NONE) を使用して接続を確立した場合を除いて、コミットメント制御を使用してステートメントを実行する試みにも適用されます。

2 フェーズの未決定または更新可能な資源を登録したか、または 1 フェーズの更新可能な資源を登録した場合、別の 1 フェーズの更新可能な資源は登録できません。

また、保護接続が非活動状態で、DDMCNV ジョブ属性が *KEEP である場合は、これらの未使用の DDM 接続により、RUW 接続管理でコンパイルしたプログラムの中の CONNECT ステートメントも失敗します。

RUW 接続管理により実行してジョブ・レベルのコミットメント定義を使用する場合、いくつかの制約事項があります。

- ジョブ・レベルのコミットメント定義が複数の活動化グループによって使用された場合、すべての RUW の接続先はローカル・リレーショナル・データベースでなければなりません。
- 接続が遠隔の場合は、1 つの活動化グループしか RUW 接続に対してジョブ・レベルのコミットメント定義を使用できません。

接続状況の決定

パラメーターを指定していない CONNECT ステートメントを使用すると、現行接続が現行の作業単位に対して更新可能かまたは読み取り専用かを判別することができます。SQLERRD(3) に 1 か 2 の値が戻されます。SQLERRD(3) の値は、次のように決定されます。

1. コミット可能な更新は、作業単位の接続上で実行することができます。

   これは、次のいずれか 1 つがあてはまる場合に起こります。

   - SQLERRD(4) の値が 1 の場合。
   - 次のすべてがあてはまる場合。
     - SQLERRD(4) の値が 3 か 5 である。

– コミット可能な更新を行うことのできる分散作業単位をサポートしないアプリケーション・サーバーへの接続がない。
– 次のいずれか 1 つがあてはまる場合。
- 最初のコミット可能な更新が、保護接続を使用している接続上で実行されるか、ローカル・データベース上で実行されるか、または ARD プログラムへの接続上で実行される場合。
- コミットメント制御下に、オープンになっている更新可能なローカル・ファイルがある場合。
- 保護接続を使用しているオープンになっている更新可能な DDM ファイルがある場合。
- 2 フェーズの API コミットメント制御資源がある場合。
- コミット可能な更新が行われなかった場合。

• 次のすべてがあてはまる場合。
– SQLERRD(4) の値が 4 である。
– コミット可能な更新を行うことのできる分散作業単位をサポートしないアプリケーション・サーバーへの接続が他にない。
– 最初のコミット可能な更新が、この接続で行われるか、コミット可能な更新が行われなかった。
– 保護接続を使用するオープンになっている更新可能な DDM ファイルがない。
– コミットメント制御下でオープンになっている更新可能なローカル・ファイルがない。
– 2 フェーズの API コミットメント制御資源がない。

2. この作業単位に対して接続上でコミット可能な更新を行うことができません。

これは、次のいずれか 1 つがあてはまる場合に起こります。

• SQLERRD(4) の値が 2 の場合。
• SQLERRD(4) の値が 3 か 5 で、次のいずれか 1 つがあてはまる場合。
– 遠隔作業単位しかサポートしない更新可能なアプリケーション・サーバーに接続が存在する場合。
– 最初のコミット可能な更新が非保護接続を使用する接続上で実行された場合。
• SQLERRD(4) の値が 4 で、次のいずれか 1 つがあてはまる場合。
– 遠隔作業単位しかサポートしない更新可能なアプリケーション・サーバーに接続が存在する場合。
– 最初のコミット可能な更新がこの接続上で実行されなかった場合。
– 保護接続を使用しているオープンになっている更新可能な DDM ファイルがある場合。
– コミットメント制御下でオープンになっている更新可能なローカル・ファイルがある場合。
– 2 フェーズの API コミットメント制御資源がある場合。

次の表は、SQLERRD(4) の値に基づく SQLERRD(3) の決定方法、遠隔作業単位だけをサポートするアプリケーション・サーバーに対して更新可能な接続があるかど

うか、および最初のコミット可能な更新が起こる場所について要約しています。

表 40. SQLERRD(3) 値の決定に関する要約

SQLERRD(4)	更新可能な遠隔作業単位アプリケーション・サーバーへの接続の有無	最初のコミット可能な更新の発生場所*	SQLERRD(3)
1	--	--	1
2	--	--	2
3	はい	--	2
3	いいえ	更新なし	1
3	いいえ	1 フェーズ	2
3	いいえ	この接続	1
3	いいえ	2 フェーズ	1
4	はい	--	2
4	いいえ	更新なし	1
4	いいえ	1 フェーズ	2
4	いいえ	この接続	1
4	いいえ	2 フェーズ	2
5	はい	--	2
5	いいえ	更新なし	1
5	いいえ	1 フェーズ	2
5	いいえ	この接続	1
5	いいえ	2 フェーズ	1

* この列の用語は次のとおりです。

- *更新なし* は、コミット可能な更新が行われておらず、保護接続を使用して更新のためにオープンになっている DDM ファイルがなく、更新のためにオープンになっているローカル・ファイルがなく、しかも登録されているコミットメント制御 API がないことを示します。
- *1 フェーズ* は、非保護接続を使用して最初のコミット可能な更新が行われたか、非保護接続を使用して DDM ファイルが更新のためにオープンになっていることを示します。
- *2 フェーズ* は、コミット可能な更新が 2 フェーズの分散作業単位アプリケーション・サーバーで実行されたか、DDM ファイルが保護接続を使用して更新のためにオープンされたか、コミットメント制御 API が登録されたか、またはコミットメント制御下でローカル・ファイルが更新のためにオープンされたことを示します。

SQLERRD(4) の値が 3、4、または 5 (ARD プログラムであるため) で、SQLERRD(3) の値が 2 のときに、接続上でコミット可能な更新を行おうとすると、作業単位はロールバックを必要とする状態に置かれます。作業単位がロールバックを必要とする状態に入ると、使用できるステートメントは ROLLBACK ステートメントだけになります。その他のステートメントを使用すると、SQLCODE -918 が出されます。

分散作業単位に関する考慮事項

分散作業単位アプリケーションで接続する場合には、考慮すべき点がたくさんあります。この節では、設計に関するいくつかの考慮事項を挙げることにします。

- 作業単位が更新を複数のアプリケーション・サーバーで行い、しかもコミットメント制御を使用する場合は、そこで更新しようとしているすべての接続をコミットメント制御を使用して必要があります。コミットメント制御を使用しないで接続を行い、後でコミット可能な更新を行う場合は、作業単位の読み取り専用接続が行われる可能性が高くなります。
- ローカル・ファイル、DDM ファイル、およびコミットメント制御 API 資源などの SQL 以外の他のコミット資源は、接続の更新可能状況および読み取り専用状況に影響を及ぼします。
- コミットメント制御を使用して、分散作業単位をサポートしないアプリケーション・サーバー (たとえば、TCP/IP を使用する V4R5 iSeries) に接続する場合、その接続は更新可能または読み取り専用のいずれかとなります。 この接続が更新可能な場合は、これが唯一の更新可能な接続になります。

接続の終了

遠隔接続は資源を使用するので、使用しなくなった接続は、できるだけ早く終了する必要があります。接続は暗黙的にも明示的にも終了することができます。接続を暗黙的に終了する時期については、 370 ページの『省略時活動化グループの場合の暗黙の接続管理』および 370 ページの『非省略時活動化グループの場合の暗黙の接続管理』を参照してください。接続は、DISCONNECT ステートメントまたは RELEASE ステートメントの後に正常な COMMIT を続けることにより明示的に終了することができます。 DISCONNECT ステートメントは、非保護接続を使用する接続またはローカル接続を指定する場合に限り使用することができます。DISCONNECT ステートメントは、ステートメントを実行しているときに接続を終了します。 RELEASE ステートメントは、保護接続または非保護接続のいずれでも使用することができます。RELEASE ステートメントを実行すると、接続は終了せず、その代わり解放状態に置かれます。解放状態にある接続は引き続き使用することができます。 接続は、正常な COMMIT を実行するまで終了されません。ROLLBACK または失敗した COMMIT は、解放状態にある接続を終了しません。

遠隔 SQL 接続を確立すると、DDM ネットワーク接続 (APPC 会話または TCP/IP 接続) が使用されます。 SQL 接続を終了すると、ネットワーク接続は未使用な状態に入れられるかまたは除去されます。接続が除去されるか、または未使用な状態に置かれるかは、DDMCNV ジョブ属性によって決まります。ジョブ属性値が *KEEP で、接続先が別の iSeries サーバーである場合は、接続は未使用になります。ジョブ属性値が *DROP で、接続先が別の iSeries サーバーである場合は、接続は除去されます。接続先が iSeries サーバー以外である場合は、接続は常に除去されます。次の状態では *DROP の使用をお勧めします。

- 未使用の接続を維持するコストが高く、接続が比較的早い時期に使用されない場合。
- 一部は RUW 接続管理を使用し、一部は DUW 接続管理を使用してコンパイルするなどプログラムを組み合わせて実行する場合。保護接続が存在する場合は、RUW 接続管理を使用してコンパイルしたプログラムを遠隔ロケーションで実行しようとすると失敗します。
- DDM または DRDA のいずれかを使用して保護接続により実行した場合。未使用の保護接続のコミットおよびロールバックに関して追加のオーバーヘッドが発生する場合。

DDM 接続再利用 (RCLDDMCNV) コマンドを使用すると、すべての未使用接続 (これらがコミット境界にある場合) を終了することができます。

分散作業単位

分散作業単位 (DUW) を使用すると同じ作業単位内で複数のアプリケーション・サーバーにアクセスすることができます。各 SQL ステートメントは、1 つのアプリケーション・サーバーにしかアクセスできません。分散作業単位を使用すると、複数のアプリケーション・サーバーでの変更を 1 つの作業単位内でコミットまたはロールバックすることができます。

分散作業単位接続の管理

CONNECT、SET CONNECTION、DISCONNECT、および RELEASE の各ステートメントは、DUW 環境で接続を管理するために使用されます。分散作業単位 CONNECT は、プログラムを省略時値である RDBCNNMTH(*DUW) を使用して事前コンパイルしたときに実行されます。この形式の CONNECT ステートメントは、既存の接続の切断は行いませんが、直前の接続を休止状態にします。 CONNECT ステートメントで指定しているリレーショナル・データベースが現行接続になります。 CONNECT ステートメントを使用できるのは、新規の接続を開始する場合に限ります。既存の接続間での切り替えを行いたい場合は、SET CONNECTION ステートメントを使用する必要があります。 接続はシステム資源を使用するので、必要でなくなったら接続を終了しなければなりません。 RELEASE または DISCONNECT ステートメントを使用すると、接続を終了することができます。接続を終了させるためには、RELEASE ステートメントの後に正常なコミットを行う必要があります。

次に示すのは、コミットメント制御を使用する DUW 環境で実行される C プログラムの例です。

```
....
 EXEC SQL WHENEVER SQLERROR GO TO done;
 EXEC SQL WHENEVER NOT FOUND GO TO done;
....
 EXEC SQL
   DECLARE C1 CURSOR WITH HOLD FOR
       SELECT PARTNO, PRICE
            FROM PARTS
            WHERE SITES_UPDATED = 'N'
            FOR UPDATE OF SITES_UPDATED;
/*    Connect to the systems      */
EXEC SQL CONNECT TO LOCALSYS;
EXEC SQL CONNECT TO SYSB;
EXEC SQL CONNECT TO SYSC;
/* Make the local system the current connection */
EXEC SQL SET CONNECTION LOCALSYS;
/* Open the cursor */
EXEC SQL OPEN C1;
```

図 12. 分散作業単位プログラムの例 (1/4)

```
while (SQLCODE==0)
   {
    /* Fetch the first row */
    EXEC SQL FETCH C1 INTO :partnumber,:price;
    /* Update the row which indicates that the updates have been
       propagated to the other sites */
    EXEC SQL UPDATE PARTS SET SITES_UPDATED='Y'
                WHERE CURRENT OF C1;
    /* Check if the part data is on SYSB */
    if ((partnumber > 10) && (partnumber < 100))
       {
        /* Make SYSB the current connection and update the price */
        EXEC SQL SET CONNECTION SYSB;
        EXEC SQL UPDATE PARTS
                    SET PRICE=:price
                    WHERE PARTNO=:partnumber;
        }
```

図 12. 分散作業単位プログラムの例 (2/4)

```
    /* Check if the part data is on SYSC */
    if ((partnumber > 50) && (partnumber < 200))
       {
        /* Make SYSC the current connection and update the price */
        EXEC SQL SET CONNECTION SYSC;
        EXEC SQL UPDATE PARTS
                    SET PRICE=:price
                    WHERE PARTNO=:partnumber;
       }
    /* Commit the changes made at all 3 sites */
    EXEC SQL COMMIT;
    /* Set the current connection to local so the next row
       can be fetched */
    EXEC SQL SET CONNECTION LOCALSYS;
   }
done:
```

図 12. 分散作業単位プログラムの例 (3/4)

```
EXEC SQL WHENEVER SQLERROR CONTINUE;
/* Release the connections that are no longer being used */
EXEC SQL RELEASE SYSB;
EXEC SQL RELEASE SYSC;
/* Close the cursor */
EXEC SQL CLOSE C1;
/* Do another commit which will end the released connections.
   The local connection is still active because it was not
   released. */
EXEC SQL COMMIT;
...
```

図 12. 分散作業単位プログラムの例 (4/4)

このプログラムでは、活動状態のアプリケーション・サーバーが 3 つあります。これらは、ローカル・システムの LOCALSYS と 2 つの遠隔システムの SYSB と SYSC です。SYSB と SYSC は、分散作業単位と 2 フェーズ・コミットもサポートします。最初、すべての接続は、トランザクションに関係しているアプリケーション・サーバーのそれぞれに CONNECT ステートメントを使用することによって活動状態になります。DUW を使用する場合、CONNECT ステートメントは直前の接

続は切断しませんが、直前の接続を休止状態にします。すべてのアプリケーション・サーバーを接続すると、ローカル接続が SET CONNECTION ステートメントにより現行接続になります。これにより、カーソルがオープンになってデータの最初の行が取り出されます。次に、どのアプリケーション・サーバーでデータを更新する必要があるかが判別されます。 SYSB を更新する必要がある場合、SET CONNECTION ステートメントにより SYSB が現行接続になり、更新が実行されます。 同じことが SYSC についても行われます。 これで、変更がコミットされます。 2 フェーズ・コミットを使用しているので、変更はローカル・システムおよび 2 つの遠隔システムでコミットされることが保証されます。カーソルに WITH HOLD が宣言されているので、コミットの後でもカーソルはオープンされたままになります。次に現行接続がローカル・システムに変更され、データの次の行を取り出せるようになります。 取り出し、更新、およびコミットの一連の動作は、すべてのデータが処理されるまで繰り返されます。 すべてのデータが取り出されると、両方の遠隔システムの接続は解放されます。 これらの接続は保護接続を使用しているので、切断することはできません。 接続が解放されると、コミットが出され、これらの接続は終了します。 ローカル・システムは接続されたままで、処理を続行します。

接続状況の検査

読み取り専用の接続が可能な環境で実行している場合、必ず接続の状況を検査してからコミット可能な更新を行う必要があります。これにより、作業単位がロールバックを必要とする状態に入らないようにします。次の COBOL の例は、接続状況の検査方法を示しています。

```
      ...
       EXEC SQL
        SET CONNECTION SYS5
       END-EXEC.
       ...
   * Check if the connection is updateable.
        EXEC SQL CONNECT END-EXEC.
   * If connection is updateable, update sales information otherwise
   * inform the user.
        IF SQLERRD(3) = 1 THEN
          EXEC SQL
            INSERT INTO SALES_TABLE
                VALUES(:SALES-DATA)
          END-EXEC
         ELSE
          DISPLAY 'Unable to update sales information at this time'.
   ...
```

図 13. 接続状況の検査の例

カーソルおよび準備されたステートメント

カーソルおよび準備されたステートメントは、コンパイル単位と接続に範囲が限られます。コンパイル単位に範囲を限るということは、個別にコンパイルした別のプログラムから呼び出したプログラムが呼び出しプログラムによりオープンまたは準備されたカーソルまたは準備されたステートメントを使用できないことを意味しま

す。接続に範囲を限るということは、プログラム内の各接続がカーソルまたは準備されたステートメントに関してその接続自身の個別のインスタンスを持てることを意味します。

次の分散作業単位の例は、同じカーソル名がどのように 2 つの異なる接続でオープンされ、カーソル C1 の 2 つのインスタンスになるかを示しています。

```
.....
EXEC SQL DECLARE C1 CURSOR FOR
           SELECT * FROM CORPDATA.EMPLOYEE;
/* Connect to local and open C1 */
EXEC SQL CONNECT TO LOCALSYS;
EXEC SQL OPEN C1;
/* Connect to the remote system and open C1 */
EXEC SQL CONNECT TO SYSA;
EXEC SQL OPEN C1;
/* Keep processing until done */
while (NOT_DONE) {
   /* Fetch a row of data from the local system */
   EXEC SQL SET CONNECTION LOCALSYS;
   EXEC SQL FETCH C1 INTO :local_emp_struct;
   /* Fetch a row of data from the remote system */
   EXEC SQL SET CONNECTION SYSA;
   EXEC SQL FETCH C1 INTO :rmt_emp_struct;
   /* Process the data */
   .....
}
/* Close the cursor on the remote system */
EXEC SQL CLOSE C1;
/* Close the cursor on the local system */
EXEC SQL SET CONNECTION LOCALSYS;
EXEC SQL CLOSE C1;
.....
```

図 14. DUW プログラムの中のカーソルの例

アプリケーション・リクエスター・ドライバー・プログラム

DRDA をインプリメントするプロダクトによって提供されるデータベース・アクセスを補足するために、DB2 UDB for iSeries は、DB2 UDB for iSeries アプリケーション・リクエスターに出口プログラムを作成するためのインターフェースを提供して、SQL 要求を処理します。 このような出口プログラムは、**アプリケーション・リクエスター・ドライバー**と呼ばれます。サーバーは、次の操作時に ARD プログラムを呼び出します。

- CRTSQLPKG コマンドまたは CRTSQLxxx コマンドを使用して行ったパッケージ作成時に、リレーショナル・データベース (RDB) パラメーターが ARD プログラムに対応する RDB 名に一致した場合。
- 現行接続が ARD プログラムに対応する RDB 名に対して行われているときの SQL ステートメントの処理時。

これらの呼び出しを使用すると、ARD プログラムは SQL ステートメントに関する情報を遠隔リレーショナル・データベースに渡して、結果をシステムに戻すことができます。 次にシステムはアプリケーションまたはユーザーに結果を戻します。 ARD プログラムがアクセスしたリレーショナル・データベースへのアクセスは、非

類似環境で DRDA アプリケーション・サーバーへのアクセスのようになります。ただし、DRDA の機能のすべてが ARD 環境でサポートされているわけではありません。サポートされていない機能の例に、ラージ・オブジェクト (LOB) および長いパスワード (passphrases) があります。

アプリケーション・リクエスター・ドライバー・プログラムに関する詳細については、OS/400 ファイル API を参照してください。

問題処理

分散データベース機能に関するエラー情報を収集し、報告するとき中心となる機能は、**第 1 障害データ検知 (FFDC)** と呼ばれます。FFDC サポートの目的は、OS/400 システムの DDM コンポーネントで検出されたエラーに関して正確な情報を提供し、その情報を基にして APAR (プログラム診断依頼書) を作成できるようにすることです。 この機能を使用すると、キー構造および DDM データ・ストリームが自動的にスプール・ファイルにダンプされます。エラー情報の最初の 1024 バイトは、システム・エラー・ログにも記録されます。 エラーが最初に見つかったときエラー情報が自動的にダンプされるので、ユーザーは障害を再現して、それを報告する必要はありません。 FFDC は、OS/400 DDM コンポーネントのアプリケーション・リクエスター側とアプリケーション・サーバー側の両方でアクティブです。ただし、FFDC データがログに記録されるためには、システム値 QSFWERRLOG が *LOG にセットされていなければなりません。

注: 負の SQLCODE はすべてがダンプされるとは限りません。APAR を作成するときに使用されるものだけがダンプされます。 分散リレーショナル・データベース操作で起こる問題の処理について詳しくは、*Distributed Database Problem Determination Guide* を参照してください。

SQL エラーが検出されると、対応する SQLSTATE が入った SQLCODE が SQLCA に戻されます。これらのコードについて詳しくは、iSeries Information Center の中の SQL メッセージおよびコードを参照してください。

DRDA ストアード・プロシージャーに関する考慮事項

iSeries DRDA サーバーは、ストアード・プロシージャーからの 1 つ以上の結果セットの戻りをサポートします。 ただし、V5R1 では、サーバーの使用可能化だけが提供されているので、この機能は、ストアード・プロシージャーの結果セットをサポートする iSeries 以外のクライアントからのみ使用できることに注意してください。

V5R2 においては、SQL で CLI インターフェースを使用するアプリケーションの場合に iSeries クライアント・サイド・サポートが追加されました。ただし、V5R1 iSeries サーバーがストアード・プロシージャー結果セットを V5R2 iSeries クライアントに戻せるようにする PTF を適用する必要があります。 PTF Cover Letter Database を参照して、リリース、日付、または修正番号ごとに分類されている、カバー・レターのリストを調べてください。

結果セットは、ストアード・プロシージャー内で、SQL SELECT ステートメントに関連した 1 つ以上の SQL カーソルをオープンすることによって生成することがで

きます。 さらに、最大 1 つの結果セットの配列も戻すことができます。 結果セットを戻すストアード・プロシージャーの作成について詳しくは、SQL 解説書の SET RESULT SETS ステートメント、CREATE PROCEDURE (SQL) ステートメント、および CREATE PROCEDURE (External) ステートメントの説明を参照してください。 DRDA でのストアード・プロシージャーの使用に関する一般情報については、分散データベース・プログラミングを参照してください。

付録 A. DB2 UDB for iSeries サンプル表

この章には、本書および SQL 解説書で参照または使用されているサンプル表が記載されています。表と一緒に、表を作成するための SQL ステートメントも記載されています。表作成の詳しい説明については、16 ページの 『表 (テーブル) の作成および使用』 を参照してください。

グループとして、表には、社員、部門、プロジェクト、および活動を記述する情報が入っています。この情報は、DB2 UDB Query Manager and SQL Development Kit ライセンス・プログラムの一部の機能を示すサンプル・アプリケーションを構成します。すべての例は、これらの表が CORPDATA (企業データを意味する) と名付けたスキーマに入っていると想定しています。

ストアード・プロシージャーはシステムの一部として出荷され、すべての表を作成する DDL ステートメントと、表にデータを入れる INSERT ステートメントが含まれています。プロシージャーは、プロシージャーへの呼び出しで指定されたスキーマを作成します。これは、SQL 外部ストアード・プロシージャーであるので、対話式 SQL および iSeries ナビゲーターを含む、すべての SQL インターフェースから呼び出すことができます。 作成したいスキーマが *SAMPLE* である場合、プロシージャーを起動するには、次のステートメントを出します。

```
CALL QSYS.CREATE_SQL_SAMPLE ('SAMPLE')
```

スキーマ名は、大文字で指定しなければなりません。スキーマは、すでに存在するものであってはなりません。

表には次のものがあります。

索引、別名、および視点は、これらの表のほとんどに作成されます。視点の定義は、ここには含まれていません。

最初のセットには関連しない以下の 3 つの表も作成されます。

注:

1. これらのサンプル表において、疑問符 (?) はヌル値を示しています。

部門表 (DEPARTMENT)

部門表には、社内の各部門が記述され、部門管理者および直属の上位部門が指定されます。部門表は、以下の CREATE TABLE ステートメントおよび ALTER TABLE ステートメントを使用して作成します。

```
CREATE TABLE DEPARTMENT
      (DEPTNO    CHAR(3)           NOT NULL,
       DEPTNAME  VARCHAR(36)       NOT NULL,
       MGRNO     CHAR(6)                    ,
       ADMRDEPT  CHAR(3)           NOT NULL
       LOCATION  CHAR(16),
       PRIMARY KEY (DEPTNO))

ALTER TABLE DEPARTMENT
      ADD FOREIGN KEY ROD (ADMRDEPT)
          REFERENCES DEPARTMENT
          ON DELETE CASCADE
```

以下の外部キーが、後で追加されます。

```
ALTER TABLE DEPARTMENT
      ADD FOREIGN KEY RDE (MGRNO)
          REFERENCES EMPLOYEE
          ON DELETE SET NULL
```

以下の索引が作成されます。

```
CREATE UNIQUE INDEX XDEPT1
       ON DEPARTMENT (DEPTNO)

CREATE INDEX XDEPT2
       ON DEPARTMENT (MGRNO)

CREATE INDEX XDEPT3
       ON DEPARTMENT (ADMRDEPT)
```

以下の別名が、表用に作成されます。

```
CREATE ALIAS DEPT FOR DEPARTMENT
```

次の表は列の内容を示しています。

表 41. 部門表の列

列名	説明
DEPTNO	部門番号または ID。
DEPTNAME	部門の全体的作業を表した名前。
MGRNO	部門管理者の社員番号 (EMPNO)。
ADMRDEPT	この部門の直属の上位管理部門 (DEPTNO)。最上位レベルの部門の上位管理部門はそれ自身です。
LOCATION	部門の場所。

部門の全リストについては、 387 ページの『部門』を参照してください。

部門

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
A00	SPIFFY コンピューター・サービス事業部	000010	A00	?
B01	計画	000020	A00	?
C01	情報センター	000030	A00	?
D01	開発センター	?	A00	?
D11	製造システム	000060	D01	?
D21	管理システム	000070	D01	?
E01	サポート・サービス	000050	A00	?
E11	業務部	000090	E01	?
E21	ソフトウェア・サポート	000100	E01	?
F22	事業所 F2	?	E01	?
G22	事業所 G2	?	E01	?
H22	事業所 H2	?	E01	?
I22	事業所 I2	?	E01	?
J22	事業所 J2	?	E01	?

社員表 (EMPLOYEE)

社員表には、全社員が社員番号で識別され、基本的な個人情報が記述されています。社員表は、以下の CREATE TABLE ステートメントおよび ALTER TABLE ステートメントを使用して作成します。

```
CREATE TABLE EMPLOYEE
      (EMPNO       CHAR(6)          NOT NULL,
       FIRSTNME    VARCHAR(12)      NOT NULL,
       MIDINIT     CHAR(1)          NOT NULL,
       LASTNAME    VARCHAR(15)      NOT NULL,
       WORKDEPT    CHAR(3)                  ,
       PHONENO     CHAR(4)                  ,
       HIREDATE    DATE                     ,
       JOB         CHAR(8)                  ,
       EDLEVEL     SMALLINT         NOT NULL,
       SEX         CHAR(1)                  ,
       BIRTHDATE   DATE                     ,
       SALARY      DECIMAL(9,2)             ,
       BONUS       DECIMAL(9,2)             ,
       COMM        DECIMAL(9,2)
       PRIMARY KEY (EMPNO))

ALTER TABLE EMPLOYEE
      ADD FOREIGN KEY RED (WORKDEPT)
      REFERENCES DEPARTMENT
      ON DELETE SET NULL

ALTER TABLE EMPLOYEE
      ADD CONSTRAINT NUMBER
      CHECK (PHONENO >= '0000' AND PHONENO <= '9999')
```

以下の索引が作成されます。

```
CREATE UNIQUE INDEX XEMP1
       ON EMPLOYEE (EMPNO)

CREATE INDEX XEMP2
       ON EMPLOYEE (WORKDEPT)
```

以下の別名が、表用に作成されます。

```
CREATE ALIAS EMP FOR EMPLOYEE
```

次の表は、列の内容を示しています。

列名	説明
EMPNO	社員番号
FIRSTNME	社員の名
MIDINIT	社員のミドルネームの頭文字
LASTNAME	社員の姓
WORKDEPT	社員が所属している部門の ID
PHONENO	社員の電話番号
HIREDATE	雇用年月日
JOB	社員の職種
EDLEVEL	学歴年数
SEX	社員の性別 (M または F)
BIRTHDATE	生年月日
SALARY	給与
BONUS	賞与
COMM	年俸

社員の全リストについては、 389 ページの『社員』を参照してください。

社員

EMP NO	FIRST NAME	MID INIT	LASTNAME	WORK DEPT	PHONE NO	HIRE DATE	JOB	ED LEVEL	SEX	BIRTH DATE	SAL-ARY	BONUS	COMM
000010	CHRISTINE	I	HAAS	A00	3978	1965-01-01	PRES	18	F	1933-08-24	52750	1000	4220
000020	MICHAEL	L	THOMPSON	B01	3476	1973-10-10	MANAGER	18	M	1948-02-02	41250	800	3300
000030	SALLY	A	KWAN	C01	4738	1975-04-05	MANAGER	20	F	1941-05-11	38250	800	3060
000050	JOHN	B	GEYER	E01	6789	1949-08-17	MANAGER	16	M	1925-09-15	40175	800	3214
000060	IRVING	F	STERN	D11	6423	1973-09-14	MANAGER	16	M	1945-07-07	32250	500	2580
000070	EVA	D	PULASKI	D21	7831	1980-09-30	MANAGER	16	F	1953-05-26	36170	700	2893
000090	EILEEN	W	HENDERSON	E11	5498	1970-08-15	MANAGER	16	F	1941-05-15	29750	600	2380
000100	THEODORE	Q	SPENSER	E21	0972	1980-06-19	MANAGER	14	M	1956-12-18	26150	500	2092
000110	VINCENZO	G	LUCCHESSI	A00	3490	1958-05-16	SALESREP	19	M	1929-11-05	46500	900	3720
000120	SEAN		O'CONNELL	A00	2167	1963-12-05	CLERK	14	M	1942-10-18	29250	600	2340
000130	DOLORES	M	QUINTANA	C01	4578	1971-07-28	ANALYST	16	F	1925-09-15	23800	500	1904
000140	HEATHER	A	NICHOLLS	C01	1793	1976-12-15	ANALYST	18	F	1946-01-19	28420	600	2274
000150	BRUCE		ADAMSON	D11	4510	1972-02-12	DESIGNER	16	M	1947-05-17	25280	500	2022
000160	ELIZABETH	R	PIANKA	D11	3782	1977-10-11	DESIGNER	17	F	1955-04-12	22250	400	1780
000170	MASATOSHI	J	YOSHIMURA	D11	2890	1978-09-15	DESIGNER	16	M	1951-01-05	24680	500	1974
000180	MARILYN	S	SCOUTTEN	D11	1682	1973-07-07	DESIGNER	17	F	1949-02-21	21340	500	1707
000190	JAMES	H	WALKER	D11	2986	1974-07-26	DESIGNER	16	M	1952-06-25	20450	400	1636
000200	DAVID		BROWN	D11	4501	1966-03-03	DESIGNER	16	M	1941-05-29	27740	600	2217
000210	WILLIAM	T	JONES	D11	0942	1979-04-11	DESIGNER	17	M	1953-02-23	18270	400	1462
000220	JENNIFER	K	LUTZ	D11	0672	1968-08-29	DESIGNER	18	F	1948-03-19	29840	600	2387
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21	CLERK	14	M	1935-05-30	22180	400	1774
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05	CLERK	17	M	1954-03-31	28760	600	2301
000250	DANIEL	S	SMITH	D21	0961	1969-10-30	CLERK	15	M	1939-11-12	19180	400	1534
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11	CLERK	16	F	1936-10-05	17250	300	1380
000270	MARIA	L	PEREZ	D21	9001	1980-09-30	CLERK	15	F	1953-05-26	27380	500	2190
000280	ETHEL	R	SCHNEIDER	E11	8997	1967-03-24	OPERATOR	17	F	1936-03-28	26250	500	2100
000290	JOHN	R	PARKER	E11	4502	1980-05-30	OPERATOR	12	M	1946-07-09	15340	300	1227
000300	PHILIP	X	SMITH	E11	2095	1972-06-19	OPERATOR	14	M	1936-10-27	17750	400	1420
000310	MAUDE	F	SETRIGHT	E11	3332	1964-09-12	OPERATOR	12	F	1931-04-21	15900	300	1272
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07	FILEREP	16	M	1932-08-11	19950	400	1596
000330	WING		LEE	E21	2103	1976-02-23	FILEREP	14	M	1941-07-18	25370	500	2030
000340	JASON	R	GOUNOT	E21	5698	1947-05-05	FILEREP	16	M	1926-05-17	23840	500	1907
200010	DIAN	J	HEMMINGER	A00	3978	1965-01-01	SALESREP	18	F	1933-08-14	46500	1000	4220
200120	GREG		ORLANDO	A00	2167	1972-05-05	CLERK	14	M	1942-10-18	29250	600	2340
200140	KIM	N	NATZ	C01	1793	1976-12-15	ANALYST	18	F	1946-01-19	28420	600	2274
200170	KIYOSHI		YAMAMOTO	D11	2890	1978-09-15	DESIGNER	16	M	1951-01-05	24680	500	1974
200220	REBA	K	JOHN	D11	0672	1968-08-29	DESIGNER	18	F	1948-03-19	29840	600	2387
200240	ROBERT	M	MONTEVERDE	D21	3780	1979-12-05	CLERK	17	M	1954-03-31	28760	600	2301
200280	EILEEN	R	SCHWARTZ	E11	8997	1967-03-24	OPERATOR	17	F	1936-03-28	26250	500	2100
200310	MICHELLE	F	SPRINGER	E11	3332	1964-09-12	OPERATOR	12	F	1931-04-21	15900	300	1272
200330	HELENA		WONG	E21	2103	1976-02-23	FIELDREP	14	F	1941-07-18	25370	500	2030
200340	ROY	R	ALONZO	E21	5698	1947-05-05	FIELDREP	16	M	1926-05-17	23840	500	1907

社員の写真表 (EMP_PHOTO)

社員の写真表には、社員番号別に保管された、社員の写真が入っています。社員の写真表は、以下の CREATE TABLE ステートメントおよび ALTER TABLE ステートメントを使用して作成します。

```
CREATE TABLE EMP_PHOTO
            (EMPNO CHAR(6) NOT NULL,
             PHOTO_FORMAT VARCHAR(10) NOT NULL,
             PICTURE BLOB(100K),
             EMP_ROWID CHAR(40) NOT NULL DEFAULT '',
             PRIMARY KEY (EMPNO,PHOTO_FORMAT))

ALTER TABLE EMP_PHOTO
           ADD COLUMN DL_PICTURE DATALINK(1000)
              LINKTYPE URL NO LINK CONTROL

ALTER TABLE EMP_PHOTO
             ADD FOREIGN KEY (EMPNO)
             REFERENCES EMPLOYEE
             ON DELETE RESTRICT
```

以下の索引が作成されます。

```
CREATE UNIQUE INDEX XEMP_PHOTO
             ON EMP_PHOTO (EMPNO,PHOTO_FORMAT)
```

次の表は、列の内容を示しています。

列名	説明
EMPNO	社員番号
PHOTO_FORMAT	PICTURE に保管されたイメージのフォーマット。
PICTURE	写真のイメージ。
EMP_ROWID	固有の行 ID (現在は未使用)。

EMP_PHOTO の全リストについては、『EMP_PHOTO』を参照してください。

EMP_PHOTO

EMPNO	PHOTO_FORMAT	PICTURE	EMP_ROWID
000130	ビットマップ	?	
000130	GIF	?	
000140	ビットマップ	?	
000140	GIF	?	
000150	ビットマップ	?	
000150	GIF	?	
000190	ビットマップ	?	
000190	GIF	?	

社員の履歴表 (EMP_RESUME)

社員の写真表には、社員番号別に保管された、社員の履歴が入っています。社員の履歴表は、以下の CREATE TABLE ステートメントおよび ALTER TABLE ステートメントを使用して作成します。

```
CREATE TABLE EMP_RESUME
            (EMPNO CHAR(6) NOT NULL,
             RESUME_FORMAT VARCHAR(10) NOT NULL,
             RESUME CLOB(5K),
             EMP_ROWID CHAR(40) NOT NULL DEFAULT '',
             PRIMARY KEY (EMPNO,RESUME_FORMAT))

ALTER TABLE EMP_RESUME
           ADD COLUMN DL_RESUME DATALINK(1000)
               LINKTYPE URL NO LINK CONTROL

ALTER TABLE EMP_RESUME
             ADD FOREIGN KEY (EMPNO)
             REFERENCES EMPLOYEE
             ON DELETE RESTRICT
```

以下の索引が作成されます。

```
CREATE UNIQUE INDEX XEMP_RESUME
             ON EMP_RESUME (EMPNO,RESUME_FORMAT)
```

次の表は、列の内容を示しています。

列名	説明
EMPNO	社員番号
RESUME_FORMAT	RESUME に保管されているテキストのフォーマット。
RESUME	履歴
EMP_ROWID	固有の行 ID (現在は未使用)。

EMP_RESUME の全リストについては、『EMP_RESUME』を参照してください。

EMP_RESUME

EMPNO	RESUME_FORMAT	RESUME	EMP_ROWID
000130	ASCII	?	
000130	HTML	?	
000140	ASCII	?	
000140	HTML	?	
000150	ASCII	?	
000150	HTML	?	
000190	ASCII	?	
000190	HTML	?	

社員プロジェクト活動表 (EMPPROJACT)

社員プロジェクト活動表には、各プロジェクト別にリストされた各作業を担当する社員が示されます。各社員がプロジェクトに参加する程度 (専任か兼任か) と活動スケジュールも表に示されます。社員プロジェクト活動表は、以下の CREATE TABLE ステートメントおよび ALTER TABLE ステートメントを使用して作成します。

```
CREATE TABLE EMPPROJACT
      (EMPNO     CHAR(6)          NOT NULL,
       PROJNO    CHAR(6)          NOT NULL,
       ACTNO     SMALLINT         NOT NULL,
       EMPTIME   DECIMAL(5,2)             ,
       EMSTDATE  DATE                     ,
       EMENDATE  DATE                     )

ALTER TABLE EMPPROJACT
             ADD FOREIGN KEY REPAPA (PROJNO, ACTNO, EMSTDATE)
             REFERENCES PROJACT
             ON DELETE RESTRICT
```

以下の別名が、表用に作成されます。

```
CREATE ALIAS EMPACT FOR EMPPROJACT

CREATE ALIAS EMP_ACT FOR EMPPROJACT
```

次の表は、列の内容を示しています。

表 42. 社員プロジェクト活動表の列

列名	説明
EMPNO	社員の ID 番号
PROJNO	社員が担当するプロジェクトのプロジェクト番号
ACTNO	プロジェクトにおいて社員が担当する作業の ID
EMPTIME	社員の全作業時間に占める EMSTDATE から EMENDATE までのプロジェクト参加時間の比率 (0.00 から 1.00 まで)
EMSTDATE	作業の開始日付
EMENDATE	作業の完了日付

EMPPROJACT の全リストについては、『EMPPROJACT』を参照してください。

EMPPROJACT

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
000010	AD3100	10	.50	1982-01-01	1982-07-01
000070	AD3110	10	1.00	1982-01-01	1983-02-01
000230	AD3111	60	1.00	1982-01-01	1982-03-15
000230	AD3111	60	.50	1982-03-15	1982-04-15
000230	AD3111	70	.50	1982-03-15	1982-10-15
000230	AD3111	80	.50	1982-04-15	1982-10-15
000230	AD3111	180	.50	1982-10-15	1983-01-01
000240	AD3111	70	1.00	1982-02-15	1982-09-15
000240	AD3111	80	1.00	1982-09-15	1983-01-01
000250	AD3112	60	1.00	1982-01-01	1982-02-01
000250	AD3112	60	.50	1982-02-01	1982-03-15
000250	AD3112	60	1.00	1983-01-01	1983-02-01
000250	AD3112	70	.50	1982-02-01	1982-03-15
000250	AD3112	70	1.00	1982-03-15	1982-08-15
000250	AD3112	70	.25	1982-08-15	1982-10-15
000250	AD3112	80	.25	1982-08-15	1982-10-15
000250	AD3112	80	.50	1982-10-15	1982-12-01
000250	AD3112	180	.50	1982-08-15	1983-01-01
000260	AD3113	70	.50	1982-06-15	1982-07-01
000260	AD3113	70	1.00	1982-07-01	1983-02-01
000260	AD3113	80	1.00	1982-01-01	1982-03-01
000260	AD3113	80	.50	1982-03-01	1982-04-15
000260	AD3113	180	.50	1982-03-01	1982-04-15
000260	AD3113	180	1.00	1982-04-15	1982-06-01
000260	AD3113	180	1.00	1982-06-01	1982-07-01
000270	AD3113	60	.50	1982-03-01	1982-04-01

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
000270	AD3113	60	1.00	1982-04-01	1982-09-01
000270	AD3113	60	.25	1982-09-01	1982-10-15
000270	AD3113	70	.75	1982-09-01	1982-10-15
000270	AD3113	70	1.00	1982-10-15	1983-02-01
000270	AD3113	80	1.00	1982-01-01	1982-03-01
000270	AD3113	80	.50	1982-03-01	1982-04-01
000030	IF1000	10	.50	1982-06-01	1983-01-01
000130	IF1000	90	1.00	1982-10-01	1983-01-01
000130	IF1000	100	.50	1982-10-01	1983-01-01
000140	IF1000	90	.50	1982-10-01	1983-01-01
000030	IF2000	10	.50	1982-01-01	1983-01-01
000140	IF2000	100	1.00	1982-01-01	1982-03-01
000140	IF2000	100	.50	1982-03-01	1982-07-01
000140	IF2000	110	.50	1982-03-01	1982-07-01
000140	IF2000	110	.50	1982-10-01	1983-01-01
000010	MA2100	10	.50	1982-01-01	1982-11-01
000110	MA2100	20	1.00	1982-01-01	1983-03-01
000010	MA2110	10	1.00	1982-01-01	1983-02-01
000200	MA2111	50	1.00	1982-01-01	1982-06-15
000200	MA2111	60	1.00	1982-06-15	1983-02-01
000220	MA2111	40	1.00	1982-01-01	1983-02-01
000150	MA2112	60	1.00	1982-01-01	1982-07-15
000150	MA2112	180	1.00	1982-07-15	1983-02-01
000170	MA2112	60	1.00	1982-01-01	1983-06-01
000170	MA2112	70	1.00	1982-06-01	1983-02-01
000190	MA2112	70	1.00	1982-01-01	1982-10-01
000190	MA2112	80	1.00	1982-10-01	1983-10-01
000160	MA2113	60	1.00	1982-07-15	1983-02-01
000170	MA2113	80	1.00	1982-01-01	1983-02-01
000180	MA2113	70	1.00	1982-04-01	1982-06-15
000210	MA2113	80	.50	1982-10-01	1983-02-01
000210	MA2113	180	.50	1982-10-01	1983-02-01
000050	OP1000	10	.25	1982-01-01	1983-02-01
000090	OP1010	10	1.00	1982-01-01	1983-02-01
000280	OP1010	130	1.00	1982-01-01	1983-02-01
000290	OP1010	130	1.00	1982-01-01	1983-02-01
000300	OP1010	130	1.00	1982-01-01	1983-02-01
000310	OP1010	130	1.00	1982-01-01	1983-02-01
000050	OP2010	10	.75	1982-01-01	1983-02-01
000100	OP2010	10	1.00	1982-01-01	1983-02-01
000320	OP2011	140	.75	1982-01-01	1983-02-01

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
000320	OP2011	150	.25	1982-01-01	1983-02-01
000330	OP2012	140	.25	1982-01-01	1983-02-01
000330	OP2012	160	.75	1982-01-01	1983-02-01
000340	OP2013	140	.50	1982-01-01	1983-02-01
000340	OP2013	170	.50	1982-01-01	1983-02-01
000020	PL2100	30	1.00	1982-01-01	1982-09-15

プロジェクト表 (PROJECT)

プロジェクト表には、社内で現在進行中の各プロジェクトが記述されます。各行に記述されるデータには、プロジェクト番号、名前、担当者、およびスケジュール日付があります。プロジェクト表は、以下の CREATE TABLE ステートメントおよび ALTER TABLE ステートメントを使用して作成します。

```
CREATE TABLE PROJECT
      (PROJNO    CHAR(6)         NOT NULL,
       PROJNAME  VARCHAR(24)     NOT NULL DEFAULT,
       DEPTNO    CHAR(3)         NOT NULL,
       RESPEMP   CHAR(6)         NOT NULL,
       PRSTAFF   DECIMAL(5,2)            ,
       PRSTDATE  DATE                    ,
       PRENDATE  DATE                    ,
       MAJPROJ   CHAR(6)                 ,
       PRIMARY KEY (PROJNO))

ALTER TABLE PROJECT
              ADD FOREIGN KEY (DEPTNO)
              REFERENCES DEPARTMENT
              ON DELETE RESTRICT

ALTER TABLE PROJECT
              ADD FOREIGN KEY (RESPEMP)
              REFERENCES EMPLOYEE
              ON DELETE RESTRICT

ALTER TABLE PROJECT
              ADD FOREIGN KEY  RPP (MAJPROJ)
              REFERENCES PROJECT
              ON DELETE CASCADE
```

以下の索引が作成されます。

```
CREATE UNIQUE INDEX XPROJ1
              ON PROJECT (PROJNO)

CREATE INDEX XPROJ2
              ON PROJECT (RESPEMP)
```

以下の別名が、表用に作成されます。

```
CREATE ALIAS PROJ FOR PROJECT
```

次の表は列の内容を示しています。

列名	説明
PROJNO	プロジェクト番号
PROJNAME	プロジェクト名

列名	説明
DEPTNO	プロジェクト担当部門の部門番号
RESPEMP	プロジェクト担当者の社員番号
PRSTAFF	平均の予定要員数
PRSTDATE	プロジェクトの開始予定日
PRENDATE	プロジェクトの終了予定日
MAJPROJ	サブプロジェクトの管理プロジェクト番号

PROJECT の全リストについては、『PROJECT』を参照してください。

PROJECT

PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
AD3100	統括管理	D01	000010	6.5	1982-01-01	1983-02-01	?
AD3110	総務	D21	000070	6	1982-01-01	1983-02-01	AD3100
AD3111	給与プログラミング	D21	000230	2	1982-01-01	1983-02-01	AD3110
AD3112	人事プログラミング	D21	000250	1	1982-01-01	1983-02-01	AD3110
AD3113	経理プログラミング	D21	000270	2	1982-01-01	1983-02-01	AD3110
IF1000	照会サービス	C01	000030	2	1982-01-01	1983-02-01	?
IF2000	ユーザー教育	C01	000030	1	1982-01-01	1983-02-01	?
MA2100	溶接ライン (WL) 自動化	D01	000010	12	1982-01-01	1983-02-01	?
MA2110	W L プログラミング	D11	000060	9	1982-01-01	1983-02-01	MA2100
MA2111	W L プログラム設計	D11	000220	2	1982-01-01	1982-12-01	MA2110
MA2112	W L ロボット設計	D11	000150	3	1982-01-01	1982-12-01	MA2110
MA2113	W L 製造制御プログラム	D11	000160	3	1982-02-15	1982-12-01	MA2110
OP1000	オペレーション・サポート	E01	000050	6	1982-01-01	1983-02-01	?
OP1010	オペレーション	E11	000090	5	1982-01-01	1983-02-01	OP1000
OP2000	汎用システム・サービス	E01	000050	5	1982-01-01	1983-02-01	?
OP2010	システム・サポート	E21	000100	4	1982-01-01	1983-02-01	OP2000
OP2011	SCP システム・サポート	E21	000320	1	1982-01-01	1983-02-01	OP2010
OP2012	アプリケーション・サポート	E21	000330	1	1982-01-01	1983-02-01	OP2010
OP2013	DB/DC サポート	E21	000340	1	1982-01-01	1983-02-01	OP2010

PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
PL2100	溶接ライン企画部	B01	000020	1	1982-01-01	1982-09-15	MA2100

プロジェクト活動表 (PROJACT)

プロジェクト活動表には、社内で現在進行中の各プロジェクトが記述されます。各行のデータには、プロジェクト番号、活動番号、およびスケジュール日付があります。 プロジェクト活動表は、以下の CREATE TABLE ステートメントおよび ALTER TABLE ステートメントを使用して作成します。

```
CREATE TABLE PROJACT
            (PROJNO CHAR(6) NOT NULL,
             ACTNO SMALLINT NOT NULL,
             ACSTAFF DECIMAL(5,2),
             ACSTDATE DATE NOT NULL,
             ACENDATE DATE ,
             PRIMARY KEY (PROJNO, ACTNO, ACSTDATE))

ALTER TABLE PROJACT
             ADD FOREIGN KEY RPAP (PROJNO)
             REFERENCES PROJECT
             ON DELETE RESTRICT
```

以下の外部キーが、後で追加されます。

```
ALTER TABLE PROJACT
           ADD FOREIGN KEY RPAA (ACTNO)
               REFERENCES ACT
               ON DELETE RESTRICT
```

以下の索引が作成されます。

```
CREATE UNIQUE INDEX XPROJAC1
             ON PROJACT (PROJNO, ACTNO, ACSTDATE)
```

次の表は列の内容を示しています。

列名	説明
PROJNO	プロジェクト番号
ACTNO	活動番号
ACSTAFF	平均の予定要員数
ACSTDATE	活動開始日
ACENDATE	活動終了日

PROJACT の全リストについては、『PROJACT』を参照してください。

PROJACT

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
AD3100	10	?	1982-01-01	?
AD3110	10	?	1982-01-01	?
AD3111	60	?	1982-01-01	?
AD3111	60	?	1982-03-15	?

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
AD3111	70	?	1982-03-15	?
AD3111	80	?	1982-04-15	?
AD3111	180	?	1982-10-15	?
AD3111	70	?	1982-02-15	?
AD3111	80	?	1982-09-15	?
AD3112	60	?	1982-01-01	?
AD3112	60	?	1982-02-01	?
AD3112	60	?	1983-01-01	?
AD3112	70	?	1982-02-01	?
AD3112	70	?	1982-03-15	?
AD3112	70	?	1982-08-15	?
AD3112	80	?	1982-08-15	?
AD3112	80	?	1982-10-15	?
AD3112	180	?	1982-08-15	?
AD3113	70	?	1982-06-15	?
AD3113	70	?	1982-07-01	?
AD3113	80	?	1982-01-01	?
AD3113	80	?	1982-03-01	?
AD3113	180	?	1982-03-01	?
AD3113	180	?	1982-04-15	?
AD3113	180	?	1982-06-01	?
AD3113	60	?	1982-03-01	?
AD3113	60	?	1982-04-01	?
AD3113	60	?	1982-09-01	?
AD3113	70	?	1982-09-01	?
AD3113	70	?	1982-10-15	?
IF1000	10	?	1982-06-01	?
IF1000	90	?	1982-10-01	?
IF1000	100	?	1982-10-01	?
IF2000	10	?	1982-01-01	?
IF2000	100	?	1982-01-01	?
IF2000	100	?	1982-03-01	?
IF2000	110	?	1982-03-01	?
IF2000	110	?	1982-10-01	?
MA2100	10	?	1982-01-01	?
MA2100	20	?	1982-01-01	?
MA2110	10	?	1982-01-01	?
MA2111	50	?	1982-01-01	?
MA2111	60	?	1982-06-15	?
MA2111	40	?	1982-01-01	?
MA2112	60	?	1982-01-01	?

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
MA2112	180	?	1982-07-15	?
MA2112	70	?	1982-06-01	?
MA2112	70	?	1982-01-01	?
MA2112	80	?	1982-10-01	?
MA2113	60	?	1982-07-15	?
MA2113	80	?	1982-01-01	?
MA2113	70	?	1982-04-01	?
MA2113	80	?	1982-10-01	?
MA2113	180	?	1982-10-01	?
OP1000	10	?	1982-01-01	?
OP1010	10	?	1982-01-01	?
OP1010	130	?	1982-01-01	?
OP2010	10	?	1982-01-01	?
OP2011	140	?	1982-01-01	?
OP2011	150	?	1982-01-01	?
OP2012	140	?	1982-01-01	?
OP2012	160	?	1982-01-01	?
OP2013	140	?	1982-01-01	?
OP2013	170	?	1982-01-01	?
PL2100	30	?	1982-01-01	?

活動表 (ACT)

活動表では、各活動を記述します。活動表は、以下の CREATE TABLE ステートメントを使用して作成します。

```
CREATE TABLE ACT
            (ACTNO SMALLINT NOT NULL,
             ACTKWD CHAR(6) NOT NULL,
             ACTDESC VARCHAR(20) NOT NULL,
             PRIMARY KEY (ACTNO))
```

以下の索引が作成されます。

```
CREATE UNIQUE INDEX XACT1
             ON ACT (ACTNO)

CREATE UNIQUE INDEX XACT2
             ON ACT (ACTKWD)
```

次の表は、列の内容を示しています。

列名	説明
ACTNO	活動番号
ACTKWD	活動のキーワード
ACTDESC	活動の記述

ACT の全リストについては、『ACT』を参照してください。

ACT

ACTNO	ACTKWD	ACTDESC
10	MANAGE	MANAGE/ADVISE (管理/指導)
20	ECOST	ESTIMATE COST (コスト見積もり)
30	DEFINE	DEFINE SPECS (スペック定義)
40	LEADPR	LEAD PROGRAM/DESIGN (リード・プログラム/設計)
50	SPECS	WRITE SPECS (スペック作成)
60	LOGIC	DESCRIBE LOGIC (ロジックの記述)
70	CODE	CODE PROGRAMS (プログラムのコーディング)
80	TEST	TEST PROGRAMS (プログラムのテスト)
90	ADMQS	ADM QUERY SYSTEM (管理照会システム)
100	TEACH	TEACH CLASSES (クラスでの教育)
110	COURSE	DEVELOP COURSES (教育コースの作成)
120	STAFF	PERS AND STAFFING (人事とリクルート)
130	OPERAT	OPER COMPUTER SYS (コンピューター・システムの運用)
140	MAINT	MAINT SOFTWARE SYS (ソフトウェア・システムの保守)
150	ADMSYS	ADM OPERATING SYS (オペレーティング・システムの管理)
160	ADMDB	ADM DATA BASES (データベースの管理)
170	ADMDC	ADM DATA COMM (データ通信の管理)
180	DOC	DOCUMENT (文書/資料)

クラス・スケジュール表 (CL_SCHED)

クラス・スケジュール表には、クラス、クラスの開始時刻、クラスの終了時刻、およびクラス・コードが記述されます。クラス・スケジュール表は次の CREATE TABLE ステートメントを使用して作成します。

```
CREATE TABLE CL_SCHED
      (CLASS_CODE          CHAR(7),
       "DAY"               SMALLINT,
       STARTING            TIME,
       ENDING              TIME)
```

次の表は列の内容を示しています。

列名	説明
CLASS_CODE	クラス・コード (教室 : 講師)
DAY	4 日スケジュールの日数
STARTING	クラス開始時刻

列名	説明
ENDING	クラス終了時刻

CL_SCHED の全リストについては、『CL_SCHED』を参照してください。

CL_SCHED

CLASS_CODE	DAY	STARTING	ENDING
042:BF	4	12:10:00	14:00:00
553:MJA	1	10:30:00	11:00:00
543:CWM	3	09:10:00	10:30:00
778:RES	2	12:10:00	14:00:00
044:HD	3	17:12:30	18:00:00

未処理表 (IN_TRAY)

未処理表には、電子到着バスケットが記述されます。このバスケットにはメッセージが受信されたときからのタイム・スタンプ、メッセージの発信人のユーザー ID、およびメッセージそのものが入っています。未処理表は次の CREATE TABLE ステートメントを使用して作成します。

```
CREATE TABLE IN_TRAY
    (RECEIVED           TIMESTAMP,
     SOURCE             CHAR(8),
     SUBJECT            CHAR(64),
     NOTE_TEXT          VARCHAR(3000))
```

次の表は列の内容を示しています。

列名	説明
RECEIVED	受信した日付と時刻
SOURCE	ノート送り出し人のユーザー ID
SUBJECT	ノートの簡単な記述
NOTE_TEXT	ノート

IN_TRAY の全リストについては、 401 ページの『IN_TRAY』を参照してください。

IN_TRAY

RECEIVED	SOURCE	SUBJECT	NOTE_TEXT
1988-12-25-17.12.30.000000	BADAMSON	FWD: 良い年だ! 第4半期ボーナス。	To: JWALKER Cc: QUINTANA, NICHOLLS Jim、努力は報われた。冷蔵庫にビールがあるから、ちょっと飲みに来ないか？ Delores も Heather も来ないか？ Bruce <Forwarding from ISTERN> Subject: FWD: 良い年だ! 第4半期ボーナス。 To: Dept_D11 おめでとう。よくやった。ボーナスでエンジョイしてくれ給え。 Irv <Forwarding from CHAAS> Subject: 良い年だ! 第4半期ボーナス。 To: All_Managers 第4半期の結果が出た。 チームで皆頑張って、計画値を超えた！ 今年のボーナスは 18% だと発表できてうれしい。 良い休暇を！ Christine Haas
1988-12-23-08.53.58.000000	ISTERN	FWD: 良い年だ! 第4半期ボーナス。	To: Dept_D11 おめでとう。よくやった。 ボーナスでエンジョイしてくれ給え。 Irv <Forwarding from CHAAS> Subject: 良い年だ! 第4半期ボーナス。 To: All_Managers 第4半期の結果が出た。 チームで皆頑張って、計画値を超えた！ 今年のボーナスは 18% だと発表できてうれしい。 良い休暇を！ Christine Haas
1988-12-22-14.07.21.136421	CHAAS	良い年だ! 第4半期ボーナス。	To: All_Managers 第4半期の結果が出た。 チームで皆頑張って、計画値を超えた！ 今年のボーナスは 18% だと発表できてうれしい。 良い休暇を！ Christine Haas

組織表 (ORG)

組織表は、企業の組織を記述します。組織表は、以下の CREATE TABLE ステートメントを使用して作成します。

```
CREATE TABLE ORG
            (DEPTNUMB SMALLINT NOT NULL,
             DEPTNAME VARCHAR(14),
             MANAGER SMALLINT,
             DIVISION VARCHAR(10),
             LOCATION VARCHAR(13))
```

次の表は列の内容を示しています。

列名	説明
DEPTNUMB	部門番号
DEPTNAME	部門名
MANAGER	部門の管理者番号
DIVISION	部門内の部
LOCATION	部門の場所

ORG の全リストについては、『ORG』を参照してください。

ORG

DEPTNUMB	DEPTNAME	MANAGER	DIVISION	LOCATION
10	Head Office	160	Corporate	New York
15	New England	50	Eastern	Boston
20	Mid Atlantic	10	Eastern	Washington
38	South Atlantic	30	Eastern	Atlanta
42	Great Lakes	100	Midwest	Chicago
51	Plains	140	Midwest	Dallas
66	Pacific	270	Western	San Francisco
84	Mountain	290	Western	Denver

スタッフ表 (STAFF)

スタッフ表では、社員を記述します。スタッフ表は、以下の CREATE TABLE ステートメントを使用して作成します。

```
CREATE TABLE STAFF
            (ID  SMALLINT NOT NULL,
             NAME VARCHAR(9),
             DEPT SMALLINT,
             JOB CHAR(5),
             YEARS SMALLINT,
             SALARY DECIMAL(7,2),
             COMM DECIMAL(7,2))
```

次の表は、列の内容を示しています。

列名	説明
ID	社員番号
NAME	社員の名前
DEPT	部門番号
JOB	役職
YEARS	勤続年数
SALARY	社員のサラリー (年額)
COMM	社員のコミッション

STAFF の全リストについては、『STAFF』を参照してください。

STAFF

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	?
20	Pernal	20	Sales	8	18171.25	612.45
30	Marenghi	38	Mgr	5	17506.75	?
40	O'Brien	38	Sales	6	18006.00	846.55
50	Hanes	15	Mgr	10	20659.80	?
60	Quigley	38	Sales	7	16508.30	650.25
70	Rothman	15	Sales	7	16502.83	1152.00
80	James	20	Clerk	?	13504.60	128.20
90	Koonitz	42	Sales	6	18001.75	1386.70
100	Plotz	42	Mgr	7	18352.80	?
110	Ngan	15	Clerk	5	12508.20	206.60
120	Naughton	38	Clerk	?	12954.75	180.00
130	Yamaguchi	42	Clerk	6	10505.90	75.60
140	Fraye	51	Mgr	6	21150.00	?
150	Williams	51	Sales	6	19456.50	637.65
160	Molinare	10	Mgr	7	22959.20	?
170	Kermisch	15	Clerk	4	12258.50	110.10
180	Abrahams	38	Clerk	3	12009.75	236.50
190	Sneider	20	Clerk	8	14252.75	126.50
200	Scoutten	42	Clerk	?	11508.60	84.20
210	Lu	10	Mgr	10	20010.00	?
220	Smith	51	Sales	7	17654.50	992.80
230	Lundquist	51	Clerk	3	13369.80	189.65
240	Daniels	10	Mgr	5	19260.25	?
250	Wheeler	51	Clerk	6	14460.00	513.30
260	Jones	10	Mgr	12	21234.00	?
270	Lea	66	Mgr	9	18555.50	?

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
280	Wilson	66	Sales	9	18674.50	811.50
290	Quill	84	Mgr	10	19818.00	?
300	Davis	84	Sales	5	15454.50	806.10
310	Graham	66	Sales	13	21000.00	200.30
320	Gonzales	66	Sales	4	16858.20	844.00
330	Burke	66	Clerk	1	10988.00	55.50
340	Edwards	84	Sales	7	17844.00	1285.00
3650	Gafney	84	Clerk	5	13030.50	188.00

販売表 (SALES)

販売表では、販売員の販売高を記述します。 販売表は、以下の CREATE TABLE ステートメントを使用して作成します。

```
CREATE TABLE SALES
            (SALES_DATE DATE,
             SALES_PERSON VARCHAR(15),
             REGION VARCHAR(15),
             SALES INTEGER)
```

次の表は列の内容を示しています。

列名	説明
SALES_DATE	販売が行われた日付
SALES_PERSON	販売を行った人
REGION	販売が行われた地域
SALES	販売数

SALES の全リストについては、『SALES』を参照してください。

SALES

SALES_DATE	SALES_PERSON	REGION	SALES
12/31/1995	LUCCHESSI	Ontario-South	1
12/31/1995	LEE	Ontario-South	3
12/31/1995	LEE	Quebec	1
12/31/1995	LEE	Manitoba	2
12/31/1995	GOUNOT	Quebec	1
03/29/1996	LUCCHESSI	Ontario-South	3
03/29/1996	LUCCHESSI	Quebec	1
03/29/1996	LEE	Ontario-South	2
03/29/1996	LEE	Ontario-North	2
03/29/1996	LEE	Quebec	3
03/29/1996	LEE	Manitoba	5
03/29/1996	GOUNOT	Ontario-South	3

SALES_DATE	SALES_PERSON	REGION	SALES
03/29/1996	GOUNOT	Quebec	1
03/29/1996	GOUNOT	Manitoba	7
03/30/1996	LUCCHESSI	Ontario-South	1
03/30/1996	LUCCHESSI	Quebec	2
03/30/1996	LUCCHESSI	Manitoba	1
03/30/1996	LEE	Ontario-South	7
03/30/1996	LEE	Ontario-North	3
03/30/1996	LEE	Quebec	7
03/30/1996	LEE	Manitoba	4
03/30/1996	GOUNOT	Ontario-South	2
03/30/1996	GOUNOT	Quebec	18
03/30/1996	GOUNOT	Manitoba	1
03/31/1996	LUCCHESSI	Manitoba	1
03/31/1996	LEE	Ontario-South	14
03/31/1996	LEE	Ontario-North	3
03/31/1996	LEE	Quebec	7
03/31/1996	LEE	Manitoba	3
03/31/1996	GOUNOT	Ontario-South	2
03/31/1996	GOUNOT	Quebec	1
04/01/1996	LUCCHESSI	Ontario-South	3
04/01/1996	LUCCHESSI	Manitoba	1
04/01/1996	LEE	Ontario-South	8
04/01/1996	LEE	Ontario-North	?
04/01/1996	LEE	Quebec	8
04/01/1996	LEE	Manitoba	9
04/01/1996	GOUNOT	Ontario-South	3
04/01/1996	GOUNOT	Ontario-North	1
04/01/1996	GOUNOT	Quebec	3
04/01/1996	GOUNOT	Manitoba	7

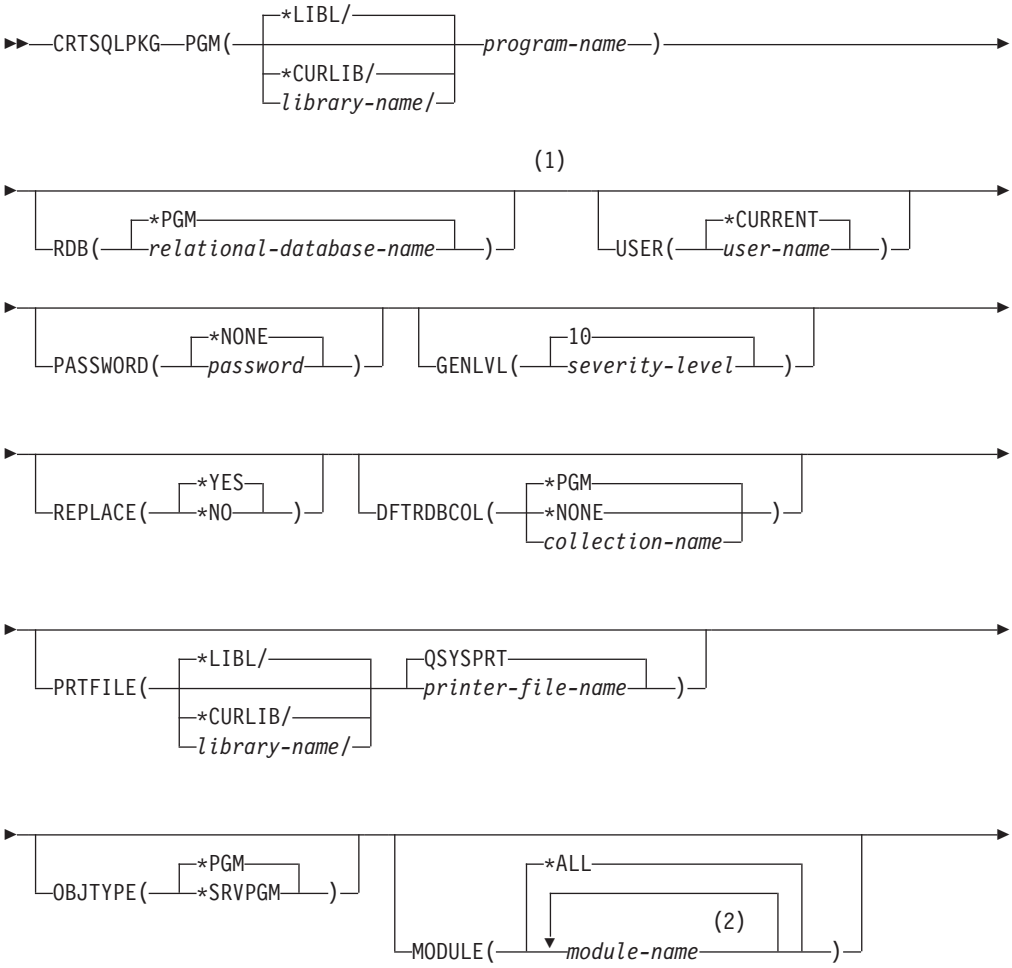
付録 B. DB2 UDB for iSeries CL コマンドの説明

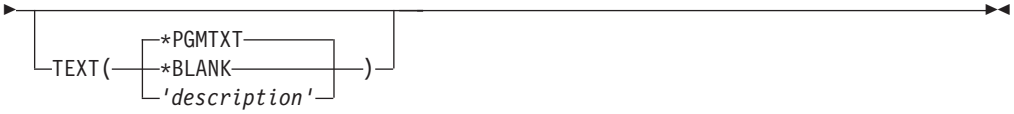
この章には、本書および SQL 解説書で参照または使用されている構文図が記載されています。

コマンドの説明については、以下のトピックを参照してください。

CRTSQLPKG (SQL パッケージ作成) コマンド

ジョブ: B,I　プログラム: B,I　REXX: B,I　Exec





注:

1 これより前にあるパラメーターはすべて、定位置形式で指定されます。

2 最高 256 個のモジュールが指定できます。

目的

SQL パッケージ作成 (CRTSQLPKG) コマンドは、既存の分散 SQL プログラムからリレーショナル・データベース上に SQL パッケージを作成 (または再作成) するために使用します。分散 SQL プログラムは、CRTSQLxxx (ここで xxx = C、CI、CBL、CBLI、FTN、PLI、または RPG または RPGI) コマンドで RDB パラメーターを指定することによって作成されるプログラムです。

パラメーター

PGM

作成しようとする SQL パッケージの対象となるプログラムの修飾名を指定します。プログラムは、分散 SQL プログラムでなければなりません。

プログラムの名前は、次のライブラリー値のどれか 1 つで修飾することができます。

***LIBL:** 最初の一致が見つかるまで、ジョブのライブラリー・リスト内のすべてのライブラリーが探索されます。

***CURLIB:** ジョブ用の現行ライブラリーが探索されます。ジョブ用の現行ライブラリーとして指定されたライブラリーがない場合は、QGPL ライブラリーが使用されます。

library-name: 探索するライブラリーの名前を指定します。

program-name: 作成しようとするパッケージの対象となるプログラムの名前を指定します。

RDB

SQL パッケージが作成されているリレーショナル・データベースの名前を指定します。

***PGM:** SQL プログラムに指定したリレーショナル・データベース名が使用されます。リレーショナル・データベース名は、分散 SQL プログラムの RDB パラメーターで指定します。

relational-database-name: 作成しようとする SQL パッケージが置かれるリレーショナル・データベースの名前を指定します。このパラメーターで有効なリレーショナル・データベース登録簿項目処理 (WRKRDBDIRE) コマンドを使用してください。

USER

会話を開始するときに遠隔システムに送られるユーザー名を指定します。

***CURRENT:** 現行ジョブに関連するユーザー名が使用されます。

user-name: アプリケーション・サーバー・ジョブで使用するユーザー名を指定します。

PASSWORD

遠隔システムで使用するパスワードを指定します。

***NONE:** パスワードは送られません。この値を指定する場合は、USER(*CURRENT) も指定する必要があります。

password: USER パラメーターに指定した、ユーザー名のパスワードを指定します。

GENLVL

SQL パッケージ作成時に検出されるエラーに許される最大重大度レベルを指定します。指定したレベルを超えるエラーが検出された場合、SQL パッケージは作成されません。

10: 省略時の重大度レベルは 10 です。

重大度レベル: 最大重大度レベルを指定します。有効な値は 0 〜 40 の範囲です。

REPLACE

既存のパッケージを新規パッケージで置き換えるかどうかを指定します。このパラメーターについての詳細は、*CL 解説書* の付録 A「パラメーターの補足説明」にあります。

***YES:** 同じ名前の既存の SQL パッケージは、新規 SQL パッケージで置き換えられます。

***NO:** 同じ名前の既存の SQL パッケージは、置き換えられません。パッケージが指定したライブラリー内にすでに存在している場合、新規 SQL パッケージは作成されません。

DFTRDBCOL

表、視点、索引および SQL パッケージの非修飾名として使用するコレクション名を指定します。このパラメーターは、パッケージ内の静的 SQL ステートメントにだけ適用されます。

***PGM:** SQL プログラムに指定されたコレクション名が使用されます。省略時のリレーショナル・データベース・コレクション名は、分散 SQL プログラムの DFTRDBCOL パラメーターで指定します。

***NONE:** 表、視点、索引および SQL パッケージの非修飾名には、プログラムを作成するために使用された CRTSQLxxx コマンドの OPTION パラメーターに指定された探索規則が使用されます。

collection-name: 修飾されていない表、視点、索引、および SQL パッケージに使用するコレクション名を指定します。

PRTFILE

SQL パッケージ作成エラー・リストを送る印刷装置ファイルの修飾名を指定します。SQL パッケージの作成時にエラーが検出されないときは、リストは作成されません。

印刷装置ファイルの名前は、次のライブラリー値のどれか 1 つで修飾することができます。

***LIBL:** 最初の一致が見つかるまで、ジョブのライブラリー・リスト内のすべてのライブラリーが探索されます。

***CURLIB:** ジョブ用の現行ライブラリーが探索されます。ジョブ用の現行ライブラリーとして指定されたライブラリーがない場合は、QGPL ライブラリーが使用されます。

library-name: 探索するライブラリーの名前を指定します。

QSYSPRT: ファイル名の指定がない場合、SQL パッケージ作成エラー・リストは IBM 提供の印刷装置ファイル QSYSPRT に送られます。

印刷装置ファイル名: SQL パッケージ作成エラー・リストが送られる印刷装置ファイルの名前を指定します。

OBJTYPE

作成する SQL パッケージの対象となるプログラムのタイプを指定します。

***PGM:** PGM パラメーターに指定されたプログラムから SQL パッケージを作成します。

***SRVPGM:** PGM パラメーターに指定されたサービス・プログラムから SQL パッケージを作成します。

MODULE

バインド・プログラム内のモジュールのリストを指定します。

***ALL:** SQL パッケージは、プログラム内の各モジュールごとに作成されます。プログラム内のどのモジュールにも SQL ステートメントが入っていない場合、または、どのモジュールも分散モジュール以外の場合、エラー・メッセージが送られます。

注: CRTSQLPKG が処理できるのは、1024 以下のモジュールが入っているプログラムです。

モジュール名: 作成しようとする SQL パッケージの対象となるプログラム内の最高 256 個までのモジュールの名前を指定します。SQL パッケージを作成する必要があるモジュールが 256 個を超える場合、複数の CRTSQLPKG コマンドを使用しなければなりません。

同一プログラムでモジュール名は重複できません。 このコマンドは、プログラム内の各モジュールを探し、MODULE パラメーターに *ALL またはモジュール名が指定されていれば、SQL パッケージを作成すべきかどうかを判別する処理を続けます。モジュールが SQL を使用して作成されており、事前コンパイル・コマンドに RDB パラメーターが指定されている場合、このモジュールの SQL パッケージが作成されます。SQL パッケージは、結合プログラムのモジュールと関連付けられています。

TEXT

SQL パッケージおよびその機能について簡単に記述するテキストを指定します。

***PGMTXT:** 作成しようとする SQL パッケージの対象となるプログラムから取り出したテキストが使用されます。

***BLANK:** テキストは指定されません。

'description': 最高 50 文字までのテキストをアポストロフィ (') で囲んで指定します。

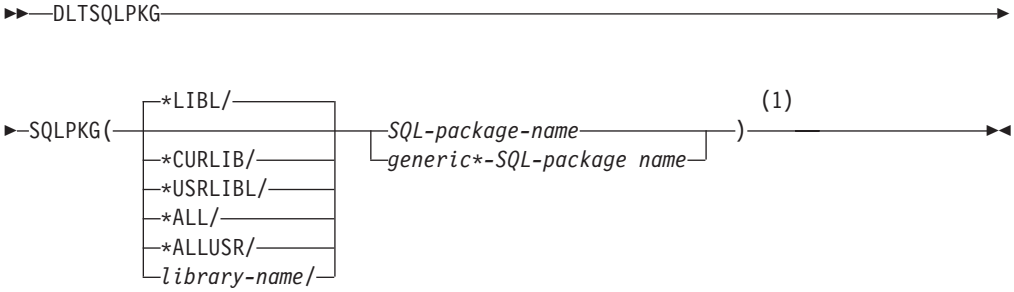
例:

```
CRTSQLPKG   PAYROLL  RDB(SYSTEMA)
  TEXT('Payroll Program')
```

このコマンドは、リレーショナル・データベース SYSTEMA 上の分散 SQL プログラム PAYROLL から SQL パッケージを作成するものです。

DLTSQLPKG (SQL パッケージ削除) コマンド

ジョブ: B,I　プログラム: B,I　REXX: B,I　Exec



注:

1　これより前にあるパラメーターはすべて、定位置形式で指定されます。

目的

SQL パッケージ削除 (DLTSQLPKG) コマンドは、1 つまたは複数の SQL パッケージを削除するために使用します。

DLTSQLPKG はローカル・コマンドであり、削除しようとする SQL パッケージが存在している iSeries システム上で使用しなければなりません。

iSeries システムである遠隔システム上の SQL パッケージを削除するには、遠隔コマンド投入 (SBMRMTCMD) コマンドを使用して、DLTSQLPKG コマンドを遠隔システムで実行します。

iSeries システム以外の遠隔システムから SQL パッケージを削除するには、以下のステップに従ってください。

- 対話式 SQL を使用して、CONNECT 命令および DROP PACKAGE 命令を実行します。
- 遠隔システムにサインオンして、そのシステムにローカルなコマンドを使用します。
- DROP PACKAGE SQL ステートメントが入っている SQL プログラムを作成して実行します。

パラメーター

DLTSQLPKG

SQLPKG

削除しようとする SQL パッケージの修飾名を指定します。特定の SQL パッケージ名または総称 SQL パッケージ名を指定することができます。

SQL パッケージの名前は、次のライブラリー値のどれか 1 つで修飾することができます。

***LIBL:** 最初の一致が見つかるまで、ジョブのライブラリー・リスト内のすべてのライブラリーが探索されます。

***CURLIB:** ジョブ用の現行ライブラリーが探索されます。 ジョブ用の現行ライブラリーとして指定されたライブラリーがない場合は、QGPL ライブラリーが使用されます。

***USRLIBL:** ジョブのライブラリー・リストのユーザー部分にあるライブラリーだけが探索されます。

***ALL:**システム内のすべてのライブラリー (QSYS を含む) を探索します。

***ALLUSR:** すべてのユーザー・ライブラリーは検索されます。 文字 Q で始まっていない名前を持つすべてのライブラリーは、次の場合を除き、検索されます。

```
#CGULIB     #DFULIB     #RPGLIB     #SEULIB
#COBLIB     #DSULIB     #SDALIB
```

次の Qxxx ライブラリーが IBM によって提供されていますが、これらには概して、頻繁に変更されるユーザー・データが入っています。 したがって、これらのライブラリーはユーザー・ライブラリーと見なされ、これらも検索されます。

```
QDSNX          QRCL          QUSRBRM          QUSRSYS
QGPL           QS36F         QUSRIJS          QUSRVxRxMx
QGPL38         QUSER38       QUSRINFSKR
QPFRDATA       QUSRADSM      QUSRRDARS
```

注: QUSRVxRxMx という形の、異なるライブラリー名は、 IBM がサポートする各リリースごとにユーザーが作成することができます。 VxRxMx とは、ライブラリーのバージョン、リリース、およびモディフィケーション・レベルです。

library-name: 探索するライブラリーの名前を指定します。

SQL-package-name: 削除しようとする SQL パッケージの名前を指定します。

generic-SQL-package-name:* 削除しようとする SQL パッケージの総称名を指定します。 総称名は、1 つまたは複数の文字からなる文字ストリングで、アスタリスク (*) が続きます (たとえば、ABC*)。総称名を指定すると、その総称名で始まる名前を持つすべての SQL パッケージで、それに関する権限をユーザーが持っているパッケージが削除されます。総称 (接頭部) 名と一緒にアスタリスクを指定しないと、システムは、総称名を完全な SQL パッケージ名と想定します。

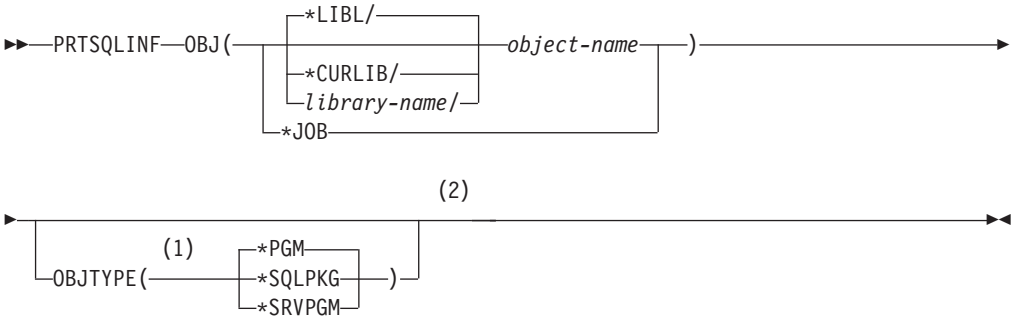
例:

```
DLTSQLPKG  SQLPKG(JONES)
```

このコマンドは、SQL パッケージ JONES を削除します。

PRTSQLINF (SQL 情報印刷) コマンド

ジョブ: B,I　プログラム: B,I　REXX: B,I　Exec



注:

1　OBJTYPE パラメーターは、OBJ(*JOB) が指定されているときには、使用できません。

2　これより前にあるパラメーターはすべて、定位置形式で指定されます。

目的

SQL 情報印刷 (PRTSQLINF) コマンドは、プログラム、SQL パッケージ、サービス・プログラム、またはジョブの中の SQL ステートメントについての情報を印刷します。この情報には、SQL ステートメント、ステートメントの実行時に使用されるアクセス・プラン、および、オブジェクトのソース・メンバーの事前コンパイル中または SQL ステートメントが実行されるときに定義されるコマンド・パラメーターのリストが含まれます。

パラメーター

OBJ

SQL 情報を印刷したいオブジェクトの名前、または、ジョブの SQL 情報を印刷することを表す '*JOB' を指定します。名前付きオブジェクトは、プログラム、SQL パッケージ、または、サービス・プログラムのいずれでもかまいません。

オブジェクトの名前は、以下のライブラリー値のいずれかで修飾することができます。

***LIBL:** 最初の一致が見つかるまで、ジョブのライブラリー・リスト内のすべてのライブラリーが探索されます。

***CURLIB:** ジョブ用の現行ライブラリーが探索されます。 ジョブ用の現行ライブラリーとして指定されたライブラリーがない場合は、QGPL ライブラリーが使用されます。

library-name: 探索するライブラリーの名前を指定します。

object-name: 情報を印刷したいプログラム、SQL パッケージ、または、サービス・プログラムの名前を指定します。

- ***JOB:** 現行ジョブの SQL 情報を印刷することを示します。

オプション・パラメーター

OBJTYPE

オブジェクトのタイプを指定します。

- ***PGM:** オブジェクトはプログラムです。
- ***SQLPKG:** オブジェクトは SQL パッケージです。
- ***SRVPGM:** オブジェクトはサービス・プログラムです。

例:

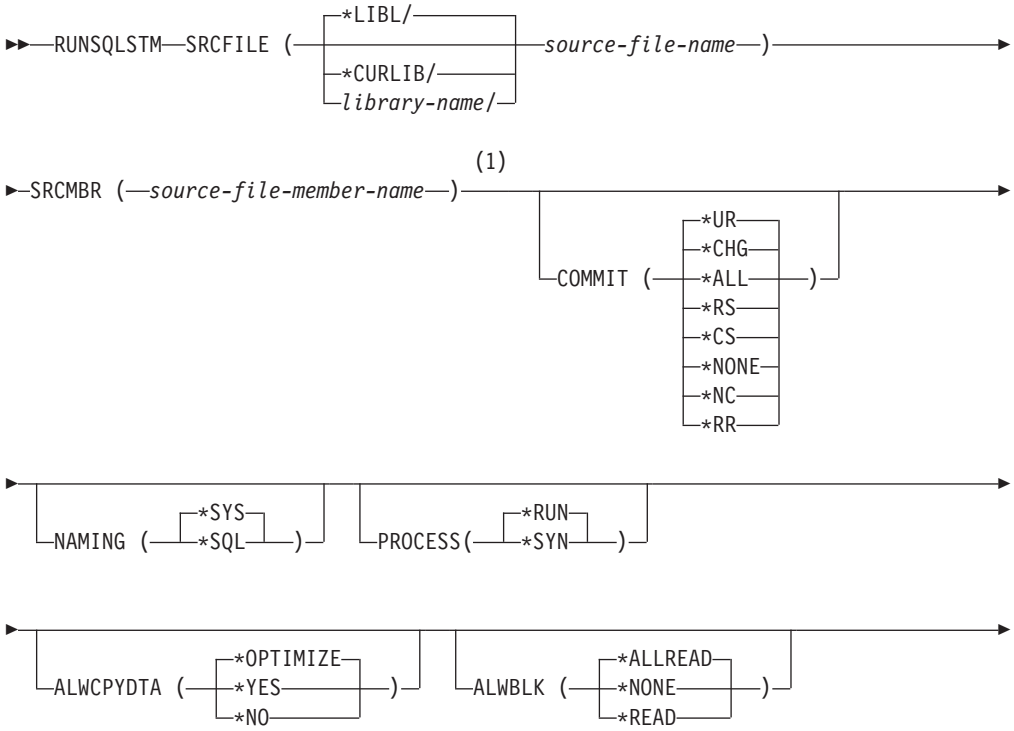
```
PRTSQLINF PAYROLL
```

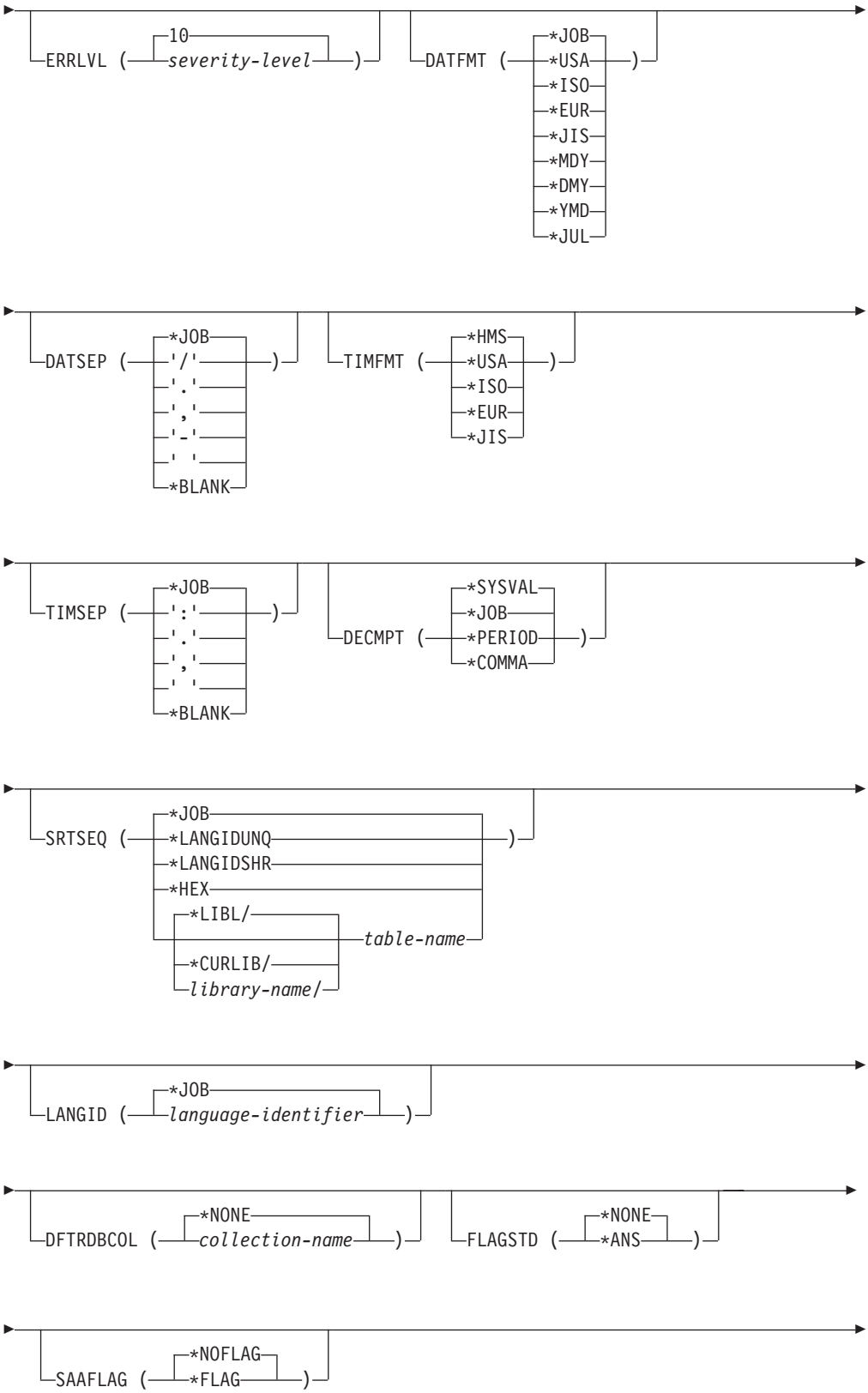
このコマンドは、プログラム PAYROLL に入っている SQL ステートメントについての情報を印刷します。

現在、OBJTYPE は従属キーワードで、 OBJ パラメーターが *JOB でないときにのみプロンプトが出されることに注意してください。

RUNSQLSTM (SQL ステートメント実行) コマンド

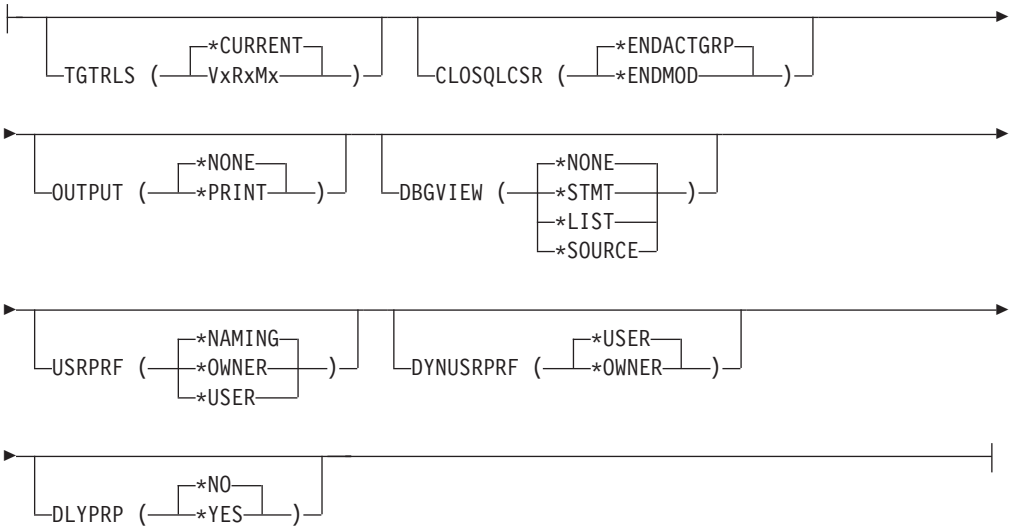
ジョブ: B,I　プログラム: B,I　REXX: B,I　Exec




ERRLVL (10 severity-level)
DATFMT (*JOB *USA *ISO *EUR *JIS *MDY *DMY *YMD *JUL)
DATSEP (*JOB '/' '.' ',' '-' ' ' *BLANK)
TIMFMT (*HMS *USA *ISO *EUR *JIS)
TIMSEP (*JOB ':' '.' ',' ' ' *BLANK)
DECMPT (*SYSVAL *JOB *PERIOD *COMMA)
SRTSEQ (*JOB *LANGIDUNQ *LANGIDSHR *HEX *LIBL/ *CURLIB/ library-name/ table-name)
LANGID (*JOB language-identifier)
DFTRDBCOL (*NONE collection-name)
FLAGSTD (*NONE *ANS)
SAAFLAG (*NOFLAG *FLAG)

RUNSQLSTM

SQL ルーチン用のパラメーター:



注:

1 これより前にあるパラメーターはすべて、定位置形式で指定されます。

目的

SQL ステートメント実行 (RUNSQLSTM) コマンドは、SQL ステートメントのソース・ファイルを処理します。

パラメーター

SRCFILE

実行しようとする SQL ステートメントが入っているソース・ファイルの修飾名を指定します。

ソース・ファイルの名前は、次のライブラリー値のどれか 1 つで修飾することができます。

***LIBL:** 最初の一致が見つかるまで、ジョブのライブラリー・リスト内のすべてのライブラリーが探索されます。

***CURLIB:** ジョブ用の現行ライブラリーが探索されます。 ジョブ用の現行ライブラリーとして指定されたライブラリーがない場合は、QGPL ライブラリーが使用されます。

library-name: 探索するライブラリーの名前を指定します。

source-file-name: 実行しようとする SQL ステートメントが入っているソース・ファイルの名前を指定します。ソース・ファイルは、データベース・ファイルでもインライン・データ・ファイルでも構いません。

SRCMBR

実行しようとする SQL ステートメントが入っているソース・ファイル・メンバーの名前を指定します。

COMMIT

ソース・ファイル内の SQL ステートメントをコミットメント制御のもとで実行するかどうかを指定します。

***CHG または *UR:** SQL の ALTER、 CALL、 COMMENT ON、 CREATE、 DROP、GRANT、LABEL ON、RENAME、 および REVOKE ステートメントで参照されるオブジェクトと更新、削除、および挿入される行が、作業単位 (トランザクション) の終わりまでロックされることを指定します。他のジョブにおけるコミットされていない変更は見ることができます。

***ALL または *RS:** SQL の ALTER、CALL、COMMENT ON、CREATE、DROP、GRANT、LABEL ON、RENAME、および REVOKE ステートメントで参照されるオブジェクトと、選択、更新、削除、および挿入された行を、作業単位 (トランザクション) が終了するまでロックすることを指定します。他のジョブでのコミットされていない変更は見ることができません。

***CS:** SQL の ALTER、CALL、COMMENT ON、CREATE、DROP、GRANT、LABEL ON、RENAME、および REVOKE ステートメントで参照されるオブジェクトと、更新、削除、および挿入された行を、作業単位 (トランザクション) が終了するまでロックすることを指定します。選択されたが、更新されていない行は、次の行が選択されるまでロックされます。 他のジョブでのコミットされていない変更は見ることができません。

***NONE または *NC:** コミットメント制御が使用されないことを指定します。他のジョブにおけるコミットされていない変更を見ることができます。 SQL の DROP COLLECTION ステートメントがプログラムに組み込まれている場合、*NONE または *NC を使用する必要があります。 RDB パラメーターでリレーショナル・データベースを指定していて、そのリレーショナル・データベースが AS/400 以外のシステム上にある場合は、 *NONE または *NC を指定することはできません。

***RR:** SQL の ALTER、 CALL、 COMMENT ON、 CREATE、 DROP、 GRANT、 LABEL ON、 RENAME、および REVOKE ステートメントの中で参照されたオブジェクト、および選択、更新、削除、または挿入された行を、作業単位 (トランザクション) が終了するまでロックすることを指定します。他のジョブでのコミットされていない変更は見ることができません。
SELECT、UPDATE、DELETE、および INSERT ステートメントの中で参照されたすべての表は、作業単位 (トランザクション) が終了するまで、排他的にロックされます。

NAMING

SQL ステートメントでオブジェクトに名前を付けるときに使用する命名規則を指定します。

***SYS:** システムの命名規則 (library-name/file-name) を使用します。

***SQL:** SQLの命名規則 (collection-name.table-name) を使用します。

PROCESS

ソース・ファイル・メンバー内の SQL ステートメントを実行するか、構文検査するだけかを指定します。

***RUN:** ステートメントを構文検査して実行します。

***SYN:** ステートメントを構文検査するだけです。

ALWCPYDTA

SELECT ステートメントでデータのコピーが使用できるかどうかを指定します。

***OPTIMIZE:** システムは、データベースから直接取り出したデータを使用するか、データのコピーを使用するかを決定します。その決定は、どの方式で最良のパフォーマンスが得られるかに基づきます。 COMMIT が *CHG または *CS で、ALWBLK が *ALLREAD でない場合、あるいは COMMIT が *ALL または *RR の場合、データのコピーが使用されるのは、照会を実行する必要があるときだけです。

***YES:** 必要なときだけデータのコピーが使用されます。

***NO:** データのコピーは使用されません。 照会を実行するのにデータの一時コピーが必要な場合には、エラー・メッセージが戻されます。

ALWBLK

データベース・マネージャーがレコードのブロック化を使用できるかどうか、および読み取り専用カーソルについてどの程度までブロック化が使用できるかを指定します。

***ALLREAD:** COMMIT パラメーターで *NONE または *CHG が指定されている場合は、行は読み取り専用カーソルに対してブロックされます。プログラム内で、明示的に更新することができないすべてのカーソルは、プログラム内に EXECUTE または EXECUTE IMMEDIATE ステートメントがある場合でも、読み取り専用にオープンされます。

*ALLREAD を指定すると、

- *READ の場合に許されるブロック化に加え、コミットメント制御レベル *CHG のもとでレコードのブロック化を行うことができます。
- プログラムの中のほとんどすべての読み取り専用カーソルのパフォーマンスを向上できますが、照会が次のように制限されます。
  - ロールバック (ROLLBACK) コマンド、ホスト言語で書いた ROLLBACK ステートメント、または ROLLBACK HOLD SQL ステートメントは、*ALLREAD の指定があるとき、読み取り専用カーソルの再位置決めはしません。
  - カーソル内の行を更新するために、位置付けされた UPDATE または DELETE 使用ステートメントの動的実行 (たとえば、EXECUTE IMMEDIATEによる) をすることはできません。ただし、カーソルに関する DECLARE ステートメントに FOR UPDATE 文節が含まれている場合を除きます。

***NONE:** カーソルの対象となるデータを検索するとき、行がブロック化されません。

*NONE を指定すると、

- 検索されるデータが最新であることが保証されます。
- 照会の対象となるデータの最初の行を検索するときの所要時間が短縮します。
- 照会の最初の数行だけが検索されてから照会がクローズされるときは、プログラムによって使用されないデータ行のブロックの検索をデータベース・マネージャーに中止させます。
- 大量の行を検索する照会全体のパフォーマンスが低下する可能性があります。

***READ:** 次のとき、カーソルの対象となるデータを読み取り専用で検索するときレコードがブロック化されます。

- COMMIT パラメーターに *NONE の指定があるとき。これは、コミットメント制御が使用されないことを示します。
- カーソルが FOR FETCH ONLY 文節を使用して宣言されているか、あるいは位置指定の UPDATE または DELETE ステートメントをカーソルに対して実行できる動的ステートメントがないとき。

*READ を指定すると、上記条件を満足する照会の全体のパフォーマンスが向上し、大量のレコードを検索することができます。

ERRLVL

SQL ステートメントの処理によって生成されるメッセージの重大度に基づいて、処理が成功かどうかを指定します。このパラメーターに指定した値より大きいエラーが処理中に検出されると、これ以後のステートメントは処理されません。処理がコミットメント制御の下で実行されている場合は、ステートメントはロールバックされます。

10: 重大度レベル 10 を超えるエラー・メッセージを受け取った場合、ステートメント処理が停止します。

重大度レベル: 使用する重大度レベルを指定します。

DATFMT

日付結果列にアクセスするときに使用する形式を指定します。入力日付ストリングの場合、指定された値を使用して、日付が有効な形式で指定されたかどうかを判別します。

注: *USA、*ISO、*EUR、または *JIS の形式を使用する入力日付ストリングは常に有効です。

***JOB:** ジョブに指定された形式が使用されます。ジョブの現行日付形式を判別するには、ジョブ表示 (DSPJOB) コマンドを使用してください。

***USA:** 米国の日付形式 (mm/dd/yyyy) が使用されます。

***ISO:** 国際標準化機構 (ISO) の日付形式 (yyyy-mm-dd) が使用されます。

***EUR:** ヨーロッパの日付形式 (dd.mm.yyyy) が使用されます。

***JIS:** 日本工業規格の日付形式 (yyyy-mm-dd) が使用されます。

***MDY:** 日付形式 (mm/dd/yy) が使用されます。

***DMY:** 日付形式 (dd/mm/yy) が使用されます。

***YMD:** 日付形式 (yy/mm/dd) が使用されます。

***JUL:** 年間通算日の日付形式 (yy/ddd) が使用されます。

DATSEP
日付結果列にアクセスするときに使用される区切り記号を指定します。

注: このパラメーターは、*JOB、*MDY、*DMY、*YMD、または *JUL が DATFMT パラメーターで指定されたときだけ適用されます。

***JOB:** ジョブに関して指定された日付区切り記号が使用されます。 ジョブ表示 (DSPJOB) コマンドを使用すると、ジョブの現在の値を確かめることができます。

'/': スラッシュ (/) が使用されます。

'.': ピリオド (.) が使用されます。

',': コンマ (,) が使用されます。

'-': ダッシュ (-) が使用されます。

' ': ブランク ( ) が使用されます。

***BLANK:** ブランク ( ) が使用されます。

TIMFMT
時刻結果列にアクセスするときに使用される形式を指定します。 入力時刻ストリングについては、時刻が有効な形式で指定されているかどうかを判別するために、指定された値が使用されます。

注: *USA、*ISO、*EUR、または *JIS の形式を使用する入力日付ストリングは常に有効です。

***HMS:** hh:mm:ss 形式が使用されます。

***USA:** 米国の時刻形式 hh:mm xx が使用されます。ただし、xx は AM か PM です。

***ISO:** 国際標準化機構 (ISO) の時刻形式 hh.mm.ss が使用されます。

***EUR:** ヨーロッパの時刻形式 hh.mm.ss が使用されます。

***JIS:** 日本工業規格の時刻形式 hh:mm:ss が使用されます。

TIMSEP
時刻結果列にアクセスするときに使用される区切り記号を指定します。

注: このパラメーターは、TIMFMT パラメーターで *HMS が指定されたときだけ適用されます。

***JOB:** ジョブに関して指定された時刻区切り記号が使用されます。 ジョブ表示 (DSPJOB) コマンドを使用すると、ジョブの現在の値を確かめることができます。

':': コロン (:) が使用されます。

'.': ピリオド (.) が使用されます。

',': コンマ (,) が使用されます。

' ': ブランク ( ) が使用されます。

***BLANK:** ブランク ( ) が使用されます。

DECMPT

SQL ステートメント内の数値定数に使用される小数点を指定します。

***JOB:** SQL で数値定数の小数点として使用される値は、ステートメントを実行するジョブによって指定された小数点の表現になります。

***SYSVAL:** 小数点として QDECFMT システム値が使用されます。

***PERIOD:** ピリオドが小数点になります。

***COMMA:** コンマが小数点になります。

SRTSEQ

SQL ステートメントの中のストリング比較に使用される分類順序表を指定します。

***JOB:** ジョブの LANGID 値が取り出されます。

***LANGIDSHR:** 分類順序表は複数の文字に同じ重みを使用するので、LANGID パラメーターで指定した言語に関連付けられた共用分類順序表になります。

***LANGIDUNQ:** LANGID パラメーターで指定した言語用の固有の分類表が使用されます。

***HEX:** 分類順序表は使用しません。 分類順序を決定するには、文字の 16 進値を使用します。

表名は、次のライブラリー値のどれか 1 つで修飾することができます。

***LIBL:** 最初の一致が見つかるまで、ジョブのライブラリー・リスト内のすべてのライブラリーが探索されます。

***CURLIB:** ジョブ用の現行ライブラリーが探索されます。 ジョブ用の現行ライブラリーとして指定されたライブラリーがない場合は、QGPL ライブラリーが使用されます。

library-name: 探索するライブラリーの名前を指定します。

table-name: 使用される分類順序表の名前を指定します。

LANGID

SRTSEQ(*LANGIDUNQ) または SRTSEQ(*LANGIDSHR) が指定されたときに使用される言語識別コードを指定します。

***JOB:** ジョブの LANGID 値は事前コンパイル時に検索されます。

language-identifier: 言語識別コードを指定してください。

DFTRDBCOL

表、表、索引、および SQL パッケージの非修飾名用に使用されるコレクション名を指定します。

***NONE:** OPTION パラメーターで定義した命名規則が使用されます。

collection-name: コレクション識別名を指定します。 この値は、OPTION パラメーターで指定した命名規則の代わりに使用されます。

FLAGSTD

米国規格協会 (ANSI) のフラグ付け機能を指定します。 このパラメーターは、SQL ステートメントにフラグを付けて、それらが次の規格に適合するかどうかを検査します。

```
ANSI X3.135-1992 entry
ISO 9075-1992 entry
FIPS 127.2 entry
```

***NONE:** SQL ステートメントに対して、ANSI 規格に準拠しているかどうかを判別するための検査は行われません。

***ANS:** SQL ステートメントに対して、ANSI 規格に準拠しているかどうかを判別するための検査が行われます。

SAAFLAG

IBM SQL のフラグ付け機能を指定します。 このパラメーターは、SQL ステートメントが IBM SQL 構文に準拠しているかどうかを検査するために、SQL ステートメントにフラグを付けます。 IBM データベース・プロダクトの IBM SQL 構文に関する詳細は、*DRDA IBM SQL Reference* (SC26-3255-00) に記載されています。

***NOFLAG:** SQL ステートメントに対して、IBM SQL 構文に準拠しているかどうかを判別するための検査は行われません。

***FLAG:** SQL ステートメントに対して、IBM SQL 構文に準拠しているかどうかの検査が行われます。

PRTFILE

RUNSQLSTM 印刷出力から送られる印刷装置ファイルの修飾名を指定します。ファイルの長さは 132 バイト以上でなければなりません。レコード長が 132 バイト未満のファイルを指定すると、情報が失われます。

印刷装置ファイルの名前は、次のライブラリー値のどれか 1 つで修飾することができます。

***LIBL:** 最初の一致が見つかるまで、ジョブのライブラリー・リスト内のすべてのライブラリーが探索されます。

***CURLIB:** ジョブ用の現行ライブラリーが探索されます。 ジョブ用の現行ライブラリーとして指定されたライブラリーがない場合は、QGPL ライブラリーが使用されます。

library-name: 探索するライブラリーの名前を指定します。

QSYSPRT: ファイル名の指定がない場合、RUNSQLSTM 印刷出力は IBM 提供の印刷装置ファイル QSYSPRT に送られます。

printer-file-name: RUNSQLSTM 印刷出力が送られる印刷装置ファイルの名前を指定します。

SQL ルーチン用のパラメーター:

以下にリストされているパラメーターは、SQL プロシージャー、SQL 関数、および SQL トリガーを作成するソース・ファイル内のステートメントにのみ適用されます。これらのパラメーターは、SQL プロシージャー、SQL 関数、および SQL トリガーに関連付けられているプログラム・オブジェクトの作成時に使用されます。

TGTRLS

ユーザーが作成中のオブジェクトを使用する予定の、オペレーティング・システムのリリース・レベルを指定します。

*CURRENT 値の例では、VxRxMx 形式でリリースを指定する *release-level* 値が示されています。ここで Vx はバージョン、Rx はリリース、Mx はモディフィケーション・レベルです。たとえば、V2R3M0 はバージョン 2、リリース 3、モディフィケーション・レベル 0 です。

***CURRENT** ユーザーのシステム上で現在稼動しているオペレーティング・システムのリリースで、オブジェクトが使用されます。たとえば、システムで V2R3M5 が稼動している場合、*CURRENT は、 V2R3M5 が導入されたシステムでそのオブジェクトを使用する予定であることを意味します。また、ユーザーは、それ以降のリリースのオペレーティング・システムを導入したシステム上でも、そのオブジェクトを使用できます。

注: V2R3M5 がシステム上で稼動している場合、および V2R3M0 が導入されたシステムでオブジェクトが使用される場合、TGTRLS(*CURRENT) ではなく TGTRLS(V2R3M0) を使用してください。

release-level: VxRxMx 形式でリリースを指定します。そのオブジェクトは、指定されたリリース、あるいはそれ以降のリリースのオペレーティング・システムを導入したシステムで使用できます。

有効な値は、現行バージョン、リリース、モディフィケーション・レベルによります。また、有効な値は各新規リリースで変わります。コマンドがサポートする初期リリース・レベルよりもさらに前のリリース・レベルを指定した場合には、エラー・メッセージはこれがサポートする最も初期のリリース・レベルを示して送信されます。

CLOSQLCSR

SQL カーソルが暗黙にクローズされ、SQL で準備されたステートメントが暗黙に破棄され、LOCK TABLE ロックが解除される時点を指定します。 SQL カーソルは、CLOSE、COMMIT、または ROLLBACK (HOLD を指定しない) SQL ステートメントを出すときに、明示的にクローズされます。

***ENDACTGRP:** SQL カーソルがクローズされ、SQL の準備したステートメントを暗黙に廃棄します。

ENDMOD: モジュールが終了した時点で、SQL カーソルをクローズし、SQL が準備したステートメントを暗黙に廃棄します。呼び出しスタック上の最初の SQL プログラムが終了すると、LOCK TABLE ロックが解除されます。

OUTPUT
事前コンパイラーのリストを生成するかどうかを指定します。

***NONE:** 事前コンパイラーのリストは生成されません。

***PRINT:** 事前コンパイラーのリストが生成されます。

DBGVIEW
SQL 事前コンパイラーによって提供されるソース・デバッグ情報のタイプを指定します。

***NONE:** ソース・プログラム・表は生成されません。

***STMT:** コンパイルしたモジュールを、プログラム・ステートメント番号と記号識別コードによってデバッグできます。

***LIST:** コンパイルしたモジュール・オブジェクトをデバッグするためのリスト表示を生成します。

***SOURCE:** SQL プロシージャーのソース視点、関数、およびトリガーが生成されます。

USRPRF
コンパイル済みプログラム・オブジェクトが実行されるときに使用されるユーザー・プロファイル (プログラム・オブジェクトが SQL ステートメント内の各オブジェクトに対して所有する権限を含む) を指定します。プログラム・オブジェクトによって使用できるオブジェクトの制御には、プログラム所有者またはプログラム・ユーザーのプロファイルが使用されます。

***NAMING:** ユーザー・プロファイルが命名規則によって判別されます。命名規則が *SQL である場合は、USRPRF(*OWNER) が使用されます。命名規則が *SYS である場合は、USRPRF(*USER) が使用されます。

***USER:** プログラム・オブジェクトを実行するユーザーのプロファイルが使用されます。

***OWNER:** プログラム所有者とプログラム・ユーザーの両方のユーザー・プロファイルがプログラムの実行時に使用されます。

DYNUSRPRF
動的 SQL ステートメントで使用するユーザー・プロファイルを指定します。

***USER:** ローカル動的 SQL ステートメントは、プログラムのユーザー・プロファイルの下で実行されます。分散動的 SQL ステートメントは、SQL パッケージのユーザー・プロファイルの下で実行されます。

***OWNER:** ローカル動的 SQL ステートメントは、プログラムの所有者のプロファイルの下で実行されます。分散動的 SQL ステートメントは、SQL パッケージの所有者のプロファイルの下で実行されます。

DLYPRP
PREPARE ステートメントについての動的ステートメント妥当性検査が、OPEN、EXECUTE、または DESCRIBE ステートメントが実行されるまで遅らされるかどうかを指定します。妥当性検査を遅らせると、余分な妥当性検査が除かれるため、パフォーマンスが向上します。

***NO:** 動的ステートメント妥当性検査は遅らされません。動的ステートメントが準備されるとき、アクセス・プランが妥当性検査されます。 動的ステートメン

トが OPEN または EXECUTE ステートメントで使用される場合、アクセス・プランが再度妥当性検査されます。 動的ステートメントによって参照されるオブジェクトの権限または存在は変化する場合があるので、OPEN または EXECUTE ステートメントを出した後、SQLCODE または SQLSTATE を検査して、動的ステートメントがまだ有効であるか確かめる必要があります。

***YES:** 動的ステートメント妥当性検査は、動的ステートメントが OPEN、EXECUTE、または DESCRIBE SQL ステートメントで使用されるまで遅らされます。動的ステートメントが使用されたときは、その妥当性検査が行われて、アクセス・プランが作られます。このパラメーターで *YES を指定する場合は、OPEN、EXECUTE、または DESCRIBE ステートメントを実行した後 SQLCODE と SQLSTATE を調べて、動的ステートメントが有効であるかどうかを確かめておく必要があります。

注: *YES を指定したときは、PREPARE ステートメントで INTO 文節が使用されている場合や、OPEN が動的ステートメントに対して出される前に DESCRIBE ステートメントがその動的ステートメントを使用した場合は、パフォーマンスは向上しません。

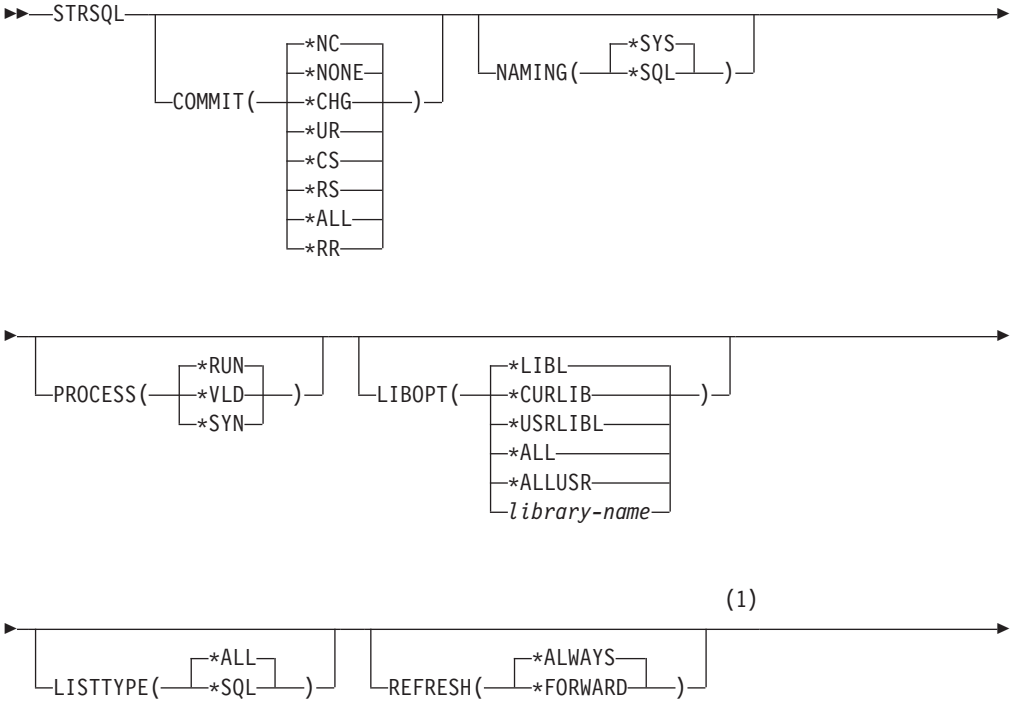
例:

```
RUNSQLSTM  SRCFILE(MYLIB/MYFILE)  SRCMBR(MYMBR)
```

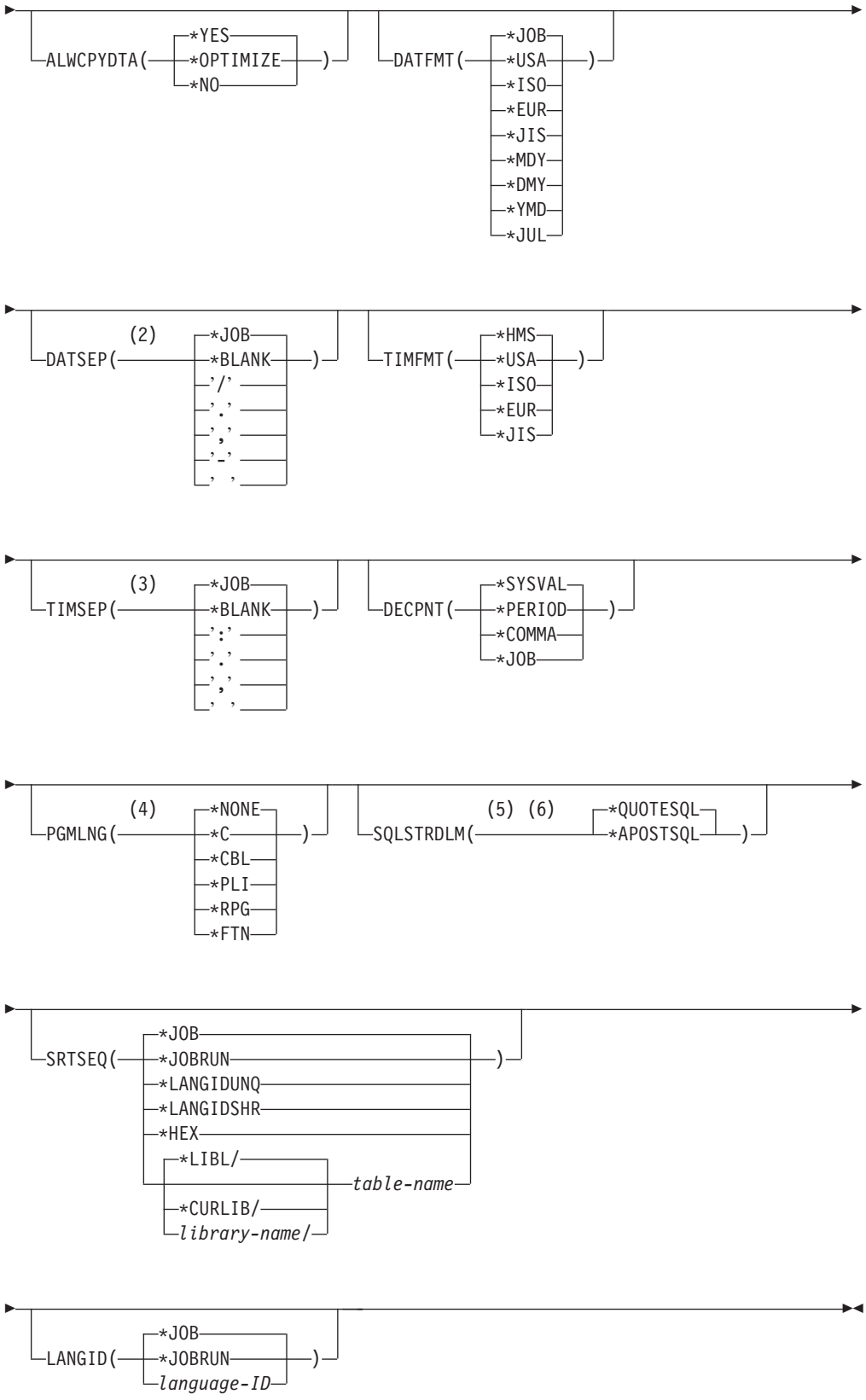
このコマンドは、ライブラリー MYLIB 内のファイル MYFILE にあるメンバー MYMBR の SQL ステートメントを処理します。

STRSQL (SQL 対話式セッション開始) コマンド

ジョブ: I　プログラム: I　REXX: I　Exec



STRSQL



注:

1 これより前にあるパラメーターはすべて、定位置形式で指定されます。

2 DATSEP は、DATFMT パラメーターに *MDY、*DMY、*YMD、または *JUL の指定があるときだけ有効です。

3 TIMSEP は、TIMFMT(*HMS) の指定があるときだけ有効です。

4 PGMLNG と SQLSTRDLM は、PROCESS(*SYN) の指定があるときだけ有効です。

5 PGMLNG と SQLSTRDLM は、PROCESS(*SYN) の指定があるときだけ有効です。

6 SQLSTRDLM は、PGMLNG(*CBL) の指定があるときだけ有効です。

目的

SQL 対話式セッション開始 (STRSQL) コマンドは、対話式構造化照会言語 (SQL) プログラムを開始させるものです。このプログラムは対話式 SQL プログラムのステートメント入力を開始し、直ちに「SQL ステートメント入力」画面を表示します。この画面からは、対話環境で SQL ステートメントを作成、編集、入力、および実行することができます。プログラムの実行中に出されたメッセージは、この画面に表示されます。

パラメーター

COMMIT

SQL ステートメントがコミットメント制御のもとで実行されるかどうかを指定します。

***NONE または *NC:** コミットメント制御を使用しないことを指定します。 他のジョブにおけるコミットされていない変更を見ることができます。 SQL の DROP COLLECTION ステートメントがプログラムに組み込まれている場合、*NONE または *NC を使用する必要があります。 RDB パラメーターでリレーショナル・データベースを指定していて、そのリレーショナル・データベースが iSeries 以外のシステム上にある場合は、*NONE または *NC を指定することはできません。

***CHG または *UR:** SQL の ALTER、 CALL、 COMMENT ON、 CREATE、DROP、GRANT、LABEL ON、RENAME、および REVOKE ステートメントで参照されるオブジェクトと更新、削除、および挿入される行が、作業単位 (トランザクション) の終わりまでロックされることを指定します。他のジョブにおけるコミットされていない変更を見ることができます。

***CS:** SQL の ALTER、CALL、COMMENT ON、CREATE、DROP、GRANT、LABEL ON、RENAME、および REVOKE ステートメントで参照されるオブジェクトと、更新、削除、および挿入された行を、作業単位 (トランザクション) が終了するまでロックすることを指定します。選択されたが、更新されていない行は、次の行が選択されるまでロックされます。 他のジョブでのコミットされていない変更は見ることができません。

***ALL または *RS:** SQL の ALTER、CALL、COMMENT ON、CREATE、DROP、GRANT、LABEL ON、RENAME、および REVOKE ステートメントで参照されるオブジェクトと、選択、更新、削除、および挿入された行を、作業単位 (トランザクション) が終了するまでロックすることを指定します。他のジョブでのコミットされていない変更は見ることができません。

***RR:** SQL の ALTER、 CALL、 COMMENT ON、 CREATE、 DROP、 GRANT、 LABEL ON、 RENAME、および REVOKE ステートメントの中で参照されたオブジェクト、および選択、更新、削除、または挿入された行を、作業単位 (トランザクション) が終了するまでロックすることを指定します。他のジョブでのコミットされていない変更は見ることができません。SELECT、UPDATE、DELETE、および INSERT ステートメントの中で参照されたすべての表は、作業単位 (トランザクション) が終了するまで、排他的にロックされます。

注: CRTSQLXXX コマンド (ここで XXX=CI、CPPI、CBL、FTN、PLI、CBLI、RPG または RPGI) のこのパラメーターの省略時値は *CHG です。

NAMING

SQL ステートメントでオブジェクトに名前を付けるときに使用する命名規則を指定します。

***SYS:** システムの命名規則 (library-name/file-name) を使用します。

***SQL:** SQLの命名規則 (collection-name.table-name) を使用します。

PROCESS

SQL ステートメントの処理に使用する値を指定します。

***RUN:** ステートメントは、構文検査とデータ検査が行われた上で実行されます。

***VLD:** ステートメントは、構文検査とデータ検査が行われますが、実行されません。

***SYN:** ステートメントは、構文検査だけが行われます。

LIBOPT

F4、F16、F17、または F18 機能キーを押したとき、どのコレクションおよびライブラリーを基礎にしてコレクション・リストを作成するかを指定します。

コレクション・リストの名前は、次のライブラリー値のどれか 1 つで修飾することができます。

***LIBL:** 最初の一致が見つかるまで、ジョブのライブラリー・リスト内のすべてのライブラリーが探索されます。

***CURLIB:** ジョブ用の現行ライブラリーが探索されます。 ジョブ用の現行ライブラリーとして指定されたライブラリーがない場合は、QGPL ライブラリーが使用されます。

***USRLIBL:** ジョブのライブラリー・リストのユーザー部分にあるライブラリーだけが探索されます。

***ALL:** QSYS を含む、システム内のすべてのライブラリーが探索されます。

***ALLUSR:** すべてのユーザー・ライブラリーは検索されます。 文字 Q で始まっていない名前を持つすべてのライブラリーは、次の場合を除き、検索されます。

```
#CGULIB     #DFULIB     #RPGLIB     #SEULIB
#COBLIB     #DSULIB     #SDALIB
```

次の Qxxx ライブラリーが IBM によって提供されていますが、これらには概して、頻繁に変更されるユーザー・データが入っています。 したがって、これらのライブラリーはユーザー・ライブラリーと見なされ、これらも検索されます。

```
QDSNX          QRCL          QUSRBRM          QUSRSYS
QGPL           QS36F         QUSRIJS          QUSRVxRxMx
QGPL38         QUSER38       QUSRINFSKR
QPFRDATA       QUSRADSM      QUSRRDARS
```

注: QUSRVxRxMx という形の異なるライブラリー名は、IBM がサポートする各リリースごとにユーザーが作成することができます。 VxRxMx とは、ライブラリーのバージョン、リリース、およびモディフィケーション・レベルです。

library-name: 探索するライブラリーの名前を指定します。

LISTTYPE

F4、F16、F17、または F18 機能キーを押すことによりリスト・サポートで表示されるオブジェクトのタイプを指定します。

***ALL:** すべてのオブジェクトが表示されます。

***SQL:** SQL 作成のオブジェクトだけが表示されます。

REFRESH

表示選択出力データをいつ再表示するかを指定します。

***ALWAYS:** データは、通常、正方向スクロールや逆方向スクロールの際に再表示されます。

***FORWARD:** データは、データの終わりまで初めて正方向スクロールするときだけ再表示されます。逆方向スクロールの際は、すでに表示されているデータのコピーが表示されます。

ALWCPYDTA

SELECT ステートメントでデータのコピーが使用できるかどうかを指定します。 COMMIT(*ALL) の指定があると、SQL 実行時には ALWCPYDTA 値は無視され、現行データが使用されます。

***YES:** 必要なときにデータのコピーが使用されます。

***OPTIMIZE:** システムは、データベースから取り出したデータを使用するか、データのコピーを使用するかを決定します。どちらの方が最良のパフォーマンスが得られるかに基づいて決定が行われます。

***NO:** データのコピーを使用することはできません。照会を実行するのにデータの一時コピーが必要な場合には、エラー・メッセージが戻されます。

DATFMT

SQL ステートメントで使用する日付形式を指定します。

***JOB:** ジョブ属性 DATFMT で指定された形式が使用されます。

***USA:** 米国の日付形式 (mm/dd/yyyy) が使用されます。

***ISO:** 国際標準化機構の日付形式 (yyyy-mm-dd) が使用されます。

***EUR:** ヨーロッパの日付形式 (dd.mm.yyyy) が使用されます。

***JIS:** 日本工業規格の西暦の日付形式 (yyyy-mm-dd) が使用されます。

***MDY:** 月、日、年の日付形式 (mm/dd/yy) が使用されます。

***DMY:** 日、月、年の日付形式 (dd/mm/yy) が使用されます。

***YMD:** 年、月、日の日付形式 (yy/mm/dd) が使用されます。

***JUL:** 年間通算日の日付形式 (yy/ddd) が使用されます。

DATSEP
SQL ステートメントで使用する日付区切り記号を指定します。

***JOB:** ジョブ属性で指定された日付区切り記号が使用されます。新しい対話式 SQL セッションで *JOB が指定されたときは、現行値が保管され使用されます。ジョブの日付区切り記号に対してあとで変更を加えても、対話式 SQL はその変更を検出しません。

***BLANK:** ブランク ( ) が使用されます。

'/': スラッシュ (/) が使用されます。

'.': ピリオド (.) が使用されます。

',': コンマ (,) が使用されます。

'-': ダッシュ (-) が使用されます。

' ': ブランク ( ) が使用されます。

TIMFMT
SQL ステートメントで使用する時刻形式を指定します。

***HMS:** 時-分-秒の時刻形式 (hh:mm:ss) が使用されます。

***USA:** 米国の時刻形式 (hh:mm xx、ただし、xx は AM か PM) が使用されます。

***ISO:** 国際標準化機構の時刻形式 (hh.mm.ss) が使用されます。

***EUR:** ヨーロッパの時刻形式 (hh.mm.ss) が使用されます。

***JIS:** 日本工業規格の西暦の時刻形式 (hh:mm:ss) が使用されます。

TIMSEP
SQL ステートメントで使用する時刻区切り記号を指定します。

***JOB:** ジョブ属性で指定された時刻区切り記号が使用されます。新しい対話式 SQL セッションで *JOB が指定されたときは、現行値が保管され使用されます。ジョブの時刻区切り記号にあとで変更を加えても、対話式 SQL はその変更を検出しません。

***BLANK:** ブランク ( ) が使用されます。

':': コロン (:) が使用されます。

'.': ピリオド (.) が使用されます。

',': コンマ (,) が使用されます。

' ': ブランク ( ) が使用されます。

DECPNT
使用する小数点の種類を指定します。

***JOB:** SQL で数値定数の小数点として使用される値は、ステートメントを実行するジョブ用に指定されている小数点の表現になります。

***SYSVAL:** 小数点はシステム値から得られます。新しい対話式 SQL セッションで *SYSVAL が指定されたときは、現行値が保管され使用されます。システムの時刻区切り記号に対してあとで変更を加えても、対話式 SQL はその変更を検出しません。

***PERIOD:** ピリオドが小数点になります。

***COMMA:** コンマが小数点になります。

PGMLNG
使用するプログラム言語の構文規則を指定します。このパラメーターを使用するためには、PROCESS パラメーターで *SYN を選択しなければなりません。

***NONE:** 特定言語の構文検査規則は使用されません。

サポートされる言語は次のとおりです。

***C:** 構文検査は C 言語の構文規則にしたがって行われます。

***CBL:** 構文検査は COBOL 言語の構文規則にしたがって行われます。

***PLI:** 構文検査は PL/I 言語の構文規則にしたがって行われます。

***RPG:** 構文検査は RPG 言語の構文規則にしたがって行われます。

***FTN:** 構文検査は FORTRAN 言語の構文規則にしたがって行われます。

SQLSTRDLM
SQL ストリング区切り文字を指定します。このパラメーターを使用するためには、COBOL (*CBL) 文字セットを使用しなければなりません。

***QUOTESQL:** 引用符が SQL ストリング区切り文字になります。

***APOSTSQL:** アポストロフィ (') が SQL ストリング区切り文字になります。

SRTSEQ
「SQL ステートメント入力」画面上で SQL ステートメント内のストリング比較に使用される分類順序表を指定します。

***JOB:** ジョブの SRTSEQ 値が取り出されます。

***JOBRUN:** ユーザーが対話式 SQL を開始するたびに、ジョブの SRTSEQ 値が取り出されます。

***LANGIDUNQ:** LANGID パラメーターで指定した言語用の固有の分類表が使用されます。

***LANGIDSHR:** LANGID パラメーターで指定した言語用の共用分類表が使用されます。

***HEX:** 分類順序表は使用しません。分類順序を決定するには、文字の 16 進値を使用します。

表名は、次のライブラリー値のどれか 1 つで修飾することができます。

***LIBL:** 最初の一致が見つかるまで、ジョブのライブラリー・リスト内のすべてのライブラリーが探索されます。

***CURLIB:** ジョブ用の現行ライブラリーが探索されます。 ジョブ用の現行ライブラリーとして指定されたライブラリーがない場合は、QGPL ライブラリーが使用されます。

library-name: 探索するライブラリーの名前を指定します。

table-name: 対話式 SQL セッションで使用される分類順序表の名前を指定します。

LANGID

SRTSEQ(*LANGIDUNQ) または SRTSEQ(*LANGIDSHR) が指定されたときに使用される言語識別コードを指定します。

***JOB:** ジョブの LANGID 値が取り出されます。

***JOBRUN:** 対話式 SQL が開始されるたびに、ジョブの LANGID 値が取り出されます。

language-ID: 使用される言語識別コードを指定します。

例:

```
STRSQL   PROCESS(*SYN)  NAMING(*SQL)
  DECPNT(*COMMA)  PGMLNG(*CBL)
  SQLSTRDLM(*APOSTSQL)
```

このコマンドを入力すると、対話式 SQL セッションが開始され、SQL ステートメントの構文だけが検査されます。構文検査機能によって使用される文字セットでは、COBOL 言語の構文規則を使用します。このセッションでは SQL 命名規則が使用されます。コンマが小数点として使用され、アポストロフィが SQL ストリング区切り文字として使用されます。

参考文献

以下にリストしている資料には、本書で説明または参照している事項に関する詳しい情報が記載されています。 以下には、正式資料名と資料番号が示されていますが、本書の本文中では、これらの資料は略式名で参照されています。

- バックアップおよび回復の手引き

  この手引きには、バックアップおよび回復の手引きにある情報のサブセットが記載されています。この資料には、バックアップと回復方法の計画、保管と回復手順に使用できる媒体のタイプ、およびディスク回復手順についての説明が記載されています。バックアップからシステムを再導入する方法も紹介されています。
- ファイル管理

  この資料は、アプリケーション・プログラムにおけるファイルの使い方を説明しています。
- DB2 UDB for iSeries データベース・プログラミング

  この手引きには、システムにあるデータベース・ファイルを作成/記述/更新する方法を含む、DB2 UDB for iSeries データベース編成の詳細な説明があります。
- CL プログラミング

  この資料は、DB2 UDB for iSeries のプログラミングに関係する事柄を広範囲にわたって論じています。主なものを挙げると、オブジェクトとライブラリーの全般的な説明、CL プログラミング、プログラム相互間の流れと通信の制御、CL プログラムにおけるオブジェクトの扱い方、および CL プログラムの作成方法などがあります。その他に、事前定義のメッセージと即時メッセージ、ユーザーが定義するコマンドとメニューの取り扱い、定義、および作成の方法、デバッグ・モード、停止点、トレースなどを含むアプリケーションのテスト、および表示機能についても説明しています。
- CL 解説書

  この解説書には、DB2 UDB for iSeries 制御言語 (CL) およびその OS/400 コマンドに関する説明があります。(OS/400 以外のコマンドの説明は、各ライセンス・プログラムに関する資料にあります。) この資料には、サーバー用のすべての CL コマンドの解説、およびこれらのコマンドをコーディングするために必要な構文規則の説明もあります。
- iSeries 機密保護解説書

  この資料には、システム機密保護の概念、機密保護のための計画方法、およびシステムにおける機密保護のセットアップ方法に関する情報が記載されています。システムとデータを正当な権限をもたない人による使用から保護する方法、意図的または偶発的損傷や破壊からデータを保護する方法、機密保護を更新する方法、およびシステムで機密保護をセットアップする方法についても説明しています。
- DB2 UDB for iSeries SQL 解説書

  この解説書には、DB2 UDB for iSeries ステートメントおよびそのパラメーターに関する説明があります。付録では、SQL 連絡域 (SQLCA) と SQL 記述域 (SQLDA) について説明しています。
- IDDU Use

  この手引きには、DB2 UDB for iSeries 対話式データ定義ユーティリティー (IDDU) を使用して、データ・ディクショナリー、ファイル、およびレコードを、システムに対して記述する方法が説明されています。
- WebSphere Development Studio: ILE COBOL プログラマーの手引き

  この手引きには、iSeries システムで、COBOL for iSeries のプログラムを設計/作成/テスト/保守するのに必要な情報が記載されています。
- WebSphere Development Studio: ILE RPG プログラマーの手引き

  この手引きには、iSeries システムで、ILE RPG for iSeries のプログラムを設計/作成/テスト/保守するのに必要な情報が記載されています。
- ILE C for AS/400 Language Reference

この手引きには、iSeries システムで、 ILE C for iSeries のプログラムを設計/作成/テスト/保守するのに必要な情報が記載されています。

- WebSphere Development Studio: ILE C/C++ Programmer's Guide

  この手引きには、iSeries システムで、 ILE C for iSeries のプログラムを設計/作成/テスト/保守するのに必要な情報が記載されています。

- WebSphere Development Studio: ILE COBOL 解説書

  この解説書には、iSeriesCOBOL システムで、COBOL for iSeries プログラムを設計/作成/テスト/保守するのに必要な情報が記載されています。

- REXX/400 Programmer's Guide

  この手引きには、iSeries システムで、REXX/400 プログラムを設計/作成/テスト/保守するのに必要な情報が記載されています。

- DB2 マルチ・システム

  この資料は、分散リレーショナル・データベース・ファイル、ノード・グループ、および区分化の基礎概念を説明しています。この資料には、複数の iSeries システムにわたって区分化されたデータベース・ファイルを作成し使用するために必要な情報が記載されています。情報には、システムの構成方法、ファイルの作成方法、およびアプリケーション内でのそのファイルの使用方法があります。

- Performance Tools for iSeries

  この資料には、プログラマーがシステム、ジョブ、またはプログラムのパフォーマンスに関するデータを収集するのに必要な情報が記載されています。また、パフォーマンス・データの印刷と分析によって既存の非効率性を識別し、訂正するためのヒントも載せられています。管理機能およびエージェント機能に関する情報も含まれています。

- DB2 UDB for iSeries SQL 呼び出しレベル・インターフェース (ODBC)

  この資料には、アプリケーション・プログラマーが DB2 の呼び出しレベル・インターフェースを使ってアプリケーションを作成するのに必要な情報が記載されています。

- IBM Developer Kit for Java

  この資料には、iSeries システムで、Java プログラムを設計/作成/テスト/保守するのに必要な情報が記載されています。また、「iSeries Developer Kit for Java JDBC driver」の章には、JDBC または SQLj を使用して Java プログラムからデータベース・ファイルにアクセスする方法についての説明があります。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

[カ行]

［サ行］

[タ行]

[ナ行]

[ハ行]

［マ行］

[ヤ行]

[ラ行]

[数字]

A

B

C

D

E

F

G

H

I

J

L

N

O

P

Q

R

S

U

W

X

Printed in Japan

日本アイ・ビー・エム株式会社
〒106-8711　東京都港区六本木3-2-12