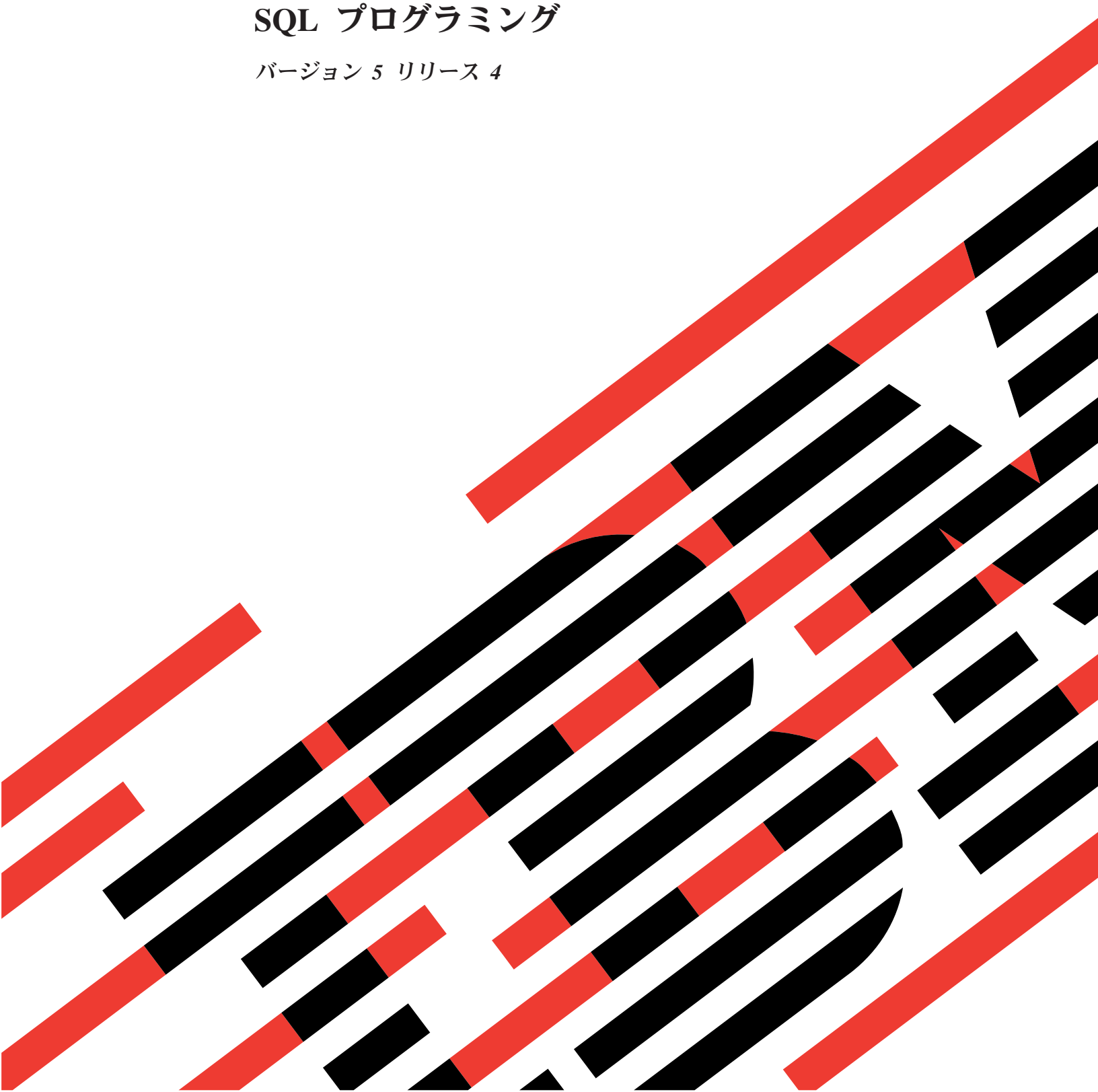
IBM Systems - iSeries

データベース
DB2 Universal Database for iSeries
SQL プログラミング

バージョン 5 リリース 4



IBM Systems - iSeries

データベース
DB2 Universal Database for iSeries
SQL プログラミング

バージョン 5 リリース 4

ご注意！

本書および本書で紹介する製品をご使用になる前に、 327 ページの『特記事項』に記載されている情報をお読みください。

本書は、IBM i5/OS (製品番号 5722-SS1) のバージョン 5、リリース 4、モディフィケーション 0 に適用されます。また、改訂版で断りがない限り、それ以降のすべてのリリースおよびモディフィケーションに適用されます。このバージョンは、すべての RISC モデルで稼働するとは限りません。また CISC モデルでは稼働しません。

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)

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原　典： IBM Systems - iSeries
Database DB2 Universal Database for iSeries SQL programming
Version 5 Release 4

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 2006.2

この文書では、平成明朝体™W3、平成明朝体™W7、平成明朝体™W9、平成角ゴシック体™W3、平成角ゴシック体™W5、および平成角ゴシック体™W7を使用しています。この(書体*)は、(財) 日本規格協会と使用契約を締結し使用しているものです。フォントとして無断複製することは禁止されています。

注* 平成明朝体™W3、平成明朝体™W7、平成明朝体™W9、平成角ゴシック体™W3、
平成角ゴシック体™W5、平成角ゴシック体™W7

目次

SQL プログラミング

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。

このトピック集で示されている SQL ステートメントの例は、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。
- 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。
- 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 プリコンパイラー・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、単一引用符 (') 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。

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。

このトピック集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。

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

関連資料

306 ページの『DB2 UDB for iSeries サンプル表』
このトピックには、このトピック、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。

関連情報

組み込み SQL プログラミング

SQL 解説書

V5R4 の新機能

このトピックでは、V5R4 版のトピック集で加えられた変更点を中心に説明します。

以下の新規トピックが追加されました。

- 69 ページの『再帰的照会の使用』
- 259 ページの『例: 割り振られた SQL 記述子を使用したステートメントの選択』
- 59 ページの『OLAP 指定の使用』
- 195 ページの『INSTEAD OF SQL トリガー』

以下のトピックは、一部変更、および更新されました。

- 割り振られた SQL 記述子に関する情報が、 262 ページの『パラメーター・マーカー』、 251 ページの『SQL 記述域』、 250 ページの『可変リスト SELECT ステートメント』 および 249 ページの『SELECT ステートメントの処理と記述子の使用』 の各トピックに追加されました。
- 51 ページの『SQL ステートメント内の特殊レジスター』 にさらに特殊レジスターが追加されました。
- 6 ページの『SQL ステートメントのタイプ』が変更されました。
- 278 ページの『SQL ステートメント処理プログラムの使用』に、SQL ステートメント処理プログラムで使用可能なステートメントが、さらに追加されました。

新規情報、変更点を確認するには

技術的な変更があった点を確認するには、以下の情報を使用します。

-  イメージは新規、および変更された箇所の開始点を意味します。
-  イメージは新規、および変更された箇所の終了点を意味します。

このリリースでの他の新規、変更点を参照するには、ユーザーへの注意点 (Memo to users) を参照してください。

印刷可能な PDF

これを使用して、この情報の PDF を表示および印刷します。

本書の PDF 版を表示またはダウンロードするには、SQL プログラミングを選択します。

PDF ファイルの保管

表示または印刷のために PDF をワークステーションに保存するには、以下のようにします。

1. ブラウザーで PDF を右マウス・ボタン・クリックする (上部のリンクを右マウス・ボタン・クリック)。
2. PDF をローカルに保管するオプションをクリックする。
3. PDF を保存したいディレクトリーに進む。
4. 「**保存**」をクリックする。

Adobe Reader のダウンロード

これらの PDF を表示または印刷するには、Adobe Reader がシステムにインストールされていなければなりません。 Adobe Web サイト (www.adobe.com/products/acrobat/readstep.html)  から無料でダウンロードできます。

DB2 UDB for iSeries 構造化照会言語の紹介

このトピックでは、DB2 UDB for iSeries および DB2 UDB Query Manager and SQL Development Kit のライセンス・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。組み込み SQL について詳しくは、組み込み SQL (Embedded SQL) を参照してください。

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。

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*) を使用してリモート・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。

関連資料

281 ページの『分散リレーショナル・データベース機能と SQL』
分散リレーショナル・データベース は、相互に接続されたコンピュータ・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。

分散データベース・プログラミング

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 i5/OS™ ライセンス・プログラムの一部であり、このサポートよって、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 プリコンパイラー

  SQL プリコンパイラーは、ホスト言語の組み込み SQL ステートメントのプリ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 ホスト言語プリコンパイラーは、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プリ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。プリコンパイルについて詳しくは、「組み込み SQL プログラミング」の「SQL ステートメントを含むプログラムの準備と実行」を参照してください。プリ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、265 ページの『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。

- 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、278 ページの『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、「組み込み SQL プログラミング」情報の 「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方法」 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・プログラムへのバインド済み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、プリ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、iSeries Information Center のデータベース・セクションで「SQL 呼び出しレベル・インターフェース (ODBC)」トピック集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 API についての詳細は、QSQPRCED を参照してください。API についての一般情報は、i5/OS 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 API についての詳細は、QSQCHKS を参照してください。API についての一般情報は、i5/OS 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 について詳しくは、「DB2 マルチシステム」トピックを参照してください。

- DB2 UDB Symmetric Multiprocessing

  オペレーティング・システムのこの機能により、Query 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・リソース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。 並行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「照会の並列処理の制御」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

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。

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**。一連の列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。	**ビュー**。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**

関連資料

分散データベース・プログラミング

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 (*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 (*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、いくつかの基本タイプがあります。ここでは各タイプの機能ごとに、リストされています。

- SQL スキーマ・ステートメント。データ定義言語 (DDL) ステートメントとしても知られています。
- SQL データおよびデータ変更ステートメント。データ操作言語 (DML) ステートメントとしても知られています。
- 動的 SQL ステートメント
- 組み込み SQL ホスト言語ステートメント

SQL スキーマ・ステートメント

ALTER SEQUENCE
ALTER TABLE
COMMENT ON
CREATE ALIAS
CREATE DISTINCT TYPE
CREATE FUNCTION
CREATE INDEX
CREATE PROCEDURE
CREATE SCHEMA
CREATE SEQUENCE
CREATE TABLE
CREATE TRIGGER
CREATE VIEW
DROP ALIAS
DROP DISTINCT TYPE
DROP FUNCTION
DROP INDEX
DROP PACKAGE
DROP PROCEDURE
DROP SEQUENCE
DROP SCHEMA
DROP TABLE
DROP TRIGGER
DROP VIEW
GRANT DISTINCT TYPE
GRANT FUNCTION
GRANT PACKAGE
GRANT PROCEDURE
GRANT SEQUENCE
GRANT TABLE
LABEL ON
RENAME
REVOKE DISTINCT TYPE
REVOKE FUNCTION
REVOKE PACKAGE
REVOKE PROCEDURE
REVOKE SEQUENCE
REVOKE TABLE

SQL データ・ステートメント

CLOSE
DECLARE CURSOR
DELETE
FETCH
FREE LOCATOR
HOLD LOCATOR
INSERT
LOCK TABLE
OPEN
REFRESH TABLE
SELECT INTO
SET 変数
UPDATE
VALUES INTO

SQL データ変更ステートメント

DELETE
INSERT
UPDATE

SQL 接続ステートメント

CONNECT
DISCONNECT
RELEASE
SET CONNECTION

SQL トランザクション・ステートメント
COMMIT
RELEASE SAVEPOINT
ROLLBACK
SAVEPOINT
SET TRANSACTION

SQL セッション・ステートメント
DECLARE GLOBAL TEMPORARY TABLE
SET CURRENT DEGREE
SET ENCRYPTION PASSWORD
SET PATH
SET SCHEMA
SET SESSION AUTHORIZATION

動的 SQL ステートメント
ALLOCATE DESCRIPTOR
DEALLOCATE DESCRIPTOR
DESCRIBE
DESCRIBE INPUT
DESCRIBE TABLE
EXECUTE
EXECUTE IMMEDIATE
GET DESCRIPTOR
PREPARE
SET DESCRIPTOR

組み込み SQL ホスト言語ステートメント
BEGIN DECLARE SECTION
DECLARE PROCEDURE
DECLARE STATEMENT
DECLARE VARIABLE
END DECLARE SECTION
GET DIAGNOSTICS
INCLUDE
SET OPTION
SET RESULT SETS
SIGNAL
WHENEVER

SQL 制御ステートメント
CALL

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としても) 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。

関連概念

16 ページの『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、順序、カタログ、索引、および別名が含まれます。

40 ページの『データ操作言語』
データ操作言語 (DML) は、データの操作または制御を行えるようにする、SQL の部分を記述します。

関連資料

SQL 解説書

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。

関連概念

SQL 通信域 (SQLCA)

関連資料

SQL エラー戻りコードの処理

SQL 診断域

SQL 診断域は、データベース・マネージャーが維持する、最後に実行された SQL ステートメントに関する一連の情報です。 GET DIAGNOSTICS SQL ステートメントを使用して、プログラムからアクセスできます。

関連概念

GET DIAGNOSTICS ステートメント

関連タスク

SQL 診断域の使用

SQL オブジェクト

SQL オブジェクト とは、スキーマ、ジャーナル、カタログ、表、別名、視点、索引、制約、トリガー、順序、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。

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。

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。

関連概念

ジャーナル処理

コミットメント制御

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。

関連概念

SQL カタログ

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。

マテリアライズ照会表 は、選択ステートメントによって指定される 1 つ以上のソース表から派生するマテリアライズ・データを含むために使用される表です。

区分表 は、表のデータが 1 つ以上のローカル・パーティション (メンバー) に含まれる表です。

関連概念

データ・タイプ

関連資料

22 ページの『マテリアライズ照会表の作成および変更』
照会の結果が表定義の基になっている表をマテリアライズ照会表と呼びます。 そのため、マテリアライズ照会表は通常、その定義の元となる表 (複数可) にあるデータに基づいて事前に計算された結果を含みます。

DB2 マルチシステム

別名

別名とは、表またはビューの代替名のことです。

既存の表またはビューを参照できる場合に、別名を使用して表やビュー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。

関連情報

別名

視点

ビューは、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、ビュー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 つのビューには、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。ビュー内での列の配置は、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 のビューは、特殊な形式のキーなし論理ファイルです。

関連情報

ビュー

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。

関連情報

索引方針の作成

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で許可され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。

関連資料

126 ページの『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。

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またはビュー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。

関連情報

データベース内での自動イベントのトリガー

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、サービス・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 (System/36™) のプログラム、またはプロシージャー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。

関連概念

130 ページの『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。

順序

順序は、固有の数を生成する素早く、簡単な方法を提供するデータ域オブジェクトです。

順序を使用して、IDENTITY 列またはユーザー生成数値列と置き換えることができます。順序はこれらの代替手段と同様に使用します。

関連資料

25 ページの『順序の作成および使用』
順序は、値を素早く簡単に生成できるようにするオブジェクトです。

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。

関連概念

162 ページの『ユーザー定義関数 (UDF) の使用』
書き込み SQL アプリケーションで、いくつかのアクションまたは操作を UDF として、またはアプリケーションのサブルーチンとしてインプリメントすることができます。新規操作をアプリケーションのサブルーチンとしてインプリメントする方がより簡単に見える場合がありますが、代わりに UDF を使用する利点についても考慮することができます。

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。

関連概念

218 ページの『ユーザー定義特殊タイプ (UDT) の使用』
ユーザー定義の特殊タイプは、設定済みの組み込みデータ・タイプよりさらに DB2 UDB の機能を発揮させるメカニズムです。

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 ステートメントがリモート・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 (CRTSQL*xxx*) 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。

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。このトピック集において「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。

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/400 言語の場合は RPG、ILE RPG 言語の場合は RPGI) を表します。

関連資料

281 ページの『分散リレーショナル・データベース機能と SQL』
分散リレーショナル・データベース は、相互に接続されたコンピュータ・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。

関連情報

QSQPRCED

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。

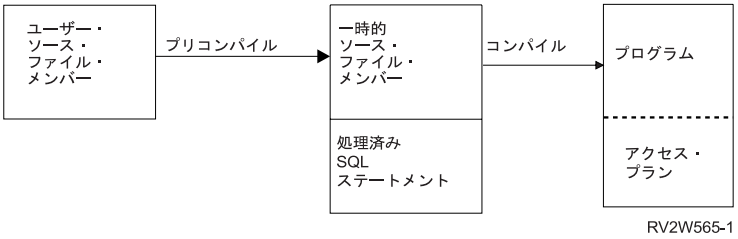
DB2 UDB for iSeries は、ILE 以外のプリコンパイラーおよび ILE プリコンパイラーの両方をサポートします。アプリケーション・プログラムは、分散型でも非分散型でも構いません。

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

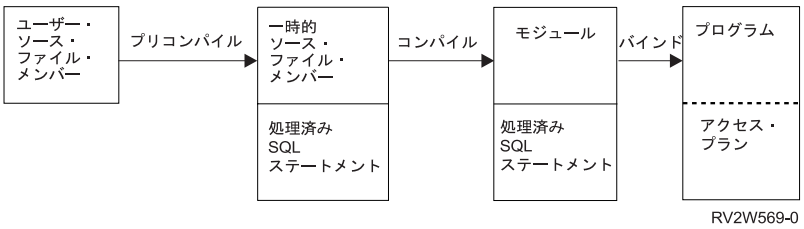
- 元のソース
- ILE プログラムのモジュール・オブジェクト (任意)

- 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

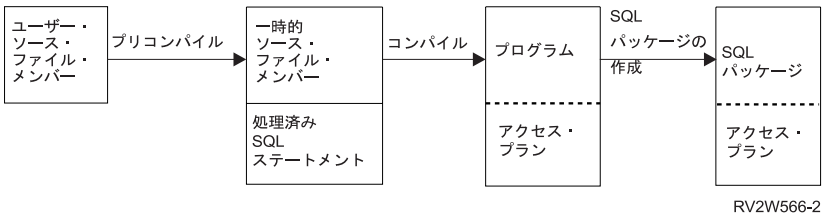
非分散型の非 ILE DB2 UDB for iSeries プログラムでは、管理しなければならないのは、元のソースと結果のプログラムだけです。以下に、非分散型の非 ILE DB2 UDB for iSeries プログラムのプリコンパイルおよびコンパイル・プロセスで生じる、必要なオブジェクトとステップを示します。



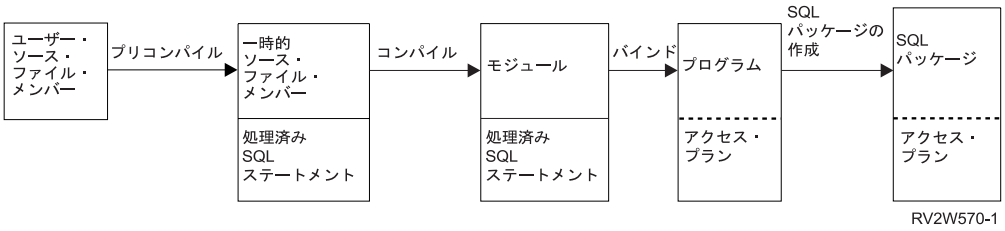
非分散型の ILE DB2 UDB for iSeries プログラムでは、元のソース、モジュール、および結果のプログラムまたはサービス・プログラムを管理する必要があります。以下に、プリコンパイル・コマンドで OBJTYPE(*PGM) が指定された場合に、非分散型の ILE DB2 UDB for iSeries プログラムのプリコンパイルおよびコンパイル・プロセスで生じる、必要なオブジェクトとステップを示します。



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 プログラムのプリ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 プログラムのプリ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。

関連情報

SQL ステートメントを含むプログラムの準備と実行

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 プリコンパイルにより、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、プリコンパイル・プロセスが一時ソース・ファイル QSQLTxxxxx を QTEMP に作成するか、ユーザーが、プリコンパイル・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。 プリコンパイル・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

デフォルト解釈により、プリ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。

関連情報

SQL ステートメントを含むプログラムの準備と実行

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ビュー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ビュー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、表またはビューが見つからないか、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、CRT*xxx*MOD コマンド (あるいは、CRTBND*xxx* コマンド。ここで *xxx* には C、CBL、CPP、または RPG が入る) を使用してソース・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。

関連情報

プログラム作成 (CRTPGM) コマンド

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。

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、順序、カタログ、索引、および別名が含まれます。

関連概念

6 ページの『SQL ステートメントのタイプ』
SQL ステートメントには、いくつかの基本タイプがあります。 ここでは各タイプの機能ごとに、リストされています。

関連情報

SQL を初めて使用する場合

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
```

関連情報

CREATE SCHEMA ステートメント

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。

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: 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 から 9999 の間の整数で、ヌルは許されない
- 記述: 長さ 0 から 24 の文字
- 在庫数量: 0 から 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))
```

関連情報

データ・タイプ

表への制約の追加および除去

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 に置き換えます。

同じ ALTER TABLE ステートメントを使用して制約を除去できます。

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

参照保全および表

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。

次の例を検討してください。

- 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。

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。

関連資料

306 ページの『DB2 UDB for iSeries サンプル表』
このトピックには、このトピック、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。

関連情報

CREATE TABLE

ALTER TABLE

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 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、参照制約を追加または変更します。

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。

関連情報

CREATE TABLE

ALTER TABLE

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,
       FIRSTNME 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。

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 表に関して定義された REPORTS_TO_EXISTS と EMPLOYEE 表に関して定義された 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
```

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。

関連情報

検査保留状態になっている制約の処理

参照制約内の検査保留状況

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。

- デフォルト値
- ヌル可/不可

指定された表またはビュー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 のデフォルトの動作は、EXCLUDING IDENTITY です。 指定される表またはビューが、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
```

関連情報

CREATE TABLE

AS を使用した表の作成

CREATE TABLE AS ステートメントは、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ビュー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 のデフォルトの動作は、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ことを示しています。新規の表 EMPLOYEE3 にデータを入れたい場合は、WITH DATA 文節を組み込みます。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。

関連概念

40 ページの『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を収集するために照会を調整する多様な方法を学習します。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、文節を使用することで特定の方法でデータを収集することができます。

関連情報

CREATE TABLE

マテリアライズ照会表の作成および変更

照会の結果が表定義の基になっている表をマテリアライズ照会表と呼びます。そのため、マテリアライズ照会表は通常、その定義の元となる表 (複数可) にあるデータに基づいて事前に計算された結果を含みます。

最適化プログラムによりマテリアライズ照会表が評価され、基本表を照会するより、マテリアライズ照会表を照会した方が効率がよくなるか判定されます。照会の実行速度が速くなる場合には、マテリアライズ照会表に対して実行されるようになります。マテリアライズ照会表は、直接照会することができます。 最適化プログラムでどのようにマテリアライズ照会表が使用されるかについての詳細は、 データベースのパフォーマンスとQuery 最適化 のトピックを参照してください。

TRANS という名前の非常に大規模なトランザクション表に、会社によって処理されるトランザクションごとに 1 つの行が含まれると仮定します。表は多くの列によって定義されます。以下を発行することによって、トランザクションの日付と量に関する毎日の集計データの入った TRANS 表に対して、マテリアライズ照会表を作成します。

```
CREATE TABLE STRANS
  AS (SELECT YEAR AS SYEAR, MONTH AS SMONTH, DAY AS SDAY, SUM(AMOUNT) AS SSUM
      FROM TRANS
      GROUP BY YEAR, MONTH, DAY )
  DATA INITIALLY DEFERRED
  REFRESH DEFERRED
  MAINTAINED BY USER
```

このマテリアライズ照会表は、DATA INITIALLY DEFERRED 文節を使用して、表の作成時にはデータが取り込まれないように指定します。 REFRESH DEFERRED は、TRANS に対する変更が STRANS には反映されないことを示します。さらに、この表はユーザーによって維持され、ユーザーが ALTER、INSERT、DELETE、および UPDATE を使用できるようにします。

マテリアライズ照会表にデータを取り込む、またはデータが取り込まれた後に表を最新表示するには、REFRESH TABLE ステートメントを使用します。これによって、マテリアライズ照会表に関連付けられている照会が実行され、表には照会の結果が取り込まれます。表 STRANS にデータを取り込むには、以下のステートメントを実行します。

```
REFRESH TABLE STRANS
```

選択ステートメントの結果が既存の表にある列と一致する一連の列を提供する場合 (同じ列の数および互換性のある列定義)、既存の基本表からマテリアライズ照会表を作成できます。たとえば、表 TRANSCOUNT を作成します。その後に、基本表 TRANSCOUNT をマテリアライズ照会表に変更します。

表を作成するには、以下のようにします。

```
CREATE TABLE TRANSCOUNT
    (ACCTID SMALLINT NOT NULL,
     LOCID SMALLINT,
     YEAR DATE
     CNT INTEGER)
```

この表をマテリアライズ照会表に変更できます。

```
ALTER TABLE TRANSCOUNT
  ADD MATERIALIZED QUERY
      (SELECT  ACCTID, LOCID, YEAR, COUNT(*) AS CNT
      FROM TRANS
      GROUP BY ACCTID, LOCID, YEAR )
  DATA INITIALLY DEFERRED
  REFRESH DEFERRED
  MAINTAINED BY USER
```

最後に、マテリアライズ照会表を基本表に戻すことができます。たとえば、次の通りです。

```
ALTER TABLE TRANSCOUNT
      DROP MATERIALIZED QUERY
```

この例では、表 TRANSCOUNT は除去されていませんが、マテリアライズ照会表ではありません。

関連概念

10 ページの『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。

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、他の表の場合と同じように、SELECT、 INSERT、UPDATE、および DELETE の各ステートメントを発行することができます。この表は、次のように DROP TABLE ステートメントを発行して除去することができます。

```
DROP TABLE ORDERS
```

関連情報

DECLARE GLOBAL TEMPORARY TABLE

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 から 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。

値の範囲が大きい場合は、列に対して INTEGER または BIGINT を指定します。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。

関連資料

27 ページの『識別列と順序の比較』
IDENTITY 列と順序は多くの点で類似していますが、違いもあります。

90 ページの『識別列への値の挿入』
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。

94 ページの『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。

ROWID の使用

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)
```

順序の作成および使用

順序は、値を素早く簡単に生成できるようにするオブジェクトです。

順序は、どちらも固有値を生成するという点で識別列と同じです。ただし、順序は表から独立したオブジェクトです。そのため、列には結合されておらず、別個にアクセスされます。さらに、順序はトランザクションの作業単位の一部としては扱われません。

CREATE SEQUENCE ステートメントを使用して順序を作成します。たとえば、識別列の例と同様に、順序 ORDER_SEQ を作成します。

```
CREATE SEQUENCE ORDER_SEQ
START WITH 500
INCREMENT BY 1
MAXVALUE 1000
CYCLE
CACHE 24
```

この順序は、開始値 500 で定義され、使用されるたびに 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、順序の最大値は 1000 です。この値が 1000 に達したら、もう一度 500 から再開されます。

一度この順序が作成されると、順序を使用して列に値を挿入できます。たとえば、順序 ORDER_SEQ の次の値を表 ORDERS の列 ORDERNO と CUSTNO に挿入します。

まず、表 ORDERS を作成します。

```
CREATE TABLE ORDERS
(ORDERNO SMALLINT NOT NULL,
CUSTNO SMALLINT);
```

次に、順序の値を挿入します。

```
INSERT INTO ORDERS (ORDERNO, CUSTNO)
VALUES (NEXT VALUE FOR ORDER_SEQ, 12)
```

以下のステートメントを実行すると、列の値を戻します。

```
SELECT *
FROM ORDERS
```

表 2. 表 ORDERS からの SELECT の結果

ORDERNO	CUSTNO
500	12

この例では、順序 ORDER の次の値が ORDERNO 列に挿入されます。 INSERT ステートメントをもう一度発行します。その後で、SELECT を実行します。

表 3. 表 ORDERS からの SELECT の結果

ORDERNO	CUSTNO
500	12
501	12

PREVIOUS VALUE 式を使用して、順序 ORDER の前の値を挿入することもできます。以下の式で NEXT VALUE および PREVIOUS VALUE を使用できます。

- ステートメントに DISTINCT キーワード、GROUP BY 文節、 ORDER BY 文節、UNION キーワード、 INTERSECT キーワード、または EXCEPT キーワードが含まれていない場合、 SELECT ステートメントまたは SELECT INTO ステートメントの*選択文節* 内
- INSERT ステートメントの VALUES 文節内
- INSERT ステートメントの全選択の*選択文節* 内
- 検索または配置された UPDATE ステートメントの SET 文節内。しかし、NEXT VALUE は SET 文節にある式の副選択の*選択文節* では指定できません。

ALTER SEQUENCE ステートメントを発行することによって順序を変更できます。順序は、以下の方法で変更できます。

- 順序の再始動
- 今後の順序値の増分の変更
- 最小値または最大値の設定または除去
- キャッシュ付き順序番号の数の変更
- 順序が循環できるかどうかを判別する属性の変更
- 要求の順に順序番号を生成しなければならないかどうかの変更

たとえば、順序 ORDER の値の増分を 1 から 5 に変更するには:

```
ALTER SEQUENCE ORDER_SEQ
INCREMENT BY 5
```

この変更が完了した後、INSERT ステートメントをもう一度実行し、その後に SELECT を実行します。表には以下の列が含まれています。

表 4. 表 ORDERS からの SELECT の結果

ORDERNO	CUSTNO
500	12
501	12
528	12

順序が使用する次の値が 528 であることに注意してください。一見すると、この数は誤りであるかに見えます。しかし、この割り当てに至る事象をご覧ください。まず、順序が最初に作成されたとき、キャッシュ値 24 が割り当てられました。システムはこのキャッシュの最初の 24 の値を割り当てます。次に、順序は変更されました。 ALTER SEQUENCE ステートメントが発行されたとき、システムは割り当てられた値を除去し、次の有効値を使用して再始動します。この場合、キャッシュされた元の 24 に、次の増分 5 が足されます。元の CREATE SEQUENCE ステートメントに CACHE 文節がない場合、システムは自動的にデフォルトのキャッシュ値である 20 を割り当てます。順序が変更された場合、次の有効値は 25 です。

関連概念

12 ページの『順序』
順序は、固有の数を生成する素早く、簡単な方法を提供するデータ域オブジェクトです。

関連資料

『識別列と順序の比較』
IDENTITY 列と順序は多くの点で類似していますが、違いもあります。

識別列と順序の比較

IDENTITY 列と順序は多くの点で類似していますが、違いもあります。

どちらを使用するか決定する前に、これらの相違について検討してください。

識別列には以下の特性があります。

- 識別列は、表が作成されたときにのみ表の一部として定義されます。一度表が作成されると、表を変更して識別列を追加することはできません。 (ただし、既存の識別列の特性は変更が可能です。)
- 識別列は自動的に単一表の値を生成します。

- 識別列が GENERATED ALWAYS として定義される場合、使用される値は常にデータベース・マネージャーによって生成されます。表の内容が変更されている間、アプリケーションは独自の値を提供できません。
- IDENTITY_VAL_LOCAL 機能を使用して、識別列に最後に割り当てられた値を見ることができます。

順序には以下の特性があります。

- 順序は、表に結合されていない *DTAARA タイプのシステム・オブジェクトです。
- 順序は任意の SQL ステートメントで使用できる順次値を生成します。
- 順序内の次の値を検索する、およびその順序に割り当てられた前の値を見るために使用できる 2 つの式があります。PREVIOUS VALUE 式は、現行セッションの前のステートメントに指定された順序の、最後に生成された値を戻します。 NEXT VALUE 式は、指定された順序の次の値を戻します。これらの式を使用することによって、複数の表の複数の SQL ステートメントで同じ値を使用できます。

これらは 2 つの項目の特性のすべてではありませんが、これらの特性はデータベースの設計やデータベースで使用するアプリケーションによってどちらを使用すべきかを決定するのに役立つでしょう。

関連資料

24 ページの『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。

25 ページの『順序の作成および使用』
順序は、値を素早く簡単に生成できるようにするオブジェクトです。

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、ビュー名、索引名、順序名、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ことで、これらの名前に対し、より記述的なラベルを作成できます。

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、索引、順序、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'
```

関連情報

LABEL ON

COMMENT ON を使用した SQL オブジェクトの記述

表、ビュー、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、トリガー、または順序のような SQL オブジェクトの作成後、以後の参照用として、ユーザーは情報を提供することができます。COMMENT ON ステートメントを使用することで、情報を追加できます。

情報には、オブジェクトの目的、オブジェクトを使用するユーザー、また、そのオブジェクトについて通常と異なる点や特別な点を含めることができます。表またはビューの各列についても、同様の情報を組み込むことができます。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。通常、注釈は 2000 文字以下ですが、連続では 500 文字以下にする必要があります。

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'
```

関連情報

COMMENT ON

表定義の変更

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 (その長さおよびデフォルト値などの変更)、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。

関連情報

ALTER TABLE

列の追加

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デフォルト値で初期設定されます。NOT NULL を指定する場合は、デフォルト値も指定しなければなりません。

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。

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。

関連情報

ALTER TABLE

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。文字、グラフィック、またはバイナリー列を、固定長から可変長または LOB に、あるいは可変長または LOB から固定長に常に変更できます。

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。

関連資料

『可能な変換』
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。

関連情報

ALTER TABLE

可能な変換

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。

表 5. 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 または UTF-16 グラフィック
DBCS 混用	文字
DBCS 混用	UCS-2 または UTF-16 グラフィック
DBCS 択一	文字
DBCS 択一	DBCS 混用
DBCS 択一	UCS-2 または UTF-16 グラフィック
DBCS 専用	DBCS 混用
DBCS 専用	DBCS グラフィック
DBCS 専用	UCS-2 または UTF-16 グラフィック
DBCS グラフィック	UCS-2 または UTF-16 グラフィック
UCS-2 または UTF-16 グラフィック	文字
UCS-2 または UTF-16 グラフィック	DBCS 混用
UCS-2 または UTF-16 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。

関連資料

30 ページの『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。

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。

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
```

関連情報

ALTER TABLE

ALTER TABLE ステートメントの操作の順序

ALTER TABLE ステートメントはこの一連のステップで示されるように実行されます。

1. 制約の除去。
2. マテリアライズ照会表の除去。
3. 区画情報の除去。
4. RESTRICT オプションが指定されている列の除去。
5. 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)。
6. マテリアライズ照会表の追加および変更。
7. 表への区分化の追加。
8. 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

既存の表またはビューを参照する場合、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。 SQL CREATE ALIAS ステートメントを使用することで、これを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。

- 表またはビュー
- 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
```

関連情報

CREATE ALIAS

視点の作成と使用

ビューを使用すると、1 つまたは複数の表内のデータにアクセスできます。 SELECT ステートメントを使用してビューを作成します。

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ビューを作成するには、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'
```

ビューを作成したら、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。以下の SELECT ステートメントは EMP_MANAGERS の内容を表示します。

```
SELECT *
      FROM CORPDATA.EMP_MANAGERS
```

結果は次のとおりです。

LASTNAME	WORKDEPT
THOMPSON	B01
KWAN	C01
GEYER	E01
STERN	D11
PULASKI	D21
HENDERSON	E11
SPENSER	E21

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、ビューの列リストを指定する必要があります。次の例では、ビューの列は LASTNAME と YEARSOFSERVICE です。

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

このビューの照会の結果は現行の年が変わると変更されるので、ここでは含まれていません。

前述のビューは、選択リスト内で AS 文節を使用してビュー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
```

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 つのビュー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ビューです。

表 6. UNION の結果としてのビューの作成

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

ビューは、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か、 UCS-2 または UTF-16 グラフィック比較に適用されます。

ビュー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 を使用してビューを作成することもできます。

関連概念

40 ページの『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を収集するために照会を調整する多様な方法を学習します。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、文節を使用することで特定の方法でデータを収集することができます。

107 ページの『SQL での分類順序および正規化』
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。 正規化によって文字の結合を含むストリングを比較することができます。

関連資料

75 ページの『副選択結合時の 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。

関連情報

CREATE VIEW

ビューに関する 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、ビュー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、ビューを介して挿入または更新されるすべての行は、ビューの定義に従う必要があります。

WITH CHECK OPTION は、ビューが読み取り専用である場合には指定できません。ビューの定義には、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ビューを作成する場合、ビューに対して実行される挿入および更新操作は、ビュー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、ビューが WITH CHECK OPTION を含む別のビュー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。ビューの定義が使用されないため、ビューの定義に準拠していない行がビュー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、そのビュー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。

関連情報

CREATE VIEW

WITH CASCADED CHECK OPTION:

WITH CASCADED CHECK OPTION 文節は、ビューを通して挿入または更新されるすべての行がビュー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。行がビューの定義に準拠していない場合は、ビュー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、次の更新可能なビュー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、挿入される値がビュー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 を指定して別のビューを作成します。

```
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 つのビュー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 文節は、ある行を更新して、これ以上ビュー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 文節と同じです。これは、ビューが WITH CHECK OPTION 文節を指定しないで定義されたビューに、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ビュー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 番目のビューを作成し、今回は WITH LOCAL CHECK OPTION を指定します。

```
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 ステートメントが、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)
```

V2 に基づいて作成するもう 1 つのビュー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 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。

カスケード検査オプション:

次の例では、カスケード検査オプションの使用方法を説明します。

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 でどのビュー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、ビュー 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ビュー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。

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。コード化ベクトル索引 (EVI) 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。

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 文字フィールドか、UCS-2 または UTF-16 グラフィック・フィールドに適用されます。

関連概念

107 ページの『SQL での分類順序および正規化』
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。 正規化によって文字の結合を含むストリングを比較することができます。

関連情報

CREATE INDEX

索引方針の作成

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。

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。

関連情報

カタログ視点

表に関するカタログ情報の入手

ビュー SYSTABLES には、SQL スキーマ内の各表およびビューにつき 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ビュー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 TABLE_NAME = 'DEPARTMENT'
```

列に関するカタログ情報の入手

ビュー SYSCOLUMNS には、スキーマ内のすべての表およびビューの各列につき 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 TABLE_NAME = '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 COLUMN_NAME, TABLE_NAME, DATA_TYPE, LENGTH, HAS_DEFAULT
  FROM CORPDATA.SYSCOLUMNS
  WHERE TABLE_NAME = '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
- 列がデフォルト値を認めるかどうか

結果は次のようになります。

COLUMN_NAME	TABLE_NAME	DATA_TYPE	LENGTH	HAS_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・オブジェクトのドロップ

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発行します。

```
DROP TABLE EMPLOYEE RESTRICT
```

関連情報

DROP ステートメント

データ操作言語

データ操作言語 (DML) は、データの操作または制御を行えるようにする、SQL の部分を記述します。

関連概念

6 ページの『SQL ステートメントのタイプ』
SQL ステートメントには、いくつかの基本タイプがあります。 ここでは各タイプの機能ごとに、リストされています。

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を収集するために照会を調整する多様な方法を学習します。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、文節を使用することで特定の方法でデータを収集することができます。

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。

関連資料

22 ページの『AS を使用した表の作成』
CREATE TABLE AS ステートメントは、SELECT ステートメントの結果から表を作成します。

33 ページの『視点の作成と使用』
ビューを使用すると、1 つまたは複数の表内のデータにアクセスできます。 SELECT ステートメントを使用してビューを作成します。

基本 SELECT ステートメント

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、このトピックで示されている例よりも多様に対応できます。

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。プリ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。
SELECT ステートメントは、プログラムのカーソルによっても使用できます。最後に、SELECT ステートメントは、動的アプリケーションで準備することができます。

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。

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) には、大/小文字の区別があります。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/小文字が区別されない場合があります。

SELECT ステートメントには、次の項目を指定することができます。

1. 結果に含めたい各列の名前。
2. データが入っている表またはビューの名前
3. 必要な情報が入っている行を識別するための検索条件。
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 表名またはビュー名
  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、およびスカラー全選択を含む) も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。

```
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 7. 照会の結果

LASTNAME	RAISE
NATZ	1421

WHERE 文節を使用する検索条件の指定

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。

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、次のステートメントを出します。

```
... 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 8. 結果表

DEPTNAME	MGRNO
情報センター	000030

検索条件に文字か、UCS-2 または UTF-16 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。

関連概念

107 ページの『SQL での分類順序および正規化』
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。 正規化によって文字の結合を含むストリングを比較することができます。

関連資料

55 ページの『複雑な検索条件の定義』
検索条件には、基本的な比較述部 (=、>、< など) に加え、BETWEEN、IN、EXISTS、IS NULL、LIKE などの述部も含めることができます。

58 ページの『WHERE 文節内の複数の検索条件』
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。

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。

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。
- **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。最もよく使用される式のオペランドは次のとおりです。
  - 定数
  - 列
  - ホスト変数
  - 関数
  - 特殊レジスター
  - スカラー全選択
  - 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
  ```
- スカラー全選択。

検索条件には、AND や OR で区切った複数の述部を指定できます。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。

WHERE 文節を十分に理解するためには、SQL が検索条件と述部を評価し、式の値を比較する順序を知っ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」のトピック集で説明されています。

関連概念

99 ページの『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。副照会は式が使用できる場所であればどこでも使用可能です。

関連資料

55 ページの『複雑な検索条件の定義』
検索条件には、基本的な比較述部 (=、>、< など) に加え、BETWEEN、IN、EXISTS、IS NULL、LIKE などの述部も含めることができます。

式

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、表またはビューの個々の行の特性ではありません。

GROUP BY 文節がない場合、SQL 集約関数のアプリケーションが実行されると、*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。

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。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. 文字か、UCS-2 または UTF-16 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。

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

SUM(PRSTAFF)	MAJPROJ
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、まず部門番号別に行をグループ分けし、次に(各部門内で) 性別でグループ分けします。

関連概念

107 ページの『SQL での分類順序および正規化』
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。 正規化によって文字の結合を含むストリングを比較することができます。

関連資料

48 ページの『ORDER BY 文節』
ORDER BY 文節は戻される選択行について、ユーザーが希望する順番を指定します。 順番はある列の値、または式の値の昇順または降順の照合順序でソートされます。

HAVING 文節

HAVING 文節は GROUP BY 文節に基づいて選択されるグループに対し、検索条件を指定します。

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 文節には集約関数を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 集約関数を使用し、WORKDEPT ごと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。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。

引数が集約関数でない述部は、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。

検索条件に文字か、UCS-2 または UTF-16 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。

関連概念

107 ページの『SQL での分類順序および正規化』
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。 正規化によって文字の結合を含むストリングを比較することができます。

関連資料

233 ページの『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。

ORDER BY 文節

ORDER BY 文節は戻される選択行について、ユーザーが希望する順番を指定します。順番はある列の値、または式の値の昇順または降順の照合順序でソートされます。

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。昇順の照合順序がデフォルト値です。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
```

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 か UTF-16 の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。

関連概念

107 ページの『SQL での分類順序および正規化』
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。 正規化によって文字の結合を含むストリングを比較することができます。

関連資料

44 ページの『GROUP BY 文節』
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。

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。

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ビューに列が追加され、『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 (SQLWARN3 には 『W』 が入ります)。GET DIAGNOSTICS ステートメントを使用する場合、RETURNED_SQLSTATE 項目は値が '01503' になります。

関連資料

233 ページの『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。

ヌル値の処理

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。ヌル値は不明を意味しています。ヌル値は、WHERE および HAVING 文節の中で条件として使用できます。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。ヌル値が入った列を使用する基本的な比較述部は、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。NULL 値を検査するために、IS NULL 述部が使用されます。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
```

結果は次のとおりです。

DEPTNO	DEPTNAME	ADMRDEPT
D01	開発センター	A00
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

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
```

NULL 値を含むことができる値を比較するのに役立つ他の述部は、DISTINCT 述部です。両方の列に等しい非ヌル値が含まれている場合、通常の等価比較 (COL1 = COL2) を使用した 2 つの列の比較は真になります。両方の列がヌルである場合、ヌルは他の値、他のヌル値とさえ決して等しくないので、結果は偽にな

ります。 DISTINCT 述部を使用すると、ヌル値は等しいと考えられます。両方の列に等しい非ヌル値が含まれている場合、および両方の列がヌル値である場合にも、「COL1 は COL2 からの NOT DISTINCT である」は真になります。

たとえば、ヌル値を含む 2 つの表から情報を選択したいとします。最初の表 (T1) には以下の値の入った列 (C1) があります。

C1
2
1
ヌル

2 番目の表 (T2) には以下の値の入った列 (C2) があります。

C2
2
ヌル

以下の SELECT ステートメントを実行します。

```
SELECT *
    FROM T1, T2
    WHERE C1 IS DISTINCT FROM C2
```

結果は次のとおりです。

C1	C2
1	2
1	-
2	-
-	2

NULL 値の使用に関する詳細については、SQL 解説書 トピック集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 DEGREE	データベース・マネージャーが並行して実行すべきタスクの数。

特殊レジスター	内容
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。
SESSION_USER USER	ジョブの実行時権限 ID (ユーザー・プロファイル)。
SYSTEM_USER	データベースに接続されたユーザーの権限 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、リモート・システム側の現在の日付と時刻。
CURRENT DEGREE	リモート・システムで、データベース・マネージャーが並行して実行すべきタスクの数。
CURRENT TIMEZONE CURRENT_TIMEZONE	リモート・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。

特殊レジスター	内容
CURRENT SCHEMA	リモート・システムの現行スキーマ値。
CURRENT PATH CURRENT_PATH CURRENT FUNCTION PATH	リモート・システムの現行パス値。
SESSION_USER USER	リモート・システム側のジョブの実行時権限 ID (ユーザー・プロファイル)。
SYSTEM_USER	リモート・システム上のデータベースに接続されたユーザーの権限 ID (ユーザー・プロファイル)

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。分散表についての詳細は、DB2 マルチシステム (Multisystem) トピック集を参照してください。

データ・タイプのキャスト

式のタイプ 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を基にしたユーザー定義タイプなど、タイプの異なる 2 つの列を比較したい場合、文字を整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 指定を使用して、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
```

関連情報

データ・タイプ間のキャスト

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。

- ANSI/ISO の標準日付、時刻、またはタイム・スタンプ形式に使用される文字ストリングの値。例えば、**DATE** '1950-01-01'。
- 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。

関連情報

データ・タイプ

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。

関連情報

日時の算術計算

重複行の処理

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 キーワードの後に式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/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。

関連概念

107 ページの『SQL での分類順序および正規化』
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。 正規化によって文字の結合を含むストリングを比較することができます。

複雑な検索条件の定義

検索条件には、基本的な比較述部 (=、>、< など) に加え、BETWEEN、IN、EXISTS、IS NULL、LIKE などの述部も含めることができます。

検索条件にはスカラー全選択を含めることができます。

文字か、UCS-2 または UTF-16 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。

複数の検索条件を実行することもできます。

- **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、ある式が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 述部および MIXED データに関する詳細な情報については、iSeries Information Center の「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 '@'
```

LIKE ストリングの最初と最後にあるパーセント文字は、通常の LIKE パーセント文字として解釈されます。'@%' の組み合わせは、パーセント文字として解釈されます。

関連概念

99 ページの『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。副照会は式が使用できる場所であればどこでも使用可能です。

107 ページの『SQL での分類順序および正規化』
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。 正規化によって文字の結合を含むストリングを比較することができます。

関連資料

42 ページの『WHERE 文節を使用する検索条件の指定』
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。

43 ページの『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。

58 ページの『WHERE 文節内の複数の検索条件』
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。

『LIKE に関する特殊な考慮事項』
ここでは LIKE の使用に関する特殊な考慮事項について取り上げます。

関連情報

述部

LIKE に関する特殊な考慮事項:

ここでは LIKE の使用に関する特殊な考慮事項について取り上げます。

- 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'
```

  関連資料

55 ページの『複雑な検索条件の定義』
検索条件には、基本的な比較述部 (=、>、< など) に加え、BETWEEN、IN、EXISTS、IS NULL、LIKE などの述部も含めることができます。

WHERE 文節内の複数の検索条件:

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、DISTINCT、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 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、望む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 日より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 より高い。

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

複数の等価比較を組み合わせる場合、以下に示す例のとおり、AND を用いて述部を記述できます。

```
...
  WHERE WORKDEPT = 'E11' AND EDLEVEL = 12 AND JOB = 'CLERK'
```

また、次の例のように、2 つのリストを比較することもできます。

```
...
  WHERE (WORKDEPT, EDLEVEL, JOB) = ('E11', 12, 'CLERK')
```

2 つのリストが使用される場合、1 つ目のリストの 1 つ目の項目が、2 つ目のリストの 1 つ目の項目と比較されます。比較は双方のリストでこのような方法で行われます。そのため、各リストのエントリー数が同じでなければなりません。リストを使用することは、AND で照会を記述するのと同じです。これらのリストは、等号および不等号比較演算子と共にのみ、使用可能です。

関連資料

55 ページの『複雑な検索条件の定義』
検索条件には、基本的な比較述部 (=、>、< など) に加え、BETWEEN、IN、EXISTS、IS NULL、LIKE などの述部も含めることができます。

42 ページの『WHERE 文節を使用する検索条件の指定』
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。

OLAP 指定の使用

OLAP 指定は、照会の結果行に対するランキング番号と行番号を戻すために使用されます。
RANK、DENSE_RANK、および ROW_NUMBER を指定できます。

例: ランキングおよび行の番号付け

上位 10 名の給与と、そのランキングのリストを作成するとします。以下の照会により、ランキングの番号が生成されます。

```
SELECT EMPNO, SALARY
       RANK() OVER(ORDER BY SALARY DESC),
       DENSE_RANK() OVER(ORDER BY SALARY DESC),
       ROW_NUMBER() OVER(ORDER BY SALARY DESC)
FROM EMPLOYEE
FETCH FIRST 10 ROWS ONLY
```

この照会は次の情報を戻します。

表 9. 前述の照会の結果

EMPNO	SALARY	RANK	DENSE_RANK	ROW_NUMBER
000010	52,750.00	1	1	1
000110	46,500.00	2	2	2
200010	46,500.00	2	2	3
000020	41,250.00	4	3	4
000050	40,175.00	5	4	5
000030	38,250.00	6	5	6

表 9. 前述の照会の結果 (続き)

EMPNO	SALARY	RANK	DENSE_RANK	ROW_NUMBER
000070	36,170.00	7	6	7
000060	32,250.00	8	7	8
000220	29,840.00	9	8	9
200220	29,840.00	9	8	10

この例では、上位 10 名の SALARY が降順で戻されます。 RANK 列は各給与の相対ランキングを示します。2 の位置に同じ給与の行が 2 つあるのがわかります。どちらの行にも同じランキングが割り当てられています。次の行は 4 の値が割り当てられています。RANK は 1 つの行に対し 1 つの値を戻します。この値は対象としている行の前に存在する行の合計数に 1 を加えた数です。重複ができると常に、順序を示す番号に、抜けている数があることになります。

対照的に DENSE_RANK 列では、重複していた 2 位の行の次に 3 という値を表示します。
DENSE_RANK は、対象行の前にある別の内容の行の値に 1 を加えた数を戻します。順序を示す番号は常に連続しています。

ROW_NUMBER は各行に対して固有の番号を戻します。指定された順序に従って値が重複する行では、行番号の割り当ては任意になります。照会が再度実行される際には、重複する行に対して異なる順序で行番号を割り当てることができます。

例: ランキング・グループ

この例では、平均給与が最も高い部署を検索しようとしているとします。次の照会はデータを部署ごとにグループ分けし、各部署ごとに給与の平均を算出して結果の平均値にランク付けを行います。

```
SELECT WORKDEPT, INT(AVG(SALARY)) AS AVERAGE,
         RANK() OVER(ORDER BY AVG(SALARY) DESC) AS AVG_SALARY
   FROM EMPLOYEE
   GROUP BY  WORKDEPT
```

この照会は次の情報を戻します。

表 10. 前述の照会の結果

WORKDEPT	AVERAGE	AVG_SALARY
B01	41,250	1
A00	40,850	2
E01	40,175	3
C01	29,722	4
D21	25,668	5
D11	25,147	6
E21	24,086	7
E11	21,020	8

例: 部署内でのランキング

部署内の従業員をボーナスのランク別でリストしようとしているとします。 PARTITION BY 文節を使用することで、別々に番号付けされるグループを指定することができます。

```
SELECT LASTNAME, WORKDEPT, BONUS,
        DENSE_RANK() OVER(PARTITION BY WORKDEPT ORDER BY BONUS DESC)
          AS BONUS_RANK_IN_DEPT
   FROM EMPLOYEE
   WHERE WORKDEPT LIKE 'E%'
```

この照会は次の情報を戻します。

表 11. 前述の照会の結果

LASTNAME	WORKDEPT	BONUS	BONUS_RANK_in_DEPT
GEYER	E01	800.00	1
HENDERSON	E11	600.00	1
SCHNEIDER	E11	500.00	2
SCHWARTZ	E11	500.00	2
SMITH	E11	400.00	3
PARKER	E11	300.00	4
SETRIGHT	E11	300.00	4
SPRINGER	E11	300.00	4
SPENSER	E21	500.00	1
LEE	E21	500.00	1
GOUNOT	E21	500.00	1
WONG	E21	500.00	1
ALONZO	E21	500.00	1
MENTA	E21	400.00	2

例: 表式の結果によるランキングおよび順序付け

給与が高い上位 5 名の従業員を部署名とともに検索しようとしているとします。部署名は *department* 表に存在するため、結合が必要になります。順序付けはネストされた表の式ですでに完了しているため、ROW_NUMBER 値の判別に順序付けを用いることも可能です。これを実現するためには ORDER BY ORDER OF 表 文節が使用されます。

```
SELECT ROW_NUMBER() OVER(ORDER BY ORDER OF EMP),
               EMPNO, SALARY, DEPTNO, DEPTNAME
   FROM (SELECT EMPNO, WORKDEPT, SALARY
               FROM EMPLOYEE
               ORDER BY SALARY DESC
               FETCH FIRST 5 ROWS ONLY) EMP,
        DEPARTMENT
   WHERE DEPTNO = WORKDEPT
```

この照会は次の情報を戻します。

表 12. 前述の照会の結果

ROW_NUMBER	EMPNO	SALARY	DEPTNO	DEPTNAME
1	000010	52,750.00	A00	SPIFFY コンピューター・サービス事業部
2	000110	46,500.00	A00	SPIFFY コンピューター・サービス事業部
3	200010	46,500.00	A00	SPIFFY コンピューター・サービス事業部

表 12. 前述の照会の結果 (続き)

ROW_NUMBER	EMPNO	SALARY	DEPTNO	DEPTNAME
4	000020	41,250.00	B01	計画
5	000050	40,175.00	E01	サポート・サービス

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。

結合操作の使用上の注意

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。しかし、USING 文節 を結合で使用すると、表名を指定せずに両方の表にある列を識別することができます。
- 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ビュー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か、UCS-2 または UTF-16 グラフィック列に適用されます。

内部結合:

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。

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。この場合、ORPDATA.EMPLOYEE 表の *EMPNO* および *LASTNAME* 列と、CORPDATA.PROJECT 表の *PROJNO* 列を取り出すということになります。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 JOIN 構文トピックを使用した内部結合として実行するには、WHERE 文節で結合条件と追加の選択条件の両方を入力して指定します。

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。

USING 文節を使用したデータの結合:

USING 文節を使用して、結合条件を定義する速記方法を使用できます。 USING 文節は、左の表からの各列を右の表の同じ名前の列と比較する結合条件と同じです。

たとえば、このステートメントの USING 文節をご覧ください。

```
SELECT EMPNO, ACSTDATE
      FROM CORPDATA.PROJACT INNER JOIN CORPDATA.EMPPROJACT
      USING (PROJNO, ACTNO)
      WHERE ACSDATE > '1982-12-31';
```

このステートメントの構文は、以下のステートメントにおける結合条件に有効であり同等です。

```
SELECT EMPNO, ACSTDATE
      FROM CORPDATA.PROJACT INNER JOIN CORPDATA.EMPPROJACT
           ON CORPDATA.PROJACT.PROJNO = CORPDATA.EMPPROJACT.PROJNO AND
              CORPDATA.PROJACT.ACTNO = CORPDATA.EMPPROJACT.ACTNO
      WHERE ACSTDATE > '1982-12-31';
```

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	-
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	-
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 13. 表 A

ACOL1	ACOL2
A1	AA1
A2	AA2
A3	AA3

表 14. 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。

DB22 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。すべての社員と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	PLANNING	PL2100
000060	STERN	製造システム	MA2110
000100	SPENSER	ソフトウェア・サポート	OP2010
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	-

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、ビューの代わりに表式を使用して、ビューの一般使用が不要なときにビュー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

MGRNO	DEPTNO	MAXSAL
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るとします。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 行だけを選択します。

再帰的照会の使用

このトピックでは再帰的共通表式および再帰的視点の使用について解説します。

一部のアプリケーションには、必然と再帰的なデータを取り扱うものがあります。たとえば、部品表 (BOM) アプリケーションは各種部品およびそれを構成する副部品にわたって処理します。たとえば、椅子はシート・ユニットと足部の組み立て部品からできています。シート・ユニットは 1 つのシートと 2 つのアームで構成されている場合があります。これらの部品のいずれも、椅子を組み立てるのに必要なすべての部品がリストされるまで、さらに細かい副部品へと分解できます。このタイプの照会は再帰的共通表式または再帰的ビューを使用して処理することができます。

次に示す旅行プランナーの例では、航空機と列車との接続を使用して、各都市間の移動経路を検索します。以下の表定義とデータが例の中で使用されます。

```
CREATE TABLE FLIGHTS (DEPARTURE CHAR(20),
                      ARRIVAL CHAR(20),
                      CARRIER CHAR(15),
                      FLIGHT_NUMBER CHAR(5),
                      PRICE INT)


INSERT INTO FLIGHTS VALUES('New York', 'Paris', 'Atlantic', '234', 400)
INSERT INTO FLIGHTS VALUES('Chicago', 'Miami', 'NA Air', '2334', 300)
INSERT INTO FLIGHTS VALUES('New York', 'London', 'Atlantic', '5473', 350)
INSERT INTO FLIGHTS VALUES('London', 'Athens'  , 'Mediterranean', '247', 340)
INSERT INTO FLIGHTS VALUES('Athens', 'Nicosia' , 'Mediterranean', '2356', 280)
INSERT INTO FLIGHTS VALUES('Paris', 'Madrid' , 'Euro Air',  '3256', 380)
INSERT INTO FLIGHTS VALUES('Paris', 'Cairo' , 'Euro Air', '63', 480)
INSERT INTO FLIGHTS VALUES('Chicago', 'Frankfurt', 'Atlantic', '37', 480)
INSERT INTO FLIGHTS VALUES('Frankfurt', 'Moscow', 'Asia Air', '2337', 580)
INSERT INTO FLIGHTS VALUES('Frankfurt', 'Beijing', 'Asia Air',  '77', 480)
INSERT INTO FLIGHTS VALUES('Moscow', 'Tokyo', 'Asia Air', '437', 680)
INSERT INTO FLIGHTS VALUES('Frankfurt', 'Vienna', 'Euro Air', '59', 200)
INSERT INTO FLIGHTS VALUES('Paris', 'Rome', 'Euro Air', '534', 340)
INSERT INTO FLIGHTS VALUES('Miami', 'Lima', 'SA Air', '5234', 530)
INSERT INTO FLIGHTS VALUES('New York', 'Los Angeles', 'NA Air', '84', 330)
INSERT INTO FLIGHTS VALUES('Los Angeles', 'Tokyo', 'Pacific Air', '824', 530)
INSERT INTO FLIGHTS VALUES('Tokyo', 'Hong Kong', 'Asia Air', '94', 330)
INSERT INTO FLIGHTS VALUES('Washington', 'Toronto', 'NA Air', '104', 250)

CREATE TABLE TRAINS(DEPARTURE CHAR(20),
                    ARRIVAL CHAR(20),
                    RAILLINE CHAR(15),
                    TRAIN CHAR(5),
                    PRICE INT)

INSERT INTO TRAINS VALUES('Chicago', 'Washington', 'UsTrack', '323', 90)
INSERT INTO TRAINS  VALUES('Madrid', 'Barcelona', 'EuroTrack', '5234', 60)
INSERT INTO TRAINS  VALUES('Washington' , 'Boston' , 'UsTrack', '232', 50)
```

これで表がセットアップされ、航空ネットワークに関する情報を検索するためにデータを照会することができます。シカゴから始めた場合に、フライトのある都市を検索し、その都市にたどり着くまでに必要となるフライトの数を検索すると想定します。以下の照会で、それらの情報が得られます。

```
WITH destinations (origin, departure, arrival, flight_count) AS
    (SELECT a.departure, a.departure, a.arrival, 1
            FROM flights a
            WHERE a.departure = 'Chicago'
     UNION ALL
     SELECT r.origin, b.departure, b.arrival, r.flight_count + 1
            FROM destinations r, flights b
            WHERE r.arrival = b.departure)
SELECT origin, departure, arrival, flight_count
    FROM destinations
```

この照会は次の情報を戻します。

表 15. 前述の照会の結果

ORIGIN	DEPARTURE	ARRIVAL	FLIGHT_COUNT
Chicago	Chicago	Miami	1
Chicago	Chicago	Frankfurt	1
Chicago	Miami	Lima	2
Chicago	Frankfurt	Moscow	2
Chicago	Frankfurt	Beijing	2
Chicago	Frankfurt	Vienna	2
Chicago	Moscow	Tokyo	3
Chicago	Tokyo	Hong Kong	4

この再帰的照会は 2 つの部分で構成されています。共通表式の最初の部分は、*初期化全選択*と呼ばれます。これは共通表式の結果セットの最初の行を選択します。この例では、シカゴから他の場所に直接移動する *flights* 表で、2 行選択します。また、フライトの行程数を、選択する行ごとで 1 に初期化しています。

再帰的照会の 2 番目の部分では、共通表式の現在の結果セットからの行と、オリジナル表からの他の行を結合しています。これは*反復全選択* と呼ばれます。ここが再帰が導入される箇所です。結果セット用にすでに選択されている行は、表名として共通表式の名前を使用し、また列名として共通表式の結果列名を使用して、参照される点に注意してください。

この照会の再帰的な部分は、あらかじめユーザーが選択した各到着都市から、フライトが可能な行程を示す行が元の表から選択されている点です。最初に選択した行の到着都市は新規の出発都市になります。この再帰的選択の行ごとに、目的地へのフライト・カウントが 1 フライト分増えます。これらの新しい行は、共通表式の結果セットに追加され、さらに結果セット行を生成するために反復全選択へと追加されます。最終結果用のデータでは、フライトの合計数が実際に、目的地に到達するまでに必要となった再帰結合 (プラス 1) の合計数であることが分かります。

再帰的ビューは再帰的共通表式に非常によく似ています。先に取り上げた再帰的共通表式を次のような再帰的ビューに書き変えることができます。

```
CREATE VIEW destinations (origin, departure, arrival, flight_count) AS
     SELECT departure, departure, arrival, 1
            FROM flights
            WHERE departure = 'Chicago'
```

```
      UNION ALL
      SELECT r.origin, b.departure, b.arrival, r.flight_count + 1
             FROM destinations r, flights b
             WHERE r.arrival = b.departure)
```

このビュー定義の対話式全選択部はビュー自体を参照しています。このビューからの選択は、先に解説した再帰的共通表式から取得した行と同じ行を返します。

例: 2 つの開始都市

次は、照会をもう少し複雑にするために、シカゴまたはニューヨークのいずれかからフライトを開始した場合の可能目的地とそれにかかる費用を検索しようとしているとします。

```
WITH destinations (departure, arrival, connections, cost) AS
    (SELECT a.departure, a.arrival, 0, price
             FROM flights a
             WHERE a.departure = 'Chicago' OR
                   a.departure = 'New York'
     UNION ALL
     SELECT r.departure, b.arrival, r.connections + 1,
                   r.cost + b.price
             FROM destinations r, flights b
             WHERE r.arrival = b.departure)
SELECT departure, arrival, connections, cost
       FROM destinations
```

この照会は次の情報を戻します。

表 16.

DEPARTURE	ARRIVAL	CONNECTIONS	COST
Chicago	Miami	0	300
Chicago	Frankfurt	0	480
New York	Paris	0	400
New York	London	0	350
New York	Los Angeles	0	330
Chicago	Lima	1	830
Chicago	Moscow	1	1,060
Chicago	Beijing	1	960
Chicago	Vienna	1	680
New York	Madrid	1	780
New York	Cairo	1	880
New York	Rome	1	740
New York	Athens	1	690
New York	Tokyo	1	860
Chicago	Tokyo	2	1,740
New York	Nicosia	2	970
New York	Hong Kong	2	1,190
Chicago	Hong Kong	3	2,070

戻された各行について、結果には出発都市と、最終の目的地都市が示されています。この照会ではフライトの合計数ではなく、必要な接続数をカウントし、すべてのフライトにかかるコストを合計します。

例: 再帰的に使用される 2 つの表

次はシカゴを起点に飛行機に加え、鉄道という別の移動手段を使用して到達できる都市の名前を検索するとします。

以下の照会は次の情報を戻します。

```
WITH destinations (departure, arrival, connections, flights, trains, cost) AS
  (SELECT f.departure, f.arrival, 0, 1, 0, price
          FROM  flights f
          WHERE f.departure = 'Chicago'
   UNION ALL
   SELECT t.departure, t.arrival, 0, 0, 1, price
          FROM trains t
          WHERE t.departure = 'Chicago'
   UNION ALL
   SELECT r.departure, b.arrival, r.connections + 1 , r.flights + 1, r.trains,
             r.cost + b.price
          FROM  destinations r, flights b
          WHERE r.arrival = b.departure
   UNION ALL
   SELECT r.departure, c.arrival, r.connections + 1 ,
             r.flights, r.trains + 1, r.cost + c.price
          FROM destinations r, trains c
          WHERE r.arrival = c.departure)
SELECT departure, arrival, connections, flights, trains, cost
       FROM destinations
```

この照会は次の情報を戻します。

表 17. 前述の照会の結果

DEPARTURE	ARRIVAL	CONNECTIONS	FLIGHTS	TRAINS	COST
Chicago	Miami	0	1	0	300
Chicago	Frankfurt	0	1	0	480
Chicago	Washington	0	0	1	90
Chicago	Lima	1	2	0	830
Chicago	Moscow	1	2	0	1,060
Chicago	Beijing	1	2	0	960
Chicago	Vienna	1	2	0	680
Chicago	Toronto	1	1	1	340
Chicago	Boston	1	0	2	140
Chicago	Tokyo	2	3	0	1,740
Chicago	Hong Kong	3	4	0	2,070

この例では、照会に初期化値を提供する共通表式に、飛行機用と鉄道用の 2 つの部分があります。どの結果行についても、直前の到着位置から次の可能目的地に至るまで、2 つの再帰参照があります。 1 つは飛行機を使って続行した場合で、もう 1 つは鉄道で続行した場合です。最終結果では、必要な接続の数、利用可能な空路の数と、陸路の数が表示されます。

例: DEPTH FIRST および BREADTH FIRST オプション

ここで解説する 2 つの例では、再帰により深さが先に処理されたか、幅が先に処理されたかを基に結果セットの行の順序が異なる点を確認できます。

| **注:** 検索文節は再帰的視点ではサポートされていません。この機能を実行するには再帰的共通表式を含むビュー
| を定義することができます。

| 幅あるいは深さを先に使用して結果を判別するためのオプションは、SEARCH BY 文節に指定した再帰結合列を基にした再帰的関係ソートになります。再帰で幅が先に処理されると、すべての子が先に処理され、続いてその子、さらにその子へと処理が移ります。再帰で深さが先に処理されると、子の全再帰的祖先のチェーンが、次の子に移る前に処理されます。

| これらいずれの場合でも、深さ先行、あるいは幅先行の順序を、再帰プロセスがトラックし続けるために使用する追加の列名を指定します。この列は、行の順番を指定したとおりに戻すため、外部照会の ORDER BY 文節で使用される必要があります。この列が ORDER BY で使用されない場合、DEPTH FIRST または BREADTH FIRST 処理オプションは無視されます。

| SEARCH BY 列で使用する列を選択することは重要です。意味のある結果を出すには、初期化全選択から結合するために、反復全選択で使用される列でなければなりません。この例では、ARRIVAL が使用される列名です。

| 以下の照会は次の情報を戻します。

```
| WITH destinations (departure, arrival, connections, cost) AS
|     (SELECT f.departure, f.arrival, 0, price
|             FROM flights f
|             WHERE f.departure = 'Chicago'
|      UNION ALL
|      SELECT r.departure, b.arrival, r.connections + 1,
|                 r.cost + b.price
|             FROM destinations r, flights b
|             WHERE r.arrival = b.departure)
|     SEARCH DEPTH FIRST BY arrival SET ordcol
| SELECT *
|    FROM destinations
|    ORDER BY ordcol
```

| この照会は次の情報を戻します。

| *表 18. 前述の照会の結果*

DEPARTURE	ARRIVAL	CONNECTIONS	COST
Chicago	Miami	0	300
Chicago	Lima	1	830
Chicago	Frankfurt	0	480
Chicago	Moscow	1	1,060
Chicago	Tokyo	2	1,740
Chicago	Hong Kong	3	2,070
Chicago	Beijing	1	960
Chicago	Vienna	1	680

| この結果データでは、シカゴ-マイアミの行から生成されたすべての目的地が、シカゴ-フランクフルト行の目的地の前にリストされていることが分かります。

| 次に、同じ照会を実行できますが、幅先行で配列した結果を要求できます。

```
| WITH destinations (departure, arrival, connections, cost) AS
|     (SELECT f.departure, f.arrival, 0, price
|             FROM flights f
```

```
            WHERE f.departure='Chicago'
      UNION ALL
      SELECT r.departure, b.arrival, r.connections + 1,
                  r.cost + b.price
            FROM destinations r, flights b
            WHERE r.arrival = b.departure)
    SEARCH BREADTH FIRST BY arrival SET ordcol
SELECT *
  FROM destinations
  ORDER BY ordcol
```

この照会は次の情報を戻します。

表 19. 前述の照会の結果

DEPARTURE	ARRIVAL	CONNECTIONS	COST
Chicago	Miami	0	300
Chicago	Frankfurt	0	480
Chicago	Lima	1	830
Chicago	Moscow	1	1,060
Chicago	Beijing	1	960
Chicago	Vienna	1	680
Chicago	Tokyo	2	1,740
Chicago	Hong Kong	3	2,070

この結果データでは、シカゴから直接到着できる目的地がすべて、接続フライトの前にリストされていることが分かります。このデータは先に実行した照会の結果と同一ですが、順序が幅先行になります。

例: 循環

再帰的プロセスで重要なことは、再帰的プログラミング・アルゴリズム、再帰的データの照会のいずれであっても、再帰は有限でなければならないということです。有限でない場合、無限ループに入ることになります。 CYCLE オプションは、循環データに対する保護機能を備えています。このオプションでは、繰り返しの循環を終了させるだけでなく、循環データを検出しやすくする循環マーク標識を出力することも選択できます。

注: 循環文節は再帰的視点ではサポートされていません。再帰的共通表式を含むビューを定義すると、この機能を実行できます。

最後の例では、データで循環が起きていると想定します。表にもう 1 行追加することで、カイロからパリへのフライトと、パリからカイロへのフライトができました。この例のように、故意に循環データを作り出そうとしなくても、データの処理時に無限ループに入る照会が生成されるのは、よくあることです。

以下の照会は次の情報を戻します。

```
INSERT INTO FLIGHTS VALUES('Cairo', 'Paris', 'Euro Air', '1134', 440)


WITH destinations (departure, arrival, connections, cost, itinerary) AS
    (SELECT f.departure, f.arrival, 1, price,
                CAST(f.departure CONCAT f.arrival AS VARCHAR(2000))
            FROM flights f
            WHERE f.departure = 'New York'
      UNION ALL
      SELECT r.departure, b.arrival, r.connections + 1 ,
                  r.cost + b.price, CAST(r.itinerary CONCAT b.arrival AS VARCHAR(2000))
```

```
            FROM destinations r, flights b
            WHERE r.arrival = b.departure)
    CYCLE arrival SET cyclic_data TO '1' DEFAULT '0'
SELECT departure, arrival, itinerary, cyclic_data
      FROM destinations
      ORDER BY cyclic_data
```

この照会は次の情報を戻します。

表 20. 前述の照会の結果

DEPARTURE	ARRIVAL	ITINERARY	CYCLIC_DATA
New York	Paris	New York Paris	0
New York	London	New York London	0
New York	Los Angeles	New York Los Angeles	0
New York	Madrid	New York Paris Madrid	0
New York	Cairo	New York Paris Cairo	0
New York	Rome	New York Paris Rome	0
New York	Athens	New York London Athens	0
New York	Tokyo	New York Los Angeles Tokyo	0
New York	Paris	New York Paris Cairo Paris	1
New York	Nicosia	New York London Athens Nicosia	0
New York	Hong Kong	New York Los Angeles Tokyo Hong Kong	0

この例では、データ内の循環を検出するために使用する列として、ARRIVAL 列が CYCLE 文節に定義されています。循環が見つかると、特殊な列 (このケースでは CYCLIC_DATA) には、結果セットの循環行として、文字値「1」がセットされます。その他の行にはデフォルト値である「0」が入っています。ARRIVAL 列に循環が見つかると、処理はそれ以上データの処理を行わず、無限ループは発生しません。ご使用のデータに実際に循環参照があるかを確認するには、外部照会で CYCLIC_DATA 列を参照してください。

副選択結合時の 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。選択文節をコード化するときには、異なる文節および手法を使用できます。

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
```

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。

CORPDATA.EMPLOYEE からの EMPNO
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。

CORPDATA.EMPPROJACT からの EMPNO
000230
000230
000240
000230
000230
000240
000230
000150
000170
000190

CORPDATA.EMPPROJACT からの EMPNO
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、ビュー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。

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。

関連概念

107 ページの『SQL での分類順序および正規化』
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。 正規化によって文字の結合を含むストリングを比較することができます。

関連資料

33 ページの『視点の作成と使用』
ビューを使用すると、1 つまたは複数の表内のデータにアクセスできます。 SELECT ステートメントを使用してビューを作成します。

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
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。以下はその例です。

```
(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。

EXCEPT キーワードの使用

EXCEPT キーワードは、2 番目の副選択から任意の一致する行を引いた最初の副選択の結果セットを戻します。

次のものを含む従業員数のリストを検出したいとします。

- 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を*引いた* 社員

この照会は、プロジェクト MA2112、MA2113、および AD3111で働いていない 部門 D11 のすべての社員を戻します。

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

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。

CORPDATA.EMPLOYEE からの EMPNO
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。

CORPDATA.EMPPROJACT からの EMPNO
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、最初の中間結果表を取り、2 番目の中間結果表にも表示されるすべての行を除去し、重複行を除去し、結果を順序付けます。

```
SELECT EMPNO
     FROM CORPDATA.EMPLOYEE
     WHERE WORKDEPT = 'D11'
EXCEPT
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
000200
000220
200170

EMPNO
200220

INTERSECT キーワードの使用

INTERSECT キーワードは、両方の結果セットにあるすべての行が結合された結果セットを戻します。

次のものを含む従業員数のリストを検出したいとします。

- 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

INTERSECT は両方の結果セットにあるすべての従業員数を戻します。言い換えると、この照会はプロジェクト MA2112、MA2113、および AD3111 で働いている部門 D11 のすべての従業員を戻します。

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

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。

CORPDATA.EMPPROJACT からの EMPNO
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、最初の中間結果表を取り、2 番目の中間結果表と比較し、任意の重複行を引いた両方の表にある行を戻し、結果を順序付けます。

```
SELECT EMPNO
     FROM CORPDATA.EMPLOYEE
     WHERE WORKDEPT = 'D11'
INTERSECT
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
000150
000160
000170
000180
000190

EMPNO
000210

データ取り出しエラー

ステートメントの実行の際、エラーが生じる場合があります。

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、以下の処理を行います。

- 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、または SQL 診断域の RETURNED_SQLSTATE を '01004' 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。

複数行 FETCH におけるデータ・マッピング・エラーの場合、警告 SQLSTATE として報告される各マッピング・エラーには、SQL 診断域に別個の条件域があります。SQL は最初のエラーで停止するので、エラー SQLSTATE として報告される 1 つのマッピング・エラーのみが SQL 診断域に戻されることに注意してください。

他の SQL ステートメントの場合、最後の警告 SQLSTATE のみが SQL 診断域に戻されます。

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、文字が変換できない場合。
  - 式が 10 進数列であり、有効でない数値が検出された場合。

INSERT ステートメントを使用した行の挿入

このトピックでは、表および視点内のデータの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。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。

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ビューに新規の行を追加することができます。

- 追加される列について、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、別の表またはビュー内のどのデータを新しい行に入れるかを SQL に指示する。
- 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。

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 (その列にデフォルト値がない場合)。表またはビュー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。
- **スカラー全選択。**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デフォルト値を持つか、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。ある列にデフォルト値が入る場合には、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、その列にデフォルト値が入れられます。

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、またはデフォルト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、デフォルト値が使用されます。列がデフォルト値を持つように定義されている場合は、列にデフォルト値が入ります。明示のデフォルト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・タイプについてのデフォルト値を列に入れます。列用に定義されたデフォルト値がないが、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、デフォルト値は 0 です。
- 固定長文字またはグラフィック列の場合、デフォルト値はブランクです。
- 固定長バイナリー列の場合、デフォルトは 16 進数のゼロです。
- 可変長文字、グラフィック列、バイナリー列の場合、または LOB 列の場合、デフォルトは長さが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、デフォルト値は現在の日付、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、デフォルト値の日付/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デフォルト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、デフォルト値は DLVALUE(’’,’URL’,’’) に対応する値になります。
- 特殊タイプ列の場合、デフォルト値は対応するソース・タイプのデフォルト値になります。
- ROWID 列、または AS IDENTITY で定義された列の場合、データベース・マネージャーがデフォルト値を生成します。

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、SQLCA の SQLERRD(3) に反映されます。 GET DIAGNOSTICS ステートメントの ROW_COUNT 診断項目からも使用可能です。

関連情報

INSERT ステートメント

VALUES キーワードを使用した行の挿入

VALUES キーワードを使用して、表に単一行または複数行を挿入することができます。

この例では DEPARTMENT 表に新規行を挿入します。新規行の列は次のようになります。

- 部門番号 (DEPTNO) は 'E31'。
- 部門名 (DEPTNAME) は 'ARCHITECTURE'。
- 管理者番号 (MGRNO) は '00390'。
- (ADMRDEPT) 部門への報告書は 'E01'。

この新規行に対する INSERT ステートメントは次のようになります。

```
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, MGRNO, ADMRDEPT)
   VALUES('E31', 'ARCHITECTURE', '00390', 'E01')
```

VALUES 文節を使用して、表に複数行を挿入することもできます。次の例は PROJECT 表に 2 つの行を挿入します。プロジェクト番号 (PROJNO)、プロジェクト名 (PROJNAME)、部門番号 (DEPTNO)、および担当従業員 (RESPEMP) の値が値リストに与えられます。プロジェクト開始日 (PRSTDATE) の値は現在日付を使用します。列リストにリストされていない表の列の残りには、デフォルト値が割り当てられます。

```
INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTDATE)
   VALUES('HG0023', 'NEW NETWORK', 'E11', '200280', CURRENT DATE),
         ('HG0024', 'NETWORK PGM', ''E11", '200310', CURRENT DATE)
```

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、選択ステートメントの結果表から 0 行、1 行、または 2 行以上を表に挿入することができます。

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、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。

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。

注:

1. 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。
2. 選択ステートメントで指定した 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。
3. 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。この値は、GET DIAGNOSTICS ステートメントの ROW_COUNT 診断項目からも使用可能です。
4. 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。

ブロック化 INSERT ステートメントを使用した表への複数行の挿入

ブロック化 INSERT ステートメントを使用すると、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。

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短い整数フィールド 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、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でサポートされます。

関連情報

組み込み SQL プログラミング

参照制約付き表へのデータの挿入

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)。

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。

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')
```

識別列への値の挿入

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、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。

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。

関連資料

24 ページの『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。

関連情報

IDENTITY_VAL_LOCAL

UPDATE ステートメントを使用した表内のデータの変更

このトピックでは、表および視点内のデータの更新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。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。
- **スカラー全選択**。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。列の現行値を列のデフォルト値で置換します。列は、その列用に定義されたデフォルト値を持つか、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、表またはビューの各行が、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、表内のどの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。この値は、GET DIAGNOSTICS ステートメントの ROW_COUNT 診断項目からも使用可能です。

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'
```

関連情報

UPDATE ステートメント

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。

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、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。

関連資料

117 ページの『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。

118 ページの『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、更新、挿入、DDL、削除操作のようなデータベース変更のグループを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。

例: UPDATE 規則:

これらの例で UPDATE 規則について解説します。

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。

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、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'
```

関連資料

24 ページの『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。

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。

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、すべての列を更新できます。

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。次の例を検討してください。

表 21. 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。
... 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。	

表 21. 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
CLOSE-THISEMP. EXEC SQL **CLOSE** THISEMP END-EXEC.	

関連資料

233 ページの『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。

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。ビュー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ビュー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。

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、SQLCA の SQLERRD(3) に反映されます。この値は、GET DIAGNOSTICS ステートメントの ROW_COUNT 診断項目からも使用可能です。

関連情報

DROP ステートメント

DELETE ステートメント

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 ステートメント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。 従属行の検査は直ちに実行されます。

  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デフォルト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デフォルト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デフォルト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デフォルト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デフォルト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。SQLERRD(3) の値は、 GET DIAGNOSTICS ステートメントの ROW_COUNT 項目からも使用可能です。 SQLERRD(5) の値は、DB2_ROW_COUNT_SECONDARY 項目からも使用可能です。

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'
```

306 ページの『DB2 UDB for iSeries サンプル表』に記載されているような表とデータがある場合、まず DEPARTMENT 表から 1 行が削除され、続いて、EMPLOYEE 表が更新されて、値が 'E11' の箇所で WORKDEPT の値がデフォルトに設定されます。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	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

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 表で、影響を受ける一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
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

関連資料

306 ページの『DB2 UDB for iSeries サンプル表』
このトピックには、このトピック、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。

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。副照会は式が使用できる場所であればどこでも使用可能です。

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。

関連資料

43 ページの『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。

55 ページの『複雑な検索条件の定義』
検索条件には、基本的な比較述部 (=、>、< など) に加え、BETWEEN、IN、EXISTS、IS NULL、LIKE などの述部も含めることができます。

SELECT ステートメントの副照会

副照会は検索条件をさらに絞り込むのに役立ちます。

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

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。

副照会の使用上の注意:

副照会を使用するとき、ユーザーはこれらの使用上の注意を知っておく必要があります。

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 から 255 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. 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 以上になります。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。
3. 以下の述部の場合、副照会に対し、行の全選択が使用されます。つまり、副照会は 1 行に複数の値を戻すことができることを示します。
   - 基本述部の等号比較、あるいは不等号比較。
   - =ANY、=ALL、および =SOME を使用する比較述部
   - IN および NOT IN 述部

   行全選択が使用される場合は、以下のようにします。

- 選択リストに SELECT * を含めるべきではありません。値は明示的に指定する必要があります。
- 行全選択は行の式と比較しなければなりません。行の式とは括弧で囲まれた値のリストです。行の式にある数と、副照会から返された値とは同じ数でなければなりません。
- IN または NOT IN 述部の行の式には、タイプなしパラメーター・マーカーを入れることはできません。 CAST を使用して、これらのパラメーター・マーカーに結果のデータ・タイプを提供します。
- 副照会には、UNION、EXCEPT、または INTERSECT あるいは相関参照を持つことはできません。

4. 副照会には ORDER BY、FOR READ ONLY、 FETCH FIRST *n* ROWS、UPDATE、OPTIMIZE 文節を含めることはできません。

WHERE 文節または HAVING 文節に副照会を組み込む:

ここでは、WHERE 文節または HAVING 文節に副照会を組み込む方法をいくつか示します。

- 基本比較
- 限量化比較 (ALL、ANY、および SOME)
- IN キーワード
- 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のは 1 行のみです。等号演算子または不等号演算子が使用されている場合、行に対し複数の値を返す場合があります。 SQL は副照会行の各値を比較演算子の他方にある対応する値と比較します。 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 つの行を返すほかに、まったく行を返さない場合もあります。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、返される行は 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**すべての行**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各行の結果より大きい (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、行が 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行のうち少なくとも *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行の少なくとも 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。

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行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行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 つ以上の行を返したかどうかを検査します。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 日より後の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ません。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。

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 ...)
```

相関副照会

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ができます。 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、表の名前を提供します。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、表名の後に相関名を入れます。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 で定義し、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、次のような値が入ります。

表 25. 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、次のような値が入ります。

表 26. 前述の照会の結果のセット

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
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	業務部	EILEEN W HENDERSON
E21	ソフトウェア・サポート	THEODORE Q SPENSER

例: UPDATE ステートメントの相関副照会:

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。

例: DELETE ステートメントの相関副照会:

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。

SQL での分類順序および正規化

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。正規化によって文字の結合を含むストリングを比較することができます。

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や、UCS-2 および UTF-16 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。

関連資料

33 ページの『視点の作成と使用』
ビューを使用すると、1 つまたは複数の表内のデータにアクセスできます。 SELECT ステートメントを使用してビューを作成します。

38 ページの『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。

42 ページの『WHERE 文節を使用する検索条件の指定』
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。

44 ページの『GROUP BY 文節』
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。

47 ページの『HAVING 文節』
HAVING 文節は GROUP BY 文節に基づいて選択されるグループに対し、検索条件を指定します。

48 ページの『ORDER BY 文節』
ORDER BY 文節は戻される選択行について、ユーザーが希望する順番を指定します。 順番はある列の値、または式の値の昇順または降順の照合順序でソートされます。

54 ページの『重複行の処理』
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。

55 ページの『複雑な検索条件の定義』
検索条件には、基本的な比較述部 (=、>、< など) に加え、BETWEEN、IN、EXISTS、IS NULL、LIKE などの述部も含めることができます。

75 ページの『副選択結合時の 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。

関連情報

分類順序

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確認するには、このトピックに記載されている STAFF 表に対して例を実行します。

JOB 列の値は、大/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 27. 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

表 27. STAFF 表 (続き)

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
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 を使用する固有重み分類順序

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

ORDER BY を指定した分類順序の仕組みを説明します。

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
```

以下の表では *HEX 分類順序を使用した場合の結果を示しています。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 28. *HEX 分類順序を使用した場合の結果*

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

以下の表では、固有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。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 29. ENU 言語 ID に固有重み分類順序を使用した場合の結果

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
70	Rothman	15	Sales	7	16502.83	1152.00
60	Quigley	38	SALES	0	16808.30	650.25

以下の表では、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。この表では、「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 30. ENU 言語 ID に共用重み分類順序を使用した場合の結果

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

行選択を指定した分類順序の仕組みを説明します。

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'
```

最初の表は、*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。列 JOB の行選択基準に一致する行が、選択ステートメントで指定されたとおり、正確に選択されています。すなわち、大文字の 'MGR' のみが選択されています。

*表 31. *HEX 分類順序を使用した場合の結果*

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
30	Merenghi	38	MGR	5	17506.75	0

表 2 は、固有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。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 32. ENU 言語 ID に固有重み分類順序を使用した場合の結果

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
30	Merenghi	38	MGR	5	17506.75	0

以下の表では、共用重み分類順序を使用した場合に行の選択がどのように行われるかを示しています。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。すべての値「mgr」、「Mgr」、および「MGR」が選択されていることに注意してください。

表 33. ENU 言語 ID に共用重み分類順序を使用した場合の結果

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

ビューは、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 文節でビューが参照されると、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、ビュー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 (文字か、UCS-2 または UTF-16 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ビュー 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 34. "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

ビュー V1 に対して実行されるすべての照会は、上記の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 35. "SELECT * FROM V1 WHERE JOB = 'MGR'" ENU 言語 ID に対する固有重み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、文字キーや、UCS-2 および UTF-16 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、文字か、UCS-2 または UTF-16 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。

関連情報

分類順序と一緒に索引を使用する

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。

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。

ICU 分類順序

ICU (International Components for Unicode) 分類順序表が使用される場合、表のロケールに対し、指定された言語の特定の規則にしたがって、データの重みを決定するため、データベースはシステムの ICU サポート (オプション 39) を使用します。

en_us (米国英語ロケール) と指定された ICU 分類順序表は、たとえば fr_FR (フランス語ロケール) と指定された他の ICU 表とは異なったデータの分類ができます。

システムの ICU サポートは、正規化されていないデータを適切に処理し、データが正規化されている場合と同じ結果を出します。システムの ICU 分類順序表は、すべての文字、グラフィック、およびユニコード (UTF-8、UTF-16、および UCS-2) データを分類できます。

たとえば、 NAME と指定された UTF-8 文字列には次の 3 つの名前が含まれています (列の 16 進値も提供されます)。

NAME	HEX (NAME)
Gómez	47C3B36D657A
Gomer	476F6D6572
Gumby	47756D6279

*HEX 分類順序は次のように NAME 値を配列します。

NAME
Gomer
Gumby
Gómez

en_us と指定された ICU 分類順序表は NAME 値を正確に配列します。

NAME
Gomer
Gómez
Gumby

ICU 分類順序表が指定されるとき、その表を使用する SQL ステートメントのパフォーマンスは、非 ICU 分類順序表または *HEX 分類順序を使用する場合よりもはるかに遅くなることがあります。このパフォーマンスの低下は、システムの ICU サポートを呼び出して、分類する必要のあるデータの各部分の重み付きの値を得るために生じます。 ICU 分類順序表は、分類機能を強化しますが、 SQL ステートメントの実行が遅くなるという代償を払うことになります。しかし、ICU 分類順序表によって作成される索引は、列を超えて作成され、システムの ICU サポートを呼び出す必要がなくなります。この場合、索引キーにはすでに ICU の重み付きの値が含まれているので、システムの ICU サポートを呼び出す必要はありません。

関連情報

International Components for Unicode

正規化

正規化によって文字の結合を含むストリングを比較することができます。

UTF-8 または UTF-16 CCSID によってタグ付けされたデータには、結合文字を含めることができます。文字の結合によって、生じる文字を複数の文字を複合したものにできます。複合文字の先頭文字の後に、データ・ストリング内でウムラウトやアクセントなどの様々な非スペース文字の 1 つを続けることができます。結果の文字がすでに文字セットで定義されている場合、ストリングを正規化すると、複数の結合文字が定義済み文字の値によって置き換えられます。たとえば、ストリングに「a」という文字が含まれていて「..」が続く場合、ストリングは単一文字の「ä」を含むように正規化されます。

正規化を行うと、ストリングを正確に比較できるようになります。データが正規化されない場合、表示上では同一に見えても 2 つのストリングが保管されている表記は異なる場合があるため、同等のものとしては比較されない可能性があります。 UTF-8 および UTF-16 ストリング・データが正規化されていない場合、表の列にはウムラウト文字が続く文字「a」の入った行と、結合された「ä」文字の入った行が別々に存在する可能性があります。これらの 2 つの値は、両方とも比較述部 (WHERE C1 = 'ä') で同等とは比較されません。この理由で、表のすべてのストリング列を正規形で保管することをお勧めします。

挿入または更新前にユーザー自身がデータを正規化するか、またはデータベースによって自動的に正規化が行われるように表の列を定義することができます。データベースに正規化を実行させるには、列定義の一部として NORMALIZED を指定します。このオプションは、1208 (UTF-8) または 1200 (UTF-16) の CCSID でタグ付けされた列でのみ可能です。データベースは表のすべての列が正規化されたと想定します。

NORMALIZED 文節は、関数およびプロシージャー・パラメーターにも指定できます。入力パラメーターに指定される場合、関数またはプロシージャーを呼び出す前に、パラメーター値で指定されたデータベースによって正規化が行われます。出力パラメーターに指定される場合、その文節は実行されません。この場合は、ユーザーのルーチン・コードが正規化された値を戻すと想定されます。

QAQQINI ファイルの NORMALIZE_DATA オプションは、システムが UTF-8 および UTF-16 データを処理するときに、正規化を実行するかどうかを指定するために使用されます。このオプションは、ストリングを結合するリテラル、ホスト変数、パラメーター・マーカー、および式を、システムが SQL で使用する前に正規化するかどうかを制御します。このオプションは、正規化を実行しないように初期設定されていま

す。表のデータおよびアプリケーションのリテラル値が、他のメカニズムによって常にすでに正規化されているか、または正規化が必要な文字を含んでいない場合には、これは適切な値です。このような場合、照会でシステムの正規化を行う場合のオーバーヘッドを避けてください。データが正規化されていない場合、このオプションの値を切り替えて、システムに正規化を実行させてください。

関連情報

照会属性の変更 (CHGQRYA) コマンドによる照会の属性の変更

データ保護

このトピックでは、SQL 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。

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。

SQL の GRANT ステートメントおよび REVOKE ステートメントは、SQL の各関数で、SQL パッケージ、SQL プロシージャー、特殊タイプ、順序、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。

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) 命名の場合のデフォルト値です。
- 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 がデフォルト値です。

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。

関連情報

iSeries 機密保護解説書 (PDF)

GRANT (表またはビュー特権)

REVOKE (表またはビュー特権)

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、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

ビューは、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。ビューは、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。ビューは 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。

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。

関連情報

iSeries 機密保護解説書 (PDF)

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。

並行性

並行性とは、同じ表またはビュー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。

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。このため、別の FETCH が実行されるまで、UPDATE または DELETE ステートメントが最初のカーソルを参照するのを防ぎます。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獲得を防御しません。

デフォルト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、デフォルトのレコード待機時間 (60 秒) およびデフォルト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、ジョブのデフォルト待機時間 (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。 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。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 文節を含むビューである場合。
  - 照会に、GROUP BY 文節の入っているビューを参照する 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。

関連情報

LOCK TABLE ステートメント

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 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。

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。

関連資料

93 ページの『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。

関連情報

ジャーナル処理

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、更新、挿入、DDL、削除操作のようなデータベース変更のグループを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 SCHEMA
- 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 は単にデフォルトのロック・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・レベルとがデフォルトのロック・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。

プログラムのプリコンパイル時に 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 500,000,000 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 システム上にないリモート・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 システム上にないリモート・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 36. 行のロック期間

SQL ステートメント	COMMIT パラメーター (注 5 を参照)	行ロックの期間	ロック・タイプ
SELECT INTO	*NONE	ロックなし。	
SET 変数	*CHG	ロックなし。	
VALUES INTO	*CS (注 6 を参照)	読み取りおよび解放時に行がロックされる。	READ
	*ALL (注 2 および注 7 を参照)	読み取りから次の ROLLBACK または COMMIT まで。	READ
FETCH (読み取り専用カーソル)	*NONE	ロックなし。	
	*CHG	ロックなし。	
	*CS (注 6 を参照)	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
	*ALL (注 2 および注 7 を参照)		READ

表 36. 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 5 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。読み取りから	UPDATE
	*ALL	次の ROLLBACK または COMMIT まで。	UPDATE
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[3]
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	UPDATE
	*CS	COMMIT まで。読み取りから次の ROLLBACK または	UPDATE
	*ALL	COMMIT まで。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	UPDATE
	*CS	COMMIT まで。読み取りから次の ROLLBACK または	UPDATE
	*ALL	COMMIT まで。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	UPDATE
	*CS	COMMIT まで。読み取りから次の ROLLBACK または	UPDATE
	*ALL	COMMIT まで。読み取りから次の ROLLBACK または COMMIT まで。	UPDATE
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	UPDATE
	*CS	COMMIT まで。読み取りから次の ROLLBACK または	UPDATE
	*ALL	COMMIT まで。読み取りから次の ROLLBACK または COMMIT まで。	UPDATE

表 36. 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 5 を参照)	行ロックの期間	ロック・タイプ
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 *CHG *CS *ALL (注 2 を参照)	読み取りから次の FETCH まで。 読み取りから次の FETCH まで。 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 READ READ
副照会 (読み取り専用カーソルまたは SELECT INTO)	*NONE *CHG *CS *ALL	ロックなし。 ロックなし。 各行は読み取りの期間ロックされる。 読み取りから次の ROLLBACK または COMMIT まで。	 READ READ

注:

1. 結果表が読み取り専用ではなく、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ビュー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. 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。
4. 反復可能読み取りを満足するために表またはビュー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。
5. 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。
6. 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。
7. USE AND KEEP EXCLUSIVE LOCKS 文節に *RS または *RR 分離レベルが指定されている場合、READ ロック (読み取りロック) の代わりに、行の UPDATE ロック (更新ロック) が適用されます。

関連情報

DECLARE CURSOR ステートメント

分離レベル

コミットメント制御

保管ポイント

保管ポイント は 1 つの作業単位中の特定の時点での、データとスキーマの状態を表す名前付きエンティティーです。トランザクション内で保管ポイントを作成できます。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リモート・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。

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、現行の保管ポイント・レベルの外側に設定された保管ポイント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、現行の保管ポイント・レベルの外側に設定された保管ポイント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。

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
- REFRESH TABLE

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 の実行時に従属ビュー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。

- ALTER PROCEDURE
- ALTER SEQUENCE
- CREATE ALIAS
- CREATE DISTINCT TYPE
- CREATE FUNCTION
- CREATE INDEX
- CREATE PROCEDURE
- CREATE SCHEMA
- CREATE SEQUENCE
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- DROP ALIAS
- DROP DISTINCT TYPE
- DROP FUNCTION
- DROP PROCEDURE
- DROP SCHEMA

- DROP SEQUENCE
- 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す必要がある場合があります。また、DROP SCHEMA ステートメントの場合には、スキーマを再び除去するか、または CL コマンドのライブラリー削除 (DLTLIB) を使用してスキーマの残り部分を除去する必要がある場合があります。

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。

関連概念

11 ページの『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。

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 CREATE TABLE および ALTER TABLE ステートメントを使用して、検査制約を追加または除去します。

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。

関連情報

ALTER TABLE ステートメント

CREATE TABLE ステートメント

保管/復元

i5/OS 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、順序、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、およびスキーマをディスク (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、SQL 関数、またはソース関数用に作成されたプログラムまたはサービスは復元時に、そのプロシージャーまたは関数が同じシグニチャーおよびプログラム名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る可能性があります。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。

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。

順序のための *DTAARA が復元されるとき、順序は自動的に SYSSEQUENCES カタログに追加されます。カタログが正常に更新されない場合、*DTAARA は変更されて、順序としては使用できなくなり、SQL9020 情報メッセージはジョブ・ログに出力されます。

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。

注: 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、ビュー、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® オペレーティング・システム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、順序、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と、カタログの中に記述されている表、視点、SQL パッケージ、順序、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、i5/OS CL コマンド、System/38™ 環境の CL コマンド、System/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、i5/OS DLTF CL コマンドを実行すること、System/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 37. 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
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

表 37. 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
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。

関連情報

バックアップおよび回復 PDF

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。

ルーチン

ルーチンとは、操作を実行するために呼び出すコードまたはプログラムの断片のことです。

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。

ストアード・プロシージャーに関する詳細については、「DB2 Universal Database for iSeries でのストアード・プロシージャー、トリガー、およびユーザー定義関数 (Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database for iSeries) PDF」を参照してください。

関連概念

12 ページの『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。

関連資料

305 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』
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。

関連情報

CREATE PROCEDURE ステートメント

Java SQL ルーチン

外部プロシージャーの定義

外部プロシージャーの CREATE PROCEDURE ステートメントは、プロシージャーに名前を付け、パラメーターと属性を定義し、システムがプロシージャーを呼び出す際、システムが使用するプロシージャーに関するその他の情報を提供します。

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、以下の処理を行います。

- 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数を渡したいことを示します。追加の引数は、N 個の短い整数の配列です。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 標識引数を渡すことはできません。

関連資料

138 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』
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。

SQL プロシージャーの定義

SQL プロシージャーの CREATE PROCEDURE ステートメントは、プロシージャーの名前付け、パラメーターおよびその属性の定義、プロシージャーが呼び出されるときに使用されるプロシージャーに関する他の情報の提供、プロシージャー本体の定義を行います。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、以下の処理を行います。

- 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、以下の処理を行います。

- 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 PROCEDURE RETURN_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、以下の処理を行います。

- 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。

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、以下の処理を行います。

- 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、以下の処理を行います。

- 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数を指定します。引数としては、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、通常は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

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 つの短い整数になります。この引数には、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 プリ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

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 から戻された引数がホスト変数に再びマップされます。

CALL ステートメントに渡された数値定数の場合、以下の規則が適用されます。
- すべての整数定数はフルワード・バイナリー整数として渡される。
- すべての 10 進定数はパック 10 進数値として渡される。精度および位取りは定数値を基に判別されます。たとえば、123.45 という値は、パック 10 進数 (5,2) として渡されます。同様に、001.01 という値は、精度 5、および位取り 2 で渡されます。
- 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る。

動的 CALL ステートメント上に指定された特殊レジスターは、以下のように渡されます。

CURRENT DATE
ISO 形式で 10 バイトの文字ストリングとして渡されます。

CURRENT DEGREE
5 バイトの文字ストリングとして渡されます。

CURRENT TIME
ISO 形式で 8 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 で、パック 10 進数として渡されます。

CURRENT TIMESTAMP
IBM SQL 形式で 26 バイトの文字ストリングとして渡されます。

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。

SESSION_USER
128 バイトの可変長文字ストリングとして渡されます。

SYSTEM_USER
128 バイトの可変長文字ストリングとして渡されます。

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)、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。

以下に C 言語で例示します。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。

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

```
#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。

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

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: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

CREATE PROCEDURE および CALL の例

```
/*************************************************************/
/*********** 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 */
```

```
/*****************************************************/
/* 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 before 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 *********************************/
/*****************************************************************/
```

サンプル・プロシージャー P1

```
/******** 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 can 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];

/*********************************************************/
/* 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 works 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 *****************************/
```

サンプル・プロシージャー P2

```
/******** 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;
```

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た REXX プロシージャー:

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

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

```
/*************************************************************/
/*********** 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()
        {
        /* ************************************************************/
        /* 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   ***********************************/
/*********************************************************************/

/*********************************************************************/
/****** 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
        /*********************************************************/
        /* 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   *********************************/
```

ストアード・プロシージャーから結果セットを戻す

ストアード・プロシージャーには、 CALL ステートメントを出すことによって、出力パラメーターを戻すことに加えて、ストアード・プロシージャーで開かれたカーソルに関連した結果表 (結果セットと呼ばれる) をアプリケーションに戻す機能があります。そのアプリケーションは、その後フェッチ要求を出して、結果セットのカーソルの行を読み取ることができます。

結果セットが戻されるかどうかは、カーソルの戻り属性によって決まります。カーソルの戻り属性は DECLARE CURSOR ステートメントに明示的に指定するか、デフォルトを使用することができます。 SET RESULT SETS ステートメントでは、結果セットをどこに戻すかについても指定できます。デフォルトでは、ストアード・プロシージャーで開かれるカーソルは、RETURN TO CALLER の戻り属性を持つよう定義されます。カーソルに関連した結果セットを呼び出しスタックで最外部プロシージャーと呼ばれるアプリケーションに戻すには、DECLARE CURSOR ステートメントに RETURN TO CLIENT の戻り属性を指定します。これによって、アプリケーションがネストされたプロシージャーを呼び出すときに、内部プロシージャーは結果セットを戻すことができます。結果セットが呼び出し元またはクライアントに戻されないカーソルの場合、DECLARE CURSOR ステートメントに WITHOUT RETURN の戻り属性を指定します。

多くの場合、アプリケーションで直接カーソルを開くよりも、ストアード・プロシージャーでカーソルを開き、その結果セットを戻す方が利点があります。たとえば、照会で参照される表の機密保護はストアード・プロシージャーから取り入れることができるので、アプリケーションのユーザーは表に対する権限を直接与えられる必要はありません。その代わりに、表にアクセスする適切な権限でコンパイルされたストアード・プロシージャーを呼び出す権限が与えられます。ストアード・プロシージャーでカーソルを開く他の利点は、単一の呼び出しからストアード・プロシージャーに複数の結果セットを戻すことができることです。これはアプリケーションを呼び出して別個にカーソルを開くよりも効率的です。さらに、異なるアプリケーションによって呼び出しが行われると、同じストアード・プロシージャーへの呼び出しでも戻される結果セットの数が異なる場合があります。

ストアード・プロシージャーの結果セットと連動するインターフェースには、JDBC、CLI、および ODBC が含まれます。これらの API インターフェースを使用してストアード・プロシージャーの結果セットと連動する方法は、以下で例示されています。

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

例 1: 単一の結果セットを戻すストアード・プロシージャーの呼び出し:

この例は、結果セットを戻すストアード・プロシージャーを呼び出すときに ODBC アプリケーションが行う API 呼び出しを示しています。

この例では、DECLARE CURSOR ステートメントが明示的な戻りを指定していないことに注意してください。呼び出しスタックに単一のストアード・プロシージャーしかない場合、 RETURN TO CLIENT の戻り属性だけでなく RETURN TO CALLER の戻り属性でも、結果セットがアプリケーションの呼び出し元に戻されるようにします。ストアード・プロシージャーは DYNAMIC RESULT SETS 文節で定義されることにも注意してください。 SQL プロシージャーの場合、ストアード・プロシージャーが結果セットを戻している場合に、この文節が要求されます。

ストアード・プロシージャーの定義:

```
PROCEDURE prod.resset

CREATE PROCEDURE prod.resset () LANGUAGE SQL
 DYNAMIC RESULT SETS 1
 BEGIN
 DECLARE C1 CURSOR FOR SELECT * FROM QIWS.QCUSTCDT;
 OPEN C1;
 RETURN;
 END
```

ODBC アプリケーション

注: ロジックのいくつかは除去されています。

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

```
 :
strcpy(stmt,"call prod.resset()");
rc = SQLExecDirect(hstmt,stmt,SQL_NTS);
if (rc == SQL_SUCCESS)
 {
   // CALL statement has executed successfully. Process the result set.
   // Get number of result columns for the result set.
   rc = SQLNumResultCols(hstmt, &wNum);
   if (rc == SQL_SUCCESS)
       // Get description of result columns in result set
      { rc = SQLDescribeCol(hstmt,ã);
        if (rc == SQL_SUCCESS)
             :

       {
       // Bind result columns based on attributes returned
       //
         rc = SQLBindCol(hstmt,ã);
       :
      // FETCH records until EOF is returned

        rc = SQLFetch(hstmt);
        while (rc == SQL_SUCCESS)
          { // process result returned on the SQLFetch
                 :
            rc = SQLFetch(hstmt);
          }
         :
          }
        // Close the result set cursor when done with it.
        rc = SQLFreeStmt(hstmt,SQL_CLOSE);
       :
```

例 2: ネストされたプロシージャーから結果セットを戻すストアード・プロシージャーの呼び出し:

この例は、ネストされたストアード・プロシージャーが最外部プロシージャーにどのように開いて、結果セットを戻すことができるかを示しています。

ネストされたストアード・プロシージャーのある環境で最外部プロシージャーに結果セットを戻すには、DECLARE CURSOR ステートメントまたは SET RESULT SETS ステートメントで RETURN TO CLIENT の戻り属性を使用して、カーソルが最外部プロシージャーを呼び出したアプリケーションに戻されることを示す必要があります。このネストされたプロシージャーはクライアントに 2 つの結果セットを戻すことに注意してください。最初に配列結果セット、2 番目にカーソル結果セットを戻します。以下に、 ODBC と JDBC の両方のクライアント・アプリケーションと、それぞれのストアード・プロシージャーを示します。

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

ストアード・プロシージャーの定義

```
CREATE PROCEDURE prod.rtnnested () LANGUAGE CL DYNAMIC RESULT SET 2
        EXTERNAL NAME prod.rtnnested GENERAL

CREATE PROCEDURE prod.rtnclient () LANGUAGE RPGLE
        EXTERNAL NAME prod.rtnclient GENERAL
```

ストアード・プロシージャー prod.rtnnested の CL ソース

```
PGM
                        CALL       PGM(PROD/RTNCLIENT)
```

ストアード・プロシージャー prod.rtnclient の ILE RPG ソース

```
DRESULT           DS                  OCCURS(20)
D COL1                     1     16A
C     1             DO        10            X                2 0
C     X             OCCUR     RESULT
C                   EVAL      COL1='array result set'
C                   ENDDO
C                   EVAL      X=X-1
C/EXEC SQL DECLARE C2 CURSOR WITH RETURN TO CLIENT
C+ FOR SELECT LSTNAM FROM QIWS.QCUSTCDT FOR FETCH ONLY
C/END-EXEC
C/EXEC SQL
C+  OPEN C2
C/END-EXEC
C/EXEC SQL
C+ SET RESULT SETS FOR RETURN TO CLIENT ARRAY :RESULT FOR :X ROWS,
C+  CURSOR C2
C/END-EXEC
C                   SETON                                        LR
C                   RETURN
```

ODBC アプリケーション

```
//****************************************************************
//
// Module:
//    Examples.C
//
// Purpose:
//    Perform calls to stored procedures to get back result sets.

//
// ****************************************************************


#include "common.h"
#include "stdio.h"


// ****************************************************************
//
// Local function prototypes.
//
// ****************************************************************

SWORD   FAR PASCAL RetClient(lpSERVERINFO lpSI);
BOOL    FAR PASCAL Bind_Params(HSTMT);
BOOL    FAR PASCAL Bind_First_RS(HSTMT);
BOOL    FAR PASCAL Bind_Second_RS(HSTMT);
```

```
// ********************************************************************
//
// Constant strings definitions for SQL statements used in
// the auto test.
//
// ********************************************************************
//
// Declarations of variables global to the auto test.
//
// ********************************************************************
#define ARRAYCOL_LEN 16
#define LSTNAM_LEN 8
char   stmt[2048];
char   buf[2000];

UDWORD rowcnt;
char arraycol[ARRAYCOL_LEN+1];
char lstnam[LSTNAM_LEN+1];
SDWORD cbcol1,cbcol2;

lpSERVERINFO lpSI;  /* Pointer to a SERVERINFO structure. */
// ********************************************************************
//
//  Define the auto test name and the number of test cases
//     for the current auto test.  These informations will
//     be returned by AutoTestName().
//
// ********************************************************************

LPSTR szAutoTestName =  CREATE_NAME("Result Sets Examples");
UINT  iNumOfTestCases = 1;



// ********************************************************************
//
//  Define the structure for test case names, descriptions,
//     and function names for the current auto test.
//     Test case names and descriptions will be returned by
//     AutoTestDesc().  Functions will be run by
//     AutoTestFunc() if the bits for the corresponding test cases
//     are set in the rglMask member of the SERVERINFO
//     structure.
//
// ********************************************************************
struct TestCase  TestCasesInfo[] =
{
        "Return to Client",
        "2 result sets   ",
         RetClient

};


// ********************************************************************
//
// Sample return to Client:
//      Return to Client result sets. Call a CL program which in turn
//      calls an RPG program which returns 2 result sets. The first
//      result set is an array result set and the second is a cursor
//      result set.
//
//
// ********************************************************************
SWORD FAR PASCAL RetClient(lpSERVERINFO lpSI)
{
   SWORD      sRC = SUCCESS;
```

```
RETCODE    returncode;
HENV       henv;
HDBC       hdbc;
HSTMT      hstmt;



if (FullConnect(lpSI, &henv, &hdbc, &hstmt) == FALSE)
{
   sRC = FAIL;
   goto ExitNoDisconnect;
}
// *********************************************************
// Call CL program PROD.RTNNESTED, which in turn calls RPG
// program RTNCLIENT.
// *********************************************************
strcpy(stmt,"CALL PROD.RTNNESTED()");
// *************************************************************
// Call the CL program prod.rtnnested. This program will in turn
// call the RPG program proc.rtnclient, which will open 2 result
// sets for return to this ODBC application.
// *************************************************************
returncode = SQLExecDirect(hstmt,stmt,SQL_NTS);
if (returncode != SQL_SUCCESS)
{
   vWrite(lpSI, "CALL PROD.RTNNESTED is not Successful", TRUE);
}
else
{
   vWrite(lpSI, "CALL PROC.RTNNESTED was Successful", TRUE);
}
// *************************************************************
// Bind the array result set output column. Note that the result
// sets are returned to the application in the order that they
// are specified on the SET RESULT SETS statement.
// *************************************************************
if (Bind_First_RS(hstmt) == FALSE)
{
   myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS,
              returncode, "Bind_First_RS");
   sRC = FAIL;
   goto ErrorRet;
}
else
{
   vWrite(lpSI, "Bind_First_RS Complete...", TRUE);
}
// *************************************************************
// Fetch the rows from the array result set. After the last row
// is read, a returncode of SQL_NO_DATA_FOUND will be returned to
// the application on the SQLFetch request.
// *************************************************************
returncode = SQLFetch(hstmt);
while(returncode == SQL_SUCCESS)
{
 wsprintf(stmt,"array column = %s",arraycol);
 vWrite(lpSI,stmt,TRUE);
 returncode = SQLFetch(hstmt);
}
if  (returncode == SQL_NO_DATA_FOUND) ;
else {
       myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS_WITH_INFO,
                  returncode, "SQLFetch");
       sRC = FAIL;
       goto ErrorRet;
     }
// *********************************************************
```

```
   // Get any remaining result sets from the call. The next
   // result set corresponds to cursor C2 opened in the RPG
   // Program.
   // *******************************************************
   returncode = SQLMoreResults(hstmt);
   if (returncode != SQL_SUCCESS)
    {
     myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS, returncode, "SQLMoreResults");
     sRC = FAIL;
     goto ErrorRet;
    }
   // *************************************************************
   // Bind the cursor result set output column. Note that the result
   // sets are returned to the application in the order that they
   // are specified on the SET RESULT SETS statement.
   // *************************************************************

   if (Bind_Second_RS(hstmt) == FALSE)
   {
      myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS,
                 returncode, "Bind_Second_RS");
      sRC = FAIL;
      goto ErrorRet;
   }
   else
   {
      vWrite(lpSI, "Bind_Second_RS Complete...", TRUE);
   }
   // *************************************************************
   // Fetch the rows from the cursor result set. After the last row
   // is read, a returncode of SQL_NO_DATA_FOUND will be returned to
   // the application on the SQLFetch request.
   // *************************************************************
    returncode = SQLFetch(hstmt);
   while(returncode == SQL_SUCCESS)
   {
    wsprintf(stmt,"lstnam = %s",lstnam);
    vWrite(lpSI,stmt,TRUE);
    returncode = SQLFetch(hstmt);
   }
   if (returncode == SQL_NO_DATA_FOUND) ;
   else {
          myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS_WITH_INFO,
                     returncode, "SQLFetch");
          sRC = FAIL;
          goto ErrorRet;
         }

   returncode = SQLFreeStmt(hstmt,SQL_CLOSE);
   if (returncode != SQL_SUCCESS)
   {
      myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS,
                          returncode, "Close statement");
      sRC = FAIL;
      goto ErrorRet;
   }
   else
   {
      vWrite(lpSI, "Close statement...", TRUE);
   }



ErrorRet:
   FullDisconnect(lpSI, henv, hdbc, hstmt);
   if (sRC == FAIL)
```

```
   {
      // a failure in an ODBC function that prevents completion of the
      // test - for example, connect to the server
      vWrite(lpSI, "\t\t *** Unrecoverable RTNClient Test FAILURE ***",  TRUE);
   } /* endif */

ExitNoDisconnect:

   return(sRC);
} // RetClient




BOOL FAR PASCAL Bind_First_RS(HSTMT hstmt)
{
 RETCODE rc = SQL_SUCCESS;
   rc = SQLBindCol(hstmt,1,SQL_C_CHAR,arraycol,ARRAYCOL_LEN+1, &cbcol1);
   if (rc != SQL_SUCCESS) return FALSE;
   return TRUE;
}
BOOL FAR PASCAL Bind_Second_RS(HSTMT hstmt)
{
 RETCODE rc = SQL_SUCCESS;
   rc = SQLBindCol(hstmt,1,SQL_C_CHAR,lstnam,LSTNAM_LEN+1,&dbcol2);
   if (rc != SQL_SUCCESS) return FALSE;
   return TRUE;
}
```

JDBC アプリケーション

```
//-------------------------------------------------------------
// Call Nested procedures which return result sets to the
// client, in this case a JDBC client.
//-------------------------------------------------------------
import java.sql.*;
public class callNested
{
  public static void main (String argv[])                // Main entry point
  {
    try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException e) {
      e.printStackTrace();
    }

   try {
      Connection jdbcCon =

DriverManager.getConnection("jdbc:db2:lp066ab","Userid","xxxxxxx");
      jdbcCon.setAutoCommit(false);
      CallableStatement cs = jdbcCon.prepareCall("CALL PROD.RTNNESTED");
      cs.execute();
      ResultSet rs1 = cs.getResultSet();
      int r = 0;
while (rs1.next())
      {
      r++;
      String s1 = rs1.getString(1);
      System.out.print("Result set 1 Row: " + r + ": ");
      System.out.print(s1 + "   " );
      System.out.println();
      }
      cs.getMoreResults();
      r = 0;
```

```
      ResultSet rs2 = cs.getResultSet();
      while (rs2.next())
      {
      r++;
      String s2 = rs2.getString(1);
      System.out.print("Result set 2 Row: " + r + ": ");
      System.out.print(s2 + "   ");
      System.out.println();
      }
   }
   catch ( SQLException e ) {
      System.out.println( "SQLState: " + e.getSQLState() );
      System.out.println( "Message : " + e.getMessage() );
      e.printStackTrace();
   }
  }   // main
}
```

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 ステートメントおよび関数の呼び出しでは、サポートされるすべてのホスト言語で書かれたプログラムおよび REXX プロシージャーに引数を渡すことができます。

次の表で示されているように、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。列がブランクの場合、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または関数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 プロシージャーまたは関数の呼び出し時には、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。

表 38. 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	適用外	PIC S9(4) BINARY
INTEGER	long	適用外	PIC S9(9) BINARY
BIGINT	long long	適用外	PIC S9(18) BINAR **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)	適用外	適用外	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	適用外	COMP-1 **注:** ILE COBOL for iSeries でのみサポートされます。

表 38. 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	適用外	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]	適用外	可変長文字ストリング
VARCHAR(n) FOR BIT DATA	VARCHAR 構造化フォーム	適用外	可変長文字ストリング
CLOB	CLOB 構造化フォーム	適用外	CLOB 構造化フォーム **注:** ILE COBOL for iSeries でのみサポートされます。
GRAPHIC(n)	wchar_t ... [n+1]	適用外	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 構造化フォーム	適用外	可変長グラフィック・ストリング **注:** ILE COBOL for iSeries でのみサポートされます。
DBCLOB	DBCLOB 構造化フォーム	適用外	DBCLOB 構造化フォーム **注:** ILE COBOL for iSeries でのみサポートされます。
BINARY	BINARY 構造化フォーム	適用外	BINARY 構造化フォーム
VARBINARY	VARBINARY 構造化フォーム	適用外	VARBINARY 構造化フォーム
BLOB	BLOB 構造化フォーム	適用外	BLOB 構造化フォーム **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) **注:** ILE COBOL for iSeries の場合のみ、FORMAT DATE。
TIME	char ... [9]	TYPE(*CHAR) LEN(8)	PIC X(8) **注:** ILE COBOL for iSeries の場合のみ、FORMAT TIME。
TIMESTAMP	char ... [27]	TYPE(*CHAR) LEN(26)	PIC X(26) **注:** ILE COBOL for iSeries の場合のみ、FORMAT TIMESTAMP。

表 38. 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
ROWID	ROWID 構造化フォーム	適用外	ROWID 構造化フォーム
データ・リンク	適用外	適用外	適用外
標識変数	short	適用外	PIC S9(4) BINARY

表 39. パラメーターのデータ・タイプ

SQL データ・タイプ	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	short	short	FIXED BIN(15)
INTEGER	int	int	FIXED BIN(31)
BIGINT	long	long	適用外
DECIMAL (p,s)	BigDecimal	BigDecimal	FIXED DEC(p,s)
NUMERIC(p,s)	BigDecimal	BigDecimal	適用外
REAL または FLOAT(p)	float	float	FLOAT BIN(p)
DOUBLE PRECISION または FLOAT または FLOAT(p)	double	double	FLOAT BIN(p)
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	バイト[ ]	com.ibm.db2.app.Blob	CHAR(n) VAR
CLOB	java.sql.Clob	com.ibm.db2.app.Clob	CLOB 構造化フォーム
GRAPHIC(n)	文字列	文字列	適用外
VARGRAPHIC(n)	文字列	文字列	適用外
DBCLOB	java.sql.Clob	com.ibm.db2.app.Clob	DBCLOB 構造化フォーム
BINARY	バイト[ ]	com.ibm.db2.app.Blob	BINARY 構造化フォーム
VARBINARY	バイト[ ]	com.ibm.db2.app.Blob	VARBINARY 構造化フォーム
BLOB	java.sql.Blob	com.ibm.db2.app.Blob	BLOB 構造化フォーム
DATE	日付	文字列	CHAR(10)
TIME	時間	文字列	CHAR(8)
TIMESTAMP	タイム・スタンプ	文字列	CHAR(26)
ROWID	バイト[]	com.ibm.db2.app.Blob	ROWID 構造化フォーム
データ・リンク	適用外	適用外	適用外
標識変数	適用外	適用外	FIXED BIN(15)

表 40. 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	適用外	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 から 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 5、41 から 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 452 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 から 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 10、41 から 42 桁目は 00 とする。
BIGINT	適用外	適用外	データ指定。サブフィールド指定の 40 桁目は ***I***、長さ 20、41 から 42 桁目は 00 とする。
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 から 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***、41 から 42 桁目は 00 から 31 とする。
NUMERIC(p,s)	適用外	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 から 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 から 42 桁目を 00 から 31 とする。
REAL 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	適用外	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	適用外	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 から 42 桁目を**ブランク** とする。

表 40. 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
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	適用外	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 から 42 桁目を**ブランク** とし、44 から 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	適用外	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 から 42 桁目を**ブランク** とし、44 から 80 桁目にキーワード VARYING を置く。
CLOB	適用外	適用外	CLOB 構造化フォーム
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	適用外	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	適用外	データ指定。サブフィールド指定の 40 桁目は ***G*** とし、44 から 80 桁目にキーワード VARYING を置く。
DBCLOB	適用外	適用外	DBCLOB 構造化フォーム
BINARY	適用外	適用外	BINARY 構造化フォーム
VARBINARY	適用外	適用外	VARBINARY 構造化フォーム
BLOB	適用外	適用外	BLOB 構造化フォーム
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 から 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 から 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。
ROWID	適用外	適用外	ROWID 構造化フォーム

表 40. 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
データ・リンク	適用外	適用外	適用外
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 から 42 桁目は 00 とする。

関連情報

組み込み SQL プログラミング

Java SQL ルーチン

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を使用する場合)。

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デフォルト値が渡され、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、およびプロシージャーから戻される出力パラメーター共に同じです。 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 番目の引数として) 方法にも注目してください。

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

プロシージャーは、次のように定義されているとします。

```
CREATE PROCEDURE PROC1
   (INOUT DECIMALOUT DECIMAL(7,2), INOUT DECOUT2 DECIMAL(7,2))
   EXTERNAL NAME LIB1.PROC1 LANGUAGE RPGLE
   GENERAL WITH NULLS)
```

CALL ステートメント内の標識変数の処理

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
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
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
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
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

呼び出しプログラムへの完了状況の戻り

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。

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側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。プロシージャーを出す呼び出し側プログラムは 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。

関連情報

SQL 制御ステートメント

ユーザー定義関数 (UDF) の使用

書き込み SQL アプリケーションで、いくつかのアクションまたは操作を UDF として、またはアプリケーションのサブルーチンとしてインプリメントすることができます。新規操作をアプリケーションのサブルーチンとしてインプリメントする方がより簡単に見える場合がありますが、代わりに UDF を使用する利点についても考慮することができます。

たとえば、新しい操作が、他のユーザーまたはプログラムが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、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、データベースは、異なるが互換性のあるデータ・タイプに関数を使用することが可能になります。

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
```

それぞれの行を受け取ると、プログラムの SELECTION_CRITERIA 関数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 の各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 1 つの列だけが、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しされます。

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。

関連概念

12 ページの『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。

UDF の概念

このトピックでは、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、各関数は、*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)、SQL で書き込まれるコード、またはソース・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 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。

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 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 が解決するための主要概念です。パスは、UDF および UDT への修飾なしの参照を解決するために使用されるスキーマ名の順序リストです。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 のプリコンパイル・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デフォルト値が入っています。

```
"QSYS","QSYS2","<ID>"
```

このデフォルト値は静的 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、デフォルトのパスは *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 つの関数は同じシグニチャーを持つことはできません。関数シグニチャーは、修飾された関数名、および定義された順序内のすべての関数パラメーターのデータ・タイプです。

関数解決

多重定義と関数パスを考慮に入れて、各関数参照 (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。

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、 UDF を呼び出す SQL ステートメントは失敗します。

データベースで使用している UDF のデフォルトの待ち時間は、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。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。

関連情報

照会オプション・ファイル (QAQQINI)

SQL 関数としての UDF の作成

SQL 関数とは、CREATE FUNCTION SQL 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、パラメーターが渡される方法の詳細がデータベースに定義されます。

例: SQL スカラー UDF:

この例では、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
```

例: SQL 表 UDF:

この例では、日付を基にデータを戻す表関数について解説します。

```
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ができます。

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。

関連情報

Java SQL ルーチン

UDF の登録:

UDF は、SQL 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。

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。

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。

関連情報

CREATE FUNCTION ステートメント

例: 指数:

この例では、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、どちらかの引数が NULL の場合に、結果値を NULL にする必要があるので、 RETURNS NULL ON NULL INPUT が指定されています。 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:

ある短い文字列 (引数として渡される) が、ある CLOB 値 (これも引数として渡される) 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。

FLOAT 結果を戻すために、C プログラムが作成されました。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 FINDSTRING
  EXTERNAL NAME 'MYLIB/MYPGM(FINDSTR)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  RETURNS NULL ON NULL INPUT
```

UDF プログラムが実際に 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 SQL ステートメントにこの値を戻す前に、この値を INTEGER にキャストしたいとします。また、この関数にユーザーの特定の名前を付けたいとします。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。

例: BLOB 文字列検索:

この例では、FINDSTRING 関数が CLOB 上で機能するのと同じく BLOB 上でも機能するようにしたいとします。これを実行するには、BLOB を最初のパラメーターに指定して、FINDSTRING をもう 1 つ定義します。

```
CREATE FUNCTION FINDSTRING (BLOB(500K), VARCHAR(200))
  RETURNS INTEGER
  CAST FROM FLOAT
  SPECIFIC FINDSTRING_BLOB
  EXTERNAL NAME 'MYLIB/MYPGM(FINDSTR)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  RETURNS NULL ON NULL INPUT
```

この例は UDF 名の多重定義を示しており、複数の UDF が同一のプログラム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、DB2 と UDF プログラムの間の BLOB と CLOB に対する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。

例: UDT での文字列検索:

BLOB 文字列検索の FINDSTRING 関数の目的を達した後、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 をもつ値でも機能するようにしたいので、もう 1 つの FINDSTRING 関数を作成します。この関数は、BLOB 値に対して機能する FINDSTRING の上にソース化されます。次の例では FINDSTRING にさらに多重定義されていることにご注意ください。

```
CREATE FUNCTION FINDSTRING (BOAT, VARCHAR(200))
  RETURNS INT
  SPECIFIC "slick_fboat"
  SOURCE SPECIFIC FINDSTRING_BLOB
```

この FINDSTRING 関数は 167 ページの『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。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、167 ページの『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。

関連資料

167 ページの『例: BLOB 文字列検索』
この例では、FINDSTRING 関数が CLOB 上で機能するのと同じく BLOB 上でも機能するようにしたいとします。 これを実行するには、BLOB を最初のパラメーターに指定して、FINDSTRING をもう 1 つ定義します。

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。

限定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。

```
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。 関数は呼び出されるたびに、この値を増分し、それを戻します。

```
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 というデフォルト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 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:

この例では、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。 RETURNS 文節は、DOCMATCH からの出力を定義します (列名 DOC_ID を含む)。この表関数に FINAL CALL を指定する必要はありません。表関数は並列に動作できないので、DISALLOW PARALLEL キーワードが要求されます。 DOCMATCH からの出力のサイズが大容量の表であっても、CARDINALITY 20 が指定されているため、最適化プログラムは良い判断を行うことができます。

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。

DB2 から外部関数へ引数を渡す:

DB2 は、UDF に渡されるすべてのパラメーターのための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。

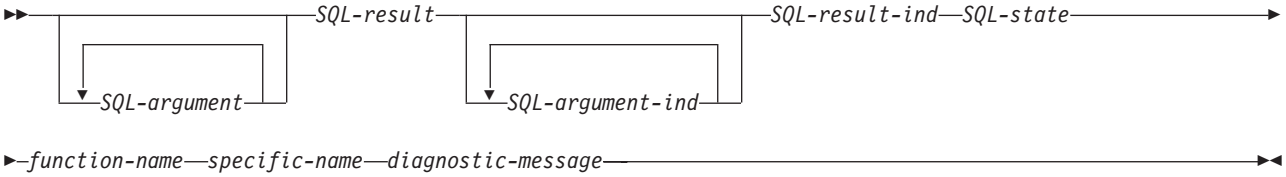
UDF の中でパラメーターを定義し、使用する場合は、あるパラメーター用に定義されたよりも多くの記憶域が、そのパラメーターに対して参照されないように十分な注意が必要です。パラメーターはすべて同じ記憶スペースに格納されるので、あるパラメーターの記憶スペースを超えると、別のパラメーターの値を上書きすることになります。このことは、無効な入力データを関数が使用する原因になったり、データベースに戻される値が無効になる原因になります。

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
この引数は、UDF を呼び出す前に、DB2 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数の数です。これらの引数の各値は、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 によって無視されます。

SQL-result
この引数は、DB2 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 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 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
この引数は、UDF を呼び出す前に、DB2 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。各標識は、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 引数が存在し、ヌルではない。

-1 引数がヌルである。

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 によって無視されます。

SQL-result-ind
この引数は、DB2 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、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、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。

function-name
この引数は、UDF を呼び出す前に、DB2 によって設定されます。 引数は、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 によって無視されます。

specific-name
この引数は、UDF を呼び出す前に、DB2 によって設定されます。 引数は、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。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 によって無視されます。

diagnostic-message
この引数は、UDF を呼び出す前に、DB2 によって設定されます。 この引数は、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 によって無視されます。

関連情報

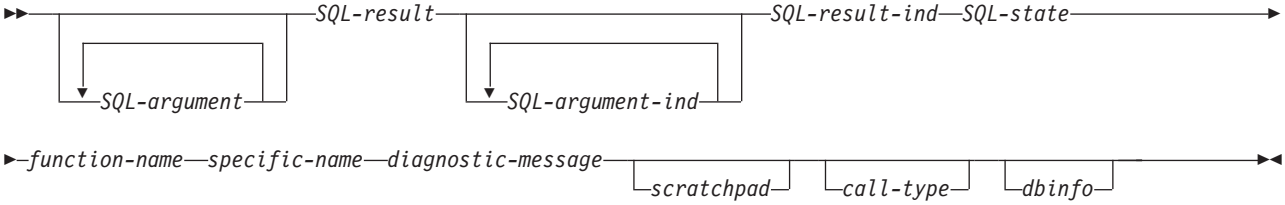
SQL メッセージおよびコード

パラメーター・スタイル DB2SQL:

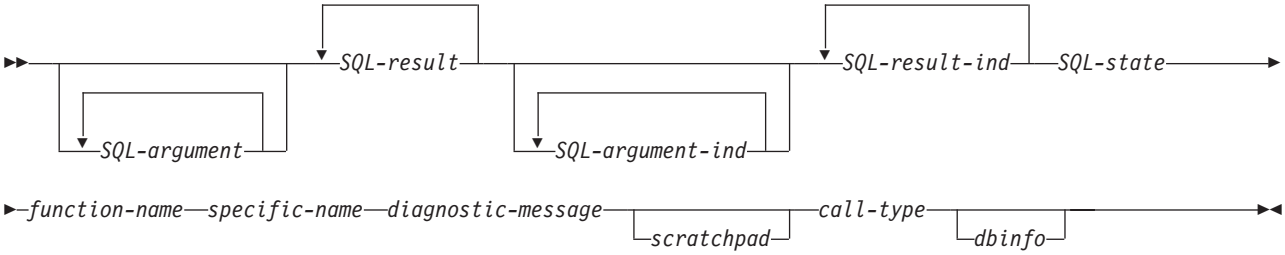
DB2SQL パラメーター・スタイルを使用すると、パラメーター・スタイル SQL の場合に渡されるのと同じように、同じパラメーターが同じ順序で、外部プログラムまたはサービス・プログラムに渡されます。ただし、DB2SQL では、追加のオプション・パラメーターも渡すことができます。

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

この引数は、UDF を呼び出す前に、DB2 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 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
この引数は、UDF を呼び出す前に、DB2 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* 引数の後に続きます。スクラッチパッド引数が存在しない場合は、*diagnostic-message* 引数の後に続きます。この引数は、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 引数、SQL-state、または diagnostic-message 引数を使用して応答を戻すことはありません。これらの引数は、UDF から戻るときにシステムに無視されます。

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「end-of-table (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。

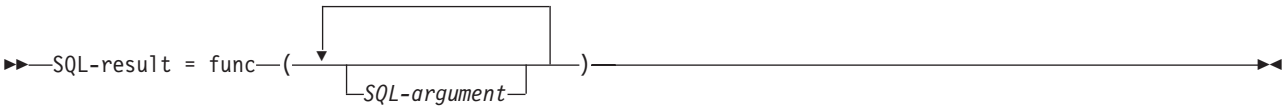
2 *最終呼び出し* を示します。 *SQL-argument* または *SQL-argument-ind* の値は渡されません。 UDF は、SQL-result、SQL-result-ind 引数、SQL-state、または diagnostic-message 引数を使用して応答を戻すことはありません。これらの引数は、UDF から戻るときにシステムに無視されます。

dbinfo この引数は、UDF を呼び出す前に、DB2 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
この引数は、UDF を呼び出す前に、DB2 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数の数です。これらの引数の各値は、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 によって無視されます。

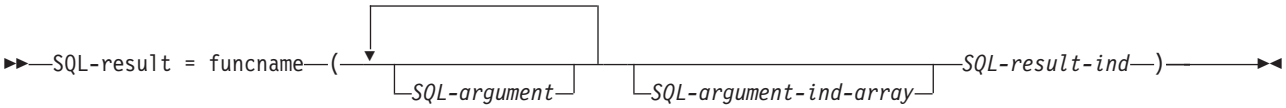
SQL-result
この値は UDF によって戻されます。DB2 は値をデータベース記憶域にコピーします。値を正しく戻すためには、関数コードは値を戻す関数でなければなりません。データベースは、CREATE FUNCTION ステートメントで指定された戻り値として定義された値だけをコピーします。CREATE FUNCTION ステートメントで CAST FROM 文節が使用されている場合は、DB2 は、UDF が CAST FROM 文節で定義された値を戻すと想定し、CAST FROM 文節が使用されていない場合は、 DB2 は、UDF が RETURNS 文節で定義された値を戻すと想定します。

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・プログラムに対して以下のように (指定された順序で) 渡されます。



SQL-argument
この引数は、UDF を呼び出す前に、DB2 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数の数です。これらの引数の各値は、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 によって無視されます。

SQL-argument-ind-array
この引数は、UDF を呼び出す前に、DB2 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 引数が存在し、ヌルではない。

-1 引数がヌルである。

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 によって無視されます。

SQL-result-ind

この引数は、DB2 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、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、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 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 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 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、単一引用符付き、あるいは単一引用符なしで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。

関連情報

Java SQL ルーチン

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。

関連情報

Java SQL ルーチン

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 ステートメントの 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* 条件を目印に照会処理を行います。 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。

UDF のエラー処理:

UDF の処理中にエラーが発生すると、システムは指定されたモデルに従います。

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、それ以上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、それ以上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、それ以上 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。

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。

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。

関連資料

178 ページの『隔離または隔離解除の考慮事項』
ユーザー定義関数 (UDF) を作成するときは、その UDF を隔離解除 UDF にするかどうか考慮します。

関連情報

マルチスレッド・プログラミングでのデータベースについての考慮事項

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。

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。

関連情報

マルチスレッド・プログラミングでのデータベースについての考慮事項

隔離または隔離解除の考慮事項:

ユーザー定義関数 (UDF) を作成するときは、その UDF を隔離解除 UDF にするかどうか考慮します。

デフォルトでは、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、デフォルト値に従って 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 オプションを介してより高速な実行を要求します。
BEGIN
RETURN parameter1 * 3;
END;
```

関連資料

177 ページの『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。

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い場合、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 *SRVPGM 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。

プログラム・オブジェクトが更新できない場合でも、関数は作成されます。

例: UDF コード

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:

この例では、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
```

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

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 関数の使用

CREATE FUNCTION ステートメント:

```
CREATE FUNCTION SQUARE( inval INT) RETURNS INT
LANGUAGE SQL
SET OPTION DBGVIEW=*SOURCE
BEGIN
RETURN(inval*inval);
END
```

この例は、デバッグできる SQL 関数を作成します。

外部関数、パラメーター・スタイル SQL の使用

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 の使用

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:

SELECT ステートメントで、複数の行に番号を付けていきたいとします。その場合、数を増分しカウンターを戻す UDF を作成します。

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

以下の例は、DB2 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 は、スクラッチパッド引数の割り振り、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 入力引数以外の引数に依存するので (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:

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。

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

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 */ (See note 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  */
/* need 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++ ;
    }
    /* can 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 */ (See note 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 clean up (no real args!): Close table */ (See note 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、OPEN で獲得した記憶域、およびスクラッチパッドにアンカーする記憶域を CLOSE で解放します。

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)';
```

埋め込まれた番号付きの注を参照しながら、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受け付けず、7 つ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 入力引数以外の引数に依存するので、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
```

SQL ステートメント内での 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。検査制約も、UDF としてシステムによってインプリメントされる組み込み関数への参照を含むことができません。
- 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。

関数の引数としてのパラメーター・マーカーまたは NULL 値の使用:

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。

次のようにコード化することはできません。

```
BLOOP(?)
```

あるいは

```
BLOOP(NULL)
```

関数解決では、引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データ・タイプを指定できます。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 DB2 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. 1 番目の BLOOP を DECIMAL または DOUBLE 引数を使用して呼び出す必要がある場合は、意図に応じて、明示的な処置を選択できます。

   ```
   SELECT PABLO.BLOOP(INTEGER(COLUMN1)) FROM T
   SELECT PABLO.BLOOP(FLOOR(COLUMN1)) FROM T
   ```

関連資料

『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ことができます。 この場合、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。

219 ページの『UDT の定義』
UDT は CREATE DISTINCT TYPE ステートメントで定義されます。

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ことができます。この場合、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
```

「修飾された関数参照を使用する」のセクションで 2 つの BLOOP 関数が作成され、そのうちの 1 つを選択したいとします。以下のデフォルト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プリ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"
```

スキーマ KATHY に BLOOP 関数がある場合、関数選択はその関数を非常にうまく解決し、ステートメントはエラーなしで実行します。 DB2 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です。

関連資料

188 ページの『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検索をそのスキーマに制限します。

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、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) が指定名を持つか、引数として定義されたパラメーターと同じ番号を持つか、各パラメーターが対応する引数のタイプと同じであるか、またはその引数のタイプから生成可能であるかをすべて調べます。

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。

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'
  ```

  この方法では、>、=、 LIKE、IN、などの組み込み条件付き演算子を多重定義することはできません。
- 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 が実行される前に、戻された整数が VARCHAR としてキャストされるので、ステートメントは期待される結果を戻さない場合があります。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。システムは、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数として有効です。 167 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 外部 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、外部 UDF に限られます。そのような関数 (UDF) への引数は、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。DB2 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 の限定タイプの指定は、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、タイプ BOAT のオブジェクトをその引数としてとる 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。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、結果が予測不能になります。

トリガー

トリガー は、指定した表あるいはビューに対して、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 SQL あるいは外部として定義できます。

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。

| トリガーが表またはビューに関連付けられると、表またはビュー、あるいは、その表またはビューに基づいて作成されたすべての論理ファイルまたはビュー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。 ビューに定義できるのは、SQL トリガーのみです。1 つの表またはビューに定義できるトリガーの数は最大 200 です。

| 表に対するそれぞれの変更操作について、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)

| 1 つのビューに対する各変更操作では、挿入、更新、削除の代わりに、一連のアクションを実行するトリガーの代替を呼び出すことができます。ビューは以下のトリガーに関連付けることができます。

| - 削除トリガーの代替
| - 挿入トリガーの代替
| - 更新トリガーの代替

関連情報

データベース内での自動イベントのトリガー

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、表のグループをアクティブに制御、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 表あるいはビューが変更されると、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。
| CREATE TRIGGER ステートメントで参照されるすべてのオブジェクト (表、関数など) は、必ず存在して
| いなければなりません。存在しない場合、トリガーは作成されません。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。

関連概念

203 ページの『SQL ルーチンのデバッグ』
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、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。

関連情報

SQL 制御ステートメント

CREATE TRIGGER ステートメント

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。このタイプのトリガーは AFTER トリガーと呼ばれます。

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。

INSTEAD OF SQL トリガー:

INSTEAD OF トリガーとは、SQL UPDATE ステートメント、DELETE ステートメントまたは INSERT ステートメントの「代わりに」処理される SQL トリガーです。 SQL BEFORE トリガーや AFTER トリガーとは異なり、INSTEAD OF トリガーを定義できるのはビューのみで、表では定義できません。

INSTEAD OF トリガーは、本質的には挿入、更新、削除ができないビューで、挿入、更新、削除を可能にします。削除可能、更新可能、挿入可能の各ビューについての詳細は、CREATE VIEW を参照してください。

つまり、SQL INSTEAD OF トリガーをビューに追加すると、それまでは読み取りしかできなかったビューが、挿入、更新、または削除操作のターゲットとして使用できるようになります。 INSTEAD OF トリガーはビューを保守するために必要な操作を定義します。

表へのアクセスを制御するためにビューを使用できるようになります。 INSTEAD OF トリガーは、表へのアクセス制御の保守を単純化します。

INSTEAD OF トリガーの使用

以下に示すビュー V1 の定義は、更新、削除、挿入が可能です。

```
CREATE TABLE T1 (C1 VARCHAR(10), C2 INT)
CREATE VIEW V1(X1) AS SELECT C1 FROM T1 WHERE C2 > 10
```

以下の挿入ステートメントの場合、表 T1 の C1 が値 'A' に割り当てられます。C2 には NULL 値が割り当てられます。NULL 値が原因で、ビュー V1 の C2 > 10 という選択基準に新規行が一致しない可能性があります。

```
INSERT INTO V1 VALUES('A')
```

INSTEAD OF トリガー IOT1 を追加することにより、ビューで選択される行に異なる値を指定できるようになります。

```
CREATE TRIGGER IOT1 INSTEAD OF INSERT ON V1
REFERENCING NEW AS NEW_ROW
FOR EACH ROW MODE DB2SQL
INSERT INTO T1 VALUES(NEW_ROW.X1, 15)
```

ビューを削除可能にする

以下の結合ビュー V3 の定義は、更新、削除、挿入ができません。

```
CREATE TABLE A (A1 VARCHAR(10), A2 INT)
CREATE VIEW V1(X1) AS SELECT A1 FROM A

CREATE TABLE B (B1 VARCHAR(10), B2 INT)
CREATE VIEW V2(Y1) AS SELECT  B1 FROM B

CREATE VIEW V3(Z1, Z2) AS SELECT V1.X1, V2.Y1 FROM V1, V2 WHERE V1.X1 = 'A' AND V2.Y1 > 'B'
```

INSTEAD OF トリガー IOT2 を追加するには、ビュー V3 を削除できるようにします。

```
CREATE TRIGGER IOT2 INSTEAD OF DELETE ON V3
REFERENCING OLD AS OLD_ROW
FOR EACH ROW MODE DB2SQL
BEGIN
  DELETE FROM A WHERE A1 = OLD_ROW.Z1;
  DELETE FROM B WHERE B1 = OLD_ROW.Z2;
END
```

このトリガーを使用することで、以下の DELETE ステートメントが許可されます。このトリガーにより、「A」の値 A1 を持つ表 A からすべての行と、「X」の値 B1 を持つ表 B からすべての行が削除されます。

```
DELETE FROM V3 WHERE Z1 = 'A' AND Z2 = 'X'
```

複数の視点で定義された視点を持つ INSTEAD OF トリガー

以下に示す、V1 に定義されたビュー V2 の定義は本質的に挿入、更新、削除ができません。

```
CREATE TABLE T1 (C1 VARCHAR(10), C2 INT)
CREATE TABLE T2 (D1 VARCHAR(10), D2 INT)
CREATE VIEW V1(X1, X2) AS SELECT C1, C2 FROM T1
UNION SELECT D1, D2 FROM T2

CREATE VIEW V2(Y1, Y2) AS SELECT X1, X2 FROM V1
```

V1 に INSTEAD OF トリガー IOT1 を追加しても、V2 は更新可能になりません。

```
CREATE TRIGGER IOT1 INSTEAD OF UPDATE ON V1
REFERENCING OLD AS OLD_ROW NEW AS NEW_ROW
FOR EACH ROW MODE DB2SQL
BEGIN
  UPDATE T1 SET C1 = NEW_ROW.X1, C2 = NEW_ROW.X2 WHERE
    C1 = OLD_ROW.X1 AND C2 = OLD_ROW.X2;
  UPDATE T2 SET D1 = NEW_ROW.X1, D2 = NEW_ROW.D2 WHERE
    D1 = OLD_ROW.X1 AND D2 = OLD_ROW.X2;
END
```

ビュー V2 の元の定義が更新不可のままになるため、ビュー V2 も更新不可のままになります。

BEFORE トリガーおよび AFTER のトリガー INSTEAD OF との使用

INSTEAD OF トリガーをビューに追加しても、基礎となる表に定義された BEFORE および AFTER トリガーが競合することはありません。

```
CREATE TABLE T1 (C1 VARCHAR(10), C2 DATE)
CREATE TABLE T2 (D1 VARCHAR(10))

CREATE TRIGGER AFTER1 AFTER DELETE ON T1
REFERENCING OLD AS OLD_ROW
FOR EACH ROW MODE DB2SQL
  DELETE FROM T2 WHERE D1 = OLD_ROW.C1

CREATE VIEW V1(X1, X2) AS SELECT SUBSTR(T1.C1, 1, 1), DAYOFWEEK_ISO(T1.C2) FROM T1

CREATE TRIGGER IOT1 INSTEAD OF DELETE ON V1
REFERENCING OLD AS OLD_ROW
FOR EACH ROW MODE DB2SQL
  DELETE FROM T1 WHERE C1 LIKE (OLD_ROW.X1 CONCAT '%')
```

ビュー V1 に対する削除処理を行うことで、結果的に AFTER DELETE トリガーである AFTER1 が活動化されます。これは、トリガー IOT1 が表 T1 上で削除を実行するためでもあります。つまり、表 T1 の削除が結果的に AFTER1 トリガーを活動化します。

依存関係にある視点と INSTEAD OF トリガー

ビューに INSTEAD OF トリガーを追加するときに、ビューの定義が複数のビューを参照し、その中に INSTEAD OF トリガーを定義したビューが含まれる場合は、UPDATE、DELETE、および INSERT の 3 種類すべての操作について、INSTEAD OF トリガーを定義します。その際、定義されたビューの機能と、その他の依存関係にあるビューがもつ機能とが混乱しないようにします。

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 ステートメントの処理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を備えるようになります。

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。たとえば、YearToDateTotals 表が不慮の事故によって削除された場合は、 sqlexception 出口ハンドラーが実行します。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 トリガーの変換表:

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、表 StudentProfiles の影響を受けたすべての行に集合関数 MAX を適用します。

```
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処理の後で、トリガーが 1 回処理されます。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、*PGM オブジェクト作成をサポートする高水準言語を使用して、一連のトリガー・アクションを含むプログラムを定義できます。

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

関連情報

データベース内での自動イベントのトリガー

外部トリガーのプログラム例:

このトピックにはサンプルの外部トリガー・プログラムが含まれています。このプログラムは ILE C で書かれ、組み込み SQL を含みます。

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

サンプルのトリガー・プログラム

```
#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;

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);

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  */
/*****************************************************************/
/*   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];

                           /*************************************/
                           /* 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);
    }
}
```

SQL ルーチンのデバッグ

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、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・ビューは、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ルーチン・ライブラリーにあるソース・ファイル QSQDSRC に作成します。ライブラリーを判別できない場合、QSQDSRC は QTEMP に作成されます。 ソース・ビューは、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にご注意ください。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。

関連概念

192 ページの『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、表のグループをアクティブに制御、モニター、管理する方法を提供します。

プロシージャーおよび関数のパフォーマンスの向上

ストアード・プロシージャーおよびユーザー定義関数 (UDF) を作成する場合に、iSeries の SQL プロシージャー型言語プロセッサーは、常にもっとも効率的なコードを生成するわけではありません。しかし、一部に変更を加えることで、必要なデータベース・エンジンの呼び出し回数を減らし、パフォーマンスを向上することができます。

ルーチンの設計で行う変更と、インプリメンテーションで行う変更があります。たとえば、C 言語コンパイラーがホスト変数を処理する方法と、SQL プロシージャー型プロセッサーが処理するホスト変数を要求する方法の相違によって、データベース・エンジンに多くの呼び出しがなされる場合があります。これらの呼び出しは多くのリソースを必要とするため、何度も呼び出されると、パフォーマンスが著しく低下します。

プロシージャーおよび関数のインプリメンテーションの向上

関数またはプロシージャーの処理時間を減らすための有効な方法として、以下のようなシンプルなコーディング技法を用いることをお勧めします。関数は多数の異なるプロシージャーから複数回呼び出される場合があるため、これらのヒントは特に複数の関数で実施することが重要です。

- UDF が呼び出し元と同じスレッドで実行されるように、 NOT FENCED オプションを使用します。
- 同一の入力に同じ結果を戻すプロシージャーおよび UDF で、DETERMINISTIC オプションを使用します。これによって、最適化プログラムは関数呼び出しまたは命令の結果をキャッシュすることができます。このキャッシュで、実行ストリームの関数を呼び出して実行時間を減らすことができます。
- 関数の有効範囲の外ではアクションを行わない UDF で NO EXTERNAL ACTION オプションを使用します。たとえば、外部アクションには、トランザクション要求を完了するために異なる処理を開始する関数があります。

SQL ルーチンの本体で使用されるコーディング技法は、生成された C プログラムの実行時パフォーマンスに大きな影響を与えます。割り当ておよび比較について C コードをさらに使用するようにルーチンを作成することによって、等価の SQL ステートメントによるオーバーヘッドを避けることができます。以下のヒントは、ルーチンが多くの C コードを生成し、 SQL ステートメントの生成を少なくするのに役立ちます。

- 可能であれば、ホスト変数を NOT NULL として宣言します。こうすると、生成コードでヌル値フラグが検査および設定されなくなります。すべての変数を自動的に NOT NULL に設定しないでください。NOT NULL を指定する場合、デフォルト値も指定する必要があります。変数が常にルーチンで使用される場合、デフォルト値が役立つ場合があります。しかし、変数が常に使用されるわけではない場合、デフォルト値を設定すると、追加の初期設定による不必要なオーバーヘッドを引き起こす場合があります。デフォルト値の割り当てを処理する追加のデータベースが必要とされない数値に関しては、デフォルト値が最適です。
- 可能であれば、文字および日付データ・タイプの使用を避けます。たとえば、 0、1、2、または 3 という値を持った、フラグとして使用される変数です。この値が整数の代わりに単一文字の変数として宣言される場合、これは回避できるデータベース・エンジンへの呼び出しとなります。
- 特に変数をカウンターとして使用する場合、ゼロのスケールのある 10 進数の代わりに整数を使用します。
- 一時変数を使用しないでください。 次の例をご覧ください。

```
IF M_days<=30 THEN
 SET I = M_days-7;
 SET J = 23
 RETURN decimal(M_week_1 + ((M_month_1 - M_week_1)*I)/J,16,7);
END IF
```

この例は、以下のように一時変数を使わないものに書き直すことができます。

```
IF M_days<=30 THEN
 Return decimal(M-week_1 + ((M_month_1 - M_week_1)* (M_days-7))/23,16,7);
END IF
```

- 複合の SET ステートメントを 1 つのステートメントに結合します。これは、CCSIDS またはデータ・タイプのために C コードだけを生成できないステートメントに適用されます。

```
SET var1 = function1(var2);
SET var2 = function2();
```

これは 1 つのステートメントに書き直すことができます。

```
SET var1 = function1(var2), var2 = function2();
```

- IF (x AND y) の代わりに IF () ELSE IF () ... ELSE ... 構成体を使用することによって、不必要な比較を避けることができます。
- できる限り SELECT ステートメントで処理を行います。

```
SELECT A INTO Y FROM B;
 SET Y=Y||'X';
```

この例を以下のように書き直します。

```
SELECT A || 'X' INTO Y FROM B
```

- 必要でない限り、ループ内での文字または日付の比較を行うのを避けます。もし必要であれば、比較がループより先に行われるように移動して、ループ内で使用される整変数がその比較によって設定されるようにループを書き直すことができます。これによって、複雑な式が 1 回で評価されます。ループ内の整数比較は、生成された C コードによって行われるため、より効率的になります。
- 使用されない可能性のある変数を設定しないようにします。たとえば、変数が IF ステートメントの外で設定される場合、変数は実際には IF ステートメントのすべてのインスタンスで使用されることになります。そうでない場合には、実際に使用される IF ステートメントの部分でのみ変数を設定します。
- 可能な場合、コードのセクションを単一の SELECT ステートメントに置き換えます。次のコードの断片をご覧下さい。

```
SET vnb_decimal = 4;
cdecimal:
   FOR vdec AS cdec CURSOR FOR
   SELECT nb_decimal
   FROM K$FX_RULES
   WHERE first_currency=Pi_curl AND second_currency=P1_cur2
   DO
    SET vnb_decimal=SMALLINT(cdecimal.nb_decimal);
END FOR cdecimal;

 IF vnb_decimal IS NULL THEN
  SET vnb_decimal=4;
 END IF;
 SET vrate=ROUND(vrate1/vrate2,vnb_decimal);
 RETURN vrate;
```

このコードの断片を次のように書き直すと、さらに効率的になります。

```
RETURN( SELECT
   CASE
    WHEN MIN(nb_decimal) IS NULL THEN ROUND(Vrate1/Vrate2,4)
   ELSE ROUND(Vrate1/Vrate2,SMALLINT(MIN(nb_decimal)))
   END
   FROM K$FX_RULES
   WHERE first_currency=Pi_curl AND second_currency=Pi_cur2);
```

- 両方のオペランドの CCSID が同じである場合、片方の CCSID が 65535 である場合、 CCSID が UTF8 でない場合、および文字データの切り捨てが不可能な場合に、 C コードは文字データの割り当ておよび比較でのみ使用されます。変数の CCSID が指定されていない場合、CCSID はプロシージャーが呼び出されるまで決定されません。この場合、実行時に CCSID を決定し比較するためにコードを生成する必要があります。代替照合順序が指定されているか、または *JOBRUN が指定されている場合、文字比較のために C コードは生成されません。
- 割り当てで共に使用される数値変数には、同じデータ・タイプ、長さ、および位取りを使用します。切り捨てが不可能な場合にのみ、C コードは生成されます。

```
DECLARE v1, v2 INT;
SET v1 = 100;
SET v1 =  v2;
```

パフォーマンスのためのルーチンの再設計

インプリメンテーションのすべてのヒントに従ったとしても、プロシージャーまたは関数のパフォーマンスは要求レベルにまで達しない場合があります。そのような場合には、プロシージャーまたは UDF の設計を調べて、パフォーマンスを向上するために行える変更があるかどうかを確認する必要があります。

その変更には、2 つのタイプがあります。

最初の変更は、プロシージャーが行うデータベース呼び出しまたは関数呼び出しの数を減らすものであり、SQL ステートメントに変換できるコードのブロックを探すのに似た処理です。コードに追加のロジックを加えることによって、呼び出し回数を減らすことができます。

もう 1 つの設計変更はさらに高度なもので、同じ結果を異なる方法で得るように関数全体を構成し直すことです。たとえば、関数が特定の基準セットに合う経路を検出するために SELECT ステートメントを使用して、そのステートメントを動的に実行しているとします。その場合、関数が実行している作業を調べ、ロジックを変更して、関数が静的 SELECT 照会を使用して回答を検出できるようにし、それによってパフォーマンスが向上するようにすることができます。

ネストされた複合ステートメントを使用して、例外処理やカーソルをローカライズする必要もあります。いくつかの特定のハンドラーが指定される場合、各ステートメントの後にエラーが生じたかどうかを検査するためにコードが生成されます。複合ステートメントでエラーが生じる場合、カーソルを閉じ、保管ポイントを処理するためのコードも生成されます。複数のハンドラーおよび複数のカーソルのある単一の複合ステートメントを持つルーチンでは、各 SQL ステートメントの後に各ハンドラーとカーソルを処理するためにコードが生成されます。ハンドラーおよびカーソルの有効範囲をネストされた複合ステートメントに限定した場合、ハンドラーおよびカーソルはネストされた複合ステートメント内のみで検査されます。

次のルーチンでは、SQLSTATE '22H11' のエラーを検査するコードが、 lab2 複合ステートメント内のステートメントのためだけに生成されます。このエラーの検査は lab2 ブロックの外のルーチンのステートメントに対しては行われません。 SQLEXCEPTION エラーを検査するコードは、lab1 および lab2 ブロックの両方のすべてのステートメントに対して生成されます。同様に、カーソル c1 を閉じる場合のエラー処理は lab2 のステートメントに限定されます。

```
Lab1: BEGIN
   DECLARE var1 INT;
   DECLARE EXIT HANDLER FOR SQLEXCEPTION
         RETURN -3;
   lab2:  BEGIN
      DECLARE EXIT HANDLER FOR SQLSTATE '22H11'
           RETURN -1;
      DECLARE c1 CURSOR FOR SELECT col1 FROM table1;
```

```
      OPEN c1;
      CLOSE c1;
   END lab2;
END Lab1
```

ルーチン全体の再設計には多くの時間と労力が必要とされるため、アプリケーションを全体的に調べるよりも、パフォーマンスの主なボトルネックとなっているルーチンを調べてください。既存のパフォーマンスのボトルネックを再設計するよりも、むしろアプリケーションの設計段階で時間を費やし、設計によるパフォーマンス効果について検討することのほうが重要です。使用頻度が高いと予想されるアプリケーションに注目し、パフォーマンスを考慮に入れてそれらを設計すると、後からそれらの領域を再設計しなくて済みます。

特別なデータ・タイプの処理

INTEGER および CHARACTER などのほとんどのデータ・タイプには、特別な処理特性はありません。しかし、使用するために特別な機能やロケーターを必要とするデータ・タイプが少数あります。

ラージ・オブジェクト (LOB) の使用

VARCHAR データ・タイプ、VARGRAPHIC データ・タイプ、および VARBINARY データ・タイプの記憶域限界は 32 KB (KB = 1024 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。 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 = 1 073 741 824 バイト) のストリングとして保管するための 3 つのデータ・タイプがあります。

3 つのデータ・タイプは次のとおりです。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。それぞれの表には大量の関連 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 GB を超えることはできませんが、1 つの表には、256 GB に近い LOB データを入れることができます。

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、ホスト変数はプログラムの記憶域を使用するので、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。

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・データ・タイプはここで定義されます。

- 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 VARBINARY (32K バイトが限度) よりも大容量のバイナリー・データを保管できます。このデータ・タイプは、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保管するのに適しています。
- 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保管するのに適しています。情報のコード・ページ変換がサポートされています。

- 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管するのに適しています。この場合も、情報のコード・ページ変換がサポートされています。

ラージ・オブジェクト・ロケーターについて

ラージ・オブジェクト (LOB) ロケーターは、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します。

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、UDF に渡されることも、UDF から戻されることもできます。 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。

プログラムの中でどのように LOB 値を使用するかによって、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。

関連資料

213 ページの『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。

209 ページの『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。

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。

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。この例では、C 言語が使用されるので、 check_error は CHECKERR として再定義され、util.c ファイルに置かれます。

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

関連概念

208 ページの『ラージ・オブジェクト・ロケーターについて』
ラージ・オブジェクト (LOB) ロケーターは、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します。

例: C の LOBLOC.SQC:

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

```
 #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 */
```

例: COBOL の LOBLOC.SQB:

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

```
 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ます。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、このシステムがサーバーです。

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はファイル参照変数構造の data_length フィールドを介して、長さを戻します。COBOL 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。

関連概念

208 ページの『ラージ・オブジェクト・ロケーターについて』
ラージ・オブジェクト (LOB) ロケーターは、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します。

関連情報

統合ファイル・システム

例: ファイルへの文書の抽出

次のプログラムの例は、文字ラージ・オブジェクト (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 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、デフォルト値で、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 ファイルに置かれます。

COBOL
CHECKERR は checkerr.cbl という名前の外部プログラムです。

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

例: C の LOBFILE.SQC:

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

```
 #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 */
```

例: COBOL の LOBFILE.SQB:

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

```
 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。

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。

```
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 から 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 から 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。

ユーザー定義特殊タイプ (UDT) の使用

ユーザー定義の特殊タイプは、設定済みの組み込みデータ・タイプよりさらに DB2 UDB の機能を発揮させるメカニズムです。

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。

これらの利点は次の UDT に関連しています。

- **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。

- **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。

- **一貫性のある動作**

  限定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。
- **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。
- **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。
- **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。

関連概念

13 ページの『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。

UDT の定義

UDT は CREATE DISTINCT TYPE ステートメントで定義されます。

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システムが使用するタイプです。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、システムは、以下を行う場合に常に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。

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 CREATE、DROP、または COMMENT ON ステートメントのメイン・オブジェクトである場合を除きます。

関連資料

188 ページの『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検索をそのスキーマに制限します。

関連情報

CREATE DISTINCT TYPE ステートメント

例: 通貨:

この例では、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。ただし照会の際に DB2 が、異なる通貨が互いに直接比較したり、操作したりしないようにします。

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 EURO AS DECIMAL (9,2)
```

例: 履歴書:

この例では、会社への応募者が記入した応募書式を表で維持管理し、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:

それぞれの国の売上高を保持する表を定義するとします。

以下の表を作成します。

```
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         EURO)
```

上の例の UDT は、219 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。

例: 応募用紙:

応募者が記入した書式を保持する表を定義したいとします。

以下のように表を作成します。

```
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、デフォルト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*限定タイプの指定* があります。限定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。

限定タイプの指定は、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、ユーロをカナダ・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。

限定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。

例: UDT の使用

これらは UDT の使用例です。

例: UDT と定数との比較:

米国で、1998 年の 7 月 (7/98) に US $100 000.00 を超える売上のあった製品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  = 1998
```

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 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 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。 上の照会は、以下のように書く場合もあります。

```
SELECT PRODUCT_ITEM
  FROM US_SALES
  WHERE  TOTAL > CAST (100000 AS us_dollar)
  AND    MONTH = 7
  AND    YEAR  = 1998
```

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。 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。

以下に示すユーロを米国ドルに変換する関数は、上の例に似ています。

```
CREATE FUNCTION EURO_TO_US_DOUBLE(DOUBLE)
  RETURNS DOUBLE
  EXTERNAL NAME 'MYLIB/CURRENCIES(C_EURO_US)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC

CREATE FUNCTION EURO_TO_US_DEC (DECIMAL(9,2))
  RETURNS DECIMAL(9,2)
```

```
  SOURCE EURO_TO_US_DOUBLE(DOUBLE)

CREATE FUNCTION US_DOLLAR(EURO) RETURNS US_DOLLAR
  SOURCE EURO_TO_US_DEC (DECIMAL())
```

例: UDT が関係する比較:

2003 年 3 月 (3/03) に、カナダやドイツよりも米国で多く売れた製品は何かを知りたいとします。

以下の SELECT ステートメントを出します。

```
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  = 2003
  AND CDN.MONTH = 3
  AND CDN.YEAR  = 2003
  AND GERMAN.MONTH = 3
  AND GERMAN.YEAR  = 2003
```

米国ドルを直接カナダ・ドルやユーロ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、ユーロ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、ユーロで SUM をサポートしたいとします。

関数ステートメントは次のようになります。

```
CREATE FUNCTION SUM (EURO)
  RETURNS EURO
  SOURCE SYSIBM.SUM (DECIMAL())
```

2004 年のドイツでの、各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 = 2004
  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。

関連資料

224 ページの『例: さまざまな UDT が関係する割り当て』
「例: UDT が関係する、ソース化された UDF」のユーロに対してソース化された UDF と同様に、組み込み SUM 関数上でソース化された UDF を米国ドル用とカナダ・ドル用にそれぞれ SUM をサポートするよう定義します。

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。

関連資料

『例: 動的 SQL での割り当て』
動的 SQL での UDT が関係する割り当ての例で使用されている同じステートメントを使用する場合は、 パラメーター・マーカーを使用できます。

例: 動的 SQL での割り当て:

動的 SQL での UDT が関係する割り当ての例で使用されている同じステートメントを使用する場合は、パラメーター・マーカーを使用できます。

ステートメントは次のようになります。

```
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 のキャスト指定を使用して、DB2 に、パラメーター・マーカーのタイプが CLOB(32K) (UDT 列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。

関連資料

223 ページの『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。

例: さまざまな UDT が関係する割り当て:

「例: UDT が関係する、ソース化された UDF」のユーロに対してソース化された UDF と同様に、組み込み SUM 関数上でソース化された UDF を米国ドル用とカナダ・ドル用にそれぞれ SUM をサポートするよう定義します。

```
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_04
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR)

CREATE TABLE GERMAN_SALES_04
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR)

CREATE TABLE CANADIAN_SALES_04
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR)

INSERT INTO US_SALES_04
  SELECT PRODUCT_ITEM, SUM (TOTAL)
  FROM US_SALES
  WHERE YEAR = 2004
  GROUP BY PRODUCT_ITEM

INSERT INTO GERMAN_SALES_04
  SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
  FROM GERMAN_SALES
  WHERE YEAR = 2004
  GROUP BY PRODUCT_ITEM

INSERT INTO CANADIAN_SALES_04
  SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
  FROM CANADIAN_SALES
  WHERE YEAR = 2004
  GROUP BY PRODUCT_ITEM
```

カナダ・ドルとユーロ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。

関連資料

223 ページの『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、ユーロで SUM をサポートしたいとします。

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 ステートメントは次のようになります。

```
  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、ユーロ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 とそのソース・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 の使用例

以下の例では、ユーザー定義タイプ (UDT)、ユーザー定義関数 (UDF)、および大規模なオブジェクト (LOB) を複雑なアプリケーションで同時に使用する方法について示します。

例: UDT と UDF の定義

会社に送られてきた電子メール (e-mail) を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 for iSeries 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 を使用することで、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ためです。

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。

ステートメントは次のようになります。

```
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。つまり、UDT 電子メールは完全にカプセル化されていると言え

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 とし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) であるため、 BLOB ロケーターが UDT で定義された UDF の引数として使用されるたびに、BLOB ロケーターを明示的に UDT に変換します。

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考えに基づきます。

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、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(「SQL解説書」の『組み込み関数』を参照してください)。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、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、各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。しかし、クリップ自体は、データ・リンク列を使用して参照されます。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再生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。

関連情報

データ・タイプ

データ・リンクのリンク制御レベル

異なるリンク制御を用いてデータ・リンク列を作成できます。

異なる制御レベルは次のとおりです。

NO LINK CONTROL:

NO LINK CONTROL レベルと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。

FS 許可を使用した FILE LINK CONTROL:

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 レベル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。

DB 許可を使用した FILE LINK CONTROL:

データ・リンク列が、データベース (DB) 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・リンク列をもつ 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 以外のデフォルト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開かれ、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。システム上にデータ・リンク・ファイル・マネージャー (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ための CL コマンドは、データ・リンク・ファイル・マネージャーへのプレフィックスの追加 (ADDPFXDLFM) コマンドです。

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。URL への以下のリンクは、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。

```
http://TESTSYS1/mydir/datalinks/videos/file1.mpg
```

あるいは

```
file://TESTSYS1/mydir/datalinks/text/story1.txt
```

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、通常使用される関数ではありません。なぜならこの関数が実行できるのは、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。

注:

1. 以下のディレクトリー、またはどのサブディレクトリーも、データ・リンクのプレフィックスとしては使用しないでください。
   - /QIBM
   - /QReclaim
   - /QSR
   - /QFPNWSSTG
2. さらに、プレフィックスがいずれかの基本ディレクトリーのサブディレクトリーでない場合は、以下のような共通の基本ディレクトリーも使用しないでください。
   - /home
   - /dev
   - /bin
   - /etc
   - /tmp
   - /usr
   - /lib

ホスト・データベースの追加

ホスト・データベースは、リンク要求が出される起点となるリレーショナル・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 Add Host Database to DataLink File Manager (ADDHDBDLFM) コマンドです。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。次のコマンドを使用します。

```
ADDHDBDLFM HOSTDBLIB((TESTDB) (PRODDB)) HOSTDB(TESTSYS1)
```

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。

異なる環境で SQL を使用します。

SQL は様々な環境で使用することができます。その一部をここで取り上げます。

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。

関連資料

94 ページの『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。

関連情報

DECLARE CURSOR ステートメント

CLOSE ステートメント

FETCH ステートメント

DELETE ステートメント

UPDATE ステートメント

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。

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、これがデフォルト値です。
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。

例: カーソルの使用

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。

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

表 41. 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.	237 ページの『ステップ 1: カーソルを定義する』
EXEC SQL **OPEN** THISEMP END-EXEC.	238 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.	238 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.	239 ページの『ステップ 4: カーソルを用いて行を取り出す』
... 部門 11 のすべての 社員について、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。	239 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.	240 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。	

表 41. 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
CLOSE-THISEMP. EXEC SQL **CLOSE** THISEMP END-EXEC.	240 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 42. 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.	237 ページの『ステップ 1: カーソルを定義する』
EXEC SQL **OPEN** THISEMP END-EXEC.	238 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.	238 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』
...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。	239 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.	240 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 cursor-name 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  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、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く、カーソルがスクロール可能でない場合は、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 つ以上の表またはビューを指定している。
- 最初の FROM 文節で読み取り専用ビューを指定している。
- 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は UNION、UNION ALL、EXCEPT、あるいは INTERSECT 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、 SENSITIVE キーワードと FOR UPDATE OF 文節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。
- SCROLL キーワードが指定されており、FOR UPDATE OF 文節が指定されておらず、 SENSITIVE キーワードが指定されていない。
- 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。

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ことができます。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可能です。

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。

ステートメントの形式は次のとおりです。

```
  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、プリコンパイル時に 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、プログラムのプリコンパイル時に 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 で表またはビュー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 プリ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。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。

  関連情報

  組み込み SQL プログラミング

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。

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

```
      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。

関連情報

SQLCA (SQL 通信域)

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。

次の PL/I の例を検討してください。

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

```
      *....+....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';
```

```
...
    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:
```

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。サンプル表に記載されているサンプルの 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、結果表に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。

関連資料

306 ページの『DB2 UDB for iSeries サンプル表』
このトピックには、このトピック、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。

関連情報

付録 D. SQLDA (SQL 記述子域)

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: CRTSQL*xxx* 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。CRTSQL*xxx* 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、『動的 SQL アプリケーション』を参照してください。

関連概念

『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。

関連情報

コミットメント制御

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以下の処理を行います。

- 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 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. 動的 SQL の処理は、実行時にステートメントを完全に処理しなければならない場合があるため、静的 SQL の処理に比べ、オーバーヘッドが大幅に高くなる可能性があります。最悪の場合には、ステートメントを実行する前に、準備、バインド、データベースによる最適化が完全に行われていなければなりません。それ以外の場合でも、準備が整う前にステートメントが実行されると、データベースがアルゴリズムを使用してキャッシュを保持してしまうため、処理の一部がスキップされる場合があります。このような場合にも、DB2 for iSeries は、動的 SQL ステートメントで優れたパフォーマンスを実現します。動的アプリケーションのパフォーマンスを重視する場合は、QSQPRCED API を使用した拡張動的機能の使用を考慮してください。この機能により、アプリケーションは SQL ステートメントの永続キャッシュを保持できるようになり、アプリケーション実行時のランタイム・オーバーヘッドが大幅に抑えられます。
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、ブロック化を使用してカーソル行を検索するので、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 を持つカーソル使用の利点は、上記2 に説明したとおりです。

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/400 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 の各プログラムの助けが必要です。

関連概念

265 ページの『対話式 SQL の使用』
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。

関連資料

245 ページの『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。

248 ページの『PREPARE ステートメントおよび EXECUTE ステートメントの使用』
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。

248 ページの『非 SELECT ステートメントの処理』
動的 SQL の非 SELECT ステートメントをビルドするには、ビルドしようとしている SQL ステートメントが、動的に実行可能であることを確認してから SQL ステートメントをビルドする必要があります。

Process Extended Dynamic SQL (QSQPRCED) API

関連情報

SQL ステートメントで許可されるアクション

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、プリ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。

関連情報

iSeries Access for Windows: プログラミング

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 は NOT 記号 (¬) が値 'BA'X に置かれることを想定しているという意味です。

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。

非 SELECT ステートメントの処理

動的 SQL の非 SELECT ステートメントをビルドするには、ビルドしようとしている SQL ステートメントが、動的に実行可能であることを確認してから SQL ステートメントをビルドする必要があります。

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;
```

関連概念

246 ページの『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。

265 ページの『対話式 SQL の使用』
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。

PREPARE ステートメントおよび EXECUTE ステートメントの使用:

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ステートメント名を提供します。 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。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。

関連概念

246 ページの『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。

SELECT ステートメントの処理と記述子の使用

SELECT ステートメントには、2 つの基本タイプがあります。**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、SQL 記述子は必要ありません。

| 可変リスト SELECT ステートメントを処理するには、最初に SQL 記述域 (SQLDA) 構造を宣言するか、
| SQLDA を割り当てる必要があります。 SQL 記述子のどちらの形式も、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ったりするのに使用することができ
| ます。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、SQL 記述域 (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 (可変リスト SELECT ステートメントを使用するかのように、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 SELECT 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。 REXX は SQLDA 構造を使用して
| 定義された SQL 記述子のみをサポートします。割り振られた SQL 記述子はサポートされません。

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. 割り振られた SQL 記述子の場合、ALLOCATE DESCRIPTOR ステートメントを実行して、使用する予定の記述子を定義します。
6. 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. SQLDA を使用していて、取得し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 ステートメントを再発行しなければなりません。

      割り振られた SQL 記述子を使用していて、その記述子の大きさが十分でない場合は、記述子を割り振り解除し、エントリー数を大きくして割り振り直し、DESCRIBE ステートメントを再発行します。
7. SQLDA 記述子の場合、取得したデータの 1 行分を収めるのに必要なだけの記憶域量を割り振る。
8. SQLDA 記述子の場合、取得し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 に入れる。
9. FETCH で行を取り出す。
10. SQL 記述子で戻されたデータを処理する。
11. SQL 戻りコードが戻されたら、それを処理する。
12. データの終わりが現れたとき、カーソルをクローズする。
13. 割り振られた SQL 記述子の場合、DEALLOCATE DESCRIPTOR ステートメントを実行して、記述子を削除します。

関連資料

255 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』
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、表示から読み取られるか、他のアプリケーションから渡される、あるいは、ご使用のアプリケーションで動的にビルドされる可能性もあります。

SQL 記述域:

動的 SQL は SQL 記述域 (SQLDA) を使用して、SQL とアプリケーション間で SQL ステートメントに関する情報を渡します。記述子は DESCRIBE ステートメント、DESCRIBE INPUT 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あり、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・プログラムに提供します。 DESCRIBE INPUT では、SQL 記述子域はアプリケーション・プログラムに対し、準備されたステートメントのパラメーター・マーカーに関する情報を提供します。 DESCRIBE TABLE では、SQLDA は表またはビュー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、ホスト変数を使用するためにそのデータ値を変更し、そして FETCH ステートメントで同じ記述子を再使用します。

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。

SQLDA には 2 つのタイプがあります。1 つは ALLOCATE DESCRIPTOR ステートメントで定義されます。もう 1 つは SQLDA 構造を使用して定義されます。

ALLOCATE DESCRIPTOR は REXX ではサポートされていません。 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/400 にはポインターを設定する手段がないため、SQLDA は、PL/I、C、C++、COBOL、または ILE RPG の各プログラムにより、RPG/400 プログラムの外部で設定する必要があります。その後、そのプログラムが RPG/400 プログラムを呼び出す必要があります。

関連情報

SQLCA (SQL 通信域)

SQLDA (SQL 記述域)

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。

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、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。 有効な値の表については、「SQLTYPE と SQLLEN」を参照してください。 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。i5/OS 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。このフィールドの CCSID データのレイアウトについては、「SQLDATA または SQLNAME 内の CCSID の値」を参照してください。

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。デフォルト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、ジョブのデフォルト値の CCSID が使用されます。

グラフィック・ホスト変数のデフォルト値は、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。

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。

関連資料

『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、表示から読み取られるか、他のアプリケーションから渡される、あるいは、ご使用のアプリケーションで動的にビルドされる可能性もあります。

関連情報

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方法

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、表示から読み取られるか、他のアプリケーションから渡される、あるいは、ご使用のアプリケーションで動的にビルドされる可能性もあります。

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処理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らないとします。

```
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。このコンテキスト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 43. SQLDA ヘッダー

説明	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 44. SQLVAR 要素 1

説明	値
SQLTYPE	453
SQLLEN	3
SQLDATA (3:4)	37
SQLNAME	8 WORKDEPT

表 45. SQLVAR 要素 2

説明	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。

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。

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。このコンテキストにおいて意味のある項目のみを示します。

表 46. SQLDA ヘッダー

説明	値
SQLAID	'SQLDA'
SQLDABC	1616
SQLN	20
SQLD	2

表 47. SQLVAR 要素 1

説明	値
SQLTYPE	453
SQLLEN	3
SQLDATA	CHAR(3) 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

表 48. SQLVAR 要素 2

説明	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 からの結果の行が、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。

関連資料

250 ページの『可変リスト SELECT 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。

252 ページの『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。

例: 割り振られた SQL 記述子を使用したステートメントの選択:

アプリケーションが、動的 SELECT ステートメント (一度使用して次に使用するまでに変更される) を処理できる必要があるとします。このステートメントは、画面から読み取り可能であるか、他のアプリケーションから受け渡しできるか、あるいはご使用のアプリケーションで動的にビルドできるかのいずれかになります。

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処理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らないとします。

```
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。これを実行するには、必要と思われる SQL 記述子に、最大数のエントリーを割り振る必要があります。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。

```
EXEC SQL
ALLOCATE DESCRIPTOR 'mydescr' WITH MAX 20;
```

これで、記述子が割り振られ、列情報を入手するために DESCRIBE ステートメントを実行できるようになります。

```
EXEC SQL
DESCRIBE S1 USING DESCRIPTOR 'mydescr';
```

DESCRIBE ステートメントが実行されると、SQL はステートメントの選択リストに関する情報を示す値を「mydescr」によって定義された SQL 記述域に入れます。

記述子に割り振られたエントリーが十分でないと DESCRIBE が判定した場合、 SQLCODE +239 が発行されます。この診断の一部として、 2 番目の置き換えられるテキストの値は、必要なエントリーの数を示します。以下のコード・サンプルでは、この条件を検出する方法について解説し、より大きなサイズで割り振られた記述子を示します。

```
/* Determine the returned SQLCODE from the DESCRIBE statement */
EXEC SQL
  GET DIAGNOSTICS CONDITION 1: returned_sqlcode = DB2_RETURNED_SQLCODE;

if returned_sqlcode = 239 then do;

/* Get the second token for the SQLCODE that indicated
     not enough entries were allocated */

EXEC SQL
    GET DIAGNOSTICS CONDITION 1: token = DB2_ORDINAL_TOKEN_2;
  /* Move the token variable from a character host variable into an integer host variable */
  EXEC SQL
    SET :var1 = :token;
  /* Deallocate the descriptor that is too small */
  EXEC SQL
    DEALLOCATE DESCRIPTOR 'mydescr';
```

```
|    /* Allocate the new descriptor to be the size indicated by the retrieved token */
|    EXEC SQL
|      ALLOCATE DESCRIPTOR 'mydescr' WITH MAX :var1;
|    /* Perform the describe with the larger descriptor */
|    EXEC SQL
|     DESCRIBE s1 USING DESCRIPTOR 'mydescr';
|  end;
```

| この時点で、記述子には SELECT ステートメントに関する情報が含まれています。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 SQL の場合、SELECT INTO ステートメントは使用できません。カーソルを使用する必要があります。

```
|  EXEC SQL
|    DECLARE C1 CURSOR FOR S1;
```

| 完全な SELECT ステートメントの代わりとして、準備したステートメントの名前が、カーソルの宣言に使用されているはずです。ここまでの処理で、選択した各行を順に読み取って処理しながら、選択した行をループすることができます。以下のコード・サンプルではこの操作の実行方法を示しています。

```
|  EXEC SQL
|    OPEN C1;
|
|  EXEC SQL
|      FETCH C1 USING SQL DESCRIPTOR 'mydescr';
|  do while not at end of data;
|
|   /* process current data returned  (see below for discussion of doing this) */
|
|  /* then read the next row */
|
|   EXEC SQL
|      FETCH C1 USING SQL DESCRIPTOR 'mydescr';
|  end;
|
|  EXEC SQL
|    CLOSE C1;
```

| カーソルがオープンされます。次に SELECT ステートメントからの結果の行が、FETCH ステートメントを使用して一度に 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 SQL に対し、記述子域に結果を返すように指示します。

| FETCH が処理された後、 GET DESCRIPTOR ステートメントを使用して値を読み取れるようになります。使用された記述子のエントリー数を示すヘッダー値を最初に読み取る必要があります。

```
|  EXEC SQL
|    GET DESCRIPTOR 'mydescr' :count = COUNT;
```

| 次に、各記述子エントリーに関する情報を読み取ることができます。結果列のデータ・タイプを判別した後、別の GET DESCRIPTOR を実行して、実際の値を戻すことが可能です。標識の値を取得するには、INDICATOR アイテムを指定します。INDICATOR アイテムが負の値である場合、DATA アイテムの値は定義されません。別の FETCH が完了するまで、記述子アイテムがそれらの値を保持します。

```
|  do i = 1 to count;
|    GET DESCRIPTOR 'mydescr' VALUE :i  /* set entry number to get */
|                           :type = TYPE,                    /* get the data type */
|                           :length = LENGTH,                /* length value */
|                           :result_ind = INDICATOR;
|   if result_ind >= 0 then
|      if type = character
|       GET DESCRIPTOR 'mydescr' VALUE :i
|                          :char_result = DATA;       /* read data into character field */
```

```
|      else
|      if type = integer
|         GET DESCRIPTOR 'mydescr' VALUE :i
|                         :int_result = DATA;       /* read data into integer field */
|    else
|        /* continue checking and processing for all data types that might be returned */
|  end;
```

| 他の記述子アイテムの一部には、その結果データの処理方法を決定するため、確認する必要があるものも存
| 在します。PRECISION、SCALE、DB2_CCSID、および DATETIME_INTERVAL_CODE などがその一例で
| す。データ値読み込み DATA 値を持つホスト変数は、読み込むデータと同じデータ・タイプおよび
| CCSID を持つ必要があります。データ・タイプの長さが多岐にわたる場合、ホスト変数の実際の長さより
| 長く宣言することが可能です。その他のデータ・タイプについては、正確な長さで宣言しなければなりませ
| ん。

| NAME、DB2_SYSTEM_COLUMN_NAME、および DB2_LABEL は、結果の列に対して名前に関連した値
| の取得に使用できます。GET DESCRIPTOR ステートメント、および TYPE 値の定義に関して戻されたア
| イテムの詳細な情報については、 GET DESCRIPTOR を参照してください。

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・マーカーを使用する場合は、実行時まで、パラメーターに対してアプリケーションがデー
| タ・タイプや値を設定する必要はありません。 OPEN ステートメント上に記述子を指定することにより、
| SELECT ステートメントでパラメーター・マーカーの値を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、記述子文節を伴う 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記述子エントリー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、その値に関する有効な定義が入っている記述子を識別するものでなければなりません。この記述
| 子は、SELECT リストの一部となっているデータ項目についての情報を返すためには使用されません。こ
| の記述子からは、SELECT ステートメントのパラメーター・マーカーを置き換えるために使用される値に
| ついての情報が得られます。この情報は、アプリケーションから渡されるので、アプリケーションは、記述
| 子の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。これで、SQL は記述子を使
| 用して、実際の値でパラメーター・マーカーを置き換えられるようになります。

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・マーカーを値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 (SQLTYPE, SQLLEN, および SQLNAME)。
- 標識変数が必要かどうか

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。

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ステップを実行する必要があります。これは SQLDA または割り振られた記述子を使用して行うことができます。

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. そのサイズの SQLDA を割り振るか、ALLOCATE DESCRIPTOR を使用してエントリーの数に記述子を割り振ります。これは REXX には適用されません。
4. SQLDA の場合、SQLN および 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。割り振られた記述子の場合、SET DESCRIPTOR を使用して、パラメーター・マーカーの数に COUNT エントリーをセットします。
5. SQLDA の場合、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. 各パラメーター・マーカーは以下を行います。
   a. データ・タイプ、長さ、および標識を判別する。
   b. SQLDA の場合、SQLTYPE および SQLLEN を各パラメーター・マーカーにセットする。割り振られた記述子の場合、SET DESCRIPTOR を使用して、各パラメーター・マーカーに TYPE、LENGTH、PRECISION、および SCALE のエントリーをセットする。
   c. SQLDA の場合、入力値を保持するためにストレージを割り振る。
   d. SQLDA の場合、ストレージにこれらの値をセットする。
   e. SQLDA の場合、SQLDATA および SQLIND (可能な場合) を各パラメーター・マーカーにセットする。割り振られた記述子の場合、SET DESCRIPTOR を使用して、各パラメーター・マーカーに、DATA および INDICATOR (可能な場合) のエントリーをセットする。
   f. 文字変数が使用され、それらにジョブのデフォルト CCSID 以外の CCSID がある場合、またはグラフィック変数が使用され、ジョブ CCSID に関連付けられた DBCS CCSID 以外の CCSID がある場合。
      - SQLDA の場合、適宜 SQLNAME (REXX では SQLCCSID) をセットする。
      - 割り振られた SQL 記述子の場合、SET DESCRIPTOR を使用して DB2_CCSID 値をセットする。
   g. OPEN ステートメントに USING DESCRIPTOR 文節 (SQLDA の場合)、または USING SQL DESCRIPTOR 文節 (割り振られた記述子の場合) を付けて発行し、カーソルをオープンしてパラメーター・マーカーの値をそれぞれ置き換えます。

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。

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。

Java プログラムからのデータ・アクセス

Developer Kit for Java データベース・コネクティビティー (JDBC) 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

関連情報

IBM Developer Kit for Java (JDBC driver) を使用するための設定

Domino を使用したデータ・アクセス

Domino® for iSeries は Domino サーバー製品であり、これを使用することにより、DB2 UDB for iSeries データベースと Domino データベースのデータを双方向で統合できます。

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。

関連情報

Domino for iSeries

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・アプリケーション間、およびサーバーとの間で、データを効果的に共有できるようになります。

関連情報

ODBC の管理

i5/OS ポータブル・アプリケーション・ソリューション環境 (i5/OS PASE) を使用したデータへのアクセス

i5/OS PASE は iSeries システムで実行される AIX® アプリケーションや、UNIX などの他のアプリケーションのための統合ランタイム環境です。

関連情報

i5OS PASE

iSeries Access for Windows OLE DB Provider を使用したデータ・アクセス

iSeries Access for Windows の OLE DB Provider は、Programmer's Toolkit と組み合わせて使用することによって、iSeries クライアント/サーバー・アプリケーションの開発を Windows クライアント PC から迅速かつ簡単に行うことができます。

iSeries Access for Windows OLE DB Provider は、iSeries プログラマーに対し、iSeries の論理および物理 DB2 Universal Database™ (UDB) for iSeries データベース・ファイルへのレコード・レベルのアクセス・インターフェースを提供します。加えて、SQL、データ待ち行列、プログラム、およびコマンドをサポートします。 Visual Basic を使用している場合には、Visual Basic ウィザードによって、カスタマイズされた作業アプリケーションを単純かつ簡単に開発できるようになります。

関連情報

iSeries Access for Windows OLE DB Provider

NET.data を使用したデータ・アクセス

Net.Data は、サーバー上で実行するアプリケーションであり、これによって Web マクロと呼ばれる 動的 Web 文書を簡単に作成することができます。 Net.Data 用に作成される Web マクロには、CGI-BIN アプリケーションの機能性を備えた HTML の簡易性も持ち合わせます。

Net.Data によって、ライブ・データを静的 Webページに簡単に追加することができます。ライブ・データには、データベース、ファイル、アプリケーション、およびシステム・サービスに保管される情報が含まれます。

関連情報

HTTP Server の Net.data プログラム

Linux パーティションを介したデータ・アクセス

IBM とさまざまな Linux® ディストリビューターは、Linux オペレーティング・システムに iSeries サーバーの信頼性を組み込むため、互いにパートナーとして取り組んできました。

Linux は、 iSeries に新世代の Web ベース・アプリケーションをもたらします。 IBM は 2 次論理区画で実行されるよう Linux PowerPC® カーネルを修正し、そのカーネルを Linux コミュニティーに提供しました。

関連情報

Linux と iSeries サーバー

分散リレーショナル・データベース (DRDA) を使用したデータへのアクセス

分散リレーショナル・データベース は、相互に接続されたコンピュータ・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。 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。

関連資料

281 ページの『分散リレーショナル・データベース機能と SQL』
分散リレーショナル・データベース は、相互に接続されたコンピュータ・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。

対話式 SQL の使用

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。
- リスト選択機能を開始する。
- 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。

注:

1. 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ます。
2. コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

関連概念

246 ページの『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。

関連資料

248 ページの『非 SELECT ステートメントの処理』
動的 SQL の非 SELECT ステートメントをビルドするには、ビルドしようとしている SQL ステートメントが、動的に実行可能であることを確認してから SQL ステートメントをビルドする必要があります。

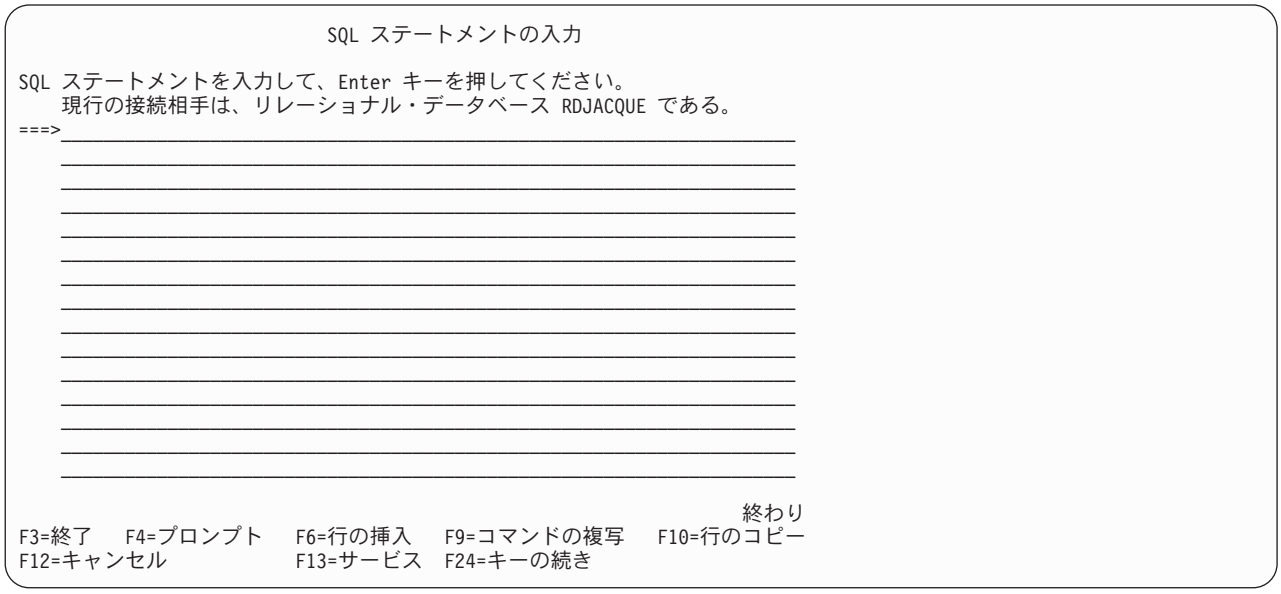
対話式 SQL の開始

対話式 SQL の使用を開始するには、i5/OS コマンド行で STRSQL とタイプ入力します。

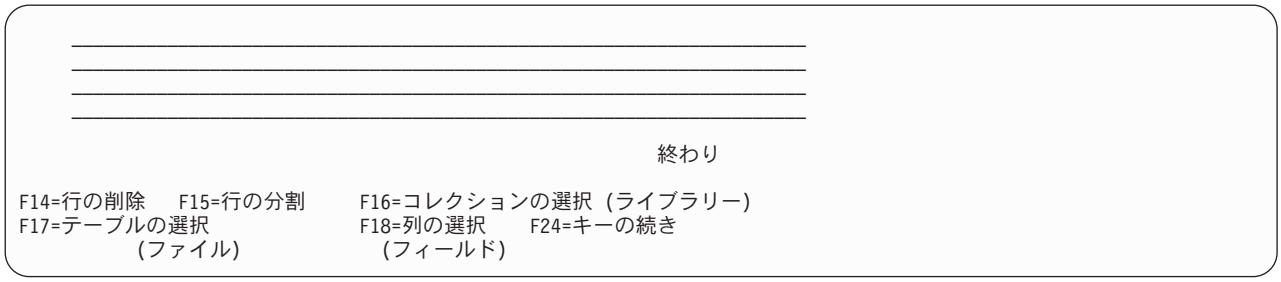
これにより、「SQL ステートメントの入力」画面が表示されます。これは、メインの「対話式 SQL」画面です。この画面から SQL ステートメントを入力して、次のキーを使用することができます。

- 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



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 。



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。

関連情報

SQL 対話式セッションの開始 (STRSQL)コマンド

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、Enter 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。

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 では大括弧で囲んだコメント (/* */) を入力できます。ただし、対話式 SQL では簡易コメント (つまり、-- で開始するコメント) を使用すべきではありません。これらのコメントはコメントの中に SQL ステートメントの残りを含むためです。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。

前ページに戻って、前のステートメント、コマンド、およびメッセージを表示することができます。 カーソルをステートメント入力行の上に置いて F9 (検索) キーを押すと、直前のステートメントが入力域にコピーされます。 F9 をもう一度押すと、もう 1 つ前のステートメントにスクロール・バックし、入力域にそれをコピーできます。F9 を押し続けることによって、目的のステートメントを見つけるまで前のステートメントにスクロール・バックすることができます。 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。

  ステートメントが解析され、完全な形の文節がプロンプト画面上に表示されます。

  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 表 . . . . . _____________________________________________
    SELECT 列 . . . . . _____________________________________________
  WHERE 条件  . . . . . _____________________________________________
  GROUP BY 列  . . . . . _____________________________________________
    HAVING 条件 . . . . . _____________________________________________
    ORDER BY 列 . . . . . _____________________________________________
    FOR UPDATE OF 列 . .  _____________________________________________

                                                                  Bottom
選択項目を入力して、Enter キーを押してください。

    結果テーブルの中の DISTINCT 行  . . . . . .  N    Y=Yes, N=No
    UNION と別の SELECT  . . . . . .          N    Y=Yes, N=No
    追加のオプションの指定 . . . . . .        N   Y=Yes, N=No



F3=取り消し          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=More keys 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=選択項目    F12=取り消し
```

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、その時点までに埋められたとおりに表示します。

プロンプト中に Enter 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埋められます。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。

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 つの列定義に関するすべての情報を入力できる画面が表示されます。

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 . .  _____________________________________
```

Enter 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。スキーマを選択する前に表のリスト要求を行うと、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。

- i5/OS 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、Enter 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 Enter 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. Enter 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 Enter 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. Enter 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、Enter 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付け

列を選択するためには、順序番号 (1-999) を入力して、Enter キーを押してください。

Seq  列                  テーブル              タイプ               長さ
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. Enter 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、Enter キーを押してください。
===> SELECT SOCSEC, NAME
     FROM YOURCOLL2.PEOPLE
```

14. Enter 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
- SQL 規則
- CONNECT パスワード・オプション

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・ヒストリーからすべての 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ます。終了するにはいくつかのオプションがあります。

- 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。
- 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。
- 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。
- 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 2 つのいずれかの実行を選択できます。

- 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 によるリモート・データベースへのアクセス

対話式 SQL では、SQL の CONNECT ステートメントを使用してリモート・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リモート・データベースとの接続に影響する場合があります。リモート・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 49. 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	*ISO
	*DMY	*EUR
	*MDY	*USA
	*JUL	*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

表 49. セッション属性の値 (続き)

セッション属性	元の値	新しい値
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働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、デフォルト活動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 でリモート・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リモート・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、ただしリモート・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リモート・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。

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。

関連資料

295 ページの『接続タイプの決定』
リモート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。

分散データベース・プログラミング

SQL ステートメント処理プログラムの使用

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は、SQL ステートメント実行 (RUNSQLSTM) コマンドを使用することによって、使用可能になります。

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 SEQUENCE
- ALTER TABLE
- CALL
- COMMENT ON
- COMMIT
- CREATE ALIAS
- CREATE DISTINCT TYPE
- CREATE FUNCTION
- CREATE INDEX
- CREATE PROCEDURE
- CREATE SCHEMA
- CREATE SEQUENCE
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- DECLARE GLOBAL TEMPORARY TABLE
- DELETE
- DROP
- GRANT
- INSERT
- LABEL ON
- LOCK TABLE
- REFRESH TABLE
- RELEASE SAVEPOINT
- RENAME
- REVOKE
- ROLLBACK
- SAVEPOINT

| • SET CURRENT DEGREE
| • SET ENCRYPTION PASSWORD
• 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。

関連情報

SQL ステートメント実行 (RUNSQLSTM) コマンド

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ることを意味します。

| ソース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、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。ドロップするオブジェクトが検出さ
| れない場合、DROP ステートメントは重大度レベル 20 のエラーを発行します。 ERRLVL パラメーター
| の値を 20 に設定することにより、DROP ステートメント以外の重大度が高いエラーの処理を継続させず
| に、DROP ステートメントから発行されるこれらのエラーのみを無視できるようになります。

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ソース・メンバー・リスト

以下に示す例は、SQL ステートメント処理プログラムのソース・メンバー・リストを詳述します。

注: コード例を使用する場合は、 325 ページの『コードに関する特記事項』のご使用条件に同意する必要があります。

```
5722SS1 V5R4M0 060210        SQL ステートメントの実行       SCHEMA       02/10/06 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
デフォルトのコレクション..*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 ID..............*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
SQL 規則.............*DB2
10 進法結果オプション:
    最大精度.........31
    最大位取り.......31
    最小分割の位取り.....0
04/01/98  11:54:10 にソース・メンバーが変更された。
```

図 1. SQL ステートメント処理プログラムの QSYSPRT リスト

```
5722SS1 V5R4M0 060210        SQL ステートメントの実行       SCHEMA       02/10/06 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                           -- コレクション DEPT 内に EMP が作成される
    8          CREATE INDEX EMPIND ON EMP(EMPNBR)
    9                           -- DEPT 内に EMPIND が作成される
   10          GRANT SELECT ON EMP TO PUBLIC;  -- 権限の認可
   11
   12     INSERT INTO DEPT/EMP VALUES('JOHN SMITH', 1234);
   13                             /* スキーマ内にないので表は
   14                                修飾する必要がある */
   15
   16     CREATE SCHEMA AUTHORIZATION MANAGER
   17                              -- このスキーマは MANAGER のユーザー・プロファイル
   18                              -- を使用する
   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                    -- 2 つのステートメントを同一行に記述できる
* * * * *  ソ ー ス の 終 わ り  * * * * *
```

```
5722SS1 V5R4M0 060210        SQL ステートメントの実行       SCHEMA       02/10/06 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 スキーマ 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 の索引 EMPIND が DEPT に
                     作成された。
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 スキーマ 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
                     の索引 SALARYIND が MANAGER に作成された。
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  * * * * *
```

分散リレーショナル・データベース機能と SQL

分散リレーショナル・データベース は、相互に接続されたコンピュータ・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) またはタイプが異なる場合 (DB2 Universal Database (z/OS® 版)、DB2 (VSE 版および VM 版)、DB2 Universal Database (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リモート・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。

- リモート作業単位 (RUW)

  リモート作業単位は、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。

関連概念

3 ページの『DB2 UDB for iSeries 構造化照会言語の紹介』
このトピックでは、DB2 UDB for iSeries および DB2 UDB Query Manager and SQL Development Kit のライセンス・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。

13 ページの『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 ステートメントがリモート・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。

関連資料

265 ページの『分散リレーショナル・データベース (DRDA) を使用したデータへのアクセス』
分散リレーショナル・データベース は、相互に接続されたコンピュータ・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。 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。

分散データベース・プログラミング

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、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

追加のサポートは、SQL プリコンパイラー・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

関連資料

325 ページの『DB2 UDB for iSeries CL コマンドの記述』
DB2 UDB for iSeries は以下の SQL のための以下の CL コマンドを実行します。

関連情報

SQL ステートメントを含むプログラムの準備と実行

CONNECT ステートメント

DISCONNECT ステートメント

DROP ステートメント

GRANT (パッケージ) ステートメント

REVOKE (パッケージ) ステートメント

RELEASE ステートメント

SET CONNECTION ステートメント

DB2 UDB for iSeries 分散リレーショナル・データベース・プログラム例

リモート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。

- ローカルおよびリモート・ロケーションで QSQSAMP ライブラリーを作成する。
- ローカルおよびリモート・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。
- プログラムのプリコンパイル、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。
- ローカルおよびリモート・ロケーションにある表にデータをロードする。
- プログラムのプリ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 パッケージをリモート・ロケーションで作成する。
- プログラムのプリコンパイル、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
```

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
```

SQL パッケージ・サポート

オペレーティング・システムは、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。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、パッケージがプリ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プリコンパイル・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、リモート・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。

関連情報

SQL パッケージ作成 (CRTSQLPKG) コマンド

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、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、そのリモート・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。プリコンパイラーは、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リモート・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、次の例外を除いて、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 i5/OS である分散プログラム内でしか使用できません。非 iSeries アプリケーション・リクエスターは、V5R3 の iSeries アプリケーション・サーバーで、読み取り専用の融通の利かないスクロール可能カーソル操作を出すことができます。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の使用についての追加の制約事項として、これらの機能を使用するときには、BLOB、CLOB、および DBCLOB データの送信が許可されません。

関連情報

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

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に考慮する必要のある事項が多数あります。

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 は CRTSQLPKG コマンドに対する *USE 権限を持つ必要があります。

DB2 UDB for iSeries 以外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プリコンパイラー・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) パラメーター:

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、プリ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 プログラムのプリコンパイル時に、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、常にデフォルト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。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・エラーが起こる場合があります。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、あるいはリモート・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、リモート・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:

すでにプリ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 を開始すると、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。リモート・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、ビュー、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 で稼働し、システム B が CCSID 500 で稼働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。
- ¬ のコードポイントは、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。これらのコードポイントは使用する 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

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。

DRDA で TCP/IP を使用する以前は、「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定されました。

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、APPC の会話に限定した説明でない限り、このトピックでは、「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ています。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しやすいようにされています。

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

以下に PGM1 のソース・コードを示します。

```
     ....
     EXEC SQL
       CONNECT TO SYSB
     END-EXEC.
     EXEC SQL
       SELECT ....
     END-EXEC.
     CALL PGM2.
     ....
```

図 2. 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

以下に PGM2 のソース・コードを示します。

```
...
        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 3. 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

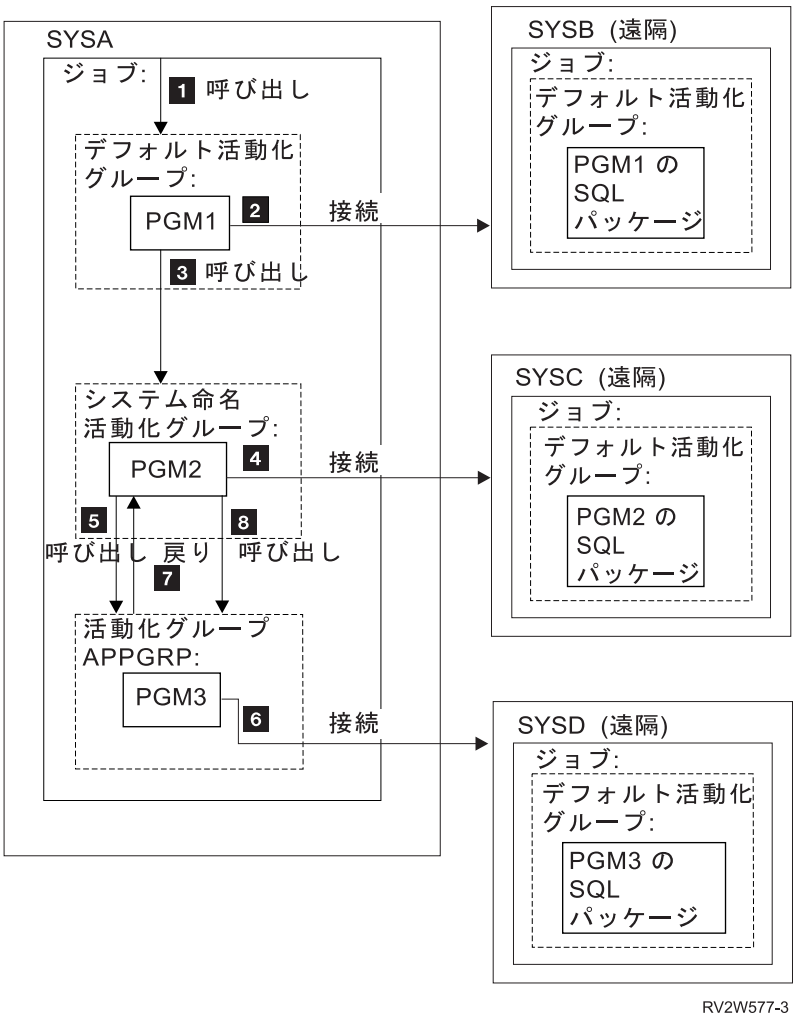
以下に PGM3 のソース・コードを示します。

```
    ...
     EXEC SQL
       INSERT INTO TAB VALUES(:st1);
     EXEC SQL COMMIT;
    ....
```

図 4. 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、デフォルト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 パラメーターのデフォルト値ではないので、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、デフォルト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、デフォルト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、

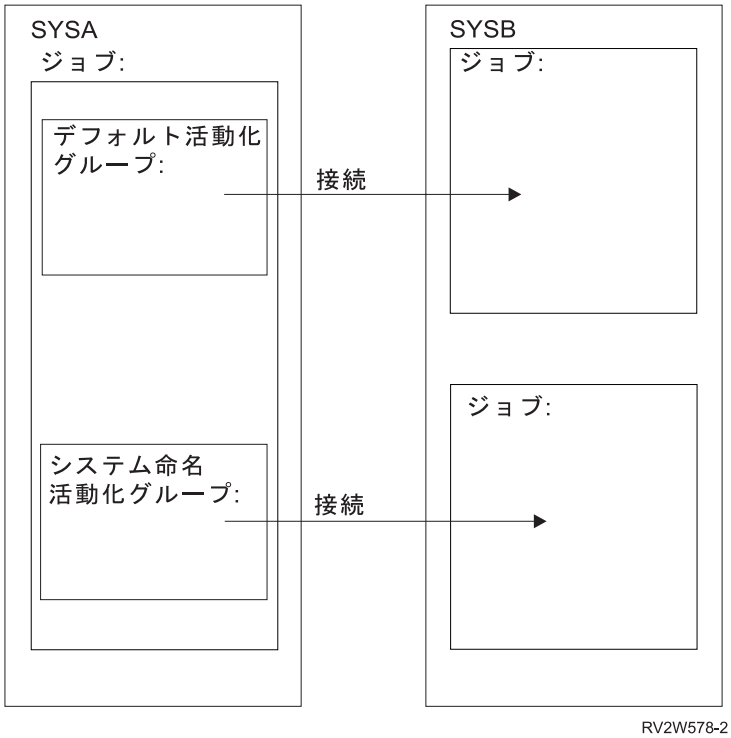
両方の接続が活動状態になります。PGM2 はカーソルをオープンし、行を取り出して更新します。PGM2 はコミットメント制御の下で実行中で、作業単位内にあり、接続可能状態ではありません。

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。



デフォルト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・リクエスターがデフォルト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 がデフォルト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 はデフォルト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) を指定してプリ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) を指定してプリ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デフォルト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。

関連資料

300 ページの『接続の終了』
リモート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。

非デフォルト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。

関連資料

300 ページの『接続の終了』
リモート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。

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。

- リモート作業単位 (RUW)

  リモート作業単位は、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。リモート作業単位は、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、接続は保護リソースです。それ以外の場合、接続は非保護リソースです。

  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。 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

リモート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、コミットメン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)、および SQL 診断域の DB2_CONNECTION_TYPE が接続のタイプを示します。

DB2_CONNECTION_TYPE の特定の値は、次のとおりです。

1. 接続先はローカル・リレーショナル・データベースであり、接続は保護されます。
2. 接続先はリモート・リレーショナル・データベースであり、接続は保護されません。
3. 接続先はリモート・リレーショナル・データベースであり、接続は保護されます。
4. 接続先はアプリケーション・リクエスター・ドライバー・プログラムであり、接続は保護されます。

SQLERRD(4) の特定の値は次のとおりです。

1. 接続先はリモート・リレーショナル・データベースであり、接続は保護されません。コミット可能な更新は、この接続で実行できます。これは、次のいずれかがあてはまる場合に起こります。
   - 接続が、リモート作業単位 (RUW) を使用して確立した場合。
   - 分散作業単位 (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、OS/400 V3R1 以前の環境下での 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更新可能なオープン・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、リモート接続の場合は 1 フェーズの更新可能な DRDA リソース、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 リソースをそれぞれ登録します。

2. 接続先はリモート・リレーショナル・データベースであり、接続は保護されません。この接続は読み取り専用です。これは、次のこと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で実行している場合、SQL は 1 フェーズの DRDA 読み取り専用リソースを登録します。
3. 接続先はリモート・リレーショナル・データベースであり、接続は保護されます。コミット可能な更新を実行できるかどうかは不明です。 これは、次のすべて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 DRDA 未決定リソースを登録します。
4. 接続先はリモート・リレーショナル・データベースであり、接続は保護されません。コミット可能な更新を実行できるかどうかは不明です。 これは、次のすべてがあてはまる場合にのみ起こります。
   - 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 1 フェーズの DRDA 未決定リソースを登録します。
5. 接続先はローカル・データベースまたはアプリケーション・リクエスター・ドライバー (ARD) プログラムであり、接続は保護されます。コミット可能な更新を実行できる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 未決定リソースを登録します。

次の表は、リモート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 50. 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・リソース	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 [1]
はい	はい	いいえ	はい	N/A [1]
はい	はい	はい	いいえ	3
はい	はい	はい	はい	3

[1]DRDA は、リモート作業単位 (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。

関連資料

276 ページの『対話式 SQL によるリモート・データベースへのアクセス』
対話式 SQL では、SQL の CONNECT ステートメントを使用してリモート・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。

関連情報

コミットメント制御

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リソースを登録したか、または 1 フェーズの更新可能なリソースを登録した場合、別の 1 フェーズの更新可能なリソースは登録できません。

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。
- 接続がリモートの場合は、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。SQLCA または SQL 診断域の DB2_CONNECTION_STATUS の SQLERRD(3) に 1 か 2 の値が戻されます。

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。
   - 接続が、リモート作業単位 (RUW) を使用して確立した場合。
   - 分散作業単位 (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。
     – 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。.
       - 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。
   - 分散作業単位 (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。
     – 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。
   - 分散作業単位 (DUW) を使用して接続を確立し、次のいずれか 1 つがあてはまる場合。
     – リモート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。
   - 分散作業単位 (DUW) を使用して接続を確立し、次のいずれか 1 つがあてはまる場合。
     – リモート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、接続タイプの値に基づく接続状況の決定方法、リモート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 51. 接続状況の値の決定に関する要約

接続方法	更新可能なリモート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 [1]	SQLERRD(3) または DB2_CONNECTION_STATUS
RUW			1
DUW	はい		2
DUW	いいえ	更新なし	1
DUW	いいえ	1 フェーズ	2
DUW	いいえ	この接続	1
DUW	いいえ	2 フェーズ	1

[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。

読み取り専用の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。

分散作業単位接続に関する考慮事項

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。

- 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。V5R3 では、データベース照会中にトリガーまたはユーザー定義関数をアクティブにしたために行われた更新は、DRDA 2 フェーズ・コミット操作中に考慮されます。

接続の終了

リモート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。

リモート 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リモート・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。

関連資料

292 ページの『デフォルト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。

293 ページの『非デフォルト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、プログラムをデフォルト値である RDBCNNMTH(*DUW) を使用してプリ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;

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;
       }

     /* 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:

 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 5. 分散作業単位プログラムの例

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 つのリモート・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を切断状態にする代わりに、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 つのリモート・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、両方のリモート・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 updatable.
         EXEC SQL CONNECT END-EXEC.
    * If connection is updatable, 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 6. 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されたプログラムは、呼び出し側プログラムがオープンまたは準備した、カーソルまたは準備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 7. 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リモート・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。

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、i5/OS 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、i5/OS 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、*「分散関係データベース問題判別の手引き」* N:SC26-4782 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。

関連情報

SQL メッセージおよびコード

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。

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。 V5R3 より前のバージョンでは、ストアード・プロシージャー内でオープンされた照会の 1 つのインスタンスのみを一度にオープンすることができました。今では、同じストアード・プロシージャーへの複数の呼び出しを結果セットのカーソルをクローズせずに行うことができるので、照会の複数のインスタンスを同時にオープンすることができます。

関連概念

130 ページの『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。

関連資料

分散データベース・プログラミング

関連情報

SET RESULT SETS ステートメント

CREATE PROCEDURE (SQL)

CREATE PROCEDURE (外部)

参照情報

SQL プログラミングの参照情報にはサンプル表と CL コマンドが含まれます。

DB2 UDB for iSeries サンプル表

このトピックには、このトピック、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。

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、プロシージャーを呼び出すには、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、大文字で指定してください。スキーマは、すでに存在するものであってはなりません。

注: 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。

関連概念

1 ページの『SQL プログラミング』
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。

関連資料

18 ページの『参照保全および表』
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。

98 ページの『例: DELETE カスケード規則』

243 ページの『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。

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 52. 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。

DEPARTMENT:

表 DEPARTMENT にあるデータの完全なリスト。

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	?

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
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

列名	説明
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

EMPLOYEE:

表 EMPLOYEE にあるデータの完全なリスト。

EMP NO	FIRST NAME	MID INIT	LASTNAME	WORK DEPT	PHONE NO	HIRE DATE	JOB	ED LEVEL	SEX	BIRTH DATE	SALARY	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:

表 EMP_PHOTO にあるデータの完全なリスト。

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	?	

社員履歴表 (EMP_RESUME)

社員の写真表には、社員の経歴が含まれ、社員番号別に保管されています。

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:

表 EMP_RESUME にあるデータの完全なリスト。

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 53. 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:

表 EMPPROJACT にあるデータの完全なリスト。

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

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
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
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

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
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
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
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:

表 PROJECT にあるデータの完全なリスト。

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	?

PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
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
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:

表 PROJACT にあるデータの完全なリスト。

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	?
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	?

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
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	?
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:

表 ACT にあるデータの完全なリスト。

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
ENDING	クラス終了時刻

CL_SCHED:

表 CL_SCHED にあるデータの完全なリスト。

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、メッセージ受信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:

表 IN_TRAY にあるデータの完全なリスト。

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

列名	説明
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:

表 ORG にあるデータの完全なリスト。

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、従業員の背景情報を記述します。

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	社員の歩合

STAFF:

表 STAFF にあるデータの完全なリスト。

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	?
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、各販売員の販売状況に関する情報を記述します。

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:

表 SALES にあるデータの完全なリスト。

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
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

SALES_DATE	SALES_PERSON	REGION	SALES
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

DB2 UDB for iSeries CL コマンドの記述

DB2 UDB for iSeries は以下の SQL のための以下の CL コマンドを実行します。

- CRTSQLPKG (SQL パッケージ作成) コマンド
- DLTSQLPKG (SQL パッケージ削除) コマンド
- PRTSQLINF (SQL 情報印刷) コマンド
- RUNSQLSTM (SQL ステートメントの実行) コマンド
- STRSQL (SQL の開始) コマンド

関連資料

282 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 Query Manager and SQL Development Kit ライセンス・プログラムは、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。

コードに関する特記事項

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。

強行法規で除外を禁止されている場合を除き、IBM、そのプログラム開発者、および供給者は「プログラム」および「プログラム」に対する技術的サポートがある場合にはその技術的サポートについて、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を含むすべての明示もしくは黙示の保証責任を負わないものとします。

IBM、そのプログラム開発者、または供給者は、いかなる場合においてもその予見の有無を問わず、以下に対する責任を負いません。

1. データの喪失、または損傷。

2. 直接損害、特別損害、付随的損害、間接損害、または経済上の結果的損害
3. 逸失した利益、ビジネス上の収益、あるいは節約すべかりし費用

国または地域によっては、法律の強行規定により、上記の責任の制限が適用されない場合があります。

付録. 特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものです。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒106-0032
東京都港区六本木 3-2-31
IBM World Trade Asia Corporation
Licensing

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、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を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、IBM 機械コードのご使用条件、またはそれと同等の条項に基づいて、 IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

表示されている IBM の価格は IBM が小売り価格として提示しているもので、現行価格であり、通知なしに変更されるものです。卸価格は、異なる場合があります。

本書はプランニング目的としてのみ記述されています。記述内容は製品が使用可能になる前に変更になる場合があります。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。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。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

© (お客様の会社名) (西暦年). このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。 © Copyright IBM Corp. _年を入れる_. All rights reserved.

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は表示されない場合があります。

プログラミング・インターフェース情報

本書「SQL programming」には、プログラムを作成するユーザーが IBM i5/OS のサービスを使用するためのプログラミング・インターフェースが記述されています。

商標

以下は、IBM Corporation の商標です。

| AIX
| DB2
| DB2 Universal Database
| Distributed Relational Database Architecture
| Domino
| DRDA
| e(ロゴ)server
| i5/OS
| IBM
| IBM (ロゴ)
| Integrated Language Environment
| iSeries
| Lotus Notes
| Operating System/400
| OS/390
| OS/400
| PowerPC
| System/36

Windows は、Microsoft Corporation の米国およびその他の国における商標です。

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標または登録商標です。

| Linux は、Linus Torvalds の米国およびその他の国における商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

使用条件

これらの資料は、以下の条件に同意していただける場合に限りご使用いただけます。

個人使用: これらの資料は、すべての著作権表示その他の所有権表示をしていただくことを条件に、非商業的な個人による使用目的に限り複製することができます。ただし、IBM の明示的な承諾をえずに、これらの資料またはその一部について、二次的著作物を作成したり、配布 (頒布、送信を含む) または表示 (上映を含む) することはできません。

商業的使用: これらの資料は、すべての著作権表示その他の所有権表示をしていただくことを条件に、お客様の企業内に限り、複製、配布、および表示することができます。ただし、 IBM の明示的な承諾をえずにこれらの資料の二次的著作物を作成したり、お客様の企業外で資料またはその一部を複製、配布、または表示することはできません。

ここで明示的に許可されているもの以外に、資料や資料内に含まれる情報、データ、ソフトウェア、またはその他の知的所有権に対するいかなる許可、ライセンス、または権利を明示的にも黙示的にも付与するものではありません。

資料の使用が IBM の利益を損なうと判断された場合や、上記の条件が適切に守られていないと判断された場合、IBM はいつでも自らの判断により、ここで与えた許可を撤回できるものとさせていただきます。

お客様がこの情報をダウンロード、輸出、または再輸出する際には、米国のすべての輸出入関連法規を含む、すべての関連法規を遵守するものとします。

IBM は、これらの資料の内容についていかなる保証もしません。これらの資料は、特定物として現存するままの状態で提供され、第三者の権利の不侵害の保証、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を含むすべての明示もしくは黙示の保証責任なしで提供されます。

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