

IBM i
バージョン 7.2

**データベース
SQL プログラミング**

IBM

IBM i
バージョン 7.2

**データベース
SQL プログラミング**

IBM

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、401 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。

本製品およびオプションに付属の電源コードは、他の電気機器で使用しないでください。

本書にはライセンス内部コードについての参照が含まれている場合があります。ライセンス内部コードは機械コードであり、IBM 機械コードのご使用条件に基づいて使用権を許諾するものです。

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 i
Version 7.2
Database
SQL programming

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

第1刷 2014.4

© Copyright IBM Corporation 1998, 2013.

目次

SQL プログラミング	1	ROWID の使用	28
IBM i 7.2 の新機能	1	順序の作成と使用	28
SQL プログラミングの PDF ファイル	4	識別列と順序の比較	30
DB2 for i 構造化照会言語の紹介	4	フィールド・プロシーチャーの定義	31
SQL の概念	4	フィールド・プロシーチャーのフィールド定義	31
SQL リレーショナル・データベースとシステム		フィールド・プロシーチャーの指定	32
用語	6	フィールド・プロシーチャーが呼び出されると	
SQL およびシステム命名規則	7	とき	32
SQL ステートメントのタイプ	7	フィールド・プロシーチャーを実行するための	
SQL 通信域	9	パラメーター・リスト	33
SQL 診断域	10	フィールド・プロシーチャー・パラメータ	
SQL オブジェクト	10	一値リスト (FPPVL)	35
スキーマ	10	フィールド・プロシーチャーのパラメータ	
ジャーナルおよびジャーナル・レシーバー	10	一値記述子	36
カタログ	10	フィールド定義 (機能コード 8)	37
表、行、および列	11	フィールド・エンコード (機能コード 0)	38
別名	11	フィールド・デコード (機能コード 4)	39
ビュー	11	フィールド・プロシーチャー・プログラム	
索引	12	の例	40
制約	12	フィールド・プロシーチャーを作成するときの	
トリガー	13	一般ガイドライン	41
ストアード・プロシーチャー	13	索引に関する考慮事項	42
順序	13	スレッドについての考慮事項	42
グローバル変数	13	データをマスクするフィールド・プロシー	
ユーザー定義関数	13	チャーを作成するためのガイドライン	43
ユーザー定義タイプ	14	データをマスクするフィールド・プロシー	
XSR オブジェクト	14	チャー・プログラムの例	45
SQL パッケージ	14	LABEL ON ステートメントを使用した記述ラ	
アプリケーション・プログラム・オブジェクト		ベルの作成	47
ユーザー・ソース・ファイル	16	COMMENT ON を使用した SQL オブジェクトの	
出力ソース・ファイル・メンバー	16	記述	48
プログラム	17	表定義の変更	49
SQL パッケージ	17	列の追加	49
モジュール	17	列の変更	49
サービス・プログラム	18	可能なデータ・タイプの変換	50
データ定義言語	18	列の削除	51
スキーマの作成	18	ALTER TABLE ステートメントの操作の順序	51
表の作成	19	ALIAS 名の作成と使用	52
制約の追加および除去	19	ビューの作成と使用	52
参照保全および表	20	ビューに関する WITH CHECK OPTION	54
参照制約の追加および除去	20	WITH CASCADED CHECK OPTION	55
例: 参照制約の追加	21	WITH LOCAL CHECK OPTION	55
例: 制約の除去	22	例: カスケード検査オプション	56
検査保留	22	索引の作成	57
LIKE を使用した表の作成	23	グローバル変数の作成と使用	58
AS を使用した表の作成	23	既存オブジェクトの置換	58
マテリアライズ照会表の作成および変更	24	データベース設計でのカタログ	59
グローバル一時表の宣言	25	表に関するカタログ情報の入手	59
リモート・サーバー・データを使用した表の作成	26	列に関するカタログ情報の入手	59
行変更タイム・スタンプ列の作成	26	データベース・オブジェクトのドロップ	60
識別列の作成および変更	27	データ操作言語	60

SELECT ステートメントを使用したデータの検索	60	DELETE ステートメントを使用した表からの行の除去	123
基本 SELECT ステートメント	61	参照制約付き表からの行の除去	124
WHERE 文節を使用する検索条件の指定	62	例: DELETE 規則	126
WHERE 文節での式	63	TRUNCATE ステートメントを使用した表からの	
比較演算子	64	行の除去	127
NOT キーワード	65	データのマージ	128
GROUP BY 文節	65	副照会の使用	129
HAVING 文節	67	SELECT ステートメントの副照会	129
ORDER BY 文節	68	副照会と検索条件	130
静的 SELECT ステートメント	70	副照会の使用上の注意	131
ヌル値の処理	70	WHERE または HAVING 文節に副照会を	
SQL ステートメント内の特殊レジスター	72	組み込む	131
データ・タイプのキャスト	73	相関副照会	133
日付、時刻、および時刻スタンプのデータ・タイプ	74	相関名と相関参照	133
現在日付値および現在時刻値の指定	74	例: WHERE 文節の相関副照会	133
日付/時刻演算	74	例: HAVING 文節の相関副照会	135
行変更式	75	例: 選択リストの相関副照会	135
重複行の処理	75	例: UPDATE ステートメントの相関副照会	136
複雑な検索条件の定義	76	例: DELETE ステートメントの相関副照会	137
LIKE に関する特殊な考慮事項	77	SQL での分類順序および正規化	137
WHERE 文節内の複数の検索条件	78	ORDER BY および行選択で使用される分類順序	138
OLAP 指定の使用	79	分類順序と ORDER BY	139
複数の表からのデータの結合	82	分類順序と行選択	140
内部結合	82	分類順序と視点	141
左方外部結合	84	分類順序と CREATE INDEX ステートメント	141
右方外部結合	84	分類順序と制約	141
例外結合	85	ICU 分類順序	142
クロス結合	85	正規化	142
全外部結合	86	データ保護	143
1 つのステートメントでの複数の結合タイプ	87	SQL オブジェクトの機密保護	143
表式の使用	88	権限 ID	145
再帰的照会の使用	89	ビュー	145
副選択結合時の UNION キーワードの使用	102	列マスクおよび行許可	145
UNION ALL キーワードの指定	105	監査	145
EXCEPT キーワードの使用	106	データ保全性	146
INTERSECT キーワードの使用	108	並行性	146
データ取り出しエラー	110	ジャーナル処理	148
INSERT ステートメントを使用した行の挿入	112	コミットメント制御	149
VALUES 文節を使用した行の挿入	114	保管ポイント	153
選択ステートメントによる行の挿入	114	アトミック・オペレーション	154
ブロック化 INSERT ステートメントを使用した		制約	157
複数の行の挿入	115	検査制約の追加および使用	157
参照制約付き表へのデータの挿入	115	保管および復元機能	158
識別列への値の挿入	116	耐損傷性	159
挿入値の選択	117	索引回復	159
リモート・データベースからのデータの挿入	118	カタログの保全性	160
UPDATE ステートメントを使用した表内のデータの変更	118	ユーザー補助記憶域プール	160
スカラー副選択を使用する表の更新	120	独立補助記憶域プール	161
別の表からの行を使用する表の更新	120	ルーチン	161
参照制約付きの表の更新	120	ストアード・プロシージャ	161
例: UPDATE 規則	121	外部プロシージャの定義	162
識別列の更新	122	SQL プロシージャの定義	163
表のデータの検索と更新	122	デフォルト・パラメーターを使用したプロシ	
		ージャの定義	168
		ストアード・プロシージャの呼び出し	170

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	171	修飾なしの関数参照の使用	232
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	171	名前付き引数を使用した UDF の呼び出し	233
SQLDA を伴う組み込み CALL ステートメントの使用	172	関数参照の要約	233
CREATE PROCEDURE が存在しない動的 CALL ステートメントの使用	173	トリガー	235
例: CALL ステートメント	174	SQL トリガー	236
ストアド・プロシージャーから戻される結果セット	180	BEFORE SQL トリガー	237
例 1: 単一の結果セットを戻すストアド・プロシージャーの呼び出し	181	AFTER SQL トリガー	238
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ド・プロシージャーの呼び出し	182	複数イベント SQL トリガー	239
ストアド・プロシージャーから結果セットを受け取るプログラムまたは SQL プロシージャーの作成	187	INSTEAD OF SQL トリガー	240
ストアド・プロシージャーおよびユーザー定義関数用のパラメーターの引き渡し規則	193	SQL トリガーのハンドラー	242
標識変数とストアド・プロシージャー	198	SQL トリガーの変換表	243
呼び出し側プログラムへの完了状況の戻り	200	外部トリガー	244
DB2 から外部プロシージャーへのパラメーターの引き渡し	201	SQL のプロシージャーおよび関数での配列サポート	244
パラメーター・スタイル SQL	201	SQL ルーチンのデバッグ	246
パラメーター・スタイル GENERAL	203	SQL ルーチンまたは SQL トリガーの難読化	247
パラメーター・スタイル GENERAL WITH NULLS	203	SQL および外部のルーチン・オブジェクトの管理	249
パラメーター・スタイル DB2GENERAL	204	プロシージャーおよび関数のパフォーマンスの向上	250
パラメーター・スタイル Java	204	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	251
動的複合ステートメント	204	パフォーマンスのためのルーチンの再設計	253
ユーザー定義関数の使用	205	特別なデータ・タイプの処理	254
UDF の概念	206	ラージ・オブジェクト	254
SQL 関数としての UDF の作成	208	ラージ・オブジェクト・データ・タイプ	254
例: SQL スカラー UDF	208	ラージ・オブジェクト・ロケーター	255
例: SQL 表 UDF	208	例: CLOB 値を処理するためのロケーターの使用	256
外部関数としての UDF の作成	209	例: C の LOBLOC	256
UDF の登録	209	例: COBOL の LOBLOC	257
DB2 から外部関数へ引数を渡す	212	標識変数および LOB ロケーター	259
表関数の考慮事項	219	LOB ファイル参照変数	259
UDF のエラー処理	219	例: ファイルへの CLOB データの抽出	261
スレッドについての考慮事項	220	例: C の LOBFILE	261
並列処理	220	例: COBOL の LOBFILE	262
隔離または隔離解除の考慮事項	221	例: CLOB 列へのデータの挿入	263
保管と復元の考慮事項	221	LOB 列のレイアウトの表示	263
デフォルト・パラメーターを使用した UDF の定義	221	LOB 列のジャーナル・エントリーのレイアウト	264
例: UDF コード	222	ユーザー定義特殊タイプ	264
例: 数の平方を求める UDF	222	UDT の定義	265
例: カウンター	224	例: 通貨	265
例: 天気の情報関数	225	例: 履歴	266
SQL ステートメント内での UDF の使用	231	UDT を使用した表の定義	266
関数の引数としてのパラメーター・マーカ	231	例: 売上高	266
ーまたは NULL 値の使用	231	例: 応募用紙	266
修飾された関数参照の使用	231	UDT の操作	267
		例: UDT の使用	267
		例: UDT と定数との比較	267
		例: UDT 間のキャスト	267
		例: UDT が関係する比較	269
		例: UDT が関係する、ソース化された UDF	269
		例: UDT が関係する割り当て	269
		例: 動的 SQL での割り当て	270

例: さまざまな UDT が関係する割り当て	270	ステートメント処理モード	313
例: UNION での UDT の使用	271	副照会	313
例: UDT、UDF、および LOB の使用	272	CREATE TABLE プロンプト	313
例: UDT と UDF の定義	272	DBCS データの入力	313
例: データベースにデータを入れるための		リスト選択機能の使用	314
LOB 関数の使用	273	例: リスト選択機能の使用	315
例: UDT のインスタンスを照会するための		セッション・サービスの説明	317
UDF の使用	273	対話式 SQL の終了	318
例: UDT インスタンスを操作するための		既存の SQL セッションの使用	318
LOB ロケーターの使用	274	SQL セッションの回復	319
データ・リンクの使用	274	対話式 SQL によるリモート・データベース	
データ・リンクのリンク制御レベル	275	へのアクセス	319
NO LINK CONTROL	275	SQL ステートメント処理プログラムの使用	321
FS 許可を使用した FILE LINK CONTROL	275	エラーが発生した後のステートメントの実行	323
DB 許可を使用した FILE LINK		SQL ステートメント処理プログラムでのコミ	
CONTROL	276	ットメント制御	323
データ・リンクの処理	276	SQL ステートメント処理プログラムのソー	
異なる環境で SQL を使用する	278	ス・リスト	323
カーソルの使用	279	1 RUNSQL CL コマンドの使用	325
カーソルのタイプ	279	分散リレーショナル・データベース関数と SQL	327
例: カーソルの使用	280	DB2 for i 分散リレーショナル・データベース・	
ステップ 1: カーソルを定義する	282	サポート	328
ステップ 2: カーソルをオープンする	283	DB2 for i 分散リレーショナル・データベース・	
ステップ 3: データの終わりに達したとき		プログラム例	329
の処置を指定する	283	SQL パッケージ・サポート	330
ステップ 4: カーソルを用いて行を取り出		SQL パッケージの中の有効な SQL ステート	
す	284	メント	330
ステップ 5a: 現在行を更新する	285	SQL パッケージ作成時の考慮事項	331
ステップ 5b: 現在行を削除する	285	CRTSQLPKG 権限	331
ステップ 6: カーソルをクローズする	286	DB2 for i 以外のデータベースでのパッケ	
複数行用 FETCH ステートメントの使用	286	ージの作成	331
ホスト構造配列を使用した複数行 FETCH	287	ターゲット・リリース (TGTRLS) パラメ	
行記憶域を使用した複数行 FETCH	288	ーター	332
作業単位とオープン・カーソル	291	SQL ステートメントのサイズ	332
動的 SQL アプリケーション	291	パッケージを必要としないステートメント	332
動的 SQL ステートメントの実行	292	オブジェクト・タイプのパッケージ	333
動的 SQL ステートメントの CCSID	292	ILE プログラムおよび ILE サービス・プ	
非 SELECT ステートメントの処理	293	ログラム	333
PREPARE ステートメントおよび		パッケージ作成の接続	333
EXECUTE ステートメントの使用	293	作業単位	333
SELECT ステートメントの処理と記述子の使		ローカルにパッケージを作成	333
用	294	ラベル	334
固定リスト SELECT ステートメント	294	整合性トークン	334
可変リスト SELECT ステートメント	295	SQL および再帰	334
SQL 記述域	296	SQL 用の CCSID に関する考慮事項	334
SQLDA の形式	297	接続管理および活動化グループ	335
SQLDA の記憶域を割り振るための		PGM1 のソース・コード	335
SELECT ステートメントの例	300	PGM2 のソース・コード	336
例: 割り振られた SQL 記述子を使用した		PGM3 のソース・コード	336
SELECT ステートメント	304	同じリレーショナル・データベースへの多重	
パラメーター・マーカ	306	接続	338
対話式 SQL の使用	308	デフォルト活動化グループの場合の暗黙の接	
対話式 SQL の開始	309	続管理	338
ステートメント入力機能の使用	311	非デフォルト活動化グループの場合の暗黙の	
プロンプト	311	接続管理	339
構文検査	313	分散サポート	340

接続タイプの決定	340	EMPPROJECT	375
接続およびコミットメント制御に関する制約 事項	343	プロジェクト表 (PROJECT).	377
接続状況の決定.	343	PROJECT.	378
分散作業単位接続に関する考慮事項.	345	プロジェクト活動表 (PROJACT)	378
接続の終了	345	PROJACT.	379
分散作業単位	346	活動表 (ACT)	381
分散作業単位接続の管理.	346	ACT	381
接続状況の検査.	349	クラス・スケジュール表 (CL_SCHED)	382
カーソルおよび準備されたステートメント	349	CL_SCHED	382
DRDA ストアード・プロシージャーに関する考 慮事項.	350	未処理表 (IN_TRAY).	383
WebSphere MQ と DB2 の組み合わせ	351	IN_TRAY.	383
WebSphere MQ メッセージ.	351	組織表 (ORG)	384
WebSphere MQ メッセージ処理	351	ORG	384
DB2 MQ サービス	352	スタッフ表 (STAFF)	385
DB2 MQ ポリシー	353	STAFF.	385
DB2 MQ 関数	353	販売表 (SALES)	386
DB2 MQ の依存関係.	355	SALES.	387
DB2 MQ 表.	355	サンプルの XML 表	388
DB2 MQ の CCSID 変換	362	製品表 (PRODUCT)	388
WebSphere MQ トランザクション	363	PRODUCT	389
WebSphere MQ による基本的なメッセージング	364	購入オーダー表 (PURCHASEORDER)	390
WebSphere MQ によるメッセージの送信	366	PURCHASEORDER	391
WebSphere MQ によるメッセージの取り出し	366	顧客表 (CUSTOMER).	395
WebSphere MQ によるアプリケーション間の接 続	367	CUSTOMER.	395
参照	368	カタログ表 (CATALOG).	396
DB2 for i サンプル表	368	CATALOG	397
部門表 (DEPARTMENT).	368	提供業者表 (SUPPLIERS)	397
DEPARTMENT	369	SUPPLIERS	397
社員表 (EMPLOYEE).	370	在庫表 (INVENTORY)	397
EMPLOYEE	371	INVENTORY	398
社員の写真表 (EMP_PHOTO)	372	製品提供業者表 (PRODUCTSUPPLIER).	398
EMP_PHOTO	373	PRODUCTSUPPLIER	398
社員履歴表 (EMP_RESUME)	373	DB2 for i CL コマンドの記述.	398
EMP_RESUME	374		
社員プロジェクト活動表 (EMPPROJECT)	374		
		特記事項.	401
		プログラミング・インターフェース情報	403
		商標	403
		使用条件	403

SQL プログラミング

DB2® for IBM® i データベースは、広範囲の構造化照会言語 (SQL) のサポートを提供します。

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、以下の記述が当てはまることを前提としています。

- 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。
- 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 を使用する場合は 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、398 ページの『コードに関するライセンス情報および特記事項』のご使用条件に同意する必要があります。

関連概念:

組み込み SQL プログラミング

関連資料:

368 ページの『DB2 for i サンプル表』

以下のサンプル表は、「SQL プログラミング」、および「SQL 解説書」トピック集で参照および使用されています。

DB2 for i5/OS SQL 解説書

IBM i 7.2 の新機能

SQL プログラミングのトピック集に関する新情報や重要な変更情報についてお読みください。

TRUNCATE ステートメント

TRUNCATE ステートメントは、表からすべての行を削除するために使用できます。詳細については、127 ページの『TRUNCATE ステートメントを使用した表からの行の除去』を参照してください。

表、ビュー、および索引のシステム名

表、ビュー、または索引を作成する際に、オブジェクトのシステム名を `create` ステートメントで指定できます。詳細については、19 ページの『表の作成』および 52 ページの『ビューの作成と使用』を参照してください。

Connect by

`CONNECT BY` 構文を使用して、階層照会を定義できます。詳細については、89 ページの『再帰的照会の使用』を参照してください。

リモート表からの挿入

非ローカル表から取得したデータを使用して、ローカル表に挿入できます。詳細については、118 ページの『リモート・データベースからのデータの挿入』を参照してください。

リモート表を参照する CREATE TABLE

非ローカル表から取得した定義およびデータを使用して、ローカル表を作成できます。詳細については、26 ページの『リモート・サーバー・データを使用した表の作成』を参照してください。

プロシージャ・パラメーターのデフォルト値および CALL でのパラメーター名の使用

デフォルト値を持つように SQL プロシージャおよび外部プロシージャのパラメーターを定義できます。デフォルト値のあるパラメーターは、プロシージャの呼び出し時に省略できます。CALL ステートメントで任意の引数でパラメーター名を指定できます。詳細については、168 ページの『デフォルト・パラメーターを使用したプロシージャの定義』を参照してください。

関数パラメーターのデフォルト値および関数呼び出しでのパラメーター名の使用

SQL 関数および外部関数のデフォルト・パラメーターを定義できます。詳細については、221 ページの『デフォルト・パラメーターを使用した UDF の定義』を参照してください。引数にパラメーター名を指定して関数を呼び出すことができます。詳細については、233 ページの『名前付き引数を使用した UDF の呼び出し』を参照してください。

SQL 関数の配列サポート

SQL スカラー関数で配列を使用できます。詳細については、244 ページの『SQL のプロシージャおよび関数での配列サポート』を参照してください。

動的複合ステートメント

動的に実行される複合ステートメントで SQL ルーチン・ロジックを使用できます。詳細については、204 ページの『動的複合ステートメント』を参照してください。

複数イベント・トリガー

複数のイベントに対して単一のトリガーを定義できます。詳細については、239 ページの『複数イベント SQL トリガー』を参照してください。

| 難読化

| SQL プロシージャ、関数、またはトリガーのコンテンツを難読化できます。詳細については、247 ページの『SQL ルーチンまたは SQL トリガーの難読化』を参照してください。

| SQL ルーチン・ステップ・デバッグ

| SQL デバッグ・ビューを直接ステップスルーすることで、SQL ルーチンをデバッグできます。詳細については、246 ページの『SQL ルーチンのデバッグ』を参照してください。

| SQL および外部のルーチンの管理

| プロシージャおよび関数は、CL コマンドで管理できるシステム・オブジェクトに結合されます。詳細については、249 ページの『SQL および外部のルーチン・オブジェクトの管理』を参照してください。

| TRANSFER OWNERSHIP

| TRANSFER OWNERSHIP ステートメントは、データベース・オブジェクトの新規所有者を割り当てます。詳細については、143 ページの『SQL オブジェクトの機密保護』を参照してください。

| 列マスクおよび行許可

| 列マスクおよび行許可を使用して、特定のユーザーが表示できるデータを制限できます。詳細については、145 ページの『列マスクおよび行許可』を参照してください。

| RUNSQLSTM OPTION パラメーター

| RUNSQLSTM コマンドで、リストを生成する必要がありません。詳細については、321 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。

| RUNSQL



| この新規 CL コマンドは、単一の SQL ステートメントを実行します。詳細については、325 ページの『RUNSQL CL コマンドの使用』を参照してください。

| SQL XML プログラミング

| SQL で XML を使用する方法に関する情報はすべて、SQL XML プログラミングに移動しました。

| 新規情報または変更情報の見分け方

| 技術上の変更が加えられた場所を見分けるのに役立つように、Information Center では以下のイメージを使用しています。

- | •  イメージにより、新規または変更された情報の開始点を示します。
- | •  イメージにより、新規または変更された情報の終了点を示します。

| PDF ファイルでは、左マージンに新規および変更情報のリビジョン・バー (I) があります。

| このリリースでの新機能または変更点についてのその他の情報は、「プログラム資料説明書」を参照してください。

SQL プログラミングの PDF ファイル

この情報の PDF ファイルを表示または印刷できます。


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 ファイルの保存

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、以下のようになります。

1. ご使用のブラウザで PDF リンクを右クリックする。
2. PDF をローカルに保存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 がご使用のシステムにインストールされている必要があります。このアプリケーションは、Adobe Web サイト (<http://get.adobe.com/reader/>)  から無償でダウンロードできます。

DB2 for i 構造化照会言語の紹介

構造化照会言語 (SQL) は、リレーショナル・データベースのデータの定義と操作を行うための標準化された言語です。これらのトピックでは、DB2 for i データベースと IBM DB2 Query Manager and SQL Development Kit for i ライセンス・プログラムを使用した SQL の IBM i 実装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 プログラミング」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 Distributed Relational Database Architecture™ (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。

関連概念:

分散データベース・プログラミング

関連資料:

327 ページの『分散リレーショナル・データベース関数と 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。

SQL の概念

DB2 for i SQL は SQL 実行時サポート、プリコンパイラー、および対話式 SQL などの複数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 IBM i ライセンス・プログラムの一部であり、このサポートによって、IBM DB2 Query Manager and SQL Development Kit for i 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++
- ILE COBOL
- COBOL
- PL/I
- RPG III (RPG 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、IBM DB2 Query Manager and SQL Development Kit for i 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、308 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、IBM DB2 Query Manager and SQL Development Kit for i ライセンス・プログラムの一部です。

- Run SQL スクリプト

System i® ナビゲーターの「SQL スクリプトの実行」ウィンドウを使用すると、SQL ステートメントのスクリプトを作成、編集、実行、およびトラブルシューティングすることができます。


- SQL ステートメント実行 (RUNSQLSTM) CL コマンド

RUNSQLSTM コマンドを使用すると、ソース・ファイルまたはソース・ストリーム・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。RUNSQLSTM コマンドについて詳しくは、321 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。

- l • Run SQL (RUNSQL) CL コマンド

- l RUNSQL コマンドを使用して、単一の SQL ステートメントを実行できます。RUNSQL コマンドについて詳しくは、325 ページの『RUNSQL CL コマンドの使用』を参照してください。

- DB2 Query Manager

DB2 Query Manager 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、IBM DB2 Query Manager and SQL Development Kit for i ライセンス・プログラムの一部です。詳しくは、Query マネージャーの使用  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 for i データベースは、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、IBM i Information Center のデータベース・セクションでSQL 呼び出しレベル・インターフェースを参照してください。

- Process Extended Dynamic SQL (QSQPRCED) API

このアプリケーション・プログラミング・インターフェース (API) は、拡張動的 SQL 機能を提供します。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 について詳しくは、Process Extended Dynamic SQL (QSQPRCED) API を参照してください。API についての一般情報は、アプリケーション・プログラミング・インターフェースを参照してください。

- SQL ステートメント構文検査 (QSQCHK) API

この API は、SQL ステートメントの構文を検査します。QSQCHK API についての詳細は、SQL ステートメント構文検査 (QSQCHK) APIを参照してください。API についての一般情報は、アプリケーション・プログラミング・インターフェース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、複数のシステム間でデータを分散させることができます。これにより、区分表の定義も可能になります。詳細については、DB2 マルチシステムを参照してください。

- DB2 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 (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・プロセッサを作動させることができるということを意味します。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」トピック集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 for i 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・プロフィール	権限名または権限 ID

関連概念:

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

システム (*SYS) または SQL (*SQL) のどちらかの命名規則を DB2 for i プログラミングで使用できません。

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、SET OPTION ステートメントを使用して選択されます。

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

あるいは

| 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

関連資料:

非修飾オブジェクト名の修飾

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 FUNCTION
 | ALTER MASK
 | ALTER PERMISSION
 ALTER PROCEDURE
 ALTER SEQUENCE
 ALTER TABLE
 | ALTER TRIGGER
 COMMENT ON
 CREATE ALIAS
 CREATE FUNCTION
 CREATE INDEX
 | CREATE MASK
 | CREATE PERMISSION
 CREATE PROCEDURE
 CREATE SCHEMA
 CREATE SEQUENCE
 CREATE TABLE
 CREATE TRIGGER
 CREATE TYPE
 CREATE VARIABLE
 CREATE VIEW
 DROP
 GRANT
 LABEL ON
 RENAME
 REVOKE
 | TRANSFER OWNERSHIP

SQL データ変更ステートメント

DELETE
 INSERT
 MERGE
 | TRUNCATE
 UPDATE

SQL データ・ステートメント

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

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 DECFLOAT ROUNDING MODE
SET CURRENT DEGREE
SET CURRENT IMPLICIT XMLPARSE OPTION
SET ENCRYPTION PASSWORD
SET PATH
SET SCHEMA
SET SESSION AUTHORIZATION

動的 SQL ステートメント

ALLOCATE DESCRIPTOR
| 複合 (動的)
DEALLOCATE DESCRIPTOR
DESCRIBE
DESCRIBE CURSOR
DESCRIBE INPUT
DESCRIBE PROCEDURE
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を操作することができます。プログラム記述ファイルの対話式データ定義ユーティリティ (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:

18 ページの『データ定義言語』

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、および削除を行う、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、ビュー、順序、カタログ、索引、変数、マス
| ク、許可、および別名が含まれます。

60 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う、SQL の部分を記述します。

関連資料:

DB2 for i5/OS SQL 解説書

SQL 通信域

SQL 通信域 (SQLCA) は、SQL ステートメントの実行に関する情報をアプリケーション・プログラムに提供する一連の変数です。SQLCA は、各 SQL ステートメントの実行の終了時に更新されます。

関連概念:

SQLCA (SQL 通信域)

SQLCA を使用した 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し、最後に実行される SQL ステートメントに関する情報を提供します。GET DIAGNOSTICS ステートメントを使用して、アプリケーション・プログラムから SQL 診断域にアクセスできます。

関連概念:

SQL 診断域の使用

関連資料:

GET DIAGNOSTICS ステートメント

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、ビュー、索引、制約、トリガー、マスク、許可、順序、ストアード・プロシージャ、ユーザー定義関数、ユーザー定義タイプ、グローバル変数、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。

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:

ジャーナル管理

コミットメント制御

カタログ

- SQL カタログ は、表、ビュー、索引、プロシージャ、関数、順序、トリガー、マスク、許可、変数、制約、プログラム、パッケージ、および XSR オブジェクトを記述する表およびビューを集めたものです。

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、スキーマ内のオブジェクトについての情報を格納する、カタログ表に基づいて作成される一連のビューがあります。

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。

関連資料:

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:

DB2 マルチシステム

関連資料:

データ・タイプ

24 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、そのデータの形式は、マテリアライズ照会表の定義の元となる 1 つ以上の表から取得される事前に計算された結果になります。

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。また、

- | 別名は表の特定のメンバーを参照できます。別名は、リモート・システムを参照する RDB 名が含まれ
- | た、3 つの部分から成る名前にもすることができます。

関連資料:

別名

ビュー

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。しかし、ビューにはデータがなく、作成対象の 1 つ以上の表を論理的に表すにすぎません。

1 つのビューには、特定の表のすべての列と行、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。

関連資料:

視点

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。

各索引は、一連の列または式をキーとして定義します。これらのキーは、順序付け、グループ化、および結合で使用されます。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。

DB2 for i では、2 つのタイプの索引 (2 進基数ツリー索引とエンコード・ベクトル索引 (EVI)) がサポートされます。

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。

関連概念:

索引方針の作成

関連資料:

CREATE INDEX

制約

制約とは、データベース・マネージャーによって実施され、表内で挿入、削除、または更新できる値を制限する規則のことです。

DB2 for i 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。CREATE TABLE または ALTER TABLE ステートメントを使用すると、固有制約を作成できます。CREATE INDEX ステートメントでは、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、以下の条件のいずれかが当てはまる場合にのみ、外部キーの値が有効になる規則です。

- それらが親キーの値となっている。
- 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。CREATE TABLE または ALTER TABLE ステートメントを使用すると、検査制約を作成できます。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:

157 ページの『制約』

DB2 for i データベースは、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 for i 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。

関連タスク:

データベース内での自動イベントのトリガー

ストアド・プロシージャ

ストアド・プロシージャとは、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 for i は、外部プロシージャおよび SQL プロシージャをサポートします。外部プロシージャは、どのシステム・プログラム、サービス・プログラム、または REXX プロシージャであっても構いません。System/36 プログラムまたはプロシージャにすることはできません。SQL プロシージャは、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:

161 ページの『ストアド・プロシージャ』

プロシージャ (しばしば、ストアド・プロシージャと呼ばれる) 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す。プロシージャには、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めることができます。SQL のプロシージャの場合も、ホスト言語のプロシージャ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、ID 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:

28 ページの『順序の作成と使用』

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。しかし、順序は表から独立したオブジェクトです。順序を使用すると、値を素早く簡単に生成できます。

グローバル変数

グローバル変数とは、SQL を使用して作成、アクセス、および変更が可能な名前付き変数のことをいいます。

グローバル変数によって、セッションに固有値を指定できます。この変数は、照会、ビューの作成、insert ステートメントなどで任意の式の一部として使用できます。

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 for i 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:

205 ページの『ユーザー定義関数の使用』

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ユーザー定義関数 (UDF) とし、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データ・タイプです。

特殊データ・タイプは、組み込みタイプにマップされます。配列データ・タイプは、組み込みタイプを要素タイプおよび最大カーディナリティー値として使用して定義されます。

関連概念:

264 ページの『ユーザー定義特殊タイプ』

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

XSR オブジェクト

XSR オブジェクトは、同じ名前 XML スキーマ・リポジトリに登録された 1 つ以上の XML スキーマ文書です。

XSR オブジェクトは、XML 文書の妥当性検査の際、またはアノテーション付き XML スキーマ分解の際に使用できます。

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 (CRTSQLxxx) コマンドでリレーショナル・データベース名 (RDB パラメーター) が指定され、プログラム・オブジェクトが作成されるときに作成されます。SQL パッケージの作成 (CRTSQLPKG) コマンドを使用してパッケージを作成することもできます。

注: このコマンドの xxx は、ホスト言語標識 (ILE C の場合は CI、ILE C++ の場合は CPPI、COBOL の場合は CBL、ILE COBOL の場合は CBLI、PL/I の場合は PLI、RPG/400® の場合は RPG、ILE RPG の場合は RPGI) を表します。

SQL パッケージは、Process Extended Dynamic SQL (QSQRCD) 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。QSQRCD API 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。

関連資料:

327 ページの『分散リレーショナル・データベース関数と 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。

Process Extended Dynamic SQL (QSQPRCED) API

アプリケーション・プログラム・オブジェクト

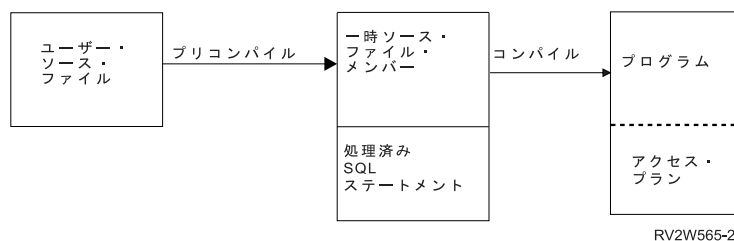
DB2 for i アプリケーション・プログラムをプリコンパイルするには、複数のオブジェクトが作成されます。

DB2 for i 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。

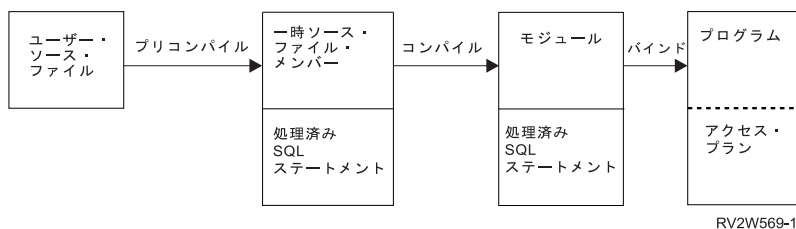
DB2 for i データベースでは、以下のオブジェクトを管理する必要が生じることがあります。

- 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

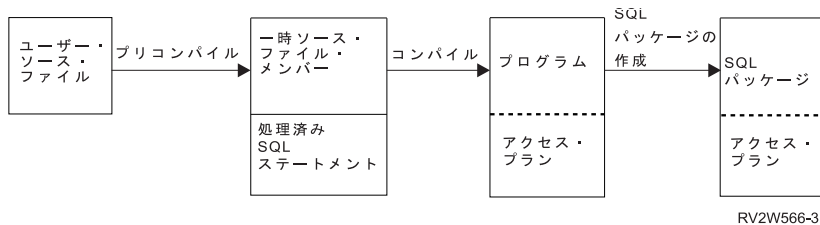
非分散型の非 ILE DB2 for i プログラムでは、管理しなければならないのは、元のソースと結果のプログラムだけです。以下の図に、非分散型の非 ILE DB2 for i プログラムのプリコンパイルおよびコンパイル・プロセスで生じる、必要なオブジェクトとステップを示します。ユーザー・ソース・ファイルはソースを一時ソース・ファイル・メンバーにプリコンパイルします。次に、このメンバーはコンパイルされてプログラムになります。



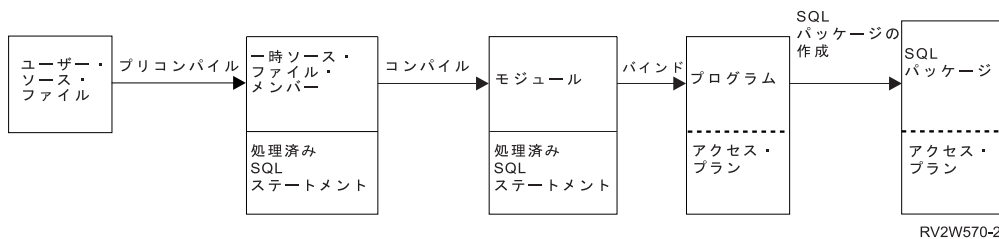
非分散型の ILE DB2 for i プログラムでは、元のソース、モジュール、および結果のプログラムまたはサービス・プログラムを管理する必要が生じることがあります。以下の図に、プリコンパイル・コマンドで OBJTYPE(*PGM) が指定された場合に、非分散型の ILE DB2 for i プログラムのプリコンパイルおよびコンパイル・プロセスで生じる、必要なオブジェクトとステップを示します。ユーザー・ソース・ファイルはソースを一時ソース・ファイル・メンバーにプリコンパイルします。次に、このメンバーはコンパイルされて、プログラムにバインドされるモジュールになります。



分散型の非 ILE DB2 for i 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。以下の図に、分散型の非 ILE DB2 for i 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。ユーザー・ソース・ファイルはソースを一時ソース・ファイル・メンバーにプリコンパイルします。次に、このメンバーはコンパイルされてプログラムになります。プログラムが作成された後に、そのプログラムを保持するために SQL パッケージが作成されます。



分散型の ILE DB2 for i 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。以下の図に、分散型の ILE DB2 for i 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。ユーザー・ソース・ファイルはソースを一時ソース・ファイル・メンバーにプリコンパイルします。次に、このメンバーはコンパイルされて、プログラムにバインドされるモジュールになります。プログラムが作成された後に、そのプログラムを保持するために SQL パッケージが作成されます。



注: DB2 for i 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。

関連タスク:

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

ソース・ファイル・メンバーまたはソース・ストリーム・ファイルには、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 Rational® Development Studio for i 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 (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

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 ライブラリーに作成します。しかし、ユーザーが、プリコンパイル・コマンドで、永続ファイル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せん。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。

関連タスク:

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。見つけることができない、またはユーザーが許可されていない表やビューなどのオブジェクトを SQL ステートメントが参照する場合、その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。

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 プログラムを、CREATE SQL (CRTSQLxxx) コマンドのリレーショナル・データベース (RDB) パラメーターを指定して正しく作成する場合。
- SQL パッケージの作成 (CRTSQLPKG) コマンドを実行する場合。

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

モジュールとは、モジュール作成 (CRTxxxMOD) コマンド (あるいは、バインド済みプログラム作成 (CRTBNDxxx) コマンドを使用してソース・コードをコンパイルすることにより作成する、Integrated Language Environment® (ILE) オブジェクトのことをいいます (xxx には C、CBL、CPP、または RPG が入ります)。

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だけに、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。

関連資料:

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 とは、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) 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 Integrated Language Environment (ILE) オブジェクトのことをいいます。

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、呼び出し側プログラムの作成時に行われます。これにより、呼び出し側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

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、および削除を行う、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、ビュー、順序、カタログ、索引、変数、マスク、許可、および別名が含まれます。

関連概念:

7 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。

関連タスク:

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。スキーマを作成するには、CREATE SCHEMA ステートメントを使用します。

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 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。

CREATE SCHEMA ステートメントを使用して、スキーマを作成することができます。例えば、DBTEMP という名前のスキーマは次のように作成します。

```
CREATE SCHEMA DBTEMP
```

関連資料:

```
CREATE SCHEMA
```

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。表を作成するには、CREATE TABLE ステートメントを使用します。

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。

- | CREATE TABLE ステートメントを使用して、表を作成することができます。表の名前を提供します。表名が有効なシステム・オブジェクト名でない場合は、オプションの FOR SYSTEM NAME 文節を使用してシステム名を指定できます。

定義には、列の名前および属性が含まれます。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))
```

関連概念:

データ・タイプ

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追加するには、CREATE TABLE ステートメントまたは ALTER TABLE ステートメントを使用します。制約を除去するには、ALTER TABLE ステートメントを使用します。

例えば、以下のように 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い限りなりません。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。

言い換えれば、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ぬか、ヌルでなければなりません。参照保全のこの定義では、次の用語を理解する必要があります。

- 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:

368 ページの『DB2 for i サンプル表』

以下のサンプル表は、「SQL プログラミング」、および「SQL 解説書」トピック集で参照および使用されています。

CREATE TABLE

ALTER TABLE

参照制約の追加および除去:

CREATE TABLE ステートメントまたは ALTER TABLE ステートメントを使用すると、参照制約を追加できます。参照制約を除去するには、ALTER TABLE ステートメントを使用します。

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。

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 (System i ナビゲーター、CL コマンド、ユーティリティ、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。

関連資料:

CREATE TABLE

ALTER TABLE

例: 参照制約の追加:

サンプル社員表内のすべての部門番号が部門表に入っていないなければならないという参照制約を定義します。この参照制約により、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。

例: 制約の除去

DEPARTMENT 表の DEPTNO 列に対する基本キーを除去すると、他の表にも影響があります。

DEPARTMENT 表に関して定義された REPORTS_TO_EXISTS 制約と EMPLOYEE 表に関して定義された WORKDEPT_EXISTS 制約も除去することになります。これは、除去す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かと判別すると (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、次のステップに従ってください。

1. 物理ファイル制約の変更 (CHGPFCS) 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. CHGPFCS 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 (DSPPCST) CL コマンドを使用して、または IBM i Navigator でデータベース保守フォルダーを探して、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。

関連概念:

参照制約内の検査保留状況

関連タスク:

検査保留状態になっている制約の処理

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。

以下の定義がコピーされます。

- 列名 (およびシステム列名)
- データ・タイプ、長さ、精度、およびスケール
- CCSID
- FIELDPROC

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。

- 列テキスト (LABEL ON)
- 列見出し (LABEL ON)
- デフォルト値
- 非表示属性
- 識別属性
- 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オプション 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。デフォルト値、非表示属性、および行変更タイム・スタンプ属性を組み込む類似のオプションがあります。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
```

関連資料:

CREATE TABLE

AS を使用した表の作成

SELECT ステートメントの結果から表を作成できます。このタイプの表を作成するには、CREATE TABLE AS ステートメントを使用します。

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。

例えば、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オプション 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。デフォルト値、非表示属性、および行変更タイム・スタンプ属性を組み込む類似のオプションがあります。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:

60 ページの『SELECT ステートメントを使用したデータの検索』

SELECT ステートメントは、データを収集するために照会を調整します。SELECT ステートメントを使用すると、特定の行を取り出したり、特定の方法でデータを取り出したりできます。

関連資料:

26 ページの『リモート・サーバー・データを使用した表の作成』

リモート・サーバーの 1 つ以上の表を参照する表をローカル・サーバーに作成できます。

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、そのデータの形式は、マテリアライズ照会表の定義の元となる 1 つ以上の表から取得される事前に計算された結果になります。

マテリアライズ照会表に対する照会の方が 1 つ以上の基本表に対する照会より実行速度が速くなると最適化プログラムが判別する場合には、照会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 ステートメント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。**STRANS** 表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。例えば、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、以下のようになります。

```
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。例えば、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:

11 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

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。一時表を作成するには、**DECLARE GLOBAL TEMPORARY TABLE** ステートメントを使用します。

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、**CREATE TABLE** ステートメントと同じで、**LIKE** または **AS** 文節を組み込みます。

例えば、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

関連資料:

DECLARE GLOBAL TEMPORARY TABLE

リモート・サーバー・データを使用した表の作成

リモート・サーバーの 1 つ以上の表を参照する表をローカル・サーバーに作成できます。

選択ステートメントとともに、コピー・オプションを指定して、デフォルト値などの属性を取得したり、新規表用にコピーされる列情報を識別したりすることができます。WITH DATA または WITH NO DATA 文節を指定して、リモート・システムから表にデータを入力するかどうかを指示する必要があります。

例えば、リモート・サーバー REMOTESYS 上の EMPLOYEE 表からの列定義が含まれた EMPLOYEE4 という名前の表を作成します。リモート・システムからのデータも含めます。

```
CREATE TABLE EMPLOYEE4 AS
  (SELECT PROJNO, PROJNAME, DEPTNO
   FROM REMOTESYS.TESTSCHEMA.EMPLOYEE
   WHERE DEPTNO = 'D11') WITH DATA
```

この表をグローバル一時表として作成することもできます (QTEMP で作成します)。この例では、新規表に複数の異なる列名を指定しています。表定義は、リモート・サーバーから列のデフォルト値を取得します。

```
DECLARE GLOBAL TEMPORARY TABLE EMPLOYEE4 (Project_number, Project_name, Department_number) AS
  (SELECT PROJNO, PROJNAME, DEPTNO
   FROM REMOTESYS.TESTSCHEMA.EMPLOYEE
   WHERE DEPTNO = 'D11') WITH DATA INCLUDING DEFAULTS
```

新規表のソースとしてリモート・サーバーを使用する場合には、以下の制限が適用されます。

- マテリアライズ照会表の文節は許可されません。
- FIELDPROC が含まれた列は選択リストにリストできません。
- リモート・サーバーが DB2 for LUW または DB2 for z/OS® の場合、コピー・オプションを指定できません。

関連資料:

23 ページの『AS を使用した表の作成』

SELECT ステートメントの結果から表を作成できます。このタイプの表を作成するには、CREATE TABLE AS ステートメントを使用します。

行変更タイム・スタンプ列の作成

行変更タイム・スタンプ列のある表内で行を追加したり変更したりするたびに、行変更タイム・スタンプ列の値は、挿入操作や更新操作の時点に対応するタイム・スタンプに設定されます。

行変更タイム・スタンプ列のデータ・タイプは TIMESTAMP でなければなりません。表に定義できる行変更タイム・スタンプ列は 1 つだけです。

表を作成するときは、表の中の 1 つの列を行変更タイム・スタンプ列として定義することができます。例えば、ORDERNO、SHIPPED_TO、ORDER_DATE、STATUS、および CHANGE_TS という名前の列を持つ表 ORDERS を作成します。CHANGE_TS を行変更タイム・スタンプ列として定義します。

```
CREATE TABLE ORDERS
  (ORDERNO SMALLINT,
   SHIPPED_TO VARCHAR(36),
   ORDER_DATE DATE,
   STATUS CHAR(1),
   CHANGE_TS TIMESTAMP FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP NOT NULL)
```

ORDERS 表に行を挿入する際に、その行に関する CHANGE_TS 列は挿入操作のタイム・スタンプに設定されます。ORDERS 内の行が更新されるたびに、その行に関する CHANGE_TS 列は、更新操作のタイム・スタンプを反映するように変更されます。

列から行変更タイム・スタンプ属性を除去できます。

```
ALTER TABLE ORDER
  ALTER COLUMN CHANGE_TS
  DROP ROW CHANGE TIMESTAMP
```

表内で列 CHANGE_TS は TIMESTAMP 列として残りますが、システムはこの列のタイム・スタンプ値を自動的に更新しなくなります。

識別列の作成および変更

識別列を使用して表に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生成されます。

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。例えば、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 に割り当てることができる値の範囲は 500 から 32 767 です。この列の値が 32 767 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。次の挿入操作は、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、列に対してデータ・タイプ 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。例えば、識別列を新規の値にして再開することもできます。

```
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:

30 ページの『識別列と順序の比較』

識別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。

116 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。

122 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、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。しかし、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)
```

順序の作成と使用

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。しかし、順序は表から独立したオブジェクトです。順序を使用すると、値を素早く簡単に生成できます。

順序は、表内の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。例えば、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。例えば、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 ステートメントを実行します。

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

例えば、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 ステートメントを実行します。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:

13 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。

識別列と順序の比較

識別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表の一部として定義することもできますし、ALTER TABLE を使用して列に追加することもできます。表の作成後、識別列の特性を変更できます。
- 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。表の内容を変更する際に、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。

これらは識別列と順序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てることができます。

関連資料:

27 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生成されます。

フィールド・プロシージャの定義

フィールド・プロシージャは、CREATE TABLE および ALTER TABLE ステートメントの FIELDPROC 節によって表に割り当てられます。フィールド・プロシージャは、1 つの列に含まれる値を変換するユーザー作成出力ルーチンです。

列の値が変更されるか、新しい値が挿入される時、値ごとにフィールド・プロシージャが呼び出され、その値を任意の方法で変換 (エンコード) できます。エンコードされた値は、保管されます。値を列から取り出すときにはデコードして元の値に戻す必要があるため、エンコードされたそれらの各値についてフィールド・プロシージャが呼び出されます。フィールド・プロシージャを使用する非派生列に対して定義される索引は、エンコードされた値を使用して作成されます。

フィールド・プロシージャが値に対して実行する変換のことを、フィールド・エンコード といいます。値を取り出すときに変換を元に戻すために同じルーチンが使用されます。その操作のことを、フィールド・デコード といいます。フィールド・プロシージャを持つ列の値は、DB2 に対して以下の 2 つの方法で記述されます。

1. CREATE TABLE または ALTER TABLE で定義された列の記述は、カタログ表 QSYS2.SYSCOLUMNS に示されます。これはフィールド・デコードされた値の説明で、列記述 と呼ばれます。
2. データベースに保管されているエンコードされた値の記述は、カタログ表 QSYS2.SYSFIELDS に示されます。これはフィールド・エンコードされた値の記述で、フィールド記述 と呼ばれます。

重要: フィールド・デコード機能は、フィールド・エンコード機能の逆の操作を忠実に行うものでなければなりません。例えば、ルーチンが「ALABAMA」を「01」にエンコードする場合、そのルーチンは「01」を「ALABAMA」にデコードしなければなりません。この規則に違反すると、予測できない結果が生じることがあります。41 ページの『フィールド・プロシージャを作成するときの一般ガイドライン』を参照してください。

フィールド・プロシージャは、デコード (取得) 時にデータのマスクングを実行することもできます。この場合、フィールド・プロシージャは、特定のユーザーまたは環境で「01」を「ALABAMA」にデコードし、他のユーザーまたは環境で代わりに「XXXXXXXX」などのマスク値を返すことができます。43 ページの『データをマスクするフィールド・プロシージャを作成するためのガイドライン』を参照してください。

フィールド・プロシージャのフィールド定義

フィールド・プロシージャは、エンコードされた値のデータ・タイプおよび属性を DB2 に対して定義するために表が作成または変更される時にも呼び出されます。その操作のことをフィールド定義 といいます。

エンコードされた値のデータ・タイプは、ROWID または DATALINK を除く任意の有効な SQL データ・タイプにすることができます。また、フィールド・プロシージャは、IDENTITY または ROW CHANGE TIMESTAMP によって生成された値を持つ列と関連付けることができません。

DDS 作成の物理ファイルが変更されてフィールド・プロシージャが追加された場合、エンコードされた属性のデータ・タイプは LOB タイプや DataLink にすることはできません。SQL 表が変更されてフィールド・プロシージャが追加された場合、エンコードされた属性の精度フィールドは、エンコードされた属性のデータ・タイプがいずれかの整数タイプである場合には 0 でなければなりません。

フィールド・プロシージャは、デフォルト値が CURRENT DATE、CURRENT TIME、CURRENT TIMESTAMP、または USER である列には追加できません。

ユーザー定義のデータ・タイプで定義された列には、ユーザー定義のデータ・タイプのソース・タイプがいずれかの許可される SQL データ・タイプである場合に、フィールド・プロシージャを設定できます。DB2 は、列の値をフィールド・プロシージャに渡す前にソース・タイプにキャストします。

フィールド・プロシージャの指定

列にフィールド・プロシージャを指定するには、CREATE TABLE または ALTER TABLE ステートメントの FIELDPROC 節に続いてプロシージャの名前を指定し、オプションでパラメーター・リストを指定します。

表の ROWID または DATALINK 列、あるいは IDENTITY または ROW CHANGE TIMESTAMP によって生成された値を持つ列では、フィールド・プロシージャは使用できません。ただし、同じ表の中の他の列では指定できます。

プロシージャ名の後ろに指定するオプションのパラメーター・リストは、括弧で囲まれた定数のリストで、リテラル・リストと呼ばれます。リテラル・リストは、DB2 によって、フィールド・プロシージャ・パラメーター値リスト (FPPVL) と呼ばれるデータ構造に変換されます。FPPVL は、フィールド定義操作の際に、フィールド・プロシージャに渡されます。そのとき、プロシージャはそれを変更するか、または変更しないで返すことができます。FPPVL の出力フォームは、変更済み FPPVL と呼ばれます。これは、列記述の一部として DB2 QSYS2.SYSFIELDS カタログ表に保管されます。フィールド・エンコードまたはフィールド・デコードのためにフィールド・プロシージャが呼び出されるときには常に、変更済み FPPVL が再度そのプロシージャに渡されます。

フィールド・プロシージャが呼び出されるとき

列に対して指定されるフィールド・プロシージャは、一般的に 3 つの状態で呼び出されます。

- フィールド定義の場合は、プロシージャを指定している CREATE TABLE または ALTER TABLE ステートメントが実行される時。この呼び出しの際に、プロシージャは以下のことを行う必要があります。
 - 列のデータ・タイプおよび属性が有効かどうかを判別する。
 - リテラル・リストを検査し、必要であればそれを変更する。
 - 列のフィールド記述を用意する。
- フィールド・エンコードの場合は、列の値をエンコードするとき。エンコードは、以下のいずれかの値に対して行われます。
 - SQL INSERT ステートメント、SQL MERGE ステートメント、またはネイティブの書き込み操作によって列に挿入された値。
 - SQL UPDATE ステートメント、SQL MERGE ステートメント、またはネイティブの更新操作によって変更された値。

- データをコピーする必要がある、ターゲット列にフィールド・プロシージャがある場合、フィールド・プロシージャが呼び出されて、コピーされるデータがエンコードされる可能性があります。例えば、SQL ステートメントの ALTER TABLE や CREATE TABLE (LIKE または *as-result-table* 節が指定されている場合)、CL コマンドの CPYF や RGZPFM などです。
- フィールド・プロシージャを持つ列と比較される値。オプティマイザーは QAQQINI オプションの FIELDPROC_ENCODED_COMPARISON を使用して、列の値をデコードするかどうか、あるいは変数、定数、または結合列をエンコードするかどうかを判別します。
- DEFAULT 値において CREATE または ALTER TABLE を実行する時に、列にフィールド・プロシージャがある場合。

後トリガーまたは読み取りトリガーがある場合、フィールド・プロシージャはそれらのトリガーの前に呼び出されます。前トリガーの場合、エンコードおよびデコード操作が含まれたフィールド・プロシージャの複数の呼び出しが存在することがあります。フィールド・プロシージャの呼び出し回数は、トリガーのタイプやトリガーがトリガー・バッファー内のデータを変更するかどうかなどの多数の要因によって異なります。データベース・マネージャーにより、確実にフィールド・プロシージャが呼び出されて、表に挿入されるデータがエンコードされます。

- フィールド・デコードの場合は、保管されていた値をフィールド・デコードで元の値に戻すとき。これは、以下のいずれかの値に対して行われます。
 - SQL SELECT または FETCH ステートメントによって、またはネイティブの読み取り操作によって取り出される値。
 - データをコピーする必要がある、ソース列にフィールド・プロシージャがある場合、フィールド・プロシージャが呼び出されて、コピーする前にデータがデコードされる可能性があります。例えば、SQL ステートメントの ALTER TABLE や CREATE TABLE (LIKE または *as-result-table* 節が指定されている場合)、CL コマンドの CPYF や RGZPFM などです。
 - フィールド・プロシージャを持つ列と比較される値。オプティマイザーは QAQQINI オプションの FIELDPROC_ENCODED_COMPARISON を使用して、列の値をデコードするかどうか、あるいは変数または定数をエンコードするかどうかを判別します。

NULL 値を処理するためにフィールド・プロシージャが呼び出されることはありません。また、表に DELETE トリガーがない場合、WHERE 節を使用しない DELETE 操作に対して呼び出されることもありません。フィールド・プロシージャは、空ストリングに対しては呼び出されます。

推奨: フィールド・プロシージャでブランクをエンコードすることは避けてください。DB2 は、長さが異なる 2 つのストリングの値を比較するとき、短い方のストリングが長い方のストリングと同じ長さになるまで一時的に適切なブランク文字 (EBCDIC や 2 バイトのブランクなど) を埋め込みます。短い方のストリングがフィールド・プロシージャを持つ列の値である場合は、エンコードされる値に対して埋め込みが行われますが、埋め込み文字はエンコードされません。そのため、プロシージャがブランクを他の文字に変更する場合、長い方のストリングの終わりにあるエンコードされたブランクは短い方のストリングの最後に埋め込まれたブランクと同じではなくなります。この状態が発生するとエラーにつながる場合があります。例えば、等しくなるはずのストリングが等しいと認識されないことがあります。そのため、フィールド・プロシージャでブランクをエンコードすることは勧められていません。

フィールド・プロシージャを実行するためのパラメーター・リスト

フィールド・プロシージャのパラメーター・リストは、一般情報をフィールド・プロシージャに渡します。

パラメーター・リストは、実行する操作を指定して、フィールド・プロシージャがエラーをシグナル通知できるようにします。DB2 は、フィールド・プロシージャ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、パラメーターはフィールド・プロシージャにアドレスで渡されます。

フィールド・プロシージャの中でパラメーターを定義し、それらを使用する場合は、特定の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おい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フィールド・プロシージャが使用する原因になったり、データベースに返される値が無効になったりする原因になります。

パラメーター 1

実行する機能を記述する小さい (2 バイトの) 整数。このパラメーターは入力のみです。以下の値がサポートされます。

- 0 – フィールド・エンコード
- 4 – フィールド・デコード
- 8 – フィールド定義

パラメーター 2

フィールド・プロシージャ・パラメーター値リスト (FPPVL) を定義する構造。

- 機能コード 8 の場合、このパラメーターは入出力です。
- 機能コード 0 および 4 の場合、このパラメーターには機能コード 8 の呼び出しに対する出力が入ります。このパラメーターは入力のみです。

パラメーター 3

列値記述子 (CVD) によって定義されるデコード・データの属性。これは CREATE TABLE または ALTER TABLE の実行時に指定された列属性です。このパラメーターは入力のみです。

パラメーター 4

デコード・データ。厳密な構造は、機能コードによって異なります。

- 機能コード 8 の場合、NULL 値です。このパラメーターは入力のみです。
- 機能コード 0 の場合、エンコードされるデータです。このパラメーターは入力のみです。
- 機能コード 4 の場合、デコード・データを配置する場所です。このパラメーターは出力のみです。

パラメーター 5

フィールド値記述子 (FVD) によって定義されるエンコード・データの属性。

- 機能コード 8 の場合、エンコード・データの属性を格納する構造。このパラメーターは出力のみです。
- 機能コード 0 または 4 の場合、機能 8 の呼び出しによって返されたエンコード・データの属性を格納する構造。このパラメーターは入力のみです。

パラメーター 6

フィールド値記述子 (FVD) によって定義されるエンコード・データ。厳密な構造は、機能コードによって異なります。

- 機能コード 8 の場合、NULL 値です。このパラメーターは入力のみです。
- 機能コード 0 の場合、エンコード・データを配置する場所です。このパラメーターは出力のみです。
- 機能コード 4 の場合、データのエンコード済み形式。このパラメーターは入力のみです。

パラメーター 7

SQLSTATE (character(5))。このパラメーターは入出力です。

このパラメーターは、フィールド・プロシージャを呼び出す前に、DB2 によって「00000」に設定されます。このパラメーターはフィールド・プロシージャで設定できます。通常、SQLSTATE はフィールド・プロシージャによって設定されませんが、以下のようにすると、データベースにエラーをシグナル通知するために使用できます。

- フィールド・プロシージャがエラーを検出する場合、SQLSTATE を「38xxx」に設定する必要があります。ここで、xxx は使用可能な複数のストリングのいずれかにできます。詳しくは、「DB2 メッセージおよびコード」を参照してください。

フィールド・プロシージャでは、警告はサポートされていません。

パラメーター 8

メッセージ・テキスト領域 (varchar(1000))。このパラメーターは入出力です。

この引数は、フィールド・プロシージャを呼び出す前に、DB2 によって空ストリングに設定されます。この引数は、SQLSTATE エラーがフィールド・プロシージャによってシグナル通知されるときに、メッセージ・テキストを送信して返すためフィールド・プロシージャが使用できる VARCHAR(1000) 値です。SQLSTATE パラメーターがフィールド・プロシージャ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。メッセージ・テキストはジョブ CCSID を使用しているという前提で処理されます。

パラメーター 9

フィールド・プロシージャの追加情報が含まれた 128 バイトの構造。このパラメーターは入力のみです。

この構造は、フィールド・プロシージャを呼び出す前に、DB2 によって設定されます。データをマスクするフィールド・プロシージャの場合、呼び出し元が、マスキングなしでデータをデコードする必要があるシステム関数であることを示します。例えば、一部の 경우에는、RGZPFM および ALTER TABLE でデータをコピーする必要が生じることがあります。フィールド・プロシージャがこのパラメーターを無視し、これらの操作の実行時にデータをマスクした場合、列データは失われます。そのため、データをマスクするフィールド・プロシージャがこのパラメーターを適切に処理するのが重要になります。

QSYSINC/H 内のインクルード SQLFP は、これらのパラメーターについて記述しています。

フィールド・プロシージャ・パラメーター値リスト (FPPVL):

フィールド・プロシージャ・パラメーター値リストは、CREATE TABLE または ALTER TABLE ステートメントで提供されたりテラル・リストを、フィールド定義の際にフィールド・プロシージャに渡します。

その時点で、フィールド・プロシージャは FPPVL を再フォーマット設定できます。この再フォーマットされた FPPVL が、フィールド・エンコードおよびフィールド・デコードの際に、変更済み FPPVL として QSYS2.SYSFIELDS に保管されてフィールド・プロシージャに渡されます。

以下の表は、FPPVL について説明しています。

表 5. sqlfpFieldProcedureParameterList_T

名前	オフセット	データ・タイプ	説明
sqlfpOptParmValueListLength	0	4 バイト整数	この構造の長さ (バイト単位)

表 5. *sqlfpFieldProcedureParameterList_T* (続き)

名前	オフセット	データ・タイプ	説明
sqlfpNumberOfOptionalParms	4	4 バイト整数	後続の値記述子の数。FIELDPROC 節のパラメーター数と同じです。リストされたパラメーターがない場合は、ゼロになります。
sqlfpParmList	8	structure sqlfpOptionalParameterValueDescriptor_T	sqlfpOptionalParameterValueDescriptor_T 項目の sqlfpNumberOfOptionalParms カウントが入ったリスト。

表 6. *sqlfpOptionalParameterValueDescriptor_T*

名前	オフセット	データ・タイプ	説明
sqlfpOptDescLength	0	4 バイト整数	この構造の長さ (バイト単位)
sqlfpParmDesc	4	構造 sqlfpParameterDescription_T	パラメーターの説明
reserved2	38	character(12)	未使用
sqlfpParmData	40		オプション・パラメーターのデータ値。 <ul style="list-style-type: none"> • 値が可変長ストリングの場合、最初の 2 バイトにその長さが入ります。 • 値が LOB または XML ストリングの場合、最初の 4 バイトにその長さが入ります。 • この値が数値の場合、データの内部的な数値表現になります。 • 日時値の場合、値は *ISO 形式になります。

フィールド・プロシーチャーのパラメーター値記述子:

パラメーター値記述子は、値のデータ・タイプおよび他の属性を記述するものです。

パラメーター値記述子は、以下の方法でフィールド・プロシーチャーと共に使用されます。

- フィールド定義の際に、フィールド・プロシーチャー・パラメーター値リスト (FPPVL) に含まれている各定数を記述します。これらのオプションのパラメーター値記述子のセットは、FPPVL 制御ブロックの一部です。
- フィールド・エンコードおよびフィールド・デコードの際に、デコードされる (列) 値およびエンコードされる (フィールド) 属性は、列値記述子 (CVD) およびフィールド値記述子 (FVD) によって記述されます。

列値記述子 (CVD) には、列値の記述が入ります。フィールド・エンコードの際、CVD はエンコードされる値を記述します。フィールド・デコードの際には、フィールド・プロシーチャーで提供されるデコードされた値を記述します。フィールド定義の際は、CREATE TABLE または ALTER TABLE ステートメントで定義されているとおりに列を記述します。

フィールド値記述子 (FVD) には、フィールド値の記述が入ります。フィールド・エンコードの際、FVD はフィールド・プロシーチャーから返されるエンコードされた値を記述します。フィールド・デコードの際には、デコードされる値を記述します。フィールド定義の際は、エンコードされた値の記述を FVD に入れる必要があります。

以下の表は、パラメーター値記述子を説明しています。

表 7. *sqlfpParameterDescription_T*

名前	オフセット	データ・タイプ	説明
sqlfpSqlType	0	2 バイト整数	このパラメーターの SQL データ・タイプ。サポートされる値については、「SQL 解説書」の『付録 D』を参照してください。
sqlfpByteLength	2	符号なし 4 バイト整数	このパラメーターの長さ (バイト単位)。日時パラメーターでは、パラメーターのストリング表記の長さ。
sqlfpLength	6	符号なし 4 バイト整数	このパラメーターの長さ (文字数)。これが文字タイプまたはグラフィック・タイプではない場合、sqlfpLength および sqlfpByteLength は同じ値になります。
sqlfpPrecision	10	2 バイト整数	これが精度を持つ数値パラメーター (精度および位取りのある 10 進数、ゾーン、およびバイナリー) の場合は、精度。
sqlfpScale	12	2 バイト整数	これが位取りを持つ数値パラメーター (精度および位取りのある 10 進数、ゾーン、およびバイナリー) の場合は、位取り。これが日付または時刻パラメーターの場合、位取りは 0。これがタイム・スタンプ・パラメーターの場合、位取りは 6。
sqlfpCcsid	14	符号なし 2 バイト整数	文字、グラフィック、または XML の場合は、このパラメーターの CCSID。
sqlfpAllocatedLength	16	符号なし 2 バイト整数	CREATE TABLE または ALTER TABLE ステートメントで列に指定された割り当ての長さ。
reserved1	18	character(14)	未使用

フィールド定義 (機能コード 8):

フィールド定義操作に対して提供される入力、および必要な出力は、次のとおりです。

- **パラメーター 1**

入力 - 実行する機能を記述する小さい (2 バイトの) 整数 (8 - フィールド定義)。

- **パラメーター 2**

入出力 - フィールド・プロシージャー・パラメーター値リスト (FPPVL) を定義する構造。これは自動実行可能なスペースです。この構造の最小の長さは、8 バイトです。この構造が返される場合の最大の長さは、32 バイトです。

- **パラメーター 3**

入力 - デコード・データの属性を格納する構造 *sqlfpParameterDescription_T*。

- **パラメーター 4**

未使用。

- **パラメーター 5**

出力 - エンコード・データの属性を格納する構造 *sqlfpParameterDescription_T*。出力 *sqlfpParameterDescription_T* は、適切な CCSID、長さ、精度、および位取りのフィールド・セットを持つ、有効なものでなければなりません。

- **パラメーター 6**
未使用。
- **パラメーター 7**
入出力 - SQLSTATE (character(5))。
- **パラメーター 8**
入出力 - メッセージ・テキスト領域 (varchar(1000))。
- **パラメーター 9**
入力 - 予約済み。

フィールド・プロシージャでエラーが返されると、SQLCODE -681 (SQLSTATE '23507') になります。これは、SQL 連絡域 (SQLCA) に設定され、さらに SQL 診断域の DB2_RETURNED_SQLCODE 条件域と RETURNED_SQLSTATE 条件域の項目にも設定されます。パラメーター 7 および 8 の内容はトークンに置かれます。トークンは、SQLCA ではフィールド SQLERRMT として、SQL 診断域では条件域の項目 MESSAGE_TEXT に置かれます。エラー・メッセージの意味は、フィールド・プロシージャによって決まります。

パラメーター 5 の sqlfpParameterDescription_T のデータが無効な場合、またはパラメーター 2 の長さが無効な場合には、SQLCODE -685 (SQLSTATE '58002') が生じます。データベース・マネージャーがフィールド・プロシージャを呼び出すことができない場合、SQLCODE -682 (SQLSTATE '57010') が返されます。

FPPVL をフィールド・プロシージャに適するように再定義して、変更済み FPPVL として返すことができますが、以下の制約があります。

- sqlfpOptParmValueListLength には変更済み FPPVL の実際の長さを含める必要があります。パラメーター・リストが返されない場合、sqlfpOptParmValueListLength を 8 に設定する必要があります。

変更済み FPPVL はカタログ表 QSYS2.SYSFIELDS に記録され、フィールド・エンコードおよびフィールド・デコードの際にフィールド・プロシージャに渡されます。変更済み FPPVL は、フィールド・プロシージャのパラメーター・リストのフォーマットと同じである必要はなく、オプションのパラメーター値記述子で定数を記述する必要もありません。

列のヌル可能性属性は、変更されないことがあります。

エンコード・データの属性が文字、グラフィック、または XML タイプである場合、CCSID 値はそのデータ・タイプで有効な CCSID に設定する必要があります。

列に非ヌルのデフォルト値がある場合、エンコードされるデフォルト値は、列のデフォルト値に許可される長さを超えてはなりません。

フィールド・エンコード (機能コード 0):

フィールド・エンコード操作に対して提供される入力、および必要な出力は、次のとおりです。

- **パラメーター 1**
入力 - 実行する機能を記述する小さい (2 バイトの) 整数 (0 - フィールド・エンコード)。
- **パラメーター 2**
入力 - 変更されたフィールド・プロシージャ・パラメーター値リスト (FPPVL) を定義する構造。
- **パラメーター 3**
入力 - デコード・データの属性を格納する、sqlfpParameterDescription_T で記述される構造。

- **パラメーター 4**

Input - エンコードされるデータ。

値が可変長ストリングの場合、最初の 2 バイトにその長さが入ります。値が LOB または XML の場合、最初の 4 バイトに長さが入ります。値が数値の場合、データの内部的な数値表現になります。日時値の場合、値は *ISO 形式になります。

- **パラメーター 5**

入力 - エンコード・データの属性を格納する、sqlfpParameterDescription_T で記述される構造。

- **パラメーター 6**

出力 - エンコード・データを配置する場所。

エンコードされた値が可変長ストリングの場合、最初の 2 バイトに長さを入れる必要があります。エンコードされた値が LOB または XML の場合、最初の 4 バイトに長さを入れる必要があります。値が数値の場合、データの内部的な数値表現になります。日時値の場合、値は *ISO 形式であることが必要です。

- **パラメーター 7**

入出力 - SQLSTATE (character(5))。

- **パラメーター 8**

入出力 - メッセージ・テキスト領域 (varchar(1000))。

- **パラメーター 9**

入力 - 予約済み。

フィールド・プロシージャでエラーが返されると、SQLCODE -681 (SQLSTATE '23507') になります。これは、SQL 連絡域 (SQLCA) に設定され、さらに SQL 診断域の DB2_RETURNED_SQLCODE 条件域と RETURNED_SQLSTATE 条件域の項目にも設定されます。パラメーター 7 および 8 の内容はトークンに置かれます。トークンは、SQLCA ではフィールド SQLERRMT として、SQL 診断域では条件域の項目 MESSAGE_TEXT に置かれます。データベース・マネージャーがフィールド・プロシージャを呼び出すことができない場合、SQLCODE -682 (SQLSTATE '57010') が返されます。

フィールド・デコード (機能コード 4):

フィールド・デコード操作に対して提供される入力、および必要な出力は、次のとおりです。

- **パラメーター 1**

入力 - 実行する機能を記述する小さい (2 バイトの) 整数 (4 - フィールド・デコード)。

- **パラメーター 2**

入力 - 変更されたフィールド・プロシージャ・パラメーター値リスト (FPPVL) を定義する構造。

- **パラメーター 3**

入力 - デコード・データの属性を格納する、sqlfpParameterDescription_T で記述される構造。

- **パラメーター 4**

出力 - デコード・データを配置する場所。

デコードされた値が可変長ストリングの場合、最初の 2 バイトに長さを入れる必要があります。デコードされた値が LOB または XML の場合、最初の 4 バイトに長さを入れる必要があります。値が数値の場合、データの内部的な数値表現になります。日時値の場合、値は *ISO 形式であることが必要です。

- **パラメーター 5**

入力 - エンコード・データの属性を格納する、sqlfpParameterDescription_T で記述される構造。

- **パラメーター 6**

入力 - エンコード・データ

値が可変長ストリングの場合、最初の 2 バイトにその長さが入ります。値が LOB または XML の場合、最初の 4 バイトに長さが入ります。値が数値の場合、データの内部的な数値表現になります。日時値の場合、値は *ISO 形式になります。

- **パラメーター 7**

入出力 - SQLSTATE (character(5))。

- **パラメーター 8**

入出力 - メッセージ・テキスト領域 (varchar(1000))。

- **パラメーター 9**

入力 - 呼び出し元が、マスキングなしでデータをデコードする必要があるシステム関数であることを示します。

フィールド・プロシージャでエラーが返されると、SQLCODE -681 (SQLSTATE '23507') になります。これは、SQL 連絡域 (SQLCA) に設定され、さらに SQL 診断域の DB2_RETURNED_SQLCODE 条件域と RETURNED_SQLSTATE 条件域の項目にも設定されます。パラメーター 7 および 8 の内容はトークンに置かれます。トークンは、SQLCA ではフィールド SQLERRMT として、SQL 診断域では条件域の項目 MESSAGE_TEXT に置かれます。データベース・マネージャーがフィールド・プロシージャを呼び出すことができない場合、SQLCODE -682 (SQLSTATE '57010') が返されます。

フィールド・プロシージャ・プログラムの例:

フィールド・プロシージャ FP1 を列 C1 に追加します。フィールド・プロシージャ FP1 には、フィールド・プロシージャが操作の対象とする列のバイト数を示す 1 つのパラメーターがあります。

```
ALTER TABLE TESTTAB ALTER C1 SET FIELDPROC FP1(10)
```

```
#include "string.h"
#include <QSYSINC/H/SQLFP>
void reverse(char *in, char *out, long length);
main(int argc, void *argv[])
{
    short *funccode = argv[1];
    sqlfpFieldProcedureParameterList_T *optionalParms = argv[2];
    char *sqlstate = argv[7];
    sqlfpMessageText_T *msgtext = argv[8];
    int bytesToProcess;
    sqlfpOptionalParameterValueDescriptor_T *optionalParmPtr;

    if (optionalParms->sqlfpNumberOfOptionalParms != 1)
    {
        memcpy(sqlstate, "38001", 5);
        return;
    }
    optionalParmPtr = (void *)&(optionalParms->sqlfpParmList);
    bytesToProcess = *((int *)&optionalParmPtr->sqlfpParmData);
    if (*funccode == 8) /* create time */
    {
        sqlfpParameterDescription_T *inDataType = argv[3];
        sqlfpParameterDescription_T *outDataType = argv[5];
        if (inDataType->sqlfpSqlType != 452 &&
            inDataType->sqlfpSqlType != 453 ) /* only support fixed length char */
        {
            memcpy(sqlstate, "38002", 5);
        }
    }
}
```



```

        return;
    }
    /* do something here to determine the result data type */
    /* ..... */
    /* in this example input and output types are exactly the same */
    /* so just copy */
    memcpy(outDataType, inDataType, sizeof(sqlfParameterDescription_T));
}
else if (*funccode == 0)    /* encode */
{
    char *decodedData = argv[4];
    char *encodedData = argv[6];

    reverse(decodedData, encodedData, bytesToProcess);
}
else if (*funccode == 4)    /* decode */
{
    char *decodedData = argv[4];
    char *encodedData = argv[6];

    reverse(encodedData, decodedData, bytesToProcess);
}
else /* unsupported option -- error */
    memcpy(sqlstate, "38003",5);
}

void reverse(char *in, char *out, long length)
{
    int i;
    for (i=0;i<length; ++i) {
        out[length - (i+1)] = in[i];
    }
}

```

フィールド・プロシージャを作成するときの一般ガイドライン

フィールド・プロシージャを作成するときには、以下の考慮事項を検討する必要があります。

- ILE *PGM オブジェクトでなければなりません。 *SRVPGM、OPM *PGM、および JAVA オブジェクトはサポートされません。
- フィールド・プロシージャ *PGM オブジェクトに対する権限は、フィールド・プロシージャが表に追加されるときに検査されます。フィールド・プロシージャが呼び出されるときには、権限検査は行われません。
 - フィールド・プロシージャ・プログラムは、それを作成したユーザーのプロファイルの下で実行されるように作成します。この方法では、そのプログラムへの同じレベルの権限を持たないユーザーに対してエラーが発生しません。
 - USRPRF(*OWNER) および *EXCLUDE の PUBLIC 権限を使用してプログラムを作成します。フィールド・プロシージャ・プログラムに対する権限を USER(*PUBLIC) に認可しないでください。他のユーザーが、フィールド・プロシージャ・プログラムの変更および置き換えをしないようにします。
- フィールド・プロシージャでは SQL は許可されません。
- エンコードまたはデコードされるデータが NULL 値の場合には、フィールド・プロシージャは呼び出されません。
- エンコード操作では、パック 10 進数およびゾーン 10 進数値は、ユーザー・フィールド・プロシージャ・プログラムを呼び出す前に、設定された符号に変換されます。
- フィールド・プロシージャは、決定論的でなければなりません。SQE では、結果のキャッシングは QAAQINI FIELDPROC_ENCODED_COMPARISON に基づいて実行されます。

- フィールド・プロシージャは並列可能でなければならず、またマルチスレッド環境で実行可能でなければなりません。RPG の場合、これは、THREAD(*SERIALIZE) 制御仕様を指定する必要があることを意味します。
- 隔離および非隔離環境の両方で実行可能でなければなりません。
- ACTGRP(*NEW) を使用してプログラムを作成することはできません。ACTGRP(*CALLER) を使用してプログラムを作成すると、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。
- フィールド・プロシージャ・プログラムは、短時間の実行を想定しています。フィールド・プロシージャ・プログラムでは、コミットメント制御およびネイティブ・データベース操作を避けることをお勧めします。
- プログラムは物理ファイルのライブラリー内に作成します。
- フィールド・プロシージャ・プログラムでエラーが発生したかまたは検出された場合、フィールド・プロシージャ・プログラムは SQLSTATE およびメッセージ・テキスト・パラメーターを設定する必要があります。SQLSTATE パラメーターがエラーを知らせるように設定されていない場合、データベースはフィールド・プロシージャが正常に実行されたと思なします。この場合、ユーザー・データが一貫していない状態で終了する可能性があります。

警告: フィールド・プロシージャは、アプリケーション機能を提供し、情報を管理するための有効な方法です。しかし、フィールド・プロシージャ・プログラムを使用すると、悪意のある人がシステムに「トロイの木馬」を作成できるようになる可能性があります。¹そのため、表を変更する権限を持つユーザーを制限することが重要です。オブジェクト権限を注意して管理していれば、一般のユーザーがフィールド・プロシージャ・プログラムを追加する権限を持つことはありません。

索引に関する考慮事項:

索引は、CRTPF、CRTLF、CHGPF、または CHGLF のいずれかのコマンドの RECOVER パラメーターに基づいて、IPL 時に復元できます。フィールド・プロシージャを持つ列に基づく索引については、特別な考慮事項があります。

フィールド・プロシージャを持つ列を含んだ式に対して索引キーが構築される場合、または関連フィールド・プロシージャを持つ列を疎索引の基準で参照する場合には、フィールド・プロシージャ内での PASE(QSH) および JAVA の使用を避ける必要があります。PASE または JAVA を使用する必要がある場合、索引を RECOVER(*NO) に変更することを考慮してください。このようにすると、索引は IPL プロセスの際に復元されるのではなく、オープン操作の際に復元されるようになります。

スレッドについての考慮事項:

フィールド・プロシージャは、そのフィールド・プロシージャを開始した操作と同じジョブで実行されます。ただし、フィールド・プロシージャは、開始要求のスレッドとは異なる別のシステム・スレッド (fenced または not fenced) で実行されることも、同じシステム・スレッドになることもあります。

フィールド・プロシージャは同じジョブの中で実行されるために、開始要求と同じ環境の多くを共有します。ただし、それは別のスレッドのもとで実行されることもあるので、スレッドに関する以下の考慮事項が必要になります。

1. 歴史上、トロイの木馬というのは中が空洞の巨大な木馬で、中にはギリシャ人兵士がたくさん入っていました。木馬がトロイの壁の内側に入られると、兵士たちが木馬の中から出てきてトロイ人と闘ったのです。コンピューターの世界では、有害な機能が潜んだプログラムのことを、よくトロイの木馬と呼びます。

- フィールド・プロシージャは、要求が開始されたときに活動状態であったプログラム借用権限を継承しません。フィールド・プロシージャの権限は、フィールド・プロシージャ・プログラムに関連付けられている権限から、または実行しているユーザーの権限から得られます。
- フィールド・プロシージャは、2 次スレッドで実行がブロックされている操作は実行できません。
- フィールド・プロシージャ・プログラムは、名前付き活動化グループのもと、あるいは、その呼び出し元の活動化グループ (ACTGRP パラメーター) の中で実行されるように作成しなければなりません。
*CALLER を指定するプログラムは、デフォルト活動化グループで実行されます。

データをマスクするフィールド・プロシージャを作成するためのガイドライン

フィールド・プロシージャを使用して、データのデコード時に特定のユーザーまたは環境に対してデータをマスクすることができます。データをマスクするフィールド・プロシージャは、データが破損しないようにするために特殊な状態を処理するようにコーディングする必要があります。

データをマスクするフィールド・プロシージャでは、以下の特殊な状態を処理する必要があります。

- フィールド・デコード
 - マスキングは、フィールド・デコードでのみ実行する必要があります。フィールド・エンコードで実行してはなりません。フィールド・エンコードでマスキングが実行された場合、マスク・データが表に保管され、実際の値が失われます。
 - 一部の 경우에는、システム・コードでデータを内部的にコピーする必要があります (このような場合、データはユーザーに返されません)。例えば、一部の 경우에는、RGZPFM、ALTER TABLE、および CRTDUPOBJ でデータを内部的にコピーする必要があります。同様に、トリガーに内部的に渡されるデータをマスクしてはなりません。これらの操作中には、データの読み取り時にフィールド・デコードが行われ、データの書き込み時にフィールド・エンコードが行われます。このような場合にフィールド・デコード中にマスキングが実行されると、マスク・データが書き込まれ、実際のデータが失われます。

破損しないようにするには、フィールド・プロシージャに対する 9 番目のパラメーターで、これがマスキングを実行してはならないシステム操作であるのかどうかを指示します。フィールド・デコード時にこのパラメーターを検査するようにフィールド・プロシージャを作成することが重要であり、このパラメーターにより、マスキングを実行してはならないことが指示されている場合には、フィールド・プロシージャは、ユーザーや環境に関係なく、マスクしてはなりません。

- フィールド・エンコード
 - ネイティブの更新および挿入操作では、フィールド・プロシージャは、マスク・データがフィールド・プロシージャに渡されたときにそれを識別し、特殊な処置を実行できる必要があります。例えば、クレジット・カード番号列をマスクするようにフィールド・プロシージャを作成することが考えられます。同じユーザーに、READ および UPDATE 操作を実行する RPG アプリケーションを介して表を読み取る権限および更新する権限があることがあります。READ が実行されると、ユーザーが表示できないようにクレジット・カード番号がマスクされますが、ユーザーが UPDATE を実行すると、マスク・データが UPDATE 操作でデータベースに渡され、フィールド・プロシージャが呼び出されてデータがエンコードされます。渡された値がマスクされていることをフィールド・プロシージャが認識していない場合、マスク・データはエンコードされて表に保管され、行内の元の値は、エンコードされたマスク・データで破損してしまいます。

破損しないようにするために、フィールド・プロシージャは、フィールド・エンコード時にデータがマスクされていることを認識する必要があります。データをエンコードする代わりに、フィールド・プロシージャは、7 番目のパラメーターで警告 SQLSTATE 値の「09501」を返す必要があります。

- UPDATE 操作では、「09501」は、列の現行値を使用する必要があることを DB2 に指示します。
- INSERT 操作では、「09501」は、関連付けられた列値でデフォルト値を使用する必要があることを DB2 に指示します。

照会の考慮事項: データをマスクするフィールド・プロシージャがある表の列を参照する照会には、以下のいくつかの考慮事項が適用されます。

- 最適化プログラムが照会を実装する方法に応じて、同じ照会が、ユーザーごとまたは環境ごとに異なる行および値を返すことがあります。これは、最適化プログラムが比較を実行するため、または照会内の式を評価するためにデータをデコードする必要がある場合に生じます。あるユーザーに対してマスキングが実行されるが、別のユーザーに対しては実行されない場合、デコード操作の結果は非常に異なるものになるため、この 2 人のユーザーに対して結果の行および値も非常に異なるものになることがあります。

例えば、フィールド・プロシージャが、ユーザー・プロファイル MAIN に対してマスキングなしでデータを返し (デコードし)、ユーザー・プロファイル QUSER に対してマスキングありでデータを返す (デコードする) ものとしします。アプリケーションに、以下の照会が含まれているものとしします。

```
SELECT * FROM orders WHERE cardnum = '112233'
```

デフォルトでは、最適化プログラムは、検索条件を (論理的に) 以下のように実装しようとしします。

```
WHERE cardnum = FieldProc ENCODE('112233')
```

DB2 が定数「112233」のエンコードされたバージョンを、orders 表に保管されている CARDNUM 値のエンコードされたバージョンと比較できるため、これは最適な実装です。最適化プログラムが比較を実行するためにデータをデコードしなかったため、照会は、MAIN ユーザー・プロファイルと QUSER ユーザー・プロファイルに対して同じ行を返します。唯一の違いは、QUSER には、CARDNUM 列の結果行にマスク値が表示されることです。

フィールド・プロシージャ列を参照する照会の実装は、QAQQINI FIELDPROC_ENCODED_COMPARISON オプションで制御できます。このオプションのデフォルト値は *ALLOW_EQUAL です。このオプションにより、最適化プログラムは、エンコードされた値を使用して比較を実装できます。

前の例で、FIELDPROC_ENCODED_COMPARISON オプションを *NONE に変更した場合、照会は 2 人のユーザーに対して異なる行を返します。値が *NONE の場合、以下のように、等価比較は DB2 によって内部的に実装されます。

```
WHERE FieldProc DECODE(cardnum)='112233'
```

この場合、DB2 は、元の定数「112233」と比較するために、表内のすべての行で CARDNUM 値をデコードする必要があります。つまり、MAIN ユーザー・プロファイルの比較では、デコードおよびマスク解除されたカード番号値 (112233、332211 など) を「112233」と比較します。MAIN ユーザー・プロファイルには、指定されたカード番号 (112233) に関連付けられた注文が検出されます。しかし、照会では、QUSER ユーザー・プロファイルに対しては行が返されません。これは、QUSER の比較では、カード番号のマスク値 (****33、****11、など) を定数「112233」と比較しているからです。

QAQQINI FIELDPROC_ENCODED_COMPARISON オプションがフィールド・プロシージャにどのように影響するかについては詳しくは、Information Center のトピック『データベース・パフォーマンスおよび Query 最適化』を参照してください。

- マテリアライズ照会表の REFRESH は、QAQQINI FIELDPROC_ENCODED_COMPARISON オプションの影響を受けます。マテリアライズ照会表が、マスクするフィールド・プロシージャが含まれた列を

参照している場合、マスク解除したデータを表示できるユーザーによって、MQT の REFRESH を発行する必要があります。そうしないと、MQT での結果が全ユーザーに対して正しくなくなります。

- CREATE TABLE LIKE、CREATE TABLE AS、DECLARE GLOBAL TEMPORARY TABLE LIKE、または DECLARE GLOBAL TEMPORARY TABLE AS は、QAQQINI FIELDPROC_ENCODED_COMPARISON オプションの影響を受けます。マスク解除されたデータを表示できないユーザーによってステートメントが発行された場合、結果の表に、マスク・データが含まれます。
- OPNQRYF および Query/400 は、QAQQINI FIELDPROC_ENCODED_COMPARISON オプションの影響を受けません。最適化プログラムは常に、値をデコードして処理します (FIELDPROC_ENCODED_COMPARISON オプションが *NONE の場合と同様)。
- DDS 作成論理ファイルの選択/除外も、QAQQINI FIELDPROC_ENCODED_COMPARISON オプションの影響を受けません。論理ファイルは、値をデコードして処理されます (FIELDPROC_ENCODED_COMPARISON オプションが *NONE の場合と同様)。

ベスト・プラクティス: データをマスクするフィールド・プロシージャがある場合には、以下の 2 つの QAQQINI オプションを使用することを強くお勧めします。

FIELDPROC_ENCODED_COMPARISON

FIELDPROC_ENCODED_COMPARISON のデフォルト・オプションは *ALLOW_EQUAL であり、これはデータをマスクしないフィールド・プロシージャに対して非常に適切に機能します。ただし、データをマスクするフィールド・プロシージャを使用する場合には、*NONE が非常にセキュアな推奨オプションです。

CACHE_RESULTS

CACHE_RESULTS のデフォルト・オプションは *SYSTEM です。多くの場合、このオプションは適切に機能します。ただし、データをマスクするフィールド・プロシージャを使用する場合には、CACHE_RESULTS には *JOB を指定する必要があります。

これらの 2 つのオプションは、データをマスクするフィールド・プロシージャの動作に影響する可能性があるため、以下のように、許可されたユーザーのみが 新規または別の QAQQINI オプションを指定できるようにすることも重要になります。

- CHGQRYA コマンド

許可されたユーザーのみが CHGQRYA コマンドを実行できることを確認します。デフォルトでは、ジョブ制御 (*JOBCTL) 特殊権限または QIBM_DB_SQLADM 関数使用権のあるユーザーに対してのみ、CHGQRYA コマンドは許可されます。

- QUSRSYS/QAQQINI ファイル

許可されたユーザーのみが QUSRSYS/QAQQINI ファイルを作成でき、また既にこのファイルが存在する場合に更新できることを確認します。デフォルトでは、*PUBLIC は QUSRSYS に対する *USE 権限を備えていますが、これは、新規 QUSRSYS.QAQQINI ファイルを作成するには十分な権限ではありません。

データをマスクするフィールド・プロシージャ・プログラムの例:

フィールド・プロシージャ FP1 を列 C1 に追加します。フィールド・プロシージャ FP1 には、フィールド・プロシージャが操作の対象とする列のバイト数を示す 1 つの追加パラメーターがあります。

```
ALTER TABLE TESTTAB ALTER C1 SET FIELDPROC FP1(10)
```

```
#include "string.h"  
#include <QSYSINC/H/SQLFP>
```

```

void reverse(char *in, char *out, long length);
main(int argc, void *argv[])
{
    short *funccode = argv[1];
    sqlfpFieldProcedureParameterList_T *optionalParms = argv[2];
    char *sqlstate = argv[7];
    sqlfpMessageText_T *msgtext = argv[8];
    int bytesToProcess;
    sqlfpOptionalParameterValueDescriptor_T *optionalParmPtr;
    sqlfpInformation_T *info = argv[9];
    int masked;

    if (optionalParms->sqlfpNumberOfOptionalParms != 1)
    {
        memcpy(sqlstate,"38001",5);
        return;
    }
    optionalParmPtr = (void *)&(optionalParms->sqlfpParmList);
    bytesToProcess = *((int *)&optionalParmPtr->sqlfpParmData);

/*****
/* CREATE CALL
*****/
    if (*funccode == 8)          /* create time */
    {
        sqlfpParameterDescription_T *inDataType = argv[3];
        sqlfpParameterDescription_T *outDataType = argv[5];
        if (inDataType->sqlfpSqlType !=452 &&
            inDataType->sqlfpSqlType !=453 ) /* only support fixed length char */
        {
            memcpy(sqlstate,"38002",5);
            return;
        }
        /* do something here to determine the result data type */
        /* ..... */
        /* in this example input and output types are exactly the same */
        /* so just copy */
        memcpy(outDataType, inDataType, sizeof(sqlfpParameterDescription_T));
    }

/*****
/* ENCODE (WRITE) CALL
*****/
    else if (*funccode == 0)    /* encode */
    {
        char *decodedData = argv[4];
        char *encodedData = argv[6];

        /* Detect that the value passed on encode is masked.      */
        /* Return 09501 to tell DB that:                          */
        /* - The field should not be updated for an update operation */
        /* - The default value should be used for an insert operation*/
        if ( memcmp(decodedData, "XXXXXXXXXX", 12) == 0 )
        {
            memcpy(sqlstate,"09501",5);
        }
        else
        {
            reverse(decodedData, encodedData, bytesToProcess);
        }
    }

/*****
/* DECODE (READ) CALL
*****/
    else if (*funccode == 4)    /* decode */
    {

```

```

char *decodedData = argv[4];
char *encodedData = argv[6];

/* The 9th paramter indicates that the column must not be      */
/* masked. For exmaple, during ALTER TABLE or RGZPFM.        */
if ( info->sqlfpNoMask == '1' )
{
    reverse(encodedData, decodedData, bytesToProcess);
    return;
}

else
{
    reverse(encodedData, decodedData, bytesToProcess);
    /* Mask the data when appropriate                          */
    /* Assume mask is set to 0 when it should not be masked   */
    /* and 1 when it shoulbe be masked                        */
    if (masked == 1)
    {
        memcpy(decodedData, "XXXXXXXXXXXX", 12);
    }
}

return;
}

/*****
/* ERROR- UNSUPPORTED OPTION
*****/
else /* unsupported option -- error */
    memcpy(sqlstate, "38003",5);
}

/*****
/* REVERSE
*****/
void reverse(char *in, char *out, long length)
{
    int i;
    for (i=0;i<length; ++i) {
        out[length - (i+1)] = in[i];
    }
}

```

LABEL ON ステートメントを使用した記述ラベルの作成

テキスト記述が、オブジェクト (表や索引など) にとって役立ったり、列テキストまたは列見出しとして役立つことがあります。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。オブジェクトまたは列のラベルの長さは最大 50 バイトであり、列見出しのラベルの長さは最大 60 バイトです (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 LABEL ON ステートメントは、2 つのレベルを指定した DBCS 列見出しを提供します。

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

関連資料:

LABEL

COMMENT ON を使用した SQL オブジェクトの記述

表やビューなどの SQL オブジェクトの作成後、COMMENT ON ステートメントを使用して、以後の参照用にオブジェクト情報を提供することができます。

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。オブジェクトにすでに注釈が含まれている場合には、古い注釈は新しい注釈により置換されます。

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 列に格納されません。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'
```

関連資料:

COMMENT

表定義の変更

列の追加、既存の列定義 (長さやデフォルト値など) の変更、既存の列の削除、制約の追加、または制約の除去により、表の定義を変更します。

表の定義を変更するには、SQL ALTER TABLE ステートメントを使用します。

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。

関連資料:

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 VARCHAR2 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。

関連資料:

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:

『可能なデータ・タイプの変換』

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

可能なデータ・タイプの変換

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 8. 可能な変換

変換前のデータ・タイプ	変換後のデータ・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	10 進浮動
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	10 進浮動
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	10 進浮動
大きい整数、整数、小さい整数	浮動
浮動	10 進数
浮動	数値
浮動	大きい整数、整数、小さい整数
浮動	10 進浮動
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。例えば、以下のような表定義の表があるとします。

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

以下の ALTER TABLE ステートメントの実行後に、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。

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

関連資料:

49 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。

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
```

関連資料:

ALTER TABLE

ALTER TABLE ステートメントの操作の順序

ALTER TABLE ステートメントの操作は、定義されている順序で実行されます。

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。別名を作成するには、CREATE ALIAS ステートメントを使用します。

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。

例えば、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 は表 MYLIB.MYTABLE の別名であるとしします。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
```

関連資料:

```
CREATE ALIAS
```

ビューの作成と使用

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。

例えば、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します。

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

- 1 ビュー名 EMP_MANAGERS がシステム・オブジェクト名としては長すぎるため、FOR SYSTEM NAME 文節を使用してシステム名を提供できます。この文節を追加しない場合には、システム・オブジェクトに対して EMP_M00001 のような名前が生成されます。

ビューを作成し終わったら、表と同じように SQL ステートメントの中で使用することができます。また、ビューを使用して、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。

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、前述のビューを定義することもできます。例えば、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。例えば、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.EMPPROJECT
 WHERE PROJNO = 'MA2112' OR
        PROJNO = 'MA2113' OR
        PROJNO = 'AD3111')
```

このビューには以下のデータがあります。

表9. UNION としてのビューの作成結果

EMPNO
000060
000150
000160
000170
000180
000190

表 9. UNION としてのビューの作成結果 (続き)

EMPNO
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の検査レベルを指定するために、WITH CHECK OPTION 文節を使用してビューを作成することもできます。

関連概念:

60 ページの『SELECT ステートメントを使用したデータの検索』

SELECT ステートメントは、データを収集するために照会を調整します。SELECT ステートメントを使用すると、特定の行を取り出したり、特定の方法でデータを取り出したりできます。

137 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、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:

102 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。

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す。この文節は、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。

WITH CHECK OPTION 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。このオプションは、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。

関連資料:

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。

例えば、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 文節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、CASCADED CHECK OPTION 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。

例えば、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。

例: カスケード検査オプション:

次の例では、検査オプションを指定して定義されている従属ビューと、検査オプションを指定せずに定義されている従属ビューに対する検査オプションの実施方法を説明します。

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** ステートメント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、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、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、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。

索引の作成

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)
```

表内の列のデータと完全一致しない索引を作成することもできます。例えば、以下のように、大文字バージョンの社員名を使用する索引を作成できます。

```
CREATE INDEX CORPDATA.INX2 ON CORPDATA.EMPLOYEE (UPPER(LASTNAME))
```

SQL で使用可能なほとんどの式を、キー列の定義中に使用できます。

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・ツリーを共有します。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:

137 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。

索引方針の作成

関連資料:

CREATE INDEX

グローバル変数の作成と使用

グローバル変数を使用して、セッションに特定の変数値を指定できます。

CREATE VARIABLE ステートメントを使用して、グローバル変数を作成します。ユーザー・クラスを定義するグローバル変数を作成する例を以下に示します。

```
CREATE VARIABLE USER_CLASS INT DEFAULT (CLASS_FUNC(USER))
```

この変数には、CLASS_FUNC という関数を呼び出したときの結果に基づく初期値セットがあります。この関数では、USER 特殊レジスター値に基づいて管理者やクラークなどのクラス値を割り当てることになっています。

グローバル変数は、それが初めて参照されるセッションでインスタンス化されます。それが設定された後は、セッション内で明示的に変更されない限り、その値が保持されます。

グローバル変数を照会内で使用すると、どの結果を返すかを指定できます。以下の例では、部門 A00 のすべての社員のリストが示されます。USER_CLASS 値が 1 のグローバル変数を持つセッションだけが、これらの社員の給与を参照できます。

```
SELECT EMPNO, LASTNAME, CASE WHEN USER_CLASS = 1 THEN SALARY ELSE NULL END  
FROM EMPLOYEE  
WHERE WORKDEPT = 'A00'
```

グローバル変数は、式が許可されるすべてのコンテキストで使用できます。ホスト変数とは異なり、グローバル変数は CREATE VIEW ステートメント内で使用できます。

既存オブジェクトの置換

CREATE ステートメントを使用すれば既存のオブジェクトを置換できます。必ずしも最初にオブジェクトを除去する必要はありません。

多くの SQL オブジェクトでは、CREATE SQL ステートメントを使用するときにオプションで既存のオブジェクトを置換できます。既存のオブジェクトは、新しいオブジェクトが作成される前に事実上除去されることとなります。以下の SQL ステートメントには、そのようなオプションがあります。

- CREATE ALIAS
- CREATE FUNCTION
- CREATE PROCEDURE
- CREATE SEQUENCE
- CREATE TRIGGER
- CREATE VARIABLE
- CREATE VIEW

これらのステートメントのいずれかで置換オプションを使用するとき、既存オブジェクトの特権は保持されます。オブジェクト定義は、新しい定義によって置換されます。

例: シーケンスの作成または置換

MYSEQUENCE という名前のシーケンスを作成するか、またはその名前のシーケンスが存在する場合にそれを置換するには、以下の SQL ステートメントを使用します。

```
CREATE OR REPLACE SEQUENCE MYSEQUENCE AS BIGINT
```

そのシーケンスがまだ存在しない場合には、それが作成されます。存在する場合には、既存のシーケンスの特権が新しいシーケンスに転送されます。

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atalogだけが更新されません。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。カタログ情報の挿入、削除、または更新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・ビューに対する SELECT 特権が必要です。

関連資料:

DB2 for i5/OS のカタログ・ビュー

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。SYSTABLES ビューは、オブジェクト・タイプ (表またはビュー)、オブジェクト名、オブジェクトの所有者、オブジェクトがあるスキーマなどの情報を提供します。

次の例の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。

次の例の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'
```

この例の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。

各列の特定の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。

例えば、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。

例えば、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
```

関連資料:

DROP

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う、SQL の部分を記述します。

関連概念:

7 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 ステートメントは、データを収集するために照会を調整します。SELECT ステートメントを使用すると、特定の行を取り出したり、特定の方法でデータを取り出したりできます。

SQL が検索条件を満たす行を見つけない場合は、+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:

23 ページの『AS を使用した表の作成』

SELECT ステートメントの結果から表を作成できます。このタイプの表を作成するには、CREATE TABLE AS ステートメントを使用します。

52 ページの『ビューの作成と使用』

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

SELECT ステートメントの基本的な形式と構文は、複数の必須文節とオプション文節から成ります。

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 (例えば、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。例えば、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。例えば、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 10. 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。例えば、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の結果は 1 行になります。

表 11. 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:

137 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:

76 ページの『複雑な検索条件の定義』

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

78 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。例えば、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
 - グローバル変数
 - 関数
 - 特殊レジスター
 - スカラー全選択
 - 別の式

例えば、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。例えば、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。例えば、次の通りです。

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

- 特殊レジスタは、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。例えば、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
```

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、DB2 for i 「SQL 解説書」のトピック集で説明されています。

関連概念:

129 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:

76 ページの『複雑な検索条件の定義』

検索条件には、= や > などの基本的な比較述部に加え、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、複数の比較演算子がサポートされています。

比較演算子	説明
<> または != または ! =	等しくない
=	等しい
<	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。例えば、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 つの行が返されません。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返されます。

例えば、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。例えば、**CORPDATA.PROJECT** 表で記述されている各主要プロジェクトに従事している社員の数のリストを入手したいとします。次のステートメントを発行することができます。

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

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

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

複数の列または式に基づいて行をグループ分けすることを指定することもできます。例えば、**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
```

結果は次のようになります。

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

WORKDEPT	SEX	AVG_WAGES
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:

137 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、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:

68 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。例えば、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
```

結果は次のようになります。

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:

137 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、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:

279 ページの『カーソルの使用』

SQL が SELECT 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。

例えば、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
```

結果は次のようになります。

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

LASTNAME	WORKDEPT
PEREZ	D21
HENDERSON	E11
SCHNEIDER	E11
SETRIGHT	E11
SCHWARTZ	E11
SPRINGER	E11
WONG	E21

注: ナ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。例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, FIRSTNAME
       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。例えば、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 FIRSTNAME 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 FIRSTNAME
       FROM CORPDATA.EMPLOYEE
       ORDER BY LASTNAME CONCAT FIRSTNAME
```

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。例えば、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返し、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:

137 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:

65 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。例えば、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:

279 ページの『カーソルの使用』

SQL が SELECT 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

ヌル値とは、ある行の列値が存在しないことを意味します。ヌル値は不明な値で、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、WHERE および HAVING 文節の中で条件として使用できます。例えば、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
```

結果は次のようになります。

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、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。

例えば、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
```

結果は次のようになります。

C1	C2
1	2
1	-
2	-
-	2

NULL 値の使用に関する詳細については、DB2 for iSQL 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。CURRENT DATE などの特殊レジスターには、SQL ステートメント中で参照できる情報が含まれています。

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 CLIENT_ACCTNG CLIENT ACCTNG	クライアント接続の会計情報ストリング。
CURRENT CLIENT_APPLNAME CLIENT APPLNAME	クライアント接続のアプリケーション名。
CURRENT CLIENT_PROGRAMID CLIENT PROGRAMID	クライアント接続のプログラム ID。
CURRENT CLIENT_USERID CLIENT USERID	クライアント接続のクライアント・ユーザー ID。
CURRENT CLIENT_WRKSTNNAME CLIENT WRKSTNNAME	クライアント接続のワークステーション名。
CURRENT DATE CURRENT_DATE	現在の日付。
CURRENT DEBUG MODE	ルーチンの作成または変更時に使用されるデバッグ・モード。
CURRENT DECFLOAT ROUNDING MODE	10 進浮動小数点値の処理時に使用される丸めモード。
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。
CURRENT IMPLICIT XMLPARSE OPTION	検証なしで暗黙に解析するとき、XML データに対して使用する空白文字処理オプション。
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。

特殊レジスター	内容
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。
CURRENT_USER CURRENT_USER	ジョブのプライマリー許可 ID。
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、特殊レジスターの値は、リモート・システムで判別されません。

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。

例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。例えば、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
```

関連資料:

データ型間のキャスト

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せん。

- 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。例えば、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。

6 を超える精度で CURRENT TIMESTAMP 特殊レジスターまたは TIMESTAMP データ・タイプの変数を使用した場合、タイム・スタンプ値はシステム・クロックと固有ビットの組み合わせになります。固有ビットは、昇順に割り当てられます。そのため、すべての精度のタイム・スタンプの比較演算は、タイム・スタンプが割り当てられたときの正確な順序を表します。

関連資料:

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、および CURRENT TIMESTAMP の特殊レジスターの 1 つを使用することによって、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。例えば、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。

関連資料:

SQL における日付/時刻の算術演算

行変更式

ROW CHANGE TIMESTAMP 式と ROW CHANGE TOKEN 式は、行が最後に変更された時点を判別するのに使用できます。

表に関する ROW CHANGE TIMESTAMP 式を使用するには、行変更タイム・スタンプ列があるように表を定義しなければなりません。

以下の照会は、少なくとも過去 4 週間の注文をすべて検索し、それらの注文が最後に変更された時点をリストできます。

```
SELECT ORDERNO, ROW CHANGE TIMESTAMP FOR ORDERS
FROM ORDERS
WHERE ORDER_DATE < CURRENT DATE - 4 WEEKS
```

ROW CHANGE TOKEN 式は、行変更タイム・スタンプがある表とない表の両方で使用できます。このタイム・スタンプは、行の変更点を表します。表に行変更タイム・スタンプがある場合は、この式はタイム・スタンプから派生します。表に行変更タイム・スタンプがない場合は、この式は行ベースでない内部変更時に基づくので、行変更タイム・スタンプのある表の場合ほど正確ではありません。

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)。例えば、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ため、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せん。

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:

137 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、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、= や > などの基本的な比較述部に加え、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。例えば、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。例えば、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。例えば、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。例えば、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せん。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、DB2 for i SQL 解説書のトピック集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。例えば、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。例えば、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:

129 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。

137 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、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:

62 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。

63 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。

述部

LIKE に関する特殊な考慮事項:

ここでは LIKE 述部の使用に関するいくつかの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、すべて空白で初期設定されます。

例えば、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 だと想定すると次の値が返されます。

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

注: 返される値のすべては 'ABC' で始まり、最低 6 つの空白で終わります。空白が使用されているのは、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いためです。

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

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

WHERE 文節内の複数の検索条件:

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。例えば、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。例えば、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。例えば、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。例えば、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:

62 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 12. 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
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となります。

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となります。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

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 13. 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 14. 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、結合操作が必要になります。順序付けはネストされた表の式です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 15. 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
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 for i 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せられて、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、CORPDATA.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 文節で提供されません。

この照会は次の出力を戻します。

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、INNER JOIN 構文の使用時に実行するものと同じ結合を実行するには、WHERE 文節で結合条件と追加の選択条件の両方を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:

結合条件を定義する速記方法として USING 文節を使用できます。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。

例えば、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.PROJECT INNER JOIN CORPDATA.EMPPROJECT
USING (PROJNO, ACTNO)
WHERE ACSTDATE > '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.PROJECT INNER JOIN CORPDATA.EMPPROJECT
ON CORPDATA.PROJECT.PROJNO = CORPDATA.EMPPROJECT.PROJNO AND
CORPDATA.PROJECT.ACTNO = CORPDATA.EMPPROJECT.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せんが、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:

クロス結合は、デカルト積結合とも呼ばれ、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せられる結果表を返します。

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 16. 表 A

ACOL1	ACOL2
A1	AA1
A2	AA2
A3	AA3

表 17. 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 SELECT 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:

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、不一致行も双方の表から戻します。

すべての社員と、その社員らのすべてのプロジェクトを調べたいとします。社員が割り当てられていないプロジェクトと、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 FULL OUTER JOIN CORPDATA.PROJECT
ON EMPNO = RESPEMP
WHERE LASTNAME > 'S'
```

社員が割り当てられていないプロジェクトはないので、この照会は左方外部結合と同じ行を戻します。結果は以下のとおりです。

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	-

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 表を結合する必要があります。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'
```

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

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

照会の結果は以下のようになります。

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

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

例えば、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て、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

- | 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。このタイプのデータを照会するには、階層照会または再帰的共通表式を使用できます。

再帰的データの一例として、各種部品およびそれを構成する副部品にわたって処理する部品表 (BOM) アプリケーションがあります。例えば、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せん。

DB2 for i は、再帰的照会を定義するための 2 つの方法を提供します。最初の方法は階層照会と呼ばれ、CONNECT BY 文節を使用して、親の行を子の行に関連付ける方法を定義します。2 番目の方法では、再帰的共通表式を使用します。これは、共通表式を使用して最初の (シード) 行を定義してから、UNION を使用して子の行を決定する方法を定義します。

再帰的照会を定義するこれらの方法にはそれぞれ、利点と欠点があります。CONNECT BY 構文の方が、はるかに理解しやすいですが、照会でデータを派生させる方法が少なくなります。CONNECT BY は、照会内の任意の場所の任意の副選択で指定できます。再帰的共通表式では、子の行を生成するために UNION を定義する方法のためのオプションが、より多く用意されています。

connect by 再帰的照会と再帰的共通表式照会とは、動作の違いがいくつかあります。まず、循環データの処理方法が違います。この違いについては、例で説明します。次に、connect by では、兄弟間でのソートが可能です。これについても例で示します。最後に、2 つの実装では、再帰の実装に使用されるキューにデータを配置する方法が違います。デフォルトでは、再帰的共通表式のデータは、先入れ先出しの幅優先順序になる傾向にあります。connect by では、順序は深さ優先になるように設計されています。つまり、再帰的ステップの行は親の行の直後に配置されることとなります。再帰的共通表式の構文では、SEARCH 文節を追加することで、階層順序を深さ優先にするか幅優先にするかを選択できます。connect by 構文では常に深さ優先です。

これらの再帰的方法のための旅行プランナーの例では、フライトと鉄道路線を使用して、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', 'Hawaii', '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);

CREATE TABLE FLIGHTSTATS(FLIGHT# CHAR(5),
                          ON_TIME_PERCENT DECIMAL(5,2),
                          CANCEL_PERCENT DECIMAL(5,2));
```

```

INSERT INTO FLIGHTSTATS VALUES('234', 85.0, 0.20);
INSERT INTO FLIGHTSTATS VALUES('2334', 92.0, 0.10);
INSERT INTO FLIGHTSTATS VALUES('5473', 86.2, 0.10);
INSERT INTO FLIGHTSTATS VALUES('247', 91.0, 0.10);
INSERT INTO FLIGHTSTATS VALUES('2356', 91.0, 0.10);
INSERT INTO FLIGHTSTATS VALUES('3256', 92.0, 0.10);
INSERT INTO FLIGHTSTATS VALUES('63', 90.5, 0.10);
INSERT INTO FLIGHTSTATS VALUES('37', 87.0, 0.20);
INSERT INTO FLIGHTSTATS VALUES('2337', 80.0, 0.20);
INSERT INTO FLIGHTSTATS VALUES('77', 86.0, 0.10);
INSERT INTO FLIGHTSTATS VALUES('437', 81.0, 0.10);
INSERT INTO FLIGHTSTATS VALUES('59', 85.0, 0.10);
INSERT INTO FLIGHTSTATS VALUES('534', 87.0, 0.10);
INSERT INTO FLIGHTSTATS VALUES('5234', 88.0, 0.20);
INSERT INTO FLIGHTSTATS VALUES('84', 88.0, 0.1);
INSERT INTO FLIGHTSTATS VALUES('824', 93.0, 0.10);
INSERT INTO FLIGHTSTATS VALUES('94', 92.0, 0.10);
INSERT INTO FLIGHTSTATS VALUES('104', 93.0, 0.10);

```

CONNECT BY 階層照会の使用

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。

```

SELECT CONNECT_BY_ROOT departure AS origin, departure, arrival, LEVEL AS flight_count
FROM flights
START WITH departure = 'Chicago'
CONNECT BY PRIOR arrival = departure

```

この照会は次の情報を戻します。

表 18. 前述の照会の結果

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

この階層照会には複数の部分があります。始めの選択では、再帰の初期シードを定義しています。この場合、START WITH departure を「Chicago」とした flights 表からの行です。CONNECT BY 文節を使用して、既に生成されている行を「接続」して、照会の後続の反復用にさらなる行を生成する方法を定義しています。PRIOR 単項演算子は、前の行の結果に基づいて新規行を選択する方法を DB2 に指示します。START WITH 文節の結果によって選択された再帰的結合列 (通常 1 列であるが、複数列も可能) が、PRIOR キーワードによって参照されています。これは、前の行の ARRIVAL 都市が、新しい行の DEPARTURE 都市の PRIOR 値になることを意味します。これは、節 CONNECT BY PRIOR arrival = departure にカプセル化されます。

この照会例では、他の 2 つの connect by フィーチャーが示されています。初期化ステップで決定されて、生成されるすべての再帰的結果行で同じになる固定式値を定義するために、単項演算子 CONNECT_BY_ROOT が使用されています。通常、複数の START WITH 値がある可能性があるため、そ

特定の反復における開始値です。この照会では、結果セットで、Chicago からの異なる目的地オプションの *ORIGIN* を定義します。START WITH 文節が複数の都市を選択した場合、*ORIGIN* は、行が開始値として使用した都市を示します。

LEVEL は、connect by 再帰の使用時に使用可能な 3 つの疑似列の 1 つです。*LEVEL* の値は、現在行の再帰レベルを反映します。この例では、*LEVEL* は、*ORIGIN* の都市 (Chicago) から別の *ARRIVAL* 都市に行くために必要なフライト数も反映しています。

階層照会は、それに相当する再帰的共通表式照会と同じように実行され、同じ結果セットを生成します。96 ページの『再帰的共通表式および再帰的ビューの使用』を参照してください。唯一の違いは、返される行の順序です。connect by 照会は、行を深さ優先順序で返します。つまり、結果セットの各行が、その親の行の直後に配置されます。再帰的共通表式照会は、行を幅優先順序で返します。つまり、あるレベルのすべての行が返されてから、前のレベルで生成されたすべての行が返されます。

例: CONNECT BY を使用した再帰に使用される 2 つの表

ここでは、Chicago を起点として、フライトに加えて鉄道による移動オプションを追加し、行くことができる都市と必要な乗り継ぎ回数が知りたい場合を考えます。

以下の connect by 照会は、そのような情報を返します。なお、対応する再帰的共通表式の例 (98 ページの『例: 再帰的共通表式を使用した再帰に使用される 2 つの表』) では、鉄道とフライトの乗り継ぎ回数を識別したり、当該目的地までのチケットのコストを合計したりすることもできます。その計算は、より複雑であるが、より柔軟な再帰的共通表式構文を使用した場合に可能になる導出例です。その機能は、connect by 構文を使用した場合には使用できません。

```
SELECT CONNECT_BY_ROOT departure AS departure, arrival, LEVEL - 1 connections
FROM
  ( SELECT departure, arrival FROM flights
    UNION
    SELECT departure, arrival FROM trains) t
START WITH departure = 'Chicago'
CONNECT BY PRIOR arrival = departure;
```

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
Chicago	Miami	0
Chicago	Lima	1
Chicago	Frankfurt	0
Chicago	Vienna	1
Chicago	Beijing	1
Chicago	Moscow	1
Chicago	Tokyo	2
Chicago	Hawaii	3
Chicago	Washington	0
Chicago	Boston	1
Chicago	Toronto	1

この例では、再帰をフィードするデータ・ソースが 2 つあります (フライトのリストと列車のリスト)。最終結果では、都市間の移動に必要な乗り継ぎ回数が分かります。

例: CONNECT BY を使用した兄弟の順序付け

再帰的共通表式の欠点の 1 つは、特定の列値に基づいて兄弟間で結果を順序付けることができないことです。これは、connect by を使用した場合には可能です。例えば、New York からの目的地を出力するときに、同時に当該目的地までのチケットのコストなどの特定の値によって兄弟間で階層データを順序付ける必要がある場合、ORDER SIBLINGS BY 文節を指定することでそれを実現できます。

```
SELECT CONNECT_BY_ROOT departure AS origin, departure, arrival,  
       LEVEL level, price ticket_price  
FROM   flights  
START WITH departure = 'New York'  
CONNECT BY PRIOR arrival = departure  
ORDER SIBLINGS BY price ASC
```

この照会は次の情報を戻します。

表 20. 前述の照会の結果

ORIGIN	DEPARTURE	ARRIVAL	LEVEL	TICKET_PRICE
New York	New York	LA	1	330
New York	LA	Tokyo	2	530
New York	Tokyo	Hawaii	3	330
New York	New York	London	1	350
New York	London	Athens	2	340
New York	Athens	Nicosia	3	280
New York	New York	Paris	1	400
New York	Paris	Rome	2	340
New York	Paris	Madrid	2	380
New York	Paris	Cairo	2	480

結果表は、起点都市を New York として可能なすべての目的地を示しています。すべての兄弟目的地 (同じ出発都市を起点とする目的地) が、チケット価格でソートされて出力されます。例えば、Paris からの目的地は、Rome、Madrid、および Cairo です。これらは、チケット価格の昇順で順序付けられて出力されます。なお、出力では、New York からの最初の目的地として New York 発 LA 行きが示されます。これは、London または Paris への直行便のチケット価格 (それぞれ 350 と 400) よりもチケット価格が低い (330) ためです。

例: CONNECT BY を使用した循環データ・チェック

再帰的プロセスで重要なことは、再帰的プログラム、再帰的照会のいずれであっても、再帰は有限でなければならぬということです。有限でない場合、無限ループに入ることになります。CONNECT BY は、照会に対して制御が効かなくなることがないように、常に無限再帰がないかを検査し、該当する循環を自動的に終了するという点で、再帰的共通表式とは異なります。

デフォルトでは、connect by は循環データを検出すると、SQL エラーの「SQ20451: Cycle detected in hierarchical query」を発行します。このエラーにより、照会が終了するため、結果が返されません。

結果が必要で、無限循環が停止すればよいだけの場合、CONNECT BY 文節で NOCYCLE キーワードを指定できます。これは、循環データに対してエラーを発行しないことを意味します。

CONNECT_BY_ISCYCLE 疑似列とともに NOCYCLE オプションを使用すると、循環データを検出し、必要に応じてデータを修正することができます。

以下の行を *FLIGHTS* 表に挿入すると、Paris 発 Cairo 行き、Cairo 発 Paris 行きという経路ができてしまうため、無限再帰が生じる可能性があります。

```
INSERT INTO flights VALUES ('Cairo', 'Paris', 'Atlantic', '1134', 440);
```

以下の照会では、*NOCYCLE* を指定した循環データの許容を示します。また、*CONNECT_BY_ISCYCLE* 疑似列を使用して循環行を識別し、関数 *SYS_CONNECT_BY_PATH* を使用して、目的地に至るすべての就航都市の *Itinerary* (旅程) ストリングを作成します。*SYS_CONNECT_BY_PATH* は、*CLOB* データ・タイプとして実装されているため、深い再帰を反映する大きな結果列が得られます。

```
SELECT CONNECT_BY_ROOT departure AS origin, arrival,
       SYS_CONNECT_BY_PATH(TRIM(arrival), ': ') itinerary, CONNECT_BY_ISCYCLE cyclic
FROM flights
START WITH departure = 'New York'
CONNECT BY NOCYCLE PRIOR arrival = departure;
```

この照会は次の情報を戻します。

表 21. 前述の照会の結果

ORIGIN	ARRIVAL	ITINERARY	CYCLIC
New York	Paris	: Paris	0
New York	Rome	: Paris : Rome	0
New York	Cairo	: Paris : Cairo	0
New York	Paris	: Paris : Cairo : Paris	1
New York	Madrid	: Paris : Madrid	0
New York	London	: London	0
New York	Athens	: London : Athens	0
New York	Nicosia	: London : Athens : Nicosia	0
New York	LA	: LA	0
New York	Tokyo	: LA : Tokyo	0
New York	Hawaii	: LA : Tokyo : Hawaii	0

なお、多くの場合、循環データを反映する結果セット行は、*START WITH* 文節で循環のどこで開始するかによって異なります。

例: *CONNECT BY* での疑似列 *CONNECT_BY_ISLEAF*

再帰データを処理しているとき、再帰がそれ以上なくなる行を知りたい場合があります。つまり、階層内でリーフ行である行や子がない行についてです。

以下の照会では、どの目的地が最終目的地なのか (つまり、どの目的地に出発便がないのか) が分かります。*CONNECT_BY_ISLEAF* 疑似列は、リーフではない場合に 0、リーフの場合に 1 になります。また、*WHERE* 述部で *CONNECT_BY_ISLEAF* を指定して、リーフ行のみを表示することもできます。

```
SELECT CONNECT_BY_ROOT departure AS origin, arrival,
       SYS_CONNECT_BY_PATH(TRIM(arrival), ': ') itinerary, CONNECT_BY_ISLEAF leaf
FROM flights
START WITH departure = 'New York'
CONNECT BY PRIOR arrival = departure;
```

この照会は次の情報を戻します。

表 22. 前述の照会の結果

ORIGIN	ARRIVAL	ITINERARY	LEAF
New York	Paris	: Paris	0
New York	Rome	: Paris : Rome	1
New York	Cairo	: Paris : Cairo	1
New York	Madrid	: Paris : Madrid	1
New York	London	: London	0
New York	Athens	: London : Athens	0
New York	Nicosia	: London : Athens : Nicosia	1
New York	LA	: LA	0
New York	Tokyo	: LA : Tokyo	0
New York	Hawaii	: LA : Tokyo : Hawaii	1

例: CONNECT BY での join 述部および where 文節の選択

多くの場合、データの階層性は 1 つの表に反映されますが、行の出力を完全に決定するために、その結果を追加の表に結合する必要が生じます。

connect by 照会では、DB2 for i でサポートされる任意のタイプの結合 (INNER JOIN、LEFT OUTER JOIN、LEFT EXCEPTION JOIN など) を使用できます。JOIN 文節を明示的に使用した場合、ON 文節に指定した述部がまず適用されてから、connect by 操作が適用され、connect by 照会内のすべての WHERE 文節は再帰後に適用されます。再帰的プロセス結果があまりにも早期に終了してしまうことがないように、WHERE 選択は、connect by の後に適用されます。

以下の照会では、ON_TIME_PERCENT が 90% を超えている、New York 発のすべてのフライトを検索しています。

```
SELECT CONNECT_BY_ROOT departure AS origin, departure, arrival,
       flight_number, on_time_Percent AS onTime
FROM flights INNER JOIN flightstats ON flight_number = flight#
WHERE on_time_percent > 90
START WITH departure = 'New York'
CONNECT BY PRIOR arrival = departure;
```

この照会は次の情報を戻します。

表 23. 前述の照会の結果

ORIGIN	DEPARTURE	ARRIVAL	FLIGHT#	ONTIME
New York	Paris	Cairo	63	90.50
New York	Paris	Madrid	3256	92.00
New York	London	Athens	247	91.00
New York	Athens	Nicosia	2356	91.00
New York	LA	Tokyo	824	93.00
New York	Tokyo	Hawaii	94	92.00

この照会は、JOIN 構文を使用しなくても表現できます。照会最適化プログラムは、最初に処理する必要がある join 述部である述部を WHERE 文節から取り除き、再帰後に評価する残りのすべての WHERE 述部を残します。


```

| SELECT CONNECT BY ROOT departure AS origin, departure, arrival, flight_number, on_time_percent AS onTime
|   FROM flights, flightstats
|   WHERE flight_number = flight# AND on_time_percent > 90
|   START WITH departure = 'New York'
|   CONNECT BY PRIOR arrival = departure;

```

この 2 番目の例では、WHERE 述部がより複雑な場合に、*flights* 表と *flightstats* 表との間の JOIN 述部を明示的に取り除き、ON 文節と WHERE 文節の両方を使用することで、最適化プログラムを支援する必要が生じることがあります。

再帰プロセスの一部として追加検索条件を適用する場合 (例えば、定時パーセンテージが 90% 未満のフライトを除外する場合)、join 述部と WHERE 文節を使用して派生表に結合を入れることで、結合結果を制御することもできます。

```

| SELECT CONNECT BY ROOT departure AS origin, departure, arrival, flight_number, on_time_percent AS onTime
|   FROM (SELECT departure, arrival, flight_number, on_time_percent
|         FROM flights, flightstats
|         WHERE flight_number = flight# AND on_time_percent > 90) t1
|   START WITH departure='New York'
|   CONNECT BY PRIOR arrival = departure;

```

別の選択肢として、START WITH 文節および CONNECT BY 文節に選択述部を入れることができます。

```

| SELECT CONNECT BY ROOT departure AS origin, departure, arrival, flight_number, on_time_percent AS onTime
|   FROM flights, flightstats
|   WHERE flight_number = flight#
|   START WITH departure = 'New York' AND on_time_percent > 90
|   CONNECT BY PRIOR arrival = departure AND on_time_percent > 90

```

この場合、定時統計が 90% を超える、New York からの直行便がないため、うまくいきません。再帰にシードするものがないため、照会から返される行はありません。

再帰的共通表式および再帰的ビューの使用

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 24. 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

表 24. 前述の照会の結果 (続き)

ORIGIN	DEPARTURE	ARRIVAL	FLIGHT_COUNT
Chicago	Moscow	Tokyo	3
Chicago	Tokyo	Hawaii	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、選択する行ごとに 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)
```

このビュー定義の反復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。比較として、connect by 再帰では、SELECT が許可される任意の場所で使用できるため、ビュー定義に簡単に含めることができます。

例: 再帰的共通表式を使用した 2 つの出発都市

Chicago または New York 発のフライトを利用する場合の可能な目的地とその費用を検索するとします。

```
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 25. 前述の照会の結果

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	Hawaii	2	1,190
Chicago	Hawaii	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: 再帰的共通表式を使用した再帰に使用される 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 26. 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	Hawaii	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: 再帰的共通表式の 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、再帰的ビューでは直接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。

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關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 が使用する列です。

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 27. 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	Hawaii	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 28. 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	Hawaii	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。ご覧のとおり、深さ優先または幅優先処理で使用される列の値に基づいて順序付けが行われていません。順序付けを行うには、再帰の CONNECT BY 形式で使用可能な ORDER SIBLINGS BY 構造を使用できます。

例: 再帰的共通表式を使用した循環データ

再帰の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、再帰的ビューでは直接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 || 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 || 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
```

この照会は次の情報を戻します。

表 29. 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	Hawaii	New York Los Angeles Tokyo Hawaii	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。循環が見つかったら、特殊な列 (このケースでは *CYCLIC_DATA*) で、結果セットの循環行に、文字値「1」が設定されます。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* 列に循環が見つかったら、それ以上データの処理は行われず、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。述部 *WHERE*
| *CYCLIC_DATA* = 0 を追加することで、循環行の除外を選択できます。

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。例えば、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.EMPPROJECT
  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'
```

この照会は以下の中間結果表を戻します。

CORPDATA.EMPLOYEE からの EMPNO

000060

000150

000160

000170

000180

000190

000200

000210

CORPDATA.EMPLOYEE からの EMPNO

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'
```

この照会は別の中間結果表を戻します。

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、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
```

この照会は、結合された結果表を戻し、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。例えば、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:

137 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:

52 ページの『ビューの作成と使用』

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 キーワードの指定:

UNION 演算の結果で重複を残したいときは、UNION ではなく UNION ALL キーワードを指定してください。

このトピックでは、102 ページの『副選択結合時の UNION キーワードの使用』と同じステップと例を使用します。

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.EMPPROJECT
 WHERE PROJNO='MA2112' OR
        PROJNO= 'MA2113' OR
        PROJNO= 'AD3111'
 ORDER BY EMPNO
```

この照会は、重複行が含まれ、順序付けられた結果表を戻します。

EMPNO

000060

000150

000150

000150

000160

000160

000170

000170

000170

000170

000180

000180

000190

000190

000190

000200

000210

EMPNO

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.EMPPROJECT
```

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.EMPPROJECT)
```

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。

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部門 D11 の社員を含む従業員数のリストを検出したいとします。

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。

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

```
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'
```

この照会は中間結果表を戻します。

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'
```

この照会は別の中間結果表を戻します。

CORPDATA.EMPPROJACT からの EMPNO

000230

000230

000240

000230

000230

000240

000230

000150

000170

000190

CORPDATA.EMPPROJECT からの EMPNO

000170
000190
000150
000160
000180
000170
000210
000210

ステップ 3. SQL は、最初の間結果表を取り、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.EMPPROJECT
     WHERE PROJNO='MA2112' OR
           PROJNO= 'MA2113' OR
           PROJNO= 'AD3111'
ORDER BY EMPNO
```

この照会は、結合された結果表を戻し、値が昇順に並びます。

EMPNO

000060
000200
000220
200170
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。

- | プロジェクト MA2112、MA2113、および AD3111 が割り当てられている、部門 D11 内の社員を含む従業員番号のリストを検出したいとします。

INTERSECT 操作は両方の結果セットにあるすべての従業員番号を戻します。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。

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

この照会では中間結果表を戻します。

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'
```

この照会では別の中間結果表を戻します。

CORPDATA.EMPPROJACT からの EMPNO

000230

000230

000240

000230

000230

000240

000230

000150

000170

000190

000170

000190

000150

000160

CORPDATA.EMPPROJECT からの EMPNO

000180

000170

000210

000210

ステップ 3. SQL は、最初の間結果表を取り、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.EMPPROJECT
 WHERE PROJNO='MA2112' OR
        PROJNO= 'MA2113' OR
        PROJNO= 'AD3111'
ORDER BY EMPNO
```

この照会は、結合された結果表を戻し、値が昇順に並びます。

EMPNO

000150

000160

000170

000180

000190

000210

データ取り出しエラー

データ取り出し時に発生するエラーを SQL が処理する方法を理解するには、この情報を利用してください。

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string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string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string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でない。
- +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 (例えば、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。
- +364 - DECFLOAT 算術計算エラー。
- +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

単一行または複数行を表またはビューに追加するには、INSERT ステートメントの形式を使用してください。

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。
- **グローバル変数。** グローバル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、例えば 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 select 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。
- XML 列では、NULL 値以外に許可されるデフォルトはありません。

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時、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。

関連資料:

INSERT

VALUES 文節を使用した行の挿入

表に単一行または複数行を挿入するには、INSERT ステートメント中で VALUES 文節を使用します。

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。例えば、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.EMPPROJECT
```

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。

関連資料:

118 ページの『リモート・データベースからのデータの挿入』

選択ステートメントを使用してリモート・サーバーから行を取得して、ローカル・サーバー上の表に挿入できます。

ブロック化 INSERT ステートメントを使用した複数の行の挿入

ブロック化 INSERT ステートメントを使用すると、1 つの INSERT 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。

例えば、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 IBM i プロダクトである分散アプリケーションでサポートされます。

関連概念:

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、以下の規則を考慮する必要があります。

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。

例えば、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。例えば、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:

27 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生成されます。

スカラー関数

挿入値の選択

SELECT ステートメントの FROM 文節に INSERT ステートメントを指定すると、挿入される行の値を取り出せます。

1 つ以上の行を表に挿入する際に、挿入操作の結果行を選択できます。これらの行には以下の値のうち 1 つが入ります。

- 識別列、ROWID 列、または行変更タイム・スタンプ列などの、生成された列の値
- 列で使用されたデフォルト値
- 複数行挿入操作で挿入されたすべての行のすべての値
- 挿入前トリガーによって変更された値

以下の例は、次のように定義されている表を使用します。

```
CREATE TABLE EMPSAMP
(EMPNO    INTEGER GENERATED ALWAYS AS IDENTITY,
 NAME     CHAR(30),
 SALARY   DECIMAL(10,2),
 DEPTNO   SMALLINT,
 LEVEL    CHAR(30),
 HIREDATE VARCHAR(30) NOT NULL DEFAULT 'New Employee',
 HIREDATE DATE NOT NULL WITH DEFAULT)
```

新入社員に関する行を挿入し、EMPNO、HIREDATE、および HIREDATE で使用された値を参照するには、次のステートメントを使用します。

```
SELECT EMPNO, HIREDATE, HIREDATE
FROM FINAL TABLE ( INSERT INTO EMPSAMP (NAME, SALARY, DEPTNO, LEVEL)
VALUES('Mary Smith', 35000.00, 11, 'Associate'))
```

戻り値は、EMPNO の場合は生成された値、HIRETYPE の場合は 'New Employee'、および HIREDATE の場合は現在日付になります。

リモート・データベースからのデータの挿入

選択ステートメントを使用してリモート・サーバーから行を取得して、ローカル・サーバー上の表に挿入できます。

昨日の売り上げについて、REMOTESYS 上の SALES 表からのすべての行を挿入するには、以下のステートメントを使用します。

```
INSERT INTO SALES
  (SELECT * FROM REMOTESYS.TESTSCHEMA.SALES WHERE SALES_DATE = CURRENT DATE - 1 DAY)
```

DB2 for i は REMOTESYS に接続して SELECT を実行し、選択された行をローカル・システムに返し、ローカルの SALES 表に挿入します。CURRENT DATE の値は、REMOTESYS での現在日付になります。

3 つの部分から成るオブジェクト名、または表またはビューの 3 つの部分から成る名前を参照するように定義された別名により、アプリケーション・サーバーへの暗黙的接続が作成されるため、サーバー認証項目が存在している必要があります。サーバー名、ユーザー ID、およびパスワードを指定して、アプリケーション要求側でサーバー認証項目の追加 (ADDSVRAUTE) コマンドを使用します。サーバー名およびユーザー ID は、大文字で入力する必要があります。

```
ADDSVRAUTE USRPRF(yourprf) SERVER(DRDASERVERNAME) USRID(YOURUID) PASSWORD(yourpwd)
```

DRDA のサーバー認証の使用法に関する追加詳細については、『分散データベース・プログラミング』を参照してください。

関連資料:

114 ページの『選択ステートメントによる行の挿入』

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。

UPDATE ステートメントを使用した表内のデータの変更

表またはビュー内のデータを更新するには、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 ...
```

ある社員が異動になったとします。この移動を反映するように CORPDATA.EMPLOYEE 表を更新するには、以下のステートメントを実行します。

```
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。以下のタイプの値を指定することができます。

- **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。
- **グローバル変数。** 列の現行値をグローバル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 (例えば、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。

以下の UPDATE ステートメントは多数のさまざまな値を使用します。

```
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、この UPDATE ステートメントを書き込むこともできます。

```
UPDATE EMPLOYEE
  SET (WORKDEPT, PHONENO, JOB)
    = ('D11', '7213', 'DESIGNER')
  WHERE EMPNO = '000270'
```

関連資料:

UPDATE

スカラー副選択を使用する表の更新

スカラー副選択を使用すると、ある表の 1 つ以上の列を、別の表から選択した 1 つ以上の値で更新できません。

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

別の表の行からの値を用いて、ある表の行全体を更新できます。

マスター・クラス・スケジュール表を、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。例えば、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:

148 ページの『ジャーナル処理』

DB2 for i 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。

149 ページの『コミットメント制御』

DB2 for i コミットメント制御サポートは、更新、挿入、削除操作またはデータ定義言語 (DDL) 操作などの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、参照制約のある表に関する UPDATE 規則について解説します。

例えば、DEPARTMENT 表内のある部門に、PROJECT 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 ステートメントは、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 ステートメントが許可されると、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。

例えば、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:

27 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生成されます。

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 30. 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.
```

表 30. 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
<pre>EXEC SQL WHENEVER NOT FOUND GO TO CLOSE-THISEMP END-EXEC.</pre>	
<pre>EXEC SQL FETCH NEXT FROM THISEMP FOR 5 ROWS INTO :DEPTINFO :IND-ARRAY END-EXEC.</pre>	<p>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。</p>
<p>... 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。もしあれば、そのレコードを新たな最低金額 \$500.00 として更新する。</p>	
<pre>EXEC SQL FETCH RELATIVE :NUMBACK FROM THISEMP END-EXEC.</pre>	<p>... 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。</p>
<pre>EXEC SQL UPDATE CORPDATA.EMPLOYEE SET BONUS = 500 WHERE CURRENT OF THISEMP END-EXEC.</pre>	<p>... 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。</p>
<pre>EXEC SQL FETCH RELATIVE :NUMBACK FROM THISEMP FOR 5 ROWS INTO :DEPTINFO :IND-ARRAY END-EXEC.</pre>	<p>... 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))</p>
<p>... 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。</p>	
<p>... ブランチで戻り、次の行ブロックを検索して処理する。</p>	
<pre>CLOSE-THISEMP. EXEC SQL CLOSE THISEMP END-EXEC.</pre>	

関連資料:

279 ページの『カーソルの使用』

SQL が SELECT 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。

行の削除では、その行全体が除去されます。DELETE ステートメントは、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 は表から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 ...
```

例えば、部門 D11 が別の場所に移転するとします。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します。

```
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。

関連概念:

127 ページの『TRUNCATE ステートメントを使用した表からの行の除去』
表からすべての行を除去するときは、TRUNCATE ステートメントを使用します。

関連資料:

DROP

DELETE

参照制約付き表からの行の除去

表に基本キーがあって従属関係がない場合、または表に外部キーのみあって基本キーがない場合は、DELETE ステートメントは参照制約がない表の場合と同じ方法で機能します。表に基本キーと従属表がある場合、DELETE ステートメントは、表に関して指定された削除規則に従って機能します。

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 操作が失敗します。

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。

例えば、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。

例えば、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。

例えば、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 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るとします。

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'
```

368 ページの『DB2 for i 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 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 31. 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 32. 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

表 32. 部分的な EMPLOYEE 表 (続き): DELETE ステートメントの前の内容の一部

EMPNO	FIRSTNME	MI	LASTNAME	WORKDEPT	PHONENO	HIREDATE
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 33. 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

TRUNCATE ステートメントを使用した表からの行の除去

表からすべての行を除去するときは、TRUNCATE ステートメントを使用します。

TRUNCATE ステートメントの形式は次のとおりです。

```
TRUNCATE TABLE table-name
```

これは、以下の DELETE ステートメントと同等です。

```
DELETE FROM table-name
```

TRUNCATE ステートメントには、切り捨て操作時にトリガーを処理する方法および切り捨ての完了後に開ける表の ID 列の動作を制御するための、DELETE ステートメントでは使用できない追加のオプションがいくつか用意されています。

TRUNCATE のデフォルトでは、切り捨て時に削除トリガーは活動化されません。削除トリガーを活動化する場合は、DELETE ステートメントを使用する必要があります。

ID 列について、切り捨て操作が行われなかった場合と同様に ID 値の生成を続行することを指定するか、最初に定義された初期値から ID 列が開始するように要求することができます。デフォルトでは、値の生成を続行します。

- | 注: TRUNCATE ステートメントは、SQLCA の SQLERRD(3) で削除された行の数および GET
- | DIAGNOSTICS ステートメントの ROW_COUNT 診断項目を返しません。
- | 関連概念:
- | 123 ページの『DELETE ステートメントを使用した表からの行の除去』
- | 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。
- | 関連資料:
- | TRUNCATE

データのマージ

MERGE ステートメントを使用して、表またはビューの行を条件に基づいて挿入、更新、または削除します。

MERGE ステートメントを使用すると、ターゲット表を別の表、派生表、または他のいずれかの表参照に基づいて更新できます。この別の表のことを、ソース表と呼びます。最も簡単な形式のソース表は、値のリストです。

MERGE ステートメントは、ソース表の行がターゲット表に存在するかどうかに基づいて、新規に行を挿入したり、一致する行を更新または削除したりできます。

MERGE ステートメントの最も基本的な形式は、行がまだ存在しない場合にはそれを新規に挿入し、行が存在する場合にはそれを更新するというものです。SQLCODE または SQLSTATE に基づいて挿入または更新を試行した後に他のオプションを試行する代わりに、MERGE ステートメントを使用すると適切な操作が実行されます。

この例では、ある部門において 1 行を追加または置換します。その部門がまだ存在しない場合は、行が挿入されます。その部門が存在する場合、部門の情報が更新されます。

```
MERGE INTO DEPARTMENT USING
  (VALUES ('K22', 'BRANCH OFFICE K2', 'E01')) INSROW (DEPTNO, DEPTNAME, ADMRDEPT)
ON DEPARTMENT.DEPTNO = INSROW.DEPTNO
WHEN NOT MATCHED THEN
  INSERT VALUES(INSROW.DEPTNO, INSROW.DEPTNAME, INSROW.ADMRDEPT)
WHEN MATCHED THEN
  UPDATE SET DEPTNAME = INSROW.DEPTNAME, ADMRDEPT = INSROW.ASMRDEPT
```

EMP_PHOTO 表のコピーである一時表があると想定します。この表で、新しい従業員の写真情報を追加し、既存の従業員の写真を更新しました。一時表には変更内容だけが含まれていて、未変更の写真情報の行はありません。

これらの更新をマスター EMP_PHOTO 表にマージするには、以下の MERGE ステートメントを使用できます。

```
MERGE INTO EMP_PHOTO target USING TEMP_EMP_PHOTO source
  ON target.EMPNO = source.EMPNO
    AND target.PHOTO_FORMAT = source.PHOTO_FORMAT
WHEN NOT MATCHED THEN
  INSERT VALUES(EMPNO, PHOTO_FORMAT, PICTURE)
WHEN MATCHED THEN
  UPDATE SET PICTURE = source.PICTURE
```

このステートメントを実行すると、ターゲット表の行は EMPNO および PHOTO_FORMAT 列を使用してソース表の行と比較されます。これらは表の主キーを形成する列であるので、固有性が保証されます。ソース表にある行のうち、一致する EMPNO および PHOTO_FORMAT 行がターゲット表に存在しないもの (NOT MATCHED) については、INSERT 操作が実行されます。この場合、ソース表の

EMPNO、PHOTO_FORMAT、および PICTURE の各値が入った新しい行がターゲット表に挿入されます。ソース表にある行のうち、対応する行がターゲット表に既に存在するもの (MATCHED) については、ターゲット表の行がソース表の PICTURE 値で更新されます。

マージを少し複雑にするために、1 列を EMP_PHOTO 表に追加することにします。

```
ALTER TABLE EMP_PHOTO ADD COLUMN LAST_CHANGED TIMESTAMP
GENERATED BY DEFAULT FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP
```

ここで、TEMP_EMP_PHOTO 表を保守するユーザーが、EMP_PHOTO 表のマスター・コピーにマージ済みのいくつかの行を一時表の中に既に持っていると想定します。MERGE を行う際に、値が変更されていないので、同じ行を再度更新しないようにと考えています。また、他のユーザーがさらに新しい写真を使用してマスター EMP_PHOTO を更新した可能性もあります。

```
MERGE INTO EMP_PHOTO target USING TEMP_EMP_PHOTO source
ON target.EMPNO = source.EMPNO
AND target.PHOTO_FORMAT = source.PHOTO_FORMAT
WHEN NOT MATCHED THEN
INSERT VALUES(EMPNO, PHOTO_FORMAT, PICTURE, LAST_CHANGED)
WHEN MATCHED AND target.LAST_CHANGED < source.LAST_CHANGED THEN
UPDATE SET PICTURE = source.PICTURE,
LAST_CHANGED = source.LAST_CHANGED
```

このステートメントでは、MATCHED 節に追加の条件が追加されています。LAST_CHANGED 列の比較を追加することにより、現在のマスターのタイム・スタンプより古いタイム・スタンプを持つ写真によってマスター EMP_PHOTO 表が更新されることを防止します。

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:

63 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。

76 ページの『複雑な検索条件の定義』

検索条件には、= や > などの基本的な比較述部に加え、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 SELECT ステートメント中の検索条件をさらに絞り込みます。

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。例えば、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て、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.EMPPROJECT
   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.EMPPROJECT
 WHERE PROJNO= 'MA2100')
```

この結果は、以下のような中間結果表になります。

CORPDATA.EMPPROJECT 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 ステートメント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	LUCCHESI	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。例えば、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。例えば、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となります。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:

ここでは、副照会の使用による検索条件の絞り込みに関するいくつかの考慮事項について取り上げます。

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、以下のようになります。

- 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・マーカール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・マーカールに結果のデータ・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 または HAVING 文節に副照会を組み込む:

基本比較、限量化比較、IN キーワード、または EXISTS キーワードを使用して、WHERE または HAVING 文節に副照会を組み込むことができます。

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。例えば、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。例えば、「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) 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。例えば、「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) 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できます。例えば、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。例えば、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、副照会自体が関連になり、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 ...)
```

関連副照会

関連副照会とは、外部レベル SELECT ステートメントで新しい行の検査 (WHERE 文節) または行グループの検査 (HAVING 文節) に移る際に、SQL が評価し直すような副照会のことです。

関連名と関連参照:

関連参照は、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 は関連名であり、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、関連名を定義することができます。関連名は、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関連名を指定すれば、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。

関連名は、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。例えば、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 で使用される関連名は、B、A、または外部レベル SELECT で定義することができます。関連名を定義するには、表名の後に関連名を入れます。表名とその関連名との間に 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、その関連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関連名 TA と TB を定義していますが、表 TABLEC に対しては関連名を定義していません。

```
FROM TABLEA TA, TABLEC, TABLEB TB
```

関連参照は副照会にいくつでも置くことができます。例えば、ある検索条件の 1 つの関連名は外部レベル SELECT で定義し、別の関連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関連参照に代入されます。

例: WHERE 文節の関連副照会:

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。例えば、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。例えば、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。例えば、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 34. 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	LUCCHESI	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

表 34. 前述の照会の結果のセット (続き)

EMPNO	LASTNAME	WORKDEPT	EDLEVEL
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。例えば、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。例えば、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となります。次の SQL ステートメントを実行する。

```

SELECT DEPTNO, DEPTNAME,
       (SELECT FIRSTNAME 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 35. 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。

例えば、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。

1 まず、以下のように、表を変更して PRIORITY 列を追加します。

```
1 ALTER TABLE CORPDATA.PROJECT ADD COLUMN PRIORITY INT
```

次に、以下の照会を実行して更新を実行します。

```

UPDATE CORPDATA.PROJECT X
  SET PRIORITY = 1
  WHERE '1983-09-01' >
        (SELECT MAX(EMENDATE)
         FROM CORPDATA.EMPPROJECT
         WHERE PROJNO = X.PROJNO)

```

SQL は、CORPDATA.EMPPROJECT 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よ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。これらの表は、CORPDATA サンプル・データベースに存在しません。


```

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.EMPPROJECT 表内の行も削除しなければなりません。そのためには、以下のステートメントを実行します。

```

DELETE FROM CORPDATA.EMPPROJECT 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:

52 ページの『ビューの作成と使用』

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。

57 ページの『索引の作成』

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。

62 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。

65 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。

67 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。

68 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。

75 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。

76 ページの『複雑な検索条件の定義』

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

102 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。

照合順序

ORDER BY および行選択で使用される分類順序

この例は、使用される分類順序において、行が順序付けられて選択される方法を示しています。

JOB 列の値は、大/小文字混合です。'Mgr'、'MGR'、および 'mgr' などの値があります。

表 36. STAFF 表

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	0
20	Pernal	20	Sales	8	18171.25	612.45
30	Merenghi	38	MGR	5	17506.75	0
40	OBrien	38	Sales	6	18006.00	846.55
50	Hanes	15	Mgr	10	20659.80	0
60	Quigley	38	SALES	0	16808.30	650.25
70	Rothman	15	Sales	7	16502.83	1152.00
80	James	20	Clerk	0	13504.60	128.20
90	Koonitz	42	sales	6	18001.75	1386.70
100	Plotz	42	mgr	6	18352.80	0

以下の例では、次の分類順序を使用する各ステートメントの結果を示します。

- *HEX 分類順序
- 言語 ID ENU を使用する共用重み分類順序
- 言語 ID ENU を使用する固有重み分類順序

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

分類順序は、ORDER BY 文節によって行われる順序付けに影響します。

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 37. *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 38. 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 39. 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

分類順序と行選択

分類順序は行の選択に影響します。

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 40. *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 41. 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 42. 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 43. "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 44. "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。

関連概念:

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る表が、ある分類順序を用いて定義される場合、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

International Components for Unicode (ICU) 分類順序表が使用される場合、表のロケールに基づいてデータの重みを決定するため、データベースは言語固有の規則を使用します。

en_us (米国英語ロケール) と指定された ICU 分類順序表は、例えば 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。

ICU サポート (IBM i オプション 39) 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。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。

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

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 分類順序表は次のように 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 SQL ステートメントのパフォーマンス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、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、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、ICU サポートを呼び出す必要はありません。

関連概念:

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。例えば、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・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、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、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。

関連タスク:

QAQQINI による照会の動的な制御

データ保護

DB2 for i データベースは、SQL データを権限のないユーザーから保護し、データ保全性を確保するさまざまな方法を提供します。

SQL オブジェクトの機密保護

SQL オブジェクトも含めて、システム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。

- 1 ユーザーは、SQL GRANT ステートメントと REVOKE ステートメント、TRANSFER OWNERSHIP ステートメント、または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) CL コマンドのいずれかにより 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 関数、SQL パッケージ、SQL プロシージャ、特殊タイプ、配列型、シーケンス、変数、表、ビュー、XSR オブジェクト、および表とビュー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。

- TRANSFER OWNERSHIP ステートメントは、表、索引、またはビューでオブジェクトの所有権を現行所有者から新規所有者に移すために使用できます。

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せん。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。

関連資料:

機密保護解説書

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ユーザー・プロファイル・オブジェクトです。ユーザー・プロファイル作成 (CRTUSRPRF) コマンドを使用して、権限 ID を作成することができます。

ビュー

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 (例えば、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 (例えば、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。

1 列マスクおよび行許可

1 表に対して列マスクおよび行許可を定義して、特定のユーザーが表示できるデータを制限できます。

1 列マスクは、列に対して返す代替値を定義できます。マスクの定義ではロジックを使用して、データを照会しているアプリケーションまたはユーザー・タイプを検査してデータを表示する方法を決定できます。マスク値の例として、桁の多くに XXX が含まれたクレジット・カード番号が挙げられます。

1 行許可は、照会から使用可能な行を制限します。マスクと同様に、制限を定義するロジックがあります。行許可の例としては、管理職は、所属する部門の従業員に関する情報のみを表示できるが、人事部門のスタッフはすべての従業員を表示できるようにする場合があります。

1 マスクおよび許可のロジックで VERIFY_GROUP_FOR_USER スカラー関数を使用して、異なるカテゴリーのユーザーを区別できます。

1 関連資料:

1 CREATE MASK

1 CREATE PERMISSION

1 VERIFY_GROUP_FOR_USER

1 行および列のアクセス制御

監査

DB2 for i データベースは、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。C2 レベルの主な機能は、システム上の監査を実行できることです。

DB2 for i 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) 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 (例えば、権限の失敗や、作成、削除、認可、および取り消し操作など) を監査するかを制御します。

DB2 for i は、DB2 for i 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。例えば、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。

関連資料:

機密保護解説書

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。

関連資料:

XA API

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 for i 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 for i 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 for i 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。

例えば、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 for i 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、データ操作言語 (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。

すべてのデータ定義言語 (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場合は、デッドロック検出が備えられます。例えば、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ため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、DB2 for i 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こと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ことができます。

パフォーマンスを向上させるために、DB2 for i 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 for i 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 for i 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。

注: 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ます。

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 (例えば、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 (例えば、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。一時的な結果表の作成時に関する情報については、SQL 解説書のトピック集の DECLARE CURSOR を参照してください。
- 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。

並行アクセス解決オプションを使用すると、トランザクションの待ち時間を最小化できます。このオプションは、特定の分離レベルでのレコード・ロック競合の処理方法についてデータベース・マネージャーに指示するものです。

並行アクセス解決オプションには、次のいずれかの値を設定できます。

結果を待つ

これはデフォルトです。更新または削除対象として処理中のロック・データが検出されたときにコミットまたはロールバックを待機するように、この値でデータベース・マネージャーに指示します。挿入対象として処理中のロックされた行は、スキップされません。このオプションは、COMMIT(*NONE) または COMMIT(*CHG) の設定で実行されている読み取り専用の照会には適用されません。

現在コミット済みを使用

この値を使用すると、更新または削除対象として処理中のロック・データが検出されたとき、読み取り専用の照会においては現在コミットされたデータのバージョンをデータベース・マネージャーで使用できます。挿入対象として処理中のロックされた行は、スキップできます。このオプションは、COMMIT(*CS) の設定で実行されているときに可能であれば適用され、それ以外の設定の場合には無視されます。

ロック・データをスキップ

レコード・ロック競合が発生した場合に行をスキップするように、この値でデータベース・マネージャーに指示します。このオプションは、照会が COMMIT(*CS) または COMMIT(*ALL) の設定で実行されているときにのみ適用されます。

並行アクセス解決値の「現在コミット済みを使用」および「ロック・データをスキップ」を使用すると、ロック待機を回避することにより並行性を改善できます。ただし、これらのオプションを使用すると、アプリケーションの機能に影響を与えることがあるので注意が必要です。

以下のようにして、並行アクセス解決値を指定できます。

- SELECT ステートメント、SELECT INTO、検索された UPDATE、または検索された DELETE のステートメント・レベルの *concurrent-access-resolution* 節によって。
- CRTSQLxxx コマンドまたは RUNSQLSTM コマンドで CONACC キーワードを使用することによって。
- SET OPTION ステートメントの CONACC 値によって。
- PREPARE ステートメントの *attribute-string* で。

並行アクセス解決オプションがアプリケーションによって直接設定されない場合は、QAQQINI 照会オプション・ファイルの SQL_CONCURRENT_ACCESS_RESOLUTION オプションの値が使用されます。

関連資料:

LOCK TABLE

ジャーナル処理

DB2 for i 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 for i 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)。ライブラリー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。例えば、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 スキーマ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善性は得られません。

関連概念:

ジャーナル管理

関連資料:

120 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。

コミットメント制御

DB2 for i コミットメント制御サポートは、更新、挿入、削除操作またはデータ定義言語 (DDL) 操作などの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。例えば、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。例えば、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 for i 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 for i は、STRCMTCTL コマンドで、LCKLVL パラメーターと一緒に 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 パラメーターは、SQL の作成 (CRTSQLxxx)、SQL 対話式セッションの開始 (STRSQL)、または SQL ステートメント実行 (RUNSQLSTM) 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 パラメーターは、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 パラメーターを指定できます。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 for i 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:

- 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 (DML) 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。
- 指定した LCKLVL パラメーターは単にデフォルトのロック・レベルに過ぎません。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。また、LCKLVL パラメーターは、IBM i の従来のシステム・インター

フェース (非 SQL) によって要求されるコミットメント制御操作のみに適用されます。LCKLVL パラメーター上で指定されるロック・レベルは、それ以降の、SET TRANSACTION ステートメントなどを使用して加えられる SQL 分離レベルへの変更の影響を受けません。

集約関数、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 for i 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・プロセスの処理 (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 for i 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、該当の表での並行プロセスの処理 (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(*RR) を使用する競合性の高い環境では、アプリケーションは、SQL0913 を取得してデータベース・アンロック・メカニズム時間が作動できるようにした後で、操作を再試行する必要があることがあります。

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 for i 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せん。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まれます。(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。

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 ステートメント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、IBM i プラットフォー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、コミット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、IBM i プラットフォー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 45. 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 SET 変数 VALUES INTO	*NONE *CHG *CS (注 6 を参照) *ALL (注 2 および注 7 を参照)	ロックなし。 ロックなし。 読み取りおよび解放時に行がロックされる。 読み取りから次の ROLLBACK または COMMIT まで。	READ READ
FETCH (読み取り専用カーソル)	*NONE *CHG *CS (注 6 を参照) *ALL (注 2 および注 7 を参照)	ロックなし。 ロックなし。 読み取りから次の FETCH まで。 読み取りから次の ROLLBACK または COMMIT まで。	READ READ
FETCH (更新または削除が可能なカーソル) (注 1 を参照)	*NONE *CHG *CS *ALL	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。 行が更新される場合は、読み取りから次の FETCH まで。 行が削除される場合は、読み取りから次の DELETE まで。 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。 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。 読み取りから次の ROLLBACK または COMMIT まで。	UPDATE UPDATE UPDATE UPDATE
INSERT (目標表) MERGE, INSERT sub-statement	*NONE *CHG *CS *ALL	ロックなし。 挿入から次の ROLLBACK または COMMIT まで。 挿入から次の ROLLBACK または COMMIT まで。 挿入から次の ROLLBACK または COMMIT まで。	UPDATE UPDATE UPDATE ³
INSERT (部分選択の表)	*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

表 45. 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
UPDATE (カーソルなし)	*NONE *CHG *CS *ALL	各行は更新の期間ロックされる。 読み取りから次の ROLLBACK または COMMIT まで。 読み取りから次の ROLLBACK または COMMIT まで。 読み取りから次の ROLLBACK または COMMIT まで。	UPDATE UPDATE UPDATE UPDATE
DELETE (カーソルなし)	*NONE *CHG *CS *ALL	各行は削除の期間ロックされる。 読み取りから次の ROLLBACK または COMMIT まで。 読み取りから次の ROLLBACK または COMMIT まで。 読み取りから次の ROLLBACK または COMMIT まで。	UPDATE UPDATE UPDATE UPDATE
UPDATE (カーソルつき) MERGE, UPDATE sub-statement	*NONE *CHG *CS *ALL	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。 読み取りから次の ROLLBACK または COMMIT まで。 読み取りから次の ROLLBACK または COMMIT まで。	UPDATE UPDATE UPDATE UPDATE
DELETE (カーソルつき) MERGE, DELETE sub-statement	*NONE *CHG *CS *ALL	ロックは行の削除時に解放される。 読み取りから次の ROLLBACK または COMMIT まで。 読み取りから次の ROLLBACK または COMMIT まで。 読み取りから次の ROLLBACK または COMMIT まで。	UPDATE UPDATE UPDATE UPDATE
副照会 (更新または削除可能カーソルまたは UPDATE または DELETE カーソルなし)	*NONE *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:

- 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)
- 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ないように排他的ロックが取得されます。
- 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。
- 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。
- 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。
- 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。
- 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。

関連概念:

コミットメント制御

関連資料:

DECLARE CURSOR

分離レベル

XA API

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できます。例えば、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に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、またはストアド・プロシージャ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じの保管ポイント・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ド・プロシージャが呼び出された時	ストアド・プロシージャ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 (注 2 を参照)
- LABEL (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. 複数の alter table オプションを指定した場合、操作は一度に 1 つずつ処理されるため、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とおりです。
 - 制約の除去。
 - マテリアライズ照会表の除去。
 - 区画の除去。
 - 区画化の除去。
 - 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)。
 - 区画の変更。
 - マテリアライズ照会表の追加および変更。
 - 区画または区画化の追加。
 - 制約の追加。
2. COMMENT ステートメントまたは LABEL 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。それぞれの列またはオブジェクトに対する COMMENT または LABEL 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 FUNCTION
- | • ALTER MASK
- | • ALTER PERMISSION
- ALTER PROCEDURE
- ALTER SEQUENCE
- | • ALTER TRIGGER
- CREATE ALIAS
- CREATE TYPE
- CREATE FUNCTION
- CREATE INDEX
- | • CREATE MASK
- | • CREATE PERMISSION
- CREATE PROCEDURE
- CREATE SCHEMA
- CREATE SEQUENCE
- CREATE TABLE
- CREATE TRIGGER
- CREATE VARIABLE
- CREATE VIEW
- DECLARE GLOBAL TEMPORARY TABLE
- DROP ALIAS
- DROP FUNCTION
- | • DROP MASK
- | • DROP PERMISSION
- DROP PROCEDURE
- DROP SCHEMA
- DROP SEQUENCE
- DROP TRIGGER
- DROP TYPE
- DROP VARIABLE
- RENAME (注 1 を参照)

注: 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。

例えば、CREATE TABLE ステートメントは、DB2 for i 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 for i データベースは、固有限制、参照制約、および検査制約をサポートします。

固有限制は、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 for i は、すべてのデータ操作言語 (DML) 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 for i 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。

- 固有限制は、索引と一緒に組み込まれます。固有限制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 for i 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:

12 ページの『制約』

制約とは、データベース・マネージャーによって実施され、表内で挿入、削除、または更新できる値を制限する規則のことです。

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、挿入および更新操作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。

関連資料:

ALTER TABLE

CREATE TABLE

保管および復元機能

IBM i 保管および復元機能は、SQL オブジェクトをディスク (保管ファイル) に、または外部メディアに保管する場合に使用します。

保管したバージョンは、あとで任意の IBM i オペレーティング・システムに復元することができます。保管および復元機能を使用すると、スキーマ全体、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。この機能を使用すると、表またはスキーマ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、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、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。

グローバル変数の *SRVPGM が復元されると、その変数は SYSVARIABLES カタログに自動的に追加されます。カタログが正常に更新されない場合、そのオブジェクトはグローバル変数として使用できません。その名前がスキーマ内で重複してしまうような状況では、オブジェクトを別のスキーマに移動することにより、それをグローバル変数として認識させることが可能な場合があります。

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・ビュー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・ビューは、すべてのオブジェクトを適切に反映します。

耐損傷性

DB2 for i データベースには、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。

例えば、ミラーリング、チェックサム、および新磁気ディスク制御機構 (RAID) はすべて、ディスク問題の可能性を軽減します。DB2 for i 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、少なくとも表、ビュー、SQL パッケージ、索引、プロシージャ、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 for i 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 for i データベースには、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

アクセス・パスの回復編集 (EDTRCYAP) CL コマンドを使用すると、DB2 for i 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 for i 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 for i 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い場合は、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 IBM i 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。

カタログの健全性

カタログ内の情報が常に正確であることを保証するために、DB2 for i データベースは、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し、カタログの中で記述されている SQL オブジェクトに対して変更が行われた場合に情報を暗黙に維持管理します。

カタログの健全性は、SQL ステートメント、IBM i CL コマンド、システム/38 環境の CL コマンド、システム/36 環境の機能、または IBM i プラットフォー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。例えば、SQL DROP ステートメントを実行すること、IBM i ファイル削除 (DLTF) CL コマンドを実行すること、システム/38 ファイル削除 (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、表を削除できます。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、表が削除されたときにカタログから表の記述を取り除きます。以下の表には、さまざまな機能と、カタログに作用する関連した効果がリストされています。

表 46. さまざまな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
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 文節を CREATE SCHEMA ステートメントで使用して、スキーマをユーザーの補助記憶域プール (ASP) に作成できます。

ライブラリー作成 (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。

関連概念:

バックアップおよび回復

ディスク管理

独立補助記憶域プール

独立ディスク・プールは、システム上でユーザー・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。1 次独立ディスク・プールは、データベースをセットアップするために使用されます。

複数のデータベースを処理できます。システムは、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システムに組み込まれ、System i 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・プールのセットアップが終わると、System i ナビゲーターの「データベース」フォルダーに別のデータベースとして表示されます。

関連概念:

ディスク管理

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。

ストアド・プロシージャ

プロシージャ (しばしば、ストアド・プロシージャと呼ばれる) 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す。プロシージャには、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めることができます。SQL のプロシージャの場合も、ホスト言語のプロシージャの場合と同じ利点があります。

DB2 のストアド・プロシージャ・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を定義し、呼び出すための手段を提供します。ストアド・プロシージャは、分散型と非分散型の両方の DB2 アプリケーションで使用することができます。ストアド・プロシージャを使用することの利点の 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 回の処理で、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。

プロシージャは、SQL プロシージャまたは外部プロシージャとして定義することができます。外部プロシージャとしては、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を除く) または REXX プロシージャが可能です。このプロシージャは、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。

ストアド・プロシージャ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ド・プロシージャの定義
- CALL ステートメントによるストアド・プロシージャの呼び出し
- パラメーターの引き渡し規則
- プロシージャを呼び出しているプログラムに完了状況に戻す方法

ストアド・プロシージャは、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。

外部プロシージャまたは SQL プロシージャを作成するには、SQL CREATE PROCEDURE ステートメントを使用できます。

以下の節では、ストアド・プロシージャの定義と呼び出しに使用される SQL ステートメント、ストアド・プロシージャにパラメーターを渡す方法、およびストアド・プロシージャの使用例について説明します。

ストアド・プロシージャに関する詳細については、ストアド・プロシージャ、トリガー、およびユ

ーザー定義関数 DB2 Universal Database™ for iSeries®  を参照してください。

関連概念:

13 ページの『ストアド・プロシージャ』

ストアド・プロシージャとは、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。

Java SQL ルーチン

関連資料:

350 ページの『DRDA ストアド・プロシージャに関する考慮事項』

IBM i 分散リレーショナル・データベース・アーキテクチャー (DRDA) サーバーは、ストアド・プロシージャ中の 1 つ以上の結果セットを返す機能をサポートします。

CREATE PROCEDURE

外部プロシージャの定義

外部プロシージャの CREATE PROCEDURE ステートメントは、プロシージャに名前を付け、パラメーターと属性を定義し、システムがプロシージャを呼び出す際、システムが使用するプロシージャ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との間で受け渡されるときが決まります。
- プロシージャ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 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に渡される方法にも影響します (例えば、ILE C プロシージャの場合、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。例えば、NULL 標識引数を渡すことはできません。

関連資料:

171 ページの『プロシージャ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 プロシージャの定義

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

プロシージャ本体は、プロシージャ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。
- プロシージャが SQL データを変更する SQL プロシージャであることを示す。
- プロシージャ本体を単一の 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、次のステートメントから構成されます。

- 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。
- プロシージャが SQL データを変更する SQL プロシージャであることを示す。
- プロシージャ本体を定義する。このプロシージャが呼び出されると、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。
- プロシージャが SQL データを読み取る SQL プロシージャであることを示す。
- プロシージャ本体を定義する。
 - 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 FIRSTNAME, MIDINIT, LASTNAME, BONUS
    FROM CORPDATA.EMPLOYEE
    WHERE WORKDEPT = CREATE_BONUS_TABLE.DEPT_NUMBER
  DO
    IF BONUS > 0 THEN
      SET NAME = FIRSTNAME 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 FIRSTNAME、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。例えば、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 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,
      FIRSTNAME 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, FIRSTNAME, 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。
- プロシージャが SQL プロシージャであることを示す。
- プロシージャ本体を定義する。
 - 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。

デフォルト・パラメーターを使用したプロシージャの定義

- | 外部プロシージャおよび SQL プロシージャは、オプション・パラメーターを使用して作成できます。
- | プロシージャのオプション・パラメーターは、デフォルト値を持つように定義されます。

| 以下のようなパラメーターを使用して定義されたプロシージャがあるものとします。これが SQL プロシ
 | ージャの一部であるのか、外部プロシージャ・ステートメントの一部であるのかは重要ではありません
 | ん。ここでは、パラメーター定義についてのみ説明します。

```

|      CREATE PROCEDURE UPDATE_EMPLOYEE_INFO
|      (IN EMPLOYEE_NUMBER CHAR(10),
|      IN EMP_DEPT CHAR(3),
|      IN PHONE_NUMBER CHAR(4)
|      . . .

```

| このプロシージャは長期間使用されており、多くの場所から呼び出されているものとします。今、あるユ
 | ーザーが、同じ表内のいくつかの別の列 (JOB および EDLEVEL) をこのプロシージャで更新するよう
 | すれば便利であると提案したとします。このプロシージャに対するすべての呼び出しを検出して変更する
 | のは多大な作業ですが、デフォルト値を持つ新規パラメーターを追加すれば非常に簡単になります。

| 以下の CREATE PROCEDURE ステートメントのパラメーター定義では、従業員番号を除くすべての列を
 | オプションとして渡すことができます。

```

|      CREATE OR REPLACE PROCEDURE UPDATE_EMPLOYEE_INFO
|      (IN EMPLOYEE_NUMBER CHAR(10),
|      IN EMP_DEPT CHAR(3) DEFAULT NULL,

```

```

|         IN PHONE_NUMBER CHAR(4) DEFAULT NULL,
|         IN JOB CHAR(8) DEFAULT NULL,
|         IN EDLEVEL SMALLINT DEFAULT NULL)
|         . . .

```

このプロシージャのコードは、SQL ルーチンであっても外部プログラムであっても、新規パラメーターを処理し、ヌル値が渡されたときに既存の 2 つのパラメーターを正しく処理するように変更する必要があります。デフォルト・パラメーターはオプションであるため、このプロシージャに対する既存の呼び出しを変更する必要はありません。つまり、2 つの新規パラメーターはヌルの値をプロシージャ・コードに渡します。新規パラメーターを必要とする呼び出し元は、SQL CALL ステートメントでそれらを含めることができます。

この例ではすべてのデフォルト値にヌルを使用していますが、ほぼすべての式を使用できます。単純な定数や複雑な照会にすることができます。他のパラメーターを参照することはできません。

CALL ステートメントでデフォルト値を使用する方法はいくつかあります。

- 使用する必要がないパラメーターを末尾で省略します。

```

| CALL UPDATE_EMPLOYEE_INFO('123456', 'D11', '4424')

```

JOB パラメーターおよび EDLEVEL パラメーターにはデフォルト値が使用されます。

- 省略するパラメーターに対して DEFAULT キーワードを使用します。

```

| CALL UPDATE_EMPLOYEE_INFO('123456', DEFAULT, '4424', DEFAULT, DEFAULT)

```

このステートメントでは、すべてのパラメーターが表されています。EMP_DEPT、JOB、および EDLEVEL の各パラメーターでデフォルト値が使用されます。

- 対応するパラメーター名を使用して引数のいくつかを指定し、使用しないパラメーターを省略します。

```

| CALL UPDATE_EMPLOYEE_INFO('123456', EDLEVEL => 18)

```

この CALL ステートメントでは、パラメーター名を使用することで、他の 3 つのパラメーターを表す必要がありません。EMP_DEPT、PHONE_NUMBER、および JOB の各パラメーターでデフォルト値が使用されます。

CALL ステートメントでは、名前付き引数は任意の順序で配置できます。名前が指定されていない引数は、プロシージャのパラメーター定義の順序に一致している必要があります。名前付き引数よりも前に指定する必要があります。ステートメントで名前付き引数を使用されると、それ以降のすべての引数にも名前を指定する必要があります。CALL ステートメントで引数が指定されていないパラメーターには、デフォルト値が定義されている必要があります。

以下の例では、プロシージャは、特定の部門のすべての従業員が含まれている表を作成します。それが作成されるスキーマは、それが使用される環境用に常にハードコーディングされています。ただし、テストのためには、テスト・スキーマで表を作成すると便利です。

```

| CREATE OR REPLACE PROCEDURE CREATE_DEPT_TABLE2
|         (IN P_DEPT CHAR(3),
|         IN SCHEMA_NAME VARCHAR(128) DEFAULT 'CORPDATA')
| BEGIN
|     DECLARE DYNAMIC_STMT VARCHAR(1000);
|     DECLARE MESSAGE CHAR(20);
|     DECLARE TABLE_NAME VARCHAR(200);
|     DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
|         SET MESSAGE = 'ok';
|
|     SET TABLE_NAME = ''' CONCAT SCHEMA_NAME CONCAT
|         '''.DEPT_' CONCAT P_DEPT CONCAT '_T';

```

```

|
| SET DYNAMIC_STMT = 'DROP TABLE ' CONCAT TABLE_NAME;
| EXECUTE IMMEDIATE DYNAMIC_STMT;
|
| SET DYNAMIC_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))';
| EXECUTE IMMEDIATE DYNAMIC_STMT;
|
| SET DYNAMIC_STMT = 'INSERT INTO ' CONCAT TABLE_NAME CONCAT
| 'SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY
| FROM CORPDATA.EMPLOYEE
| WHERE WORKDEPT = ?';
| PREPARE INSERT_INTO_DEPARTMENT_STMT FROM DYNAMIC_STMT;
| EXECUTE INSERT_INTO_DEPARTMENT_STMT USING P_DEPT;
|
| END;

```

2 番目のパラメーターは、スキーマ名を渡すように定義されています。そのデフォルト値は「CORPDATA」です。これは、過去にプロシージャで使用されていた値です。

実稼働環境で実行する場合、CALL ステートメントは以下のようにする可以考虑されます。

```
CALL CREATE_DEPT_TABLE2('D21')
```

SCHEMA_NAME パラメーターが指定されていないため、デフォルト・パラメーター値が使用されます。表 DEPT_D21 が CORPDATA に作成されます。

テスト環境で実行する場合、CALL ステートメントは以下のようにする可以考虑されます。

```
CALL CREATE_DEPT_TABLE2('D21', 'TESTSCHEMA')
```

これにより、表 DEPT_D21 が指定スキーマ TESTSCHEMA に作成されます。

ストアド・プロシージャの呼び出し

SQL CALL ステートメントは、ストアド・プロシージャを呼び出します。

CALL ステートメントでは、ストアド・プロシージャの名前および任意の引数を指定します。引数には、定数、特殊レジスター、ホスト変数、または式を使用できます。CALL ステートメントで指定する外部ストアド・プロシージャには、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。

DB2 for i データベースではタイプごとにルールが異なるので、以下に CALL ステートメントのタイプについて説明します。

- プロシージャ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 (例えば、Run SQL スクリプトまたは 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 プロシージャへの呼び出し時に、SQL 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 文字を追加します。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

以下の PL/I の例は、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、SQL 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。前述の例では、*SYS (システム命名) の命名オプションが使用されることと、SQL PL/I
プログラム作成 (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。例えば、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、定義されているデータ・タイプおよび長さ
として渡されますが、以下の例外があります。

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 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 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。

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 ステートメントでも (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 または SQL 記述子を渡すことができます。

以下の C の例で、SQLDA 構造を使用した CALL ステートメントの実行を示します。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、398 ページの『コードに関するライセンス情報および特記事項』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* Ssqldap;
...
main()
{
    Ssqldap dap;
    short col1;
    char col2[4];
    int bc;
    dap = (Ssqldap) 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].sqlllen = 2;
    dap->sqlvar[0].sqldata = (char*) &col1;
    dap->sqlvar[0].sqlname.length = 0;
    dap->sqlvar[1].sqltype = NUL_TERM_CHAR;
    dap->sqlvar[1].sqlllen = 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名の代わりに使用することもできます。例えば、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

例: CREATE PROCEDURE が存在しない動的 CALL ステートメント

次の C の例は、動的 CALL ステートメントを示しています。

```
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。

例: CALL ステートメント:

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と、このプロシージャー中のローカル変数でこの引数を受け取る方法が示されています。

例 1: ILE C プログラムから呼び出される ILE C および PL/I プロシージャー:

この例では、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 プログラムを示します。プロシージャー P1 は ILE 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。

P1 および P2 プロシージャーの定義

```
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

```

注: コード例を使用する場合は、398 ページの『コードに関するライセンス情報および特記事項』のご使用条件に同意する必要があります。

P1 および P2 プロシージャの呼び出し

```

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

```

プロシージャ 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                        */
strcpy(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 *****/

```

プロシージャ 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;

```

例 2: ILE C プログラムから呼び出される REXX プロシージャ:

この例では、ILE C プログラムから呼び出される REXX プロシージャを示します。

REXX プロシージャの定義

```

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

```

注: コード例を使用する場合は、398 ページの『コードに関するライセンス情報および特記事項』のご使用条件に同意する必要があります。

REXX プロシージャの呼び出し

```

/*****
/***** START OF SQL C Application *****/

#include <decimal.h>
#include <stdio.h>
#include <string.h>
#include <wchar.h>
/*-----*/
exec sql include sqlca;
exec sql include sqllda;
/* *****/
/* 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'STTUUVVWXXYYZZAA'" 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" 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・プロシージャから戻される結果セット

ストアード・プロシージャは、CALL ステートメントを出すアプリケーションに、出力パラメーターを戻すことに加えて、結果セット（つまり、ストアード・プロシージャで開かれたカーソルに関連した結果表）を戻すことができます。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を呼び出すときに、内部プロシージャ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。

注: COBOL を使用する場合は、COBOL プログラムのセットアップのために、結果セットは自動的に閉じられます。EXIT PROGRAM ステートメントを EXIT PROGRAM AND CONTINUE RUN UNIT に変更してください。そうすると、結果セットが戻されるはずですが。

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。例えば、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。

例 1: 単一の結果セットを戻すストアード・プロシージャの呼び出し:

この例は、結果セットを戻すストアード・プロシージャを呼び出すときに Open Database Connectivity (ODBC) アプリケーションが使用できる 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 プロシージャの場合、ストアード・プロシージャが結果セットを戻している場合に、この文節が必要です。

ストアード・プロシージャの定義:

```
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、398 ページの『コードに関するライセンス情報および特記事項』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,à);
    if (rc == SQL_SUCCESS)
      :

  {
  // Bind result columns based on attributes returned
  //
  rc = SQLBindCol(hstmt,à);
  :
  // 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。

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

```

注: コード例を使用する場合は、398 ページの『コードに関するライセンス情報および特記事項』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 **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 cbcoll,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 rgfMask 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)
{
    sprintf(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,1stnam,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:db2:1p066ab","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
}

```

ストアド・プロシージャから結果セットを受け取るプログラムまたは SQL プロシージャの作成

内容が分かっている一定の数の結果セット、または内容が分からない可変の数の結果セットをストアド・プロシージャから受け取るプログラムを作成できます。

既知の数の結果セットを返す場合の方が作成は簡単ですが、可変の数の結果セットを扱うコードを作成すれば、ストアード・プロシージャが変更された場合にプログラムを大幅に変更せずに済みます。

結果セットを受け取るための基本的な手順は、以下のとおりです。

1. 返される結果セットごとに、ロケータ変数を宣言します。

何個の結果セットが返されるか分からない場合には、返される可能性のある結果セットの最大数に応じて、十分な数の結果セット・ロケータを宣言します。

2. ストアード・プロシージャを呼び出し、SQL 戻りコードを調べます。

CALL ステートメントからの SQLCODE が +466 の場合、ストアード・プロシージャは結果セットを返しています。

3. ストアード・プロシージャが何個の結果セットを返すかを判別します。

ストアード・プロシージャが何個の結果セットを返すか既に知っている場合、この手順は省略できます。

SQL ステートメント DESCRIBE PROCEDURE を使用して、結果セットの数を判別します。DESCRIBE PROCEDURE は、結果セットに関する情報を SQLDA または SQL 記述子に入れます。

SQL 記述子の場合、DESCRIBE PROCEDURE ステートメントが完了すると、以下の値を取り出すことができます。

- DB2_RESULT_SETS_COUNT には、ストアード・プロシージャによって返される結果セットの数が入ります。
- 結果セットごとに、以下に示す 1 つの記述子域の項目が返されます。
 - DB2_CURSOR_NAME には、ストアード・プロシージャが結果セットを返すために使用するカーソルの名前が入ります。
 - DB2_RESULT_SET_ROWS には、結果セット内の行数の推定値が入ります。値の -1 は、結果セット内の行数の推定値を入手できないことを示します。
 - DB2_RESULT_SET_LOCATOR には、結果セットに関連付けられた結果セット・ロケータの値が入ります。

SQLDA の場合、ストアード・プロシージャが返す可能性のある最大個数の結果セットを保持できるように、SQLDA を十分な大きさに設定します。DESCRIBE PROCEDURE ステートメントが完了すると、SQLDA 内のフィールドには以下の値が入ります。

- SQLD には、ストアード・プロシージャによって返される結果セットの数が入ります。
- それぞれの SQLVAR 項目に、結果セットに関する情報が記述されます。SQLVAR 項目には、以下のものが含まれます。
 - SQLNAME フィールドには、ストアード・プロシージャが結果セットを返すために使用するカーソルの名前が入ります。
 - SQLIND フィールドには、結果セット内の行数の推定値が入ります。値の -1 は、結果セット内の行数の推定値を入手できないことを示します。
 - SQLDATA フィールドには、結果セット・ロケータの値 (結果セットのアドレス) が入ります。

4. 結果セット・ロケータを結果セットにリンクします。

SQL ステートメント `ASSOCIATE LOCATORS` を使用して、結果セット・ロケータを結果セットにリンクできます。 `ASSOCIATE LOCATORS` ステートメントは、結果セット・ロケータの変数に値を割り当てます。返される結果セットの数よりもロケータを多く指定した場合、余分のロケータは無視されます。

前に `DESCRIBE PROCEDURE` ステートメントを実行した場合、結果セット・ロケータの値は、SQL 記述子内の `DB2_RESULT_SET_LOCATOR` または `SQLDA` の `SQLDATA` フィールドから取得できます。これらのフィールドの値は、結果セット・ロケータの変数に手でコピーすることもできますし、 `ASSOCIATE LOCATORS` ステートメントを実行してコピーすることもできます。

`ASSOCIATE LOCATORS` または `DESCRIBE PROCEDURE` ステートメントで指定するストアード・プロシージャ名は、結果セットを返す `CALL` ステートメントで既に使用されたプロシージャ名でなければなりません。

5. 結果セットから行を取り出すためのカーソルを割り当てます。

SQL ステートメント `ALLOCATE CURSOR` を使用して、各結果セットをカーソルにリンクさせます。結果セットごとに、1 つの `ALLOCATE CURSOR` ステートメントを実行します。これらのカーソル名は、ストアード・プロシージャにおけるカーソル名と異なっていても構いません。

6. 結果セットの内容を判別します。

結果セットのフォーマットを既に知っている場合、この手順は省略できます。

SQL ステートメント `DESCRIBE CURSOR` を使用して結果セットのフォーマットを判別し、この情報を SQL 記述子または `SQLDA` に入れます。結果セットごとに、結果セット内のすべての列の記述を保持するのに十分な大きさの `SQLDA` が必要です。

`DESCRIBE CURSOR` は、前に `ALLOCATE CURSOR` を実行したカーソルに対してのみ使用できません。

`DESCRIBE CURSOR` を実行した後、結果セットのカーソルが `WITH HOLD` と宣言される場合、SQL 記述子では `DB2_CURSOR_HOLD` を調べることができます。 `SQLDA` では、 `SQLDA` 内のフィールド `SQLDAID` の 8 バイト目の上位ビットが 1 に設定されます。

7. `ALLOCATE CURSOR` ステートメントで割り当てたカーソルを使用して、結果セットから行を取り出してホスト変数に入れます。

`DESCRIBE CURSOR` ステートメントを実行した場合は、行を取り出す前に以下の手順を実行してください。

- a. ホスト変数および標識変数用にストレージを割り振ります。 `DESCRIBE CURSOR` ステートメント実行後の SQL 記述子または `SQLDA` の内容を使用して、ホスト変数ごとに必要なストレージの大きさを判別します。
- b. 各ホスト変数のストレージのアドレスを、 `SQLDA` の該当する `SQLDATA` フィールドに入れます。
- c. 各標識変数のストレージのアドレスを、 `SQLDA` の該当する `SQLIND` フィールドに入れます。

結果セットからの行の取り出しは、表からの行の取り出しと同じです。

次の例では、これらの各ステップを行う C 言語コードを示します。他の言語のコーディングも類似しています。

次の例では、返される結果セットの数と各結果セットの内容を知っている場合に、結果セットを受け取る方法を示します。

```

/*****
/* Declare result set locators. For this example, */
/* assume you know that two result sets will be returned. */
/* Also, assume that you know the format of each result set. */
/*****
EXEC SQL BEGIN DECLARE SECTION;
    static volatile SQL TYPE IS RESULT_SET_LOCATOR loc1, loc2;
EXEC SQL END DECLARE SECTION;

:
/*****
/* Call stored procedure P1. */
/* Check for SQLCODE +466, which indicates that result sets */
/* were returned. */
/*****
EXEC SQL CALL P1(:parm1, :parm2, ...);
if(SQLCODE==+466)
{
/*****
/* Establish a link between each result set and its */
/* locator using the ASSOCIATE LOCATORS. */
/*****
EXEC SQL ASSOCIATE LOCATORS (:loc1, :loc2) WITH PROCEDURE P1;

:
/*****
/* Associate a cursor with each result set. */
/*****
EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :loc1;
EXEC SQL ALLOCATE C2 CURSOR FOR RESULT SET :loc2;
/*****
/* Fetch the result set rows into host variables. */
/*****
while(SQLCODE==0)
{
EXEC SQL FETCH C1 INTO :order_no, :cust_no;

:
}
while(SQLCODE==0)
{
EXEC SQL FETCH C2 :order_no, :item_no, :quantity;

:
}
}

```

次の例では、返される結果セットの数または各結果セットの内容を知らない場合に、結果セットを受け取る方法を示します。

```

/*****
/* Declare result set locators. For this example, */
/* assume that no more than three result sets will be */
/* returned, so declare three locators. Also, assume */
/* that you do not know the format of the result sets. */
/*****
EXEC SQL BEGIN DECLARE SECTION;
    static volatile SQL TYPE IS RESULT_SET_LOCATOR loc1, loc2, loc3;
EXEC SQL END DECLARE SECTION;

:
/*****
/* Call stored procedure P2. */
/* Check for SQLCODE +466, which indicates that result sets */
/* were returned. */
/*****

```

```

EXEC SQL CALL P2(:parm1, :parm2, ...);
if(SQLCODE==+466)
{
/*****/
/* Determine how many result sets P2 returned, using the */
/* statement DESCRIBE PROCEDURE. :proc_da is an SQLDA */
/* with enough storage to accommodate up to three SQLVAR */
/* entries. */
/*****/
EXEC SQL DESCRIBE PROCEDURE P2 INTO :proc_da;

:
/*****/
/* Now that you know how many result sets were returned, */
/* establish a link between each result set and its */
/* locator using the ASSOCIATE LOCATORS. For this example, */
/* we assume that three result sets are returned. */
/*****/
EXEC SQL ASSOCIATE LOCATORS (:loc1, :loc2, :loc3) WITH PROCEDURE P2;

:
/*****/
/* Associate a cursor with each result set. */
/*****/
EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :loc1;
EXEC SQL ALLOCATE C2 CURSOR FOR RESULT SET :loc2;
EXEC SQL ALLOCATE C3 CURSOR FOR RESULT SET :loc3;

/*****/
/* Use the statement DESCRIBE CURSOR to determine the */
/* format of each result set. */
/*****/
EXEC SQL DESCRIBE CURSOR C1 INTO :res_da1;
EXEC SQL DESCRIBE CURSOR C2 INTO :res_da2;
EXEC SQL DESCRIBE CURSOR C3 INTO :res_da3;

:
/*****/
/* Assign values to the SQLDATA and SQLIND fields of the */
/* SQLDAs that you used in the DESCRIBE CURSOR statements. */
/* These values are the addresses of the host variables and */
/* indicator variables into which DB2 will put result set */
/* rows. */
/*****/

:
/*****/
/* Fetch the result set rows into the storage areas */
/* that the SQLDAs point to. */
/*****/
while(SQLCODE==0)
{
EXEC SQL FETCH C1 USING :res_da1;

:
}
while(SQLCODE==0)
{
EXEC SQL FETCH C2 USING :res_da2;

:
}
while(SQLCODE==0)
{
EXEC SQL FETCH C3 USING :res_da3;
}

```



```
⋮
}
}
```

次の例では、SQL 記述子を使用して結果セットを受け取る方法を示します。

これは、呼び出される SQL プロシージャです。

```
create procedure owntbl()
dynamic result sets 1
begin
  declare c1 cursor for
    select name, dbname from qsys2.systables
      where creator = system_user ;
  open c1 ;
  return ;
end
```

これは、結果セットを処理するプログラムです。

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

EXEC SQL INCLUDE SQLCA;
/*****
/* Declare result set locators. For this example,
/* you know that only one result set will be returned,
/* so only one locator is declared.
*****/
EXEC SQL BEGIN DECLARE SECTION;
static volatile SQL TYPE IS RESULT_SET_LOCATOR loc1;
struct {
  short len;
  char data[128];
} tblName; /* table name */

struct {
  short len;
  char data[128];
} schName; /* schema name */
EXEC SQL END DECLARE SECTION;

void main(int argc, char* argv[])
{
/*****
/* Call the procedure that might return a result set. Check
/* the returned SQLCODE to see if result sets were returned.
*****/
  int noMoreData = 0;
  EXEC SQL WHENEVER SQLERROR GOTO error;
  EXEC SQL CALL OWNTBL ;
  if (SQLCODE != 466) {
    goto error;
  }
/*****
/* Since you know only one result set can be returned from
/* this procedure, associate a locator with the result set
/* and define a cursor to be used with it.
*****/
  EXEC SQL ASSOCIATE LOCATORS (:loc1) WITH PROCEDUREOWNTBL ;
  EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :loc1 ;
/*****
/* Define the descriptor to use for fetching data from the
/* cursor.
*****/
```

```

EXEC SQL ALLOCATE DESCRIPTOR 'desc' WITH MAX 10 ;
EXEC SQL DESCRIBE CURSOR C1 USING SQL DESCRIPTOR 'desc' ;
EXEC SQL WHENEVER NOT FOUND GOTO enddata;
while ( noMoreData == 0 ) {
    EXEC SQL FETCH C1 INTO SQL DESCRIPTOR 'desc' ;
    memset(tblName.data,0x00,sizeof(tblName.data));
    memset(schName.data,0x00,sizeof(schName.data));
    EXEC SQL GET DESCRIPTOR 'desc' VALUE 1 :tblName = DATA;
    EXEC SQL GET DESCRIPTOR 'desc' VALUE 2 :schName = DATA;
    printf("Table: %s Schema: %s %n",
        tblName.data,schName.data);
}

enddata:
printf("All rows fetched.%n");
return;

error:
printf("Unexpected error, SQLCODE = %d %n", SQLCODE);
return;
}

```

次の例では、SQL プロシージャを使用して結果セットを受け取る方法を示します。これは、大きな SQL プロシージャの断片に過ぎません。

```

DECLARE RESULT1 RESULT_SET_LOCATOR VARYING;
DECLARE RESULT2 RESULT_SET_LOCATOR VARYING;
:
:
CALL TARGETPROCEDURE();

ASSOCIATE RESULT SET LOCATORS(RESULT1,RESULT2)
WITH PROCEDURE TARGETPROCEDURE;
ALLOCATE RSCUR1 CURSOR FOR RESULT1;
ALLOCATE RSCUR2 CURSOR FOR RESULT2;

WHILE AT_END = 0 DO
    FETCH RSCUR1 INTO VAR1;
    SET TOTAL1 = TOTAL1 + VAR1;
END WHILE;

WHILE AT_END = 0 DO
    FETCH RSCUR2 INTO VAR2;
    SET TOTAL2 = TOTAL2 + VAR2;
END WHILE;
:
:

```

ストアド・プロシージャおよびユーザー定義関数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。SQL データ・タイプは、各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 for i 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 47. 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 および ILE COBOL
SMALLINT	short		PIC S9(4) BINARY
INTEGER	long		PIC S9(9) BINARY
BIGINT	long long		PIC S9(18) BINARY 注: ILE COBOL でのみサポートされます。
DECIMAL (p,s)	decimal(p,s)	TYPE(*DEC) LEN(p s)	PIC S9(p-s)V9(s) PACKED-DECIMAL 注: 精度が 18 を超えてはなりません。
NUMERIC(p,s)			PIC S9(p-s)V9(s) DISPLAY SIGN LEADING SEPARATE 注: 精度が 18 を超えてはなりません。
DECFLOAT	_Decimal32, _Decimal64, _Decimal128 注: C でのみサポートされます。		
REAL または FLOAT(p)	float		COMP-1 注: ILE COBOL でのみサポートされます。
DOUBLE PRECISION または FLOAT または FLOAT(p)	double		COMP-2 注: ILE COBOL でのみサポートされます。
CHARACTER(n)	char ... [n+1]	TYPE(*CHAR) LEN(n)	PIC X(n)
VARCHAR(n)	char ... [n+1]		可変長文字ストリング
VARCHAR(n) FOR BIT DATA	VARCHAR 構造化フォーム		可変長文字ストリング
CLOB	CLOB 構造化フォーム		CLOB 構造化フォーム 注: ILE COBOL でのみサポートされます。
GRAPHIC(n)	wchar_t ... [n+1]		PIC G(n) DISPLAY-1 または PIC N(n) 注: ILE COBOL でのみサポートされます。
VARGRAPHIC(n)	VARGRAPHIC 構造化フォーム		可変長グラフィック・ストリング 注: ILE COBOL でのみサポートされます。

表 47. 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 および ILE COBOL
DBCLOB	DBCLOB 構造化フォーム		DBCLOB 構造化フォーム 注: ILE COBOL でのみサポートされます。
BINARY	BINARY 構造化フォーム		BINARY 構造化フォーム
VARBINARY	VARBINARY 構造化フォーム		VARBINARY 構造化フォーム
BLOB	BLOB 構造化フォーム		BLOB 構造化フォーム 注: ILE COBOL でのみサポートされます。
DATE	char ... [11]	TYPE(*CHAR) LEN(10)	PIC X(10) 注: ILE COBOL の場合のみ、FORMAT DATE。
XML	XML 構造化フォーム		XML 構造化フォーム
TIME	char ... [9]	TYPE(*CHAR) LEN(8)	PIC X(8) 注: ILE COBOL の場合のみ、FORMAT TIME。
TIMESTAMP(n)	char ... [20] (n = 0 の場合) char ... [n+21] (n > 0 の場合)	TYPE(*CHAR) LEN(19) (n = 0 の場合) TYPE(*CHAR) LEN(n+20) (n > 0 の場合)	PIC X(19) (n = 0 の場合) PIC X(n+20) (n > 0 の場合) 注: ILE COBOL の場合のみ、FORMAT TIMESTAMP。
ROWID	ROWID 構造化フォーム		ROWID 構造化フォーム
データ・リンク			
アレイ			
標識変数	short		PIC S9(4) BINARY

表 48. 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	
DECIMAL (p,s)	BigDecimal	BigDecimal	FIXED DEC(p,s)
NUMERIC(p,s)	BigDecimal	BigDecimal	
DECFLOAT	BigDecimal	BigDecimal	
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)

表 48. パラメーターのデータ・タイプ (続き)

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
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	
VARGRAPHIC(n)	文字列	文字列	
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
XML AS CLOB	java.sql.CLOB		
XML AS BLOB	java.sql.BLOB		
DATE	日付	文字列	CHAR(10)
TIME	時間	文字列	CHAR(8)
TIMESTAMP(n)	タイム・スタンプ	文字列	CHAR(19) (n = 0 の場合) CHAR(n+20) (n > 0 の場合)
ROWID	バイト[]	com.ibm.db2.app.Blob	ROWID 構造化フォーム
データ・リンク			
アレイ	java.sql.Array		
標識変数			FIXED BIN(15)

表 49. パラメーターのデータ・タイプ

SQL データ・タイプ	REXX	RPG	ILE RPG
SMALLINT		1 つのサブフィールドを含むデータ構造。サブフィールド指定の 43 桁目は B 、長さは 2、52 桁目は 0 とする。	データ指定。サブフィールド指定の 40 桁目は B 、長さ <= 4、41 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 52 桁目は 0 とする。	データ指定。サブフィールド指定の 40 桁目は B 、長さ <=09 かつ >=05、41 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。

表 49. 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
BIGINT			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)		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。
DECFLOAT			
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		データ指定。40 桁目は F 、長さは 4 とする。
DOUBLE PRECISION または FLOAT または FLOAT(p)	数字の次に E が入り、(次にオプションの先行符号)、次に数字が入ったストリング		データ指定。40 桁目は F 、長さは 8 とする。
CHARACTER(n)	2 つのアポストロフィの間に n 個の文字が入ったストリング	サブフィールドを含まないデータ構造または 1 つのサブフィールドを含むデータ構造。サブフィールド指定の 43 桁目と 52 桁目は ブランク とする。あるいは、文字入力フィールドまたは計算結果フィールド。	データ指定。サブフィールド指定の 40 桁目は A とする。あるいは、40 桁目と 41 から 42 桁目を ブランク とする。
VARCHAR(n)	2 つのアポストロフィの間に n 個の文字が入ったストリング		データ指定。サブフィールド指定の 40 桁目は A とする。あるいは、40 桁目と 41 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		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			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		データ指定。サブフィールド指定の 40 桁目は G とする。

表 49. 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
VARGRAPHIC(n)	G' で始まり、n 個の 2 バイト文字の後に 'が入ったストリング		データ指定。サブフィールド指定の 40 桁目は G とし、44 から 80 桁目にキーワード VARYING を置く。
DBCLOB			DBCLOB 構造化フォーム
BINARY			BINARY 構造化フォーム
VARBINARY			VARBINARY 構造化フォーム
BLOB			BLOB 構造化フォーム
XML			XML 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(n)	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。n = 0 の場合、長さは 19。n > 0 の場合、長さは n+20。	n = 0 の場合は長さ 19、n > 0 の場合は長さ n+20 のタイム・スタンプ・データ・タイプ。
ROWID			ROWID 構造化フォーム
データ・リンク			
アレイ			
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。

関連概念:

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

CALL ステートメントでホスト変数と標識変数を使用すると、プロシージャとの間で追加の情報を受け渡すことができます。

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、標識変数は負の値 -1、-2、-3、-4、または -6 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。

- 標識変数が負の値の場合、CALL ステートメント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、ホスト変数と標識変数は未変更のまま渡されます。

*EXTIND オプションを指定せずにコンパイルされた SQL プロシージャまたは外部プロシージャが呼び出される際には、拡張標識値 -5 および -7 を渡せません。その後 CALL ステートメントに関するエラーが発行されます。*EXTIND オプションを指定してコンパイルされた外部プロシージャが呼び出される際には、拡張標識値を渡せます。

処理規則は、プロシージャへの入力パラメーター、およびプロシージャから戻される出力パラメーター共に同じです。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、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 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、398 ページの『コードに関するライセンス情報および特記事項』のご使用条件に同意する必要があります。

プロシージャは、次のように定義されているとします。 ILE RPG プログラムは拡張標識を使用できないようにコンパイルされています。

```
CREATE PROCEDURE PROC1
  (INOUT DECIMALOUT DECIMAL(7,2), INOUT DECOUT2 DECIMAL(7,2))
  EXTERNAL NAME LIB1.PROC1 LANGUAGE RPGLE
  GENERAL WITH NULLS)
+++++
Program CRPG
+++++
D INOUT1          S           7P 2
D INOUT1IND       S           4B 0
D INOUT2          S           7P 2
D INOUT2IND       S           4B 0
C                 EVAL       INOUT1 = 1
C                 EVAL       INOUT1IND = 0
C                 EVAL       INOUT2 = 1
C                 EVAL       INOUT2IND = -2
C/EXEC SQL CALL PROC1 (:INOUT1 :INOUT1IND , :INOUT2
C+                   :INOUT2IND)
C/END-EXEC
C                 EVAL       INOUT1 = 1
C                 EVAL       INOUT1IND = 0
C                 EVAL       INOUT2 = 1
C                 EVAL       INOUT2IND = -2
C/EXEC SQL CALL PROC1 (:INOUT1 :INOUT1IND , :INOUT2
C+                   :INOUT2IND)
C/END-EXEC
C   INOUT1IND     IFLT       0
C*               :
C*               HANDLE NULL INDICATOR
C*               :
C                 ELSE
C*               :
C*               INOUT1 CONTAINS VALID DATA
C*               :
C                 ENDIF
```

```

C*          :
C*          HANDLE ALL OTHER PARAMETERS
C*          IN A SIMILAR FASHION
C*          :
C           RETURN
+++++
End of PROGRAM CRPG
+++++
+++++
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*          CODE FOR INOUTP DOES NOT CONTAIN MEANINGFUL DATA
C*          :
C           ELSE
C*          :
C*          CODE FOR INOUTP CONTAINS MEANINGFUL DATA
C*          :
C           ENDF
C*          PROCESS ALL REMAINING VARIABLES
C*
C*          BEFORE RETURNING, SET OUTPUT VALUE FOR FIRST
C*          PARAMETER AND SET THE INDICATOR TO A NON-NEGATIVE
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) = -1
C           RETURN
+++++
End of PROGRAM PROC1
+++++

```

呼び出し側プログラムへの完了状況の戻り

SQL プロシージャと外部プロシージャとでは、呼び出し側プログラムに状況情報を戻す方法が異なります。

SQL プロシージャでは、プロシージャ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・パラメータをコーディングして、それをプロシージャ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、外部プロシージャを呼び出す呼び出し側プログラム (オペレーティング・システム・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。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。メッセージが通知されると、エラーが SQLCODE/SQLSTATE -443/38501 として返されます。

関連資料:

SQL 制御ステートメント

DB2 から外部プロシージャへのパラメーターの引き渡し

DB2 は、プロシージャ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プロシージャ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・プログラムでは、プロシージャ・コードの中でパラメーターを正しく定義してください。

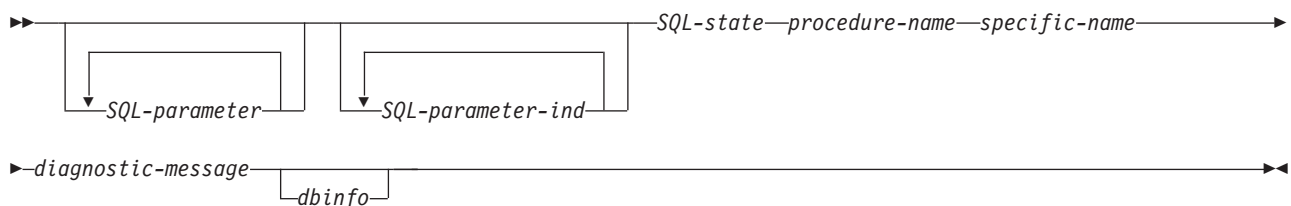
外部プロシージャの中でパラメーターを定義し、それらを使用する場合は、特定のパラメーター用に定義されたよりも多くの記憶域がそのパラメーターにおいて参照されないように十分な注意が必要です。パラメーターはすべて同じ記憶スペースに格納されるので、あるパラメーターの記憶スペースを超えると、別のパラメーターの値を上書きすることになります。このことは、無効な入力データをプロシージャが使用する原因になったり、データベースに返される値が無効になったりする原因になります。

外部プロシージャ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:

SQL パラメーター・スタイルは、業界標準の SQL に準拠しています。

パラメーター・スタイル SQL を使用すると、パラメーターは外部プログラムに、以下のように渡されます。



SQL-parameter

この引数は、プロシージャを呼び出す前に、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、プロシージャ定義で指定されたパラメーターの数です。これらの各パラメーターの値は、CALL ステートメントで指定された式からとられます。この表記は、CREATE PROCEDURE ステートメントで定義されたパラメーターのデータ・タイプになります。注: INPUT として定義されたパラメーターに対する変更は、返されるときに DB2 によって無視されます。

SQL-parameter-ind

この引数は、プロシージャを呼び出す前に、DB2 によって設定されます。これは、対応する SQL-parameter がヌルかどうかを判断するために、プロシージャによって使用されます。前に述

べたように、 n 番目の *SQL-parameter-ind* は、 n 番目の *SQL-parameter* 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 パラメーターが存在し、ヌルではない。
- 1 パラメーターがヌルである。

注: INPUT パラメーターに対応する標識に対する変更は、返されるときに DB2 によって無視されます。

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、プロシーチャーの結果状態としてプロシーチャーによって設定されます。通常、SQLSTATE はプロシーチャー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シグナル通知するのに使用することができます。

01Hxx プロシーチャー・コードが警告状態を検出しました。これにより、SQL 警告が出されます。ここで xx は、可能ないくつかのストリングのいずれかです。

38xxx プロシーチャー・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。

procedure-name

この引数は、プロシーチャーを呼び出す前に、DB2 によって設定されます。引数は、プロシーチャーの名前が入っている VARCHAR(139) 値です。プロシーチャー・コードがこのプロシーチャーのために呼び出されます。

渡されるプロシーチャー名の形式は、以下のようになります。

`<schema-name>.<procedure-name>`

プロシーチャー・コードが複数のプロシーチャー定義で使用され、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。プロシーチャー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、プロシーチャーを呼び出す前に、DB2 によって設定されます。引数は、プロシーチャーの特定の名前が入っている VARCHAR(128) 値です。プロシーチャー・コードがこのプロシーチャーのために呼び出されます。

procedure-name と同様に、プロシーチャー・コードが複数のプロシーチャー定義で使用され、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。プロシーチャー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、プロシーチャーを呼び出す前に、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プロシーチャーによってシグナル通知されたときに、メッセージ・テキストを送信して戻すためにプロシーチャーが使用する VARCHAR(70) 値です。

これは、プロシーチャーに入力があつたときにデータベースによって初期設定され、記述情報がプロシーチャーによって設定されます。SQL-state パラメーターがプロシーチャー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。

dbinfo この引数は、プロシーチャーを呼び出す前に、DB2 によって設定されます。これは、プロシ

ャーの CREATE PROCEDURE ステートメントが DBINFO キーワードを指定したときのみ存在します。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:

GENERAL パラメーター・スタイルを使用すると、パラメーターは、CREATE PROCEDURE ステートメントで指定されたのと同じように、外部プログラムに渡されます。

形式は以下のようになります。



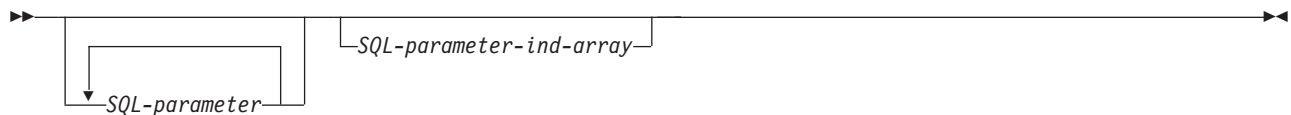
SQL-parameter

この引数は、プロシージャを呼び出す前に、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、プロシージャ呼び出しで指定されたパラメーターの数です。これらの各パラメーターの値は、CALL ステートメントで指定された式からとられます。この表記は、CREATE PROCEDURE ステートメントで定義されたパラメーターのデータ・タイプになります。
注: INPUT として定義されたパラメーターに対する変更は、返されるときに DB2 によって無視されます。

パラメーター・スタイル GENERAL WITH NULLS:

GENERAL WITH NULLS パラメーター・スタイルは、パラメーター値および NULL 標識値の両方を外部プログラムに渡します。

このパラメーター・スタイルを使用すると、パラメーターはプログラムに対して以下のように渡されます。



SQL-parameter

この引数は、プロシージャを呼び出す前に、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、プロシージャ呼び出しで指定されたパラメーターの数です。これらの各パラメーターの値は、CALL ステートメントで指定された式からとられます。この表記は、CREATE PROCEDURE ステートメントで定義されたパラメーターのデータ・タイプになります。
注: INPUT として定義されたパラメーターに対する変更は、返されるときに DB2 によって無視されます。

SQL-parameter-ind-array

この引数は、プロシージャを呼び出す前に、DB2 によって設定されます。これは、ヌルである SQL-parameter が 1 つ以上あるかどうかを判別するために、プロシージャ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-parameter 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 パラメーターが存在し、ヌルではない。
- 1 パラメーターがヌルである。

ヌルの入力があるかどうかをプロシージャで検査する必要があります。注: INPUT として定義されたパラメーターに対応する標識配列項目に対する変更は、返されるときに DB2 によって無視されます。

パラメーター・スタイル DB2GENERAL:

DB2GENERAL パラメーター・スタイルは、Java プロシージャによって使用されます。

関連概念:

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。

関連概念:

Java SQL ルーチン

動的複合ステートメント

動的複合ステートメントは、永続オブジェクトを作成する必要がないという点を除いて、SQL プロシージャと似ています。例えば、動的複合ステートメントを使用して、ロジックをスクリプトに追加できます。

動的複合ステートメントを実行すると、プログラムの作成、実行、および廃棄のオーバーヘッドが生じます。このオーバーヘッドのため、ステートメントを頻繁に実行する必要がある場合は、SQL プロシージャを使用する必要があります。動的複合ステートメントには入出力パラメーターがありません。代わりに、グローバル変数を使用して入力値および戻り値を渡します。

定数値が含まれる表をセットアップする必要があるスクリプトがあるものとします。例えば、年の毎日に 1 行がある表があるとします (1 から 366 の整数)。

```
CREATE TABLE day_numbers (day_value INT)
```

表にすべての正しい値が含まれているかが分からない場合、すべての行を削除してから、再び行を挿入する必要があります。スクリプトに複合 (動的) ステートメントを導入することで、表が既に正しく構築されている場合、データの再入力が必要なくなります。

```
BEGIN
  DECLARE day_count INT;
  DECLARE unique_day_count INT DEFAULT 0;
  DECLARE insert_cnt INT;
  DECLARE CONTINUE HANDLER FOR SQLSTATE VALUE '42704'
    /* Handle table does not exist error */
    CREATE TABLE day_numbers (day_value INT);
  SELECT COUNT(DISTINCT day_value) , COUNT(day_value)
    INTO unique_day_count, day_count
    FROM day_numbers;
  IF day_count = 366 AND unique_day_count = 366 THEN
    BEGIN END;          /* Table correctly populated */
  ELSE
    BEGIN
      DELETE FROM day_numbers; /* Remove all rows */
      SET insert_cnt = 1;
      WHILE insert_cnt < 367 DO
        INSERT INTO day_numbers VALUES insert_cnt;
        SET insert_cnt = insert_cnt + 1;
      END WHILE;
    END
```

```
|      END;  
|      END IF;  
|  
| END
```

| 関連資料:

| 複合 (動的) ステートメント

ユーザー定義関数の使用

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ユーザー定義関数 (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。

例えば、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。例えば、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:

13 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

ユーザー定義関数 (UDF) とは、CREATE FUNCTION ステートメントによって DB2 データベース・システムに対して定義され、SQL ステートメント中で参照できる関数のことです。UDF は外部関数でも SQL 関数でもかまいません。

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。
- QSYS2.DISPLAY_JOURNAL などの一部のシステム提供サービスは、システムによって定義されて保守されますが、ユーザー定義関数と見なされます。

さらに、各関数は、スカラー関数、集約関数、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、単一値の応答を返します。例えば組み込み関数 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。

集約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返します。組み込み関数のいくつかは集約関数です。集約関数の例に、組み込み関数 AVG() があります。外部 UDF は、集約関数として定義することはできません。ただし、UDF が組み込み集約関数のいずれかのソースになっている場合には、その UDF は集約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。例えば特殊タイプ 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) を既存の組み込み集約関数 AVG(INTEGER) のソースになっている集約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返します。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。例えば、あるファイルを手に入れたら表に変換したり、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 命名における関数のフルネームは <schema-name>/<function-name> または <schema-name>.<function-name> です。DML ステートメントでは、関数名は *SYS 命名でスラッシュ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。例えば、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。例えば、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 ステートメントに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。

関連タスク:

QAQQINI による照会の動的な制御

関連資料:

最適の判別

SQL 関数としての UDF の作成

SQL 関数 とは、CREATE FUNCTION ステートメントを使用して定義、作成、および登録を行うユーザー定義関数 (UDF) のことです。

SQL 関数は、SQL 言語のみで作成され、その定義は 1 つの (大きくなる可能性がある) CREATE FUNCTION ステートメント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スカラー関数を示します。

```
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を示します。

```
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 PROJECT
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

ユーザー定義関数 (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。

関連概念:

Java SQL ルーチン

UDF の登録:

ユーザー定義関数 (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。CREATE FUNCTION ステートメントを使用して UDF を登録できます。

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。

関連資料:

CREATE FUNCTION

例: 指数:

浮動小数点値の指数を求める外部関数を作成し、その関数を MATH スキーマに登録するとします。

```
CREATE FUNCTION MATH.EXPON (DOUBLE, DOUBLE)
  RETURNS DOUBLE
  EXTERNAL NAME 'MYLIB/MYPGM(MYENTRY)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  RETURNS NULL ON NULL INPUT
  ALLOW PARALLEL
```

この例では、どちらかの引数が NULL の場合に、結果値を NULL にする必要があるため、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:

ある文字列 (引数として渡される) が、ある文字ラージ・オブジェクト (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ユーザー定義関数 (UDF) を作成するとします。UDF は、文字列があった場合には、その文字列の CLOB 内の位置を返し、なかった場合はゼロを返します。

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:

FINDSTRING 関数が文字ラージ・オブジェクト (CLOB) 上で機能するのと同じくバイナリー・ラージ・オブジェクト (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 関数をもう 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: ユーザー定義タイプ (UDT) での文字列検索:

バイナリー・ラージ・オブジェクト (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 を定義するとします。

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『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ため、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ため、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、210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。

例: ユーザー定義タイプ (UDT) での AVG:

この例は、CANADIAN_DOLLAR 特殊タイプについて AVG 集約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 引数を取りません。この関数は、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:

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するとします。

このユーザー定義関数 (UDF) は、次のように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、ユーザー定義関数 (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。

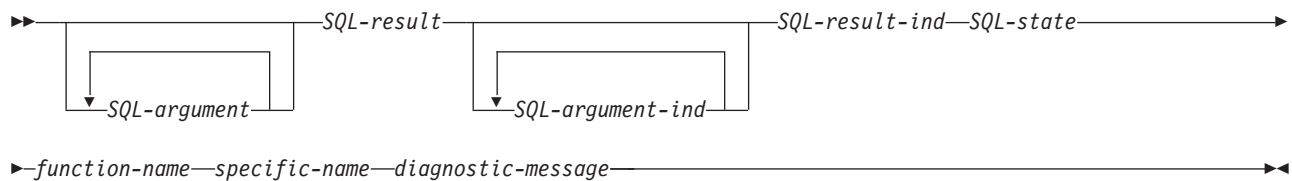
UDF の中でパラメーターを定義し、使用する場合は、あるパラメーター用に定義されたよりも多くの記憶域が、そのパラメーターに対して参照されないように十分な注意が必要です。パラメーターはすべて同じ記憶スペースに格納されるので、あるパラメーターの記憶スペースを超えると、別のパラメーターの値を上書きすることになります。このことは、無効な入力データを関数を使用する原因になったり、データベースに戻される値が無効になる原因になります。

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:

SQL パラメーター・スタイルは、業界標準の 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・ユーザー定義関数 (UDF) のみと併用できます。

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な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。

関連資料:

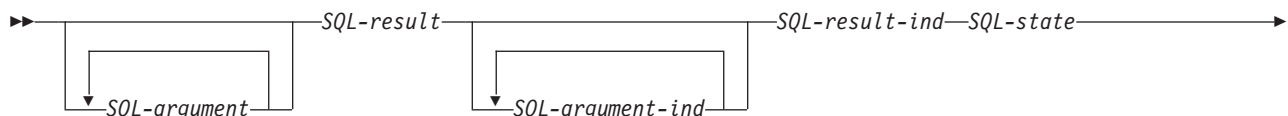
SQL メッセージおよびコード

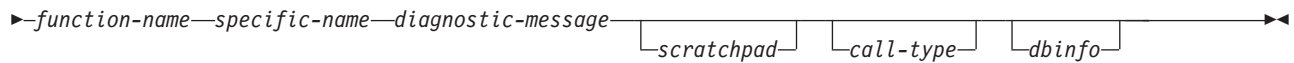
パラメーター・スタイル DB2SQL:

DB2SQL パラメーター・スタイルを使用すると、SQL パラメーター・スタイルと同じように、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。

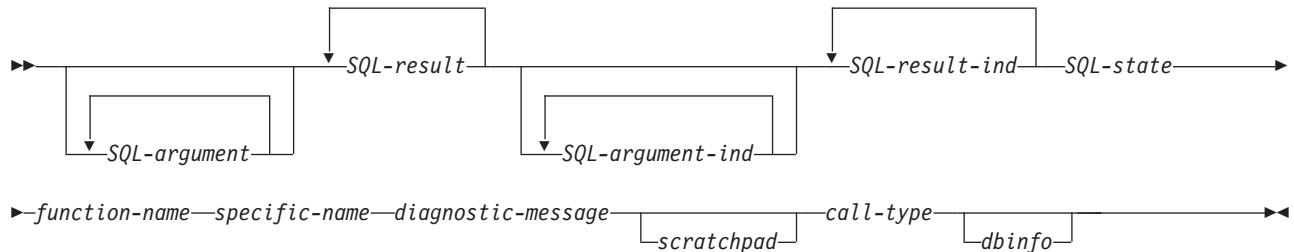
UDF 定義に以下のオプション・パラメーターが複数指定された場合は、以下に定義された順序で UDF に渡されます。共通のパラメーターについては、パラメーター・スタイル SQL を参照してください。このパラメーター・スタイルは、スカラー UDF と表 UDF の両方で使用できます。

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ようになります。

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**:

GENERAL パラメーター・スタイルを使用すると、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・ユーザー定義関数 (UDF) のみと併用できます。

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。

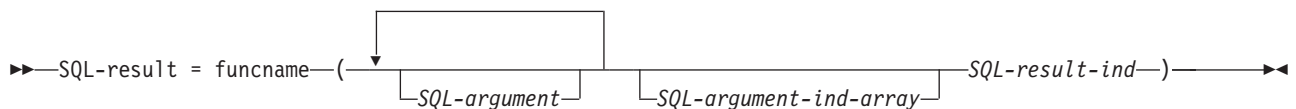
CREATE FUNCTION ステートメントで CAST FROM 文節が使用されている場合は、DB2 は、UDF が CAST FROM 文節で定義された値を戻すと想定し、CAST FROM 文節が使用されていない場合は、DB2 は、UDF が RETURNS 文節で定義された値を戻すと想定します。

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:

GENERAL WITH NULLS パラメーター・スタイルは、スカラー・ユーザー定義関数 (UDF) のみと併用できます。

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。例えば、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で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:

DB2GENERAL パラメーター・スタイルは、Java ユーザー定義関数 (UDF) によって使用されます。

関連概念:

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。

関連概念:

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る SQL ステートメントに対して表を引き渡すユーザー定義関数 (UDF) です。表関数参照は、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一度に 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-idx 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ると、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:

ユーザー定義関数 (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ってそのエラーを扱います。

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 として定義されるユーザー定義関数 (UDF) は、その関数を呼び出した 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有します。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。

関連資料:

マルチスレッド・アプリケーション

221 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。

並列処理:

並列処理が行えるように 1 つのユーザー定義関数 (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。

関連資料:

マルチスレッド・アプリケーション

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:

220 ページの『スレッドについての考慮事項』

FENCED として定義されるユーザー定義関数 (UDF) は、その関数を呼び出した 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、QSYS2、SYSIBM、SYSPROC、および SYSIBMADM 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。
- | 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、SQL7909 警告が返されます。
- | SQL7909 警告には、プログラムを更新できなかった理由を示す理由コードが含まれています。

| デフォルト・パラメーターを使用した UDF の定義

- | デフォルト値を持つように UDF のパラメーターを定義できます。

パラメーターのデフォルト値があるということは、そのパラメーターが関数呼び出しで必須ではないということを示します。デフォルト値を使用することで、すべての関数呼び出しですべてのパラメーター値を渡すということが必要ではなくなるため、柔軟性が得られます。これは、既存の関数を変更する際に役立つことがあります。

デフォルト値は、定数やヌル値などの単純な値にすることができます。また、SQL 式にすることもできるため、非常に複雑になる場合もあります。デフォルト値は、デフォルト式の一部として別のパラメーターの値を使用して定義することはできません。パラメーターのデフォルト値で参照されるすべてのオブジェクトは、関数の作成時に存在している必要があります。

プロジェクト番号である 1 つのパラメーターを渡すと、プロジェクト活動表から、その番号で登録されているすべての行を返す表関数があるものとします。返される行数を開始日の範囲に制限するように要求された場合、日付範囲のデフォルト・パラメーターを追加することでこれを簡単に実現できます。

```
CREATE FUNCTION ActProj (ProjNum CHAR(6),
                        StartAfterDate DATE DEFAULT NULL,
                        StartBeforeDate DATE DEFAULT NULL)
    RETURNS TABLE (PROJNO CHAR(6),
                  ACTNO SMALLINT,
                  STARTDATE DATE)
```

関数ロジックは、日付が含まれるかヌル値になる可能性がある 2 つの新規パラメーターを受け入れるように変更する必要があります。日付範囲を渡す理由がない関数の呼び出しは、変更する必要がありません。引き続き、以下ようになります。

```
ActProj(:projnum)
```

日付範囲が必要なアプリケーションは、以下のように、日付パラメーターの 1 つまたは両方を渡すことができます。

```
ActProj(:projnum, :StartDate, :EndDate)
```

デフォルト値は SQL 関数または外部関数に対して定義できます。

例: UDF コード

以下の例は、SQL 関数と外部関数を使用してユーザー定義関数 (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:

ある数の平方を返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、398 ページの『コードに関するライセンス情報および特記事項』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ユーザー定義関数 (UDF) を作成します。

注: コード例を使用する場合は、398 ページの『コードに関するライセンス情報および特記事項』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[92];
};

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: 天気の変関数:

米国内のさまざまな都市の天気情報を戻す変関数を作成するとします。

注: コード例を使用する場合は、398 ページの『コードに関するライセンス情報および特記事項』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詳細は、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_ptr = 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_ptr] == ( 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_ptr] ) ;
            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, ' ', 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, ' ', 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] = ' ' ;
            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注を参照しながら、以下の点にご注意ください。

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. 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)';

```

この表関数定義の場合、以下の点にご注意ください。

- 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。
- デフォルトの 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。

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

スカラーおよび列ユーザー定義関数 (UDF) は、式が有効であれば SQL ステートメントのほとんどどこからでも呼び出すことができます。表 UDF は、SELECT ステートメントの FROM 文節で呼び出すことができます。UDF の使用に関する 2、3 の制約事項を以下にリストします。

- 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 (ALWCOPYDTA(*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:

- 1 関数引数としてパラメーター・マーカーまたは NULL 値を使用できます。関数解決には、タイプなし引数が関数選択プロセス時にどのように使用されるのかについてのルールがあります。

- 1 パスが正しくセットアップされていれば、以下をコーディングできます。

```
BLOOP(?)
```

あるいは

```
BLOOP(NULL)
```

- 1 CAST 指定を使用して、正しい関数を検出するために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。例えば、以下のようになります。

```
BLOOP(CAST(? AS INTEGER))
```

あるいは

```
BLOOP(CAST(NULL AS INTEGER))
```

関連資料:

最適の判別

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指定されたスキーマに制限します。

例えば、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。この場合、両方の PABLO.BLOOP が数値の引数をとります。COLUMN1 が数値タイプにキャストできない場合、ステートメントは失敗します。COLUMN1 が文字またはグラフィック・タイプの場合、関数解決のためのキャスト可能なプロセスは、2 番目の BLOOP 関数に解決されます。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。一致する関数を検索する際に、DB2 は通常関数パスを使用して参照を修飾します。

265 ページの『UDT の定義』

CREATE DISTINCT TYPE ステートメントを使用して、ユーザー定義タイプ (UDT) を定義します。

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。一致する関数を検索する際に、DB2 は通常関数パスを使用して参照を修飾します。

DROP FUNCTION および COMMENT ON FUNCTION ステートメントのケースでは、これらのステートメントが *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。例えば、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想定しているため、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:

231 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指定されたスキーマに制限します。

名前付き引数を使用した UDF の呼び出し:

パラメーター値とともにパラメーター名を渡して、UDF を呼び出すことができます。

関数の呼び出し時に、パラメーターは通常、関数に定義された順序でリストされます。パラメーター名を使用することで、任意の順序でリストすることもできます。デフォルト値があるパラメーターの引数は、完全に省略できます。これは、任意のユーザー定義関数、SQL、または外部関数に当てはまります。

以下のような関数を定義しているものとします。

```
CREATE FUNCTION testfunc (parm1 INT, parm2 INT, parm3 INT DEFAULT 10, parm4 INT DEFAULT -1) ...
```

最初の 2 つのパラメーターにはデフォルト値がないため、これらのパラメーターは関数呼び出しのどこかに含める必要があります。最後の 2 つのパラメーターは、デフォルト値があるため、省略できます。以下のすべての例は有効であり、同じ値を関数に渡しています。

```
testfunc(10, 20, DEFAULT, 22)
testfunc(parm1=>10, parm2 => 20, parm3 => DEFAULT, parm4 => 22)
testfunc(10, 20, parm4=> 22)
testfunc(parm4 => 22, parm2 => 20, parm1 =>10)
```

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、関数参照はネストできるということです。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、集約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。例えば、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。例えば、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。209 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。

関連資料:

最適の判別

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。SQL トリガーは、単一のトリガー・プログラム内でこれらの複数のイベントを持つように定義することもできます。

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せん。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 300 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

関連タスク:

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、表またはビュー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。

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たびに、SQL トリガーは、ストアード・プロシージャ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。トリガーの作成時にオブジェクトが存在しない場合、動的 SQL を使用して、オブジェクトを参照するステートメントを生成できます。SQL トリガーのルーチン本体の中のステートメントは、SQL によってプログラム・オブジェクト (*PGM) に変

換されます。プログラムは、トリガー名修飾子で指定されるスキーマに作成されます。指定されたトリガーは、SYSTRIGGERS、SYSTRIGDEP、SYSTRIGCOL、および SYSTRIGUPD の各 SQL カタログに登録されます。

関連概念:

246 ページの『SQL ルーチンのデバッグ』

CREATE PROCEDURE、CREATE FUNCTION、または CREATE TRIGGER 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。

関連資料:

SQL 制御ステートメント

CREATE TRIGGER

BEFORE SQL トリガー:

前トリガーは、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、ユーザー定義タイプ (UDT) およびストアード・プロシージャにアクセスし、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、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、ストアド・プロシージャ、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:

後トリガーは、対応する変更の挿入、更新、または削除が表に適用された後に実行されます。

SQL トリガーの中で WHEN 条件を使用して、条件を指定することができます。条件の評価の結果が true (真) の場合は、SQL トリガーのルーチン本体の中の SQL ステートメントが実行されます。条件が false (偽) の場合は、SQL トリガーのルーチン本体の中の SQL ステートメントは実行されず、制御はデータベース・システムに戻されます。

次の例では、照会が評価され、トリガーが活動化されたときに、トリガー・ルーチン本体内のステートメントを実行するべきかどうかが決められます。

```

CREATE TABLE TodaysRecords(TodaysMaxBarometricPressure FLOAT,
  TodaysMinBarometricPressure FLOAT)

CREATE TABLE OurCitysRecords(RecordMaxBarometricPressure FLOAT,
  RecordMinBarometricPressure FLOAT)

CREATE TRIGGER UpdateMaxPressureTrigger
AFTER UPDATE OF TodaysMaxBarometricPressure ON TodaysRecords
REFERENCING NEW AS new_row
FOR EACH ROW MODE DB2ROW
WHEN (new_row.TodaysMaxBarometricPressure>
  (SELECT MAX(RecordMaxBarometricPressure) FROM
    OurCitysRecords))
  UPDATE OurCitysRecords
    SET RecordMaxBarometricPressure =
      new_row.TodaysMaxBarometricPressure

CREATE TRIGGER UpdateMinPressureTrigger
AFTER UPDATE OF TodaysMinBarometricPressure
ON TodaysRecords
REFERENCING NEW AS new_row
FOR EACH ROW MODE DB2ROW
WHEN(new_row.TodaysMinBarometricPressure<
  (SELECT MIN(RecordMinBarometricPressure) FROM

```



```

    OurCitysRecords))
UPDATE OurCitysRecords
    SET RecordMinBarometricPressure =
        new_row.TodaysMinBarometricPressure

```

まず、表の現行値が初期設定されます。

```

INSERT INTO TodaysRecords VALUES(0.0,0.0)
INSERT INTO OurCitysRecords VALUES(0.0,0.0)

```

次の SQL 更新ステートメントでは、OurCitysRecords の中の RecordMaxBarometricPressure が、UpdateMaxPressureTrigger によって更新されます。

```

UPDATE TodaysRecords SET TodaysMaxBarometricPressure = 29.95

```

しかし、翌日、TodaysMaxBarometricPressure が 29.91 でしかなかった場合は、RecordMaxBarometricPressure は更新されません。

```

UPDATE TodaysRecords SET TodaysMaxBarometricPressure = 29.91

```

SQL では、1 つのトリガー・アクションに複数のトリガーを定義することができます。上の例では、2 つの AFTER UPDATE トリガー、すなわち、UpdateMaxPressureTrigger と UpdateMinPressureTrigger があります。これらのトリガーが活動化されるのは、表 TodaysRecords の特定の列が更新されたときだけです。

後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。

複数イベント SQL トリガー:

複数の挿入、更新、または削除イベントに対して単一のトリガーを定義できます。

挿入、更新、および削除イベントの 1 つ、2 つ、または 3 つすべてを処理するように単一のトリガーを定義できます。イベントが共通の BEFORE、AFTER、または INSTEAD OF トリガー時間を共有する必要があります。これが唯一の制限です。トリガー・イベント述部をトリガー本体内で使用して、トリガーを活動化したイベントを区別できます。

以下の例では、複数の警告 SQLSTATE を単一のトリガー定義にグループ化しています。

```

CREATE TABLE PARTS (INV_NUM INT, PART_NAME CHAR(20), ON_HAND INT, MAX_INV INT,
                    PRIMARY KEY (INV_NUM))

CREATE OR REPLACE TRIGGER INVENTORY_WARNINGS
AFTER DELETE OR INSERT OR UPDATE OF ON_HAND ON PARTS
REFERENCING NEW AS N_INV
              OLD AS O_INV
FOR EACH ROW MODE DB2SQL
BEGIN
    IF INSERTING THEN
        IF (N_INV.ON_HAND > N_INV.MAX_INV) THEN
            BEGIN
                SIGNAL SQLSTATE '75001' ('Inventory on hand exceeds maximum allowed.');
```



```

|         BEGIN
|         SIGNAL SQLSTATE '75002' ('Inventory low - Re-order soon. ');
|         END;
|         END IF;
|         ELSEIF DELETING THEN
|         IF (O_INV.ON_HAND > 0) THEN
|         BEGIN
|         SIGNAL SQLSTATE '75004' ('Deleting a part while inventory still in stock. ');
|         END;
|         END IF;
|         END IF;
|         END

```

トリガー本体で、INSERTING、UPDATING、および DELETING 述部を使用して、トリガーを活動化したイベントを判別します。定義された各イベントに対して別個の一節のコードが実行されます。イベントごとに警告条件が処理されます。

次の例では、トリガー・イベント述部を INSERT ステートメントで使用して、トランザクション履歴表の行を生成しています。

```

| CREATE TABLE TRANSACTION_HISTORY(INV_NUM INT, POSTTIME TIMESTAMP,
|                                     TRANSACTION_TYPE CHAR(1))
|
| CREATE TRIGGER SET_TRANS_HIST
| AFTER INSERT OR UPDATE OR DELETE ON PARTS
| REFERENCING NEW AS N
|               OLD AS O
| FOR EACH ROW MODE DB2SQL
| BEGIN
|   INSERT INTO TRANSACTION_HISTORY VALUES (
|     CASE
|       WHEN INSERTING OR UPDATING
|       THEN N.INV_NUM
|       WHEN DELETING
|       THEN O.INV_NUM
|     END,
|     CURRENT_TIMESTAMP,
|     CASE
|       WHEN INSERTING
|       THEN 'I'
|       WHEN UPDATING
|       THEN 'U'
|       WHEN DELETING
|       THEN 'D'
|     END
|   );
| END

```

このトリガーでは、3 つすべてのトリガー・イベントで同じルーチン・ロジックが使用されています。トリガー・イベント述部を使用して、在庫数を行値の前と後のどちらから読み取る必要があるのかを判別し、さらにもう 1 回使用してトランザクションのタイプを設定しています。

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 前トリガーや後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。

前トリガーおよび 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 および後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。例えば、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。例えば、トリガーは、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する必要があります。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物理ファイル・トリガー追加 (ADDPFTRG) CL コマンドで表に追加するか、または System i 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

関連タスク:

データベース内での自動イベントのトリガー

SQL のプロシージャおよび関数での配列サポート

- 1 SQL プロシージャおよび SQL スカラー関数は、配列型のパラメーターおよび変数をサポートします。配列は、アプリケーションとストアード・プロシージャの間、2 つのストアード・プロシージャの間、または関数呼び出し時において、一時的なデータ集合を渡すための便利な方法です。

SQL のプロシージャおよび関数内では、配列は従来型のプログラミング言語での配列と同様に扱うことができます。さらに、配列がリレーショナル・モデルに統合されることによって、配列で表現されるデータを表に変換する操作と、表の列内のデータを配列に集約する操作を容易に行えます。以下の例では、配列上でのいくつかの操作を例示します。

例 1

この例では、2 つのプロシージャ sum および main を取り上げます。プロシージャ main は、配列コンストラクターを使用して、6 つの整数で構成される配列を作成します。次いでその配列をプロシージャ sum に渡し、そこで入力配列のすべての要素の合計を計算した後、結果が main に戻ります。プロシージャ sum は、配列の副索引作成の使用法、および配列内の要素数を返す CARDINALITY 関数の使用法を例示するものです。

```

CREATE TYPE intArray AS INTEGER ARRAY[100]

CREATE PROCEDURE sum(IN inList intArray, OUT total INTEGER)
BEGIN
  DECLARE i, n INTEGER;

  SET n = CARDINALITY(inList);

```

```

SET i = 1;
SET total = 0;

WHILE (i <= n) DO
  SET total = total + inList[i];
  SET i = i + 1;
END WHILE;

END

CREATE PROCEDURE main(OUT arrayTotal INTEGER)
BEGIN
  DECLARE numList intArray;

  SET numList = ARRAY[1,2,3,4,5,6];

  CALL sum(numList,arrayTotal);

END

```

例 2

この例は例 1 と似ていますが、1 つのプロシージャ main を使用して、sum という名前の関数を呼び出しています。

```

CREATE TYPE intArray AS INTEGER ARRAY[100]

CREATE FUNCTION sum(inList intArray) RETURNS INTEGER
BEGIN
  DECLARE i, n, total INTEGER;

  SET n = CARDINALITY(inList);

  SET i = 1;
  SET total = 0;

  WHILE (i <= n) DO
    SET total = total + inList[i];
    SET i = i + 1;
  END WHILE;

  RETURN total;

END

CREATE PROCEDURE main(OUT arrayTotal INTEGER)
BEGIN
  DECLARE numList intArray;

  SET numList = ARRAY[1,2,3,4,5,6];

  SET arrayTotal = sum(numList);

END

```

例 3

この例では、2 つの配列データ・タイプ (intArray と stringArray)、および 2 つの列 (id と name) がある persons 表を使用します。プロシージャ processPersons は、追加の 3 人を表に追加し、文字「a」を含んだ個人名を id で順序付けした配列を返します。追加される 3 人の ID および名前は、2 つの配列 (id および name) で表されます。これらの配列は UNNEST 関数の引数として使用され、そこで配列は 2 列の表に変換されます。その後、それらの要素は persons 表に挿入されます。最後に、プロシージャの最後の set ステートメントで、ARRAY_AGG 集約関数を使用して出力パラメーターの値を計算します。


```

CREATE TYPE intArray AS INTEGER ARRAY[100]

CREATE TYPE stringArray AS VARCHAR(10) ARRAY[100]

CREATE TABLE persons (id INTEGER, name VARCHAR(10))

INSERT INTO persons VALUES(2, 'Tom'),
                           (4, 'Gina'),
                           (1, 'Kathy'),
                           (3, 'John')

CREATE PROCEDURE processPersons(OUT witha stringArray)
BEGIN
DECLARE ids intArray;
DECLARE names stringArray;

SET ids = ARRAY[5,6,7];
SET names = ARRAY['Denise', 'Randy', 'Sue'];

INSERT INTO persons(id, name)
  (SELECT t.i, t.n FROM UNNEST(ids, names) AS t(i, n));

SET witha = (SELECT ARRAY_AGG(name ORDER BY id)
            FROM persons
            WHERE name LIKE '%a%');

END

```

SQL ルーチンのデバッグ

CREATE PROCEDURE、CREATE FUNCTION、または CREATE TRIGGER 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。

- | 「SQL オブジェクト・プロセッサ・ルート・ビュー (SQL Object Processor Root View)」を使用して、
- | デバッグで SQL ルーチンをステップスルーできます。各ステップ命令により、生成された基盤の C ステートメントではなく、次の SQL ステートメントに移動します。これにより、ルーチン・ロジックのフロー
- | を辿るのが容易になります。

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。そのような名前に対して名前が生成されるため、このことによってもデバッグや評価を行うことが難しくなります。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。

SQL 配列データ・タイプは、SQL プロシージャでのみサポートされます。また、生成された C プログラム内に配列変数の永続ストレージがないために、このデータ・タイプには特殊なデバッグ規則があります。C 配列とは異なり、SQL 配列は 1 ベースです。配列の 5 番目の要素を表示するには、EVAL コマンドに配列指標 5 を指定します (EVAL TSTARR.ABC[5])。配列指標を指定せずに配列変数名を指定することはできません。これを行うと、変数が見つからなかったことを示すエラーが出されます。配列内のある範囲の値を参照するには、EVAL コマンドに配列指標の範囲を指定して使用できます。例えば、配列内の最初の 25 個の要素を参照するには、EVAL TSTARR.ABC[1..25] とします。EVAL コマンドは、配列変数が NULL 値を持つ場合にそれを示します。NULL である配列変数への参照が含まれる式の評価は NULL になります。数値配列型は、式の中で使用できます。他のタイプは式の中で使用できません。プレフィックスのないストリング・タイプは、ポインターとしてではなくストリングとして表示されます。

すべてのデバッグ・コマンドが SQL 配列型をサポートしているわけではありません。SQL 配列型を割り当ての左側で使用することはできません (EVAL TSTARR.ABC[5] = 3)。それらを「&」アドレス演算子のターゲットとして使用することはできません (EVAL &TSTARR.ABC[3])。配列を WATCH デバッグ・コマンドと一緒に使用することもできません。配列変数は、デバッグ・シンボル表に含まれないため、EVAL %LOCALVARS コマンドで表示されません。

SQL ルーチンまたは SQL トリガーの難読化

- | SQL 関数、プロシージャ、およびトリガーは、ルーチン本体ロジックおよびステートメントがシステムで表示されないようにするために、難読化することができます。
- | SQL プロシージャ、関数、またはトリガーのテストが完了したら、システム上のどのユーザーからも読み取れないようにするために、そのテキストを難読化することができます。これを行うには、WRAP SQL 関数または CREATE_WRAPPED SQL プロシージャを使用します。ともに、スキーマ SYSIBMADM 内のシステム提供ルーチンです。
- | WRAP スカラー関数は、入力引数として指定された SQL ステートメントの難読化形式を返します。
- | CREATE_WRAPPED プロシージャは SQL ステートメントを難読化してから実行します。
- | 難読化されたルーチンが作成されると、ルーチン本体は、カタログ表およびプログラム・オブジェクトに表示されなくなります。リストは生成されず、ルーチンの作成時に使用された一時ファイルはすべて削除されます。ルーチンは常に、デバッグ不能として作成されるため、ルーチン・コンテンツが含まれたデバッグ・ビューは存在しません。
- | System i ナビゲーター の SQL 生成機能には、多数の SQL ルーチンを一度に難読化するのを容易にするオプションが用意されています。

ルーチンを難読化した後に、SQL ルーチンのプログラムまたはサービス・プログラム・オブジェクトを保存して、他のシステムに復元できます。これは、System i ナビゲーター の SQL 生成機能のオブジェクトとして使用できます。DB2 for i は、ステートメントを正しく処理する方法を認識するため、難読化されたままになります。

例えば、以下のプロシージャを使用するものとします。

```
CREATE PROCEDURE UPDATE_STAFF (  
    IN P_EmpNo CHAR(6),  
    IN P_NewJob CHAR(5),  
    IN P_Dept INTEGER)  
BEGIN  
    UPDATE STAFF  
        SET JOB = P_NewJob  
        WHERE DEPT = P_Dept and ID = P_EmpNo;  
END;
```

プロシージャをテストして動作することを確認した後で、WRAP 関数を使用してステートメントの難読化形式を生成できます。

```
VALUES(SYSIBMADM.WRAP(  
'CREATE PROCEDURE UPDATE_STAFF (  
    IN P_EmpNo CHAR(6),  
    IN P_NewJob CHAR(5),  
    IN P_Dept INTEGER)  
BEGIN  
    UPDATE STAFF  
        SET JOB = P_NewJob  
        WHERE DEPT = P_Dept and ID = P_EmpNo;  
END'  
));
```

このステートメントの結果は、以下に示すステートメントのような値が含まれた CLOB 値です。難読化プロセス時にタイム・スタンプが使用されるため、実行するたびに結果は異なるものになる可能性があります。ここでは、便宜上、値を複数行で示します。ステートメント・テキストの折り返し部分で、改行文字は許可されません。

```
CREATE PROCEDURE UPDATE_STAFF ( IN P_EMPNO CHAR ( 6 ) , IN P_NEWJOB CHAR ( 5 )  
, IN P_DEPT INTEGER ) WRAPPED QSQ07010 aacxw8p1w8FjG8pnG8VzG8FD68Fj68:H18:dY_p  
B2qpdw8pdw8pdw_praqebaqebaGEMj_vsPBs5b0JUUqnHVayE1_ogA1Gwqz2jJCIE1dQEjt33hd5Sps  
5cYGViD1urv7vGKe0cC4CwpCibbmeMfsW3XzXWn1plyX9wun0CqFidqaB1
```

これは、実行可能な SQL ステートメントです。元の SQL ステートメントとまったく同じように実行できます。WRAPPED キーワード以降の文字を変更すると、ステートメントは失敗します。

このステートメントをデプロイするために、難読化形式を RUNSQLSTM ソース・メンバーまたはソース・ストリーム・ファイルに組み込むことができます。難読化バージョンの文字をそのまま含めるように、細心の注意を払う必要があります。

SQL ルーチンを難読化する 2 つ目の方法では、以下のように、CREATE_WRAPPED SQL プロシージャを使用します。

```
CALL SYSIBMADM.CREATE_WRAPPED(  
'CREATE PROCEDURE UPDATE_STAFF (  
    IN P_EmpNo CHAR(6),  
    IN P_NewJob CHAR(5),  
    IN P_Dept INTEGER)  
BEGIN  
    UPDATE STAFF
```

```

|         SET JOB = P_NewJob
|         WHERE DEPT = P_Dept and ID = P_EmpNo;
| END'
| );

```

これはプロシージャを作成し、SQL ルーチン本体全体が難読化されます。SYSROUTINES の ROUTINE_DEFINITION 列を見ると、WRAPPED キーワードから始まる、ルーチン本体の難読化形式が示されています。将来に参照するために元のルーチン・ソースが必要な場合は、難読化テキストから元のソースを生成する方法は存在しないため、ソースを保存する必要があります。

関連資料:

WRAP スカラー関数

CREATE_WRAPPED プロシージャ

SQL および外部のルーチン・オブジェクトの管理

SQL および外部の関数およびプロシージャは、システム・プログラムおよびサービス・プログラムを使用して実装されます。これらのオブジェクトは、正しく管理すれば、QSYS2/SYSROUTINES、QSYS2/SYSPARMS、および QSYS2/SYSROUTINEDEP システム・カタログでルーチン登録を再生成できます。

SQL プロシージャまたは関数が作成または変更されると、ILE C プログラムまたはサービス・プログラムが生成されます。このオブジェクトには、実行可能ステートメントが含まれている他、SQL ルーチンとして定義するために使用されるすべての情報が含まれています。

外部プロシージャまたは関数が作成され、ILE プログラムまたはサービス・プログラム・オブジェクトに関連付けられると、ルーチン定義に関する情報が、*PGM または *SRVPGM オブジェクトに保存されます。ルーチン情報をオブジェクトに正常に追加すれば、オブジェクトの保守で必要になる手動作業が低減します。

外部プロシージャまたは関数定義は、以下の場合にのみ保存できます。

- 外部プログラムが ILE *PGM または *SRVPGM オブジェクトである。
- プログラムまたはサービス・プログラムが QSYS、QSYS2、SYSIBM、SYSPROC、および SYSIBMADM 内にない。
- CREATE PROCEDURE または CREATE FUNCTION ステートメントの発行時にプログラムが存在し、プログラムが見つかる。
- CREATE または ALTER ステートメントがプログラムに対する排他ロックを取得できる。
- プログラムに 32 ルーチンの属性がまだ含まれていない。

外部ルーチンの作成、変更、または除去時にメッセージ SQL7909 が発行され、*PGM および *SRVPGM を変更することができません。このメッセージには、理由コードが含まれています。

これらのプロシージャおよび関数の *PGM および *SRVPGM オブジェクトが他のシステム・オブジェクトと同様に管理されている場合、SYSROUTINES および SYSPARMS カタログ情報を保守するために、オブジェクトに保存されている情報が使用されます。以下の CL コマンド (およびそれに対応する API) により、カタログが、プロシージャおよび関数の実行可能オブジェクトと同期された状態に保たれます。

保存/復元 (SAVOBJ/RSTOBJ および SAVLIB/RSTLIB)

ルーチンごとに 1 行が SYSROUTINES カタログに挿入されます。EXTERNAL_NAME 列には、新規に復元された実行可能オブジェクトの名前が含まれます。

複製オブジェクト作成 (CRTDUPOBJ)

新規行が SYSROUTINES カタログに挿入されます。EXTERNAL_NAME 列には、新規に複製された実行可能オブジェクトの名前が含まれます。

ライブラリー・コピー (CPYLIB)

新規行が SYSROUTINES カタログに挿入されます。EXTERNAL_NAME 列には、新規に複製された実行可能オブジェクトの名前が含まれます。

オブジェクト名変更 (RNMOBJ)

SYSROUTINES の既存行が変更されます。EXTERNAL_NAME 列には、名前変更された実行可能オブジェクトの名前が含まれます。

オブジェクトの移動 (MOV OBJ)

SYSROUTINES の既存行が変更されます。EXTERNAL_NAME 列には、移動された実行可能オブジェクトの名前が含まれます。

ライブラリー消去 (CLRLIB)

SPECIFIC_SCHEMA 列が消去対象ライブラリーの名前に一致している SYSROUTINES 内のすべての行が削除されます。

プログラム削除 (DLTPGM) およびサービス・プログラムの削除 (DLTSRVPGM)

SQL ルーチンの場合、SPECIFIC_NAME および SPECIFIC_SCHEMA 列を使用して、SYSROUTINES 内の対応する行が削除されます。外部ルーチンの場合、更新は行われません。

システム・コマンドの使用時に *PGM または *SRVPGM が操作されると、メッセージ SQL9015 が発行され、関連ルーチンのカタログ・エントリーを更新できません。

プロシージャおよび関数のパフォーマンスの向上

ストアド・プロシージャまたはユーザー定義関数 (UDF) を作成する場合に、SQL プロシージャ型言語プロセッサは、常に最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、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させることができます。

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。例えば、C 言語コンパイラーがホスト変数を処理する方法と、SQL プロシージャ型プロセッサ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。

IBM i 6.1 リリースでは、SQL ルーチン内で生成されるコードのパフォーマンスが大幅に改善されました。IBM i 6.1 になる前から再ビルドされていない SQL プロシージャまたは関数がある場合は、改善されたコード生成機能が確実に反映されるように、プロシージャまたは関数をドロップして再作成することをお勧めします。

SQL プロシージャのパフォーマンスを改善するための別の簡単な操作として、PROGRAM TYPE SUB 節を使用する方法があります。これを省略したとき、または CREATE PROCEDURE (SQL) ステートメントで PROGRAM TYPE MAIN を使用したときには、そのプロシージャのために ILE C プログラム (*PGM) が作成されます。PROGRAM TYPE SUB を使用すると、そのプロシージャのために ILE C サービス・プログラム (*SRVPGM) が作成されます。PROGRAM TYPE SUB を使用する場合の影響が最も大きいのは、パフォーマンスが重要なアプリケーション内で頻繁に呼び出されるプロシージャの場合です。ILE サービス・プログラムは活動化グループごとに 1 回だけ活動化されますが、ILE プログラムは呼び出しごとに活動化されるため、PROGRAM TYPE SUB を使用するプロシージャのパフォーマンスの方が優れています。ILE 活動化のコストは、プロシージャのサイズ、複雑さ、パラメーターの数、変数の数、およびパラメーターと変数のサイズに関連しています。

PROGRAM TYPE SUB を使用するときには注意する必要がある唯一の機能上の違いは、EXTERNAL_NAME 列の QSYS2.SYSROUTINES カタログ項目にエクスポート名とサービス・プログラム名が共に表示されるようにフォーマット設定されることです。

プロシージャおよび関数のインプリメンテーションの向上

以下のコーディング技法は、関数またはプロシージャの処理時間を減らすのに役立ちます。

関数は多数の異なるプロシージャから複数呼び出される場合があるため、以下のヒントは特に関数にとって重要です。

- 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。
- 同一の入力と同じ結果を返すプロシージャおよび 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。例えば、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。例えば、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も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();
```

- SET ステートメントで 1 つの割り当てだけが実行される時、単純配列要素の割り当てを C コードで実装できます。

```
SET array_var[1] = 10, array_var[2] = 20;
```

これは、次のように書き直す必要があります。

```
SET array_var[1] = 10;  
SET array_var[2] = 20;
```

- 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、 unnecessary 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 CONCAT 'X';
```

この例を以下のように書き直します。

```
SELECT A CONCAT '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。例えば、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せん。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ください。

```
SET vnb_decimal = 4;  
cdecimal:  
  FOR vdec AS cdec CURSOR FOR  
  SELECT nb_decimal  
  FROM K$FX_RULES  
  WHERE first_currency=Pi_cur1 AND second_currency=Pi_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_cur1 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 *JOB RUN 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;
```

- 配列要素を設定または取得するために同一の属性を使用すると、整数、文字、varchar、10 進、および数値の各タイプに対して C コードが生成される場合があります。CCSID 処理が必要な文字変数では、SQL SET ステートメントの生成が必要な場合があります。
- 配列要素を使用する比較が C で生成されることはありません。比較の中には、比較で使用できるローカル変数に最初に要素値を割り当てておくとパフォーマンスが改善されるものもあります。

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、プロシージャまたは関数のパフォーマンスは十分でない場合があります。そのような場合には、プロシージャまたはユーザー定義関数 (UDF) の設計を調べて、パフォーマンスを向上させ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。

その設計変更には、さまざまなタイプがあります。

最初の変更は、プロシージャ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。例えば、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、効果的に使用するために特別な機能やロケータを必要とするデータ・タイプが少数あります。

レンジ・オブジェクト

レンジ・オブジェクト (LOB) とは、サイズが 0 バイトから 2 GB (GB = 1 073 741 824 バイト) のストリング・データ・タイプのことです。

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。また他にも音声、ビデオ、図面、テキストとグラフィックスの混合したもの、イメージなど、さまざまなデータ・タイプを保管しなければならない場合もあります。一部のデータ・タイプでは、これらのデータ・オブジェクトを、サイズが最大 2 GB のストリングとして保管できます。

この種のデータ・タイプは次のとおりです。バイナリー・レンジ・オブジェクト (BLOB)、1 バイト文字レンジ・オブジェクト (CLOB)、および 2 バイト文字レンジ・オブジェクト (DBCLOB) です。各表には、関連する LOB データを大量に含めることができます。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 LOB 値を収容するには大きさが足りないことがあるプログラムの記憶域を使用するので、他の方法でレンジ値を操作する必要が生じることがあります。ロケータは、データベース・サーバーでレンジ・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、レンジ・オブジェクト値 (あるいは、その大部分) を物理的に移動するのに便利です。

レンジ・オブジェクト・データ・タイプ

バイナリー・レンジ・オブジェクト (BLOB)、文字レンジ・オブジェクト (CLOB)、および 2 バイト文字レンジ・オブジェクト (DBCLOB) の定義を以下に示します。

- バイナリー・レンジ・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

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で、より大きな値を参照するのに使用します。

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・サーバーに保持されている 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:

259 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、統合ファイル・システムのファイルにまたは統合ファイル・システムのファイルから (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。

『例: CLOB 値を処理するためのロケータの使用』

アプリケーション・プログラムが文字ラージ・オブジェクト (CLOB) 値のロケータを取り出し、次にロケータを使用して CLOB 値からデータを抜き出すとします。

例: CLOB 値を処理するためのロケータの使用

アプリケーション・プログラムが文字ラージ・オブジェクト (CLOB) 値のロケータを取り出し、次にロケータを使用して CLOB 値からデータを抜き出すとします。

このメソッドを使用すると、プログラムは、CLOB データの一部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。

LOBLOC プログラム例の作動方法

このプログラム例には、CLOB ロケータ変数を処理する 3 つのセクションがあります。

1. **ホスト変数を宣言する。** CLOB ロケータ・ホスト変数が宣言されます。
2. **CLOB 値をフェッチしてロケータ・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 CLOB フィールドの位置を入手し、ロケータ・ホスト変数に入れます。
3. **CLOB ロケータを解放する。** この例で使用されている CLOB ロケータが解放され、以前に関連していた値からロケータを解放します。

関連概念:

255 ページの『ラージ・オブジェクト・ロケータ』

ラージ・オブジェクト (LOB) ロケータは、簡単に管理できる小さな値で、より大きな値を参照するのに使用します。

例: C の LOBLOC:

このプログラム例は、C で作成されており、ロケータを使用して CLOB 値を取り出します。

注: コード例を使用する場合は、398 ページの『コードに関するライセンス情報および特記事項』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>

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[]) {

    EXEC SQL BEGIN DECLARE SECTION;
        char number[7];
        long deptInfoBeginLoc;
        long deptInfoEndLoc;
        SQL TYPE IS CLOB_LOCATOR resume;      [1]
        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;
```



```

}
else if (argc == 3) {
    strcpy (userid, argv[1]);
    strcpy (passwd, argv[2]);
    EXEC SQL CONNECT TO sample USER :userid USING :passwd;
}
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;

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;

        /* Locate the beginning of "Education" section (end of "Dept.Info" */
        EXEC SQL VALUES (POSSTR(:resume, 'Education'))
            INTO :deptInfoEndLoc;

        /* Obtain ONLY the "Department Information" section by using SUBSTR */
        EXEC SQL VALUES (SUBSTR(:resume, :deptInfoBeginLoc,
            :deptInfoEndLoc - :deptInfoBeginLoc)) INTO :deptBuffer;

        /* Append the "Department Information" section to the :buffer var. */
        EXEC SQL VALUES (:buffer || :deptBuffer) INTO :buffer;
    } /* endif */
} while ( 1 );

printf ("%s %n", buffer);

EXEC SQL FREE LOCATOR :resume, :deptBuffer; [3]

EXEC SQL CLOSE c1;

EXEC SQL CONNECT RESET;
return 0;
}
/* end of program : LOBLOC */

```

例: COBOL の LOBLOC:

このプログラム例は、COBOL で作成されており、ロケータを使用して CLOB 値を取り出します。

注: コード例を使用する場合は、398 ページの『コードに関するライセンス情報および特記事項』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.
    EXEC SQL INCLUDE SQLCA END-EXEC.

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
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.  [1]
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.

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.

* 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 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]

    EXEC SQL CLOSE c1 END-EXEC.

    EXEC SQL CONNECT RESET END-EXEC.
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.

* Locate the beginning of "Education" section (end of Dept.Info)
EXEC SQL VALUES (POSSTR(:resume, 'Education'))
INTO :di-end-loc END-EXEC.

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.

* Append the "Department Information" section to the :buffer var
EXEC SQL VALUES (:buffer || :di-buffer) INTO :buffer
END-EXEC.

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、統合ファイル・システムのファイルにまたは統合ファイル・システムのファイルから（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 データを統合ファイル・システムの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・リクエスト・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:

255 ページの『ラージ・オブジェクト・ロケータ』

ラージ・オブジェクト (LOB) ロケータは、簡単に管理できる小さな値で、より大きな値を参照するのに使用します。

統合ファイル・システム

例: ファイルへの CLOB データの抽出

文字ラージ・オブジェクト (CLOB) 要素を表から外部ファイルに取り出す必要があるとします。

LOBFILE プログラム例の作動方法

このプログラム例には、CLOB ファイル参照変数を処理する 3 つのセクションがあります。

1. **ホスト変数を宣言する。**CLOB ファイル参照ホスト変数が宣言されます。この宣言は、ファイル名、ファイル名の長さ、およびファイル・オプションの宣言を含む構造として拡張されます。
2. **CLOB ファイル参照ホスト変数がセットアップされる。**ファイル参照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 ファイル参照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。

例: C の LOBFILE:

このプログラム例は、C で作成されており、表から外部ファイルに CLOB データを抽出します。

注: コード例を使用する場合は、398 ページの『コードに関するライセンス情報および特記事項』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>

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[]) {

    EXEC SQL BEGIN DECLARE SECTION;
    SQL TYPE IS CLOB_FILE resume;      [1]
    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;

    else if (argc == 3) {
        strcpy (userid, argv[1]);
        strcpy (passwd, argv[2]);
        EXEC SQL CONNECT TO sample USER :userid USING :passwd;

    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;
    return 0;
}
/* end of program : LOBFILE */

```

例: COBOL の LOBFILE:

このプログラム例は、COBOL で作成されており、表から外部ファイルに CLOB データを抽出します。

注: コード例を使用する場合は、398 ページの『コードに関するライセンス情報および特記事項』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.
    EXEC SQL INCLUDE SQLCA END-EXEC.

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
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. [1]
01 lobind          pic s9(4) comp-5.
    EXEC SQL END DECLARE SECTION END-EXEC.

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 "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.
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文字ラージ・オブジェクト (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 (例えば、物理ファイル・メンバーの表示 (DSPPFM)) を使用して、ラージ・オブジェクト (LOB) 列を収容している表の 1 つのデータ行を表示するときには、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。

例えば、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、ジャーナルされたラージ・オブジェクト (LOB) データへのアドレス可能性をユーザーに与えるバッファを戻します。

- 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としています。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

ユーザー定義の特殊タイプ (UDT) は、設定済みの組み込みデータ・タイプよりさらに DB2 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なります。

• 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:

14 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データ・タイプです。

UDT の定義

CREATE DISTINCT TYPE ステートメントを使用して、ユーザー定義タイプ (UDT) を定義します。

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。例えば、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:

231 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指定されたスキーマに制限します。

CREATE TYPE

例: 通貨:

異なる通貨を処理す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:

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

いくつかのユーザー定義タイプ (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。

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)
```

前述の例の UDT は、265 ページの『例: 通貨』で使用したのと同じ CREATE DISTINCT TYPE ステートメントを使用して作成されています。前述の例では、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:

応募者が記入したフォームを保持するための表を定義する必要があるとします。

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

ユーザー定義タイプ (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。例えば、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、例えば、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

以下に、さまざまな状態でユーザー定義タイプ (UDT) を使用する例を示します。

例: UDT と定数との比較:

米国で、1998 年の 7 月に US \$100 000.00 を超える総売上で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ユーザー定義関数 (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ため、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) を返し、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ユーザー定義関数 (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:

270 ページの『例: さまざまな UDT が関係する割り当て』

米ドルおよびカナダ・ドルの SUM をサポートする SUM 組み込み関数に、以下のソース派生ユーザー定義関数 (UDF) を定義したとします。

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 を使用して応募書式を格納する場合は、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 を使用して応募書式を格納する場合は、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:

269 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:

米ドルおよびカナダ・ドルの `SUM` をサポートする `SUM` 組み込み関数に、以下のソース派生ユーザー定義関数 (UDF) を定義したとします。

```
CREATE FUNCTION SUM (CANADIAN_DOLLAR)  
RETURNS CANADIAN_DOLLAR  
SOURCE SYSIBM.SUM (DECIMAL())  
  
CREATE FUNCTION SUM (US_DOLLAR)  
RETURNS US_DOLLAR  
SOURCE SYSIBM.SUM (DECIMAL())
```

ここで、上司の要求により、各製品の年間総売上高を国別に、米国ドルで、別々の表に維持管理する必要があります。

```

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:

269 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ユーザー定義関数 (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。

例: UDT、UDF、および LOB の使用

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方法について示します。

例: UDT と UDF の定義

会社へ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、例えば、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 for i データベース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 customers 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。ユーザー定義タイプ (UDT) がラージ・オブジェクト (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 (UDT のソース・タイプ) であるため、BLOB ロケータが 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。例えば、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済みます。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。例えば、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ため、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」と見なされます。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。

関連資料:

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。

NO LINK CONTROL:

NO LINK CONTROL を使用してデータ・リンク列を作成すると、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は検査されますが、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) 許可を使用した FILE LINK CONTROL を指定して作成されると、システムは、各データ・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となります。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) 許可を使用した FILE LINK CONTROL を指定して作成される
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、通常的手段
(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・リンクの処理

データ・リンクを処理するには、データ・リンク処理環境について理解する必要があります。

データ・リンクというデータ・タイプのサポートは、以下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 ENDTCPMSVR *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。

例えば、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、パスに有効なプレフィックスがあるので、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。

例えば、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 TESTDB または PRODDB ライブラリーの中の 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。以下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 を使用する

SQL は様々な環境で使用することができます。

カーソルの使用

SQL が SELECT 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:

122 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。

CLOSE

DECLARE CURSOR

DELETE

FETCH

UPDATE

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で使用されます。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。例えば、 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

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処理するためにプログラムに組み込める SQL ステートメントを示します。

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。シリアルまたはスクロール可能カーソルを使用すると、CORPDATA.EMPLOYEE 表からその部門に関する情報を取り出すことができます。

シリアル・カーソルの例では、プログラムは表から取り出したすべての行を処理して、部門 D11 の全社員の職種を更新し、他の部門からの社員のレコードを削除します。

表 50. 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
<pre>EXEC SQL DECLARE THISEMP CURSOR FOR SELECT EMPNO, LASTNAME, WORKDEPT, JOB FROM CORPDATA.EMPLOYEE FOR UPDATE OF JOB END-EXEC.</pre>	<p>282 ページの『ステップ 1: カーソルを定義する』.</p>
<pre>EXEC SQL OPEN THISEMP END-EXEC.</pre>	<p>283 ページの『ステップ 2: カーソルをオープンする』.</p>
<pre>EXEC SQL WHENEVER NOT FOUND GO TO CLOSE-THISEMP END-EXEC.</pre>	<p>283 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』.</p>
<pre>EXEC SQL FETCH THISEMP INTO :EMP-NUM, :NAME2, :DEPT, :JOB-CODE END-EXEC.</pre>	<p>284 ページの『ステップ 4: カーソルを用いて行を取り出す』.</p>
<p>... 部門 11 のすべての社員について、JOB の値を更新する。</p>	<p>285 ページの『ステップ 5a: 現在行を更新する』.</p>
<pre>EXEC SQL UPDATE CORPDATA.EMPLOYEE SET JOB = :NEW-CODE WHERE CURRENT OF THISEMP END-EXEC.</pre>	
<p>... 次にその行を印刷する。</p>	
<p>... 他の社員については、行を削除する。</p>	<p>285 ページの『ステップ 5b: 現在行を削除する』.</p>
<pre>EXEC SQL DELETE FROM CORPDATA.EMPLOYEE WHERE CURRENT OF THISEMP END-EXEC.</pre>	
<p>FETCH に戻り、次の行を処理する。</p>	
<pre>CLOSE-THISEMP. EXEC SQL CLOSE THISEMP END-EXEC.</pre>	<p>286 ページの『ステップ 6: カーソルをクローズする』.</p>

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 51. 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
<pre>EXEC SQL DECLARE THISEMP DYNAMIC SCROLL CURSOR FOR SELECT EMPNO, LASTNAME, SALARY FROM CORPDATA.EMPLOYEE WHERE WORKDEPT = 'D11' END-EXEC.</pre>	『ステップ 1: カーソルを定義する』.
<pre>EXEC SQL OPEN THISEMP END-EXEC.</pre>	283 ページの『ステップ 2: カーソルをオープンする』.
<pre>EXEC SQL WHENEVER NOT FOUND GO TO CLOSE-THISEMP END-EXEC.</pre>	283 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』.
<p>...プログラムの合計給与変数を初期設定する。</p> <pre>EXEC SQL FETCH RELATIVE 3 FROM THISEMP INTO :EMP-NUM, :NAME2, :JOB-CODE END-EXEC.</pre> <p>...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。</p>	284 ページの『ステップ 4: カーソルを用いて行を取り出す』.
<p>...平均給与を計算する。</p> <pre>CLOSE-THISEMP. EXEC SQL CLOSE THISEMP END-EXEC.</pre>	286 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:

結果表にアクセスするカーソルを定義するには、DECLARE CURSOR ステートメントを使用します。

DECLARE CURSOR ステートメントでは、カーソルの名前を指定し、選択ステートメントを指定します。選択ステートメントでは、結果表を (概念上) 構成する行のセットを定義します。シリアル・カーソルの場合、ステートメントは次のようになります (FOR UPDATE OF 文節は任意指定です)。

```
EXEC SQL
  DECLARE カーソル名 CURSOR FOR
  SELECT 列 1, 列 2 ,...
  FROM 表名 , ...
  FOR UPDATE OF 列 2 ,...
END-EXEC.
```

スクロール可能カーソルの場合、ステートメントは次のようになります (WHERE 文節は任意指定です)。

```
EXEC SQL
  DECLARE カーソル名 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。

例えば、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。

結果表とカーソルが読み取り専用である時点に関する情報については、SQL 解説書のトピック集の DECLARE CURSOR を参照してください。

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:

データの終わり条件は、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 ステートメントが発行された場合に起こります。

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。

例えば、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、カーソルをクローズ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行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、WHERE CURRENT OF 文節を指定した 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。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・カーソルを使用して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 (データベース・ファイル一時変更 (OVRDBF) コマンド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 32 767 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、ヌル値可能な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な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。

関連概念:

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 ステートメント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、398 ページの『コードに関するライセンス情報および特記事項』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。

関連資料:

SQLCA (SQL 通信域)

行記憶域を使用した複数行 FETCH:

複数行用 FETCH ステートメントを行記憶域とともに使用するには、その前にアプリケーションで、行記憶域および関連した記述域を定義しなければなりません。

行記憶域とは、アプリケーション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 ステートメントの結果が入ります。行記憶域は、複数行用 FETCH ステートメント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、398 ページの『コードに関するライセンス情報および特記事項』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;
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 に入っている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:

368 ページの『DB2 for i サンプル表』

以下のサンプル表は、「SQL プログラミング」、および「SQL 解説書」トピック集で参照および使用されています。

SQLDA (SQL 記述域)

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 文節は無視されません)。

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: SQL の作成 (CRTSQLxxx) コマンドで ALWBLK(*ALLREAD) パラメーターを指定すると、読み取り専用カーソルのカーソル位置の復元を変更することができます。CRTSQLxxx コマンドで ALWBLK パラメーターおよびその他のパフォーマンス関連オプションを使用する方法の詳細については、『動的 SQL アプリケーション』を参照してください。

関連概念:

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るか、または SQL ステートメントを作成します。アプリケーションは、SQL ステートメントのタイプ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。

注:

- 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、ポインター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、ポインター変数の管理のために 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:

308 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:

SQL ステートメントで許可されるアクション

Process Extended Dynamic SQL (QSQRCD) API

291 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。

動的 SQL ステートメントの実行

動的 SQL ステートメントは、プリコンパイル時に準備されず、実行時に準備されるので、動的 SQL ステートメントを出すためには、EXECUTE ステートメントまたは EXECUTE IMMEDIATE ステートメントを使用します。

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。

Open Database Connectivity (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。

関連概念:

System i 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。ホスト変数のコード化文字セット ID (CCSID) が、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。例えば、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、その前にこの SQL ステートメントが動的に実行可能であることを確認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:

308 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には、PREPARE および EXECUTE ステートメントを使用して実行する必要があります。

PREPARE ステートメントは非 SELECT ステートメント (例えば、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。

SELECT ステートメントの処理と記述子の使用

SELECT ステートメントの基本タイプは、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します。この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、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。

例えば、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、実行結果の列の数および様式が予測できない場合、可変リスト SELECT ステートメントが使用されます。すなわち、必要とする変数のデータ・タイプまたは数が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 がないときは、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:

300 ページの『SQLDA の記憶域を割り振るための SELECT ステートメントの例』アプリケーションが、一度使用して次に使用するまでに変更される動的 SELECT ステートメントを処理する必要があります。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。

記述子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、記述子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・マーカに関する情報を提供します。DESCRIBE TABLE では、記述子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、記述子はホスト変数に関する情報を提供します。例えば、DESCRIBE ステートメントを使用して記述子に値を読み取り、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、記述子をコーディングすることができます。

記述子には 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、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。

関連資料:

SQLCA (SQL 通信域)

SQLDA (SQL 記述域)

SQLDA の形式:

SQL 記述子域 (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。IBM i では、ポインタは 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 が使用される時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へのポインタが入ります。ローケータが 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 発生数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る直前の 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。

関連タスク:

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

関連資料:

300 ページの『SQLDA の記憶域を割り振るための SELECT ステートメントの例』アプリケーションが、一度使用して次に使用するまでに変更される動的 SELECT ステートメントを処理する必要があるとします。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。

SQLDA の記憶域を割り振るための SELECT ステートメントの例:

アプリケーションが、一度使用して次に使用するまでに変更される動的 SELECT ステートメントを処理する必要があるとします。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があります。

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することが必要です。

例えば、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。

表 52. 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 53. SQLVAR 要素 1

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

表 54. 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あることがあります。例えば、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 (例えば、SUM(SALARY))、式 (例えば、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 55. SQLDA ヘッダー

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

表 56. 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 57. 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:

295 ページの『可変リスト SELECT ステートメント』

動的 SQL では、実行結果の列の数および様式が予測できない場合、可変リスト SELECT ステートメントが使用されます。すなわち、必要とする変数のデータ・タイプまたは数が分かっていない場合です。

297 ページの『SQLDA の形式』

SQL 記述子域 (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 SELECT ステートメント:

アプリケーションが、一度使用して次に使用するまでに変更される動的 SELECT ステートメントを処理する必要があります。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。

例えば、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が、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タ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、各記述子エンタ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;
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 などがある一例です。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:

パラメーター・マーカーは、動的ステートメント・ストリング中の疑問符 (?) で表されます。この疑問符は、ステートメント・ストリングが静的 SQL ステートメントであるとすれば、ホスト変数が来ることになる位置を示すことができます。

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。

```
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 を使用するときには、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。

関連資料:

300 ページの『SQLDA の記憶域を割り振るための SELECT ステートメントの例』

アプリケーションが、一度使用して次に使用するまでに変更される動的 SELECT ステートメントを処理する必要がありますとします。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。

304 ページの『例: 割り振られた SQL 記述子を使用した SELECT ステートメント』

アプリケーションが、一度使用して次に使用するまでに変更される動的 SELECT ステートメントを処理する必要がありますとします。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。

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、サポートされる 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、使用を許可されたりレシヨナル・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。

関連概念:

291 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るか、または SQL ステートメントを作成します。アプリケーションは、SQL ステートメントのタイプを知る必要はありません。

関連資料:

293 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、その前にこの SQL ステートメントが動的に実行可能であることを確認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、IBM i コマンド行から STRSQL と入力します。

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

SQL ステートメントの入力

SQL ステートメントを入力して、実行キーを押してください。
現在の接続相手は、リレーショナル・データベース RDJACQUE である。

====>

終わり

F3=終了 F4=プロンプト F6=行の挿入 F9=コマンドの複写 F10=行のコピー
F12=取り消し F13=サービス F24=キーの続き

F24 (キーの続き) を押すことにより、残りの機能キーが表示されます。

終わり

F14=行の削除 F15=行の分割 F16=コレクションの選択 (ライブラリー)
F17=テーブルの選択 F18=列の選択 F24=キーの続き
(ファイル) (フィールド)

注: システム命名規則を使用している場合は、上記の名称の代わ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。

関連資料:

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 を選択するときユーザーが最初に使用する機能です。各対話式 SQL ステートメントの処理が終わるたびに、ステートメント入力機能に戻ります。

ステートメント入力機能では、ユーザーは 1 つの SQL ステートメント全体を入力するか、プロンプトを使用して入力した上で、実行キーを押すと、そのステートメントは処理のために送られます。

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。プロンプトが使用できない SQL ステートメントもあり、多くの SQL ステートメントでは完全ではありません。`*SYS` 命名モードで名前を修飾するためにピリオド (`.`) を使用するステートメントをプロンプトした場合、ピリオドはスラッシュ (`/`) に変更されます。`GRANT` および `REVOKE` ステートメントでは、`*SYS` 命名でピリオドを使用している場合にプロンプトがサポートされません。

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 テーブル _____
SELECT 列 _____
WHERE 条件 _____
GROUP BY 列 _____
HAVING 条件 _____
ORDER BY 列 _____
FOR UPDATE OF 列 _____

終わり

選択項目を入力して、実行キーを押してください。

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

F3=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= 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。

プロンプト中に実行キーを押すと、プロンプト画面で作成したステートメントがセッションに挿入されます。ステートメント処理モードが *RUN の場合は、そのステートメントが実行されます。エラーを検出した場合は、プロンプト機能に制御権が残ります。

構文検査:

SQL ステートメントの構文は、そのステートメントがプロンプト機能に入ったときに検査されます。

プロンプト機能は構文が正しくないステートメントを受け付けません。構文を訂正するかまたはステートメントの正しくない部分を削除しないと、プロンプトは使用できません。

ステートメント処理モード:

ステートメント処理モードは、「セッション属性変更」画面で選択することができます。

*RUN (実行) モードまたは *VLD (妥当性検査) モードでは、プロンプトを出すことができるのは、対話式 SQL で実行できるステートメントに限ります。*SYN (構文検査) モードでは、すべての SQL ステートメントを使用することができます。ステートメントは、*SYN モードや *VLD モードでは実際には実行されません。構文とオブジェクトの存在だけが検査されます。

副照会:

副照会は、WHERE 文節または HAVING 文節が表示されている任意の画面で選択することができます。

副照会画面を表示するには、カーソルが WHERE または HAVING の入力行に置かれているときに F9 (副照会の指定) キーを押してください。部分選択情報を入力できる画面が表示されます。F9 を押したときカーソルが副照会の括弧の中にあったときは、副照会情報は次に表示される画面で埋められます。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 ステートメントのプロンプトが出されたときは、列定義を個別に入力できます。

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 2 バイト文字セット (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 条件が入力された想定します。シフト文字が、この画面の 2 つの各行のストリング・セクションの始まりと終わりに表示されています。

SELECT ステートメントの指定

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

FROM テーブル	TABLE1	_____
SELECT 列	*	_____
WHERE 条件	COL1 = '<AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQ>	_____
		<RRSS>'	_____
GROUP BY 列		_____
HAVING 条件		_____
ORDER BY 列		_____
FOR UPDATE OF 列		_____

実行キーを押すと、文字ストリングが連結され、余分なシフト文字が除去されます。このステートメントは、「SQL ステートメントの入力」画面で次のように表示されます。

```
SELECT * FROM TABLE1 WHERE COL1 = '<AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSS>'
```

リスト選択機能の使用

特定のプロンプト画面で F4 (プロンプト) を押すと、リスト選択機能にアクセスできます。「SQL ステートメントの入力」画面でこの機能にアクセスするには、F16 (コレクションの選択)、F17 (表の選択)、または F18 (列の選択) 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、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。例えば、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 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。

関連資料:

309 ページの『対話式 SQL の開始』

対話式 SQL の使用を開始するには、IBM i コマンド行から STRSQL と入力します。

例: リスト選択機能の使用:

この例は、リスト選択機能を使用して 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。

- IBM i コマンド行で STRSQL とタイプ入力して、対話式 SQL に入ったばかりである。
- まだ、リストの選択も入力も行っていない。
- 命名規則として *SQL を選択した。

注: この例は、ユーザーのサーバーにないリストを示しています。これらは、一例として使用されているに過ぎません。

次のようにして SQL ステートメントの使用を開始します。

1. 最初のステートメント入力行に SELECT と入力します。
2. 2 番目のデータ入力行に FROM と入力します。
3. カーソルは FROM の後の位置に置いたままにしておきます。

SQL ステートメントの入力

SQL ステートメントを入力して、実行キーを押してください。
====> SELECT
 FROM _

4. FROM の後には表名を指定したいので、表のリストを得るために F17 (表の選択) キーを押してください。

希望した表のリストは表示されず、コレクションのリストが表示されます (「コレクションの選択および順序づけ」画面)。SQL セッションに入ったばかりで、処理の対象とするスキーマをまだ選択していないためです。

5. YOURCOLL2 スキーマの横の SEQ 列に 1 を入力してください。

コレクションの選択および順序づけ

コレクションを選択するには順序番号 (1-999) を入力して実行キーを押してください。

SEQ	コレクション	タイプ	テキスト
	YOURCOLL1	SYS	会社の利益
1	YOURCOLL2	SYS	社員のパーソナル・データ
	YOURCOLL3	SYS	ジョブの分類/要件
	YOURCOLL4	SYS	会社の保険

6. 実行キーを押します。

「テーブルの選択および順序づけ」画面が表示され、YOURCOLL2 スキーマの中にある表 (テーブル) が表示されます。

7. PEOPLE 表の横の SEQ 列に 1 を入力します。

テーブルの選択および順序づけ

テーブルを選択するためには、順序番号 (1-999) を入力して実行キーを押してください。

SEQ	テーブル	コレクション	タイプ	テキスト
	EMPLCO	YOURCOLL2	TAB	社員の会社データ
1	PEOPLE	YOURCOLL2	TAB	社員の個人データ
	EMPLEXP	YOURCOLL2	TAB	社員の経歴
	EMPLEVL	YOURCOLL2	TAB	社員の査定報告書
	EMPLBEN	YOURCOLL2	TAB	社員の給付金記録
	EMPLMED	YOURCOLL2	TAB	社員の健康診断記録
	EMPLINVST	YOURCOLL2	TAB	社員の出資記録

8. 実行キーを押します。

「SQL ステートメントの入力」画面が再び現れ、FROM の後 YOURCOLL2.PEOPLE という表名が表示されます。表名は *SQL 命名規則でスキーマ名により修飾されます。

SQL ステートメントの入力

SQL ステートメントを入力して、実行キーを押してください。
 ==> SELECT
 FROM YOURCOLL2.PEOPLE _

9. SELECT の後にカーソルを置きます。
 10. SELECT の後に列名を指定したいので、列のリストを表示するために F18 (列の選択) キーを押します。

「列の選択および順序づけ」画面が表示され、PEOPLE 表の中にある列が表示されます。

11. NAME 列の横の SEQ 列に 2 を入力します。
 12. SOCSEC 列の横の SEQ 列に 1 を入力します。

列の選択および順序付け

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

SEQ	列	テーブル	タイプ	長さスケール
2	NAME	PEOPLE	CHARACTER	6
	EMPLNO	PEOPLE	CHARACTER	30
1	SOCSEC	PEOPLE	CHARACTER	11
	STRADDR	PEOPLE	CHARACTER	30
	CITY	PEOPLE	CHARACTER	20
	ZIP	PEOPLE	CHARACTER	9
	PHONE	PEOPLE	CHARACTER	20

13. 実行キーを押します。

「SQL ステートメントの入力」画面が再び現れ、SELECT の後に、SOCSEC、NAME が表示されます。

SQL ステートメントの入力

SQL ステートメントを入力して、実行キーを押してください。
 ==> SELECT SOCSEC, NAME
 FROM YOURCOLL2.PEOPLE

14. 実行キーを押します。

作成したステートメントはこれで実行されます。

いったんリスト機能を使用すると、選択した値は、その値を変更するか、あるいは「セッション属性の変更」画面でスキーマのリストを変更するまで、引き続き効力があります。

セッション・サービスの説明

「Session Services (セッション・サービス)」画面からセッション属性を変更できます。また、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。

「セッション・サービス (Session Services)」画面にアクセスするには、「SQL ステートメントの入力」画面で、F13 (サービス) を押します。

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tring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ず。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 (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

対話式 SQL 環境を終了するには、「SQL ステートメントの入力」画面で、F3 (終了) を押します。終了するには、いくつかのオプションを使用できます。

- 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 (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、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) を選択して対話式 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。

例えば、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ています。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してみます。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、SQL 対話式セッションの開始 (STRSQL) コマンドを入力したときに、対話式 SQL は、次のセッションの開始時に「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・リレーショナル・データベースに対する暗黙の接続を確立します。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。対話式 SQL が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、または対話式 SQL が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メント制御レベルが *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・データベースに接続する前に、以下のいずれかのタスクを実行することをお勧めします。

- 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 ステートメント、およびその後続けて COMMIT ステートメントを発行し、ローカル・データベース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。
- DB2 for i 以外のアプリケーション・サーバーに接続するときは、RELEASE ALL ステートメント、およびその後続けて COMMIT ステートメントを発行し、ローカル・データベース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 for i 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 58. セッション属性の値

セッション属性	元の値	新しい値
日付の形式	*YMD *DMY *MDY *JUL	*ISO *EUR *USA *USA
時刻形式	*HMS (区切り記号 (:) 付き) *HMS (他の区切り記号付き)	*JIS *EUR
コミットメント制御	*CHG, *NONE *ALL	*CS 反復可能読み取り
命名規則	*SYS	*SQL
データ・コピー使用可	*NO, *YES	*OPTIMIZE
データ再表示	*ALWAYS	*FORWARD
小数点	*SYSVAL	*PERIOD
分類順序	*HEX 以外の値	*HEX

注: DB2 for Linux, UNIX, and Windows または DB2 for z/OS 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 DB2 for i 以外のアプリケーション・サーバーにリモート・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。

関連概念:

分散データベース・プログラミング

関連資料:

340 ページの『接続タイプの決定』

リモート SQL 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。

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・メンバーまたはソース・ストリーム・ファイルから実行することができます。ソース中のステートメントは、ソース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 FUNCTION
- | • ALTER MASK
- | • ALTER PERMISSION
- ALTER PROCEDURE
- ALTER SEQUENCE
- ALTER TABLE
- | • ALTER TRIGGER
- CALL
- COMMENT
- COMMIT
- | • 複合 (動的)
- CREATE ALIAS
- CREATE FUNCTION
- CREATE INDEX
- | • CREATE MASK
- | • CREATE PERMISSION
- CREATE PROCEDURE
- CREATE SCHEMA
- CREATE SEQUENCE
- CREATE TABLE
- CREATE TRIGGER
- CREATE TYPE
- CREATE VARIABLE

- CREATE VIEW
- DECLARE GLOBAL TEMPORARY TABLE
- DELETE
- DROP
- GRANT
- INSERT
- LABEL
- LOCK TABLE
- MERGE
- REFRESH TABLE
- RELEASE SAVEPOINT
- RENAME
- REVOKE
- ROLLBACK
- SAVEPOINT
- SET CURRENT DECFLOAT ROUNDING MODE
- SET CURRENT DEGREE
- SET CURRENT IMPLICIT XMLPARSE OPTION
- SET ENCRYPTION PASSWORD
- SET PATH
- SET SCHEMA
- SET TRANSACTION
- | • TRANSFER OWNERSHIP
- | • TRUNCATE
- UPDATE

ソース中で、SQL ステートメントは EXEC SQL では始まりません。各ステートメントはセミコロンで終了します。ソース・メンバーの場合、デフォルトの右マージンは 80 です。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。RUNSQLSTM コマンドで MARGINS パラメーターを使用すると、右マージンを他の値に変更できます。ソース・ストリーム・ファイルの場合、ファイル全体が読み取られます。マージンは使用されません。

ソース中の注釈は、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、対応する */ 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、CL コマンド、およびコマンドです。CL コマンドには、接頭部として「CL:」が必要です。例えば、次の通りです。

```
CL: ADDLIB MYLIB;
INSERT INTO T1 VALUES('A', 17);
```

SQL ステートメントおよび CL コマンドの処理結果のメッセージを含む出力リストが、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 QSYSVRT です。

- | OPTION パラメーターにより、出力リストを取得するか、ジョブ・ログにエラーを出力するかを選択できます。処理中にエラーが発生したときにのみリストを生成するオプションもあります。

ソース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。リスト中のエラー・メッセージに関する詳細情報を参照するには、SECLVLTXT(*YES) パラメーターを指定してください。

関連資料:

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、SQL ステートメント実行 (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 SQL ステートメント実行 (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・リスト

この例は、SQL ステートメント処理プログラムの出力ソース・リストを示します。

注: コード例を使用する場合は、398 ページの『コードに関するライセンス情報および特記事項』のご使用条件に同意する必要があります。


```

xxxxSS1 VxRxMx yymmdd      SQL ステートメントの実行      SCHEMA      02/15/08 15:35:18  PAGE  1
ソース・ファイル.....CORPDATA/SRC
メンバー.....SCHEMA
コミット.....*NONE
名前指定.....*SYS
生成レベル.....10
日付の形式.....*JOB
日付区切り記号.....*JOB
右マージン.....80
時刻の形式.....*HMS
時刻区切り記号.....*JOB
省略時のコレクション.....*NONE
IBM SQL フラグづけ.....*NOFLAG
ANS フラグづけ.....*NONE
小数点.....*JOB
ソート順序.....*JOB
言語 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.....37
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.....*ALLREAD
SQL 規則.....*DB2
10 進数結果オプション:
  最大精度.....31
  最大位取り.....31
  除算の最小位取り.....0
11/01/07 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

```

xxxxSS1 VxRxMx yymmdd      SQL ステートメントの実行      SCHEMA      02/15/08 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 SCHEMA DEPT;
3  DROP SCHEMA MANAGER;
4
5  CREATE SCHEMA DEPT
6      CREATE TABLE EMP (EMPNAME CHAR(50), EMPNBR INT)
7      -- スキーマ 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 *****

```

図 2. SQL ステートメント処理プログラムの QSYSPRT リスト (続き)

```

xxxxSS1 VxRxMx yymmdd      SQL ステートメントの実行      SCHEMA      02/15/08 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 がスキーマ 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。
      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  打ち切り
   13   13    0    0    0    0
ソースに 00 レベルの重大度エラーが見つかった。
***** リストの終わり *****

```

図 3. SQL ステートメント処理プログラムの QSYSPRT リスト (続き)

RUNSQL CL コマンドの使用

RUNSQL CL コマンドにより、ソース・ファイルなしで SQL ステートメントを制御言語プログラム内から実行できます。

以下のステートメントを RUNSQL で使用できます。

- ALTER FUNCTION
- ALTER MASK
- ALTER PERMISSION
- ALTER PROCEDURE
- ALTER SEQUENCE
- ALTER TABLE
- ALTER TRIGGER
- CALL
- COMMENT
- COMMIT
- 複合 (動的)
- CREATE ALIAS
- CREATE FUNCTION
- CREATE INDEX
- CREATE MASK
- CREATE PERMISSION
- CREATE PROCEDURE
- CREATE SCHEMA
- CREATE SEQUENCE
- CREATE TABLE

- | • CREATE TRIGGER
- | • CREATE TYPE
- | • CREATE VARIABLE
- | • CREATE VIEW
- | • DECLARE GLOBAL TEMPORARY TABLE
- | • DELETE
- | • DROP
- | • GRANT
- | • INSERT
- | • LABEL
- | • MERGE
- | • REFRESH TABLE
- | • RELEASE SAVEPOINT
- | • RENAME
- | • REVOKE
- | • ROLLBACK
- | • SAVEPOINT
- | • SET CURRENT DECFLOAT ROUNDING MODE
- | • SET CURRENT DEGREE
- | • SET CURRENT IMPLICIT XMLPARSE OPTION
- | • SET ENCRYPTION PASSWORD
- | • SET PATH
- | • SET SCHEMA
- | • SET TRANSACTION
- | • TRANSFER OWNERSHIP
- | • TRUNCATE
- | • UPDATE

| ステートメント・ストリングの長さは、最大で 5000 文字にすることができます。セミコロンで終了しては
| なりません。

| ステートメント・ストリングで注釈を使用できます。行の注釈は 2 重のハイフン (--) で開始し、その行の
| 末尾 (復帰/改行) またはストリングの末尾で終了します。ブロックの注釈は /* で開始し、対応する */ に
| 到達するまで続きます。ブロックの注釈はネストすることができます。

| RUNSQL でファイルを開いた場合、そのファイルは、制御が呼び出し元に返される前に閉じられます。コ
| ミットメント制御が活動状態の場合、ユーザーのアプリケーションでコミットまたはロールバックを実行す
| る必要があります。

| コマンドは、呼び出し元の活動化グループで実行されます。コンパイル済みの制御言語プログラムに
| RUNSQL が含まれている場合、プログラムの活動化グループが使用されます。

| 出力リストは生成されません。障害が発生した場合、SQL メッセージがエスケープ・メッセージとして呼
| び出し元に送信されます。構文エラーを返す複雑な SQL ステートメントの場合、データベース・モニター

| を使用して、エラーの原因の検出に役立てることができます。データベース・モニターを開始し、RUNSQL
| コマンドを実行し、System i ナビゲーター を使用してデータベース・モニターを分析します。

| 以下のように、CL から INSERT ステートメントを実行します。

```
| RUNSQL SQL('INSERT INTO prodLib/work_table VALUES(1, CURRENT TIMESTAMP)')
```

| 制御言語プログラムでは、ファイル受信 (RCVF) コマンドを使用して、以下の照会用に生成された表の結
| 果を読み取ることができます。

```
| RUNSQL SQL('CREATE TABLE qtemp.worktable AS  
| (SELECT * FROM qsys2.systables WHERE table_schema = 'MYSCHEMA') WITH DATA')  
| COMMIT(*NONE) NAMING(*SQL)
```

| 以下のように、ステートメントの一部として入力パラメーターを使用して、SQL ステートメントを構成し
| て実行する制御言語プログラムを作成します。

```
| RUNSQL1: PGM PARM(&LIB)  
| DCL &LIB TYPE(*CHAR) LEN(10)  
| DCL &SQLSTMT TYPE(*CHAR) LEN(1000)  
| CHGVAR VAR(&SQLSTMT) +  
| VALUE('DELETE FROM qtemp.worktable1 +  
| WHERE table_schema = '' || &LIB || ''')  
| RUNSQL SQL(&SQLSTMT) COMMIT(*NONE) NAMING(*SQL)  
| RUNSQL1: ENDPGM
```

分散リレーショナル・データベース関数と 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、(例えば、DB2 for i データベース) またはタイプが異なる場合 (DB2 for z/OS、DB2 for VSE and VM、また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 for i は 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 for i 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 for i は以下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 for i は、拡張プログラム間通信機能 (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 for i は DUW over APPC および DUW over TCP/IP をサポートします。

関連概念:

4 ページの『DB2 for i 構造化照会言語の紹介』

構造化照会言語 (SQL) は、リレーショナル・データベースのデータの定義と操作を行うための標準化された言語です。これらのトピックでは、DB2 for i データベースと IBM DB2 Query Manager and SQL Development Kit for i ライセンス・プログラムを使用した SQL の IBM i 実装について説明します。

14 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。

分散データベース・プログラミング

DB2 for i 分散リレーショナル・データベース・サポート

IBM DB2 Query Manager and SQL Development Kit for i ライセンス・プログラムは、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。

DB2 Query Manager and SQL Development Kit for IBM i は、以下の SQL ステートメントにより、分散リレーショナル・データベース・サポートを提供します。

- 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せん。

- 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

関連タスク:

SQL ステートメントを含むプログラムの準備と実行

関連資料:

398 ページの『DB2 for i CL コマンドの記述』

DB2 for i は SQL のための以下の CL コマンドを提供します。

CONNECT (タイプ 1)
CONNECT (タイプ 2)
DISCONNECT
DROP
GRANT (パッケージ特権)
REVOKE (パッケージ特権)
RELEASE (接続)
SET CONNECTION

DB2 for i 分散リレーショナル・データベース・プログラム例

リモート作業単位を使用するリレーショナル・データベース・プログラム例は、SQL プロダクトと共に出荷されます。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 for i のプログラム例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、分散アプリケーション例のスキーマ、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。

プログラム例を使用する場合は、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、そこでプログラム例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
```

次の画面がオープンします。この画面では、ユーザーのデータベース・プログラム例をカスタマイズすることができます。

SAMPLE 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.....:

F3=EXIT

SQL パッケージ・サポート

IBM i オペレーティング・システムは、SQL パッケージをサポートします。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。IBM i オペレーティング・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。

関連資料:

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 for i を使用してコンパイルされ、DB2 for i でないリモート・データベースを参照しているプログラムは、そのリモート・データベース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。例えば、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 OS/400® V5R2 または IBM i V5R3 以降で稼働している分散プログラム内でしか使用できません。IBM i オペレーティング・システム上で実行していないアプリケーション・リクエスターは、IBM i V5R3 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 ステートメント、ブロック化 INSERT ステートメント、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、バイナリー・ラージ・オブジェクト (BLOB)、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) データの送信が許可されません。

関連情報:

SQL ステートメントの特性

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は、以下の側面を考慮してください。

CRTSQLPKG 権限:

SQL パッケージを IBM i オペレーティング・システム上に作成するときは、使用する権限 ID は SQL パッケージの作成 (CRTSQLPKG) コマンドに対する *USE 権限を持つ必要があります。

DB2 for i 以外のデータベースでのパッケージの作成:

DB2 for i 以外のデータベースにプログラムか SQL パッケージを作成して、その他の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、SQL の作成 (CRTSQLxxx) コマンドの 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。例えば、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 値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 プロダクト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 for i でないデータベースを指定する場合は、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(*JOB RUN)
- SRTSEQ(*LANGIDUNQ)
- SRTSEQ(*LANGIDSHR)
- SRTSEQ(ライブラリー名/表名)

注: DB2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:

パッケージを作成する際に、どのリリースがその機能をサポートできるか判断するために 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。例えば、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中に、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時、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。

例えば、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使用される接続のタイプは、RDBCNNMTH パラメーターで指定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 (オブジェクト保管 (SAVOBJ) コマンド) から、別のシステムに復元する (オブジェクト復元 (RSTOBJ) コマンド) ことができます。プログラムを実行させるとき、ローカル・システムに接続されていると、SQL パッケージは使用されません。RDB パラメーターに *LOCAL を指定すると、*SQLPKG オブジェクトは作成されませんが、パッケージ情報は *PGM オブジェクトに保管されます。

ラベル:

LABEL ON ステートメントを使用すると、SQL パッケージの記述を作成することができます。

整合性トークン:

プログラムとその関連 SQL パッケージに整合性トークンがあり、これは、SQL パッケージの呼び出しを行うときに検査されます。

整合性トークンが一致しないとパッケージを使用することはできません。プログラムと SQL パッケージとが整合性がないように見えることがあります。プログラムとアプリケーション・サーバーが、2 つの異なる IBM i オペレーティング・システム上にあるとします。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となります。

SQL の作成 (CRTSQLxxx)、SQL パッケージの作成 (CRTSQLPKG)、および SQL 対話式セッションの開始 (STRSQL) コマンドと 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 IBM i プロダクトでないときは、IBM i プラットフォーム上のサーバーは、ジョブのコード化文字セット ID (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。これは分散リレーショナル・データベース・アーキテクチャー (DRDA) で必要になるためです。リモート・システムが IBM i プロダクトにあるときは、これらの名前は CCSID 500 からジョブの 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-blc" 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-blc" 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。
- ー 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-blc." 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。

分散リレーショナル・データベース・アーキテクチャー (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。つまり、接続は、リレーショナル・データベース (RDB) に対して CONNECT TO ステートメントが実行された時点で開始し、DISCONNECT または RELEASE ALL ステートメントに続いて正常な COMMIT 操作が実行された時点で終了します。拡張プログラム間通信機能 (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 IBM i プロダクト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4. 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5. 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;
...

```

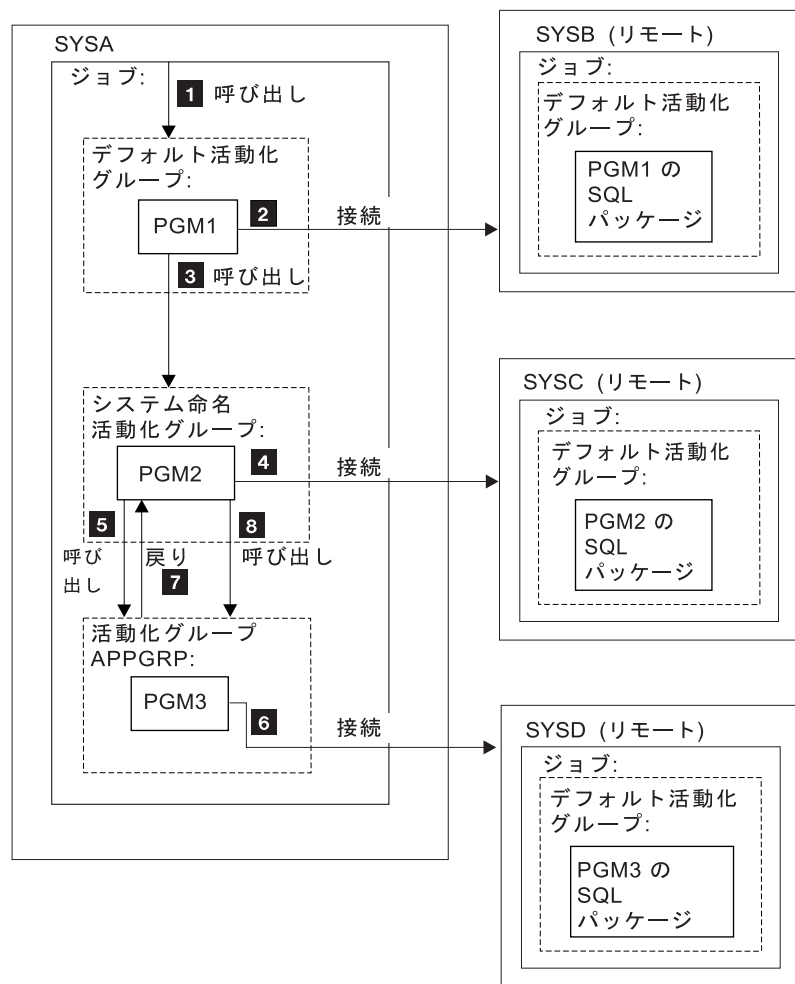
図6. PGM3 のソース・コード

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)

```



RV2W577-3

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、

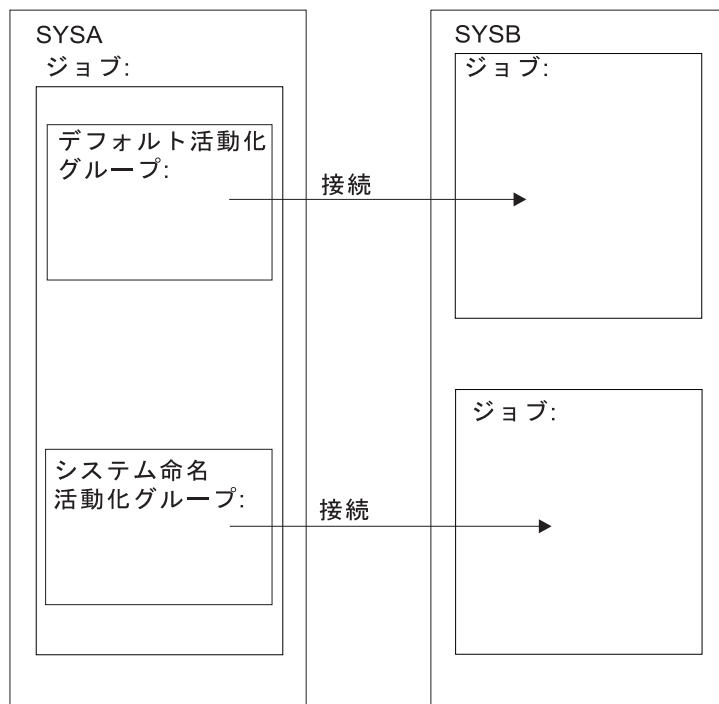
両方の接続が活動状態になります。PGM2 はカーソルをオープンし、行を取り出して更新します。PGM2 はコミットメント制御の下で実行中で、作業単位内にあり、接続可能状態ではありません。

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。



RV2W578-2

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・リクエスターをアプリケーション・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
 - 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:

345 ページの『接続の終了』

リモート SQL 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・リクエスターをアプリケーション・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:

345 ページの『接続の終了』

リモート SQL 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 for i は以下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。

接続タイプの決定

リモート SQL 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。
 - 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 59. 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・サーバーが、2 フェーズ・コミットをサポートする。	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。	その他の更新可能な 1 フェーズ・リソースが登録されている。	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

表 59. 接続タイプの要約 (続き)

コミットメント制御下の接続	アプリケーション・サーバーが、2 フェーズ・コミットをサポートする。	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。	その他の更新可能な 1 フェーズ・リソースが登録されている。	SQLERRD(4)
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

¹DRDA 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 IBM i TCP/IP 接続が含まれます。

関連概念:

コミットメント制御

関連資料:

319 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。

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を確立できるかについては、制約事項があります。これらの制約事項は、接続上で COMMIT(*NONE) を指定した場合を除いて、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 60. 接続状況の値の決定に関する要約

接続方法	更新可能なり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、以下の点を考慮してください。

- 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 (例えば、TCP/IP を使用する V4R5 システム) 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 以降、分散リレーショナル・データベース・アーキテクチャー (DRDA) 2 フェーズ・コミット操作には、以下の更新が考慮事項に入れられています。
 - トリガーの結果として行われる更新
 - データベース照会中に活動化されるユーザー定義関数の結果として行われる更新

接続の終了

リモート SQL 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、分散データ管理 (DDM) ネットワーク接続 (拡張プログラム間通信機能 (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 IBM i プラットフォーム上の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 IBM i プラットフォーム上のサーバーである場合は、接続は除去されます。接続先が IBM i プラットフォーム以外である場合は、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う失敗します。
- 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:

338 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・リクエスターをアプリケーション・サーバーと暗黙に接続することができます。

339 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・リクエスターをアプリケーション・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、分散作業単位 (DUW) 環境で接続を管理するために使用できます。

分散作業単位 CONNECT は、プログラムをデフォルト値である RDBCNMTH(*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

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7. 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ため、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、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場合、ご使用のプログラムは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 8. 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 9. DUW プログラムの中のカーソルの例

DRDA ストアード・プロシージャに関する考慮事項

IBM i 分散リレーショナル・データベース・アーキテクチャー (DRDA) サーバーは、ストアード・プロシージャ中の 1 つ以上の結果セットを返す機能をサポートします。

結果セットは、ストアード・プロシージャ内で、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:

161 ページの『ストアード・プロシージャ』

プロシージャ (しばしば、ストアード・プロシージャと呼ばれる) 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す。プロシージャには、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めることができます。SQL のプロシージャの場合も、ホスト言語のプロシージャの場合と同じ利点があります。

分散データベース・プログラミング

関連資料:

SET RESULT SETS

CREATE PROCEDURE (SQL)

CREATE PROCEDURE (外部)

WebSphere MQ と DB2 の組み合わせ

WebSphere® MQ はメッセージ処理システムであり、異なるオペレーティング・システムとネットワークにまたがる分散環境でアプリケーション間の通信を可能にします。

WebSphere MQ は、アプリケーション・プログラミング・インターフェース (API) を使用して、プログラム間の通信を処理します。次のいずれかの API を使用して、WebSphere MQ メッセージ処理システムと対話できます。

- Message Queue Interface (MQI)
- Java 用 WebSphere MQ クラス
- Java Message Service (JMS) 用 WebSphere MQ クラス

DB2 は、DB2 MQ 関数と呼ばれる一連の外部ユーザー定義関数によって、WebSphere MQ メッセージ処理システムに独自のアプリケーション・プログラミング・インターフェースを提供します。SQL ステートメント内でこれらの関数を使用して、DB2 データベース・アクセスと WebSphere MQ メッセージ処理を組み合わせることができます。DB2 MQ 関数は、MQI API を使用します。

WebSphere MQ メッセージ

WebSphere MQ は、アプリケーション間で情報を受け渡すためにメッセージを使用します。

このメッセージは、次の部分から構成されます。

- メッセージの属性。これは、メッセージとそのプロパティを識別します。
- メッセージ・データ。これは、メッセージに入れて送信されるアプリケーション・データです。

WebSphere MQ メッセージ処理

概念的には、WebSphere MQ メッセージ処理システムは、情報 (メッセージ) を受け取り、それをその宛先に送信します。ネットワークにどのような混乱が生じて、MQ は配信を保証します。

WebSphere MQ では、宛先はメッセージ・キューと呼ばれ、キューはキュー・マネージャー内に存在します。アプリケーションは、キューにメッセージを入れたり、キューからメッセージを取り出したりできます。

DB2 は、DB2 MQ 関数と呼ばれる一連の外部ユーザー定義関数によって、WebSphere MQ メッセージ処理システムと通信します。これらの関数は、MQI API を使用します。

MQI API を使用してメッセージを送信する際には、次の 3 つのコンポーネントを指定する必要があります。

メッセージ・データ

プログラム間で送信される内容を定義します。

サービス

メッセージの送信先または発信元を定義します。キューを管理するためのパラメーターはサービス

内で定義され、通常はシステム管理者がそのサービスを定義します。アプリケーション・プログラムは、サービス内部の複雑なパラメーターを扱う必要はありません。

ポリシー

メッセージの処理方法を定義します。ポリシーは、次のような項目を制御します。

- メッセージの属性 (優先順位など)。
- 送信および受信操作のオプション (操作が作業単位の一部であるかどうかなど)。

MQ 関数は、SYSIBM.MQSERVICE_TABLE および SYSIBM.MQPOLICY_TABLE という 2 つの DB2 表で定義されるサービスおよびポリシーを使用します。これらの表は DB2 によって自動的に作成されますが、作成された後にはユーザー管理となり、通常はシステム管理者によって保守されます。DB2 は、デフォルト・サービスおよびデフォルト・ポリシー用の行を最初に各表に設定します。デフォルト・サービスは DB2.DEFAULT.SERVICE、デフォルト・ポリシーは DB2.DEFAULT.POLICY です。

アプリケーション・プログラム側では、これらの表に定義されたサービスおよびポリシーの詳細を知る必要はありません。アプリケーションでは、送信および受信する各メッセージで使用されるサービスおよびポリシーを指定することだけが必要です。アプリケーションは、DB2 MQ 関数を呼び出すときに、この情報を指定します。

DB2 MQ サービス:

サービスは、アプリケーションがメッセージを送信する宛先、またはアプリケーションが受け取るメッセージの送信元を記述します。DB2 MQ サービスは、DB2 表 SYSIBM.MQSERVICE_TABLE で定義されます。アプリケーション・プログラムが DB2 MQ 関数を呼び出す際に、プログラムは SYSIBM.MQSERVICE_TABLE からサービスを選択して、それをパラメーターとして指定します。

SYSIBM.MQSERVICE_TABLE は DB2 によって自動的に作成されますが、作成された後にはユーザー管理となり、通常はシステム管理者によって保守されます。DB2 は最初に、デフォルト・サービス用の行を設定します。デフォルト・サービスは、DB2.DEFAULT.SERVICE です。

新しいサービスは、INSERT ステートメントを実行するだけで追加できます。例えば、以下のステートメントを実行すると、MYSERVICE という名前の新しいサービスが追加されます。MQ デフォルト・キュー・マネージャーおよび MYQUEUE 入力キューが、この新しいサービスのために使用されます。

```
INSERT INTO SYSIBM.MQSERVICE_TABLE (SERVICENAME, QUEUEMANAGER, INPUTQUEUE)
VALUES('MYSERVICE', SPACE(48), 'MYQUEUE')
```

サービス (デフォルト・サービスを含む) は、UPDATE ステートメントを実行するだけで変更できます。DB2 は、パフォーマンス上の理由でサービス情報をキャッシュに入れるので、MQ 関数を既に使用した既存のジョブは、通常、サービスに対する同時変更を検出しません。以下のステートメントを実行すると、入力キューを MYQUEUE2 に変更することで MYSERVICE という名前のサービスが更新されます。

```
UPDATE SYSIBM.MQSERVICE_TABLE
SET INPUTQUEUE = 'MYQUEUE2'
WHERE SERVICENAME = 'MYSERVICE'
```

注: デフォルト・サービス DB2.DEFAULT.SERVICE によって最初に使用されるデフォルトの入力キューは、SYSTEM.DEFAULT.LOCAL.QUEUE という名前の MQ モデル・キューです。他の DB2 製品によって使用されるデフォルトの入力キューは、DB2MQ_DEFAULT_Q です。他の DB2 製品との互換性のために、DB2MQ_DEFAULT_Q という新しい入力キューを作成して、デフォルト・サービスを更新することもできます。例えば、次の通りです。

```
CL: CRTMQMQ QNAME(DB2MQ_DEFAULT_Q) QTYPE(*LCL);
```

```
UPDATE SYSIBM.MQSERVICE_TABLE  
SET INPUTQUEUE = 'DB2MQ_DEFAULT_Q'  
WHERE SERVICENAME = 'DB2.DEFAULT.SERVICE'
```

DB2 MQ ポリシー:

ポリシーは、MQ メッセージの処理方法を制御します。DB2 MQ ポリシーは、DB2 表 SYSIBM.MQPOLICY_TABLE で定義されます。アプリケーション・プログラムが DB2 MQ 関数を呼び出す際に、プログラムは SYSIBM.MQPOLICY_TABLE からポリシーを選択して、それをパラメーターとして指定します。

SYSIBM.MQPOLICY_TABLE は DB2 によって自動的に作成されますが、作成された後にはユーザー管理となり、通常はシステム管理者によって保守されます。DB2 は最初に、デフォルト・ポリシー用の行を設定します。デフォルト・ポリシーは、DB2.DEFAULT.POLICY です。

新しいポリシーは、INSERT ステートメントを実行するだけで追加できます。例えば、以下のステートメントを実行すると、MYPOLICY という名前の新しいポリシーが追加されます。SYNCPOINT 列の値が「N」なので、MYPOLICY を使用する MQ 関数はトランザクションの下では実行されません。

```
INSERT INTO SYSIBM.MQPOLICY_TABLE (POLICYNAME, SYNCPOINT, DESCRIPTION)  
VALUES('MYPOLICY', 'N', 'Policy to not run under a transaction')
```

ポリシー (デフォルト・ポリシーを含む) は、UPDATE ステートメントを実行するだけで変更できます。DB2 は、パフォーマンス上の理由でポリシー情報をキャッシュに入れるので、MQ 関数を既に使用した既存のジョブは、通常、ポリシーに対する同時変更を検出しません。以下のステートメントを実行すると、再試行間隔を 5 秒に変更することによって MYPOLICY というポリシーが更新されます。

```
UPDATE SYSIBM.MQPOLICY_TABLE  
SET SEND_RETRY_INTERVAL = 5000  
WHERE POLICYNAME = 'MYPOLICY'
```

DB2 MQ 関数

DB2 MQ 関数を使用して、メッセージをメッセージ・キューに送信したり、メッセージ・キューからメッセージを受信したりできます。要求をメッセージ・キューに送信して、応答を受信することができます。

DB2 MQ 関数は、次のタイプの操作をサポートします。

- 応答不要送信 (応答が不要の場合)。
- 読み取りまたは受信。1 つまたはすべてのメッセージをキューから除去しないで読み取るか、あるいは受信してキューから除去する場合。
- 要求と応答。送信元アプリケーションが要求に対する応答を必要とする場合。

WebSphere MQ サーバーは、DB2 と同じ IBM i 上にあります。DB2 MQ 関数は DB2 に登録され、MQI API を使用して WebSphere MQ サーバーへのアクセスを提供します。

次の表で、DB2 MQ スカラー関数について説明します。

表 61. DB2 MQ スカラー関数

スカラー関数	説明
MQREAD	MQREAD は、 <i>service-policy</i> で定義されたポリシーを使用して、 <i>receive-service</i> で指定した MQ の場所から VARCHAR 変数にメッセージを返します。この操作は、キューの先頭からメッセージを除去せずに、それを返します。返すメッセージがない場合、NULL 値が返されます。
MQREADCLOB	MQREADCLOB は、 <i>service-policy</i> で定義されたポリシーを使用して、 <i>receive-service</i> で指定した MQ の場所から CLOB 変数にメッセージを返します。この操作は、キューの先頭からメッセージを除去せずに、それを返します。返すメッセージがない場合、NULL 値が返されます。
MQRECEIVE	MQRECEIVE は、 <i>service-policy</i> で定義されたポリシーを使用して、 <i>receive-service</i> で指定した MQ の場所から VARCHAR 変数にメッセージを返します。この操作は、キューからメッセージを除去します。 <i>correlation-id</i> を指定した場合、一致する相関 ID を持つ最初のメッセージが返されます。 <i>correlation-id</i> を指定しない場合、キューの先頭のメッセージが返されます。返すメッセージがない場合、NULL 値が返されます。
MQRECEIVECLOB	MQRECEIVECLOB は、 <i>service-policy</i> で定義されたポリシーを使用して、 <i>receive-service</i> で指定した MQ の場所から CLOB 変数にメッセージを返します。この操作は、キューからメッセージを除去します。 <i>correlation-id</i> を指定した場合、一致する相関 ID を持つ最初のメッセージが返されます。 <i>correlation-id</i> を指定しない場合、キューの先頭のメッセージが返されます。返すメッセージがない場合、NULL 値が返されます。
MQSEND	MQSEND は、 <i>service-policy</i> で定義されたポリシーを使用して、VARCHAR または CLOB 変数 <i>msg-data</i> 内のデータを <i>send-service</i> で指定した MQ の場所に送信します。オプションのユーザー定義メッセージ相関 ID は、 <i>correlation-id</i> で指定できます。戻り値は、成功した場合は 1、失敗した場合は 0 です。
注:	
1. メッセージは、VARCHAR 変数または CLOB 変数に入れて送信または受信できます。 VARCHAR 変数内のメッセージの最大長は 32000 です。 CLOB 変数内のメッセージの最大長は 2 MB です。	

次の表で、DB2 MQ 表関数を説明します。

表 62. DB2 MQ 表関数

表関数	説明
MQREADALL	MQREADALL は、 <i>service-policy</i> で定義されたポリシーを使用して、 <i>receive-service</i> で指定した MQ の場所から、メッセージおよびメッセージ・メタデータを含む表を VARCHAR 変数に返します。この操作は、キューからメッセージを除去しません。 <i>num-rows</i> が指定されている場合は、最大の <i>num-rows</i> メッセージが戻されます。 <i>num-rows</i> が指定されていない場合は、使用可能なすべてのメッセージが戻されます。
MQREADALLCLOB	MQREADALLCLOB は、 <i>service-policy</i> で定義されたポリシーを使用して、 <i>receive-service</i> で指定した MQ の場所から、メッセージおよびメッセージ・メタデータを含む表を CLOB 変数に返します。この操作は、キューからメッセージを除去しません。 <i>num-rows</i> が指定されている場合は、最大の <i>num-rows</i> メッセージが戻されます。 <i>num-rows</i> が指定されていない場合は、使用可能なすべてのメッセージが戻されます。

表 62. DB2 MQ 表関数 (続き)

表関数	説明
MQRECEIVEALL	MQRECEIVEALL は、 <i>service-policy</i> で定義されたポリシーを使用して、 <i>receive-service</i> で指定した MQ の場所から、メッセージおよびメッセージ・メタデータを含む表を VARCHAR 変数に返します。この操作は、メッセージをキューから除去します。 <i>correlation-id</i> を指定した場合、一致する相関 ID を持つメッセージだけが返されます。 <i>correlation-id</i> を指定しない場合、すべての使用可能なメッセージが返されます。 <i>num-rows</i> が指定されている場合は、最大の <i>num-rows</i> メッセージが返されます。 <i>num-rows</i> が指定されていない場合は、使用可能なすべてのメッセージが返されます。
MQRECEIVEALLCLOB	MQRECEIVEALLCLOB は、 <i>service-policy</i> で定義されたポリシーを使用して、 <i>receive-service</i> で指定した MQ の場所から、メッセージおよびメッセージ・メタデータを含む表を CLOB 変数に返します。この操作は、メッセージをキューから除去します。 <i>correlation-id</i> を指定した場合、一致する相関 ID を持つメッセージだけが返されます。 <i>correlation-id</i> を指定しない場合、すべての使用可能なメッセージが返されます。 <i>num-rows</i> が指定されている場合は、最大の <i>num-rows</i> メッセージが返されます。 <i>num-rows</i> が指定されていない場合は、使用可能なすべてのメッセージが返されます。
注:	
<ol style="list-style-type: none"> 1. メッセージは、VARCHAR 変数または CLOB 変数に入れて送信または受信できます。 VARCHAR 変数内のメッセージの最大長は 32000 です。 CLOB 変数内のメッセージの最大長は 2 MB です。 2. DB2 MQ 表関数の結果表の最初の列に、メッセージが入っています。 	

DB2 MQ の依存関係

DB2 MQ 関数を使用するには、IBM MQSeries® for IBM i がインストールおよび構成されていて、操作可能になっていなければなりません。

インストールおよび構成の詳細情報については、Websphere MQSeries インフォメーション・センターを参照してください。IBM MQSeries for IBM i のインストールが完了した後、少なくとも以下のステップを行うことが必要です。

1. MQSeries サブシステムを開始します。

CL: STRSBS SBSD(QMQM/QMQM);

2. デフォルトの MQ メッセージ・キュー・マネージャーを作成します。例えば、次の通りです。

CL: ADDLIBLE QMQM;

CL: CRTMQM MQMNAME(MJAQM) DFTQMGR(*YES);

MQ メッセージ・キュー・マネージャーの CCSID は作成時にジョブ・デフォルト CCSID に設定されるので、ジョブ・デフォルト CCSID が MQ の 1 次使用に設定されるようにします。

3. デフォルトの MQ メッセージ・キュー・マネージャーを開始します。

CL: STRMQM MQMNAME(MJAQM);

DB2 MQ 表

DB2 MQ 表は、DB2 MQ 関数によって使用されるサービスおよびポリシー定義を格納します。

DB2 MQ 表は、SYSIBM.MQSERVICE_TABLE および SYSIBM.MQPOLICY_TABLE です。これらの表は、ユーザーによって管理されます。表は最初に DB2 によって作成され、1 つのデフォルト・サービス (DB2.DEFAULT.SERVICE) および 1 つのデフォルト・ポリシー (DB2.DEFAULT.POLICY) が取り込まれま

す。デフォルト・サービスおよびポリシーの属性は、表の行を更新することによって変更できます。表に追加の行を挿入することにより、追加のサービスおよびポリシーを追加できます。

次の表は、SYSIBM.MQSERVICE_TABLE の列を説明しています。

表 63. SYSIBM.MQSERVICE_TABLE 列の説明

列名	説明
SERVICENAME	この列には、MQ 関数のオプション入力パラメーターであるサービス名が入ります。 この列は、SYSIBM.MQSERVICE_TABLE 表の主キーです。
QUEUEMANAGER	この列には、MQ 関数が接続を確立するキュー・マネージャーの名前が入ります。 列が 48 個のブランクで構成される場合、デフォルトの MQSeries キュー・マネージャーの名前が使用されます。
INPUTQUEUE	この列には、MQ 関数がメッセージを送信および取得するためのキューの名前が入ります。
CODEDCHARSETID	この列には、MQ 関数によって送受信されるメッセージ内のデータの文字セット ID (CCSID) が入ります。 この列は、メッセージ記述子 (MQMD) の CodedCharSetId フィールド (MDCSI) に対応しています。MQ 関数はこの列の値を使用して CodedCharSetId フィールドを設定します。 この列のデフォルト値は -3 です。この場合、DB2 は CodedCharSetId field (MDCSI) をデフォルトのジョブ CCSID に設定します。
ENCODING	この列には、MQ 関数によって送受信されるメッセージ内の数値データのエンコード値が入ります。 この列は、メッセージ記述子 (MQMD) の Encoding フィールド (MDENC) に対応しています。MQ 関数はこの列の値を使用して Encoding フィールドを設定します。 この列のデフォルト値は 0 です。この場合、Encoding フィールド (MDENC) が値 MQENC_NATIVE に設定されます。
DESCRIPTION	この列には、サービスの詳細説明が入ります。

次の表は、SYSIBM.MQPOLICY_TABLE の列を説明しています。

表 64. SYSIBM.MQPOLICY_TABLE 列の説明

列名	説明
POLICYNAME	この列には、MQ 関数のオプション入力パラメーターであるポリシー名が入ります。 この列は、SYSIBM.MQPOLICY_TABLE 表の主キーです。
SEND_PRIORITY	この列には、メッセージの優先順位が入ります。 この列は、メッセージ記述子 (MQMD) の Priority フィールドに対応しています。MQ 関数はこの列の値を使用して Priority フィールドを設定します。 この列のデフォルト値は -1 です。この場合、MQMD の Priority フィールドが値 MQQPRI_PRIORITY_AS_Q_DEF に設定されます。

表 64. SYSIBM.MQPOLICY_TABLE 列の説明 (続き)

列名	説明
SEND_PERSISTENCE	<p>この列は、システム障害が発生したり、キュー・マネージャーの再始動が行われたりした場合でも、メッセージが持続するかどうかを指定します。</p> <p>この列は、メッセージ記述子 (MQMD) の Persistence フィールドに対応しています。MQ 関数はこの列の値を使用して Persistence フィールドを設定します。</p> <p>この列の値は、次のいずれかです。</p> <p>Q MQMD の Persistence フィールドを値 MQPER_PERSISTENCE_AS_Q_DEF に設定します。この値はデフォルトです。</p> <p>Y MQMD の Persistence フィールドを値 MQPER_PERSISTENT に設定します。</p> <p>N MQMD の Persistence フィールドを値 MQPER_NOT_PERSISTENT に設定します。</p>
SEND_EXPIRY	<p>この列には、メッセージの有効期限 (10 分の 1 秒単位) が入ります。</p> <p>この列は、メッセージ記述子 (MQMD) の Expiry フィールドに対応しています。MQ 関数はこの列の値を使用して Expiry フィールドを設定します。</p> <p>デフォルト値は -1 です。この場合、Expiry フィールドが値 MQEI_UNLIMITED に設定されます。</p>
SEND_RETRY_COUNT	<p>この列には、プロシージャが失敗したときに MQ 関数がメッセージの送信を試行する回数が入ります。</p> <p>デフォルト値は 5 です。</p>
SEND_RETRY_INTERVAL	<p>この列には、メッセージの送信を試行する間隔 (ミリ秒単位) が入ります。</p> <p>デフォルト値は 1000 です。</p>
SEND_NEW_CORRELID	<p>この列は、相関 ID が MQ 関数に入力パラメーターとして渡されない場合に、相関 ID を設定する方法を指定します。相関 ID は、メッセージ記述子 (MQMD) の CorrelId フィールドで設定されます。</p> <p>この列には次のどちらかの値を入れることができます。</p> <p>N MQMD の CorrelId フィールドを 2 進ゼロに設定します。この値はデフォルトです。</p> <p>Y キュー・マネージャーが新しい相関 ID を生成し、MQMD の CorrelId フィールドをその値に設定するように指定します。この「Y」値は、メッセージ書き込みオプション (MQPMO) の Options フィールドに MQPMO_NEW_CORREL_ID オプションを設定した場合と同等です。</p>

表 64. SYSIBM.MQPOLICY_TABLE 列の説明 (続き)

列名	説明
SEND_RESPONSE_MSGID	<p>この列は、レポート・メッセージおよび応答メッセージに対して、メッセージ記述子 (MQMD) の MsgId フィールドをどのように設定するかを指定します。</p> <p>この列は、MQMD の Report フィールドに対応しています。MQ 関数はこの列の値を使用して Report フィールドを設定します。</p> <p>この列には次のどちらかの値を入れることができます。</p> <p>N MQMD の Report フィールドに MQRO_NEW_MSG_ID オプションを設定します。この値はデフォルトです。</p> <p>P MQMD の Report フィールドに MQRO_PASS_MSG_ID オプションを設定します。</p>
SEND_RESPONSE_CORRELID	<p>この列は、レポート・メッセージおよび応答メッセージに対して、メッセージ記述子 (MQMD) の CorrelID フィールドをどのように設定するかを指定します。</p> <p>この列は、MQMD の Report フィールドに対応しています。MQ 関数はこの列の値を使用して Report フィールドを設定します。</p> <p>この列には次のどちらかの値を入れることができます。</p> <p>C MQMD の Report フィールドに MQRO_COPY_MSG_ID_TO_CORREL_ID オプションを設定します。この値はデフォルトです。</p> <p>P MQMD の Report フィールドに MQRO_PASS_CORREL_ID オプションを設定します。</p>
SEND_EXCEPTION_ACTION	<p>この列は、元のメッセージを宛先キューに送信できない場合に行う処置を指定します。</p> <p>この列は、メッセージ記述子 (MQMD) の Report フィールドに対応します。MQ 関数はこの列の値を使用して Report フィールドを設定します。</p> <p>この列には次のどちらかの値を入れることができます。</p> <p>Q MQMD の Report フィールドに MQRO_DEAD_LETTER_Q オプションを設定します。この値はデフォルトです。</p> <p>D MQMD の Report フィールドに MQRO_DISCARD_MSG オプションを設定します。</p> <p>P MQMD の Report フィールドに MQRO_PASS_DISCARD_AND_EXPIRY オプションを設定します。</p>

表 64. SYSIBM.MQPOLICY_TABLE 列の説明 (続き)

列名	説明
SEND_REPORT_EXCEPTION	<p>この列は、指定された宛先キューにメッセージを送信できないときに例外レポート・メッセージを生成するかどうか、メッセージを生成する場合はそのレポート・メッセージの内容を指定します。</p> <p>この列は、メッセージ記述子 (MQMD) の Report フィールドに対応します。MQ 関数はこの列の値を使用して Report フィールドを設定します。</p> <p>この列には次のどちらかの値を入れることができます。</p> <p>N 例外レポート・メッセージを生成しないように指定します。Report フィールドにオプションは設定されません。この値はデフォルトです。</p> <p>E MQMD の Report フィールドに MQRO_EXCEPTION オプションを設定します。</p> <p>D MQMD の Report フィールドに MQRO_EXCEPTION_WITH_DATA オプションを設定します。</p> <p>F MQMD の Report フィールドに MQRO_EXCEPTION_WITH_FULL_DATA オプションを設定します。</p>
SEND_REPORT_COA	<p>この列は、メッセージが宛先キューに入れられたときにキュー・マネージャーが到着時確認 (COA) レポート・メッセージを送信するかどうか、送信する場合はその COA メッセージの内容を指定します。</p> <p>この列は、メッセージ記述子 (MQMD) の Report フィールドに対応します。MQ 関数はこの列の値を使用して Report フィールドを設定します。</p> <p>この列には次のどちらかの値を入れることができます。</p> <p>N COA メッセージを送信しないように指定します。Report フィールドにオプションは設定されません。この値はデフォルトです。</p> <p>C MQMD の Report フィールドに MQRO_COA オプションを設定します。</p> <p>D MQMD の Report フィールドに MQRO_COA_WITH_DATA オプションを設定します。</p> <p>F MQMD の Report フィールドに MQRO_COA_WITH_FULL_DATA オプションを設定します。</p>
SEND_REPORT_COD	<p>この列は、アプリケーションが宛先キューからメッセージを取り出して削除したときにキュー・マネージャーが送信時確認 (COD) レポート・メッセージを送信するかどうか、送信する場合はその COD メッセージの内容を指定します。</p> <p>この列は、メッセージ記述子 (MQMD) の Report フィールドに対応します。MQ 関数はこの列の値を使用して Report フィールドを設定します。</p> <p>この列には次のどちらかの値を入れることができます。</p> <p>N COD メッセージを送信しないように指定します。Report フィールドにオプションは設定されません。この値はデフォルトです。</p> <p>C MQMD の Report フィールドに MQRO_COD オプションを設定します。</p> <p>D MQMD の Report フィールドに MQRO_COD_WITH_DATA オプションを設定します。</p> <p>F MQMD の Report フィールドに MQRO_COD_WITH_FULL_DATA オプションを設定します。</p>

表 64. SYSIBM.MQPOLICY_TABLE 列の説明 (続き)

列名	説明
SEND_REPORT_EXPIRY	<p>この列は、メッセージがアプリケーションに送信される前に廃棄された場合にキュー・マネージャーが期限切れレポート・メッセージを送信するかどうか、送信する場合はそのメッセージの内容を指定します。</p> <p>この列は、メッセージ記述子 (MQMD) の Report フィールドに対応します。MQ 関数はこの列の値を使用して Report フィールドを設定します。</p> <p>この列には次のどちらかの値を入れることができます。</p> <p>N 期限切れレポート・メッセージを送信しないように指定します。Report フィールドにオプションは設定されません。この値はデフォルトです。</p> <p>C MQMD の Report フィールドに MQRO_EXPIRATION オプションを設定します。</p> <p>D MQMD の Report フィールドに MQRO_EXPIRATION_WITH_DATA オプションを設定します。</p> <p>F MQMD の Report フィールドに MQRO_EXPIRATION_WITH_FULL_DATA オプションを設定します。</p>
SEND_REPORT_ACTION	<p>この列は、受信側アプリケーションが肯定アクション通知 (PAN)、否定アクション通知 (NAN)、またはその両方を送信するかどうかを指定します。</p> <p>この列は、メッセージ記述子 (MQMD) の Report フィールドに対応します。MQ 関数はこの列の値を使用して Report フィールドを設定します。</p> <p>この列には次のどちらかの値を入れることができます。</p> <p>N どちらの通知も送信しないように指定します。Report フィールドにオプションは設定されません。この値はデフォルトです。</p> <p>P MQMD の Report フィールドに MQRO_PAN オプションを設定します。</p> <p>T MQMD の Report フィールドに MQRO_NAN オプションを設定します。</p> <p>B MQMD の Report フィールドに MQRO_PAN と MQRO_NAN の両方のオプションを設定します。</p>
SEND_MSG_TYPE	<p>この列には、メッセージのタイプが入ります。</p> <p>この列は、メッセージ記述子 (MQMD) の MsgType フィールドに対応しています。MQ 関数はこの列の値を使用して MsgType フィールドを設定します。</p> <p>この列には次のどちらかの値を入れることができます。</p> <p>DTG MQMD の MsgType フィールドを MQMT_DATAGRAM に設定します。この値はデフォルトです。</p> <p>REQ MQMD の MsgType フィールドを MQMT_REQUEST に設定します。</p> <p>RLY MQMD の MsgType フィールドを MQMT_REPLY に設定します。</p> <p>RPT MQMD の MsgType フィールドを MQMT_REPORT に設定します。</p>

表 64. SYSIBM.MQPOLICY_TABLE 列の説明 (続き)

列名	説明
REPLY_TO_Q	<p>この列には、MQGET 呼び出しを発行したアプリケーションが応答メッセージおよびレポート・メッセージを送信する先のメッセージ・キューの名前が入ります。</p> <p>この列は、メッセージ記述子 (MQMD) の ReplyToQ フィールドに対応しています。MQ 関数はこの列の値を使用して ReplyToQ フィールドを設定します。</p> <p>この列のデフォルト値は SAME AS INPUT_Q です。この場合、この名前は、メッセージの送信に使用されたサービスで定義されたキュー名に設定されます。サービスが指定されなかった場合、名前はデフォルト・サービスのキュー・マネージャーの名前に設定されます。</p>
REPLY_TO_QMGR	<p>この列には、応答メッセージおよびレポート・メッセージが送信される先のキュー・マネージャーの名前が入ります。</p> <p>この列は、メッセージ記述子 (MQMD) の ReplyToQMgr フィールドに対応しています。MQ 関数はこの列の値を使用して ReplyToQMgr フィールドを設定します。</p> <p>この列のデフォルト値は SAME AS INPUT_QMGR です。この場合、この名前は、メッセージの送信に使用されたサービスで定義されたキュー・マネージャー名に設定されます。サービスが指定されなかった場合、名前はデフォルト・サービスのキュー・マネージャーの名前に設定されます。</p>
RCV_WAIT_INTERVAL	<p>この列には、メッセージがキューに到着するまで DB2 が待機する時間 (ミリ秒単位) が入ります。</p> <p>この列は、get メッセージのオプション (MQGMO) の WaitInterval フィールドに対応しています。MQ 関数はこの列の値を使用して WaitInterval フィールドを設定します。</p> <p>デフォルトは、10 です。</p>
RCV_CONVERT	<p>この列は、関数のために使用されるサービスで定義された CodedCharSetId および Encoding の値に準拠するように、メッセージ内のアプリケーション・データを変換するかどうかを指定します。</p> <p>この列は、get メッセージのオプション (MQGMO) の Options フィールドに対応しています。MQ 関数はこの列の値を使用して Options フィールドを設定します。</p> <p>この列には次のどちらかの値を入れることができます。</p> <p>Y MQGMO の Options フィールドに MQGMO_CONVERT オプションを設定します。この値はデフォルトです。</p> <p>N データを変換しないように指定します。</p>

表 64. SYSIBM.MQPOLICY_TABLE 列の説明 (続き)

列名	説明
RCV_ACCEPT_TRUNC_MSG	<p>この列は、サイズ超過のメッセージが取り出されたときの MQ 関数の動作を指定します。</p> <p>この列は、get メッセージのオプション (MQGMO) の Options フィールドに対応しています。MQ 関数はこの列の値を使用して Options フィールドを設定します。</p> <p>この列には次のどちらかの値を入れることができます。</p> <p>Y MQGMO の Options フィールドに MQGMO_ACCEPT_TRUNCATED_MSG オプションを設定します。この値はデフォルトです。</p> <p>N メッセージを切り捨てないように指定します。メッセージが大きすぎてバッファに収まらない場合、MQ 関数はエラーを出して終了します。</p> <p>推奨: この列は Y に設定してください。こうすれば、メッセージ・バッファが小さすぎてメッセージを完全に格納できない場合に、MQ 関数はバッファに格納できる限りのメッセージをバッファに入れることができます。</p>
REV_OPEN_SHARED	<p>この列は、メッセージを取り出すときの入力キュー・モードを指定します。</p> <p>この列は、MQOPEN 呼び出しの Options パラメーターに対応しています。MQ 関数はこの列の値を使用して Options パラメーターを設定します。</p> <p>この列には次のどちらかの値を入れることができます。</p> <p>S MQOO_INPUT_SHARED オプションを設定します。この値はデフォルトです。</p> <p>E MQ オプション MQOO_INPUT_EXCLUSIVE オプションを設定します。</p> <p>D MQ オプション MQOO_INPUT_AS_Q_DEF オプションを設定します。</p>
SYNCPOINT	<p>この列は、通常の作業単位において、MQ 関数がプロトコルの範囲内で動作するかどうかを指定します。</p> <p>この列には次のどちらかの値を入れることができます。</p> <p>Y 通常の作業単位において、MQ 関数がプロトコルの範囲内で動作することを指定します。MQ 関数をトランザクションの下で実行する場合、この値を使用します。この値はデフォルトです。</p> <p>N 通常の作業単位において、MQ 関数がプロトコルの範囲外で動作することを指定します。MQ 関数をトランザクションの下で実行しない場合、この値を使用します。</p>
DESCRIPTION	この列には、ポリシーの詳細説明が入ります。

DB2 MQ の CCSID 変換

メッセージが送信される時、送信されるメッセージが DB2 によってジョブ CCSID に変換されることがあります。メッセージが読み取られるか受信される時、指定された CCSID に Websphere MQ によって変換されることがあります。

MQSEND 関数の *msg-data* パラメーターは、ジョブ CCSID を使用すると定義されています。ストリングが *msg-data* に渡される場合、ジョブ CCSID に変換されます。例えば、CCSID が 1200 のストリングが

msg-data に渡される場合、ジョブ CCSID に変換された後にメッセージ・データが Websphere MQ に渡されます。ストリングがビット・データとして定義されているか、またはストリングの CCSID がジョブ CCSID である場合、変換は行われません。

Websphere MQ は、MQSEND が実行されるときにはメッセージ・データの CCSID 変換を行いません。DB2 から渡されるメッセージ・データは、変更されずに CCSID を付けて送信されます。この CCSID を使って、メッセージの受信側にメッセージ・データの解釈方法を知らせます。送信される CCSID は、MQSEND 関数で使用されるサービスの CODEDCHARSETID に指定される値に依存します。CODEDCHARSETID のデフォルトは -3 です。この場合、渡される CCSID がデフォルトのジョブ CCSID であることを示します。-3 以外の値が CODEDCHARSETID に使用されている場合、呼び出し側は、MQSEND に渡されるメッセージ・データが DB2 によってジョブ CCSID に変換されないこと、およびその指定された CCSID でストリングがエンコードされることを確認する必要があります。

メッセージが DB2 MQ スカラー関数または表関数で読み取られるか受信されるとき、*msg-data* 戻りパラメーター (および DB2 MQ 表関数の MSG 結果列) もデフォルトのジョブ CCSID を使用するように定義されます。DB2 は変換を行わず、必要な変換の実行はすべて Websphere MQ に依存します。Websphere がメッセージ・データを変換するかどうかは、指定したポリシーで RCV_CONVERT 値を「N」に設定することで制御できます。

指定のサービスで CODEDCHARSETID の値が -3 の場合、DB2 は、読み取るか受信するメッセージを Websphere MQ がジョブ CCSID に変換するように指示します。-3 以外の値が CODEDCHARSETID に使用されている場合、DB2 は、読み取るか受信するメッセージを Websphere MQ がその CCSID に変換するように指示します。*msg-data* 戻りパラメーターおよび MSG 結果列ではデフォルトのジョブ CCSID を使用するように DB2 で定義されているため、メッセージの読み取りまたは受信に使用されるサービスでは、CODEDCHARSETID に -3 以外の値を指定することは勧められていません。

メッセージを読み取るか受信するときに、切り捨てが生じることがあります。指定したポリシーで RCV_ACCEPT_TRUNC_MSG の値が「Y」の場合、メッセージは警告なしで切り捨てられることがあります。RCV_ACCEPT_TRUNC_MSG の値が「N」の場合、メッセージが長すぎると、関数はエラーを出して終了します。

Websphere MQ トランザクション

Websphere MQ でのメッセージの送受信は、トランザクションの一部としてでもトランザクション外でも可能です。メッセージがトランザクションの一部として送受信される場合、そのトランザクションには DB2 操作などの他のリソースを含めることができます。

Websphere MQ は、トランザクション・マネージャー自体として機能することもできますし、他のトランザクション・マネージャー (CICS®、Encina、Tuxedo など) に対するリソース・マネージャーとして機能することもできます。トランザクションおよびサポートされる外部トランザクション・マネージャーの詳細情報については、Websphere MQSeries インフォメーション・センターを参照してください。

Websphere は、IBM i コミットメント制御によって管理されるトランザクションに関与することもできます。SQL ステートメントが実行されてコミットメント制御がまだ開始されていないと判別されたとき、コミットメント制御が DB2 によって自動的に開始されます。デフォルトでは、コミットメント定義のスコープは *ACTGRP です。同じジョブ内で実行される MQ 関数および DB2 操作が、IBM i コミットメント制御で管理される同じトランザクションに関与するためには、コミットメント制御をスコープ *JOB で開始する必要があります。例えば、以下の 2 つの CL コマンドは現在のコミットメント定義を終了し、スコープを *JOB にして開始します。

```
ENDCMTCTL
STRCMTCTL LCKLVL(*CHG) CMTSCOPE(*JOB)
```

サーバー・ジョブでスコープを *JOB にしてコミットメント制御を開始するには、終了と開始のコミットメント制御 CL コマンドを 1 つのプロシージャで共に実行する必要があります。以下のステップでは、START_JOB_LEVEL_COMMIT というサンプル・プロシージャを作成します。これは SQL CALL ステートメントで呼び出せます。

1. 以下のソースを MJASRC/CL 内のソース・ファイル・メンバー JOBSCOPE に入力します。

```
PGM
ENDCMTCTL
STRCMTCTL LCKLVL(*CHG) CMTSCOPE(*JOB)
ENDPGM
```

2. 以下のデフォルトの MQ メッセージ・キュー・マネージャーを使用して、CL プログラムを作成します。

```
CRTCLPGM PGM(MJATST/JOBSCOPE) SRCFILE(MJASRC/CL)
```

3. CL プログラムを参照する SQL 外部プロシージャを作成します。

```
CREATE PROCEDURE MJATST.START_JOB_LEVEL_COMMIT ()
EXTERNAL NAME 'MJATST/JOBSCOPE'
PARAMETER STYLE GENERAL
```

プロシージャは通常、1 つのジョブで 1 回呼び出されます。スコープが *JOB のコミットメント定義が開始されると、MQ 操作および DB2 操作を 1 つのトランザクションの一部として実行できるようになります。例えば、以下の INSERT ステートメントおよび MQSEND 関数は、同じトランザクションで実行されます。トランザクションがコミットされると、両方の操作がコミットされます。トランザクションがロールバックされると、両方の操作がロールバックされます。

```
CALL MJATST.START_JOB_LEVEL_COMMIT;
INSERT INTO MJATST.T1
VALUES(1);
VALUES MQSEND('A commit test message');
COMMIT;

INSERT INTO MJATST.T1
VALUES(2);
VALUES MQSEND('A rollback test message');
ROLLBACK;
```

Websphere MQ でのメッセージの送受信は、トランザクションの一部としてでもトランザクション外でも可能です。DB2 MQ 関数では、これは指定したポリシーで制御されます。各ポリシーには、SYNCPOINT 属性があります。ポリシーの SYNCPOINT 列の値が「N」である場合、そのポリシーを使用する DB2 MQ 関数はいずれもトランザクションに関与しません。ポリシーの SYNCPOINT 列の値が「Y」である場合、そのポリシーを使用して入力キューを変更する DB2 MQ 関数はすべてトランザクションに関与します。

WebSphere MQ による基本的なメッセージング

DB2 MQ の機能による最も基本的な形式のメッセージングは、すべてのデータベース・アプリケーションが同じ DB2 データベース・サーバーに接続するときになります。クライアントは、データベース・サーバーのローカル側にあっても、ネットワーク環境に分散していても構いません。

単純なシナリオとして、クライアント A が、MQSEND 関数を呼び出して、ユーザー定義ストリングをデフォルト・サービスで定義された場所へ送信します。DB2 は、データベース・サーバー上でこの操作を行う MQ 関数を実行します。しばらく後に、クライアント B が MQRECEIVE 関数を呼び出し、デフォルト・サービスで定義されたキューの先頭のメッセージを除去してから、それをクライアントに返します。DB2 は、データベース・サーバー上でこの操作を行う MQ 関数を実行します。

データベース・クライアントは、以下のようないくつかの方法で単純なメッセージングを使用できます。

- データの収集

情報を 1 つ以上のソースからメッセージの形式で受け取ります。情報源は、任意のアプリケーションを使用できます。データはキューから受け取り、追加の処理を行うためにデータベース表に保管されません。

- ワークロードの分散

同じアプリケーションの複数のインスタンスが共有するキューに作業要求を追加します。アプリケーション・インスタンスは、作業を実行する準備が整うと、作業要求が入ったメッセージをキューの先頭から受け取ります。アプリケーションの複数インスタンスが、プールされた要求の単一のキューによって表されるワークロードを共有できます。

- アプリケーションのシグナル

複数のプロセスが共同作業を行う状況では、それぞれの作業を調整するためにメッセージが使用されることがよくあります。こうしたメッセージには、実行する必要があるコマンドや要求を含めることができます。この技法について詳しくは、367 ページの『WebSphere MQ によるアプリケーション間の接続』を参照してください。

次のシナリオでは、基本的なメッセージングを拡張してリモート・メッセージングを取り入れています。マシン A がマシン B にメッセージを送信すると想定します。

1. DB2 クライアントが、マシン B 上のリモート・キューとなるように定義されたターゲット・サービスを指定して、MQSEND 関数呼び出しを実行します。
2. MQ 関数が、メッセージを送信する作業を実行します。マシン A の WebSphere MQ サーバーはメッセージを受け取り、サービスおよびマシン A の現行 MQ 構成によって定義された宛先にそれが確実に送信されるようにします。このサーバーは、宛先がマシン B 上のキューであると判別した後、マシン B の WebSphere MQ サーバーへのメッセージの送信を試行し、必要に応じて再試行します。
3. マシン B の WebSphere MQ サーバーは、マシン A のサーバーからメッセージを受け取り、それをマシン B 上の宛先キューに入れます。
4. マシン B の WebSphere MQ クライアントが、キューの先頭にあるメッセージを要求します。

MQI API からエラーが返された場合、指定された DB2 MQ 関数はエラーを返します。エラーには失敗した MQ API の名前、MQ 条件コード、および MQ 理由コードが記述されます。障害の原因を判別するには、WebSphere MQSeries インフォメーション・センターを参照する必要があります。例えば、存在しない MQSeries キュー・マネージャーの名前がサービスに含まれる場合、以下のエラーが返されます。

```
SQL State: 38401
Vendor Code: -443
Message: [SQL0443] cc=2:rc=2058:MQCONN FAILED
```

このメッセージは、MQCONN API が理由コード 2058 で失敗したことを示します。WebSphere MQSeries インフォメーション・センターには、理由コード 2058 に関する詳細な説明が記載されています。

WebSphere MQ によるメッセージの送信

WebSphere MQ を使用してメッセージを送信するときには、送信するデータの内容、その送信先、および送信する時を選択します。このタイプのメッセージングは、応答不要送信と呼ばれます。つまり、送信側はメッセージを送信しますが、メッセージを宛先に確実に届ける処理は WebSphere MQ に依存します。

WebSphere MQ を使用してメッセージを送信するには、MQSEND を使用します。複数の列の情報を送信する場合、それらの列をブランク文字で区切ります。例えば、次の通りです。

```
VALUES MQSEND(LASTNAME CONCAT ' ' CONCAT FIRSTNAME);
```

次の例では、デフォルト・サービス DB2.DEFAULT.SERVICE、および SYNCPOINT 値が「Y」のデフォルト・ポリシー DB2.DEFAULT.POLICY を使用します。この MQSEND 関数はトランザクションの下で実行されるため、COMMIT ステートメントを実行すると、メッセージは確実にキューに追加されます。

```
VALUES MQSEND('Testing msg');
COMMIT;
```

SYNCPOINT 値が「N」に設定されたポリシーを使用する場合、COMMIT ステートメントを使用する必要はありません。例えば、ポリシー MYPOLICY2 の SYNCPOINT 値が「N」であると想定します。次の SQL ステートメントを実行すると、COMMIT ステートメントを必要とせずにメッセージがキューに追加されます。

```
VALUES MQSEND('DB2.DEFAULT.SERVICE', 'MYPOLICY2', 'Testing msg')
```

メッセージの内容は、SQL ステートメント、式、関数、およびユーザー指定データを任意に組み合わせて使用できます。VARCHAR 列が LASTNAME、FIRSTNAME、および DEPARTMENT の EMPLOYEE 表があると想定します。DEPARTMENT 5LGA の各従業員にこの情報を含むメッセージを送信するために、次の SQL SELECT ステートメントを実行します。

```
SELECT MQSEND(LASTNAME CONCAT ' ' CONCAT FIRSTNAME CONCAT ' ' CONCAT DEPARTMENT)
FROM EMPLOYEE
WHERE DEPARTMENT = '5LGA';
COMMIT;
```

WebSphere MQ によるメッセージの取り出し

WebSphere MQ を使用すると、プログラムでのメッセージの読み取りまたは受信が可能になります。読み取りと受信のどちらの操作を行っても、キューの先頭にあるメッセージが返されます。ただし、読み取り操作ではメッセージはキューから除去されませんが、受信操作では除去されます。

受信操作を使用して取り出されるメッセージは一度しか取り出すことができないのに対して、読み取り操作を使用して取り出されるメッセージは、同じメッセージを何度でも取り出すことができます。

次の SQL ステートメントは、デフォルトのサービスとポリシーで指定されたキューの先頭のメッセージを読み取ります。SQL ステートメントは、VARCHAR(32000) ストリングを返します。読み取るメッセージがない場合、NULL 値が返されます。MQREAD はキューを変更しないので、COMMIT ステートメントを使用する必要はありません。

```
VALUES MQREAD()
```

次の SQL ステートメントにより、キューの内容は結果セットとして返されます。表関数の結果表 T は、デフォルト・サービスで定義されたキュー内のすべてのメッセージと、それらのメッセージに関するメタデータ

ータで構成されます。マテリアライズ結果表の最初の列はメッセージ自体で、残りの列にはメタデータが入ります。SELECT ステートメントは、メッセージとメタデータの両方を返します。

```
SELECT T.*
FROM TABLE ( MQREADALL() ) AS T;
```

次のステートメントは、メッセージだけを返します。表関数の結果表 T は、デフォルト・サービスで定義されたキュー内のすべてのメッセージと、それらのメッセージに関するメタデータで構成されます。

```
SELECT T.MSG
FROM TABLE ( MQREADALL() ) AS T;
```

次の SQL ステートメントは、キューの先頭にあるメッセージを受信 (除去) します。SELECT ステートメントは、VARCHAR(32000) スtringを返します。この MQRECEIVE 関数はトランザクションの下で実行されるため、COMMIT ステートメントを実行すると、メッセージは確実にキューから除去されます。取り出すメッセージがない場合、NULL 値が返されて、キューは変更されません。

```
VALUES MQRECEIVE()
COMMIT;
```

MESSAGES 表があり、そこに VARCHAR(32000) の列が 1 つ含まれると想定します。次の SQL INSERT ステートメントは、デフォルト・サービス・キュー内のすべてのメッセージを MESSAGES 表に挿入します。

```
INSERT INTO MESSAGES
SELECT T.MSG
FROM TABLE ( MQRECEIVEALL() ) AS T;
COMMIT;
```

WebSphere MQ によるアプリケーション間の接続

アプリケーション間の接続は、通常、多様なアプリケーション・サブシステムの集まりを統合するために使用されます。アプリケーションの統合を容易にするために、WebSphere MQ は、アプリケーションを相互接続する手段を備えています。

「要求/応答」通信方式を使用すると、1 つのアプリケーションが別のアプリケーションのサービスを要求できます。これを行うための 1 つの方法は、リクエスターがサービス・プロバイダーにメッセージを送信し、何らかの作業を実行することを要求します。作業の完了後、プロバイダーは、リクエスターに結果を送信して返すのか、それとも完了の確認だけを返すのかを決定できます。このリクエスターが応答を受け取るまで待機してから続行する場合を除き、WebSphere MQ は、その要求に応答を関連付ける方法を備えている必要があります。

WebSphere MQ には、リクエスターとプロバイダー間で交換するメッセージを相関させるための相関 ID があります。リクエスターは、既知の相関 ID を使用して、メッセージにマークを付けます。プロバイダーは、同じ相関 ID を使用して、その応答にマークを付けます。リクエスターは、関連付けられた応答を取り出すため、キューからメッセージを受け取るときにその相関 ID を提供します。一致する相関 ID を持つ最初のメッセージが、リクエスターに返されます。

次の SQL SELECT ステートメントは、相関 ID が CORRID1 のポリシー MYPOLICY を使用して、String「Msg with corr id」で構成されるメッセージをサービス MYSERVICE に送信します。ポリシー MYPOLICY の SYNCPOINT 属性が「N」であるため、WebSphere MQ はメッセージをキューに追加します。ユーザーが COMMIT ステートメントを使用する必要はありません。

```
VALUES MQSEND('MYSERVICE', 'MYPOLICY', 'Msg with corr id', 'CORRID1')
```

次の SQL ステートメントは、ポリシー MYPOLICY を使用して、サービス MYSERVICE で指定されたキューから、ID CORRID1 に一致する最初のメッセージを受信します。SQL ステートメントは、VARCHAR(32000) スtringを返します。この相関 ID を持つメッセージがない場合、NULL 値が返されて、キューは変更されません。

```
VALUES MQRECEIVE('MYSERVICE', 'MYPOLICY', 'CORRID1')
```

参照

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 for i サンプル表

以下のサンプル表は、「SQL プログラミング」、および「SQL 解説書」トピック集で参照および使用されています。

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、IBM DB2 Query Manager and SQL Development Kit for i 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。

ストアド・プロシージャはシステムの一部として組み込まれており、すべての表を作成するデータ定義言語 (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。プロシージャは、プロシージャ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、外部ストアド・プロシージャであるので、対話式 SQL および System i ナビゲーターを含む、すべての SQL インターフェイ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。

関連資料:

20 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。

288 ページの『行記憶域を使用した複数行 FETCH』

複数行用 FETCH ステートメントを行記憶域とともに使用するには、その前にアプリケーションで、行記憶域および関連した記述域を定義しなければなりません。

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 65. 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:

DEPARTMENT 表にあるデータの完全なリストを以下に示します。

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	?

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
D21	管理システム	000070	D01	?
E01	サポート・サービス	000050	A00	?
E11	業務部	000090	E01	?
E21	ソフトウェア・サポート	000100	E01	?
F22	事業所 F2	?	E01	?
G22	事業所 G2	?	E01	?
H22	事業所 H2	?	E01	?
I22	事業所 I2	?	E01	?
J22	事業所 J2	?	E01	?

社員表 (EMPLOYEE)

社員表には、全社員が社員番号で識別され、基本的な個人情報が記述されています。

社員表は、以下の CREATE TABLE ステートメントおよび ALTER TABLE ステートメントを使用して作成します。

```
CREATE TABLE EMPLOYEE
  (EMPNO      CHAR(6)          NOT NULL,
   FIRSTNME   VARCHAR(12)       NOT NULL,
   MIDINIT    CHAR(1)         NOT NULL,
   LASTNAME   VARCHAR(15)      NOT NULL,
   WORKDEPT   CHAR(3)         ,
   PHONENO    CHAR(4)         ,
   HIREDATE   DATE            ,
   JOB        CHAR(8)         ,
   EDLEVEL    SMALLINT        NOT NULL,
   SEX        CHAR(1)         ,
   BIRTHDATE  DATE            ,
   SALARY     DECIMAL(9,2)    ,
   BONUS      DECIMAL(9,2)    ,
   COMM       DECIMAL(9,2)    ,
   PRIMARY KEY (EMPNO))
```

```
ALTER TABLE EMPLOYEE
  ADD FOREIGN KEY REF (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
FIRSTNAME	社員の名
MIDINIT	社員のミドルネームの頭文字
LASTNAME	社員の姓
WORKDEPT	社員が所属している部門の ID
PHONENO	社員の電話番号
HIREDATE	雇用年月日
JOB	社員の職種
EDLEVEL	学歴年数
SEX	社員の性別 (M または F)
BIRTHDATE	生年月日
SALARY	給与
BONUS	賞与
COMM	年俸

EMPLOYEE:

EMPLOYEE 表にあるデータの完全なリストを以下に示します。

EMP NO	FIRST NAME	MID INIT	LASTNAME	WORK DEPT	PHONE NO	HIRE DATE	JOB	ED LEVEL	SEX	BIRTH DATE	SAL-ARY	BONUS	COMM
000010	CHRISTINE	I	HAAS	A00	3978	1965-01-01	PRES	18	F	1933-08-24	52750	1000	4220
000020	MICHAEL	L	THOMPSON	B01	3476	1973-10-10	MANAGER	18	M	1948-02-02	41250	800	3300
000030	SALLY	A	KWAN	C01	4738	1975-04-05	MANAGER	20	F	1941-05-11	38250	800	3060
000050	JOHN	B	GEYER	E01	6789	1949-08-17	MANAGER	16	M	1925-09-15	40175	800	3214
000060	IRVING	F	STERN	D11	6423	1973-09-14	MANAGER	16	M	1945-07-07	32250	500	2580
000070	EVA	D	PULASKI	D21	7831	1980-09-30	MANAGER	16	F	1953-05-26	36170	700	2893
000090	EILEEN	W	HENDERSON	E11	5498	1970-08-15	MANAGER	16	F	1941-05-15	29750	600	2380
000100	THEODORE	Q	SPENSER	E21	0972	1980-06-19	MANAGER	14	M	1956-12-18	26150	500	2092
000110	VINCENZO	G	LUCCHESSI	A00	3490	1958-05-16	SALESREP	19	M	1929-11-05	46500	900	3720
000120	SEAN		O'CONNELL	A00	2167	1963-12-05	CLERK	14	M	1942-10-18	29250	600	2340
000130	DOLORES	M	QUINTANA	C01	4578	1971-07-28	ANALYST	16	F	1925-09-15	23800	500	1904
000140	HEATHER	A	NICHOLLS	C01	1793	1976-12-15	ANALYST	18	F	1946-01-19	28420	600	2274
000150	BRUCE		ADAMSON	D11	4510	1972-02-12	DESIGNER	16	M	1947-05-17	25280	500	2022
000160	ELIZABETH	R	PIANKA	D11	3782	1977-10-11	DESIGNER	17	F	1955-04-12	22250	400	1780
000170	MASATOSHI	J	YOSHIMURA	D11	2890	1978-09-15	DESIGNER	16	M	1951-01-05	24680	500	1974
000180	MARILYN	S	SCOUTTEN	D11	1682	1973-07-07	DESIGNER	17	F	1949-02-21	21340	500	1707
000190	JAMES	H	WALKER	D11	2986	1974-07-26	DESIGNER	16	M	1952-06-25	20450	400	1636
000200	DAVID		BROWN	D11	4501	1966-03-03	DESIGNER	16	M	1941-05-29	27740	600	2217
000210	WILLIAM	T	JONES	D11	0942	1979-04-11	DESIGNER	17	M	1953-02-23	18270	400	1462
000220	JENNIFER	K	LUTZ	D11	0672	1968-08-29	DESIGNER	18	F	1948-03-19	29840	600	2387
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21	CLERK	14	M	1935-05-30	22180	400	1774
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05	CLERK	17	M	1954-03-31	28760	600	2301
000250	DANIEL	S	SMITH	D21	0961	1969-10-30	CLERK	15	M	1939-11-12	19180	400	1534
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11	CLERK	16	F	1936-10-05	17250	300	1380
000270	MARIA	L	PEREZ	D21	9001	1980-09-30	CLERK	15	F	1953-05-26	27380	500	2190
000280	ETHEL	R	SCHNEIDER	E11	8997	1967-03-24	OPERATOR	17	F	1936-03-28	26250	500	2100
000290	JOHN	R	PARKER	E11	4502	1980-05-30	OPERATOR	12	M	1946-07-09	15340	300	1227
000300	PHILIP	X	SMITH	E11	2095	1972-06-19	OPERATOR	14	M	1936-10-27	17750	400	1420
000310	MAUDE	F	SETRIGHT	E11	3332	1964-09-12	OPERATOR	12	F	1931-04-21	15900	300	1272
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07	FILEREP	16	M	1932-08-11	19950	400	1596
000330	WING		LEE	E21	2103	1976-02-23	FILEREP	14	M	1941-07-18	25370	500	2030
000340	JASON	R	GOUNOT	E21	5698	1947-05-05	FILEREP	16	M	1926-05-17	23840	500	1907
200010	DIAN	J	HEMMINGER	A00	3978	1965-01-01	SALESREP	18	F	1933-08-14	46500	1000	4220
200120	GREG		ORLANDO	A00	2167	1972-05-05	CLERK	14	M	1942-10-18	29250	600	2340
200140	KIM	N	NATZ	C01	1793	1976-12-15	ANALYST	18	F	1946-01-19	28420	600	2274
200170	KIYOSHI		YAMAMOTO	D11	2890	1978-09-15	DESIGNER	16	M	1951-01-05	24680	500	1974
200220	REBA	K	JOHN	D11	0672	1968-08-29	DESIGNER	18	F	1948-03-19	29840	600	2387
200240	ROBERT	M	MONTEVERDE	D21	3780	1979-12-05	CLERK	17	M	1954-03-31	28760	600	2301
200280	EILEEN	R	SCHWARTZ	E11	8997	1967-03-24	OPERATOR	17	F	1936-03-28	26250	500	2100
200310	MICHELLE	F	SPRINGER	E11	3332	1964-09-12	OPERATOR	12	F	1931-04-21	15900	300	1272
200330	HELENA		WONG	E21	2103	1976-02-23	FIELDRP	14	F	1941-07-18	25370	500	2030
200340	ROY	R	ALONZO	E21	5698	1947-05-05	FIELDRP	16	M	1926-05-17	23840	500	1907

社員の写真表 (EMP_PHOTO)

社員の写真表には、社員番号別に識別された、各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:

EMP_PHOTO 表にあるデータの完全なリストを以下に示します。

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)

社員の履歴表には、社員番号別に識別された、各社員の履歴が入っています。

社員の履歴表は、以下の CREATE TABLE ステートメントおよび ALTER TABLE ステートメントを使用して作成します。

```
CREATE TABLE EMP_RESUME
  (EMPNO CHAR(6) NOT NULL,
   RESUME_FORMAT VARCHAR(10) NOT NULL,
   RESUME CLOB(5K),
   EMP_ROWID CHAR(40) NOT NULL DEFAULT '',
   PRIMARY KEY (EMPNO,RESUME_FORMAT))
```

```
ALTER TABLE EMP_RESUME
  ADD COLUMN DL_RESUME DATALINK(1000)
  LINKTYPE URL NO LINK CONTROL
```

```
ALTER TABLE EMP_RESUME
  ADD FOREIGN KEY (EMPNO)
  REFERENCES EMPLOYEE
  ON DELETE RESTRICT
```

以下の索引が作成されます。

```
CREATE UNIQUE INDEX XEMP_RESUME
  ON EMP_RESUME (EMPNO,RESUME_FORMAT)
```

次の表は、列の内容を示しています。

列名	説明
EMPNO	社員番号

列名	説明
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:

EMP_RESUME 表にあるデータの完全なリストを以下に示します。

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 (EMPPROJECT)

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に参加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組み込まれます。

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 EMPPROJECT
  (EMPNO      CHAR(6)      NOT NULL,
   PROJNO    CHAR(6)      NOT NULL,
   ACTNO     SMALLINT     NOT NULL,
   EMPTIME   DECIMAL(5,2) ,
   EMSTDATE  DATE          ,
   EMENDATE  DATE          )

ALTER TABLE EMPPROJECT
  ADD FOREIGN KEY REPAPA (PROJNO, ACTNO, EMSTDATE)
  REFERENCES PROJECT
  ON DELETE RESTRICT
```

以下の別名が、表用に作成されます。

```
CREATE ALIAS EMPACT FOR EMPPROJECT
CREATE ALIAS EMP_ACT FOR EMPPROJECT
```

次の表は、列の内容を示しています。

表 66. 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

表 66. 社員プロジェクト活動表の列 (続き)

列名	説明
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

EMPPROJECT:

EMPPROJECT 表にあるデータの完全なリストを以下に示します。

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
000010	AD3100	10	.50	1982-01-01	1982-07-01
000070	AD3110	10	1.00	1982-01-01	1983-02-01
000230	AD3111	60	1.00	1982-01-01	1982-03-15
000230	AD3111	60	.50	1982-03-15	1982-04-15
000230	AD3111	70	.50	1982-03-15	1982-10-15
000230	AD3111	80	.50	1982-04-15	1982-10-15
000230	AD3111	180	.50	1982-10-15	1983-01-01
000240	AD3111	70	1.00	1982-02-15	1982-09-15
000240	AD3111	80	1.00	1982-09-15	1983-01-01
000250	AD3112	60	1.00	1982-01-01	1982-02-01
000250	AD3112	60	.50	1982-02-01	1982-03-15
000250	AD3112	60	1.00	1983-01-01	1983-02-01
000250	AD3112	70	.50	1982-02-01	1982-03-15
000250	AD3112	70	1.00	1982-03-15	1982-08-15
000250	AD3112	70	.25	1982-08-15	1982-10-15
000250	AD3112	80	.25	1982-08-15	1982-10-15
000250	AD3112	80	.50	1982-10-15	1982-12-01
000250	AD3112	180	.50	1982-08-15	1983-01-01
000260	AD3113	70	.50	1982-06-15	1982-07-01
000260	AD3113	70	1.00	1982-07-01	1983-02-01
000260	AD3113	80	1.00	1982-01-01	1982-03-01
000260	AD3113	80	.50	1982-03-01	1982-04-15
000260	AD3113	180	.50	1982-03-01	1982-04-15
000260	AD3113	180	1.00	1982-04-15	1982-06-01
000260	AD3113	180	1.00	1982-06-01	1982-07-01
000270	AD3113	60	.50	1982-03-01	1982-04-01
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

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
000270	AD3113	80	.50	1982-03-01	1982-04-01
000030	IF1000	10	.50	1982-06-01	1983-01-01
000130	IF1000	90	1.00	1982-10-01	1983-01-01
000130	IF1000	100	.50	1982-10-01	1983-01-01
000140	IF1000	90	.50	1982-10-01	1983-01-01
000030	IF2000	10	.50	1982-01-01	1983-01-01
000140	IF2000	100	1.00	1982-01-01	1982-03-01
000140	IF2000	100	.50	1982-03-01	1982-07-01
000140	IF2000	110	.50	1982-03-01	1982-07-01
000140	IF2000	110	.50	1982-10-01	1983-01-01
000010	MA2100	10	.50	1982-01-01	1982-11-01
000110	MA2100	20	1.00	1982-01-01	1983-03-01
000010	MA2110	10	1.00	1982-01-01	1983-02-01
000200	MA2111	50	1.00	1982-01-01	1982-06-15
000200	MA2111	60	1.00	1982-06-15	1983-02-01
000220	MA2111	40	1.00	1982-01-01	1983-02-01
000150	MA2112	60	1.00	1982-01-01	1982-07-15
000150	MA2112	180	1.00	1982-07-15	1983-02-01
000170	MA2112	60	1.00	1982-01-01	1983-06-01
000170	MA2112	70	1.00	1982-06-01	1983-02-01
000190	MA2112	70	1.00	1982-01-01	1982-10-01
000190	MA2112	80	1.00	1982-10-01	1983-10-01
000160	MA2113	60	1.00	1982-07-15	1983-02-01
000170	MA2113	80	1.00	1982-01-01	1983-02-01
000180	MA2113	70	1.00	1982-04-01	1982-06-15
000210	MA2113	80	.50	1982-10-01	1983-02-01
000210	MA2113	180	.50	1982-10-01	1983-02-01
000050	OP1000	10	.25	1982-01-01	1983-02-01
000090	OP1010	10	1.00	1982-01-01	1983-02-01
000280	OP1010	130	1.00	1982-01-01	1983-02-01
000290	OP1010	130	1.00	1982-01-01	1983-02-01
000300	OP1010	130	1.00	1982-01-01	1983-02-01
000310	OP1010	130	1.00	1982-01-01	1983-02-01
000050	OP2010	10	.75	1982-01-01	1983-02-01
000100	OP2010	10	1.00	1982-01-01	1983-02-01
000320	OP2011	140	.75	1982-01-01	1983-02-01
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

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
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

列名	説明
MAJPROJ	サブプロジェクトの管理プロジェクト番号

PROJECT:

PROJECT 表にあるデータの完全なリストを以下に示します。

PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
AD3100	統括管理	D01	000010	6.5	1982-01-01	1983-02-01	?
AD3110	総務	D21	000070	6	1982-01-01	1983-02-01	AD3100
AD3111	給与プログラミン グ	D21	000230	2	1982-01-01	1983-02-01	AD3110
AD3112	人事プログラミン グ	D21	000250	1	1982-01-01	1983-02-01	AD3110
AD3113	経理プログラミン グ	D21	000270	2	1982-01-01	1983-02-01	AD3110
IF1000	照会サービス	C01	000030	2	1982-01-01	1983-02-01	?
IF2000	ユーザー教育	C01	000030	1	1982-01-01	1983-02-01	?
MA2100	溶接ライン (WL) 自動化	D01	000010	12	1982-01-01	1983-02-01	?
MA2110	W L プログラミン グ	D11	000060	9	1982-01-01	1983-02-01	MA2100
MA2111	W L プログラム設 計	D11	000220	2	1982-01-01	1982-12-01	MA2110
MA2112	W L ロボット設計	D11	000150	3	1982-01-01	1982-12-01	MA2110
MA2113	W L 製造制御プロ グラム	D11	000160	3	1982-02-15	1982-12-01	MA2110
OP1000	オペレーション・ サポート	E01	000050	6	1982-01-01	1983-02-01	?
OP1010	オペレーション	E11	000090	5	1982-01-01	1983-02-01	OP1000
OP2000	汎用システム・サ ービス	E01	000050	5	1982-01-01	1983-02-01	?
OP2010	システム・サポー ト	E21	000100	4	1982-01-01	1983-02-01	OP2000
OP2011	SCP システム・サ ポート	E21	000320	1	1982-01-01	1983-02-01	OP2010
OP2012	アプリケーション・ サポート	E21	000330	1	1982-01-01	1983-02-01	OP2010
OP2013	DB/DC サポート	E21	000340	1	1982-01-01	1983-02-01	OP2010
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活動が記述されます。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含まれます。

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

PROJECT:

PROJECT 表にあるデータの完全なリストを以下に示します。

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	?

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
AD3112	70	?	1982-02-01	?
AD3112	70	?	1982-03-15	?
AD3112	70	?	1982-08-15	?
AD3112	80	?	1982-08-15	?
AD3112	80	?	1982-10-15	?
AD3112	180	?	1982-08-15	?
AD3113	70	?	1982-06-15	?
AD3113	70	?	1982-07-01	?
AD3113	80	?	1982-01-01	?
AD3113	80	?	1982-03-01	?
AD3113	180	?	1982-03-01	?
AD3113	180	?	1982-04-15	?
AD3113	180	?	1982-06-01	?
AD3113	60	?	1982-03-01	?
AD3113	60	?	1982-04-01	?
AD3113	60	?	1982-09-01	?
AD3113	70	?	1982-09-01	?
AD3113	70	?	1982-10-15	?
IF1000	10	?	1982-06-01	?
IF1000	90	?	1982-10-01	?
IF1000	100	?	1982-10-01	?
IF2000	10	?	1982-01-01	?
IF2000	100	?	1982-01-01	?
IF2000	100	?	1982-03-01	?
IF2000	110	?	1982-03-01	?
IF2000	110	?	1982-10-01	?
MA2100	10	?	1982-01-01	?
MA2100	20	?	1982-01-01	?
MA2110	10	?	1982-01-01	?
MA2111	50	?	1982-01-01	?
MA2111	60	?	1982-06-15	?
MA2111	40	?	1982-01-01	?
MA2112	60	?	1982-01-01	?
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	?

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
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:

ACT 表にあるデータの完全なリストを以下に示します。

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)

ACTNO	ACTKWD	ACTDESC
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:

CL_SCHED 表にあるデータの完全なリストを以下に示します。

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

CLASS_CODE	DAY	STARTING	ENDING
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、およびメッセージの内容が入っています。

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:

IN_TRAY 表にあるデータの完全なリストを以下に示します。

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

RECEIVED	SOURCE	SUBJECT	NOTE_TEXT
1988-12-23-08.53.58.000000	ISTERN	FWD: 良い年だ! 第 4 半期ボーナス。	To: Dept_D11 おめでとう。よくやった。ボーナスでエンジョイしてくれ給え。 Irv <Forwarding from CHAAS> Subject: 良い年だ! 第 4 半期ボーナス。 To: All_Managers 第 4 半期の結果が出た。チームで皆頑張って、計画値を超えた! 今年のボーナスは 18% だと発表できてうれしい。良い休暇を! Christine Haas
1988-12-22-14.07.21.136421	CHAAS	良い年だ! 第 4 半期ボーナス。	To: All_Managers 第 4 半期の結果が出た。チームで皆頑張って、計画値を超えた! 今年のボーナスは 18% だと発表できてうれしい。良い休暇を! Christine Haas

組織表 (ORG)

組織表は、企業の組織を記述します。

組織表は、以下の CREATE TABLE ステートメントを使用して作成します。

```
CREATE TABLE ORG
  (DEPTNUMB SMALLINT NOT NULL,
   DEPTNAME VARCHAR(14),
   MANAGER SMALLINT,
   DIVISION VARCHAR(10),
   LOCATION VARCHAR(13))
```

次の表は列の内容を示しています。

列名	説明
DEPTNUMB	部門番号
DEPTNAME	部門名
MANAGER	部門の管理者番号
DIVISION	部門内の部
LOCATION	部門の場所

ORG:

ORG 表にあるデータの完全なリストを以下に示します。

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

DEPTNUMB	DEPTNAME	MANAGER	DIVISION	LOCATION
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:

STAFF 表にあるデータの完全なリストを以下に示します。

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	?
20	Pernal	20	Sales	8	18171.25	612.45
30	Marengi	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	?

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
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:

SALES 表にあるデータの完全なリストを以下に示します。

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
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

SALES_DATE	SALES_PERSON	REGION	SALES
04/01/1996	GOUNOT	Ontario-North	1
04/01/1996	GOUNOT	Quebec	3
04/01/1996	GOUNOT	Manitoba	7

サンプルの XML 表

これらのサンプル表は、XML 概念を扱う際に使用できます。

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。

ストアド・プロシージャはシステムの一部として組み込まれており、すべての表を作成するデータ定義言語 (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。プロシージャは、プロシージャ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、外部ストアド・プロシージャであるので、対話式 SQL および System i ナビゲーターを含む、すべての SQL インターフェイスから呼び出すことができます。作成するスキーマが SAMPLEXML である場合、プロシージャを呼び出すには次のステートメントを実行します。

```
CALL QSYS.CREATE_XML_SAMPLE ('SAMPLEXML')
```

スキーマ名は、大文字で指定してください。スキーマがまだ存在しない場合には、作成されます。プロシージャを呼び出す前に、ジョブの CCSID は必ず 65535 以外の値に設定してください。そのようにならないと、エラーを受け取ります。

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。

製品表 (PRODUCT)

製品表は、製品 ID ですべての製品を識別する、製品の基本情報のリストです。

製品表は、以下の CREATE TABLE ステートメントを使用して作成します。

```
CREATE TABLE PRODUCT
( PID          VARCHAR(10) NOT NULL,
  NAME         VARCHAR(128),
  PRICE        DECIMAL(30,2),
  PROMOPRICE   DECIMAL(30,2),
  PROMOSTART   DATE,
  PROMOEND     DATE,
  DESCRIPTION  XML,
  PRIMARY KEY (PID) )
```

PRODUCT:

PRODUCT 表にあるデータの完全なリストを以下に示します。

PID	NAME	PRICE	PROMO PRICE	PROMO START	PROMO END	DESCRIPTION。 XMLSERIALIZE スカラー関数を使用して直列化文字データに変換したときの表示で示します。
100-100-01	Snow Shovel, Basic 22 inch	9.99	7.25	11/19/2004	12/19/2004	<product pid="100-100-01"> <description> <name>Snow Shovel, Basic 22 inch</name> <details>Basic Snow Shovel, 22 inches wide, straight handle with D-Grip</details> <price>9.99</price> <weight>1 kg</weight> </description> </product>
100-101-01	Snow Shovel, Deluxe 24 inch	19.99	15.99	12/18/2005	02/28/2006	<product pid="100-101-01"> <description> <name>Snow Shovel, Deluxe 24 inch</name> <details>A Deluxe Snow Shovel, 24 inches wide, ergonomic curved handle with D-Grip</details> <price>19.99</price> <weight>2 kg</weight> </description> </product>
100-103-01	Snow Shovel, Super Deluxe 26 inch	49.99	39.99	12/22/2005	02/22/2006	<product pid="100-103-01"> <description> <name>Snow Shovel, Super Deluxe 26 inch</name> <details>Super Deluxe Snow Shovel, 26 inches wide, ergonomic battery heated curved handle with upgraded D-Grip</details> <price>49.99</price> <weight>3 kg</weight> </description> </product>

PID	NAME	PRICE	PROMO PRICE	PROMO START	PROMO END	DESCRIPTION。XMLSERIALIZE スカラー関数を使用して直列化文字データに変換したときの表示で示します。
100-201-01	Ice Scraper, Windshield 4 inch	3.99	-	-	-	<pre> <product pid="100-201-01"> <description> <name>Ice Scraper, Windshield 4 inch</name> <details>Basic Ice Scraper 4 inches wide, foam handle</details> <price>3.99</price> </description> </product> </pre>

購入オーダー表 (PURCHASEORDER)

購入オーダー表は、すべての購入オーダーを購入オーダー ID で識別する、購入オーダーの基本情報のリストです。

購入オーダー表は、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 PURCHASEORDER
( POID          BIGINT NOT NULL,
  STATUS        VARCHAR(10) NOT NULL WITH DEFAULT 'Unshipped',
  CUSTID        BIGINT
  ORDERDATE    DATE
  PORDER       XML
  COMMENTS     VARCHAR(1000),
  PRIMARY KEY (POID) )

```

```

ALTER TABLE PURCHASEORDER
  ADD FOREIGN KEY FK_PO_CUST(CUSTID)
    REFERENCES CUSTOMER(CID)
  ON DELETE CASCADE

```

PURCHASEORDER:

PURCHASEORDER 表にあるデータの完全なリストを以下に示します。

POID	STATUS	CUSTID	ORDERDATE	PORDER。 XMLSERIALIZE スカラー関数を使用して直列化文字データに変換したときの表示で示します。	COMMENTS
5000	Unshipped	1002	02/18/2006	<pre><PurchaseOrder PoNum="5000" OrderDate="2006-02-18" Status="Unshipped"> <item> <partid>100-100-01</partid> <name>Snow Shovel, Basic 22 inch </name> <quantity>3</quantity> <price>9.99</price> </item> <item> <partid>100-103-01</partid> <name>Snow Shovel, Super Deluxe 26 inch </name> <quantity>5</quantity> <price>49.99</price> </item> </PurchaseOrder></pre>	THIS IS A NEW PURCHASE ORDER

POID	STATUS	CUSTID	ORDERDATE	PORDER。 XMLSERIALIZE スカラー関数を使用して直列化文字データに変換したときの表示で示します。	COMMENTS
5001	Shipped	1003	02/03/2005	<pre> <PurchaseOrder PoNum="5001" OrderDate="2005-02-03" Status="Shipped"> <item> <partid>100-101-01</partid> <name>Snow Shovel, Deluxe 24 inch </name> <quantity>1</quantity> <price>19.99</price> </item> <item> <partid>100-103-01</partid> <name>Snow Shovel, Super Deluxe 26 inch </name> <quantity>2</quantity> <price>49.99</price> </item> <item> <partid>100-201-01</partid> <name>Ice Scraper, Windshield 4 inch </name> <quantity>1</quantity> <price>3.99</price> </item> </PurchaseOrder> </pre>	THIS IS A NEW PURCHASE ORDER

POID	STATUS	CUSTID	ORDERDATE	PORDER。 XMLSERIALIZE スカラー関数を使用して直列化文字データに変換したときの表示で示します。	COMMENTS
5002	Shipped	1001	02/29/2004	<pre> <PurchaseOrder PoNum="5002" OrderDate="2004-02-29" Status="Shipped"> <item> <partid>100-100-01</partid> <name>Snow Shovel, Basic 22 inch </name> <quantity>3</quantity> <price>9.99</price> </item> <item> <partid>100-101-01</partid> <name>Snow Shovel, Deluxe 24 inch </name> <quantity>5</quantity> <price>19.99</price> </item> <item> <partid>100-201-01</partid> <name>Ice Scraper, Windshield 4 inch </name> <quantity>5</quantity> <price>3.99</price> </item> </PurchaseOrder> </pre>	THIS IS A NEW PURCHASE ORDER
5003	Shipped	1002	02/28/2005	<pre> <PurchaseOrder PoNum="5003" OrderDate="2005-02-28" Status="UnShipped"> <item> <partid>100-100-01</partid> <name>Snow Shovel, Basic 22 inch </name> <quantity>1</quantity> <price>9.99</price> </item> </PurchaseOrder> </pre>	THIS IS A NEW PURCHASE ORDER

POID	STATUS	CUSTID	ORDERDATE	PORDER。 XMLSERIALIZE スカラー関数を使用して直列化文字データに変換したときの表示で示します。	COMMENTS
5004	Shipped	1005	11/18/2005	<pre> <PurchaseOrder PoNum="5004" OrderDate="2005-11-18" Status="Shipped"> <item> <partid>100-100-01</partid> <name>Snow Shovel, Basic 22 inch </name> <quantity>4</quantity> <price>9.99</price> </item> < item> <partid>100-103-01</partid> <name>Snow Shovel, Super Deluxe 26 inch </name> <quantity>2</quantity> <price>49.99</price> </item> </PurchaseOrder> </pre>	THIS IS A NEW PURCHASE ORDER
5005	Shipped	1002	03/01/2006	<pre> <PurchaseOrder PoNum="5006" OrderDate="2006-03-01" Status="Shipped"> <item> <partid>100-100-01</partid> <name>Snow Shovel, Basic 22 inch </name> <quantity>3</quantity> <price>9.99</price> </item> <item> <partid>100-101-01</partid> <name>Snow Shovel, Deluxe 24 inch </name> <quantity>5</quantity> <price>19.99</price> </item> <item> <partid>100-201-01</partid> <name>Ice Scraper, Windshield 4 inch </name> <quantity>5</quantity> <price>3.99</price> </item> </PurchaseOrder> </pre>	THIS IS A NEW PURCHASE ORDER

顧客表 (CUSTOMER)

顧客表は、顧客 ID ですべての顧客を識別する、顧客の基本情報のリストです。

顧客表は、以下の CREATE TABLE ステートメントを使用して作成します。

```
CREATE TABLE CUSTOMER
( CID          BIGINT NOT NULL,
  INFO         XML,
  HISTORY      XML,
  PRIMARY KEY (CID) )
```

CUSTOMER:

CUSTOMER 表にあるデータの完全なリストを以下に示します。

CID	INFO。 XMLSERIALIZE スカラー関数を使用して直列化文字データに変換したときの表示で示します。	HISTORY
1000	<pre><customerinfo Cid="1000"> <name>Kathy Smith</name> <addr country="Canada"> <street>5 Rosewood</street> <city>Toronto</city> <prov-state>Ontario</prov-state> <pcode-zip>M6W 1E6</pcode-zip> </addr> <phone type="work">416-555-1358</phone> </customerinfo></pre>	?
1001	<pre><customerinfo Cid="1001"> <name>Kathy Smith</name> <addr country="Canada"> <street>25 EastCreek</street> <city>Markham</city> <prov-state>Ontario</prov-state> <pcode-zip>N9C 3T6</pcode-zip> </addr> <phone type="work">905-555-7258</phone> </customerinfo></pre>	?
1002	<pre><customerinfo Cid="1002"> <name>Jim Noodle</name> <addr country="Canada"> <street>25 EastCreek</street> <city>Markham</city> <prov-state>Ontario</prov-state> <pcode-zip>N9C 3T6</pcode-zip> </addr> <phone type="work">905-555-7258</phone> </customerinfo></pre>	?

CID	INFO。 XMLSERIALIZE スカラー関数を使用して直列化文字データに変換したときの表示で示します。	HISTORY
1003	<pre> <customerinfo Cid="1003"> <name>Robert Shoemaker</name> <addr country="Canada"> <street>1596 Baseline</street> <city>Aurora</city> <prov-state>Ontario</prov-state> <pcode-zip>N8X 7F8</pcode-zip> </addr> <phone type="work">905-555-7258</phone> <phone type="home">416-555-2937</phone> <phone type="cell">905-555-8743</phone> <phone type="cottage">613-555-3278</phone> </customerinfo> </pre>	?
1004	<pre> <customerinfo Cid="1004"> <name>Matt Foreman</name> <addr country="Canada"> <street>1596 Baseline</street> <city>Toronto</city> <prov-state>Ontario</prov-state> <pcode-zip>M3Z 5H9</pcode-zip> </addr> <phone type="work">905-555-4789</phone> <phone type="home">416-555-3376</phone> <assistant> <name>Gopher Runner</name> <phone type="home">416-555-3426</phone> </assistant> </customerinfo> </pre>	?
1005	<pre> <customerinfo Cid="1005"> <name>Larry Menard</name> <addr country="Canada"> <street>223 Nature Valley Road</street> <city>Toronto</city> <prov-state>Ontario</prov-state> <pcode-zip>M4C 5K8</pcode-zip> </addr> <phone type="work">905-555-9146</phone> <phone type="home">416-555-6121</phone> <assistant> <name>Goose Defender</name> <phone type="home">416-555-1943</phone> </assistant> </customerinfo> </pre>	?

カタログ表 (CATALOG)

カタログ表は、それぞれのカタログを記述します。

カタログ表は、以下の CREATE TABLE ステートメントを使用して作成します。

```
CREATE TABLE CATALOG
( NAME          VARCHAR(128) NOT NULL,
  CATLOG        XML,
  PRIMARY KEY (NAME) )
```

CATALOG:

CATALOG 表にはデータが入っていません。

提供者表 (SUPPLIERS)

提供者表は、すべての提供者を識別する、提供者の基本情報のリストです。

提供者表は、以下の CREATE TABLE ステートメントを使用して作成します。

```
CREATE TABLE SUPPLIERS
( SID          VARCHAR(10) NOT NULL,
  ADDR         XML,
  PRIMARY KEY (SID) )
```

SUPPLIERS:

SUPPLIERS 表にあるデータの完全なリストを以下に示します。

SID	ADDR
100	<pre><supplierinfo xmlns="http://posample.org" Sid="100"> <name>Harry Suppliers</name> <addr country="Canada"> <street>50 EastCreek</street> <city>Markham</city> <prov-state>Ontario</prov-state> <pcode-zip>N9C 3T6</pcode-zip> </addr> </supplierinfo></pre>
101	<pre><supplierinfo xmlns="http://posample.org" Sid="101"> <name>Johnston Suppliers</name> <addr country="Canada"> <street>302 NatureValley Road</street> <city>Toronto</city> <prov-state>Ontario</prov-state> <pcode-zip>M4C 5K8</pcode-zip> </addr> </supplierinfo></pre>

在庫表 (INVENTORY)

在庫表は、製品 ID に基づいて在庫を記述します。

在庫表は、以下の CREATE TABLE ステートメントを使用して作成します。

```
CREATE TABLE INVENTORY
( PID          VARCHAR(10) NOT NULL,
  QUANTITY     INTEGER,
  LOCATION     VARCHAR(128),
  PRIMARY KEY (PID) )
```

INVENTORY:

INVENTORY 表にあるデータの完全なリストを以下に示します。

PID	QUANTITY	LOCATION
100-100-01	5	-
100-101-01	25	Store
100-103-01	55	Store
100-201-01	99	Warehouse

製品提供業者表 (PRODUCTSUPPLIER)

製品提供業者表は、製品とその提供業者との間の関係を記述します。

製品提供業者表は、以下の CREATE TABLE ステートメントを使用して作成します。

```
CREATE TABLE PRODUCTSUPPLIER
( PID          VARCHAR(10) NOT NULL,
  SID          VARCHAR(10) NOT NULL,
  PRIMARY KEY (PID, SID) )
```

PRODUCTSUPPLIER:

PRODUCTSUPPLIER 表にあるデータの完全なリストを以下に示します。

PID	SID
100-101-01	100
100-201-01	101

DB2 for i CL コマンドの記述

DB2 for i は SQL のための以下の CL コマンドを提供します。

- SQL パッケージ作成 (CRTSQLPKG) コマンド
- SQL パッケージ削除 (DLTSQLPKG) コマンド
- SQL 情報印刷 (PRTSQLINF) コマンド
- SQL ステートメント実行 (RUNSQLSTM) コマンド
- SQL 対話式セッションの開始 (STRSQL)コマンド

関連資料:

328 ページの『DB2 for i 分散リレーショナル・データベース・サポート』

IBM DB2 Query Manager and SQL Development Kit for i ライセンス・プログラムは、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。

コードに関するライセンス情報および特記事項

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。

いかなる場合においても、IBM および IBM のサプライヤーならびに IBM ビジネス・パートナーは、その予見の有無を問わず発生した以下のものについて賠償責任を負いません。

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。

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。

〒103-8510

東京都中央区日本橋箱崎町19番21号

日本アイ・ビー・エム株式会社

法務・知的財産

知的財産権ライセンス渉外

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を提供し、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、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。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認できません。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をお願いします。

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ます、単に目標を示しているものです。

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。これらのサンプル・プログラムは特定物として現存するままの状態を提供されるものであり、いかなる保証も提供されません。IBM は、お客様の当該サンプル・プログラムの使用から生ずるいかなる損害に対しても一切の責任を負いません。

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_.

プログラミング・インターフェース情報

本書「SQL プログラミング」には、プログラムを作成するユーザーが IBM i 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

IBM、IBM ロゴおよび ibm.com は、世界の多くの国で登録された International Business Machines Corporation の商標です。他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。現時点での IBM の商標リストについては、『www.ibm.com/legal/copytrade.shtml』をご覧ください。

Adobe、Adobe ロゴ、PostScript、PostScript ロゴは、Adobe Systems Incorporated の米国およびその他の国における登録商標または商標です。

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

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

UNIX は The Open Group の米国およびその他の国における登録商標です。

Java およびすべての Java 関連の商標およびロゴは Oracle やその関連会社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。

他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。

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を提供され、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。



プログラム番号: 5770-SS1

Printed in Japan