



SQL Engine Reference

Zen v16

Activate Your Data™

Copyright © 2026 Actian Corporation. All Rights Reserved.

このドキュメントはエンドユーザーへの情報提供のみを目的としており、Actian Corporation（“Actian”）によりいつでも変更または撤回される場合があります。このドキュメントは Actian の専有情報であり、著作権に関するアメリカ合衆国国内法及び国際条約により保護されています。本ソフトウェアは、使用許諾契約書に基づいて提供されるものであり、当契約書の条件に従って使用またはコピーすることが許諾されます。いかなる目的であっても、Actian の明示的な書面による許可なしに、このドキュメントの内容の一部または全部を複製、送信することは、複写および記録を含む電子的または機械的のいかなる形式、手段を問わず禁止されています。Actian は、適用法の許す範囲内で、このドキュメントを現状有姿で提供し、如何なる保証も付しません。また、Actian は、明示的暗示的法的に関わらず、黙示的商品性の保証、特定目的使用への適合保証、第三者の有する権利への侵害等による如何なる保証及び条件から免責されます。Actian は、如何なる場合も、お客様や第三者に対して、たとえ Actian が当該損害に関してアドバイスを提供していたとしても、逸失利益、事業中断、のれん、データの喪失等による直接的間接的損害に関する如何なる責任も負いません。

このドキュメントは Actian Corporation により作成されています。

米国政府機関のお客様に対しては、このドキュメントは、48 C.F.R 第 12.212 条、48 C.F.R 第 52.227 条第 19(c)(1) 及び (2) 項、DFARS 第 252.227-7013 条または適用され得るこれらの後継的条項により限定された権利をもって提供されます。

Actian、Actian DataCloud、Actian DataConnect、Actian X、Avalanche、Versant、PSQL、Actian Zen、Actian Director、Actian Vector、DataFlow、Ingres、OpenROAD、および Vectorwise は、Actian Corporation およびその子会社の商標または登録商標です。本資料で記載される、その他すべての商標、名称、サービスマークおよびロゴは、所有各社に属します。

目次

このドキュメントについて	xv
SQL の概要	1
Zen で SQL を使用した作業	1
データ定義ステートメント	2
データ操作ステートメント	5
データ制御ステートメント	8
Zen メタデータ	10
リレーショナル エンジンの制限	11
完全修飾オブジェクト名	14
SQL ステートメント内のデリミター付き識別子	14
SQL 構文リファレンス	15
リテラル値	15
文字列値	15
数値	16
日付値	16
時刻値	16
タイムスタンプ値	17
Zen の SQL 文法	17
ADD	18
ALL	19
ALTER (名前変更)	20
ALTER GROUP	23
ALTER TABLE	26
IN DICTIONARY	28
USING	30
WITH REPLACE	32
MODIFY COLUMN と ALTER COLUMN	33
PSQL_MOVE	35
RENAME COLUMN	36
ON DELETE CASCADE	36
ALTER USER	42
ANY	44

AS	45
BEGIN [ATOMIC]	47
CALL	48
CASCADE	49
CASE (式).....	50
CASE (文字列).....	54
CLOSE	55
COALESCE	56
COMMIT	60
CREATE DATABASE	61
データベース名と IF NOT EXISTS 句	62
辞書パス.....	62
データパス	62
参照整合性.....	63
BOUND.....	63
辞書ファイル.....	63
セキュリティ.....	64
メタデータのバージョン.....	64
エンコード	65
CREATE FUNCTION.....	69
サポートされるスカラー入力パラメーターおよび戻り値のデータ型.....	70
スカラー ユーザー定義関数の呼び出し	72
CREATE GROUP	75
CREATE INDEX	76
インデックス セグメント	78
UNIQUE	79
PARTIAL.....	79
NOT MODIFIABLE.....	81
USING.....	82
IN DICTIONARY.....	82
CREATE PROCEDURE.....	85
信頼されるストアード プロシージャと信頼されないストアード プロシージャ....	87
メモリ キャッシング	87
CREATE TABLE	98
レコード サイズの制限	101
削除規則.....	102
更新規則.....	102
IN DICTIONARY.....	102

USING	103
WITH REPLACE.....	105
DCOMPRESS	105
PCOMPRESS.....	106
PAGESIZE	106
LINKDUP.....	106
CREATE (テンポラリ) TABLE	113
以前のリリースとの互換性	117
TEMPDB データベース.....	118
ローカル テンポラリ テーブルのテーブル名	118
トランザクション	118
SELECT INTO.....	119
SELECT INTO の制限	119
ストアド プロシージャのキャッシング	120
CREATE TRIGGER.....	123
CREATE USER	125
CREATE VIEW	128
ORDER BY	129
信頼されるビューと信頼されないビュー	129
DECLARE	133
DECLARE CURSOR.....	134
DEFAULT.....	136
DELETE (位置付け).....	141
DELETE.....	143
FROM 句	144
DISTINCT	146
DROP DATABASE	147
セキュリティで保護されたデータベース	147
DELETE FILES.....	148
DROP FUNCTION	149
DROP GROUP.....	150
DROP INDEX	151
部分インデックス	151
DROP PROCEDURE.....	153
DROP TABLE	154
DROP TRIGGER.....	156
DROP USER	157
DROP VIEW	159
END	160

EXECUTE.....	161
EXISTS.....	163
FETCH.....	165
FOREIGN KEY.....	167
GRANT.....	169
GRANT LOGIN TO.....	170
権限の制約.....	170
GRANT とデータのセキュリティ.....	173
ビューおよびストアド プロシージャに対する権限.....	174
GROUP BY.....	182
HAVING.....	183
IF.....	184
IN.....	186
INSERT.....	187
INSERT ON DUPLICATE KEY UPDATE.....	187
DEFAULT を使用した場合に発生するエラー.....	192
JOIN.....	194
LAG.....	200
LEAVE.....	202
LIKE、ILIKE と ESCAPE の使用.....	203
Unicode での ESCAPE.....	204
LOOP.....	207
NOT.....	208
OPEN.....	209
PARTIAL.....	210
PRIMARY KEY.....	211
PRINT.....	213
PUBLIC.....	215
RELEASE SAVEPOINT.....	216
RESTRICT.....	218
REVOKE.....	219
ROLLBACK.....	223
SAVEPOINT.....	225
SELECT.....	227
SELECT (INTO 付き).....	254
SET.....	256
SET ANSI_PADDING.....	257
SET CACHED_PROCEDURES.....	260
SET DECIMALSEPARATORCOMMA.....	264

SET DEFAULTCOLLATE.....	266
SET LEGACYTYPESALLOWED	270
SET OWNER.....	271
SET PASSWORD.....	274
SET PROCEDURES_CACHE	277
SET ROWCOUNT.....	281
SET SECURITY	283
SET TIME_ZONE.....	286
SET TRUEBITCREATE	292
SET TRUENULLCREATE	294
SIGNAL	296
SQLSTATE.....	298
START TRANSACTION.....	299
UNION.....	300
UNIQUE.....	302
★ UNMASK	303
UPDATE.....	303
UPDATE (位置付け).....	309
USER	311
WHILE.....	312
文法要素の定義	313
グローバル変数	319
ほかの特性	323
テンポラリ ファイル.....	323
NULL 値を使った作業	325
バイナリ データを使った作業.....	325
インデックスの作成	326
小数点の記号のカンマ	327

スカラー関数 **331**

ビット演算子	331
真理値表	334
算術演算子	334
日付演算	334
例	334
文字列関数	334
例	338
数値関数	341

例.....	342
時刻と日付の関数.....	342
時刻と日付関数の例.....	354
システム関数.....	356
システム関数の例.....	357
論理関数.....	357
論理関数の例.....	358
変換関数.....	359
変換関数の例.....	361
システム ストアド プロシージャ	363
Zen システム ストアド プロシージャ	363
psp_columns	363
psp_column_attributes	368
psp_column_rights	371
★ psp_dbnames	374
psp_fkeys.....	376
psp_groups	378
psp_help_sp.....	380
psp_help_trigger	383
psp_help_udf.....	384
psp_help_view.....	386
psp_indexes.....	388
psp_pkeys	390
psp_procedure_rights	392
psp_rename	394
psp_stored_procedures	396
psp_tables	399
psp_table_rights	402
psp_triggers.....	404
psp_udfs	406
psp_users.....	407
psp_view_rights	409
psp_views	412
パフォーマンスの最適化リファレンス	415
制限の分析.....	416
変形 CNF 変換.....	416

変換できない制限	416
変換が行われない条件	416
制限の最適化	418
単一述部の最適化	418
閉じた範囲の最適化	419
変形論理和の最適化	419
論理積の最適化	419
論理和標準形の最適化	420
変形論理積標準形の最適化	420
変形 CNF 最適化で開いた範囲を閉じる	421
単一結合条件の最適化	422
結合条件を持つ論理積の最適化	422
結合条件を持つ変形論理積標準形の最適化	423
変形 CNF 最適化で結合条件の開いた範囲を閉じる	423
複数インデックスの変形論理和の最適化	423
プッシュダウン フィルター	425
インデックスの有効な使用法	426
集計関数の DISTINCT	426
選択リストに先立つ DISTINCT	426
インデックスのセグメント順序	426
セグメントの昇順属性	427
検索更新の最適化	428
テンポラリ テーブルのパフォーマンス	430
行の事前取得	431
用語	433
集計関数	433
閉じた範囲	433
論理積	433
論理積標準形 (CNF)	433
論理和	434
論理和標準形 (DNF)	434
式	434
インデックス	434
結合条件	434
先頭セグメント	435
変形論理積標準形 (変形 CNF)	435

変形論理和.....	435
開いた範囲.....	435
述部.....	436
制限.....	436
システム カタログ関数	437
Zen システム カタログ関数.....	437
リターン ステータス	438
まとめ.....	439
dbo.fSQLColumns.....	439
構文.....	439
引数	440
返される結果セット	440
例.....	443
dbo.fSQLForeignKeys	443
構文.....	444
引数	444
返される結果セット	445
例.....	446
dbo.fSQLPrimaryKeys	446
構文.....	447
引数	447
返される結果セット	447
例.....	448
dbo.fSQLProcedures.....	448
構文.....	448
引数	449
返される結果セット	450
例.....	450
dbo.fSQLProcedureColumns	452
構文.....	452
引数	452
返される結果セット	453
例.....	456
dbo.fSQLSpecialColumns.....	457
構文.....	457
引数	458

返される結果セット	458
例	459
dbo.fSQLStatistics	459
構文	460
引数	461
返される結果セット	461
例	463
dbo.fSQLTables	464
構文	464
引数	464
返される結果セット	465
例	465
dbo.fSQLDBTableStat	466
構文	466
引数	466
返される結果セット	466
例	467
文字列検索パターン	468
例	468

A. データ型 471

Zen で使用できるデータ型	471
データ型の範囲	475
演算子の優先順位	477
データ型の優先順位	478
Decimal データ型の精度と小数位	480
タイムスタンプ データ型の小数位と返される関数値	481
切り捨て	482
データ型に関する注意事項	482
CHAR、NCHAR、VARCHAR、NVARCHAR、LONGVARCHAR、および NLONGVARCHAR	483
BINARY および LONGVARBINARY	483
LONGVARCHAR、NLONGVARCHAR、および LONGVARBINARY の制約	484
DATETIME	485
UNIQUEIDENTIFIER	487
無限の表現	488
旧データ型	489

Btrieve キーのデータ型	489
AUTOINCREMENT	490
AUTOTIMESTAMP	493
BFLOAT	495
STRING	496
CURRENCY	496
DATE	496
DECIMAL	497
FLOAT	498
GUID	499
INTEGER	499
LOGICAL	500
LSTRING	500
MONEY	500
NUMERIC	501
NUMERICSA	503
NUMERICSLB	503
NUMERICSLS	504
NUMERICSTB	504
NUMERICSTS	505
TIME	505
TIMESTAMP	505
TIMESTAMP2	508
UNSIGNED BINARY	510
WSTRING	510
WZSTRING	510
ZSTRING	511
キーでないデータ型	511
BLOB	511
CLOB	512
B. SQL の予約語	513
予約語	513
使用を避ける語句	519
C. システム テーブル	523
概要	523
システム テーブルの構造	525
メタデータ バージョン 1 のシステム テーブル	525

メタデータ バージョン 2 のシステム テーブル	541
D. COBOL アプリケーションの SQL アクセス	561
COBOL の Zen サポートの概要	561
制限	561
SQL ステートメント	562
コンポーネント	564
SQL アクセスの使用	566
手順 1 : サンプル XML テンプレートの変更	566
手順 2 : XML テンプレートで指定したデータ ファイルのコピー	567
手順 3 : Schema Executor ユーティリティの実行	567
手順 4 : システム テーブルの配布 (任意)	572
サンプル XML ファイルの実行例	573
補注	573
E. Query Plan Viewer	577
Query Plan の設定	577
グラフィカル ユーザー インターフェイス	579
Query Viewer	579
Plan Viewer	580
Query Plan Viewer のタスク	583
クエリ プランの検査およびクエリ パフォーマンスの評価	590
比較するクエリ プランの例を作成する	591
クエリ プランの例を表示する	592

このドキュメントについて

このドキュメントでは、Zen クエリ言語の適用範囲と機能について説明します。

このドキュメントの読者

このドキュメントでは、Zen データベースで SQL スクリプトを作成および実行するための情報を提供します。

メモ： 特段の記述がない限り、本書における Zen 製品へのすべてのリファレンスは、現行バージョンについて述べています。

詳細情報

ODBC 仕様の詳細については、Microsoft ODBC ドキュメントを参照してください。

SQL の概要

以下のトピックでは、SQL の概要を提示し、SQL に対する Zen サポートの詳細について説明します。

- [Zen で SQL を使用した作業](#)
- [Zen メタデータ](#)
- [リレーショナル エンジンの制限](#)

[SQL 構文リファレンス](#)へ移動して、Zen でサポートされる特定の SQL 文法を調べることができます。

Zen で SQL を使用した作業

構造化問い合わせ言語（SQL）は、英語に似たステートメントを使用してデータベース操作を実行します。ANSI（American National Standards Institute）と IBM は、それぞれ独自の SQL 標準を定義しています。IBM の標準は SAA（Systems Application Architecture）と呼ばれます。Zen は、ANSI SQL と IBM SAA SQL の両方の機能をほとんど実装しているほか、これらの標準では定義されていない独自の拡張機能も提供しています。次の表に、Zen で作成できる SQL ステートメントと、その各種類のステートメントを使って実現できるタスクの一覧を示します。

SQL ステートメントの種類	各種作業
データ定義	テーブルの作成、変更、および削除 ビューの作成と削除 インデックスの作成と削除 ストアド SQL プロシージャの作成と削除 トリガーの作成と削除 ユーザー定義関数の作成と削除
データ処理	テーブル内のデータの取得、挿入、更新、および削除 トランザクションの定義 ビューの定義と削除 ストアド SQL プロシージャの実行 トリガーの実行

SQL ステートメントの種類	各種作業
データ制御	辞書のセキュリティの有効化と無効化 ユーザーの作成と削除 グループに対するユーザーの追加と削除 ユーザー パスワードの変更 テーブルのアクセス権の付与と取り消し

このトピックの残り部分では、各種類の SQL ステートメントに関する一般情報を提供します。各ステートメントの詳細については、[SQL 構文リファレンス](#)を参照してください。

- [データ定義ステートメント](#)
- [データ操作ステートメント](#)
- [データ制御ステートメント](#)

メモ：ほとんどの SQL エディターは、複数のステートメントを実行するためにステートメント区切り文字を使用しませんが、ZenCC の SQL Editor は区切り文字を必要とします。ここで提供する例をほかの環境で実行するには、シャープ記号またはセミコロンの区切り文字を取り除く必要があります。

データ定義ステートメント

データ定義ステートメントではデータベースの特性を指定できます。これを使ってデータ定義ステートメントを実行すると、Zen によってデータベースの記述がデータ辞書に格納されます。情報の格納や取得を行う前に、辞書内でデータベースを定義する必要があります。

Zen では、データ定義ステートメントを作成して、次のタスクを実行できます。

- テーブルの作成、変更、および削除
- ビューの作成と削除
- インデックスの作成と削除
- トリガーの作成と削除
- ストアド プロシージャの作成と削除
- ユーザー定義関数の作成と削除

以下のトピックでは、これらのタスクに関連する SQL ステートメントの概要について説明します。データベースの特性の定義に関する一般的な情報については、『*Zen Programmer's Guide*』を参照してください。

テーブルの作成、変更、および削除

データベースのテーブルを作成、変更、および削除するには、次の SQL ステートメントを使用します。

CREATE TABLE	テーブルを定義し、オプションで対応するデータ ファイルを作成します。
ALTER TABLE	テーブル定義に変更を 1 つ追加します。ALTER TABLE ステートメントを使用すると、テーブル定義への列の追加、テーブル定義からの列の削除、列のデータ型やデータ長（あるいは、それ以外の特性）の変更、主キーまたは外部キーの追加や削除、および別のデータ ファイルとテーブル定義との関連付けを行うことができます。
DROP TABLE	データ辞書からテーブルを削除し、オプションで関連するデータ ファイルを削除します。

ビューの作成と削除

データベースのビューを作成および削除するには、次の SQL ステートメントを使用します。

CREATE VIEW	新規ビューを定義します。
DROP VIEW	ビューを削除します。

インデックスの作成と削除

データベースのインデックスを作成および削除するには、次の SQL ステートメントを使用します。

CREATE INDEX	既存のテーブルに新しいインデックス（名前付きインデックス）を定義します。
DROP INDEX	名前付きインデックスを削除します。

トリガーの作成と削除

データベースのトリガーを作成および削除するには、次の SQL ステートメントを使用します。

CREATE TRIGGER	既存のテーブルにトリガーを定義します。

DROP TRIGGER	トリガーを削除します。

Zen は、その他の SQL 制御ステートメントも提供しています。これらのステートメントは、トリガー本体内でのみ使用できます。トリガーでは次のステートメントを使用できます。

BEFORE	INSERT、UPDATE、または DELETE オペレーションの前に実行するトリガーを定義します。

AFTER	INSERT、UPDATE、または DELETE オペレーションの後で実行するトリガーを定義します。

ストアド プロシージャの作成と削除

ストアド プロシージャは、あらかじめコンパイルして辞書に保存されている複数のステートメントから成っています。ストアド プロシージャを作成および削除するには、次の SQL ステートメントを使用します。

CREATE PROCEDURE	新しいプロシージャをデータ辞書に保管します。

DROP PROCEDURE	データ辞書からストアド プロシージャを削除します。

Zen は、その他の SQL 制御ステートメントも提供しています。これらのステートメントは、ストアド プロシージャ本体内でのみ使用できます。ストアド プロシージャでは次のステートメントを使用できます。

IF...THEN...ELSE	条件の真偽値に基づいたステートメントの実行制御を可能にします。

LEAVE	ブロックまたはループ ステートメントを抜けて実行を継続します。

LOOP	ステートメントのブロックの実行を繰り返します。

WHILE	指定した条件が True である間、ステートメントのブロックの実行を繰り返します。

ユーザー定義関数（UDF）の作成と削除

組み込み関数に加えて、Zen では独自にユーザー定義関数（UDF）を作成し、それを SQL クエリで 사용할 ことができます。

ユーザー定義関数は一種のデータベース オブジェクトで、再使用できる 1 つ以上の SQL ステートメントをカプセル化したものです。ユーザー定義関数は 0 個以上の入力引数を取り、スカラー戻り値を評価します。

ユーザー定義関数はデータベースのコンテキスト内に定義されます。**CREATE FUNCTION** ステートメントの実行が成功すると、それが実行されたデータベース内に UDF 定義が格納されます。格納された UDF を変更、呼び出し、削除することができます。

UDF では、1 つ以上の SQL ステートメントを使用して単一のスカラー値を出力することができます。この値のデータ型は、**CREATE FUNCTION** ステートメントの **RETURNS** 句で指定されます。サポートされるデータ型の一覧については、[サポートされるスカラー入力パラメーターおよび戻り値のデータ型](#)を参照してください。

ユーザー定義関数を作成および削除するには、次の表にあるステートメントを使用します。

CREATE FUNCTION	スカラー ユーザー定義関数をデータベースに作成します。
DROP FUNCTION	データベースからスカラー ユーザー定義関数を削除します。

データ操作ステートメント

データ操作ステートメントを使用すると、データベースの内容にアクセスして変更を加えることができます。Zen ではデータ操作ステートメントを構築して、次のタスクを実行できます。

- テーブルからのデータの取得
- テーブル内のデータの変更
- トランザクションの定義
- ビューの作成と削除

-
- ストアド プロシージャの実行
 - トリガーの実行

次のセクションでは、これらのタスクに関連する SQL ステートメントの概要について説明します。

データの取得

データベースから情報を取得するために使用するステートメントはすべて **SELECT** に基づきます。

SELECT	データベース内の 1 つまたは複数のテーブルからデータを取得します。

SELECT ステートメントを作成する場合、さまざまな句を使って、データを取得する際のさまざまなオプションを指定できます。次の表に、**SELECT** ステートメントで使われる句の種類を示します。

FROM	データを取得するテーブルまたはビューを指定します。
WHERE	SELECT ステートメントで取得するデータを修飾する検索条件を定義します。
GROUP BY	指定した条件に従って行のセットを組み合わせ、グループ内の 1 つ以上の列を対象に集計値を取得できるようにします。
HAVING	グループの集計値が満たさなければならない条件を指定して、ビューの内容を制限します。
ORDER BY	Zen が選択した行を返す順序を決定します。

また、**UNION** キーワードを使って、複数の **SELECT** クエリから 1 つの結果テーブルを取得することもできます。

データの変更

次の表では、テーブルやビューのデータを追加、変更、または削除するステートメントを示します。

INSERT	1 つ以上のテーブルまたは 1 つのビューに行を追加します。
UPDATE	テーブルまたはビューのデータを変更します。

DELETE	テーブルまたはビューから行を削除します。

DELETE または UPDATE ステートメントを作成する場合、WHERE 句を使って、目的のデータを制限する検索条件を定義することもできます。

ビューの作成と削除

ビューを作成および削除するには、次の SQL ステートメントを使用します。

CREATE VIEW	データベース ビューを定義し、辞書にその定義を格納します。

DROP VIEW	データ辞書からビューを削除します。

ストアド プロシージャの作成

ストアド プロシージャは、あらかじめコンパイルして辞書に保存されている複数のステートメントから成っています。ストアド プロシージャを実行するには、次の SQL ステートメントを使用します。

CALL または EXEC[UTE]	あらかじめコンパイルされたプロシージャを呼び出して実行します。

システム ストアド プロシージャの実行

システム ストアド プロシージャは、データ定義言語（DDL）では取り扱われない管理タスクや情報タスクの遂行に利用できます。システム ストアド プロシージャには **psp_** プレフィックスが付きます。システム ストアド プロシージャを実行するには、次の SQL ステートメントを使用します。

CALL または EXEC[UTE]	システム ストアド プロシージャを呼び出して実行します。

詳細については、[システム ストアド プロシージャ](#)を参照してください。

トリガーの実行

トリガーは、あらかじめコンパイルして辞書に保存されている複数のステートメントから成っています。トリガーは、指定した条件になると、エンジンにより自動的に実行されます。

データ制御ステートメント

データ制御ステートメントではデータベースのセキュリティを定義できます。辞書を作成する場合、その辞書のセキュリティを明示的に有効にするまで、その辞書のセキュリティは定義されません。Zen では、データ制御ステートメントを構築して、次のタスクを実行できます。

- セキュリティの有効化または無効化
- ユーザーとグループの作成と削除
- グループからユーザーを追加または削除し、ユーザー パスワードを変更する。
- アクセス権の付与と取り消し

メモ： 保護されたデータベース内のテーブルであるファイルに **Btrieve** オーナー ネームが設定されている場合、データベースの **Master** ユーザーは、そのテーブルに対するアクセス許可を任意のユーザー（**Master** ユーザーを含む）に与える場合、**GRANT** ステートメントにオーナー ネームを含める必要があります。

次のセクションでは、これらのタスクに関連する SQL ステートメントの概要について説明します。

セキュリティの有効化と無効化

データベースのセキュリティを有効または無効にするには、次のステートメントを使用します。

SET SECURITY	データベースのセキュリティを有効または無効にし、 Master パスワードを設定します。

ユーザーとグループの作成と削除

データベースのユーザーとユーザー グループを作成または削除するには、次の SQL ステートメントを使用します。

ALTER USER	ユーザーの名前を変更したり、パスワードを変更したりします。
CREATE USER	パスワードまたはメンバーシップを持つかまたは持たない新規ユーザーをグループに作成します。
DROP USER	ユーザーを削除します。
ALTER GROUP	ユーザーをグループに追加します。グループからユーザーを削除します。
CREATE GROUP	新しいユーザーのグループを作成します。
DROP GROUP	ユーザーのグループを削除します。
GRANT LOGIN TO	ユーザーとパスワードを作成するか、ユーザーをグループに追加します。
REVOKE LOGIN FROM	辞書からユーザーを削除します。

アクセス権の付与と取り消し

ユーザーやグループにアクセス権を付与または取り消すには、次のステートメントを使用します。

GRANT (アクセス権)	指定した種類のアクセス権をユーザーまたはグループに付与します。GRANT (アクセス権) ステートメントで付与できるアクセス権は、All、Insert、Delete、Alter、Select、Update、および References です。
GRANT CREATETAB TO	テーブルを作成するためのアクセス権をユーザーまたはグループに付与します。
REVOKE (アクセス権)	アクセス権をユーザーまたはグループから取り消します。
REVOKE CREATETAB FROM	テーブルを作成するためのアクセス権をユーザーまたはグループから取り消します。

Zen メタデータ

Zen のリレーショナル インターフェイスでは、メタデータでバージョン 1 (V1) とバージョン 2 (V2) という 2 つのバージョンをサポートします。

メタデータのバージョンはデータベース プロパティの 1 つで、データベースの作成時に指定します。デフォルトは "メタデータ バージョン 1" です。"メタデータ バージョン 2" にする場合は、データベースを作成する際に指定する必要があります。

メタデータのバージョンは、そのデータベース内のすべてのデータ辞書ファイル (DDF) に適用されます。単独のデータベースで、メタデータ バージョン 1 を使用する DDF とメタデータ バージョン 2 を使用する DDF を一緒に使用することはできません。2 つのバージョンの DDF はそれぞれ情報をやり取りすることができません。

ただし、データベース エンジンで複数のデータベースへ同時にアクセスすれば、そのデータベースごとにメタデータ バージョン 1 またはメタデータ バージョン 2 を使用することができます。

PSQL v10 より前の Zen バージョンで作成されたデータベースはすべてメタデータ バージョン 1 を使用します。PSQL v10 以降を使って作成されたデータベースは、データベース作成時の設定に応じてどちらかのメタデータ バージョンを使用しています。

メタデータのバージョンの比較

メタデータ バージョン 2 では、多くの識別子名に最大 128 バイトの名前を付けることができます。追加情報については、[リレーショナル エンジンの制限](#)を参照してください。またメタデータ バージョン 2 では、ビューおよびストアード プロシージャに対する権限が認められます。[ビューおよびストアード プロシージャに対する権限](#)を参照してください。

メタデータ バージョン 2 の DDF 名はバージョン 1 用とは異なります。バージョン 2 の DDF には、追加フィールドやバージョン 1 のフィールドへの変更が含まれています。[システム テーブル](#)を参照してください。

リレーショナル エンジンの制限

次の表は、リレーショナル エンジンの機能に適用される制限または条件を示します。
Zen データベースには、40 億のオブジェクトをさまざまに組み合わせて含むことができます。オブジェクトはデータ辞書ファイルに保存されます。

『Zen Programmer's Guide』の[名前付け規則](#)も参照してください。

Zen 機能	制限または条件	メタデータ	
		V1	V2
ストアド プロシージャのパラメーター リスト内の引数の数	300	X	X
CHAR 列のサイズ	8,000 バイト ¹	X	X
文字列リテラル	文字列値 を参照してください。	X	X
テーブル内の列数	1536	X	X
トリガーまたはストアド プロシージャで使用可能な列数	300	X	X
列名 ²	20 バイト	X	
	128 バイト		X
列のサイズ	2 GB	X	X
相関名	メモリによる制限	X	X
カーソル名	18 バイト	X	X
データベース名 ²	20 バイト	X	X
データベース セッション数	メモリによる制限	X	X
データ ファイル パス名	64 バイト (データ ファイル パス名の最大長は Xf\$Loc パスおよびデータ ファイル パスの組み合わせです)	X	
	250 バイト (データ ファイル パス名の最大長は Xf\$Loc パスおよびデータ ファイル パスの組み合わせです)		X

Zen 機能	制限または条件	メタデータ	
		V1	V2
関数（ユーザー定義）名 ²	30 バイト	X	
	128 バイト		X
グループ名 ²	30 バイト	X	
	128 バイト		X
インデックス名 ²	20 バイト	X	
	128 バイト		X
キー名 ²	20 バイト	X	
	128 バイト		X
ラベル名	メモリによる制限	X	X
NCHAR 列のサイズ	4,000 UCS-2 単位 (8,000 バイト ¹)	X	X
NVARCHAR 列のサイズ	4,000 UCS-2 単位 (8,000 バイト ¹)	X	X
AND および OR でつないだ述部の数	3000	X	X
データベース オブジェクトの数	65536	X	
	40 億		X
パラメーター名	126 バイト	X	X
パスワード ²	8 バイト	X	
	128 バイト		X
プロシージャ名 ²	30 バイト	X	
	128 バイト		X
参照整合性 (RI) の制約名	20 バイト	X	
	128 バイト		X
一重引用符の表記	2 つの連続した一重引用符 (")	X	X
結果名	メモリによる制限	X	X
セーブポイント名	メモリによる制限	X	X

Zen 機能	制限または条件	メタデータ	
		V1	V2
クエリ内の SELECT リストの列数	1600	X	X
SQL ステートメント内の単独項目 (引用符で囲まれたリテラル文字列) のサイズ	ヌル終端文字と引用符を除き、 14,997 (合計 15,000)	X	X
SQL ステートメントの長さ	512 KB	X	X
セッションあたりの SQL ステートメント数	メモリによる制限	X	X
ストアド プロシージャのサイズ	64 KB	X	X
テーブル名 ²	20 バイト	X	
	128 バイト		X
テーブルの行数	13.0 または 16.0 ファイル形式 : 9,223,372,036,854,775,807 (~920 京) 古いファイル形式 : 2,147,483,647 (~21 億)	X	X
クエリあたりの結合テーブル数	メモリによる制限	X	X
トリガー名 ²	20 バイト	X	
	128 バイト		X
ユーザー名 ²	30 バイト	X	
	128 バイト		X
VARCHAR 列のサイズ	8,000 バイト ¹	X	X
変数名	メモリによる制限	X	X
ビュー名 ²	20 バイト	X	
	128 バイト		X

¹ CHAR、NCHAR、VARCHAR、または NVARCHAR 列で、完全にインデックス化できる列の最大サイズは 255 バイト (16.0 より前の形式の場合)、または 1024 バイト (16.0 形式のファイルの場合) です。

² 『Advanced Operations Guide』の[識別子の制限](#)も参照してください。

完全修飾オブジェクト名

完全修飾されたオブジェクト名は、ドット表記を使用してデータベース名とオブジェクト名を結合します。たとえば、データベース `mydbase` に `myview` というビューがある場合、完全修飾オブジェクト名は `mydbase.myview` となります。

完全修飾オブジェクト名はデータベース内で一意である必要があります。たとえば、データベース `mydbase` にテーブル `acctpay` とユーザー定義関数 `acctpay` がある場合、Zen は名前 `mydbase.acctpay` を解決できません。

SQL ステートメント内のデリミター付き識別子

テーブル、列、およびインデックスにスペースや非標準の文字が含まれている場合や、識別子がキーワードである場合は、デリミターを付ける必要があります。デリミター文字は二重引用符です。

例

```
SELECT "last-name" FROM "non-standard-tbl"
```

ハイフンは標準文字ではありません。

```
SELECT "password" FROM my_pword_tbl
```

"password" は SET PASSWORD ステートメントのキーワードです。

SQL 構文リファレンス

以下のトピックでは、Zen でサポートされる SQL 構文について説明します。

- [リテラル値](#)
- [Zen の SQL 文法](#)
- [文法要素の定義](#)
- [グローバル変数](#)
- [ほかの特性](#)

リテラル値

Zen は、標準のリテラル書式をサポートしています。このトピックでは、最も一般的な例をいくつか示します。

- [文字列値](#)
- [数値](#)
- [日付値](#)
- [時刻値](#)
- [タイムスタンプ値](#)

文字列値

文字列定数は、指定する文字列を一重引用符で囲むことによって、SQL ステートメント内に表記できます。文字列自体が一重引用符またはアポストロフィを含んでいる場合は、その文字の前にもう 1 つ一重引用符を付ける必要があります。

文字列リテラルは `VARCHAR` 型を持ちます。文字は、データベース コード ページを使用してエンコードされます。リテラルの先頭に文字 "N" が付いている場合、そのリテラルは `NVARCHAR` 型を持ち、文字は UCS-2 を使用してエンコードされます。SQL クエリ文字列に埋め込まれたリテラルは、SQL エンジンで最終的に変換する前に、SQL アクセス方法で追加のエンコード変換の手続きを踏むことがあります。特に、SQL テキストがすべての `Unicode` 文字をサポートしていないエンコードに変換されている場合は、エンジンが文字列リテラルを `NVARCHAR` へ変換する前に、SQL テキスト内の文字が失われることがあります。

例

最初の例では、文字列内に含まれるアポストロフィまたは一重引用符は、もう 1 つの一重引用符によってエスケープする必要があります。

```
SELECT * FROM t1 WHERE c1 = 'Roberta''s Restaurant'  
SELECT STREET FROM address WHERE city LIKE 'san%'
```

数値

日付値

日付定数は、SQL ステートメント内に文字列として表記するか、ベンダー文字列に埋め込むことができます。最初のケースは **CHAR** 型の文字列として、ベンダー文字列表記は **DATE** 型の値として扱われます。この違いは変換時に重要になります。

Zen では、この関数で概説しているように、拡張 SQL 文法を一部サポートしています。

Zen でサポートしている日付リテラル書式は 'YYYY-MM-DD' です。

日付は 0 年から 9999 年の範囲で指定できます。

例

次の 2 つのステートメントによって、開始日が 1995 年 6 月 5 日より後のすべての授業が返されます。

```
SELECT * FROM Class WHERE Start_Date > '1995-06-05'  
SELECT * FROM Class WHERE Start_Date > {d '1995-06-05'}
```

時刻値

Zen でサポートしている時刻リテラル書式は 'HH:MM:SS' です。

時刻定数は、SQL ステートメント内に文字列として表記するか、ベンダー文字列に埋め込むことができます。文字列表記は **CHAR** 型の文字列として、ベンダー文字列表記は **TIME** 型の値として扱われます。

Zen では、この関数で概説しているように、拡張 SQL 文法を一部サポートしています。

例

次の 2 つのステートメントによって、Class テーブルから、授業の開始時刻が 14:00:00 のレコードが取り出されます。

```
SELECT * FROM Class WHERE Start_time = '14:00:00'  
SELECT * FROM Class WHERE Start_time = {t '14:00:00'}
```

タイムスタンプ値

タイムスタンプ定数は、SQL ステートメント内に文字列として表記するか、ベンダー文字列に埋め込むことができます。Zen では、文字列表記は CHAR 型の文字列として、ベンダー文字列表記は SQL_TIMESTAMP 型の値として扱われます。

Zen でサポートしているタイムスタンプ リテラル書式は 'YYYY-MM-DD HH:MM:SS.MMM' です。

例

次の 2 つのステートメントによって、Billing テーブルから、ログの開始日時が 1996-03-28 の 17:40:49 のレコードが取り出されます。

```
SELECT * FROM Billing WHERE log = '1996-03-28 17:40:49'  
SELECT * FROM Billing WHERE log = {ts '1996-03-28 17:40:49'}
```

Zen の SQL 文法

以下のトピックでは、Zen でサポートされる SQL 文法について説明します。ステートメントおよびキーワードがアルファベット順に記載されています。

メモ：ほとんどの SQL 例は、Zen Control Center に備わっている SQL Editor を使ってテストすることができます。例外については、文法要素の解説に示されています。詳細については、『Zen User's Guide』の [SQL Editor](#) を参照してください。

メモ：よく使われる SQL エディターのほとんどは、複数のステートメントを実行するためにステートメント区切り文字を使用しません。しかし、ZenCC の SQL Editor は区切り文字を必要とします。ほかの環境で例を実行する場合は、シャープ記号またはセミコロンの区切り文字を取り除く必要があります。

ADD

備考

ADD 句は ALTER TABLE ステートメント内で使用し、追加する列の定義、列の制約、またはテーブルの制約を 1 つまたは複数指定します。

関連項目

[ALTER TABLE](#)

[FOREIGN KEY](#)

[PRIMARY KEY](#)

ALL

備考

サブクエリの前に **ALL** キーワードを指定すると、**Zen** はサブクエリを実行し、その結果を使用して外部クエリの条件を評価します。サブクエリが返すすべての行が、特定の行に対する外部クエリの条件を満たす場合、**Zen** はステートメントの最終的な結果テーブルにその行を含みます。

一般に、**ALL** の代わりに **EXISTS** キーワードまたは **NOT EXISTS** キーワードを使用できます。

例

次の **SELECT** ステートメントは、**Person** テーブルの **ID** 列と、サブクエリの結果テーブルの **ID** 列を比較します。

```
SELECT p.ID, p.Last_Name
FROM Person p
WHERE p.ID <> ALL
(SELECT f.ID FROM Faculty f WHERE f.Dept_Name = 'Chemistry')
```

Person テーブルの **ID** 値と一致する値がサブクエリの結果テーブルの **ID** 値にない場合、**Zen** はステートメントの最終的な結果テーブルに **Person** テーブルの行を含みます。

関連項目

[SELECT \(INTO 付き\)](#)

[SELECT](#)

[UNION](#)

ALTER（名前変更）

ALTER（名前変更）ステートメントにより、インデックス、ユーザー定義関数、ストアド プロシージャ、テーブル、トリガー、またはビューの名前を変更できます。

構文

ALTER オブジェクトの種類 **RENAME** 修飾されたオブジェクト名 **TO** 新しいオブジェクト名

オブジェクトの種類 ::= INDEX

FUNCTION
PROCEDURE
TABLE
TRIGGER
VIEW

修飾されたオブジェクト名 ::= データベース名 . テーブル名 . オブジェクト名

データベース名 . オブジェクト名
テーブル名 . オブジェクト名
オブジェクト名

データベース名 , テーブル名 , オブジェクト名 , 新しいオブジェクト名 ::= ユーザー定義名

備考

以下のオブジェクトは、PSQL v9 より前のバージョンの Zen で作成されている場合には、名前を変更できません。

- ストアド プロシージャ
- トリガー
- ビュー

以前のリリースでは、これらのオブジェクトの名前によるシステム テーブルのインデックスは変更不可として作成されました。これらのオブジェクトのインデックスが変更可能になったのは PSQL v9 からです。

どの**オブジェクトの種類**も**データベース名**を使って修飾できます。ただし、INDEX または TRIGGER オブジェクトを修飾する場合は、**テーブル名**も含める必要があります。**テーブル名**は、INDEX および TRIGGER オブジェクトを修飾する場合にのみ使用できます。

ALTER ステートメントで、データベース内のオブジェクトの名前を変更することができます。変更するオブジェクトが、現在セッションが**接続されていない**データベース内にある場合は、**データベース名**を使って**オブジェクトの種類**を修飾する必要があります。

す。名前変更されたオブジェクトは、同一データベース内に**データベース名**として現れます。

修飾子の**データベース名**を省略すると、現在セッションが接続されているデータベースを基にオブジェクトが識別され、名前が変更されます。

新しいオブジェクト名の修飾子にデータベース名を使用してはいけません。注意してください。新しい名前のコンテキストは、元の名前のコンテキストと常に一致させます。

メモ： データベース エンジン は、名前変更されたオブジェクトの依存関係を**チェックしません**。以前の名前で依存関係が設定されているすべてのオブジェクトを必要に応じて修正してください。たとえば、あるトリガーが `t1` という名前のテーブルを参照しているとします。テーブル `t1` の名前を `t5` に変更すると、トリガーには無効な SQL が含まれていることになり、トリガーは失敗します。

オブジェクトの名前は、`psp_rename` システム ストアド プロシージャを使用して変更することもできます。

例

次のステートメントは、現在セッションが接続されているデータベースにあるインデックス `suplid` の名前を `vendor_id` へ変更します。インデックスが適用されているテーブルは `region5` です。

```
ALTER INDEX RENAME region5.suplid TO vendor_id
```

次のステートメントは、データベース `foodforlife` にあるユーザー定義関数 `calbrned` の名前を `caloriesburned` へ変更します。

```
ALTER FUNCTION RENAME foodforlife.calbrned TO caloriesburned
```

次のステートメントは、データベース `international` にあるストアド プロシージャ `checkstatus` の名前を `isEligible` へ変更します。

```
ALTER PROCEDURE RENAME international.checkstatus TO isEligible
```

次のステートメントは、現在セッションが接続されているデータベースにあるテーブル `payouts` の名前を `accts_payable` へ変更します。

```
ALTER TABLE RENAME payouts TO accts_payable
```

次のステートメントは、データベース `electronics` のテーブル `domestic` にあるトリガー `testtrig3` の名前を `new_customer` へ変更します。

```
ALTER TRIGGER RENAME electronics.domestic.testtrig3 TO new_customer
```

次のステートメントは、現在セッションが接続されているデータベースにあるビュー `suplrcds` の名前を `vendor_codes` へ変更します。

```
ALTER VIEW RENAME suplrcds TO vendor_codes
```

関連項目

[CREATE FUNCTION](#)

[CREATE PROCEDURE](#)

[CREATE TABLE](#)

[CREATE TRIGGER](#)

[CREATE VIEW](#)

[psp_rename](#)

ALTER GROUP

ALTER GROUP ステートメントは、ユーザーをグループに追加またはグループから削除します。

構文

```
ALTER GROUP グループ名  
  < ADD USER ユーザー名 [ , ユーザー名 ]...  
  | DROP USER ユーザー名 [ , ユーザー名 ]... >
```

備考

Master ユーザーのみがこのステートメントを実行できます。

このステートメントは、使用可能な 2 つのキーワードのいずれかと共に使用する必要があります。

ユーザー アカウントをグループに追加するには、そのグループが既にデータベース内に作成済みである必要があります。ユーザーを作成して同時にグループに追加する方法は、[GRANT](#) を参照してください。

グループからユーザー アカウントを削除しても、そのグループはデータベースから削除されません。

ユーザー アカウントは、同時に複数のグループに所属することはできません。現在あるグループのメンバーであるユーザー アカウントを別のグループに追加することはできません。そのようなユーザー アカウントの場合、まず現在のグループから削除し、次に別のグループに追加します。

空白や非英数文字を含むユーザー名は二重引用符で囲む必要があります。

ユーザーとグループの詳細については、『*Advanced Operations Guide*』の [Master ユーザー、ユーザーとグループ](#)、および『*Zen User's Guide*』の [権限の割り当て作業](#) を参照してください。

例

以下の例は、ユーザー アカウントをグループに追加する方法を示します。

```
ALTER GROUP developers ADD USER pgranger
```

既存のユーザー アカウント *pgranger* が、既存のグループ *developers* に追加されます。

```
ALTER GROUP developers ADD USER "polly granger"
```

ユーザー アカウント *polly granger*（非英数文字を含む）が、グループ *developers* に追加されます。

```
ALTER GROUP developers ADD USER "polly granger", bflat
```

ユーザー アカウント *polly granger*（非英数文字を含む）および *bflat* は、グループ *developers* に追加されます。

以下の例は、グループからユーザー アカウントを削除する方法を示します。

```
ALTER GROUP developers DROP USER pgranger
```

ユーザー アカウント *pgranger* が、グループ *developers* から削除されます。

```
ALTER GROUP developers DROP USER "polly granger"
```

ユーザー アカウント *polly granger*（非英数文字を含まない名前）が、グループ *developers* から削除されます。

```
ALTER GROUP developers DROP USER "polly granger", bflat
```

ユーザー アカウント *polly granger*（非英数文字を含む）および *bflat* は、グループ *developers* から削除されます。

関連項目

[ALTER USER](#)

[CREATE GROUP](#)

[CREATE USER](#)

[DROP GROUP](#)

[GRANT](#)

REVOKE

SET SECURITY

ALTER TABLE

ALTER TABLE ステートメントにより、テーブル定義を変更します。ALTER TABLE を使用して列を変更した場合、既存の列定義に追加されるのではなく、新しい定義に置き換えられることに留意してください。

構文

ALTER TABLE テーブル名 [オプション [オプション...]] [IN DICTIONARY]
[USING ' パス名 '] [WITH REPLACE] 変更オプション

テーブル名 ::= ユーザー定義名

オプション ::= DCOMPRESS | PCOMPRESS | PAGESIZE = サイズ | LINKDUP = 個数 | SYSDATA_KEY_2

パス名 ::= ファイル名のみ、または相対パスとファイル名

変更オプション ::= 変更オプション - リスト 1 | 変更オプション - リスト 2

変更オプション - リスト 1 ::= 変更オプション | (変更オプション [, 変更オプション]...)

変更オプション ::= ADD [COLUMN] 列定義

| ADD テーブル制約定義
| ALTER [COLUMN] 列定義
| DROP [COLUMN] 列名
| DROP CONSTRAINT 制約名
| DROP PRIMARY KEY
| MODIFY [COLUMN] 列定義

変更オプション - リスト 2 ::= PSQL_MOVE [COLUMN] 列名 TO [[PSQL_PHYSICAL] PSQL_POSITION]
新しい列位置 | RENAME COLUMN 列名 TO 新しい列名

列定義 ::= 列名 データ型 [DEFAULT デフォルト値の式] [列制約定義 [列制約定義]... [CASE (文字列) |
COLLATE 照合順序名]] [MASKED WITH (FUNCTION = マスク関数)]

列名 ::= ユーザー定義名

新しい列位置 ::= 新しい位置を表す序数値 (正の整数値)。値は、0 より大きく、テーブル内の列の総数以下でなければなりません。

新しい列名 ::= ユーザー定義名

データ型 ::= データ型名 [(桁数 [, 小数位])]

マスク関数 ::= DEFAULT()

桁数 ::= 整数

小数位 ::= 整数

デフォルト値の式 ::= デフォルト値の式 + デフォルト値の式

| デフォルト値の式 - デフォルト値の式
| デフォルト値の式 * デフォルト値の式
| デフォルト値の式 / デフォルト値の式
| デフォルト値の式 & デフォルト値の式
| デフォルト値の式 | デフォルト値の式

```

| デフォルト値の式 ^ デフォルト値の式
| ( デフォルト値の式 )
| - デフォルト値の式
| + デフォルト値の式
| ~ デフォルト値の式
| ?
| リテラル
| スカラー関数
| { fn スカラー関数 }
| USER
| NULL

```

デフォルトのリテラル ::= ' 文字列 ' | N' 文字列 '

```

| 数字
| { d ' 日付リテラル ' }
| { t ' 時刻リテラル ' }
| { ts ' タイムスタンプ リテラル ' }

```

デフォルトのスカラー関数 ::= **USER()**

```

| NULL()
| NOW()
| CURDATE()
| CURTIME()
| CURRENT_DATE()
| CURRENT_TIME()
| CURRENT_TIMESTAMP()
| CONVERT()

```

列制約定義 ::= [**CONSTRAINT** 制約名] 列制約

制約名 ::= ユーザー定義名

```

列制約 ::= NOT NULL
| NOT MODIFIABLE
| UNIQUE
| PRIMARY KEY
| REFERENCES テーブル名 [ ( 列名 ) ] [ 参照アクション ]

```

参照アクション ::= 参照更新アクション [参照削除アクション]
 | 参照削除アクション [参照更新アクション]

参照更新アクション ::= **ON UPDATE RESTRICT**

参照削除アクション ::= **ON DELETE CASCADE**
 | **ON DELETE RESTRICT**

照合順序名 ::= ' 文字列 '

テーブル制約定義 ::= [**CONSTRAINT** 制約名] テーブル制約

```

テーブル制約 ::= UNIQUE ( 列名 [ , 列名 ]... )
| PRIMARY KEY ( 列名 [ , 列名 ]... )
| FOREIGN KEY ( 列名 [ , 列名 ] )
| REFERENCES テーブル名
| ( 列名 [ , 列名 ]... )
| 参照アクション

```

備考

主キーおよび外部キーと参照整合性に関する情報については、[CREATE TABLE](#) を参照してください。

CHAR、VARCHAR、または LONGVARCHAR と、NCHAR、NVARCHAR、または NLONGVARCHAR とにおける変換では、CHAR 値はデータベース コード ページを使用してエンコードされていることを前提とします。LONGVARCHAR 型の列を NLONGVARCHAR 型に変更することも、NLONGVARCHAR 型を LONGVARCHAR 型に変更することもできません。

ALTER TABLE では、テーブルが排他ロックされている必要があります。別のステートメントにより同じテーブルが開いていると、ALTER TABLE は失敗してステータス コード 88 を返します。データ操作ステートメントを実行する前に、すべてのデータ定義ステートメントを実行しておいてください。これを示す例は、[PSQL_MOVE](#) を参照してください。

SYSDATA_KEY_2 キーワードを指定した ALTER TABLE ステートメントでは、ファイルが 13.0 より前の形式である場合には自動的に 13.0 に変更されます。その後、システム データ v2 が追加され、仮想列 sys\$create および sys\$update をクエリで 사용할 ことができます。詳細については、[システム データ v2 のアクセス](#) を参照してください。

SYSDATA_KEY_2 キーワードと一緒に IN DICTIONARY を使用した場合、ALTER TABLE ステートメントは SYSDATA_KEY_2 を無視するため、テーブルの sys\$create および sys\$update 仮想列は使用できません。

IN DICTIONARY

このキーワードを使用する目的は、基となる物理データは変更しないままで DDF に変更を加えたいことを、データベース エンジンに通知することです。IN DICTIONARY は上級ユーザー向けの強力な機能です。これは、絶対に必要な場合にだけ、システム管理者のみが使用してください。通常、Zen は DDF とデータ ファイルの完全な同期を保ちますが、この機能により、ユーザーは柔軟にテーブルの辞書定義を既存のデータ ファイルに合致させることが可能になります。このキーワードは、既存のデータ ファイルと合致する定義を辞書内に作成したい場合、あるいは USING 句を使ってテーブルのデータ ファイルパス名を変更したい場合に有用です。

IN DICTIONARY をバウンド データベースで使うことはできません。

IN DICTIONARY は ALTER TABLE に加え、CREATE TABLE および DROP TABLE で使用できます。IN DICTIONARY は、どの CREATE/ALTER オプションが指定されていようと、辞書エントリにのみ影響します。Zen では複数のオプション（ADD、DROP、ADD CONSTRAINT などのあらゆる組み合わせ）が許可されることから、IN DICTIONARY は、DDF だけがスキーマの変更によって影響を受けることを保証するために、すべての状況下で優先されます。

デタッチされたテーブルまたは存在しないテーブルを照会すると、"テーブルが見つかりません" というエラーになります。テーブルの存在を確認したにもかかわらず "テーブルが見つかりません" エラーを受け取った場合、このエラーはデータ ファイルを開けなかったことが原因で発生しています。これはデタッチされたテーブルを示します。（DDF のみが存在する（データ ファイルは存在しない）テーブルは、「デタッチされた」エントリと呼ばれます。これらのテーブルには、クエリを介して、また、クエリ以外で基となる物理ファイルを開こうとする操作を介してアクセスできません。

テーブルが実際に存在するかどうかは、カタログ関数（[システム カタログ関数](#)を参照）を使用するか、X\$File の Xf\$Name 列に直接クエリを実行することにより確認できます。

```
SELECT * FROM X$File WHERE Xf$Name = ' テーブル名 '
```

SELECT ステートメントは Xf\$Loc 値を返します。これは、テーブルの物理ファイルの名前です。この名前とデータベースに定義されているデータ パスを組み合わせれば、ファイルの絶対パスを取得できます。

デタッチされたテーブルの場合は混乱を招く可能性があるため、IN DICTIONARY 機能は非常に注意して使用しなければなりません。テーブル定義を物理ファイルと合致させるために使用するものであり、テーブル定義のデタッチに使用してはならないということがきわめて重要です。次の例で、test123.btr ファイルは存在しないと仮定して考えてみましょう（USING は次のサブトピックで説明されています）。

```
CREATE TABLE t1 USING 't1.btr' (c1 INT)
ALTER TABLE t1 IN DICTIONARY USING 'test123.btr'
```

または、両方のステートメントを組み合わせると次のようになります。

```
CREATE TABLE t1 IN DICTIONARY USING 'test123.btr' (c1 INT)
```

その後 SELECT from t1 を実行すると、テーブルが見つからなかったというエラーが返されます。テーブルを作成しただけであって、どのようにしてもそれを見つけることができないので、混乱が生じることがあります。また、IN DICTIONARY を指定しないでテーブルを DROP しようとする、同様のエラーが返されます。これらのエラーは、テーブルに関連付けられたデータ ファイルが存在しないことが原因で発生します。

既存の **Btrieve** データ ファイルに対してリレーショナル インデックス定義を作成する（たとえば、**ALTER TABLE** ステートメントを発行して **IDENTITY** 型の列定義を追加する）たびに、**Zen** はそのファイルに定義されている **Btrieve** インデックスを自動的にチェックし、既存の **Btrieve** インデックスがリレーショナル インデックス定義に必要なパラメーターのセットを提供しているかどうかを判定します。既存の **Btrieve** インデックスと作成する新しい定義が一致する場合は、リレーショナル インデックス定義と既存の **Btrieve** インデックスの間に関連付けが作成されます。一致するインデックスがない場合は、**Zen** は新しいインデックス定義を作成し、**IN DICTIONARY** が指定されていなければ、データ ファイルに新しいインデックスを作成します。

USING

USING キーワードを使用すると、特定のデータ ファイルを **CREATE TABLE** または **ALTER TABLE** アクションと関連付けることができます。

Zen は接続に名前付きデータベースを必要とするので、指定するパス名は常に単純なファイル名であるか、または相対パスとファイル名でなければなりません。パスは常に、接続する名前付きデータベースに指定された最初のデータ パスとの相対になります。

渡されたパス名およびファイル名は、ステートメントの準備ができたときに部分的に検証されます。

パス名を指定するときは、次の規則に従う必要があります。

- テキストは、文法定義で示されているように、一重引用符で囲まなければなりません。
- テキストの長さは、**X\$File** の **Xf\$Loc** に収まるよう、メタデータ バージョン 1 の場合は 1 から 64 文字まで、メタデータ バージョン 2 の場合は 1 から 250 文字まででなければなりません。エントリは入力したまま正確に格納されます（ただし、後続の空白は切り捨てられ、無視されます）。
- パスは、単純な相対パスでなければなりません。サーバーまたはボリュームを参照するパスは許可されません。
- 相対パスには、1 つのピリオド（"." は現在のディレクトリ）、2 つのピリオド（ ".." は親ディレクトリ）、円記号（"¥"）、またはこれら 3 つのあらゆる組み合わせを含めることができます。パスは、**SQL** テーブル名を表すファイル名を含んでいる必要があります（パス名は円記号 "¥" またはディレクトリ名で終わってははいけません）。

CREATE TABEL または **ALTER TABLE** を使用してファイルを作成する場合、ファイ

ル名は、相対パス付きで指定されたファイル名も含めてすべて、名前付きデータベースの設定で定義されている最初のデータパスとの相対にします。(IN DICTIONARY を使用する場合、ファイル名には最初のデータの場所への相対パスを付けません。)

- ルートベースの相対パスを使用できます。たとえば、最初のデータパスを D:¥mydata¥demodata とした場合、Zen は次のステートメント内のパス名を D:¥temp¥test123.btr と解釈します。

```
CREATE TABLE t1 USING '¥temp¥test123.btr' (c1 int)
```

- 相対パス内の円記号 (¥) 文字は、好みに応じて、Linux スタイル (/) と通常使われる円記号 (¥) のどちらを指定してもかまいません。必要であれば、2種類の記号を混在させて使用することもできます。ディレクトリ構造スキーマは知っているかもしれませんが、接続されているサーバーの種類を知っている（あるいは管理している）とは限らないので、これは便利な機能です。パスは入力したとおりに X\$File に格納されます。Zen エンジンでは、パスを利用してファイルを開く際、円記号文字を適切なプラットフォームのタイプに変換します。また、データファイルはサポートされるすべてのプラットフォーム間でバイナリ互換性を共有するため、ディレクトリ構造がプラットフォーム間で同一である（および、パスに基づくファイル名が相対パスで指定されている）限りは、データベースファイルおよび DDF をこれらに変更を加えることなく、あるプラットフォームから別のプラットフォームへ移動することができます。これは、複数のプラットフォームにまたがって、標準化されたデータベーススキーマをより簡単に配置するのに役立ちます。
- 相対パスを指定する場合、まず USING 句のディレクトリ構造は存在している必要があります。Zen は、USING 句で指定されたパス条件を満たすディレクトリを作成しません。

USING 句を使用して、既存のテーブルと関連付ける既存データファイルの物理的場所と名前を指定します。USING 句ではまた、既存の辞書定義を使用して、特定の場所に新しいデータファイルを作成することができます (USING 句に指定した文字列は、X\$File 辞書ファイルの Xf\$Loc 列に格納されます)。新しいファイルを作成するとき、ファイル情報のいくつかを元のデータファイルから取得する必要があるため、元のデータファイルは使用可能でなければなりません。

Demodata サンプルデータベースでは、Person テーブルは PERSON.MKD ファイルと関連付けられています。PERSON2.MKD という名前の新しいファイルを作成すると、次の例のステートメントは、Person テーブルが新しいファイルと関連付けられるように、Person テーブルの辞書定義を変更します。


```
ALTER TABLE Person IN DICTIONARY USING 'person2.mkd'
```

USING 句には単純なファイル名か相対パスのいずれかを使用しなければなりません。相対パスを指定した場合、Zen はそのパスを、データベース名と関連付けられている最初のデータ ファイル パスとの相対と解釈します。

USING 句は、ALTER TABLE のほかの標準オプションに加えて指定することができます。これは、USING パスを指定するステートメントと同じステートメント内で列を操作できるということです。

テーブル データを格納するために現在使用しているデータ ファイル名と異なるデータ ファイル名を指定し、IN DICTIONARY を指定しないと、Zen は新しいファイルを作成し、既存ファイルのデータをすべて新規ファイルにコピーします。たとえば、Person テーブルのデータを保持している現在のデータ ファイルは person.mkd であると仮定します。上記のステートメントで示されるように、person2.mkd ファイルを使用するよう Person テーブルを変更します。person.mkd の内容が person2.mkd にコピーされます。そうすると、person2.mkd が Person テーブルと関連付けられたデータ ファイルとなり、データベース操作は person2.mkd に作用します。person.mkd は削除されませんが、データベースにはもう関連付けられなくなります。

データをコピーするのは、Zen では、USING と同時にそれ以外のすべての ALTER TABLE オプションを指定できるためです。作成された新しいデータ ファイルには、既存テーブルのデータが完全に読み込まれる必要があります。ファイル構造は単純にコピーされるのではなく、内容全体が移行されます。これは、Btrieve の BUTIL -CREATE と BUTIL -COPY を実行するのに似ています。このことは、SQL テーブルを再構築したり、以前は多数のレコードを格納していたが現在は少数のレコードしか格納していないファイルを圧縮したりするのに役立ちます。

メモ： ALTER TABLE USING は、既存のデータ ファイルの内容を新しく指定されたデータ ファイルにコピーし、古いデータ ファイルは元のまま、ただしリンクは解除して残しておきます。

WITH REPLACE

WITH REPLACE が USING と共に指定されたときはいつでも、Zen は既存のファイル名を指定されたファイル名で自動的に上書きします。オペレーティング システムがファイルの上書きを許している限り、ファイルは常に上書きされます。

WITH REPLACE はデータ ファイルにのみ作用し、DDF には作用しません。

WITH REPLACE を使用する際には次の規則が適用されます。

- WITH REPLACE は USING と併せてのみ使用できます。
- IN DICTIONARY と一緒に使用すると、WITH REPLACE は無視されます。IN DICTIONARY は DDF にのみ作用することを指定するものだからです。

メモ： ALTER TABLE で WITH REPLACE を使用しても、データは消失したり破棄されたりしません。新しく作成されたデータ ファイルは、既存ファイルを上書きしたファイルであっても、以前のファイルのデータをすべて含んでいます。ALTER TABLE コマンドを使用することで、データを失うことはありません。

Zen に既存ファイル（ファイルは、USING 句で指定された場所になければなりません）を置き換えるよう指示するには、USING 句で WITH REPLACE を使用します。WITH REPLACE を使用すると、Zen は新しいファイルを作成し、既存ファイルのデータをすべて新規ファイルにコピーします。WITH REPLACE を使用していないとき、指定された場所にファイルが存在すると、Zen はステータス コードを返し、新しいファイルを作成しません。ステータス コードはエラー -4940 です。

MODIFY COLUMN と ALTER COLUMN

列のデータ型やヌル値を許可するかどうかを変更する機能は、次の制約を受けます。

- ターゲット列では、PRIMARY/FOREIGN KEY 制約を定義することはできません。
- 旧データ型を新しいデータ型に変換することによって（演算またはサイズの）オーバーフローが発生する場合、ALTER TABLE 操作は中止されます。
- ヌル値を許可する列がヌル値を含む場合、その列をヌル値を許可しない列に変更することはできません。

主キー列または外部キー列のデータ型を変更する必要がある場合は、制約を削除し、列のデータ型を変更してから再び制約を追加することにより実現できます。関連するすべてのキー列の同期がとれているようにしなければならない、ということに留意してください。たとえば、テーブル T1 に主キーがあり、このキーがテーブル T2 および T3 の外部キーによって参照される場合は、まず外部キーを削除しなければなりません。外部キーを削除した後で、主キーを削除することができます。次に、3 つの列をすべて同じデータ型に変更する必要があります。最後に、主キーを再び追加してから、外部キーを追加します。

ANSI 標準には ALTER キーワードが含まれています。Zen では、ALTER TABLE ステートメントで MODIFY キーワードも使用できます。COLUMN キーワードは省略可能です。たとえば、次のようになります。

```
ALTER TABLE t1 MODIFY c1 INTEGER
ALTER TABLE t1 ALTER c1 INTEGER
ALTER TABLE t1 MODIFY COLUMN c1 INTEGER
ALTER TABLE t1 ALTER COLUMN c1 INTEGER
```

Zen では、実際のデータが、列の長さを小さくした新しい列でオーバーフローしないのであれば、列の長さを小さくすることができます。この動作は、Microsoft SQL Server の動作に似ています。

1 つの ALTER TABLE ステートメントで、複数の列を追加、削除、または変更することができます。これは操作を簡略化しますが、この動作は ANSI 互換とみなされません。次に、複数列の ALTER ステートメントの例を示します。

```
ALTER TABLE t1 (ALTER c2 INT, ADD D1 CHAR(20), DROP C4, ALTER C5 LONGVARCHAR, ADD D2 LONGVARCHAR NOT NULL)
```

旧データ型 (Pervasive.SQL v7 以前) を、現在の Zen リリースでサポートされているネイティブなデータ型に変換できます。逆に、新データ型を旧データ型へ変換したい場合は、Zen サポートにお問い合わせください。

NOTE/LVAR 列を持つ古いテーブルに LONGVARCHAR/LONGVARBINARY 列を追加するには、まず、NOTE/LVAR 列を LONGVARCHAR または LONGVARBINARY 列に変換しなければなりません。NOTE/LVAR 列を LONGVARCHAR/LONGVARBINARY に変換したら、テーブルにそれ以外の LONGVARCHAR/LONGVARBINARY 列を追加できます。古いエンジンでは、テーブル当たり 1 つの可変長列しか扱うことができないため、前述の新しいテーブルを使って作業することはできないので注意してください。

PSQL_MOVE

PSQL_MOVE 構文を使用すると、テーブルの列を希望する位置に保持することができます。既存の列や、追加された新しい列の位置を変更したい場合があります。列を論理的および物理的に移動させることができます。

移動の種類	結果
論理的	結果セットにリストされる列の位置は変わりますが、テーブル内の列の物理的順序は変わりません。たとえば、"SELECT * FROM テーブル名" のようなクエリで生成される結果セットにおける列の配置方法を変更できます。論理的移動は、"SELECT * FROM テーブル名" のような、列を列挙するクエリにのみ影響を及ぼします。
物理	列は、ファイル内の現在位置から新しい位置へ物理的に移動されます。物理的移動は、テーブルのデータ ファイルに影響を及ぼします。列を物理的に移動する場合は、PSQL_PHYSICAL キーワードを指定する必要があります。 PSQL_PHYSICAL キーワードを省略すると、デフォルトで論理的移動が生じます。 ALTER TABLE ステートメントで IN DICTIONARY が使用されている場合は、DDF 内の列のオフセットのみが変更されるので注意してください。データ ファイルに対し、MOVE ... PSQL_PHYSICAL より IN DICTIONARY が優先されるため、データ ファイル内の列は物理的に 移動されません 。

メモ： 一度列を論理的に移動したら、その並び順が結果セットにおける列のデフォルトの列挙順になります。たとえば、列を論理的に移動させた後で物理的に移動させた場合、"SELECT * FROM テーブル名" のようなクエリでは、論理順が使用されます。論理的な列の変更は X\$Attrib に格納されます。

PSQL_MOVE キーワードには、ゼロより大きく、列の総数よりも小さい値で列の位置を指定する必要があります。たとえば、テーブル t1 には col1 と col2 の 2 つの列だけがあるとします。次のステートメントはどちらもエラーを返します。

```
ALTER TABLE t1 PSQL_MOVE col1 to 0
ALTER TABLE t1 PSQL_MOVE col1 to 3
```

最初のステートメントは列を位置 0 へ移動しようとしています。2 番目のステートメントは列を位置 3 へ移動しようとしています。これは列の総数である 2 よりも大きい数値です。

ALTER TABLE では、テーブルが排他ロックされている必要があります。別のステートメントにより同じテーブルが開いていると、ALTER TABLE は失敗してステータス コー

ド 88 を返します。データ操作ステートメントを実行する前に、すべてのデータ定義ステートメントを実行しておいてください。

たとえば、次のストアド プロシージャでは、INSERT ステートメントがテーブル t1 を開いていることにより、ALTER TABLE ステートメントが排他ロックを取得できないため、実行が失敗し、ステータス コード 88 が返されます。

```
CREATE PROCEDURE proc1() AS
BEGIN
CREATE TABLE t1(c1 INT,c2 INT,c3 INT);
INSERT INTO t1 VALUES (123,345,678);
ALTER TABLE t1 PSQL_MOVE c3 to 1;
END;
```

これを解決する方法は、最初にテーブル作成とデータ挿入に関連するステートメントを実行してから、プロシージャを呼び出すことです。

```
CREATE TABLE t1(c1 INT,c2 INT,c3 INT);
INSERT INTO t1 VALUES (123,345,678);
CALL proc1;

CREATE PROCEDURE proc1() AS
BEGIN
ALTER TABLE t1 PSQL_MOVE c3 to 1;
END;
```

RENAME COLUMN

RENAME COLUMN を使用すると、列の名前を別の名前に変更できます。ただし、同じテーブル内の既存の列名に変更することはできません。

列名を変更することにより、以前の名前を参照しているオブジェクトが無効になる場合があります。たとえば、あるトリガーがテーブル t1 の列 c1 を参照しているとします。列名を c1 から c5 に変更すると、トリガーは正常に実行できなくなります。

列の名前を変更する場合、`psp_rename` システム ストアド プロシージャを使用することもできます。

メモ： データベース エンジン は、名前変更された列の依存関係をチェックしません。列の名前を変更したら、必ず、以前（変更元）の名前で依存関係が設定されているすべてのオブジェクトを適切に修正してください。

ON DELETE CASCADE

CREATE TABLE の削除規則を参照してください。

例

このセクションでは、ALTER TABLE のいくつかの例を示します。

次のステートメントによって、Emergency_Phone 列が Person テーブルに追加されます。

```
ALTER TABLE person ADD Emergency_Phone NUMERIC(10,0)
```

次のステートメントによって、col1 と col2 の 2 つの整数列が Class テーブルに追加されます。

```
ALTER TABLE class(ADD col1 INT, ADD col2 INT)
```

テーブル定義から列を削除するには、DROP 句内に列の名前を指定します。次のステートメントによって、Emergency_Phone 列が Person テーブルから削除されます。

```
ALTER TABLE person DROP Emergency_Phone
```

次のステートメントによって、Class テーブルから col1 と col2 が削除されます。

```
ALTER TABLE class(DROP col1, DROP col2)
```

次のステートメントによって、Faculty テーブル内の制約 c1 が削除されます。

```
ALTER TABLE Faculty(DROP CONSTRAINT c1)
```

この例では、Class テーブルに整数列 col3 が追加され、列 col2 が削除されます。

```
ALTER TABLE class(ADD col3 INT, DROP col2 )
```

次の例では、Faculty テーブル内の ID フィールドに c1 という名前の主キーが作成されます。ヌル値を許可する列には主キーを作成できないことに注意してください。作成しようとすると、エラーが返されます。

```
ALTER TABLE Faculty(ADD CONSTRAINT c1 PRIMARY KEY(ID))
```

次の例では、デフォルトのキー名である PK_ID を使用して、Faculty テーブルに主キーが作成されます。

```
ALTER TABLE Faculty(ADD PRIMARY KEY(ID))
```

次の例では、制約 UNIQUE が列 col1 と col2 に追加されます。すべての列の col1 と col2 の値の組み合わせが、テーブル内で一意になります。どちらの列も個々に一意である必要はありません。

```
ALTER TABLE Class(ADD UNIQUE(col1,col2))
```

次の例では、Faculty テーブル内の主キーが削除されます。テーブルに複数の主キーがあってはならないので、既に主キーが定義されているテーブルに主キーを追加することはできません。テーブルの主キーを変更するには、既存のキーを削除してから新しい主キーを追加します。

```
ALTER TABLE Faculty(ADD PRIMARY KEY)
```

親テーブルから主キーを削除するには、その前に、従属テーブルから対応する外部キーをすべて削除する必要があります。

次の例は、Class テーブルに新規の外部キーを追加します。Faculty_ID 列はヌル値を含まない列として定義されます。ヌル値を許可する列には外部キーを作成できません。

```
ALTER TABLE Class ADD CONSTRAINT Teacher FOREIGN KEY (Faculty_ID) REFERENCES Faculty (ID) ON DELETE RESTRICT
```

この例では、削除制限規則によって、まず教職員の全講座を変更または削除してからでないと、その教職員をデータベースから削除できないようになっています。また、REFERENCES 句にリストされている列 (ID) は任意であることに注目してください。ステートメントをより明確にするために、好みによって REFERENCES 句に列のリストを含めることができます。ただし、REFERENCES 句で参照できる列は参照テーブルの主キーのみです。

次のステートメントでは、上記の例で追加した外部キーを削除する方法を示します。Zen は従属テーブルから外部キーを削除し、従属テーブルと親テーブル間の参照制約を取り除きます。

```
ALTER TABLE Class DROP CONSTRAINT Teacher
```

次の例は、CONSTRAINT 句を使用しないで Class テーブルに外部キーを追加します。この場合、外部キーの制約は内部的に生成され、Faculty の主キー (ID) を参照するように定義されます。REFERENCES 句にリストされている列は任意です。ステートメントをより明確にするために、好みによって REFERENCES 句に列のリストを含めることができます。ただし、REFERENCES 句で使用できる列は参照テーブルの主キーのみです。

```
ALTER TABLE Class ADD FOREIGN KEY (Faculty_ID) REFERENCES Faculty (ID) ON DELETE RESTRICT
```

これにより、外部キー `FK_Faculty_ID` が作成されます。外部キーを削除するには、`CONSTRAINT` キーワードを指定します。

```
ALTER TABLE Class DROP CONSTRAINT FK_Faculty_ID
```

次の例では、テーブル内の制約と列を追加および削除する方法を示します。このステートメントにより、`Faculty` テーブル内の列 `salary` が削除され、整数型の列 `col1` が追加され、制約 `c1` が削除されます。

```
ALTER TABLE Faculty(DROP salary, ADD col1 INT, DROP CONSTRAINT c1)
```

次の例はどちらも、複数の列のデータ型を変更する方法を示しています。

```
ALTER TABLE t1 (ALTER c2 INT, ADD D1 CHAR(20), DROP C4, ALTER C5 LONGVARCHAR, ADD D2 LONGVARCHAR NOT NULL)
ALTER TABLE t2 (ALTER c1 CHAR(50), DROP CONSTRAINT MY_KEY, DROP PRIMARY KEY, ADD MYCOLUMN INT)
```

次の例では、列オプションの `ALTER` と `MODIFY` を使って、列のデフォルト値およびオルタネート コレーティング シーケンスを設定したり削除する方法が示されています。

```
CREATE TABLE t1 (c1 INT DEFAULT 10, c2 CHAR(10))
ALTER TABLE t1 ALTER c1 INT DEFAULT 20
```

ー列 `c1` のデフォルト値を 20 に再設定する

```
ALTER TABLE t1 ALTER c1 INT
```

ー列 `c1` のデフォルト値を削除する

```
ALTER TABLE t1 ALTER c2 CHAR(10)
COLLATE 'file_path¥upper.alt'
```

ー列 `c2` のオルタネート コレーティング シーケンスを設定する

```
ALTER TABLE t1 ALTER c2 CHAR(10)
```

ー列 `c2` のオルタネート コレーティング シーケンスを削除する

`upper.alt` は、ソートする際に大文字と小文字を同等に扱います。たとえば、データベースに `abc`、`ABC`、`DEF`、`Def` という値がこの順序で挿入されている場合、`upper.alt` を使ってソートすると、`abc`、`ABC`、`DEF`、`Def` のように返されます（値 `abc` と `ABC`、`DEF` と `Def` は同じものと判断され、これらは挿入された順序で返されます）。標準の ASCII ソートでは、大文字は小文字の前に配列されており、ソート結果は `ABC`、`DEF`、`Def`、`abc` のようになります。

次のステートメントは、列 `Registrar_ID` が結果セットにリストされときの位置を、現在位置から 2 番目へ論理的に移動させます。

```
ALTER TABLE Billing PSQL_MOVE Registrar_ID TO 2
```

次のステートメントは、列 `Amount_Owed` と `Amount_Paid` が結果セットにリストされときの位置を、それぞれ現在位置から 2 番目と 3 番目へ移動させます。

```
ALTER TABLE Billing ( PSQL_MOVE Amount_Owed TO 2, PSQL_MOVE Amount_Paid TO 3 )
```

次のステートメントは、列 `Registrar_ID` のデータ ファイル内の位置を、現在位置から 2 列目へ物理的に移動させます。これにより、変更を反映させるためにデータ ファイルのリビルドが生じます。

```
ALTER TABLE Billing PSQL_MOVE Registrar_ID TO PSQL_PHYSICAL 2
```

次のステートメントは、列 `Amount_Owed` と `Amount_Paid` のデータ ファイル内の位置を、それぞれ現在位置から 2 列目と 3 列目へ移動させます。

```
ALTER TABLE Billing ( PSQL_MOVE Amount_Owed TO PSQL_PHYSICAL 2, PSQL_MOVE Amount_Paid TO PSQL_PHYSICAL 3 )
```

テーブル `t1` には列 `c1` と `col2` があるとします。次のステートメントによって、列 `c1` の名前が `c2` に変更されます。

```
ALTER TABLE t1 RENAME COLUMN c1 TO c2
```

テーブル `t1` には列 `c1` と `col2` があるとします。次のステートメントは、列の名前 (`col2`) を既存の列の名前 (`c1`) に変更しようとするため、エラー (列名が重複しています) が返されます。

```
ALTER TABLE t1 (RENAME COLUMN c1 TO c2, RENAME COLUMN col2 TO c1)
```

代わりに、2 つの `ALTER` ステートメントを発行する必要があります。1 つ目は `c1` を `c2` に名前変更します。2 つ目は `col2` を `c1` に名前変更します。

```
ALTER TABLE t1 (RENAME COLUMN c1 TO c2)
```

```
ALTER TABLE t1 (RENAME COLUMN col2 TO c1)
```


関連項目

[CREATE TABLE](#)

[DROP TABLE](#)

[CREATE INDEX](#)

[DEFAULT](#)

[SET DEFAULTCOLLATE](#)

ALTER USER

ALTER USER ステートメントは、ユーザー アカウントの名前またはパスワードを変更します。

構文

```
ALTER USER ユーザー名 < RENAME TO 新しいユーザー名 | WITH PASSWORD ユーザー パスワード >
```

備考

Master ユーザーのみがユーザーの名前を変更できます。ほかのユーザーが自身のパスワードを変更するには、WITH PASSWORD 句を記述するか、SET PASSWORD を使用します。SET PASSWORD を参照してください。

このステートメントを実行するには、セキュリティ設定が有効になっている必要があります。

このステートメントは、RENAME TO オプションまたは WITH PASSWORD キーワードと共に使用する必要があります。

新しいユーザー名はデータベース内で固有の名前である必要があります。

ユーザー名とユーザー パスワードは、空白やその他の非英数文字が含まれている場合には、二重引用符で囲む必要があります。作成されるユーザーの詳細については、[ユーザーおよびグループへの権限の付与](#)を参照してください。

メモ： パスワードの制限については、『*Advanced Operations Guide*』の[識別子の制限](#)および[データベース セキュリティ](#)トピックを参照してください。ユーザーとグループの詳細については、『*Advanced Operations Guide*』の[Master ユーザー](#)、[ユーザーとグループ](#)、および『*Zen User's Guide*』の[権限の割り当て作業](#)を参照してください。

例

次の例では、ユーザー アカウントの名前の変更方法を示します。

```
ALTER USER pgranger RENAME TO grangerp
```

アカウントの名前 *pgranger* が *grangerp* に変更されます。

```
ALTER USER pgranger RENAME TO "polly granger"
```

アカウントの名前 *pgranger* は、非英数文字を含む *polly granger* に変更されます。

次の例では、ユーザー アカウントのパスワードの変更方法を示します。

```
ALTER USER pgranger WITH PASSWORD Prvsve1
```

ユーザー アカウント *pgranger* のパスワードは *Prvsve1*（大文字と小文字を区別）に変更されます。

```
ALTER USER pgranger WITH PASSWORD "Nonalfa$"
```

ユーザー アカウント *pgranger* のパスワードは、非英数文字を含む *Nonalfa\$*（大文字と小文字を区別）に変更されます。

関連項目

[ALTER（名前変更）](#)

[CREATE GROUP](#)

[CREATE USER](#)

[DROP USER](#)

[GRANT](#)

[SET PASSWORD](#)

ANY

備考

ANY キーワードは ALL キーワードと似た働きを持ちますが、サブクエリの結果テーブル内のどの行に対しても条件を満たしていれば、Zen はその比較した行を結果テーブルに含みます。

例

次のステートメントは、Person テーブルの ID 列と、サブクエリの結果テーブルの ID 列を比較します。Person テーブルの ID 値と一致する値がサブクエリの結果テーブルの ID 値にある場合、Zen では、Person テーブルの行が SELECT ステートメントの結果テーブルに含まれます。

```
SELECT p.ID, p.Last_Name
FROM Person p
WHERE p.ID = ANY
(SELECT f.ID FROM Faculty f WHERE f.Dept_Name = 'Chemistry')
```

関連項目

[SELECT](#)

AS

備考

AS 句を使用して、選択項目またはテーブルに名前を割り当てます。この名前は、ステートメントのほかの場所で、選択項目を参照するときに使用できます。名前はよくエイリアスと呼ばれます。

AS 句を非集計列に使用すると、WHERE、ORDER BY、GROUP BY、HAVING の各句で名前を参照できます。AS 句を集計列に使用すると、ORDER BY 句のみで名前を参照できます。

定義する名前は、SELECT リスト内で重複しないようにする必要があります。

列のエイリアスは列名として返されます。グループ集計を含む、列エイリアスのない計算列には、EXPR-1、EXPR-2 などのシステムによって生成された名前が割り当てられます。

例

次のステートメントの AS 句は、選択項目 SUM (Amount_Paid) に名前 Total を割り当て、結果を学生ごとの合計でソートするように Zen に指示します。

```
SELECT Student_ID, SUM (Amount_Paid) AS Total
FROM Billing
GROUP BY Student_ID
ORDER BY Total
```

AS キーワードは、次の例のようにテーブルエイリアスに使用する場合は省略可能です。FROM 句内で AS 句をテーブル名に使用すると、WHERE、ORDER BY、GROUP BY、HAVING の各句で名前を参照できます。

```
SELECT DISTINCT c.Name, p.First_Name, c.Faculty_Id
FROM Person AS p, class AS c
WHERE p.Id = c.Faculty_Id
ORDER BY c.Faculty_Id
```

このクエリは次のように、FROM 句で AS 句を使用しないよう書き直すことができます。

```
SELECT DISTINCT c.Name, p.First_Name, c.Faculty_Id
FROM Person p, class c
WHERE p.Id = c.Faculty_Id
ORDER BY c.Faculty_Id
```

テーブルエイリアスを確立したら、**WHERE** 句内にテーブル名とエイリアスが混在してはいけません。次は動作しません。

```
SELECT DISTINCT c.Name, p.First_Name, c.Faculty_Id
FROM Person p, class c
WHERE Person.Id = c.Faculty_Id
ORDER BY c.Faculty_Id
```

関連項目

[SELECT](#)

BEGIN [ATOMIC]

備考

BEGIN および END キーワードは、ストアド プロシージャ、ユーザー定義関数、またはトリガー宣言の本体を定義するために使用されます。キーワードにより、プロシージャ、関数、トリガー内に複合ステートメントが作成されます。

ATOMIC キーワードを追加すると、ステートメントのブロックが 1 つのトランザクションであるかのように、トランザクション動作を制御することができます。ATOMIC は、ブロック内のすべてのステートメントは成功するかロールバックするか、いずれかが必要であることを示します。

例

次の例では、BEGIN ... END ブロックが ATOMIC として設定されています。両方の INSERT がエラーなしで実行された場合にのみ、レコードが挿入されます。どちらかのステートメントがエラーを返す（この場合は 2 番目の INSERT がステータス 5 を返す）場合、レコードは挿入されません。

```
CREATE PROCEDURE Add_Tuition();
BEGIN ATOMIC
    INSERT INTO Tuition(ID, Degree, Residency, Cost_Per_Credit, Comments) VALUES (9, 'Test', 0,
    100.0, 'Training');
    INSERT INTO Tuition(ID, Degree, Residency, Cost_Per_Credit, Comments) VALUES (8, 'Test', 0,
    100.0, 'Training');
END
```

関連項目

[END](#)

[CREATE PROCEDURE](#)

[CREATE TRIGGER](#)

CALL

備考

ストアド プロシージャを呼び出す場合は、**CALL** ステートメントを使用します。ストアド プロシージャは、ユーザー定義のものでもシステム ストアド プロシージャでもかまいません。

例

次の例では、パラメーターのないユーザー定義のプロシージャを呼び出します。

```
CALL NoParms() または CALL NoParms
```

次の例では、パラメーターのあるユーザー定義のプロシージャを呼び出します。

```
CALL Parms(vParm1, vParm2)
CALL CheckMax(N.Class_ID)
```

次のステートメントは、システム ストアド プロシージャを呼び出すことにより、Dept テーブル内のすべての列に関する列情報をリストします。

```
CALL psp_columns('Demodata', 'Dept')
```

関連項目

[CREATE PROCEDURE](#)

[CREATE TRIGGER](#)

[EXECUTE](#)

[システム ストアド プロシージャ](#)

CASCADE

備考

外部キーの作成時に CASCADE を指定した場合、Zen では DELETE CASCADE 規則が使用されます。ユーザーが親テーブルの行を削除すると、Zen によって従属テーブルの対応する行が削除されます。

DELETE CASCADE は注意して使用してください。Zen では、自己参照するテーブルに対し、循環するカスケード削除を使用できます。『*Advanced Operations Guide*』の[削除カスケード](#)に記載されている例を参照してください。

関連項目

[ALTER TABLE](#)

[CREATE TABLE](#)

CASE（式）

CASE 式は値を返します。CASE 式には 2 つの形式があります。

- 単一 When/Then。この形式は、1 つの値式を一連の値式と比較して結果を判定します。値式は、それらが列挙されている順に評価されます。値式が TRUE と評価された場合、CASE は THEN 句の値式を返します。
- 検索 When/Then。この形式は、一連のブール式を評価して結果を判定します。ブール式は、それらが列挙されている順に評価されます。ブール式が TRUE と評価された場合、CASE は THEN 句の式を返します。

どちらの形式もオプションの ELSE 引数をサポートしています。ELSE 句を使用しない場合は、ELSE NULL であることを意味します。

構文

単一 When/Then

```
CASE case 値式
  WHEN when 式 THEN then 式 [...]
  [ ELSE else 式 ]
END
```

検索 When/Then

```
CASE
  WHEN 検索式 THEN then 式 [...]
  [ ELSE else 式 ]
END
```

引数

case 値式 ::= 単一 When/Then 形式の CASE で評価される式。

when 式 ::= *case 値式* と比較される式。*case 値式* と各 *when 式* のデータ型は同じであるか、または暗黙に型変換されなければなりません。

then 式 ::= *case 値式* = *when 式* が TRUE と評価された場合に返される式。

else 式 ::= TRUE と評価される比較演算がない場合に返される式。この引数を省略した場合、TRUE と評価される比較演算がなければ、CASE は NULL を返します。

検索式 ::= 検索形式の CASE で評価されるブール式。検索式には、有効なブール式であればどのような式でも指定できます。

備考

CASE 式は SELECT ステートメント内で使用する必要があります。SELECT ステートメントはストアド プロシージャやビュー内に記述できます。

例

次のステートメントは、単一 When/Then 形式を使って、Course テーブルにリストされている美術講座の必須条件をレポートします。

```
SELECT name 'Course ID', description 'Course Title',
CASE name
WHEN 'Art 101' THEN 'None'
WHEN 'Art 102' THEN 'Art 101 or instructor approval'
WHEN 'Art 203' THEN 'Art 102'
WHEN 'Art 204' THEN 'Art 203'
WHEN 'Art 305' THEN 'Art 101'
WHEN 'Art 406' THEN 'None'
WHEN 'Art 407' THEN 'Art 305'
END
AS 'Prerequisites' FROM Course WHERE Dept_Name = 'Art' ORDER BY name
```

このクエリによって次の一覧が返されます。

Course ID	Course Title	Prerequisites
Art 101	Drawing I	None
Art 102	Drawing II	Art 101 or instructor approval
Art 203	Drawing III	Art 102
Art 204	Drawing IV	Art 203
Art 305	Sculpture	Art 101
Art 406	Modern Art	None
Art 407	Baroque Art	Art 305

前のステートメントに ELSE 句を含めるように変更できます。

```
SELECT name 'Course ID', description 'Course Title',
CASE name
WHEN 'Art 101' THEN 'None'
WHEN 'Art 102' THEN 'Art 101 or instructor approval'
WHEN 'Art 203' THEN 'Art 102'
WHEN 'Art 204' THEN 'Art 203'
WHEN 'Art 305' THEN 'Art 101'
ELSE 'Curriculum plan for Art History majors'
END
AS 'Prerequisites' FROM Course WHERE Dept_Name = 'Art' ORDER BY name
```

このクエリによって今度は次の一覧が返されます。

Course ID	Course Title	Prerequisites
Art 101	Drawing I	None
Art 102	Drawing II	Art 101 or instructor approval
Art 203	Drawing III	Art 102
Art 204	Drawing IV	Art 203
Art 305	Sculpture	Art 101
Art 406	Modern Art	Curriculum plan for Art History majors
Art 407	Baroque Art	Curriculum plan for Art History majors

次のステートメントでは検索 When/Then 形式を使って、個人別に資格のある奨学金プログラムをレポートします。

```
SELECT last_name, first_name,
CASE
WHEN scholarship = 1 THEN 'Scholastic'
WHEN citizenship <> 'United States' THEN 'Foreign Study'
WHEN (date_of_birth >= '1960-01-01' AND date_of_birth <= '1970-01-01') THEN 'AJ-44 Funds'
ELSE 'NONE'
END
AS 'Funding Program' FROM Person ORDER BY last_name
```

以下に、このクエリによって返される一覧の一部を示します。

Last_Name	First_Name	Funding Program
Abad	Alicia	NONE
Abaecherli	David	Foreign Study
Abebe	Marta	Foreign Study
Abel	James	AJ-44 Funds
Abgoon	Bahram	Foreign Study
Abken	Richard	NONE
Abu	Austin	Foreign Study
Abuali	Ibrahim	AJ-44 Funds
Acabbo	Joseph	NONE
Acar	Dennis	Foreign Study

次の例は、ストアド プロシージャ内で CASE 式を使用する方法を示します。

```
CREATE PROCEDURE pcasetest() RETURNS (d1 CHAR(10), d2 CHAR(10));
BEGIN
```

```
SELECT c1, CASE WHEN c1 = 1 THEN c4
WHEN c1 = 2 THEN c5
ELSE
CASE WHEN c2 = 100.22 THEN c4
WHEN c2 = 101.22 THEN c5 END END
FROM tcasetest;
END

CALL pcasetest
```

次の例は、ビュー内で CASE 式を使用する方法を示します。

```
CREATE VIEW vcasetest (vc1, vc2) AS
SELECT c1, CASE WHEN c1 = 1 THEN c4
WHEN c1 = 2 THEN c5
ELSE
CASE WHEN c2 = 100.22 THEN c4
WHEN c2 = 101.22 THEN c5 END END
FROM TCASEWHEN

SELECT * FROM vcasetest
```

関連項目

[COALESCE](#)、[SELECT](#)

CASE（文字列）

備考

CASE キーワードは、文字列型の列を対象とする制限句を評価するとき、Zen に大文字小文字を無視させます。CASE は、CREATE TABLE または ALTER TABLE ステートメントや、SELECT ステートメントの ORDER BY 句で列属性として指定できます。

たとえば、Name という名前の列があり、CASE 属性付きで定義されているとします。Name = 'Smith' と Name = 'SMITH' を使って 2 行挿入した場合、制限に Name = 'smith' と指定したクエリでは、両方の列が正しく返されます。

メモ： CASE（文字列）は、マルチバイトの文字列および NCHAR 文字列を**サポートしません**。このキーワードは、文字列データがシングルバイトの ASCII であることを前提としています。これはつまり、CASE 属性は NVARCHAR および NCHAR データ型の列ではサポートされないということです。文字列関数では、マルチバイト文字列および NCHAR 文字列がサポートされます。[文字列関数](#)を参照してください。

例

次の例は、CASE キーワードを使って Student テーブルに列を追加する方法を示します。

```
ALTER TABLE Student ADD Name char(64) CASE
```

次の例は、SELECT ステートメントの ORDER BY 句で CASE を使用方法を示します。

```
SELECT Id, Last_Name+', '+First_Name AS Whole_Name, Phone FROM Person ORDER BY Whole_Name CASE
```

関連項目

[ALTER TABLE](#)

[CREATE TABLE](#)

[SELECT](#)

CLOSE

構文

CLOSE カーソル名

カーソル名 ::= ユーザー定義名

備考

CLOSE ステートメントにより、開いている SQL カーソルを閉じます。

カーソル名が示すカーソルは開いている必要があります。

このステートメントは、ストアド プロシージャ、ユーザー定義関数、およびトリガーの内部でのみ使用できます。カーソルおよび変数は、ストアド プロシージャ、ユーザー定義関数、およびトリガーの内部でのみ使用できます。

例

次の例では、カーソル BTUCursor が閉じます。

```
CLOSE BTUCursor;
```

```
CREATE PROCEDURE MyProc(OUT :CourseName CHAR(7)) AS
BEGIN
  DECLARE cursor1 CURSOR
  FOR SELECT Degree, Residency, Cost_Per_Credit
  FROM Tuition ORDER BY ID;
  OPEN cursor1;
  FETCH NEXT FROM cursor1 INTO :CourseName;
  CLOSE cursor1;
END
```

関連項目

[OPEN](#)

[CREATE PROCEDURE](#)

[CREATE TRIGGER](#)

COALESCE

COALESCE スカラー関数は 2 つ以上の引数を取り、その式リストを左から調べて最初の非ヌル引数を返します。

構文

```
COALESCE ( 式, 式 [, ... ] )
```

式 ::= 任意の有効な式

戻り値の型

COALESCE 関数は、式リスト内のいずれかの式の値を返します。戻り値のデータ型の詳細な一覧については、[COALESCE でサポートされるデータ型の組み合わせと結果のデータ型](#)を参照してください。

制限

この関数は少なくとも 2 つの引数を取ります。

```
COALESCE(10, 20)
```

無効：

```
COALESCE()
```

メモ： 無効な例では、次の解析時エラーが発生します：
"COALESCE には少なくとも 2 つの引数が必要になります。"

式リストには、少なくとも 1 つはヌルでない引数が含まれている必要があります。

```
COALESCE (NULL, NULL, 20)
```

無効：

```
COALESCE (NULL, NULL, NULL)
```

メモ： 無効な例では、次の解析時エラーが発生します：
"COALESCE のすべての引数を NULL 関数にすることはできません。"

この関数は、式リストに指定するデータ型の組み合わせによってはサポートしていないものがあります。たとえば、**BINARY** や **VARCHAR** など、暗黙的に相互に変換できない引数を COALESCE に指定することはできません。

COALESCE でサポートされるデータ型の組み合わせと結果のデータ型

次の図は、サポートされているさまざまなデータ型の組み合わせの詳細を示すと同時に、それらを COALESCE 関数で用いた場合に結果として生じるデータ型の確認に役立つ

ちます。

オペランド 2																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																	</

図表の要素	説明
↑	これらの型は COALESCE 関数で直接使用できます。結果はオペランド 2 の型になります。
←	これらの型は COALESCE 関数で直接使用できます。結果はオペランド 1 の型になります。
空のセル	これらの型には互換性がありません。COALESCE で直接使用することはできません。明示的な CONVERT が必要です。

図表の要素	説明
D	結果は SQL_DOUBLE 型になります。
B	結果は SIM_BCD 型になります。
I	結果は SQL_INTEGER 型になります。
S	結果は SQL_SMALLINT 型になります。

COALESCE 関数でサポートされていない型の組み合わせ（図で空のセルになっています）を使用すると、解析時エラーが発生します。

行のエラーです。

割り当てエラーです。

式の評価エラーです。

例

次の例で、`10+2` は `SMALLINT` として扱われ、`ResultType (SMALLINT, SMALLINT)` は `SMALLINT` になります。したがって、結果の型は `SMALLINT` になります。

```
SELECT COALESCE(NULL,10 + 2,15,NULL)
```

最初のパラメーターは `NULL` です。2 番目の式は `12` と評価されます。この値は `NULL` でなく、結果の型の `SMALLINT` へ変換できます。そのため、この例の戻り値は `12` になります。

次の例で、`10` は `SMALLINT` として扱われ、`ResultType (SMALLINT, VARCHAR)` は `SMALLINT` になります。したがって、結果の型は `SMALLINT` になります。

```
SELECT COALESCE(10, 'abc' + 'def')
```

最初のパラメーターは `10` で、この値は結果の型の `SMALLINT` へ変換できます。そのため、この例の戻り値は `10` になります。

COMMIT

COMMIT ステートメントにより、論理トランザクションの終了を示し、テンポラリデータをパーマネント データに変換します。

構文

COMMIT []

例

次の例では、ストアド プロシージャの中で、Billing テーブル内の Amount_Owed 列を更新するトランザクションが開始されます。この作業がコミットされ、別のトランザクションによって Amount_Paid 列が更新されてゼロに設定されます。最後の COMMIT WORK ステートメントにより、2 番目のトランザクションが終了します。

```
START TRANSACTION;
UPDATE Billing B
SET Amount_Owed = Amount_Owed - Amount_Paid
WHERE Student_ID IN
(SELECT DISTINCT E.Student_ID
FROM Enrolls E, Billing B
WHERE E.Student_ID = B.Student_ID);
COMMIT WORK;
START TRANSACTION;
UPDATE Billing B
SET Amount_Paid = 0
WHERE Student_ID IN
(SELECT DISTINCT E.Student_ID
FROM Enrolls E, Billing B
WHERE E.Student_ID = B.Student_ID);
COMMIT WORK;
```

```
CREATE PROCEDURE UpdateBilling( ) AS
BEGIN
START TRANSACTION;
UPDATE Billing SET Amount_Owed = Amount_Owed + Amount_Owed;
UPDATE Billing set Amount_Owed = Amount_Owed + 100 WHERE Student_ID = 10;
COMMIT;
END;
```

関連項目

[CREATE PROCEDURE](#)

[ROLLBACK](#)

[START TRANSACTION](#)

CREATE DATABASE

CREATE DATABASE ステートメントにより、新しいデータベースを作成します。データベースにログインしているユーザーは誰でも、このステートメントを発行できます。ユーザーはまた、指定の場所でファイルを作成するために、オペレーティング システムからの権限を持っている必要があります。

構文

```
CREATE DATABASE [ IF NOT EXISTS ] データベース名 DICTIONARY_PATH ' 辞書パス名 ' [ DATA_PATH ' データパス名 ' [ , ' データパス名 ' ]... ] [ NO_REFERENTIAL_INTEGRITY ] [ BOUND ] [ REUSE_DDF ] [ DBSEC_AUTHENTICATION ] [ DBSEC_AUTHORIZATION ] [ V1_METADATA | V2_METADATA ] [ ENCODING < ' コード ページ名 ' | 'CP コード ページ番号' | DEFAULT > ]
```

データベース名 ::= データベースのユーザー定義名

辞書パス名 ::= データ辞書ファイル (DDF) の場所のユーザー定義名

データパス名 ::= データ ファイルの場所のユーザー定義名

コード ページ名 ::= 有効なコード ページの名前

CP コード ページ番号 ::= "CP" で始まる有効なコード ページの番号

備考

ODBC を使用している場合は、CREATE DATABASE はデータベースのみを作成し、関連するデータ ソース名 (DSN) は作成しないということを覚えておいてください。

DSN が必要な場合は、個別に作成する必要があります。『Zen User's Guide』の [ODBC データベース アクセスの設定](#)を参照してください。

CREATE DATABASE ステートメントは、サーバー上で**最初の**データベースを作成する場合には使用できません。ユーザーは、CREATE DATABASE ステートメントを発行する前にデータベースにログオンしていなければならないからです。そのため、事前に少なくとも 1 つはデータベースが存在している必要があります。

CREATE DATABASE ステートメントは、ストアド プロシージャおよびユーザー定義関数内では使用できません。

データベース名と IF NOT EXISTS 句

データベース名には、新しいデータベースの名前を指定します。データベース名はサーバー内で一意であり、識別子の規則に従っている必要があります。『*Advanced Operations Guide*』の[識別子の制限](#)を参照してください。

データベースが存在した場合、IF NOT EXISTS 句を省略していると、エラーが発生します（ステータス コード 2303）。IF NOT EXISTS 句を記述している場合は、エラーは返されません。

辞書パス

辞書パス名は、辞書ファイル（DDF）が存在する物理的な保管場所を指定します。この場所には、[CREATE TABLE](#) ステートメントを使用した場合、あるいは Zen Control Center（ZenCC）を使ってテーブルを作成した場合に、データ ファイルも保存されます。『*Zen User's Guide*』の[辞書のロケーション](#)を参照してください。

データ パス

データ パス名には、データベースのデータ ファイルが存在する可能性のある場所を指定します（下記の「メモ」を参照してください）。複数のパス名をカンマで区切って指定することができます。

データ パス名は、呼び出し元アプリケーションの分析視点からではなく、データベース エンジンの視点から有効であれば、どのようなパスでもかまいません。指定した場所は既に存在している必要があります。[CREATE DATABASE](#) ステートメントでディレクトリは作成されません。

データ ファイルに対し辞書ファイルと同じ場所を使用する場合は、**データ パス名**を省略します。**データ パス名**に空文字列を渡しても、同じ場所を指定することができます。たとえば、`DATA_PATH ''` という指定は、データ パスが空文字列であることを示します。

メモ： [CREATE TABLE](#) ステートメントまたは ZenCC を使用してテーブルを作成した場合、**データ ファイル**は指定した最初の**辞書パス名**に保存されます。**辞書パス名**を指定していない場合、データ ファイルは**辞書パス名**の場所に作成されます。

データ パス名は、Distributed Tuning Interface（DTI）を介してテーブルを作成している

場合に有用です。DTI 関数の `PvAddTable` では、データ ファイルを配置したい場所を指定できます。『*Distributed Tuning Interface Guide*』の [PvAddTable\(\)](#) を参照してください。

参照整合性

デフォルトで、データベース エンジン は参照整合性を強要します。

`NO_REFERENTIAL_INTEGRITY` 句を指定すると、データベース内に定義されているトリガーおよび参照整合性は強要されなくなります。

[参照整合性の設定](#) および [Btrieve およびリレーショナル制約間の相互作用](#) を参照してください。

BOUND

`BOUND` を指定すると、DDF がデータベースにバインドされます。バウンド データベースは、データベース名を単一の DDF のセットと関連付けます。この DDF は 1 つのデータ ファイルのセットしか参照しません。DDF は、既存の DDF か、`CREATE DATABASE` ステートメントの実行によって作成される DDF のいずれかがバインドされます。

DDF がバインドされると、複数のデータベースでその DDF を使用することも、複数の DDF のセットでデータ ファイルを参照することもできなくなります。

`BOUND` を指定しなければ、DDF はデータベースにバインドされません。

『*Advanced Operations Guide*』の [バウンド データベースと整合性の設定](#) を参照してください。

辞書ファイル

`REUSE_DDF` キーワードは、既存の DDF をデータベースと関連付けます。既存の DDF は **辞書パス名** の場所になければなりません。

`REUSE_DDF` を省略すると、**辞書パス名** の場所に DDF が既に存在していない限り、新しい DDF が作成されます。DDF が **辞書パス名** の場所に存在すれば、新しい DDF が作成される代わりに、既存の DDF がデータベースと関連付けられます。

セキュリティ

データベース エンジン は、MicroKernel エンジンの 3 つのセキュリティ モデルをサポートしています。

- クラシック。コンピュータへのログインに成功したユーザーは、データ ファイルに対してユーザーに割り当てられているファイル システム権限のレベルに従って、データベース コンテンツへのアクセス権を持ちます。ファイル システム権限はオペレーティング システムによって割り当てられます。
- データベース。データベースのユーザー アカウントはオペレーティング システムのユーザー アカウントとは関係ありません。データに対するユーザー アクセス権は、データベースに設定されているユーザーのアクセス許可によって決定されます。
- 混合。このポリシーは、ほか 2 つのポリシーの側面を持っています。ユーザーはオペレーティング システムのユーザー名とパスワードを使ってログインしますが、データに対するユーザー アクセス権はデータベースに設定されているユーザーのアクセス許可によって決定されます。

セキュリティの完全な説明については、『*Advanced Operations Guide*』の [Zen セキュリティ](#) を参照してください。

DBSEC_AUTHENTICATION および DBSEC_AUTHORIZATION キーワードにより、データベースのセキュリティ ポリシーを設定します。

ステートメントにキーワードを含んでいるかいないか		セキュリティ モデル		
DBSEC_AUTHENTICATION	DBSEC_AUTHORIZATION	クラシック	データベース	混合
省略	省略	X		
含む	含む	X		
省略	含む	X		

メタデータのバージョン

リレーショナル エンジンでは、メタデータでバージョン 1 (V1) とバージョン 2 (V2) という 2 つのバージョンをサポートします。メタデータのバージョンは、そのデータベース内のすべてのデータ辞書ファイル (DDF) に適用されます。デフォルトは "メタデータ バージョン 1" です。

ほかにも特性はありますが、メタデータバージョン 2 では、多くの識別子名に最大 128 バイトの名前を付けることができ、ビューおよびストアド プロシージャに対する権限が認められています。詳しい説明については、[Zen メタデータ](#)を参照してください。

メタデータバージョン 1 を指定する場合、V1_METADATA キーワードは含めても省略してもかまいません。メタデータバージョン 2 を指定する場合は、V2_METADATA キーワードを含める必要があります。

エンコード

エンコードは文字セットを表す標準規格です。文字データは、コンピューターがデジタル処理できる標準形式に変換する、つまりエンコードする必要があります。エンコードは、Zen サーバー エンジンと Zen クライアント アプリケーションとの間で規定する必要があります。互換性のあるエンコードを使用すれば、サーバーとクライアントでデータが正しく変換されます。

エンコードのサポートは、データベース コード ページとクライアント エンコードに分割されました。この 2 種類のエンコードは、別個のものですが相互に関係しています。詳細については、『*Advanced Operations Guide*』の[データベース コード ページとクライアント エンコード](#)を参照してください。

データベース コード ページおよびクライアント エンコードは、リレーショナル エンジンのみに適用されます。MicroKernel エンジンには影響がありません。

コード ページは、名前または文字 CP の後にコード ページ番号を続けて指定します。いずれも、一重引用符で囲む必要があります。たとえば、有効な名前は UTF-8 であり、有効な番号は CP1251 です。

Windows および Linux オペレーティング システムは共に、OS エンコードと呼ばれるデフォルトのエンコードがあります。デフォルトの OS エンコードはオペレーティング システム間で異なります。キーワード DEFAULT を使用すると、サーバーの OS エンコードが指定されます。

ENCODING キーワードが指定されていない場合、データベースはサーバーの OS エンコードをデフォルトとします。

無効なコード ページ番号やコード ページ名を指定すると、"コード ページの値が無効です" というエラーが返されます。

複数のデータベースを使用する SQL ステートメントでは、すべてのデータベースで同じコード ページを使用していることを確認する必要があります。そうしないと、文字データが正しく返されません。

メモ： データベース エンジン は、アプリケーション がデータベースに追加するデータ およびメタデータのエンコードを **検証しません**。エンジンは、すべてのデータが、『*Advanced Operations Guide*』の **データベース コード ページとクライアント エンコード** で説明されているようにサーバーまたはクライアントのエンコードを使用して入力されるものと想定しています。

複数のデータベースを使用する SQL ステートメント（複数データベースの結合など）では、すべてのデータベースでデータベース コード ページが同じであることを確認してください。そうしないと、文字データが正しく返されません。

有効なコード ページ名とコード ページ番号

サポートされるコード ページ名とコード ページ番号は ZenCC で一覧表示することができます。ZenCC を起動して [データベースの新規作成] ダイアログを開きます（『*Zen User's Guide*』の **新規データベースを作成するには** を参照してください）。[データベース コード ページ] オプションで、[**コード ページの変更**] をクリックします。開いたダイアログで [**データベース コード ページ**] をクリックすると、使用できるコード ページの一覧が表示されます。

Linux では、**dbmaint** ユーティリティの man ページを参照すると、使用できるコード ページ名とコード ページ番号の一覧が表示されます。この例については、『*Zen User's Guide*』の **dbmaint** を参照してください。

例

このセクションでは、CREATE DATABASE の例を示します。

次の例は、inventorydb という名前のデータベースを作成し、DDF の場所をドライブ D: の mydbfiles¥ddf_location フォルダと指定しています。D:¥mydbfiles¥ddf_location には DDF が存在しないため、新しい DDF が作成されます。データ ファイルも DDF と同じ場所に保存されます。データベースはメタデータ バージョン 1 を使用します。

```
CREATE DATABASE inventorydb DICTIONARY_PATH 'D:¥mydbfiles¥ddf_location'
```

次の例は、HRUSBenefits という名前のデータベースがまだ存在していなければ作成し、DDF の場所をドライブ C: の HRDatabases¥US フォルダーと指定しています。また、データ ファイルが存在する可能性のある場所として、C: ドライブの HRDatabases¥US¥DataFiles ディレクトリと、E: ドライブの Backups¥HRUSData ディレクトリを指定しています（[データ パス](#)の「メモ」を参照してください）。DDF が DICTIONARY_PATH に存在する場合は、既存の DDF が使用されます。データベースはメタデータ バージョン 1 を使用します。

```
CREATE DATABASE IF NOT EXISTS HRUSBenefits DICTIONARY_PATH 'C:¥HRDatabases¥US' DATA_PATH 'C:¥HRDatabases¥US¥DataFiles' ; E:¥Backups¥HRUSData' REUSE_DDF
```

次の例は、EastEurope という名前のデータベースを作成します。DDF の場所をドライブ C: の Europe¥DbaseFiles フォルダーとし、新しい DDF を作成して、それをデータベースにバインドします。また、セキュリティ ポリシーを「混合」に設定し、メタデータ バージョン 2 を使用します。

```
CREATE DATABASE EastEurope DICTIONARY_PATH 'C:¥Europe¥DbaseFiles' BOUND DBSEC_AUTHORIZATION V2_METADATA
```

次の例は、Region5Acct という名前のデータベースを作成します。DDF の場所をドライブ D: の Canada¥Region5¥Accounting フォルダーとし、データベース コード ページをサーバーで使用されているデフォルトのコード ページに設定します。

```
CREATE DATABASE Region5Acct DICTIONARY_PATH 'D:¥Canada¥Region5¥Accounting' ENCODING DEFAULT
```

次の例は、Region2Inventory という名前のデータベースを作成し、DDF の場所をドライブ G: の Japan¥Region2 フォルダーと指定し、データベース コード ページを 932 に設定します。

```
CREATE DATABASE Region2Inventory DICTIONARY_PATH 'G:¥Japan¥Region2' ENCODING 'CP932'
```

次の例は、VendorCodes という名前のデータベースを作成します。DDF の場所をドライブ C: の Capitol_Equipment¥Milling フォルダーとし、新しい DDF を作成して、それらをデータベースにバインドします。また、セキュリティ ポリシーを「混合」に設定してメタデータ バージョン 2 を使用し、データベース コード ページを 1252 に設定します。

```
CREATE DATABASE VendorCodes DICTIONARY_PATH 'C:¥Capitol_Equipment¥Milling' BOUND DBSEC_AUTHORIZATION
V2_METADATA ENCODING 'CP1252'
```

関連項目

[DROP DATABASE](#)

CREATE FUNCTION

CREATE FUNCTION ステートメントにより、データベースにスカラー ユーザー定義関数 (UDF) を作成します。その後、クエリからユーザー定義関数を呼び出すことができます。

構文

```
CREATE FUNCTION 関数名 ( [ [ IN ]
{ : パラメーター名 スカラー パラメーターのデータ型 [ DEFAULT 値 | = 値 ] } [...] ] )
RETURNS スカラー戻り値のデータ型
[AS]
BEGIN
    関数本体
    RETURN スカラー式
END;
```

関数名 ::= スカラー ユーザー定義関数 (UDF) の名前。UDF 名は識別子の規則に従い、データベース内で一意である必要があります。

パラメーター名 ::= スカラー UDF のパラメーター。最大 300 個のパラメーターを使用できます。default が指定されていない場合は、関数を呼び出すときに値を指定する必要があります。

スカラー パラメーターのデータ型 ::= 指定したパラメーターのデータ型。

スカラー戻り値のデータ型 ::= スカラー UDF の戻り値のデータ型。スカラー型のみサポートされます。

値 ::= パラメーター名に割り当てるデフォルト値。DEFAULT キーワードまたは等記号のいずれかを使って指定します。

関数本体 ::= スカラー関数を構成するステートメント。

スカラー式 ::= スカラー UDF の戻り値。

備考

各 UDF 名 (**データベース名 . 関数名**) は、データベース内で一意である必要があります。UDF 名は、同じデータベース内の以下の名前と同じにできません。

- 組み込み関数名
- ほかの UDF 名
- ストアド プロシージャ名

制約

ユーザー定義関数内で CREATE DATABASE または DROP DATABASE ステートメントを使用することはできません。CREATE、ALTER、UPDATE、DELETE および INSERT のテーブル操作はユーザー定義関数内では許可されません。

スカラー入力パラメーターのみサポートされます。OUTPUT および INOUT パラメーターは使用できません。デフォルトで、すべてのパラメーターが入力になります。IN キーワードを指定する必要はありません。

制限

ユーザー定義関数を作成する際は以下の制限に注意してください。

属性	制限
パラメーター数	300
UDF 本体のサイズ	64 KB
UDF 名の最大長	『 <i>Advanced Operations Guide</i> 』の 識別子の制限 を参照してください。
UDF 変数名の最大長	半角 128 文字

サポートされるスカラー入力パラメーターおよび戻り値のデータ型

Zen でサポートされている、スカラー入力パラメーターと戻り値のデータ型を以下の表に示します。TEXT、NTEXT、IMAGE、CURSOR を除くすべてのデータ型を指定できます。

AUTOTIMESTAMP	BIGIDENTITY	BIGINT
BINARY	BIT	BLOB
CHAR	CHARACTER	CLOB
CURRENCY	DATE	DATETIME
DEC	DECIMAL	DOUBLE
FLOAT	IDENTITY	INT

INTEGER	LONG	LONGVARBINARY
LONGVARCHAR	NCHAR	NLONGVARCHAR
NUMERIC	NVARCHAR	REAL
SMALLIDENTITY	SMALLINT	TIME
TIMESTAMP	TIMESTAMP2	TINYINT
UBIGINT	UINT	UNINTEGER
UNIQUEIDENTIFIER	USMALLINT	UTINYINT
VARBINARY	VARCHAR	

例

このトピックでは、CREATE FUNCTION のいくつかの例を示します。

次の例では、**Box** テーブルに保存されている四角形のボックスの細目を基に、その四角形の面積を計算する関数を作成します。

```
CREATE FUNCTION CalculateBoxArea(:boxName CHAR(20))
RETURNS REAL
AS
BEGIN
    DECLARE :len REAL;
    DECLARE :breadth REAL;
    SELECT len, breadth INTO :len, :breadth FROM box
    WHERE name = :boxName;
    RETURN(:len * :breadth);
END;
```

次の例では、2つの整数を比較し、小さい方の値を返す関数を作成します。

```
CREATE FUNCTION GetSmallest(:A INTEGER, :B INTEGER)
RETURNS INTEGER
AS
BEGIN
    DECLARE :smallest INTEGER
    IF (:A < :B ) THEN
        SET :smallest = :A;
    ELSE
        SET :smallest = :B;
    END IF;
    RETURN :smallest;
END;
```

次の例では、 $SI = PTR/100$ という式を使って単利を計算する関数を作成します。式中の P は元金、T は年数、R は利子率です。

```
CREATE FUNCTION CalculateInterest(IN :principle FLOAT, IN :period REAL, IN :rate DOUBLE)
RETURNS DOUBLE
AS
BEGIN
    DECLARE :interest DOUBLE;
    SET :interest = ((:principle * :period * :rate) / 100);
    RETURN (:interest);
END;
```

スカラー ユーザー定義関数の呼び出し

スカラー式がサポートされる場所ならどこでも、関数名の後にカンマ区切りの引数のリストを指定することによって、ユーザー定義関数を呼び出すことができます。引数のリストはかっこで囲みます。

UDF は、データベース修飾子のプレフィックスを付けても付けなくても呼び出せます。データベース修飾子を前に付けない場合は、UDF は現在のデータベース コンテキストから実行されます。データベース修飾子を前に付けた場合は、指定したデータベースのコンテキストから実行されます。下記の例では、データベース修飾子のプレフィックスを使用しているものと使用していないものがあります。

制限

関数を呼び出すとき、引数にパラメーター名を指定することはできません。

関数を呼び出すとき、すべてのパラメーターの引数値は、CREATE FUNCTION ステートメントで定義されたパラメーターの引数値と同じ順序でなければなりません。

ユーザー定義関数の例

プロシージャ内での UDF

```
CREATE PROCEDURE procTestUdfInvoke() AS
BEGIN
    DECLARE :a INTEGER;
    SET :a = 99 + (222 + Demodata.GetSmallest(10, 9)) + 10;
    PRINT :a;
END;
CALL procTestUdfInvoke();
```

次の例は、データベース修飾子が省略されている点を除けば、前の例と似ています。

```
CREATE PROCEDURE procTestUdfInvoke2() AS
BEGIN
    DECLARE :a INTEGER;
    SET :a = 99 + (222 + GetSmallest(10, 9)) +10;
    PRINT :a;
```



```
END;  
CALL procTestUdfInvoke2
```

SELECT リスト内での UDF

```
SELECT GetSmallest(100,99)
```

WHERE 句での UDF

```
SELECT name FROM class WHERE id <= GetSmallest(10,20)
```

UDF 内での UDF

```
CREATE FUNCTION funcTestUdfInvoke() RETURNS INTEGER AS  
BEGIN  
    DECLARE :a INTEGER;  
    SET :a = 99 + (222 - Demodata.GetSmallest(10, 9));  
    RETURN :a;  
END;
```

INSERT ステートメント内での UDF

```
CREATE TABLE t1(col1 INTEGER, col2 INTEGER, col3 FLOAT)  
INSERT INTO t1 VALUES (GetSmallest(10,20), 20 , 2.0)  
INSERT INTO t1 (SELECT * FROM t1 WHERE col1 = getSmallest(10,20))
```

UPDATE ステートメント内での UDF

```
UPDATE t1 SET col2 = Demodata.GetSmallest(2,10) WHERE col1 = 2  
UPDATE t1 SET col1 = 3 WHERE col2 = Demodata.GetSmallest(10, 5)
```

GROUP BY ステートメント内での UDF

```
SELECT col2 FROM t1 GROUP BY getSmallest(10,2), col2
```

ORDER BY ステートメント内での UDF

```
SELECT col2 FROM t1 ORDER BY Demodata.getSmallest(10,2), col2
```

再帰 UDF

```
CREATE FUNCTION factorial(IN :n INTEGER) RETURNS double AS BEGIN  
    DECLARE :fact DOUBLE;
```

```
IF (:n <= 0) THEN
    SET :fact = 1;
ELSE
    SET :fact = (:n * Demodata.factorial(:n - 1));
END IF;
RETURN :fact;
END;
```

SELECT Demodata.factorial(20) を使用すると、20 の階乗値を取得できます。

デフォルト値を持つ UDF

```
CREATE FUNCTION testUdfDefault1(:z INTEGER DEFAULT 10) RETURNS INTEGER AS
BEGIN
    RETURN :z-1;
END;
```

SELECT Demodata.testUdfDefault1()。この関数は、パラメーターが提供されなかった場合には、指定したデフォルト値（10）を使用します。

```
CREATE FUNCTION testUdfDefault2(:a VARCHAR(20) = 'Accounting Report' ) RETURNS VARCHAR(20) as
BEGIN
    RETURN :a;
END;
```

SELECT Demodata.testUdfDefault2()。この関数は、パラメーターが提供されなかった場合には、指定したデフォルト値（Accounting Report）を使用します。

動的パラメーターを持つ UDF

```
SELECT name FROM class WHERE id <= GetSmallest(?,?)
```

式としての UDF

```
SELECT 10 + Demodata.Getsmallest(10,20) + 15
```

パラメーターとしての UDF

```
SELECT demodata.calculateinterest (10+demodata.getsmallest(3000, 2000), demodata.factorial(2),
demodata.testUdfDefault(3))
```

関連項目

[DECLARE](#)

[DROP FUNCTION](#)

CREATE GROUP

CREATE GROUP ステートメントにより、1 つまたは複数のセキュリティ グループを作成します。

構文

```
CREATE GROUP グループ名 [ , グループ名 ]...
```

グループ名 ::= ユーザー定義名

備考

Master ユーザーのみがこのステートメントを実行できます。

このステートメントを実行するには、セキュリティ設定が有効になっている必要があります。

例

次の例では、zengroup という名前のグループが作成されます。

```
CREATE GROUP zengroup
```

次の例では、リストを使用して一度に複数のグループを作成します。

```
CREATE GROUP zen_dev, zen_marketing
```

関連項目

ALTER USER

CREATE USER

DROP GROUP

GRANT

SET SECURITY

REVOKE

CREATE INDEX

CREATE INDEX ステートメントを使用して、指定されたテーブルに名前付きインデックスを作成します。

構文

```
CREATE [ UNIQUE | PARTIAL ] [ NOT MODIFIABLE ] INDEX インデックス名 [ USING インデックス番号 ] [
IN DICTIONARY ] ON テーブル名 [ インデックス定義 ]
インデックス名 ::= ユーザー定義名
インデックス番号 ::= ユーザー定義値 (0 から 118 までの整数)
テーブル名 ::= ユーザー定義名
インデックス定義 ::= ( インデックス セグメント定義 [ , インデックス セグメント定義 ]... )
インデックス セグメント定義 ::= 列名 [ ASC | DESC ]
```

備考

VARCHAR 型の列が CHAR 型の列と異なるのは、長さバイト (Btrieve lstring) またはゼロ終端バイト (Btrieve zstring) のいずれかが予約されており、そのため有効な記憶域が 1 バイト増える点です。言い換えると、CHAR (100) の列を作成すると、レコード内で 100 バイトを占めます。VARCHAR (100) は 101 バイトを占めます。NVARCHAR 型の列が NCHAR 型の列と異なるのは、ゼロ終端文字が予約されており、そのため有効な記憶域が 2 バイト増える点です。言い換えると、NCHAR (50) の列を作成すると、レコード内で 100 バイトを占めます。NVARCHAR (50) の列は 102 バイトを占めます。

Zen でインデックスを作成する場合、その処理は、ステートメントに IN DICTIONARY か USING、またはその両方が含まれているかどうかによって変わります。結果をまとめた表を次に示します。

操作	処理と結果	追加情報
CREATE INDEX	成功した場合、インデックスはデータ ファイルと X\$Index の両方に追加されます。 - データ ファイルにインデックスが定義されていない場合、作成されるインデックスはインデックス 0 になります。 - データ ファイルに 1 つ以上のインデックスが定義されている場合、作成されるインデックスは、未使用のインデックス番号のうち最小の番号になります。 どちらの場合も、同じ番号を持つ新しいインデックスが X\$Index に同様に挿入されます。	メタデータバージョン 1 用の X\$Index、またはメタデータバージョン 2 用の X\$Index を参照してください。

操作	処理と結果	追加情報
CREATE INDEX IN DICTIONARY	成功した場合、インデックスは X\$Index にのみ追加されます。データ ファイルには何も挿入されません。 データ ファイルを調べて、どのインデックス番号が使用可能であるかを判定します。 - データ ファイルにインデックスが定義されていない場合、X\$Index に挿入されるインデックスの番号は 0 になります。 - データ ファイルに 1 つ以上のインデックスが定義されている場合、データベース エンジンに、追加するインデックスと列およびインデックスの属性が一致するインデックスが X\$Index にまだ定義されていないかどうかを確認します。 一致が見つかった場合は、そのインデックス番号を使用して、インデックスが X\$Index に追加されます。 一致が見つからない場合は、<データ ファイルのうち最大のインデックス番号> + 1 がインデックス番号に使用されます。 X\$Index にあり、データ ファイルに一致するキーがないインデックスは ファントム インデックス と呼ばれ、データベース エンジンでは使用されません。	IN DICTIONARY を参照してください。
CREATE INDEX USING インデックス番号	成功した場合、指定した インデックス番号 を持つインデックスはデータ ファイルと X\$Index の両方に追加されます。 インデックス番号がデータ ファイルまたは X\$Index のいずれかで既に使用されている場合は、エラーが返されます。	USING を参照してください。
CREATE INDEX USING インデックス番号 IN DICTIONARY	成功した場合、指定した インデックス番号 を持つインデックスは X\$Index にのみ追加されます。データ ファイルには何も挿入されません。 指定した インデックス番号 は、データ ファイルには存在するが X\$Index には存在せず、列およびインデックスの属性が追加するインデックスと一致する場合、指定した インデックス番号 のインデックスが X\$Index に追加されます。それ以外はエラーが返されます。	IN DICTIONARY を参照してください。

インデックス セグメント

インデックス セグメントは、インデックス定義に指定されている 1 つの列に対応します。複数セグメントのインデックスは、複数列の組み合わせとして作成されたものです。

次の表で示すように、ファイルに定義するすべてのインデックスで利用できるセグメントの総数は、そのファイルのページ サイズによって異なります。

ページ サイズ (バイト数)	キー セグメントの最大数 (ファイル バージョン別)			
	8.x 以前	9.0	9.5	13.0、16.0
512	8	8	切り上げ ²	切り上げ ²
1,024	23	23	97	切り上げ ²
1,536	24	24	切り上げ ²	切り上げ ²
2,048	54	54	97	切り上げ ²
2,560	54	54	切り上げ ²	切り上げ ²
3,072	54	54	切り上げ ²	切り上げ ²
3,584	54	54	切り上げ ²	切り上げ ²
4,096	119	119	204 ³	183 ³
8,192	N/A ¹	119	420 ³	378 ³
16,384	N/A ¹	N/A ¹	420 ³	378 ³

¹ "N/A" は「適用外」を意味します。

² 「切り上げ」は、ページ サイズを、ファイル バージョンでサポートされる次のサイズへ切り上げることを意味します。たとえば、512 は 1024 に切り上げられ、2560 は 4096 に切り上げるということです。

³ 9.5 以降の形式のファイルでは 119 以上のセグメントを指定できますが、インデックスの数は 119 に制限されます。

ヌル値を許可する列には考慮も必要です。たとえば、ページ サイズが 4096 バイトのデータ ファイルでは、1 ファイル当たりのインデックス セグメント数は 119 に制限されます。真のヌルがサポートされるインデックス付きのヌル値を許可する列には 2 つのセグメントから成るインデックスが必要なため、1 つのテーブルではインデックス付き

のヌル値を許可する列（Btrieve ファイルでは、インデックス付きでヌル値を許可する真のヌル フィールド）は 59 個までしか持てません。ページ サイズが小さくなると、この制限も小さくなります。

ファイルバージョン 7.x 以降では、`TRUENULLCREATE` がオンに設定されている場合、真のヌルがサポートされます。それより前のファイル形式バージョンを使用しているファイル、または `TRUENULLCREATE` がオフに設定されているファイルには、真のヌルサポートがないため、この制限はありません。

UNIQUE

UNIQUE インデックス キーは、特定の行に対してインデックスで定義された列の組み合わせがファイル内で重複しないことを保証します。これは、複数セグメントのインデックスの場合には、個々の列が重複しないことを保証しませんし、また要求もしません。

メモ： 次のデータ型以外のすべてのデータ型にインデックスを設定できます。

BIT

BLOB

CLOB

LONGVARBINARY

LONGVARCHAR

NLONGVARCHAR

『*Status Codes and Messages*』のステータス コード **6008：セグメントが多すぎます。** も参照してください。

PARTIAL

ファイル形式バージョンの最大インデックス幅を超える列または列のグループにインデックスを作成するには、`CREATE INDEX` ステートメントで `PARTIAL` キーワードを使用します。

部分インデックスは、大きな列のプレフィックスを使用して作成されるか、あるいは小さな列を複数組み合わせで作成されるため、大きな列のプレフィックスを用いた検索の方がより速く実行できます。したがって、`WHERE` 句で `'WHERE column_name LIKE 'prefix%'` のような制限を使用しているクエリは、インデックスを何も使用しない場合とは対照的に、部分インデックスを使用することによって実行が速くなります。

CREATE INDEX ステートメントに **PARTIAL** キーワードを含めた場合、インデックス列の幅とオーバーヘッドが最大インデックス幅と同等かそれを超えない場合、**PARTIAL** キーワードは無視され、代わりに標準のインデックスが作成されます。

13.0 以前のファイルの最大インデックス幅は 255 バイトです。16.0 ファイルの場合は 1024 バイトです。

メモ： 幅は列の実際のサイズを指し、オーバーヘッドはヌル インジケータや文字列の長さなどを指します。

PARTIAL の制限事項

PARTIAL を使用する際には次の制限が適用されます。

- 部分インデックスは、データ型が **CHAR** または **VARCHAR** の列にしか追加できません。
- 部分インデックス列は、必ずインデックス定義内の最後のセグメントであるか、あるいはインデックス定義内の唯一のセグメントである必要があります。

部分インデックス列がインデックス内の唯一のセグメントである場合、列のサイズは最大 8,000 バイトにすることができますが、ユーザー データのインデックス セグメントのサイズは、使用しているファイル形式の最大インデックス幅と同じになります。

- エンジンは、厳密な等式を含んでいるクエリや、部分列にかかわる **ORDER BY**、**GROUP BY**、**JOIN** などの照合操作を実行している場合には、部分インデックスを使用しません。
- 部分インデックスは、**WHERE** 句の次のような形式の制限と照合する場合にのみ使用されます。

```
WHERE col = 'literal'
WHERE col LIKE 'literal%'
WHERE col = ?
WHERE col LIKE ?
```

ここで、**literal**、つまり実引数値にはどのような長さの値も指定できます。部分インデックス列でインデックス処理されたバイト数より短くても長くてもかまいません。**LIKE** 句の形式が **'prefix%'** でない場合、部分インデックスは使用されません。

WHERE 句が前述の制限事項に適合する場合は、実行プランを立てる際に部分インデックスが使用されます。

メモ： **ALTER TABLE** を使用して部分インデックス列の長さを変更したとき、変更後の長さがインデックスの最大幅に収まるようになった場合、あるいは変更後の長さがイ

インデックスの最大幅を超える場合、ユーザーはインデックスを削除し、必要に応じて再作成する必要があります。

例

次の CREATE PARTIAL INDEX の例は 9.5 ファイル形式に基づいています。

次の例では、データ型とサイズの指定された PartID、PartName、SerialNo、および Description 列を含む、Part_tbl という名前のテーブルが作成されます。

```
CREATE TABLE part_tbl (partid INT, partname CHAR(50), serialno VARCHAR(200), description CHAR(300));
```

次に、Description 列を使って、idx_01 という名前の部分インデックスを作成します。

```
CREATE PARTIAL INDEX idx_01 on part_tbl (description);
```

インデックスで使用される Description 列は 300 バイトありますが、PARTIAL キーワードを使用することにより、先頭の 255 バイト（オーバーヘッドを含む）だけをプレフィックスとしてインデックスで使用できるようになります。

次の例では、前の例と同じテーブルに idx_02 という部分インデックスを作成します。代わりに、この例では PartID、SerialNo、および Description 列をまとめてインデックスに使用します。

```
CREATE PARTIAL INDEX idx_02 on part_tbl (partid, serialno, description);
```

次の表は、どのようにして大きい列がインデックスに割り当てられるかを理解できるよう、インデックス列の詳細を示しています。

列名	データ型	サイズ	オーバーヘッド	インデックス内のサイズ
PartID	Integer	4		4
SerialNo	Varchar	200	1	201
Description	Char	300		50
インデックスの合計サイズ				255

NOT MODIFIABLE

この属性は、インデックスが変更されないようにします。複数セグメントのインデックスでは、この属性は**すべての**セグメントに適用されることに留意してください。このセ

グメントのいずれかを変更しようとする、ステータス コード **10 : キー フィールドは変更できません**。が返されます。

次の例では、Person テーブルに変更できないセグメント インデックスが作成されます。

```
CREATE NOT MODIFIABLE INDEX X_Person on Person(ID, Last_Name)
```

USING

インデックスを作成するときにインデックス番号を制御するには、このキーワードを使用します。インデックス番号を制御することは、リレーショナル エンジンを使ってデータにアクセスする場合だけでなく、MicroKernel エンジンを使って直接データ ファイルからアクセスする場合にも重要です。

インデックスを作成すると、データ ファイルと X\$Index の両方に指定したインデックス番号が挿入されます。

指定したインデックス番号がどちらかのファイルで既に使用されている場合は、エラーコードが返されます。X\$Index の場合はステータス コード **5 : レコードのキー フィールドに重複するキー値があります**、データ ファイルの場合はステータス コード **6 : キー番号パラメーターが無効です**。が返されます。

```
CREATE INDEX "citizen-x" USING 3 On Person (citizenship)
```

IN DICTIONARY

このキーワードは、基となる物理データは変更しないままで DDF に変更を加えたいことを、データベース エンジンに通知します。この機能を使用すると、対応するデータ ファイルと同期していないテーブルの辞書定義を訂正したり、辞書に定義を作成して既存のデータ ファイルと合致させたりすることができます。これが必要となるのは、ほとんどの場合、Btrieve (トランザクショナル) アプリケーション (DDF を使用しない) によってデータ ファイルが作成され、使用されるときです。その場限りのクエリやレポートでは、リレーショナル エンジンを使用してデータにアクセスする必要があります。

通常、データベース エンジンは DDF とデータ ファイルの完全な同期を保ちます。IN DICTIONARY ステートメントを指定しないでインデックスを作成すると、データベース エンジンは X\$Index とデータ ファイルに同じインデックス番号を割り当てます。IN DICTIONARY は、インデックスを X\$Index にのみ追加できるようにします。

注意！ IN DICTIONARY は強力で高度な機能です。これは、システム管理者によってのみ、もしくは絶対的に必要な場合にのみ使用してください。DDF の変更を基となるデータ ファイルへの変更と並行して行わないと、正しくない結果セット、パフォーマンスの問題、予期しない結果などの重大な問題が生じることがあります。

ファントム インデックス、つまり、DDF にのみ存在しデータ ファイルには存在しないインデックスを作成した場合、IN DICTIONARY を使用しないでそのインデックスを削除しようとする、ステータスコード **6：キー番号パラメーターが無効です**。が返されることがあります。このエラーは、データベース エンジンがデータ ファイルからインデックスを削除しようとしても、そのようなインデックスはデータ ファイルに存在しないために削除できないことから発生します。

インデックスの作成時に、SQL ステートメントで IN DICTIONARY と USING の両方を使用した場合、USING で指定された番号を使用する新しいインデックスは、指定されたインデックス番号のセグメントが SQL 列と合致する場合には、DDF に**のみ**挿入されます。USING キーワードで指定された番号が SQL 列と合致しないか、またはデータ ファイルに存在しない場合、SQL エンジンは「Btrieve キー定義がインデックス定義と一致しません」というエラー メッセージを返します。これにより、ファントム インデックスが作成されないようになります。

メモ： IN DICTIONARY キーワードをバウンド データベースで使うことはできません。

例

このセクションでは、IN DICTIONARY のいくつかの例を示します。

次の例では、データ ファイルと関連付けられない「デタッチされた」テーブルが作成され、その後でテーブル定義へのインデックスの追加と削除が行われます。このインデックスは、関連付けられる基となる Btrieve インデックスが存在しないため、デタッチされたインデックスとなります。

```
CREATE TABLE t1 IN DICTIONARY (c1 int, c2 int)
CREATE INDEX idx_1 IN DICTIONARY on t1(c1)
DROP INDEX t1.idx_1 IN DICTIONARY
```

次の例では、既存のテーブル T1 を使用します。データ ファイルには key1 が定義されていますが、現在のところ X\$Index には定義されていません。

```
CREATE INDEX idx_1 USING 1 IN DICTIONARY on T1 (C2)
```

関連項目

[DROP INDEX](#)

CREATE PROCEDURE

CREATE PROCEDURE ステートメントにより、新規のストアド プロシージャを作成します。ストアド プロシージャは、あらかじめ定義されデータベース辞書に保存されている SQL ステートメントです。

構文

CREATE PROCEDURE プロシージャ名

([パラメーター [, パラメーター]...])

[**RETURNS** (結果 [, 結果]...)] [備考](#)を参照

[**WITH DEFAULT HANDLER** | **WITH EXECUTE AS 'MASTER'** | **WITH DEFAULT HANDLER** , **EXECUTE AS 'MASTER'** | **WITH EXECUTE AS 'MASTER'** , **DEFAULT HANDLER**]

as またはセミコロン

プロシージャ ステートメント

プロシージャ名 ::= ユーザー定義名

パラメーター ::= パラメーター タイプ名 [データ型](#) [**DEFAULT** プロシージャ式 | = プロシージャ式] | [SQLSTATE](#)

パラメーター タイプ名 ::= パラメーター名

| パラメーター タイプ パラメーター名

| パラメーター名 パラメーター タイプ

パラメーター タイプ ::= **IN** | **OUT** | **INOUT** | **IN_OUT**

[パラメーター名](#) ::= : ユーザー定義名

プロシージャ式 ::= 通常の式と同様。ただし、**IF** 式と **ODBC** 形式のスカラー関数は使用できない

結果 ::= ユーザー定義名 [データ型](#)

as またはセミコロン ::= **AS** | ;

プロシージャ ステートメント ::= [ラベル名 :] **BEGIN** [**ATOMIC**] [プロシージャ ステートメント [; プロシージャ ステートメント]...] **END** [ラベル名]

| **CALL** プロシージャ名 (プロシージャ式 [, プロシージャ式]...)

| **CLOSE** カーソル名

| **DECLARE** カーソル名 **CURSOR FOR** 選択ステートメント [**FOR UPDATE** | **FOR READ ONLY**]

| **DECLARE** 変数名 [データ型](#) [**DEFAULT** プロシージャ式 | = プロシージャ式]

| **DELETE WHERE CURRENT OF** カーソル名

| 削除ステートメント

| **FETCH** [フェッチ方向 [**FROM**]] カーソル名 [**INTO** 変数名 [, 変数名]]

| **IF** プロシージャ検索条件 **THEN** プロシージャ ステートメント [; プロシージャ ステートメント]... [**ELSE** プロシージャ ステートメント [; プロシージャ ステートメント]...] **END IF**

| **IF** プロシージャ検索条件 プロシージャ ステートメント [**ELSE** プロシージャ ステートメント]

| 挿入ステートメント

| **LEAVE** ラベル名

| [ラベル名 :] **LOOP** プロシージャ ステートメント [; プロシージャ ステートメント]... **END LOOP** [ラベル名]

| **OPEN** カーソル名

| **PRINT** プロシージャ式 [, ' 文字列'] -- Windows プラットフォームにのみ適用される

| **RETURN** [プロシージャ式]

| トランザクション ステートメント

| **into** 付き選択ステートメント

選択ステートメント
SET 変数名 = プロシージャ式
SIGNAL [ABORT] sqlstate 値
START TRANSACTION
更新ステートメント
UPDATE SET 列名 = プロシージャ式 [, 列名 = プロシージャ式]... WHERE CURRENT OF カーソル名
[ラベル名 :] WHILE プロシージャ検索条件 DO [プロシージャ ステートメント [; プロシージャ ステートメント]]... END WHILE [ラベル名]
テーブル変更ステートメント
インデックス作成ステートメント
テーブル作成ステートメント
ビュー作成ステートメント
インデックス削除ステートメント
テーブル削除ステートメント
ビュー削除ステートメント
権限付与ステートメント
権限取消ステートメント
設定ステートメント

トランザクション ステートメント ::= コミット ステートメント

- | ロールバック ステートメント
- | リリース ステートメント

コミット ステートメント ::= **COMMIT** を参照
 ロールバック ステートメント ::= **ROLLBACK** を参照
 リリース ステートメント ::= **RELEASE SAVEPOINT** を参照
 テーブル作成ステートメント ::= **CREATE TABLE** を参照
 テーブル変更ステートメント ::= **ALTER TABLE** を参照
 テーブル削除ステートメント ::= **DROP TABLE** を参照
 インデックス作成ステートメント ::= **CREATE INDEX** を参照
 インデックス削除ステートメント ::= **DROP INDEX** を参照
 ビュー作成ステートメント ::= **CREATE VIEW** を参照
 ビュー削除ステートメント ::= **DROP VIEW** を参照
 権限付与ステートメント ::= **GRANT** を参照
 権限取消ステートメント ::= **REVOKE** を参照
 設定ステートメント ::= **SET DECIMALSEPARATORCOMMA** を参照

ラベル名 ::= ユーザー定義名

カーソル名 ::= ユーザー定義名

変数名 ::= ユーザー定義名

プロシージャ検索条件 ::= 検索条件と同様。ただし、サブクエリを含む式は使用できない

フェッチ方向 ::= **NEXT**

sqlstate 値 ::= ' 文字列 '

備考

ストアド プロシージャを実行するには、**CALL** または **EXECUTE** ステートメントを使用します。

なお、プロシージャでは変数名とパラメーター名はコロン (:) で始まる必要があります。これは、変数やパラメーターを定義するときと使用する時の両方に当てはまります。

ストアド プロシージャが結果セットまたはスカラー値を返す場合は、**RETURNS** 句が必要です。

RETURNS 句が存在する場合は、エラーが発生したときにプロシージャが引き続き実行されます。デフォルトの動作（この句が存在しない場合）では、**SQLSTATE** がステートメントによって発生したエラー状態になり、プロシージャが中止されます。

IF ステートメントの最初（または最後）に *StmtLabel* を使用することは、ANSI SQL 3 の拡張機能です。

PRINT ステートメントは Windows ベースのプラットフォームにのみ適用されます。ほかのオペレーティング システムのプラットフォームでは無視されます。

SQL Editor において、変数パラメーターを使ってストアド プロシージャをテストする唯一の方法は、そのストアド プロシージャを別のストアド プロシージャから呼び出すことです。この手法は、`pdate` の例で示されています（`CREATE PROCEDURE pdate();`）。

変数は、ストアド プロシージャ内でのみ **SELECT** 項目として使用することができます。この手法は、`varsub1` の例で示されています（`CREATE PROCEDURE varsub1();`）。

ストアド プロシージャ内で **CREATE DATABASE** または **DROP DATABASE** ステートメントを使用することはできません。

信頼されるストアド プロシージャと信頼されないストアド プロシージャ

信頼されるストアド プロシージャには **WITH EXECUTE AS 'MASTER'** 句を含めます。[信頼されるオブジェクトと信頼されないオブジェクト](#)を参照してください。

メモリ キャッシング

デフォルトで、データベース エンジン はメモリ キャッシュを作成し、SQL セッションの継続期間中、複数のストアド プロシージャを格納します。ストアド プロシージャが実行されると、そのコンパイルされたものがメモリ キャッシュに保持されます。一般的に、キャッシングによって、キャッシュされたプロシージャの次回からの呼び出しのパフォーマンスが向上します。ストアド プロシージャが初めて実行される時には

キャッシュによってパフォーマンスは向上しません。これは、プロシージャがまだメモリに読み込まれていないためです。

2 つの SET ステートメントがメモリ キャッシュに適用されます。

- **SET CACHED_PROCEDURES** – キャッシュするプロシージャ数。デフォルト値は 50 です。
- **SET PROCEDURES_CACHE** – キャッシュに使用するメモリ量。デフォルト値は 5 MB です。

キャッシュの設定やアプリケーションが実行する SQL によっては、過度のメモリ スワッピングやスラッシングが発生するので注意してください。スラッシングはパフォーマンスの低下を招きます。

キャッシングの除外

ストアド プロシージャは以下の条件に当てはまる場合、キャッシュ設定にかかわらず、キャッシュ**されません**。

- ローカルまたはグローバルのテンポラリ テーブルを参照している。ローカル テンポラリ テーブルはポンド記号 (#) で始まる名前を持ちます。グローバル テンポラリ テーブルは 2 つのポンド記号 (##) で始まる名前を持ちます。 **CREATE (テンポラリ) TABLE** を参照してください。
- データ定義言語 (DDL) ステートメントが含まれている。 **データ定義ステートメント** を参照してください。
- 文字列、または文字列を返す式を実行するために使用される EXEC[UTE] ステートメントが含まれている。たとえば、次のような例です。 `EXEC ('SELECT Student_ID FROM ' + :myinputvar)`

データ型の制限

ストアド プロシージャまたはトリガーは、以下のデータ型をパラメーターとして渡したり、変数として宣言したりすることはできません。

BFLOAT4	BFLOAT8
MONEY	NUMERICSA
NUMERICSLB	NUMERICSLS
NUMERICSTB	NUMERICSTS

直接対応する ODBC データ型がない Zen データ型を、プロシージャで利用されるように正しくマップする方法については、[例](#)を参照してください。

制限

ストアド プロシージャを作成する際、次の制限に注意を払う必要があります。

属性	制限
トリガーまたはストアド プロシージャで使用可能な列数	300
ストアド プロシージャのパラメーター リスト内の引数の数	300
ストアド プロシージャのサイズ	64 KB

例

次の例では、ストアド プロシージャ `Enrollstudent` が作成され、`Student ID` と `Class ID` が指定されてレコードが `Enrolls` テーブルに挿入されます。

```
CREATE PROCEDURE Enrollstudent(IN :Stud_id INTEGER, IN :Class_Id INTEGER, IN :GPA REAL);
BEGIN
    INSERT INTO Enrolls VALUES(:Stud_id, :Class_id, :GPA);
END;
```

次のステートメントを使用して、ストアド プロシージャを呼び出します。

```
CALL Enrollstudent(1023456781, 146, 3.2)
```

次のステートメントを使用して、新しく挿入したレコードを取得します。

```
SELECT * FROM Enrolls WHERE Student_id = 1023456781
```

`CALL` ステートメントは引数を渡してプロシージャを呼び出し、`SELECT` ステートメントは追加された行を表示します。

この例は、パラメーターにデフォルト値を割り当てる方法を示します。

```
CREATE PROCEDURE ReportTitle1 (:rpttitle1 VARCHAR(20) = 'Finance Department') RETURNS (Title
VARCHAR(20));
BEGIN
    SELECT :rpttitle1;
END;
CALL ReportTitle1;
```

```
CREATE PROCEDURE ReportTitle2 (:rpttitle2 VARCHAR(20) DEFAULT 'Finance Department', :rptdate DATE
DEFAULT CURDATE()) RETURNS (Title VARCHAR(20), Date DATE);
BEGIN
```

```
SELECT :rpttitle2, :rptdate;
END;
CALL ReportTitle2( , );
```

これらのプロシージャは、CALL でパラメーターが提供されなかった場合には、指定したデフォルト値（Finance Department）を使用します。パラメーターを省略することはできますが、プレースホルダーを指定する必要があります。

次のプロシージャでは、呼び出し側から渡された classId パラメーターを使用して Class テーブルを読み取り、講座の登録者数がまだ制限数に達していないことを確認します。

```
CREATE PROCEDURE Checkmax(in :classid integer);
BEGIN
  DECLARE :numenrolled integer;
  DECLARE :maxenrolled integer;
  SELECT COUNT(*) INTO :numenrolled FROM Enrolls WHERE class_ID = :classid;
  SELECT Max_size INTO :maxenrolled FROM Class WHERE id = :classid;
  IF (:numenrolled >= :maxenrolled) THEN
    PRINT ' 登録は失敗しました。登録された生徒数が、この講座の制限数に達しました。';
  ELSE
    PRINT ' 登録可能です。登録された生徒数は、この講座の制限数にまだ達していません。';
  END IF;
END;
CALL Checkmax(101)
```

COUNT(式) によって、述部にある式の非ヌル値がすべてカウントされるということを覚えておってください。COUNT(*) ではヌル値を含むすべての値がカウントされます。

ストアド プロシージャを作成するときに OUT パラメーターを使用する例を以下に示します。このプロシージャを呼び出すと、WHERE 句を満たす生徒数が変数 :outval に返されます。

```
CREATE PROCEDURE ProcOUT (out :outval INTEGER)
AS BEGIN
  SELECT COUNT(*) INTO :outval FROM Enrolls WHERE Class_Id = 101;
END;
```

ストアド プロシージャを作成するときに INOUT パラメーターを使用する例を以下に示します。このプロシージャの呼び出しでは、INPUT パラメーター :IOVAL が要求され、出力の値が変数 :IOVAL に返されます。プロシージャでは、入力と IF 条件に基づいてこの変数の値が設定されます。

```
CREATE PROCEDURE ProcIODATE (INOUT :IOVAL DATE)
AS BEGIN
  IF :IOVAL = '1982-03-03' THEN
    SET :IOVAL = '1982-05-05';
  ELSE
    SET :IOVAL = '1982-03-03';
  END IF;
END;
```

```
END IF;
END;
```

上記のプロシージャは、`call prociodate('1982-03-03')` のように、リテラル値を使って呼び出すことはできません。これには **OUTPUT** パラメーターが必要になります。最初に **ODBC** 呼び出しを使ってパラメーターをバインドする必要があります。そうでなければ、次に示すように、このプロシージャを呼び出す別のプロシージャを作成することで、プロシージャのテストを行えます。

```
CREATE PROCEDURE pdate();
BEGIN
    DECLARE :a DATE;
    CALL prociodate(:a);
    PRINT :a;
END
CALL pdate
```

次の例は、プロシージャで **RETURNS** 句を使用する方法を示します。この例では、**Class** テーブルの中で、**Start Date** が、**CALL** ステートメントで渡される日付と等しいデータがすべて返されます。

```
CREATE PROCEDURE DateReturnProc(IN :PDATE DATE)
RETURNS(
    DateProc_ID INTEGER,
    DateProc_Name CHAR(7),
    DateProc_Section CHAR(3),
    DateProc_Max_Size USMALLINT,
    DateProc_Start_Date DATE,
    DateProc_Start_Time TIME,
    DateProc_Finish_Time TIME,
    DateProc_Building_Name CHAR(25),
    DateProc_Room_Number UINTEGER,
    DateProc_Faculty_ID UBIGINT
);
BEGIN
    SELECT ID, Name, Section, Max_Size, Start_Date, Start_Time, Finish_Time, Building_Name,
    Room_Number, Faculty_ID FROM Class WHERE Start_Date = :PDATE;
END;
CALL DateReturnProc('1995-06-05')
```

RETURNS 句内のユーザー定義名は、この例で示されているように、選択リストに現れる列名と同じ名前にする必要がないことに注目してください。

次の例は、位置付け **DELETE** に適用される **WHERE CURRENT OF** 句の使い方を示します。

```
CREATE PROCEDURE MyProc(IN :CourseName CHAR(7)) AS
BEGIN
    DECLARE c1 CURSOR FOR SELECT name FROM course WHERE name = :CourseName FOR UPDATE;
    OPEN c1;
    FETCH NEXT FROM c1 INTO :CourseName;
    DELETE WHERE CURRENT OF c1;
```

```
CLOSE c1;
END;
CALL MyProc('HIS 305')
```

(DELETE の WHERE 句の中で SELECT を使用した場合、それは位置付け DELETE ではなく検索済み DELETE になるので注意してください。)

次の例は、変数 (:i) を SELECT 項目として使用する方法を示します。例では、table1 はまだ存在しないものとします。person テーブルの、ID が 950000000 より大きいすべてのレコードが選択され、table1 の col2 に挿入されます。col1 には、WHILE ループで定義されるとおり、値 0、1、2、3 または 4 が格納されます。

```
CREATE TABLE table1 (col1 CHAR(10), col2 BIGINT);

CREATE PROCEDURE varsub1();
BEGIN
    DECLARE :i INT;
    SET :i = 0;
    WHILE :i < 5 DO
        INSERT INTO table1 (col1, col2) SELECT :i , A.ID FROM PERSON A WHERE A.ID > 950000000;
        SET :i = :i + 1;
    END WHILE;
END
CALL varsub1
SELECT * FROM table1
-- 110 行を返します
```

一連のステートメントがすべて成功するか失敗するように、ATOMIC を使用してステートメントを 1 つにまとめる例を以下に示します。ATOMIC はストアド プロシージャ、ユーザー定義関数、またはトリガーの本体でのみ使用できます。

最初のプロシージャでは ATOMIC を指定せず、2 番目のプロシージャで指定します。

```
CREATE TABLE t1 (c1 INTEGER)
CREATE UNIQUE INDEX t1i1 ON t1 (c1)
CREATE PROCEDURE p1();
BEGIN
    INSERT INTO t1 VALUES (1);
    INSERT INTO t1 VALUES (1);
END;
CREATE PROCEDURE p2();
BEGIN ATOMIC
    INSERT INTO t1 VALUES (2);
    INSERT INTO t1 VALUES (2);
END;
CALL p1()
CALL p2()
SELECT * FROM t1
```

どちらのプロシージャも、重複のないインデックスに重複する値を挿入しようとするためにエラーが返されます。

プロシージャ p1 内の最初の INSERT ステートメントは成功しますが、2 番目のステートメントは失敗するので、t1 には 1 レコードのみが格納されるという結果になります。同様に、プロシージャ p2 内の最初の INSERT ステートメントは成功しますが、2 番目のステートメントは失敗します。ただし、プロシージャ p2 では ATOMIC が使用されるため、エラーが発生すると、プロシージャ p2 における実行内容はすべてロールバックされます。

次の例では、ストアド プロシージャを使用して 2 つのテーブルを作成し、デフォルト値を設定した 1 行をそれぞれのテーブルに挿入します。次に、セキュリティを有効にしてユーザー user1 に権限を与えます。

```
CREATE PROCEDURE p1();
BEGIN
    CREATE TABLE t1 (c1 INT DEFAULT 10, c2 INT DEFAULT 100);
    CREATE TABLE t2 (c1 INT DEFAULT 1, c2 INT DEFAULT 2);
    INSERT INTO t1 DEFAULT VALUES;
    INSERT INTO t2 DEFAULT VALUES;
    SET SECURITY = larry;
    GRANT LOGIN TO user1 u1pword;
    GRANT ALL ON * TO user1;
END;
CALL p1
SELECT * FROM t1
    -- 10、100 を返します
SELECT * FROM t2
    -- 1、2 を返します
```

メモ： ストアド プロシージャで GRANT LOGIN ステートメントを使用する場合は、ユーザー名とパスワードをコロンではなく空白文字で区切る必要があります。コロン文字は、ストアド プロシージャ内でローカル変数を識別するために予約されています。

次の例では、ストアド プロシージャを使用してユーザー user1 の権限を取り消し、前の例で作成した 2 つのテーブルを削除してデータベース セキュリティを無効にします。

```
CREATE PROCEDURE p3();
BEGIN
    REVOKE ALL ON t1 FROM user1;
    REVOKE ALL ON t2 FROM user1;
    DROP TABLE t1;
    DROP TABLE t2;
    SET SECURITY = NULL;
END;
CALL p3
SELECT * FROM t1 -- テーブルが見つからないというエラーを返します
SELECT * FROM t2 -- テーブルが見つからないというエラーを返します
```

次の例は、カーソル内をループする方法を示します。

```
CREATE TABLE atable (c1 INT, c2 INT)
INSERT INTO atable VALUES (1,1)
INSERT INTO atable VALUES (1,2)
INSERT INTO atable VALUES (2,2)
INSERT INTO atable VALUES (2,3)
INSERT INTO atable VALUES (3,3)
INSERT INTO atable VALUES (3,4)

CREATE PROCEDURE pp();
BEGIN
  DECLARE :i INTEGER;
  DECLARE c1Bulk CURSOR FOR SELECT c1 FROM atable ORDER BY c1 FOR UPDATE;
  OPEN c1Bulk;
  BulkLinesLoop:
  LOOP
    FETCH NEXT FROM c1Bulk INTO :i;
    IF SQLSTATE = '02000' THEN
      LEAVE BulkLinesLoop;
    END IF;
    UPDATE SET c1 = 10 WHERE CURRENT OF c1Bulk;
  END LOOP;
  CLOSE c1Bulk;
END

CALL pp
-- 正常終了
SELECT * FROM atable
-- 6 行を返します
```

次の例では、**InParam** という名前の信頼されるストアード プロシージャを作成します。その後、**Master** ユーザーが **User1** に対し、**InParam** の **EXECUTE** および **ALTER** 権限を与えます。この例は、テーブル **t99** が存在し、テーブルに **INTEGER** 型の 2 つの列があることを前提としています。

```
CREATE PROCEDURE InParam(IN :inparam1 INTEGER, IN :inparam2 INTEGER) WITH DEFAULT HANDLER, EXECUTE AS
'Master' AS
BEGIN
  INSERT INTO t99 VALUES(:inparam1 , :inparam2);
END;
GRANT ALL ON PROCEDURE InParam TO User1
```

Master および **User1** は、このプロシージャを呼び出せるようになりました（たとえば、**CALL InParam(2,4)**）。

次の例は、直接対応する **ODBC** データ型がない **Zen** データ型を、プロシージャで使われるように正しくマップする方法を示します。データ型 **NUMERICSA** および **NUMERICSTS** は、直接対応がないデータ型なので、代わりに **NUMERIC** にマップされます。

```
CREATE TABLE test1 (id identity, amount1 numeric(5,2), amount2 numericsa(5,2), amount3
numericsts(5,2))
CREATE PROCEDURE ptest2 (IN :numval1 numeric(5,2), IN :numval2 numeric(5,2), IN :numval3
numeric(5,2))
```

```
AS
BEGIN
Insert into test1 values(0, :numval1, :numval2, :numval3);
END;

CALL ptest2(100.10, 200.20, 300.30)
SELECT * FROM test1
```

プロシージャは、`amount` 値がすべて `NUMERIC` としてプロシージャに渡されるにもかかわらず、これらを `CREATE TABLE` ステートメントで定義された Zen データ型に従って正しく書式指定します。データ型のマップについては、[Zen で使用できるデータ型](#)を参照してください。

ストアド プロシージャの使用

一例として、`CALL foo(a, b, c)` は、パラメーター `a`、`b`、および `c` を持つストアド プロシージャ `foo` を実行します。パラメーターはいずれも動的パラメーター ('?') にできます。これは、`OUTPUT` パラメーターおよび `INOUT` パラメーターの値を取り出す場合は必須です。たとえば、`{CALL foo(?, ?, 'TX')}` のように指定します。ソースコードでは、中かっこは省略可能です。

以下は、現バージョンの Zen におけるストアド プロシージャのしくみです。

- トリガー (`CREATE TRIGGER`、`DROP TRIGGER`) が、ストアド プロシージャの形式としてサポートされます。このサポートには、テーブル、プロシージャ、およびデータベースに対するトリガーの依存性の監視が含まれます。`CREATE PROCEDURE` および `CREATE TRIGGER` は、ストアド プロシージャまたはトリガーの本体では使用できません。
- `CONTAINS`、`NOT CONTAINS`、`BEGINS WITH` はサポートされていません。
- `LOOP`：事後条件ループはサポートされていません (`REPEAT...UNTIL`)。
- `ELSEIF`：条件形式には `IF ... THEN ... ELSE` が使用されます。`ELSEIF` はサポートされていません。

全般的なストアド プロシージャ エンジンの制約

ストアド プロシージャを使用する前に、次に挙げる制約を確認してください。

- 修飾子は、`CREATE PROCEDURE` でも `CREATE TRIGGER` でもサポートされていません。
- ストアド プロシージャ変数名の最大長は半角 128 文字です。

- ストアド プロシージャ名の最大長については、『*Advanced Operations Guide*』の[識別子の制限](#)を参照してください。
- CREATE PROCEDURE または CREATE TRIGGER のとき、構文の検証は部分的にのみ行われます。列名は実行時まで確認されません。
- 現在は、式が使用されているすべての場所でサブクエリを使用できるわけではありません。たとえば、`set :arg = SELECT MIN(sal) FROM emp` を使用する UPDATE ステートメントはサポートされません。ただし、このサブクエリを `SELECT min(sal) INTO :arg FROM emp` のように書き直すことができます。
- デフォルトのエラー ハンドラーのみがサポートされています。

SQL 変数とパラメーターの制約

- 変数名はコロン (:) またはアットマーク (@) で始まる必要があります。これによって、ストアド プロシージャのパースが変数と列名を区別できるようになります。
- 変数名は、大文字と小文字が区別されません。
- セッション変数はサポートされていません。変数はプロシージャに対してローカルです。

カーソルの制約

- 位置付け UPDATE はテーブル名を受け入れません。
- グローバル カーソルはサポートされていません。

Long データを使用する際の制約

- Long データを引数として埋め込みプロシージャ（別のプロシージャを呼び出すプロシージャ）に渡すと、データは切り捨てられて 65500 バイトになります。
- プロシージャとの間でやり取りされる Long データの引数は、合計 2 MB に制限されています

Long データは、内部的にはデータ長の制限なしにカーソル間でコピーできます。Long データ列をステートメントからフェッチして別のステートメントに挿入する場合、制約はありません。ただし、1 つの Long データ変数に対して複数の宛先が要求される場合は、最初の宛先テーブルのみが複数の PutData 呼び出しを受け取ります。残りの列は切

り捨てられ、最初の 65500 バイトになります。これは ODBC GetData のメカニズムの制約です。

関連項目

[DROP PROCEDURE](#)

[SET CACHED_PROCEduRES](#)

[SET PROCEDUREs_CACHE](#)

[信頼されるオブジェクトと信頼されないオブジェクト](#)

CREATE TABLE

CREATE TABLE ステートメントにより、データベース内に新しいテーブルを作成します。

CREATE TABLE には、最小またはコア SQL 準拠よりも高度な機能が含まれています。CREATE TABLE では参照整合性機能がサポートされます。Zen は、*CollIDList* サポートを除いて、SQL 92 にほぼ準拠しています。

CREATE TABLE ステートメントを使用してテンポラリ テーブルを作成することもできます。[CREATE \(テンポラリ\) TABLE](#) を参照してください。

注意！ 同じディレクトリに、ファイル名が同じで、拡張子のみが異なるファイルが2つ存在してはいけません。たとえば、同じディレクトリ内に Invoice.btr と Invoice.mkd というデータ ファイルを作成しないでください。データベース エンジン はファイル名の拡張子を見捨てるため、2つのファイルを同一のものとして扱おうとし、エラーが発生する可能性があります。

構文

```
CREATE TABLE [ IF NOT EXISTS ] テーブル名 [ オプション [ オプション... ] ] [ IN DICTIONARY ]
[ USING 'パス名' ] [ WITH REPLACE ]
( テーブル要素 [ , テーブル要素 ]... )
```

テーブル名 ::= ユーザー定義名

オプション ::= DCOMPRESS | PCOMPRESS | PAGESIZE = サイズ | LINKDUP = 回数 | SYSDATA_KEY_2

回数 ::= ユーザー定義値 (リンク重複インデックス キーを追加するために予約しておくポインタの数を設定します)

テーブル要素 ::= 列定義 | テーブル制約定義

列定義 ::= 列名 データ型 [DEFAULT デフォルト値の式] [列制約定義 [列制約定義]... [CASE (文字列) | COLLATE 照合順序名]] [MASKED WITH (FUNCTION = マスク関数)]

列名 ::= ユーザー定義名

データ型 ::= データ型名 [(桁数 [, 小数位])]

マスク関数 ::= DEFAULT()

桁数 ::= 整数

小数位 ::= 整数

デフォルト値の式 ::= デフォルト値の式 + デフォルト値の式

	デフォルト値の式 - デフォルト値の式
	デフォルト値の式 * デフォルト値の式
	デフォルト値の式 / デフォルト値の式
	デフォルト値の式 & デフォルト値の式

```

| デフォルト値の式 | デフォルト値の式
| デフォルト値の式 ^ デフォルト値の式
| ( デフォルト値の式 )
| - デフォルト値の式
| + デフォルト値の式
| ~ デフォルト値の式
| ?
| リテラル
| スカラー関数
| { fn スカラー関数 }
| USER
| NULL

```

リテラル ::= ' 文字列 ' | N' 文字列 '

```

| 数字
| { d ' 日付リテラル ' }
| { t ' 時刻リテラル ' }
| { ts ' タイムスタンプ リテラル ' }

```

スカラー関数 ::= **スカラー関数** を参照

列制約定義 ::= [**CONSTRAINT** 制約名] 列制約

制約名 ::= ユーザー定義名

```

列制約 ::= NOT NULL
| NOT MODIFIABLE
| UNIQUE
| PRIMARY KEY
| REFERENCES テーブル名 [ ( 列名 ) ] [ 参照アクション ]

```

テーブル制約定義 ::= [**CONSTRAINT** 制約名] テーブル制約

```

テーブル制約 ::= UNIQUE ( 列名 [ , 列名 ]... )
| PRIMARY KEY ( 列名 [ , 列名 ]... )
| FOREIGN KEY ( 列名 [ , 列名 ] )
| REFERENCES テーブル名 [ ( 列名 [ , 列名 ]... ) ] [ 参照アクション ]

```

参照アクション ::= 参照更新アクション [参照削除アクション]

```

| 参照削除アクション [ 参照更新アクション ]

```

参照更新アクション ::= **ON UPDATE** **RESTRICT**

```

参照削除アクション ::= ON DELETE CASCADE
| ON DELETE RESTRICT

```

照合順序名 ::= ' 文字列 '

備考

CREATE TABLE ステートメントで作成されるインデックスは、IDENTITY、SMALLIDENTITY、BIGIDENTITY、主キー、および外部キーのみです。その他のインデックスは **CREATE INDEX** ステートメントで作成する必要があります。

外部キー制約名は、辞書内で固有の名前でなければなりません。その他の制約名はすべて、常駐するテーブル内で固有の名前であり、列とは異なる名前であればなりません。

主キー名が省略された場合は、キー内の最初の列名に接頭辞 "PK_" を付けた名前が制約名として使用されます。

参照列が一覧にない場合は、デフォルトでは参照されるテーブルの主キーが参照の対象になります。主キーが使用できない場合は、" キーがありません " というエラーが返されます。ターゲット列を列挙すると、このような状況を回避できます。

外部キー名が省略された場合は、キー内の最初の列名に接頭辞 "FK_" を付けた名前が制約名として使用されます。

UNIQUE 制約が省略された場合は、キー内の最初の列名に接頭辞 "UK_" を付けた名前が制約名として使用されます。

NOT MODIFIABLE 制約が省略された場合は、キー内の最初の列名に接頭辞 "NM_" を付けた名前が制約名として使用されます。(NOT MODIFIABLE が使用されている場合は、列に一意でない変更不可能なインデックスが作成されます。このインデックスは、NM_ 列名となります。)

NOT NULL 制約が省略された場合は、キー内の最初の列名に接頭辞 "NN_" を付けた名前が制約名として使用されます。

外部キーが同じテーブルの主キーを参照する場合があります。このキーは自己参照キーと呼ばれます。

CREATE TABLE が成功して USING 句が指定されていない場合、作成されたテーブルのデータ ファイル名は xxx.mkd になります。xxx は指定したテーブル名です。物理ファイルの xxx.mkd が既に存在する場合は xxxnnn.mkd という名前の新しいファイルが作成されます。nnn は一意の番号です。テーブルが既に存在する場合、そのテーブルは置換されず、エラー - 1303 " テーブルは既に存在します " が返されます。テーブルを置換する前に削除する必要があります。

SYSDATA_KEY_2 キーワードを指定した CREATE TABLE ステートメントでは、ファイルは自動的に 13.0 ファイル形式で作成されます。新しいファイルはシステム データ v2 を使用するため、仮想列 sys\$create および sys\$update をクエリでできるようになります。ファイル形式が 16.0 の場合、バージョンは変更されません。詳細については、[システム データ v2 のアクセス](#)を参照してください。

SYSDATA_KEY_2 キーワードと一緒に IN DICTIONARY を使用した場合、CREATE TABLE ステートメントは SYSDATA_KEY_2 を無視するため、新しいテーブルでは sys\$create および sys\$update 仮想列を使用できません。

レコード サイズの制限

データ レコードの固定長部分の合計サイズは、65535 バイトを超えてはいけません。データ レコードの固定長部分は次の要素で構成されます。

- 固定サイズのすべての列 (LONGVARCHAR 型、LONGVARBINARY 型、および NLONGVARCHAR 型以外のすべての列)
- ヌル値を許可する各列について 1 バイト
- 各可変長列 (LONGVARCHAR 型、LONGVARBINARY 型、または NLONGVARCHAR 型の列) について 8 バイト

この制限を超えるテーブルの作成を試みるか、結果としてテーブルが制限を超えるような変更を試みると、Zen はステータス コード -3016 " テーブルの最大固定長行サイズを超えています " を返します。

新しいテーブルを作成する前にレコードの固定長部分のバイト単位のサイズを調べるには、次の計算式を使用します。

(固定長列のバイト単位の記憶域サイズ合計) + (ヌル値を許可する列の数) + (8 * 可変長列の数) = バイト単位のレコード サイズ

既存データ ファイルのレコードの固定長部分のサイズを調べたい場合は、BUTIL - STAT コマンドを使用すれば、この情報を示すレポートを表示できます。

レコード サイズの制限の例

次のような列が定義されているテーブルがあるとします。

データ型	このデータ型の列数	ヌル値を許可
VARCHAR(216)	1	可
VARCHAR(213)	5	すべての列
CHAR(42)	1494	すべての列

VARCHAR にはそれぞれ余分に 2 バイトが予約されます。先頭のヌル インジケータ用の 1 バイトと末尾の 1 バイトです。末尾のバイトは、VARCHAR が ZSTRING として実装されるためです。CHAR にはそれぞれ先頭のバイトがヌル インジケータ用に予約されます。

したがって、レコード サイズは $1 \times 218 + 5 \times 215 + 1494 \times 43 = 65535$ バイトになります。

この例では、別の列がどのような長さであっても、固定長制限を超えないで追加することはできません。

削除規則

外部キー制約を持つ ON DELETE キーを含めることで、外部キー値が参照する親テーブルの行を削除しようとする場合に Zen により適用される削除規則を定義することができます。選択可能な削除規則は次のとおりです。

- CASCADE を指定した場合、Zen はカスケード削除規則を使用します。ユーザーが親テーブルの行を削除すると、データベース エンジンによって従属テーブルの対応する行が削除されます。
- RESTRICT を指定した場合、Zen は削除制限規則を使用します。外部キー値が親テーブルの行を参照している場合、ユーザーはその行を削除できません。

削除規則を指定しない場合、Zen はデフォルトで制限規則を適用します。

DELETE CASCADE は注意して使用してください。Zen では、自己参照するテーブルに対し、循環するカスケード削除を使用できます。『*Advanced Operations Guide*』の[削除カスケード](#)に記載されている例を参照してください。

更新規則

Zen は更新制限規則を適用します。この規則は、親テーブルが外部キー値と合致する主キー値を持たない場合、その外部キー値を含む行が追加されないようにします。この規則が適用されるかどうかは、更新規則を明示的に指定できるオプションの ON UPDATE 句の使用によります。

IN DICTIONARY

ALTER TABLE の IN DICTIONARY の解説を参照してください。

USING

USING キーワードを使用すると、特定のデータ ファイルを CREATE TABLE または ALTER TABLE アクションと関連付けることができます。

Zen は接続に名前付きデータベースを必要とするので、指定するパス名は常に単純なファイル名であるか、または相対パスとファイル名でなければなりません。パスは常に、接続する名前付きデータベースに指定された最初のデータ パスとの相対になります。

渡されたパス名およびファイル名は、ステートメントの準備ができたときに部分的に検証されます。

パス名を指定するときは、次の規則に従う必要があります。

- テキストは、文法定義で示されているように、一重引用符で囲まなければなりません。
- テキストの長さは、使用しているメタデータのバージョンの制限を超えることはできません。エントリは入力したまま正確に `Xf$Loc` に格納されます（ただし、後続の空白は切り捨てられ、無視されます）。`Xf$Loc`（メタデータ バージョン 1 用）および `Xf$Loc`（メタデータ バージョン 2 用）を参照してください。
- パスは、単純な相対パスでなければなりません。サーバーまたはボリュームを参照するパスは許可されません。
- 相対パスには、1 つのピリオド（現在のディレクトリ）、2 つのピリオド（親ディレクトリ）、円記号、またはこれら 3 つのあらゆる組み合わせを含めることができます。ただし、パスは SQL テーブル名を表すファイル名を含んでいる必要があります。つまり、**パス名**は円記号やディレクトリ名で終わってはいけないということです。ファイル名は、相対パス付きで指定されたファイル名も含めてすべて、名前付きデータベースの設定で定義されている最初のデータ パスとの相対になります。

次の機能は、利便性と使いやすさを提供します。

- ルート ベースの相対パスを使用できます。たとえば、最初のデータ パスを `D:¥mydata¥demodata` とした場合、Zen は次のステートメント内のパス名を `D:¥temp¥test123.btr` と解釈します。

```
CREATE TABLE t1 USING '¥temp¥test123.btr' (c1 int)
```

- 相対パス内の円記号文字は、Unix スタイル (`/`) と Windows の円記号 (`¥`) のどちらでも使用できます。必要であれば、2 種類の記号を混在させて使用することもできます。ディレクトリ構造スキーマは知っているかもしれませんが、接続されている

サーバーの種類を知っている（あるいは管理している）とは限らないので、これは便利な機能です。パスは入力したとおりに `X$File` に格納されます。**Zen** エンジン は、パスを利用してファイルを開く際、円記号文字を適切なプラットフォームのタイプに変換します。また、データ ファイルはサポートされるすべてのプラットフォーム間でバイナリ互換性を共有するため、ディレクトリ構造がプラットフォーム間で同一である（および、パスに基づくファイル名が相対パスで指定されている）限りは、データベース ファイルおよび DDF をこれらに変更を加えることなく、あるプラットフォームから別のプラットフォームへ移動することができます。これにより、標準化されたデータベース スキーマを使用したクロスプラットフォームの展開が可能になります。

- 相対パスを指定するとき、**USING** 句のディレクトリ構造は既存である必要はありません。**Zen** は必要に応じて、**USING** 句のパスのためにディレクトリを作成します。

USING 句を使用して、テーブルと関連付けるデータ ファイルの物理的な場所を指定します。これは、既存データ ファイルのテーブル定義を作成するとき、または新しいデータ ファイルの名前や物理的な場所を明示的に指定したい場合は必須です。

USING 句を使用しないと、**Zen** により、テーブル名を基に拡張子 `.mkd` の一意のファイル名が生成され、データベースのデータ ファイルパスに指定されている最初のディレクトリにファイルが作成されます。

USING 句が既存のデータ ファイルを示す場合は、**Zen** により DDF にテーブルが作成され、`SQL_SUCCESS_WITH_INFO` が返されます。返された情報メッセージは、辞書エントリが既存のデータ ファイルをポイントするようになったことを示します。**CREATE TABLE** で `SQL_SUCCESS` のみが返されるようにしたい場合は、**CREATE** ステートメントで `IN DICTIONARY` を指定します。**WITH REPLACE** を指定する（下記参照）と、同一名の既存データ ファイルはすべて破棄され、新規に作成したファイルで上書きされます。

メモ： 既存のデータ ファイルを指定した場合、**Zen** は成功を表すステータス コードを返します。

既存のデータ ファイルに対してリレーショナル インデックス定義を作成する（たとえば、**IDENTITY** 型の列定義を含む **CREATE TABLE USING** を使用する）たびに、**Zen** はそのファイルに定義されている **Btrieve** インデックスを自動的にチェックし、既存の **Btrieve** インデックスがリレーショナル インデックス定義のパラメーターのセットを提供しているかどうかを判定します。既存の **Btrieve** インデックスと新しい定義が一致する場合は、リレーショナル インデックス定義と既存の **Btrieve** インデックスの間に関連付けが作成されます。一致するインデックスがない場合は、**Zen** は新しいインデックス

定義を作成し、IN DICTIONARY が使用されていなければ、ファイルに新しいインデックスを作成します。

WITH REPLACE

WITH REPLACE が USING キーワードと共に指定されたときはいつでも、Zen は既存のファイル名を指定されたファイル名で自動的に上書きします。オペレーティング システムがファイルの上書きを許している場合、ファイルは常に上書きされます。WITH REPLACE はデータ ファイルにのみ作用します。DDF には作用しません。

WITH REPLACE を使用する際には次の規則が適用されます。

- WITH REPLACE は USING と併せてのみ使用できます。
- IN DICTIONARY と一緒に使用すると、WITH REPLACE は無視されます。IN DICTIONARY は DDF にのみ作用することを指定するものだからです。

CREATE TABLE ステートメントで WITH REPLACE を使用した場合、USING 句で指定された場所にファイルが存在すると、Zen は新規データ ファイルを作成してその既存ファイルと置き換えます。Zen は、同じ名前を持つ元のファイルに格納されているデータをすべて破棄します。WITH REPLACE を使用していないとき、指定された場所にファイルが存在すると、Zen はステータス コードを返し、新しいファイルを作成しません。ただし、テーブル定義は DDF に追加されます。

WITH REPLACE はデータ ファイルにのみ作用します。辞書内のテーブル定義には作用しません。

DCOMPRESS

DCOMPRESS オプションは、ディスク上のファイル サイズを減らすために、テーブルのデータ ファイルがレコード圧縮を使用することを指定します。次の例では、レコード圧縮を行う、ページ サイズが 1024 バイトのテーブルが作成されます。

```
CREATE TABLE t1 DCOMPRESS PAGESIZE=1024 (c1 INT DEFAULT 10, c2 CHAR(10) DEFAULT 'abc')
```

詳細については、『*Advanced Operations Guide*』の[レコードおよびページ圧縮](#)を参照してください。

PCOMPRESS

PCOMPRESS オプションは、指定したテーブルのデータ ファイルがページ圧縮を使用することを指定します。次の例では、ページ圧縮を行う、ページ サイズが 1024 バイトのテーブルが作成されます。

```
CREATE TABLE t1 PCOMPRESS PAGESIZE=1024 (c1 INT DEFAULT 10, c2 CHAR(10) DEFAULT 'abc')
```

詳細については、『*Advanced Operations Guide*』の[レコードおよびページ圧縮](#)を参照してください。

PAGESIZE

PAGESIZE オプションは、指定したテーブルのデータ ファイルが使用するページの**サイズ** バイト数を指定します。**サイズ**の値には、ファイルバージョンに応じて次のような値を指定できます。

- 512 – 4096 : 9.0 より前のファイルバージョンの場合 (512 バイトの倍数で最大 4096 バイト)
- 512、1024、1536、2048、2560、3072、3584、4096、または 8192 : ファイルバージョンが 9.0 の場合
- 1024、2048、4096、8192、または 16384 : ファイルバージョンが 9.5 の場合
- 4096、8192 または 16384 (ファイルバージョンが 13.0 および 16.0 の場合)

次の例では、ファイル圧縮を行う、ページ サイズが 8192 バイトのテーブルが作成されると共に、相対パスによって識別される特定のデータ ファイル、`..¥data1.mkd` の作成が指定されます。

```
CREATE TABLE t1 DCOMPRESS PAGESIZE=8192 USING '..¥data1.mkd' (c1 INT DEFAULT 10, c2 CHAR(10) DEFAULT 'abc')
```

LINKDUP

複数のレコードが、インデックス キーに同一の重複した値を持つことができます。重複キー値を持つレコードを記憶しておく 2 つの方法は、リンク重複 (linkdup) および繰り返し重複と呼ばれますリンク重複と繰り返し重複の詳細については、『*Advanced Operations Guide*』の[重複キーの操作方法](#)を参照してください。

LINKDUP キーワードが**指定されていない**場合、[CREATE INDEX](#) ステートメントは繰り返し重複を使用します。

物理レコード内で、リンク重複インデックスごとに 8 バイトが余分に必要になります。LINKDUP キーワードを使用すると、これ以降に作成するリンク重複インデックス用にこれらの余分なバイトを予約することができます。

したがって、LINKDUP キーワードが**指定されている**場合は、以下が適用されます。

- **CREATE INDEX** ステートメントは、ポインター数に指定した値に達するまではリンク重複方法を使用します。
- ポインター数に指定した値に達したら、**CREATE INDEX** ステートメントは繰り返し重複方法を使用します。
- ポインター数に指定した値に達しているとき、リンク重複インデックスを削除すると、**CREATE INDEX** ステートメントは次のキーに対してリンク重複方法を使用します。
- **CREATE INDEX** ステートメントは、リンク重複キー用のポインターが予約されていない場合には繰り返し重複キーを作成できません。

例

以下の例では、CREATE TABLE のさまざまな使用法を示します。

次のような構文では、指定されたデータ型の Student_ID、Transaction_Number、Log、Amount_Owed、Amount_Paid、Registrar_ID、および Comments の列から成る、Billing という名前のテーブルが作成されます。

```
CREATE TABLE Billing
(Student_ID UBIGINT,
Transaction_Number USMALLINT,
Log TIMESTAMP,
Amount_Owed DECIMAL(6,2),
Amount_Paid DECIMAL(6,2),
Registrar_ID DECIMAL(10,0),
Comments LONGVARCHAR)
```

この例では、ID、Dept_Name、Designation、Salary、Building_Name、Room_Number、および Rsch_Grant_Amount の列から成り、列 ID に基づく主キーを持つ Faculty という名前のテーブルがデータベースに作成されます。

```
CREATE TABLE Faculty
(ID UBIGINT,
Dept_Name CHAR(20) CASE,
Designation CHAR(10) CASE,
Salary CURRENCY,
Building_Name CHAR(25) CASE,
Room_Number UINTEGER,
```

```
Rsch_Grant_Amount DOUBLE,  
PRIMARY KEY(ID))
```

次の例では、Name 列にインデックスを作成し、変更不可を指定します。Name 列のデータは変更できません。

```
CREATE TABLE organizations  
(Name LONGVARCHAR NOT MODIFIABLE,  
Advisor CHAR(30),  
Number_of_people INTEGER,  
Date_started DATE,  
Time_started TIME,  
Date_modified TIMESTAMP,  
Total_funds DOUBLE,  
Budget DECIMAL(2,2),  
Avg_funds REAL,  
President VARCHAR(20) CASE,  
Number_of_executives SMALLINT,  
Number_of_meetings TINYINT,  
Office UTINYINT,  
Active BIT,)
```

次の例では、生徒の住所を保存するために StudentAddress というテーブルが必要であるとし、Student テーブルの *id* 列を主キーに変更してから、Student を主テーブルとして参照する StudentAddress テーブルを作成する必要があります (Student テーブルは Demodata サンプル データベースの一部です)。StudentAddress テーブルの作成方法を 4 種類示します。

まず、Student テーブルの *id* 列を主キーにします。

```
ALTER TABLE Student ADD PRIMARY KEY (id)
```

次のステートメントは、DELETE CASCADE 規則を使って、Student テーブルの *id* 列を参照する外部キーを持つ StudentAddress テーブルを作成します。これは、Student テーブル (この場合、Student は親テーブルになる) から行が削除されるときはいつも、StudentAddress テーブル内でその行と同じ *id* を持つすべての行も削除されることを示します。

```
CREATE TABLE StudentAddress (id UBIGINT REFERENCES Student (id) ON DELETE CASCADE, addr CHAR(128))
```

次のステートメントは、DELETE RESTRICT 規則を使って、Student テーブルの *id* 列を参照する外部キーを持つ StudentAddress テーブルを作成します。これは、Student テーブルから行が削除されたとき、その行と同じ *id* を持つ行が StudentAddress テーブルに存在する場合は常に、エラーが発生することを示します。親テーブルである Student テーブルの行を削除するには、事前に StudentAddress テーブルでその *id* を持つ行をすべて明示的に削除する必要があります。

```
CREATE TABLE StudentAddress (id UBIGINT REFERENCES Student (id) ON DELETE RESTRICT, addr CHAR(128))
```

次のステートメントは、UPDATE RESTRICT 規則を使って、Student テーブルの *id* 列を参照する外部キーを持つ StudentAddress テーブルを作成します。これは、Student テーブルに存在しない *id* を持つ行が StudentAddress テーブルに追加された場合は、エラーが発生することを示します。言い換えると、外部キーに親の行を参照させるには、その前に親の行がなければならないということです。これは Zen のデフォルトの動作です。

なお、Zen ではこれ以外の更新規則を一切サポートしていません。したがって、この規則は明示的に示しても示さなくてもかまいません。また、削除規則を明示的に指定していないので、DELETE RESTRICT が使用されます。

```
CREATE TABLE StudentAddress (id UBIGINT REFERENCES Student (id) ON UPDATE RESTRICT, addr CHAR(128))
```

この例は、テーブルの作成時にオルタネート コレーティング シーケンス (ACS) を使用する方法を示しています。使用する ACS ファイルは、Zen で提供されるサンプル ファイルです。

```
CREATE TABLE t5 (c1 CHAR(20) COLLATE 'file_path*upper.alt')
```

upper.alt は、ソートする際に大文字と小文字を同等に扱います。たとえば、データベースに *abc*、*ABC*、*DEF*、*Def* という値がこの順序で挿入されている場合、*upper.alt* を使ってソートすると、*abc*、*ABC*、*DEF*、*Def* のように返されます

abc と *ABC* という値、*DEF* と *Def* という値は、まったく同一と見なされ、挿入された順序で返されます。標準の ASCII ソートでは、大文字は小文字の前に配列されており、ソート結果は *ABC*、*DEF*、*Def*、*abc* となります。また、`SELECT c1 FROM t5 WHERE c1 = 'Abc'` ステートメントは、*abc* および *ABC* を返します。

次の例は、テーブル *t1* を作成し、リンク重複キー用に予約するポインターの数を 4 に設定します。`CREATE INDEX` ステートメントによって、テーブルにインデックス キーを作成します。

```
DROP table t1
CREATE TABLE t1 LINKDUP=4 (c1 int, c2 int, c3 int)
CREATE INDEX link_1 on t1(c1,c2)
CREATE INDEX link_2 on t1(c1,c3)
CREATE UNIQUE INDEX link_3 on t1(c3)
CREATE INDEX link_4 on t1(c1)
CREATE INDEX link_5 on t1(c2)
CREATE INDEX link_6 on t1(c2,c3)
```

`CREATE INDEX` ステートメントの結果は次のようになります。

- リンク重複キー : *link_1*、*link_2*、*link_4*、*link_5*

- 繰り返し重複キー：link_6（リンク重複キーに使用されるポインタの数が指定値の 4 に達したためです）

```
DROP INDEX link_2
CREATE INDEX link_7 on t1(c3,c1)
```

これら 2 つのステートメントを実行すると、次のようになります。

- リンク重複キー：link_1、link_4、link_5、link_7（DROP INDEX ステートメントによって、リンク重複キーに使用されるポインタの数が 3 に減ったので、link_7 を 4 番目のリンク重複キーにすることができます）
- 繰り返し重複キー：link_6

次のステートメントでは、テーブルを作成してヌル値を許可しないことを指定します（つまり、ヌル インジケータ バイトは追加されません）。

```
CREATE TABLE NoNulls
(ID UBIGINT NOT NULL,
Name CHAR(20) NOT NULL CASE,
Amount DOUBLE NOT NULL)
```

すべての列を NOT NULL として作成する必要がある場合は、まず SET TRUENULLCREATE ステートメントを使用して真のヌルの作成を無効にしてからテーブルを作成することができます。これにより、各列で NOT NULL 属性を指定することを避けられます（[SET TRUENULLCREATE](#) を参照してください）。ただし、結果として生じるレガシー テーブルでは、どの列にも NOT NULL 属性が適用されないことに注意してください。列に NOT NULL が明示的に指定されていても、NULL は許可されます。次のステートメントでは、前の例と同じテーブルを作成します。

```
SET TRUENULLCREATE=OFF
CREATE TABLE NoNulls2
(ID BIGINT,
Name CHAR(20) CASE,
Amount DOUBLE)
SET TRUENULLCREATE=ON
```

CREATE TABLE では、列に DEFAULT 値を指定することができます。これは、その列に明示的に値を指定しないで行を挿入した場合に使用されます。次のステートメントは、列のデータ型に対応するデフォルト値を持つテーブルを作成します。IDENTITY 列は暗黙的にデフォルト値ゼロを持ち、次に最も大きい値が自動生成されることに留意してください。

```
CREATE TABLE Defaults
```

```
(ID IDENTITY,
Name CHAR(20) DEFAULT 'none',
Amount DOUBLE DEFAULT 0.00,
EntryDay DATE DEFAULT CURDATE(),
EntryTime TIME DEFAULT CURTIME())
```

次のステートメントは、デフォルトを使用して 2 行を挿入します。

```
INSERT INTO Defaults (ID) VALUES (0)
INSERT INTO Defaults (ID, Name, Amount) VALUES (0, 'Joquin', '100')
```

SELECT ステートメントはデフォルト値を持つ結果を返します。

```
SELECT * FROM Defaults
ID      Name      Amount      EntryDay      EntryTime
===      =====      =====      =====      =====
1      none          0.0          curdate        curtime
2      Joquin        100.0         curdate        curtime
```

次の例では、データ ファイル `olddata.dat` に旧形式のデータ型の列を含む `Legacydata` というテーブルがあると仮定します。新しいデータベースでは旧形式のデータ型を持つテーブルを作成することはできません。ただし、`IN DICTIONARY` 句を使用して新しいデータベースに旧形式データの DDF 定義を作成することができます。

```
CREATE TABLE "Legacydata" IN DICTIONARY USING 'olddata.dat' (
"col1" LSTRING(10) NOT NULL,
"col2" VARCHAR(9) NOT NULL,
"col3" LOGICAL NOT NULL,
"col4" LOGICAL2 NOT NULL,
"col5" NOTE(100) NOT NULL);
```

この例では、`USING` 句または `REPLACE` のいずれも指定せずにテーブルを作成した場合の、`Btrieve` データ ファイルのデフォルトの作成方法を示しています。ファイルのデフォルト名は、拡張子 `.mkd` のテーブル名です。そのファイル名が既に存在する場合は、テーブル名の後に数字、`.mkd` 拡張子が続く、別の名前が生成されます。

次のようにテーブル `xyz` を作成します。これにより、データ ファイル `xyz.mkd` が生成されます。

```
CREATE TABLE xyz (c1 int, c2 char(5))
```

次に、データ ファイルが削除されないよう、`IN DICTIONARY` を使用してテーブルを削除します。

```
DROP TABLE xyz in dictionary
```

最後に、再びテーブル `xyz` を作成します。

```
CREATE TABLE xyz (c1 int, c2 char(5))
```

テーブル xyz とデータ ファイル xyz000.mkd が作成されます。

関連項目

[ALTER TABLE](#)

[DROP TABLE](#)

[CREATE INDEX](#)

[DEFAULT](#)

[SET DEFAULTCOLLATE](#)

CREATE（テンポラリ）TABLE

CREATE TABLE ステートメントを使用して、テンポラリ テーブルを作成することもできます。しかし、テンポラリ テーブルのための CREATE TABLE 構文は永続テーブルのためのものより限定的です。この理由により、またその他の特性があることから、テンポラリ テーブルについては別途説明します。[ほかの特性](#)を参照してください。

構文

CREATE TABLE <# | ##> テーブル名 (テーブル要素 [, テーブル要素]...)

テーブル名 ::= ユーザー定義名

テーブル要素 ::= 列定義 | テーブル制約定義

列定義 ::= 列名 データ型 [**DEFAULT** デフォルト値の式] [列制約定義 [列制約定義]... [**CASE** (文字列) | **COLLATE** 照合順序名]

列名 ::= ユーザー定義名

データ型 ::= データ型名 [(桁数 [, 小数位])]

桁数 ::= 整数

小数位 ::= 整数

デフォルト値の式 ::= デフォルト値の式 + デフォルト値の式

| デフォルト値の式 - デフォルト値の式
| デフォルト値の式 * デフォルト値の式
| デフォルト値の式 / デフォルト値の式
| デフォルト値の式 & デフォルト値の式
| デフォルト値の式 | デフォルト値の式
| デフォルト値の式 ^ デフォルト値の式
| (デフォルト値の式)
| - デフォルト値の式
| + デフォルト値の式
| ~ デフォルト値の式
| ?
| リテラル
| スカラー関数
| { fn スカラー関数 }
| **USER**
| **NULL**

リテラル ::= ' 文字列 ' | **N** ' 文字列 '

| 数字
| { d ' 日付リテラル ' }
| { t ' 時刻リテラル ' }
| { ts ' タイムスタンプ リテラル ' }

列制約定義 ::= [**CONSTRAINT** 制約名] 列制約

制約名 ::= ユーザー定義名

```
列制約 ::= NOT NULL
        | NOT MODIFIABLE
        | UNIQUE
        | PRIMARY KEY
        | REFERENCES テーブル名 [ ( 列名 ) ] [ 参照アクション ]
```

```
テーブル制約定義 ::= [ CONSTRAINT 制約名 ] テーブル制約
```

```
テーブル制約 ::= UNIQUE ( 列名 [ , 列名 ]... )
        | PRIMARY KEY ( 列名 [ , 列名 ]... )
        | REFERENCES テーブル名 [ ( 列名 [ , 列名 ]... ) ] [ 参照アクション ]
```

```
参照アクション ::= 参照更新アクション [ 参照削除アクション ]
        | 参照削除アクション [ 参照更新アクション ]
```

```
参照更新アクション ::= ON UPDATE RESTRICT
```

```
参照削除アクション ::= ON DELETE CASCADE
        | ON DELETE RESTRICT
```

```
照合順序名 ::= ' 文字列 '
```

備考

テンポラリ テーブルは中間結果や作業領域として使用されます。永続テーブルとは異なり、テンポラリ テーブルのデータは SQL セッション中のある時点または SQL セッションの最後に破棄されます。データはデータベースに保存されません。

テンポラリ テーブルは、中間テーブルに対して処理を継続することにより、中間結果を絞り込むのに便利です。複雑なデータ操作は、単純な手順に分割することでより簡単になります。それぞれの手順では、前の手順の結果のテーブルに対して操作を行います。テンポラリ テーブルはベース テーブルです。つまり、データはそれ自体に含まれています。これをビューと対比してみると、ビューは他テーブルのデータの間接的な表現です。

Zen は以下の 2 つのタイプのテンポラリ テーブルをサポートします。

- ローカル
- グローバル

どちらのタイプもストアド プロシージャ内で使用できます。

テンポラリ テーブルの作成される場所と使用される場所を対比してテンポラリ テーブルの特性をまとめ、次の表に示します。特性は、テーブルが作成または使用される場所がストアド プロシージャの内部か外部かによって異なります。表の終わりにある脚注に注意事項が示されています。

メタデータのバージョン 1 とバージョン 2 では、テンポラリ テーブル名の許容サイズを除き、テーブルの特性は同じです。

テーブルの特性	ローカル テンポラリ テーブル		グローバル テンポラリ テーブル	
	SP 外部 ¹	SP 内部	SP 外部	SP 内部
テーブル名の最初の文字は # である必要がある (下の 以前のリリースとの互換性 を参照してください。)	はい	はい	いいえ	いいえ
テーブル名の最初の文字は ## である必要がある (下の 以前のリリースとの互換性 を参照してください。)	いいえ	いいえ	はい	はい
テーブルのコンテキストはテーブルを作成するデータベースと同じ	はい	はい	はい	はい
2 つ以上のセッションが同一名称のテーブルを作成可能 ²	はい	はい	いいえ	いいえ
メタデータ バージョン 1 の場合、テーブル名の最大長（長さには "#", "##", アンダースコア、および ID も含む）については、『 <i>Advanced Operations Guide</i> 』の 識別子の制限 を参照してください	はい ³	はい ⁴	はい ³	はい ⁴
メタデータ バージョン 2 の場合、テーブル名の最大長（長さには "#", "##", アンダースコア、および ID も含む）については、『 <i>Advanced Operations Guide</i> 』の 識別子の制限 を参照してください	はい ³	はい ⁴	はい ³	はい ⁴
別のデータベース内のテーブルには、そのテーブル名で修飾することによりアクセス可能	いいえ	いいえ	はい	はい
SELECT、INSERT、UPDATE、および DELETE ステートメントがテーブルで許可されている	はい	はい	はい	はい
ALTER TABLE および DROP TABLE ステートメントがテーブルで許可されている	はい	はい	はい	はい
テーブルにビューを作成できる	いいえ	いいえ	いいえ	いいえ
テーブルにユーザー定義関数を作成できる	いいえ	いいえ	いいえ	いいえ
テーブルにトリガーを作成できる	いいえ	いいえ	いいえ	いいえ
テーブルで権限の許可または取り消しができる	いいえ	いいえ	いいえ	いいえ

テーブルの特性	ローカル テンポ ラリ テーブル		グローバル テンポ ラリ テーブル	
	SP 外 部 ¹	SP 内部	SP 外 部	SP 内部
FOREIGN KEY 制約が CREATE TABLE ステートメントで使用できる ⁵	いいえ	いいえ	いいえ	いいえ
SELECT INTO ステートメントでテーブルにデータを入れられる	はい	はい	はい	はい
SELECT INTO ステートメントでテーブルを作成できる ⁶	はい	はい	はい	はい
ある SQL セッションで作成されたテーブルが、ほかの SQL セッションからアクセスできる	いいえ	いいえ	はい	はい
プロシージャで作成されたテーブルは、そのプロシージャの外でアクセス可能	N/A ⁹	いいえ	N/A ⁹	はい
最上位のプロシージャで作成されたテーブルは、ネストされたプロシージャでアクセス可能	N/A ⁹	いいえ	N/A ⁹	いいえ
再帰的ストアド プロシージャ内の CREATE TABLE ステートメントは再帰呼び出しでテーブル名エラーを返す	N/A ⁹	はい ⁷	N/A ⁹	はい ⁹
テーブルは明示的削除時に削除される	はい	はい	はい	はい
テーブルはそのテーブルが作成されたセッションの最後に削除される	はい	はい ⁸	はい	はい
テーブルはそのテーブルが作成されたプロシージャの最後に削除される	N/A ⁹	はい	N/A ⁹	いいえ
テーブルは別のセッションのトランザクションの最後に削除される	N/A ⁹	N/A ⁹	はい	はい

テーブルの特性	ローカル テンポラ リ テーブル		グローバル テンポ ラリ テーブル	
	SP 外 部 ¹	SP 内部	SP 外 部	SP 内部

¹ SP はストアド プロシージャを表します。

² データベース エンジン、ストアド プロシージャの名前とセッション固有の ID をユーザー定義名に自動的に付加することによって、テーブル名が確実に一意になるようにします。この機能はユーザーに透過的です。

³ テーブル名の長さの合計には、"#" または "##" に加え、アンダースコア+セッション ID も含められます。セッション ID はオペレーティング システムによって 8、9、または 10 バイトになります。『*Advanced Operations Guide*』の[識別子の制限](#)を参照してください。

⁴ テーブル名の長さの合計には、"#" または "##" に加え、アンダースコア+ストアド プロシージャの名前+アンダースコア+セッション ID も含められます。セッション ID はオペレーティング システムによって 8、9、または 10 バイトになります。『*Advanced Operations Guide*』の[識別子の制限](#)を参照してください。

⁵ 制約は警告を返しますがテーブルは作成されます。

⁶ テーブルは、1 つの SELECT INTO ステートメントによって作成されデータが入れられます。

⁷ テーブル名は、ストアド プロシージャが最初に実行されることから既に存在しています。

⁸ プロシージャの実行が終了する前にセッションの終了が発生した場合。

⁹ N/A は「適用外」を意味します。

以前のリリースとの互換性

PSQL v9 Service Pack 2 より前の Zen のリリースは、永続テーブルに # または ## で始まる名前を付けることを許可していました。PSQL v9 Service Pack 2 以降のリリースでは、永続テーブルに # または ## で始まる名前を付けることはできません。# または ## で始まるテーブルはテンポラリ テーブルであり、TEMPDB データベース内に作成されます。

お使いのバージョンより前のバージョンの Zen で作成された # または ## で始まる永続テーブルにアクセスしようとすると、「テーブルが見つからない」というエラーが返されます。

『*Zen User's Guide*』の[ステートメント区切り文字](#)も参照してください。

TEMPDB データベース

Zen のインストールは TEMPDB という名前のシステム データベースを作成します。TEMPDB はすべてのテンポラリ テーブルを保持します。TEMPDB データベースは削除しないでください。削除してしまうと、テンポラリ テーブルが作成できなくなります。

TEMPDB は Zen 製品のインストール ディレクトリに作成されます。『*Getting Started with Zen*』の[ファイルはどこにインストールされますか？](#)を参照してください。

インストール後に、TEMPDB の辞書ファイルとデータ ファイルの場所を変更することもできます。『*Zen User's Guide*』の[データベースのプロパティ](#)を参照してください。

注意！ TEMPDB は、データベース エンジンが排他的に使用するシステム データベースです。TEMPDB を永続テーブル、ビュー、ストアド プロシージャなどの格納場所として**使用しないでください**。

ローカル テンポラリ テーブルのテーブル名

データベース エンジンは、複数のセッションで作成されたテンポラリ テーブルを区別するため、ローカル テンポラリ テーブルの名前に自動的に情報を付け加えます。付加情報の長さは、オペレーティング システムによって異なります。

ローカル テンポラリ テーブルの名前は少なくとも 10 バイトになります。ただしこれは、ローカル テンポラリ テーブルを**作成する**ストアド プロシージャの数が 1296 を超えなければならないという条件付きです。10 バイトには # 文字も含まれます。1296 の制限は同一セッション内のストアド プロシージャに適用されます。

名前の最大長は 20 バイトで、# 文字、テーブル名、および追加情報を含みます。

トランザクション

グローバル テンポラリ テーブルは、明示的に削除されるか、またはテーブルが作成されたセッションの終了時に自動的に削除されます。トランザクション内で、テーブルを作成したのではないセッションがそのテーブルを使用している場合、トランザクション完了時に削除されます。

SELECT INTO

単一の **SELECT INTO** ステートメントを使用して、テンポラリ テーブルを作成しデータを入れることができます。たとえば、**SELECT * INTO #mytmptbl FROM Billing** は、**#mytmptbl** という名前のローカル テンポラリ テーブルを作成します (**#mytmptbl** は存在していないものとします)。テンポラリ テーブルには **Demodata** サンプル データベースの **Billing** テーブルと同じデータが含まれます。

SELECT INTO ステートメントが同じテンポラリ テーブル名を使用して 2 度目に実行されると、テンポラリ テーブルが既に存在するのでエラーが返されます。

SELECT INTO ステートメントは 2 つ以上のテーブルからテンポラリ テーブルを作成することができます。ただし、テンポラリ テーブルを作成する元の各テーブルの列名は重複していない必要があります。重複しているとエラーが返されます。

このエラーは、列名をテーブル名で修飾し、各列にエイリアスを指定すれば回避することができます。たとえば、テーブル **t1** と **t2** には両方とも **col1** と **col2** の 2 つの列があるとします。次のステートメント **SELECT t1.col1, t1.col2, t2.col1, t2.col2 INTO #mytmptbl FROM t1, t2** はエラーを返します。代わりに、次のようなステートメント **SELECT t1.col1 c1, t1.col2 c2, t2.col1 c3, t2.col2 c4 INTO #mytmptbl FROM t1, t2** を使用します。

SELECT INTO の制限

- ストアド プロシージャ内部で作成されたローカル テンポラリ テーブルは、そのストアド プロシージャのスコープ内にあります。ストアド プロシージャの実行後、ローカル テンポラリ テーブルは破棄されます。
- **UNION** および **UNION ALL** キーワードは、**SELECT INTO** ステートメントと共に使用することはできません。
- **SELECT INTO** ステートメントの結果を取得できるのは、1 つのテンポラリ テーブルだけです。単独の **SELECT INTO** ステートメントによって、データを選択 (**SELECT**) し、それを複数のテンポラリ テーブルに入れることはできません。

ストアド プロシージャのキャッシング

ローカルまたはグローバルのテンポラリ テーブルを参照するストアド プロシージャは、キャッシュ設定に関わらず**キャッシュされません**。[SET CACHED_PROCEDURES](#) および [SET PROCEDURES_CACHE](#) を参照してください。

テンポラリ テーブルの例

次の例では、#b_temp という名前のローカル テンポラリ テーブルを作成し、Demodata サンプル データベースの Billing テーブルのデータを入れます。

```
SELECT * INTO "#b_temp" FROM Billing
```

次の例では、ID、Dept_Name、Building_Name、Room_Number の列から成り、列 ID に基づく主キーを持つ ##tenurefac という名前のグローバル テンポラリ テーブルを作成します。

```
CREATE TABLE ##tenurefac  
(ID UBIGINT,  
Dept_Name CHAR(20) CASE,  
Building_Name CHAR(25) CASE,  
Room_Number UINTEGER,  
PRIMARY KEY(ID))
```

次の例では、テンポラリ テーブル ##tenurefac を変更し、列 Research_Grant_Amt を追加します。

```
ALTER TABLE ##tenurefac ADD Research_Grant_Amt DOUBLE
```

次の例では、テンポラリ テーブル ##tenurefac を削除します。

```
DROP TABLE ##tenurefac
```

次の例では、ストアド プロシージャ内で 2 つのテンポラリ テーブルを作成し、そのテーブルにデータを入れ、変数に値を割り当てます。その値はテンポラリ テーブルから選択されます。

メモ： SELECT INTO は、変数に値を割り当てるために使用するのであれば、ストアド プロシージャ内で使用できます。


```

CREATE PROCEDURE "p11"()
AS BEGIN
    DECLARE :val1_int INTEGER;
    DECLARE :val2_char VARCHAR(20);
    CREATE TABLE #t11 (col1 INT, col2 VARCHAR(20));
    CREATE TABLE #t12 (col1 INT, col2 VARCHAR(20));
    INSERT INTO #t11 VALUES (1,'t1 col2 text');
    INSERT INTO #t12 VALUES (2,'t2 col2 text');
    SELECT col1 INTO :val1_int FROM #t11 WHERE col1 = 1;
    SELECT col2 INTO :val2_char FROM #t12 WHERE col1 = 2;
    PRINT :val1_int;
    PRINT :val2_char;
    COMMIT;
END;
CALL P1()

```

次の例では、グローバル テンポラリ テーブル `##enroll_student_global_temp_tbl` を作成し、次にストアード プロシージャ `Enrollstudent` を作成します。呼び出されると、プロシージャは指定された Student ID、Class ID、および GPA (Grade Point Average) を基に、`##enroll_student_global_temp_tbl` にレコードを挿入します。SELECT でテンポラリ テーブル内のすべてのレコードを選択し、その結果を表示します。グローバル テンポラリ テーブルの名前の長さは、メタデータ バージョン 2 でのみ許容されます。

```

CREATE TABLE ##enroll_student_global_temp_tbl (student_id INTEGER, class_id INTEGER, GPA REAL);
CREATE PROCEDURE Enrollstudent(in :Stud_id integer, in :Class_Id integer, IN :GPA REAL);
BEGIN
    INSERT INTO ##enroll_student_global_temp_tbl VALUES(:Stud_id, :Class_id, :GPA);
END;
CALL Enrollstudent(1023456781, 146, 3.2)
SELECT * FROM ##enroll_student_global_temp_tbl

```

次の例では、ストアード プロシージャ内で 2 つのテンポラリ テーブルを作成し、そのテーブルにデータを入れ、変数に値を割り当てます。その値はテンポラリ テーブルから選択されます。

```

CREATE PROCEDURE "p11"()
AS BEGIN
    DECLARE :val1_int INTEGER;
    DECLARE :val2_char VARCHAR(20);
    CREATE TABLE #t11 (col1 INT, col2 VARCHAR(20));
    CREATE TABLE #t12 (col1 INT, col2 VARCHAR(20));
    INSERT INTO #t11 VALUES (1,'t1 col2 text');
    INSERT INTO #t12 VALUES (2,'t2 col2 text');
    SELECT col1 INTO :val1_int FROM #t11 WHERE col1 = 1;
    SELECT col2 INTO :val2_char FROM #t12 WHERE col1 = 2;
    PRINT :val1_int;
    PRINT :val2_char;
    COMMIT;
END;
CALL P11()

```

関連項目

[ALTER TABLE](#)

[DROP TABLE](#)

[SELECT \(INTO 付き\)](#)

CREATE TRIGGER

CREATE TRIGGER ステートメントにより、データベース内に新しいトリガーを作成します。トリガーとは、**INSERT**、**UPDATE**、または **DELETE** によってテーブルデータが変更されるときに自動的に実行されるストアド プロシージャの一種です。

通常のストアド プロシージャとは違って、トリガーは直接実行できず、パラメーターを持つこともできません。また、トリガーは結果セットを返さず、トリガーをビューに定義することもできません。

構文

```
CREATE TRIGGER トリガー名 before/after 挿入 / 更新 / 削除 ON テーブル名
[ ORDER 番号 ]
[ REFERENCING 参照エイリアス ] FOR EACH ROW
[ WHEN プロシージャ検索条件 ] プロシージャ ステートメント
```

トリガー名 ::= ユーザー定義名

before/after ::= BEFORE | AFTER

挿入 / 更新 / 削除 ::= INSERT | UPDATE | DELETE

参照エイリアス ::= OLD [AS] 相関名 [NEW [AS] 相関名]
| NEW [AS] 相関名 [OLD [AS] 相関名]

相関名 ::= ユーザー定義名

備考

メモ：トリガー内では、変数名はコロン (:) で始まる必要があります。

OLD (OLD 相関名) と NEW (NEW 相関名) はトリガー内部で使用でき、通常のストアド プロシージャでは使用できません。

DELETE または UPDATE トリガーでは、文字 "OLD" または OLD 相関名を列名の前に付加して、更新または削除操作の前にデータ行内の列を参照する必要があります。

INSERT または UPDATE トリガーでは、文字 "NEW" または NEW 相関名を列名の前に付加して、挿入または更新しようとする行内の列を参照する必要があります。

トリガー名は辞書内で固有の名前でなければなりません。

トリガーはその種類によって、**UPDATE**、**INSERT**、または **DELETE** ステートメントの実行の前か後に実行されます。

メモ： CREATE TRIGGER ステートメントには、CREATE PROCEDURE と同様の長さやそれ以外の制約があります。詳細については、[制限](#)および[データ型の制限](#)を参照してください。

例

次の例では、Tuition テーブルに挿入された新規の値を TuitionIDTable に記録するトリガーが作成されます。

```
CREATE TABLE Tuitionidtable (PRIMARY KEY(id), id UBIGINT);
CREATE TRIGGER InsTrig
BEFORE INSERT ON Tuition
REFERENCING NEW AS Indata
FOR EACH ROW
INSERT INTO Tuitionidtable VALUES(Indata.ID);
```

Tuition の INSERT によりトリガーが呼び出されます。

次の例では、トリガーを使って A と B の 2 つのテーブルの同期をとる方法を示します。両方のテーブルは同じ構造体を持ちます。

```
CREATE TABLE A (col1 INTEGER, col2 CHAR(10));
CREATE TABLE B (col1 INTEGER, col2 CHAR(10));
CREATE TRIGGER MyInsert
AFTER INSERT ON A FOR EACH ROW
INSERT INTO B VALUES (NEW.col1, NEW.col2);
CREATE TRIGGER MyDelete
AFTER DELETE ON A FOR EACH ROW
DELETE FROM B WHERE B.col1 = OLD.col1 AND B.col2 = OLD.col2;
CREATE TRIGGER MyUpdate
AFTER UPDATE ON A FOR EACH ROW
UPDATE B SET col1 = NEW.col1, col2 = NEW.col2 WHERE B.col1 = OLD.col1 AND B.col2 = OLD.col2;
```

この例の OLD と NEW は、テーブル A が位置付けでない SQL ステートメントで変更された場合のみテーブルの同期を保つので注意してください。SQLSetPos API または位置付けの UPDATE か DELETE を使用した場合は、テーブル A が重複するレコードを含まない場合のみテーブルの同期が保たれます。一方のレコードは変更するが、もう一方の重複するレコードは変更しないままにしておく SQL ステートメントは作成できません。

関連項目

[DROP TRIGGER](#)

CREATE USER

CREATE USER ステートメントにより、データベースに新しいユーザー アカウントが作成します。

この関数は、パスワード付き、パスワードなし、また、グループのメンバーとしてのユーザー アカウントをデータベースに作成するのに使用できます。

構文

```
CREATE USER ユーザー名 [ WITH PASSWORD ユーザー パスワード ][ IN GROUP 参照エイリアス ]
```

備考

このステートメントは、Zen Control Center (ZenCC) を使用して作成したユーザーと同じ権限を持つユーザーを作成します。たとえば、作成されたユーザーは、そのユーザーが **Master** としてログインしていない場合でも、デフォルトでデータベースを作成することを制限されません。

Master ユーザーのみがこのステートメントを実行できます。

このステートメントを実行するには、セキュリティ設定が有効になっている必要があります。

ユーザー名および**ユーザー パスワード**は Zen データベースのみを対象としたもので、オペレーティング システム レベルで設定されているユーザー名とパスワードには無関係です。Zen ユーザー名、グループ、およびパスワードは Zen Control Center (ZenCC) を使用して設定することもできます。

ユーザー名は辞書内で重複しない名前であればなりません。

ユーザー名とパスワードに空白またはその他の非英数文字が含まれる場合は、ユーザー名とパスワードを二重引用符で囲む必要があります。

ユーザーをグループのメンバーとして作成する場合は、ユーザーを作成する前にグループを設定する必要があります。

ユーザーとグループの詳細については、『*Advanced Operations Guide*』の **Master ユーザー、ユーザーとグループ**、および『*Zen User's Guide*』の**権限の割り当て作業**を参照してください。

例

以下の例では、ログイン権限を持たず、どのグループのメンバーでもない新規ユーザーアカウントを作成する方法を示します。

```
CREATE USER pgranger
```

新しいユーザーの名前は *pgranger* です。ユーザー パスワードは `NULL` で、このユーザー アカウントはどのグループのメンバーでもありません。

```
CREATE USER "polly granger"
```

新しいユーザーの名前は *polly granger* で、非英数文字を含んでいます。ユーザー パスワードは `NULL` で、このユーザー アカウントはどのグループのメンバーでもありません。

以下の例では、ログイン権限を持ち、どのグループのメンバーでもない新規ユーザーアカウントを作成する方法を示します。

```
CREATE USER pgranger WITH PASSWORD Prvsve1
```

新しいユーザーの名前は *pgranger* です。ユーザー パスワードは *Prvsve1*（大文字と小文字を区別）です。

```
CREATE USER pgranger WITH PASSWORD "Nonalfa$"
```

新しいユーザーの名前は *pgranger* です。ユーザー パスワードは *Nonalfa\$*（大文字と小文字を区別）で、非英数文字を含んでいます。

以下の例では、ログイン権限を持たず、グループのメンバーとして新規ユーザーを作成する方法を示します。

```
CREATE USER pgranger IN GROUP developers
```

新しいユーザーの名前は *pgranger* です。新しいユーザー アカウントは、グループ *developers* に割り当てられます。

以下の例では、ログイン権限を持ち、グループのメンバーとして新規ユーザーを作成する方法を示します。

```
CREATE USER pgranger WITH PASSWORD Prvsve1 IN GROUP developers
```

新しいユーザーの名前は *pgranger* です。新しいユーザー アカウントは、グループ *developers* に割り当てられ、大文字小文字を区別するパスワード *Prvsvel* を持ちます。この構文の順序 (`CREATE USER...WITH PASSWORD...IN GROUP`) は非常に重要です。

関連項目

[ALTER USER](#)、[DROP USER](#)、[GRANT](#)

CREATE VIEW

CREATE VIEW ステートメントを使用して、ストアド ビューまたは仮想テーブルを作成します。

構文

```
CREATE VIEW ビュー名 [ ( 列名 [ , 列名 ]... ) ]  
[ WITH EXECUTE AS 'MASTER' ] AS クエリ スペック  
[ ORDER BY order-by 式 [ , order-by 式 ]... ]
```

ビュー名 ::= ユーザー定義名

列名 ::= ユーザー定義名

order-by 式 ::= 式 [CASE (文字列) | COLLATE 照合順序名] [ASC | DESC] (SELECT 構文を参照)

備考

ビューは一種のデータベース オブジェクトで、クエリを格納し、テーブルのように振る舞います。ビューが返すデータは、SELECT ステートメントにより参照される、1 つ以上のテーブルに格納されます。ビュー内の行と列は、参照されるたびにリフレッシュされます。

ビュー名の最大長については、『*Advanced Operations Guide*』の[識別子の制限](#)を参照してください。ビュー内の列の最大数は 256 です。ビュー定義には 64 KB の制限があります。

Zen はグループ化ビューをサポートしています。グループ化ビューとは、SELECT ステートメントで次のいずれかを使用しているビューと定義されます。

- [DISTINCT](#)
- [GROUP BY](#)
- [ORDER BY](#)
- [スカラー関数](#)
- [スカラー サブクエリ](#)
- [TOP または LIMIT](#)
- [UNION](#)

サブクエリが式であれば、サブクエリ内でグループ化ビューを使用できます。サブクエリは、演算子 **IN**、**EXISTS**、**ALL**、または **ANY** と関連付けられている場合は、式と**見なされません**。

ビュー定義にプロシージャを含めることはできません。

ORDER BY

ビューにおける **ORDER BY** は、**SELECT** ステートメントにおける場合と同じように動作します。以下の点に特に留意してください。

- **ORDER BY** 句ではエイリアスを使用できます。
- **ORDER BY** 句ではスカラー サブクエリを使用できます。
- **TOP** や **LIMIT** は、**ORDER BY** を使用しているビューでを使用することをお勧めします。
- エンジンが、**ORDER BY** によって順序付けされた結果を返すためにテンポラリ テーブルを使用し、クエリが動的カーソルを使用する場合、そのカーソルは静的に変換されます。たとえば、インデックス付けされていない列に対して **ORDER BY** が使用される場合には、常にテンポラリ テーブルが必要となります。前方のみのカーソルと静的カーソルは影響を受けません。

信頼されるビューと信頼されないビュー

信頼されるビューには **WITH EXECUTE AS 'MASTER'** を含めます。[信頼されるオブジェクトと信頼されないオブジェクト](#)を参照してください。

信頼されるビューと信頼されないビューの例

次のステートメントによって、大学に在籍している全員の電話番号リストを作成する、**vw_Person** という名前の信頼されないビューが作成されます。このビューには姓、名および電話番号が表示され、列ごとにヘッダーが付きます。**Person** テーブルは **Demodata** サンプル データベースの一部です。

```
CREATE VIEW vw_Person (lastn,firstn,phone) AS SELECT Last_Name, First_Name,Phone FROM Person
```

このビューに対する以降のクエリでは、**SELECT** ステートメントで列見出しを使用できます。

```
SELECT lastn, firstn FROM vw_Person
```

ビューを実行するユーザーは、**Person** テーブルの **SELECT** 権限を持っている必要があります。

次の例では類似したビューが作成されますが、これは信頼されるビューです。

```
CREATE VIEW vw_trusted_Person (lastn,firstn,phone) WITH EXECUTE AS 'MASTER' AS SELECT Last_Name,
First_Name,Phone FROM Person
```

user1 には、**vw_Person** の **SELECT** 権限が付与されているものとします。**user1** は次のように **SELECT** ステートメントで列見出しを使用できます。

```
SELECT lastn, firstn FROM vw_trusted_Person
```

user1 は **Person** テーブルの **SELECT** 権限を持っている必要はありません。権限は信頼されるビューに付与されています。

次のステートメントによって、大学に在籍している全員の電話番号リストを作成する、**vw_Person** という名前のビューが作成されます。このビューには姓、名および電話番号が表示され、列ごとにヘッダーが付きます。**Person** テーブルは **Demodata** サンプル データベースの一部です。

```
CREATE VIEW vw_Person (lastn, firstn, telephone) AS SELECT Last_Name, First_Name, Phone FROM Person
```

このビューに対する以降のクエリでは、次の例に示すように、**SELECT** ステートメントで列見出しを使用できます。

```
SELECT lastn, firstn FROM vw_Person
```

前の例に **ORDER BY** 句を含めるように変更できます。

```
CREATE VIEW vw_Person_ordby (lastn, firstn, telephone) AS SELECT Last_Name, First_Name, Phone FROM
Person ORDER BY phone
```

ビューは次のような結果を返します（簡潔にするため、すべてのレコードは表示されていません）。

Last_Name	First_Name	Phone
Vqyles	Rex	2105551871
Qulizada	Ahmad	2105552233
Ragadio	Ernest	2105554654
Luckey	Anthony	2105557628

次の例では、生徒の成績評価点平均（GPA）を降順で返し、さらに、その GPA ごとに生徒を姓の昇順で列挙するビューを作成します。

```
CREATE VIEW vw_gpa AS SELECT Last_Name, Left(First_Name, 1) AS First_Initial, Cumulative_GPA AS GPA FROM
Person LEFT OUTER JOIN Student ON Person.ID=Student.ID ORDER BY Cumulative_GPA DESC, Last_Name
```

ビューは次のような結果を返します（簡潔にするため、すべてのレコードは表示されていません）。

Last_Name	First_Initial	GPA
Abuali	I	4.000
Adachi	K	4.000
Badia	S	4.000
Rowan	A	4.000
Ujazdowski	T	4.000
Wotanowski	H	4.000
Gnat	M	3.998
Titus	A	3.998
Mugaas	M	3.995

この例では、Person テーブルの先頭 10 レコードを ID 順で返すビューが作成されます。

```
CREATE VIEW vw_top10 AS SELECT TOP 10 * FROM person ORDER BY id;
```

ビューは次のような結果を返します（簡潔にするため、すべての列は表示されていません）。

ID	First_Name	Last_Name
100062607	Janis	Nipart
100285859	Lisa	Tumbleson
100371731	Robert	Mazza
100592056	Andrew	Sugar
100647633	Robert	Reagen
100822381	Roosevelt	Bora
101042707	Avram	Japadjief

10 行が影響を受けました。

次の例では、ORDER BY を UNION と一緒に使用できることを示すためのビューを作成します。

```
CREATE VIEW vw_union_ordby_desc AS SELECT first_name FROM person UNION SELECT last_name FROM PERSON
ORDER BY first_name DESC
```

ビューは次のような結果を返します（簡潔にするため、すべてのレコードは表示されていません）。

First_Name
Zyrowski
Zynda
Zydanowicz
Yzaguirre
Yyounce
Xystros
Xyois

Xu
Wyont
Wynalda
Wykes

関連項目

[DROP VIEW](#)

[SELECT](#)

[SET ROWCOUNT](#)

[信頼されるオブジェクトと信頼されないオブジェクト](#)

DECLARE

備考

DECLARE ステートメントを使用して、SQL 変数を定義します。

このステートメントは、ストアド プロシージャ、ユーザー定義関数、およびトリガー内でのみ使用できます。

変数名はコロン (:) またはアットマーク (@) で始まる必要があります。これは、変数やパラメーターを定義するときと使用するときの両方に当てはまります。SET を使って変数に値を設定するには、まず変数が宣言されていなければなりません。

各変数について、別々の DECLARE ステートメントを使用します（単一のステートメントで複数の変数は宣言できません）。CHAR、DECIMAL、NUMERIC および VARCHAR など、サイズや有効桁数、小数部桁数が必要なデータ型には、それらの値を指定します。

例

次の例は、変数の宣言方法を、サイズ、有効桁数、または小数部桁数が必要な変数の宣言も含めて示しています。

```
DECLARE :SaleItem CHAR(15);
DECLARE :CruiseLine CHAR(25) DEFAULT 'Open Seas Tours'
DECLARE :UnitWeight DECIMAL(10,3);
DECLARE :Titration NUMERIC(12,3);
DECLARE :ReasonForReturn VARCHAR(200);
DECLARE :Counter INTEGER = 0;
DECLARE :CurrentCapacity INTEGER = 9;
DECLARE :Cust_ID UNIQUEIDENTIFIER = NEWID()
DECLARE :ISO_ID UNIQUEIDENTIFIER = '1129619D-772C-AAAB-B221-00FF00FF0099'
```

関連項目

[CREATE FUNCTION](#)

[CREATE PROCEDURE](#)

[CREATE TRIGGER](#)

[SET](#)

DECLARE CURSOR

DECLARE CURSOR ステートメントにより、SQL カーソルを定義します。

構文

```
DECLARE カーソル名 CURSOR FOR 選択ステートメント [ FOR UPDATE | FOR READ ONLY ]
```

```
カーソル名 ::= ユーザー定義名
```

備考

DECLARE では、カーソルと変数はストアド プロシージャとトリガー内部でのみ許可されるため、このステートメントはストアド プロシージャまたはトリガー内部でのみ使用できます。

カーソルのデフォルト動作は読み取り専用です。したがって、更新（書き込みまたは削除）を明示的に示すには **FOR UPDATE** を使用する必要があります。

例

次の例では、Tuition テーブルの Degree、Residency、および Cost_Per_Credit 列から値を選択し、それらを ID 番号順に並べるカーソルが作成されます。

```
DECLARE BTUCursor CURSOR
FOR SELECT Degree, Residency, Cost_Per_Credit
FROM Tuition
ORDER BY ID;
```

次の例では、削除を実行するために **FOR UPDATE** が使用されています。

```
CREATE PROCEDURE MyProc(IN :CourseName CHAR(7)) AS
BEGIN
DECLARE c1 CURSOR FOR SELECT name FROM course WHERE name = :CourseName FOR UPDATE;
OPEN c1;
FETCH NEXT FROM c1 INTO :CourseName;
DELETE WHERE CURRENT OF c1;
CLOSE c1;
END;
CALL MyProc('HIS 305')
```

```
DECLARE cursor1 CURSOR
FOR SELECT Degree, Residency, Cost_Per_Credit
FROM Tuition ORDER BY ID
FOR UPDATE;
```

関連項目

[CREATE PROCEDURE](#)、[CREATE TRIGGER](#)

DEFAULT

テーブルには、ヌル値を許可しない列がある場合があります。エラーを回避するために、新しいレコードのそれらの列が必ず有効な値を持つよう、列のデフォルト値を定義しておくことができます。これを行うには、列定義の `CREATE` または `ALTER` で `DEFAULT` キーワードを使用して、`INSERT` および `UPDATE` ステートメントで使用される列値を設定します。この値は、次の一方または両方に当てはまる場合に使用されます。

- 明確な列値が指定されていない。
- 列ではバイナリ ゼロを使用できず、ヌル以外の有効な値を必要とする。

デフォルトの列値を定義すると、行を挿入または更新するときに `VALUES` 句で `DEFAULT` キーワードを使用すれば、その値を指定することができます。

要約すると、`DEFAULT` キーワードは次の場合に使用することができます。

- `CREATE TABLE` の列定義
- `ALTER TABLE` の列定義
- `INSERT` の `VALUES` 句
- `UPDATE` の `VALUES` 句

デフォルト値はリテラルまたは式です。`CREATE TABLE` または `ALTER TABLE` ステートメントでは、次の条件を満たしている必要があります。

- 列のデータ型と一致している
- 列に設定されている範囲や長さなどの制約に従っている

`INSERT` および `UPDATE` ステートメントでは、`DEFAULT` 式が定義されている列については、`Zen` が式を評価し、その結果を挿入または更新時に書き込みます。`INSERT` の場合、既存レコードの列は変更されないことに留意してください。

構文

`DEFAULT` キーワードの使用については、次のステートメントの構文を参照してください。

- [ALTER TABLE](#)
- [CREATE TABLE](#)

- [CREATE FUNCTION](#)
- [CREATE PROCEDURE](#)
- [INSERT](#)
- [UPDATE](#)

備考

以下のトピックでは、DEFAULT としてのリテラル値または式について説明します。

- [IDENTITY データ型の制限](#)
- [列のデフォルト値としてのスカラー関数および単純な式](#)
- [ALTER TABLE での DEFAULT の使用](#)

IDENTITY データ型の制限

CREATE TABLE ステートメントまたは ALTER TABLE ステートメントでは、IDENTITY、SMALLIDENTITY、BIGIDENTITY データ型にデフォルト値ゼロ (DEFAULT 0 または DEFAULT '0') を設定することができます。これらのデータ型では、それ以外のデフォルト値は許可されません。

列のデフォルト値としてのスカラー関数および単純な式

Zen では、リテラル値やヌル値のほかに、列と同じデータ型の値を返しさえすれば、スカラー関数や単純な式を使用して DEFAULT 値を定義することができます。たとえば、CHAR や VARCHAR の文字列型の列に USER() を使用することができます。新しい INSERT ステートメントおよび UPDATE ステートメントでそれが使用されたときに、戻り値が評価されます。

次の例は、DEFAULT として単純な式やさまざまなスカラー関数を使用する方法を示します。

```
CREATE TABLE DEFAULTSAMPLE (
  a INT DEFAULT 100,
  b DOUBLE DEFAULT PI(),
  c DATETIME DEFAULT NOW(),
  d CHAR(20) DEFAULT USER(),
  e VARCHAR(10) DEFAULT UPPER(DAYNAME(CURDATE())),
  f VARCHAR(40) DEFAULT '{'+CONVERT(NEWID(),SQL_VARCHAR)+'}',
  g INTEGER DEFAULT DAYOFYEAR(NOW())
);
INSERT INTO DEFAULTSAMPLE DEFAULT VALUES;
SELECT * FROM TX;
```

a	b	c	d
100	3.141592653589793	2023-10-27 10:10:10	zen

=====	=====	=====	=====
100	3.141592653589793	2021-11-12 17:25:59.288	Master
e	f	g	
=====	=====	=====	
FRIDAY	{EF66F711-36D2-40FA-B928-BAFCC486DA6B}	316	

ALTER TABLE での DEFAULT の使用

ALTER TABLE を使用して列定義を変更し、DEFAULT 属性を追加する場合、ALTER TABLE は既存の列定義に追加するのではなく、それを置き換えるため、既に定義されている列属性を含める必要があります。また、次の例に示すように、列を変更して DEFAULT 値を変えても、既存の値には影響しません。

```
CREATE TABLE Tab10 (a INT, b CHAR(20) CASE); -- 列 b は大文字小文字を無視します
ALTER TABLE Tab10 ALTER COLUMN b CHAR(20) DEFAULT 'xyz'; -- 列 b はデフォルト値を持ちますが、大文字小文字
を無視しなくなりました
INSERT INTO Tab10 DEFAULT VALUES;
SELECT * FROM Tab10;
=====
      a      b
-----
(Null)    xyz
1 行が影響を受けました。
SELECT * FROM Tab10 WHERE b = 'XYZ'; -- 大文字小文字が一致しないので、行は返されません
      a      b
=====
0 行が影響を受けました。
ALTER TABLE Tab10 ALTER COLUMN b CHAR(20) CASE DEFAULT 'xyz'; -- 列 b は大文字小文字を無視し、「かつ」デ
フォルト値を持つようになりました
SELECT * FROM Tab10 WHERE b = 'XYZ'; -- 大文字小文字が一致する必要はありません
      a      b
=====
(Null)    xyz
1 行が影響を受けました。
```

例

次のステートメントはテーブル Tab5 を作成します。列 col5 のデフォルト値は 200 とします。

```
CREATE TABLE Tab5
(col5 INT DEFAULT 200)
```

次のステートメントはテーブル Tab1 を作成し、列 col1 は NOW() から返された結果の DATE 部分とします。

```
CREATE TABLE Tab1
(col1 DATE DEFAULT NOW())
```

次のステートメントはテーブル **Tab8** を作成し、列 **col8** は、INSERT または UPDATE ステートメントの実行時間とします。

```
CREATE TABLE Tab8
(col8 TIMESTAMP DEFAULT CURRENT_TIMESTAMP())
```

次のステートメントはテーブル **Tab6** を作成し、列 **col6** は、INSERT または UPDATE ステートメント実行後のユーザー名とします。**DEFAULT USER** を実際に利用できるのは、セキュリティが有効になっている場合のみです。有効になっていない場合、**USER** は常に **NULL** です。キーワード **USER** は **USER()** 関数と同じ結果をもたらします。

```
CREATE TABLE Tab6
(col6 VARCHAR(20) DEFAULT USER)
```

次のステートメントは**無効な**例を示しています。**TIME** は **DATE** 列で許可されているデータ型ではないため、解析時エラーになります。

```
CREATE TABLE Tab
(col DATE DEFAULT CURTIME())
```

次のステートメントは**無効な**例を示しています。**'3.1'** は数値に変換できますが、それは有効な整数ではないため、解析時エラーになります。

```
CREATE TABLE Tab
(col SMALLINT DEFAULT '3.1')
```

次のステートメントは**無効な**例を示しています。**CREATE TABLE** ステートメントは成功しますが、**-60000** は **SMALLINT** でサポートされる値の範囲外であるため、**INSERT** ステートメントは失敗します。

```
CREATE TABLE Tab
(col SMALLINT DEFAULT 3 * -20000)
INSERT INTO Tab values(DEFAULT)
```

次のステートメントは、**IDENTITY** および **SMALLIDENTITY** データ型にデフォルト値 **ゼロ** を設定する**正しい**例を示しています。

```
CREATE TABLE t1 ( c1 IDENTITY DEFAULT '0' )
ALTER TABLE t1 ALTER c1 SMALLIDENTITY DEFAULT 0
```

次のステートメントは、IDENTITY および SMALLIDENTITY データ型にデフォルト値を設定する**無効な**例を示しています。

```
CREATE TABLE t1 ( c1 IDENTITY DEFAULT 3 )  
ALTER TABLE t1 ALTER c1 SMALLIDENTITY DEFAULT 1
```

関連項目

[ALTER TABLE](#)

[CREATE TABLE](#)

[CREATE FUNCTION](#)

[CREATE PROCEDURE](#)

[INSERT](#)

[UPDATE](#)

DELETE（位置付け）

DELETE（位置付け）ステートメントを使用して、SQL カーソルに関連付けられたビューの現在行を削除します。

構文

DELETE WHERE CURRENT OF *カーソル名*

カーソル名 ::= *ユーザー定義名*

備考

このステートメントは、ストアド プロシージャ、トリガー、およびセッション レベルでのみ使用できます。

メモ： セッション レベルでは位置付け DELETE ステートメントは使用できますが、DECLARE CURSOR ステートメントは使用できません。アクティブな結果セットのカーソル名を取得する方法は、アプリケーションが使用している Zen のアクセス方法によって決まります。アクセス方法については、Zen のドキュメントを参照してください。

例

次の一連のステートメントは、位置付け DELETE ステートメントの設定を示します。位置付け DELETE ステートメントに必要なステートメントは、DECLARE CURSOR、OPEN CURSOR、および FETCH FROM *カーソル名* です。

現代ヨーロッパ史の授業が時間割から削除されたので、この例ではサンプルデータベースの Course テーブルから Modern European History（HIS 305）の行が削除されます。

```
CREATE PROCEDURE DropClass();
DECLARE :CourseName CHAR(7);
DECLARE c1 CURSOR
FOR SELECT name FROM COURSE WHERE name = :CourseName;
BEGIN
SET :CourseName = 'HIS 305';
OPEN c1;
FETCH NEXT FROM c1 INTO :CourseName;
DELETE WHERE CURRENT OF c1;
END;
```

関連項目

[CREATE PROCEDURE](#)

[CREATE TRIGGER](#)

DELETE

このステートメントにより、データベース テーブルまたはビューから指定された行を削除します。

構文

```
DELETE [ FROM ] < テーブル名 | ビュー名 > [エイリアス名 ]
[ FROM テーブル参照 [ , テーブル参照 ]...
[ WHERE 検索条件 ]
```

テーブル名 ::= ユーザー定義名

ビュー名 ::= ユーザー定義名

エイリアス名 ::= ユーザー定義名 (エイリアス名は、2 つ目の FROM 句が使用されている場合には使用できません。
FROM 句を参照してください。)

```
テーブル参照 ::= { OJ 外部結合定義 }
| [ データベース名 . ] テーブル名 [ [ AS ] エイリアス名 ]
| [ データベース名 . ] ビュー名 [ [ AS ] エイリアス名 ]
| 結合定義
| ( 結合定義 )
| ( テーブルサブクエリ ) [ AS ] エイリアス名 [ ( 列名 [ , 列名 ]... ) ]
```

外部結合定義 ::= テーブル参照 外部結合タイプ JOIN テーブル参照 ON 検索条件

外部結合タイプ ::= LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER]

```
検索条件 ::= 検索条件 AND 検索条件
| 検索条件 OR 検索条件
| NOT 検索条件
| ( 検索条件 )
| 述部
```

データベース名 ::= ユーザー定義名

ビュー名 ::= ユーザー定義名

```
結合定義 ::= テーブル参照 [ 結合タイプ ] JOIN テーブル参照 ON 検索条件
| テーブル参照 CROSS JOIN テーブル参照
| 外部結合定義
```

結合タイプ ::= INNER | LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER]

```
テーブル サブクエリ ::= クエリ スペック [ [ UNION [ ALL ]
クエリ スペック ]... ]
```

備考

DELETE ステートメントは、INSERT および UPDATE と同様にアトミックな方法で動作します。つまり、複数の行の削除に失敗した場合、同じステートメントによって実行された前の行の削除がすべてロールバックされます。

FROM 句

オプションの第 2 FROM 句と、行が削除されるテーブル（「削除テーブル」と呼びます）への参照に関して混乱が生じる可能性があります。第 2 FROM 句に削除テーブルが現れる場合、その出現のうちの 1 つは、行が削除されるテーブルと同じインスタンスになります。

たとえば、ステートメント `DELETE t1 FROM t1, t2 WHERE t1.c1 = t2.c1` の場合、DELETE の直後の `t1` と FROM の後の `t1` は、テーブル `t1` の同じインスタンスです。したがって、このステートメントは `DELETE t1 FROM t2 WHERE t1.c1 = t2.c1` と同じことになります。

第 2 FROM 句に削除テーブルが複数回現れる場合、その出現のうちの 1 つは削除テーブルと同じインスタンスであると識別される必要があります。第 2 FROM 句の参照のうち、削除テーブルと同じインスタンスであると見なされるのは、エイリアスが指定されていない参照です。

したがって、ステートメント `DELETE t1 FROM t1 a, t1 b WHERE a.c1 = b.c1` の場合、第 2 FROM 句の `t1` のインスタンスはどちらもエイリアスを持っているため、これは正しくありません。次であれば有効です。 `DELETE t1 FROM t1, t1 b WHERE t1.c1 = b.c1`

第 2 FROM 句には次の条件が適用されます。

- DELETE ステートメントにオプションの第 2 FROM 句を含める場合、FROM 句の前にあるテーブル参照にエイリアスを指定することはできません。たとえば、`DELETE t1 a FROM t2 WHERE a.c1 = t2.c1` とすると、次のエラーが返されます。

```
SQL_ERROR (-1)
SQLSTATE "37000"
" オプションの FROM を伴う UPDATE/DELETE ステートメントでは、テーブル エイリアスは使用できません。 "
```

ステートメントの有効なバージョンは、`DELETE t1 FROM t2 WHERE t1.c1 = t2.c1` または `DELETE t1 FROM t1 a, t2 WHERE a.c1 = t2.c1` です。

- 第 2 FROM 句に削除テーブルへの参照を 2 つ以上含める場合、それらの参照のうちの 1 つにだけエイリアスを指定できます。たとえば、`DELETE t1 FROM t1 a, t1 b WHERE a.c1 = b.c1` とすると、次のエラーが返されます。


```
SQL_ERROR (-1)
SQLSTATE "37000"
" テーブル t1 があいまいです。 "
```

誤りのあるステートメントでは、エイリアスを "a" とするテーブル t1 が削除テーブルと同じインスタンスであると仮定しています。正しいステートメントは、`DELETE t1 FROM t1, t1 b WHERE t1.c1 = b.c1` です。

- DELETE ステートメントにおける第 2 FROM 句はセッション レベルでのみサポートされます。DELETE ステートメントがストアド プロシージャ内で発生する場合は、第 2 FROM 句はサポート **されません**。

例

次のステートメントによって、サンプル データベースの person テーブルから名前 Ellen の行が削除されます。

```
DELETE FROM person WHERE First_Name = 'Ellen'
```

次のステートメントによって、サンプル データベースの course テーブルから Modern European History (HIS 305) の行が削除されます。

```
DELETE FROM Course WHERE Name = 'HIS 305'
```

DISTINCT

DISTINCT キーワードを SELECT ステートメントに記述すると、結果から重複する値が削除されます。DISTINCT を使用することで、選択条件を満たす一意の行をすべて取得できます。

以下の規則が適用されます。

- Zen は、サブクエリでの DISTINCT の使用をサポートします。
- 選択リストに集計が含まれる場合、DISTINCT は無視されます。集計により、結果の行は重複しないことが既に保証されています。

例

次のステートメントは、教職員 ID 111191115 が担当するすべての講座を取得します。2 番目のステートメントでは DISTINCT を使用して、列の値が重複する行を取り除いています。

```
SELECT c.Name, c.Description
FROM Course c, class cl
WHERE c.name = cl.name AND cl.faculty_id = '111191115';
```

Name	Description
CHE 203	Chemical Concepts and Properties I
CHE 203	Chemical Concepts and Properties I
CHE 205	Chemical Concepts and Properties II
CHE 205	Chemical Concepts and Properties II

```
SELECT DISTINCT c.Name, c.Description
FROM Course c, class cl
WHERE c.name = cl.name AND cl.faculty_id = '111191115';
```

Name	Description
CHE 203	Chemical Concepts and Properties I
CHE 205	Chemical Concepts and Properties II

メモ： 次のような DISTINCT の使用法は許可されていません。

```
SELECT DISTINCT column1, DISTINCT column2
```

関連項目

SELECT

DISTINCT のその他の使用法については、[集計関数の DISTINCT](#) を参照してください。

DROP DATABASE

DROP DATABASE ステートメントにより、データベースを削除します。Master ユーザーのみがこのステートメントを発行できます。

構文

```
DROP DATABASE [ IF EXISTS ] データベース名 [ DELETE FILES ]
```

データベース名 ::= ユーザー定義名

備考

このステートメントを発行するには、Master ユーザーとしてデータベースにログオンしている必要があります。DROP DATABASE ステートメントを使用すると、セキュリティ設定で削除が許可されていれば、現在ログオンしているデータベースも含め、あらゆるデータベースを削除できます。以下の[セキュリティで保護されたデータベース](#)を参照してください。

DROP DATABASE は、defaultdb や tempdb などのシステム データベースを削除する場合には使用できません。選択し、セキュリティで許可されていれば、このステートメントを使用して最後のユーザー定義データベースを削除できます。

DROP DATABASE ステートメントは、ストアド プロシージャおよびユーザー定義関数内では使用できません。

IF EXISTS 式を指定すると、データベースが存在しない場合、ステートメントはエラーではなく成功を返すようになります。IF EXISTS は、それ以外のエラーを抑制しません。

セキュリティで保護されたデータベース

「データベース」セキュリティ モデルで保護されたデータベースは削除できません。以下の方法でセキュリティ保護されているデータベースは削除できます。

- クラシック セキュリティ
- 混合セキュリティ
- リレーショナル セキュリティ (Master パスワード) と、クラシック セキュリティまたは混合セキュリティの組み合わせ

詳細については、『*Advanced Operations Guide*』の [Zen セキュリティ](#) を参照してください。

DELETE FILES

DELETE FILES 句は、データ辞書ファイル（DDF）を削除するためのものです。データファイルは削除されません。

DELETE FILES を省略した場合、DDF は物理記憶域に残りますが、データベース名は `dbnames.cfg` から削除されます。`dbnames.cfg` から名前が削除されると、そのデータベースはもうデータベース エンジンにとって存在しないことになります。DDF を保持しておくことによって、データベースの再作成を選択した場合にその実行が可能になります。

DDF を削除するとき、DDF は使用中であってはいけないことに注意してください。たとえば、Zen Control Center を開いている場合に DELETE FILES 句を使用すると、"ファイルがロックされています" というエラーが返されます。ZenCC が開いている間は、DDF は使用中であると見なされ、削除されません。

例

次の例によって、データベース名 `inventorydb` は `dbnames.cfg` から削除されますが、データベースの DDF（およびデータ ファイル）は物理記憶域に保持されます。

```
DROP DATABASE inventorydb
```

次の例では、データベース名 `HRUSBenefits` とその DDF が削除されます。`HRUSBenefits` のデータ ファイルは保持されます。

```
DROP DATABASE HRUSBenefits DELETE FILES
```

関連項目

[CREATE DATABASE](#)

DROP FUNCTION

DROP FUNCTION ステートメントにより、データベースからスカラ ユーザー定義関数 (UDF) を削除します。

メモ： 存在しない UDF を削除しようとすると、エラー メッセージが表示されます。

構文

```
DROP FUNCTION [ IF EXISTS ] { 関数名 }
```

関数名 ::= 削除するユーザー定義関数の名前

備考

IF EXISTS 式を指定すると、関数が存在しない場合、ステートメントはエラーではなく成功を返すようになります。IF EXISTS は、それ以外のエラーを抑制しません。

例

次のステートメントによって、ユーザー定義関数 `fn_MyFunc` がデータベースから削除されます。

```
DROP FUNCTION fn_MyFunc
```

関連項目

[CREATE FUNCTION](#)

DROP GROUP

このステートメントは、セキュリティで保護されているデータベース内の 1 つまたは複数のグループを削除します。

構文

```
DROP GROUP [ IF EXISTS ] グループ名 [ , グループ名 ]...
```

備考

Master ユーザーのみがこのステートメントを実行できます。複数のグループはカンマで区切ります。グループを削除するには空である必要があります。

このステートメントを実行するには、セキュリティ設定が有効になっている必要があります。

IF EXISTS 式を指定すると、グループが存在しない場合、ステートメントはエラーではなく成功を返すようになります。IF EXISTS は、それ以外のエラーを抑制しません。

例

次の例ではグループ `zengroup` が削除されます。

```
DROP GROUP zengroup
```

次の例では、リストを使用してグループを削除します。

```
DROP GROUP zen_dev, zen_marketing
```

関連項目

[ALTER GROUP](#)

[CREATE GROUP](#)

DROP INDEX

このステートメントにより、指定されたテーブルから特定のインデックスを削除します。

構文

```
DROP INDEX [ IF EXISTS ] [ テーブル名 . ] インデックス名 [ IN DICTIONARY ]
```

テーブル名 ::= ユーザー定義名

インデックス名 ::= ユーザー定義名

備考

IN DICTIONARY は高度な機能です。絶対に必要な場合に、システム管理者のみが使用してください。IN DICTIONARY キーワードを使用すると、基となるデータ ファイルからはインデックスを削除せずに、DDF からインデックスを削除できます。通常、Zen は DDF とデータ ファイルの正確な同期を保ちますが、この機能によりユーザーは、同期していないテーブルの辞書定義を既存のデータ ファイルに合致させることが柔軟に行えるようになります。これは、既存のデータ ファイルと合致する新しい定義を辞書内に作成したい場合に有用です。

注意！ DDF の変更を、基となるデータ ファイルへの変更に合わせて行わないと、重大な問題が生じることがあります。

IF EXISTS 式を指定すると、インデックスが存在しない場合、ステートメントはエラーではなく成功を返すようになります。IF EXISTS は、それ以外のエラーを抑制しません。

この機能の詳細については、[IN DICTIONARY](#) の説明を参照してください。

部分インデックス

部分インデックスを削除する場合、PARTIAL 修飾子は必要ありません。

例

次のステートメントによって、指定された名前付きインデックスが Faculty テーブルから削除されます。

```
DROP INDEX Faculty.Dept
```

次の例では、データ ファイルと関連付けられない「デタッチされた」テーブルが作成され、その後でテーブル定義へのインデックスの追加と削除が行われます。インデックスは、関連付けられる基となる **Btrieve** インデックスが存在しないため、デタッチされたインデックスとなります。

```
CREATE TABLE t1 IN DICTIONARY (c1 int, c2 int)
CREATE INDEX idx_1 IN DICTIONARY on t1(c1)
DROP INDEX t1.idx_1 IN DICTIONARY
```

関連項目

[CREATE INDEX](#)

DROP PROCEDURE

このステートメントにより、現行データベースから 1 つまたは複数のストアド プロシージャを削除します。

構文

```
DROP PROCEDURE [ IF EXISTS ] プロシージャ名
```

備考

IF EXISTS 式を指定すると、プロシージャが存在しない場合、ステートメントはエラーではなく成功を返すようになります。IF EXISTS は、それ以外のエラーを抑制しません。

例

次のステートメントによって、ストアド プロシージャ `myproc` が辞書から削除されます。

```
DROP PROCEDURE myproc
```

関連項目

[CREATE PROCEDURE](#)

DROP TABLE

このステートメントにより、指定されたデータベースからテーブルを削除します。

構文

```
DROP TABLE [ IF EXISTS ] テーブル名 [ IN DICTIONARY ]
```

テーブル名 ::= 削除するテーブルのユーザー定義名

IN DICTIONARY

[ALTER TABLE](#) の IN DICTIONARY の解説を参照してください。

備考

[CASCADE](#) および [RESTRICT](#) はサポートされていません。

テーブルに依存しているトリガーがある場合、テーブルは削除されません。

トランザクションが進行中で、それがテーブルを参照している場合は、エラーが表示されてテーブルは削除されません。

削除するテーブルにほかのテーブルが依存している場合は、テーブルの削除に失敗します。

主キーが存在する場合は削除されます。テーブルを削除する前にユーザーが主キーを削除する必要はありません。テーブルの主キーが別のテーブルに属する制約によって参照されている場合、テーブルは削除されず、エラーが表示されます。

テーブルに外部キーがある場合、外部キーは削除されます。

テーブルにその他の制約（たとえば、NOT NULL、CHECK、UNIQUE、NOT MODIFIABLE など）がある場合、テーブルが削除されるとそれらの制約は削除されます。

IF EXISTS 式を指定すると、テーブルが存在しない場合、ステートメントはエラーではなく成功を返すようになります。IF EXISTS は、それ以外のエラーを抑制しません。

例

次のステートメントによって、class テーブル定義が辞書から削除されます。

```
DROP TABLE Class
```

関連項目

[ALTER TABLE](#)

[CREATE TABLE](#)

DROP TRIGGER

このステートメントにより、現行データベースからトリガーを削除します。

構文

```
DROP TRIGGER [ IF EXISTS ] トリガー名
```

備考

IF EXISTS 式を指定すると、トリガーが存在しない場合、ステートメントはエラーではなく成功を返すようになります。IF EXISTS は、それ以外のエラーを抑制しません。

例

次の例では、InsTrig という名前のトリガーが削除されます。

```
DROP TRIGGER InsTrig
```

関連項目

[CREATE TRIGGER](#)

DROP USER

DROP USER ステートメントは、データベースからユーザー アカウントを削除します。

構文

```
DROP USER [ IF EXISTS ] ユーザー名 [ , ユーザー名 ]...
```

備考

Master ユーザーのみがこのステートメントを実行できます。

このステートメントを実行するには、セキュリティ設定が有効になっている必要があります。

複数のユーザーはカンマで区切ります。

ユーザー名に空白やその他の非英数文字が含まれる場合は、ユーザー名を二重引用符で囲む必要があります。

ユーザー アカウントを削除しても、そのユーザーが作成したテーブルや、ビュー、その他のデータベース オブジェクトは削除されません。

IF EXISTS 式を指定すると、ユーザーが存在しない場合、ステートメントはエラーではなく成功を返すようになります。IF EXISTS は、それ以外のエラーを抑制しません。

ユーザーとグループの詳細については、『*Advanced Operations Guide*』の [Master ユーザー](#)、[ユーザーとグループ](#)、および『*Zen User's Guide*』の[権限の割り当て作業](#)を参照してください。

例

次の例では、ユーザー アカウント *pgranger* を削除します。

```
DROP USER pgranger
```

次の例では、複数のユーザー アカウントを削除します。

```
DROP USER pgranger, "lester pelling"
```

関連項目

[ALTER USER](#)

[CREATE USER](#)

DROP VIEW

このステートメントにより、データベースから指定されたビューを削除します。

構文

```
DROP VIEW [ IF EXISTS ] ビュー名
```

ビュー名 ::= ユーザー定義名

備考

[[CASCADE](#) | [RESTRICT](#)] はサポートされていません。

IF EXISTS 式を指定すると、ビューが存在しない場合、ステートメントはエラーではなく成功を返すようになります。IF EXISTS は、それ以外のエラーを抑制しません。

例

次のステートメントによって、vw_person ビュー定義が辞書から削除されます。

```
DROP VIEW vw_person
```

関連項目

[CREATE VIEW](#)

END

備考

BEGIN [ATOMIC] の解説を参照してください。

EXECUTE

EXECUTE ステートメントには 2 とおりの使い方があります。

- ユーザー定義のプロシージャまたはシステム ストアド プロシージャを呼び出す場合。CALL ステートメントの代わりに EXECUTE ステートメントを使用できます。
- ストアド プロシージャ内で、文字列または、文字列を返す式を実行する場合。

構文

ストアド プロシージャを呼び出す場合：

```
EXEC[UTE] ストアド プロシージャ [ ( [ プロシージャ パラメーター [ , プロシージャ パラメーター ]... ] ) ]
```

ストアド プロシージャ ::= ストアド プロシージャの名前

プロシージャ パラメーター ::= ストアド プロシージャが必要とする入力パラメーター

ユーザー定義のストアド プロシージャ内の場合：

```
EXEC[UTE] ( 文字列 [ + 文字列 ]... )
```

文字列 ::= 文字列、文字列変数、または文字列を返す式

備考

ストアド プロシージャ構文の EXEC[UTE] (文字列 ...) は、NCHAR 値のリテラルおよび変数をサポートしていません。文字列の構築に使用される値は、実行前に CHAR 値へ変換されます。

例

次の例では、パラメーターのないプロシージャを実行します。

```
EXEC NoParms() または CALL NoParms
```

次の例では、パラメーターのあるプロシージャを実行します。

```
EXEC Parms(vParm1, vParm2)
EXECUTE CheckMax(N.Class_ID)
```

次のプロシージャは、Billing テーブルから Student_ID を取り出します。

```
CREATE PROCEDURE tmpProc(IN :vTable CHAR(25)) RETURNS (sID INTEGER) AS
BEGIN
```

```
EXEC ('SELECT Student_ID FROM ' + :vtable);  
END;  
EXECUTE tmpProc('Billing')
```

関連項目

[CALL](#)

[CREATE PROCEDURE](#)

[システム ストアド プロシージャ](#)

EXISTS

EXISTS キーワードを使用して、サブクエリの結果に行があるかどうかを検査します。サブクエリに行が含まれていれば、**True** が返されます。

構文

EXISTS (サブクエリ)

備考

外部クエリが評価するすべての行について、Zen はサブクエリ内に関連する行が存在するかどうかを検査します。Zen は、外部クエリの結果のうち、サブクエリ内の関連する行に対応する各行を、ステートメントの結果テーブルに含めます。

EXISTS はストアド プロシージャ内のサブクエリにも使用できます。ただし、ストアド プロシージャ内のサブクエリ SELECT ステートメントに COMPUTE 句または INTO キーワードを含めることはできません。

多くの場合、EXISTS を含むサブクエリは IN を使って書き直すことができます。Zen は、IN を使用しているクエリの方がより効率的に処理できます。

例

次のステートメントは、成績評価点平均が 4.0 の人だけを含むリストを返します。

```
SELECT * FROM Person p WHERE EXISTS
(SELECT * FROM Enrolls e WHERE e.Student_ID = p.id
AND Grade = 4.0)
```

IN を使って、このステートメントを書き直すことができます。

```
SELECT * FROM Person p WHERE p.id IN
(SELECT e.Student_ID FROM Enrolls WHERE Grade = 4.0)
```

次のプロシージャは、入力パラメーターの値を使って、Person テーブルから ID を選択します。プロシージャの最初の実行 (EXEC) では、"Exists は true を返しました " が返ります。2 番目の EXEC では "Exists は false を返しました " が返ります。

```
CREATE PROCEDURE ex1(IN :vID INTEGER) RETURNS ( d1 VARCHAR(30) ) AS
BEGIN
  IF EXISTS (SELECT id FROM person WHERE id < :vID)
    THEN PRINT 'Exists は true を返しました ';
```

```
        ELSE PRINT 'Exists は false を返しました ';  
    END IF;  
END;  
EXEC ex1(222222222);  
EXEC ex1(1);
```

関連項目

[SELECT](#)

FETCH

構文

```
FETCH [ [NEXT] FROM ] カーソル名 INTO 変数名
```

カーソル名 ::= ユーザー定義名

備考

FETCH ステートメントにより、指定したテーブル行に SQL カーソルを位置付け、ターゲット リストに値を入れる変数を配置して、その行から値を取り出します。

カーソルからデータをフェッチする場合は、NEXT および FROM キーワードを省略することができます。

メモ： Zen は前方スクロール タイプのカーソルのみサポートします。そのため、NEXT FROM を省略しても、カーソルレコードの流れを制御することはできません。

例

この例の FETCH ステートメントでは、カーソル c1 の値が OldName 変数に取り込まれます。また、位置付け UPDATE ステートメントにより、同じ Demodata サンプルデータベース内の Course テーブルの Modern European History (HIS 305) の行が更新されます。

```
CREATE PROCEDURE UpdateClass();
BEGIN
  DECLARE :CourseName CHAR(7);
  DECLARE :OldName CHAR(7);
  DECLARE c1 CURSOR FOR SELECT name FROM course WHERE name = :CourseName;
  OPEN c1;
  SET :CourseName = 'HIS 305';
  FETCH NEXT FROM c1 INTO :OldName;
  UPDATE SET name = 'HIS 306' WHERE CURRENT OF c1;
END;
```

```
CREATE PROCEDURE MyProc(OUT :CourseName CHAR(7)) AS
BEGIN
  DECLARE cursor1 CURSOR
  FOR SELECT Degree, Residency, Cost_Per_Credit FROM Tuition ORDER BY ID;
  OPEN cursor1;
  FETCH NEXT FROM cursor1 INTO :CourseName;
  CLOSE cursor1;
END
```

関連項目

[CREATE PROCEDURE](#)

FOREIGN KEY

備考

テーブル定義に外部キーを追加するには、**ADD** 句で **FOREIGN KEY** キーワードを使用します。

メモ： 外部キーを追加したり、その他の参照整合性 (RI) 操作を行ったりするには、その前にデータベース名を使用してデータベースにログインする必要があります。また、セキュリティが有効になっている場合は、外部キーの参照先テーブルの参照権がなければ、キーを追加できません。

従属テーブルに外部キーを定義するには、**FOREIGN KEY** 句を **CREATE TABLE** ステートメントで使用します。キーに列のリストを指定するだけでなく、キーの名前を定義できます。

外部キー列の列は、ヌル値を許可できます。ただし、インデックスに擬似ヌルの列が含まれていないことを確認してください。擬似ヌルの値はインデックスが作成されません。

外部キー名は、辞書内で固有の名前である必要があります。外部キー名を省略すると、Zen はキーの最初の列の名前を外部キー名として使用します。その名前の外部キーが辞書に既に含まれる場合は、外部キー名の重複エラーが発生します。

外部キーを指定すると、Zen によって、キーを構成する列のインデックスが作成されます。このインデックスは、対応する主キーのインデックスと同じ属性を持ちますが、値の重複が可能です。インデックスにはほかの属性を割り当てるには、**CREATE INDEX** ステートメントを使用して明示的にインデックスを作成します。その後、**ALTER TABLE** ステートメントを使用して外部キーを定義します。インデックスを作成するときは、ヌル値が許可されず、大文字小文字の区別と照合順序の属性が、対応する主キー列のインデックスの属性と一致しているようにします。

外部キーの列は、主キーの参照先の列と、データ型、長さ、順序が同じである必要があります。唯一の例外は、外部キーでは整数列を使用して、主キーの **IDENTITY**、**SMALLIDENTITY**、または **BIGIDENTITY** 列を参照できることです。この場合、2つの列は長さが同じである必要があります。

Zen は、テーブルを作成する前に、外部キーに例外がないかどうかを調べます。以前に定義された参照整合性 (RI) 制約に違反する状態が見つかった場合は、ステータスコードを生成し、テーブルを作成しません。

メモ：既にデータを含むテーブルに外部キーを作成する際には、Zen は外部キー列および主キー列に既にあるデータ値を検証しません。この制限は、外部キーが作成された後の INSERT、UPDATE、DELETE 操作に適用されます。

外部キーを定義するときに、対応する主キーが含まれるテーブルの名前を指定する REFERENCES 句を含める必要があります。親テーブルの主キーは事前に定義されている必要があります。また、データベースのセキュリティが有効になっている場合は、主キーが含まれるテーブルの参照権が必要です。

CREATE TABLE ステートメントを使用して、自己参照を行う外部キーを作成することはできません。同じテーブルの主キーを参照する外部キーを作成するには、ALTER TABLE ステートメントを使用します。

また、1つのステートメントで、同じ列セットに対する主キーと外部キーは作成できません。そのため、作成しているテーブルの主キーが別のテーブルの外部キーでもある場合は、ALTER TABLE ステートメントを使用して外部キーを作成する必要があります。

例

次のステートメントは、Class テーブルに新規の外部キーを追加します。(Faculty の ID 列はインデックスとして定義されており、ヌル値は含まれません。)

```
ALTER TABLE Class ADD CONSTRAINT Teacher FOREIGN KEY (Faculty_ID) REFERENCES Faculty (ID) ON DELETE RESTRICT
```

この例では、削除制限規則によって、ある教職員を削除しようとするとき、最初にその教職員の全講座を変更または削除しないとその教職員をデータベースから削除できないようになっています。

関連項目

[ALTER TABLE](#)

[CREATE TABLE](#)

GRANT

セキュリティで保護されたデータベースでは、GRANT ステートメントを使用して、テーブル、ビュー、およびストアド プロシージャのアクセス権限を管理します。

GRANT では、これらの権限をユーザーまたはグループに付与したり、新しいユーザーを作成したり、そのユーザーを既存のユーザー グループに割り当てたりすることができます。必要に応じて、GRANT を使用する前に、CREATE GROUP を使用して新しいグループを作成します。

以下のトピックでは、GRANT ステートメントの使用について説明します。

- [GRANT LOGIN TO](#)
- [権限の制約](#)
- [GRANT とデータのセキュリティ](#)

構文

GRANT CREATETAB | CREATEVIEW | CREATESP TO *public* またはユーザー/グループ名 [, ユーザー/グループ名]...

GRANT LOGIN TO ユーザーおよびパスワード [, ユーザーおよびパスワード]... [**IN GROUP** グループ名]

GRANT 権限 **ON** < * | [**TABLE**] テーブル名 [オーナー ネーム] | **VIEW** ビュー名 | **PROCEDURE** ストアド プロシージャ名 >

TO ユーザー/グループ名 [, ユーザー/グループ名]...

* ::= すべてのオブジェクト (つまり、すべてのテーブル、ビュー、およびストアド プロシージャ)

```
権限 ::= ALL
      | ALTER
      | DELETE
      | INSERT [ ( テーブルの列名 [ , テーブルの列名 ]... ) ]
      | REFERENCES
      | SELECT [ ( テーブルの列名 [ , テーブルの列名 ]... ) ]
      | UPDATE [ ( テーブルの列名 [ , テーブルの列名 ]... ) ]
      | UNMASK [ ( テーブルの列名 [ , テーブルの列名 ]... ) ]
      | EXECUTE
```

テーブル名 ::= ユーザー定義のテーブル名

オーナー ネーム ::= ユーザー定義のオーナー ネーム

ビュー名 ::= ユーザー定義のビュー名

ストアド プロシージャ名 ::= ユーザー定義のストアド プロシージャ名

ユーザーおよびパスワード ::= ユーザー名 [:] パスワード

public またはユーザー/グループ名 ::= **PUBLIC** | ユーザー/グループ名

ユーザー/グループ名 ::= ユーザー名 | グループ名

ユーザー名 ::= ユーザー定義のユーザー名

テーブルの列名 ::= ユーザー定義の列名 (テーブルのみ)

備考

CREATETAB、CREATESP、CREATEVIEW、および LOGIN TO キーワードは、SQL 文法の拡張機能です。GRANT ステートメントを使用して、CREATE TABLE、CREATE VIEW、および CREATE PROCEDURE に権限を付与することができます。次の表は、特定の操作に対する構文を示しています。

この操作に権限を付与する場合	GRANT で使用するキーワード
CREATE TABLE	CREATETAB
CREATE VIEW	CREATEVIEW
CREATE PROCEDURE	CREATESP
LOGIN AS USER	LOGIN TO

CREATETAB、CREATEVIEW、および CREATESP は明示的に付与する必要があります。これらの権限は、GRANT ALL ステートメントの一部として含まれません。

GRANT LOGIN TO

GRANT LOGIN TO はユーザーを作成し、そのユーザーにセキュリティで保護されたデータベースへのアクセスを許可します。ユーザーを作成するには、ユーザー名とパスワードを指定する必要があります。任意で、ユーザーに既存のグループを使用するか、または GRANT LOGIN TO を使用する前に CREATE GROUP で新しいグループを作成することができます。

権限の制約

オブジェクトの権限には次の制約が適用されます。

- オブジェクトの種類別
- ALL キーワード

オブジェクトの種類別

次の表は、オブジェクトの種類に適用される権限を示しています。詳細については、表の下注記を参照してください。

権限	テーブル	ビュー	ストアド プロ シージャ
CREATETAB	X		
CREATEVIEW		X	
CREATESP			X
ALTER	X	X	X
DELETE	X	X	
INSERT	X	X	
REFERENCES	X		
SELECT	X	X	
UNMASK	X		
UPDATE	X	X	
EXECUTE			X

注記

- 列の権限はテーブルに対してのみ付与できます。
- ビューの権限はビュー全体に対してのみ付与でき、単一の列に対して付与することはできません。
- ユーザーは、追加の権限を付与されなくても、自身が作成したテーブル、ビュー、ストアド プロシージャを変更または削除することができます。別のユーザーが作成したオブジェクトを同様に操作する場合は、対象オブジェクトに対する ALTER 権限がユーザーに付与されている必要があります。
- 信頼されるビューおよび信頼されるストアド プロシージャを作成、変更、または削除できるのは、Master ユーザーのみです。他のユーザーに CREATEVIEW または CREATESP 権限を付与することはできません。
- セキュリティで保護されたデータベース内の信頼されないビューの場合、ユーザーがビューに含まれるテーブルの SELECT 権限を持っている場合は、そのビューの

SELECT 権限をユーザーに付与する必要はありません。しかし、信頼されるビューの場合、ユーザーには、テーブル自体ではなくビューに対してのみ SELECT 権限を付与する必要があります。

- EXECUTE はストアド プロシージャにのみ適用されます。ストアド プロシージャを実行できるのは、CALL ステートメントまたは EXECUTE ステートメントのいずれかです。プロシージャは、信頼されるものでも信頼されないものでもかまいません。[信頼されるオブジェクトと信頼されないオブジェクト](#)を参照してください。

ALL キーワード

次の表は、オブジェクトの種類で ALL によって付与される権限を示しています。[オブジェクトの種類別](#)の下に列挙されている注記はここにも適用されます。

ALL によって含まれる権限	テーブル	ビュー	ストアド プロシージャ
ALTER	X	X	X
DELETE	X	X	
INSERT	X	X	
REFERENCES	X		
SELECT	X	X	
UPDATE	X	X	
UNMASK	X	X	
EXECUTE			X

注記

GRANT ALL ON * to User1 を発行した場合、User1 は表に挙げられているすべての権限を持ちます。

GRANT ALL ON VIEW myview1 TO User2 を発行した場合、User2 は myview1 の ALTER、DELETE、INSERT、UPDATE、UNMASK、および SELECT 権限を持ちます。

GRANT とデータのセキュリティ

以下のトピックでは、データのセキュリティを管理するための GRANT の特定用途について説明します。

- [ユーザーおよびグループへの権限の付与](#)
- [オーナー ネームによるアクセス権の付与](#)

ユーザーおよびグループへの権限の付与

リレーショナル セキュリティは、セキュリティを有効にしたときに、データベースへのフル アクセス権を持つ **Master** という名前のデフォルト ユーザーが存在するかどうかに基づきます。セキュリティを有効にすると、**Master** ユーザーにパスワードの指定が必要になります。

Master ユーザーは、GRANT LOGIN TO、CREATE USER、または CREATE GROUP コマンドを使用してグループやほかのユーザーを作成したり、作成したグループやユーザーのデータ アクセス権を管理したりすることができます。

すべてのユーザーに同一の権限を付与したい場合は、PUBLIC グループに権限を付与します。すべてのユーザーは、PUBLIC グループに割り当てられているデフォルトの権限を継承します。

メモ： グループを使用する場合は、ユーザーを作成する前にグループを設定する必要があります。

ユーザー名とパスワードに空白またはその他の非英数文字が含まれる場合は、ユーザー名とパスワードを二重引用符または一重引用符で囲む必要があります。

ユーザーとグループの詳細については、『*Advanced Operations Guide*』の [Master ユーザー、ユーザーとグループ](#)、および『*Zen User's Guide*』の [権限の割り当て作業](#)を参照してください。

オーナー ネームによるアクセス権の付与

オーナー ネームは、Btrieve データ ファイルへのアクセスのロックを解除するバイト文字列です。Btrieve オーナー ネームは、オペレーティング システムのユーザー名やデータベース ユーザー名とはまったく関係なく、ファイル アクセス パスワードとして機能します。詳細については、[オーナー ネーム](#)を参照してください。

セキュリティで保護された SQL データベース内のテーブルとして、オーナー ネームを持つ Btrieve ファイルがある場合、データベースの Master ユーザーは、そのテーブルにアクセスする権限をユーザー（Master ユーザー自身も含む）に付与するためには、GRANT ステートメントでオーナー ネームを提供する必要があります。

Master ユーザーが 1 人のユーザーに対して GRANT ステートメントを実行した場合、そのユーザーはデータベースにログインするだけで、オーナー ネームを指定しなくても、テーブルにアクセスできるようになります。また、SET OWNER ステートメントを使用すると、現在のデータベース接続セッションに対して 1 つまたは複数のオーナー ネームを認証でき、接続している間は持続されることに留意してください。SET OWNER を参照してください。

ユーザーがオーナー ネームを持つテーブルに対して SQL コマンドを実行しようとした場合、Master ユーザーが GRANT ステートメントでオーナー ネームを使用することにより、そのユーザーにテーブルへのアクセス権を付与していない限り、アクセスは拒否されます。

テーブルのオーナー ネームに読み取り専用設定が選択されている場合は、すべてのユーザーがこのテーブルの SELECT 権限を持ちます。

ビューおよびストアド プロシージャに対する権限

ビューおよびストアド プロシージャは、そのビューまたはストアド プロシージャによって参照されるオブジェクトに対してどのように権限を処理したいかに応じて、信頼されるものでも信頼されないものでも指定できます。

信頼されるオブジェクトと信頼されないオブジェクト

ビューおよびストアド プロシージャは、テーブルやその他のビュー、その他のストアド プロシージャなどのオブジェクトを参照します。参照されるオブジェクトごとに権限を付与することは、オブジェクトとユーザーの数に応じて、非常に時間のかかるものとなります。多様な状況に合ったより簡潔な解決法として、信頼されるビューまたは信頼されるストアド プロシージャという概念があります。

信頼されるビューまたはストアド プロシージャとは、参照されるオブジェクトごとに権限を明示的に設定しなくても実行できるものです。たとえば、信頼されるビュー myview1 がテーブル t1 と t2 を参照している場合、Master ユーザーは t1 および t2 に権限を付与しなくても myview1 に権限を付与することができます。

信頼されないビューまたはストアド プロシージャとは、参照されるオブジェクトごとに権限を明示的に設定しなければ実行できないものです。

次の表では、信頼されるオブジェクトと信頼されないオブジェクトの特性を比較しています。

オブジェクト	特性	注記
信頼されるビューまたは信頼されるストアド プロシージャ	メタデータ バージョン 2 を必要とします。	Zen メタデータ を参照してください。
	CREATE ステートメントに WITH EXECUTE AS 'MASTER' 句を必要とします	CREATE VIEW および CREATE PROCEDURE を参照してください。
	Master ユーザーのみがオブジェクトを作成できます	『 <i>Advanced Operations Guide</i> 』の Master ユーザー を参照してください。
	Master ユーザーのみがオブジェクトを削除できます	DROP VIEW および DROP PROCEDURE を参照してください。
	Master ユーザーはほかのユーザーにオブジェクト権限を付与する必要があります	デフォルトでは、Master ユーザーのみが信頼されるビューまたはストアド プロシージャにアクセスできるので、Master ユーザーがこれらのオブジェクトに権限を付与しなければなりません。
	GRANT および REVOKE ステートメントはオブジェクトに適用されます	REVOKE も参照してください。
	オブジェクトは、セキュリティで保護された V2 データベースまたは保護されていない V2 データベースに存在することができます	『 <i>Advanced Operations Guide</i> 』の Zen セキュリティ を参照してください。
	信頼されるオブジェクトを信頼されないオブジェクトに変更する（またはその逆）には、オブジェクトを削除してから再作成する必要があります	ビューやストアド プロシージャ用の ALTER ステートメントを使用して、オブジェクトの信頼に関する特性の追加、削除を行うことはできません。信頼されるオブジェクトを信頼されないオブジェクトに変更しなければならない場合は、まずそのオブジェクトを削除してから、WITH EXECUTE AS 'MASTER' 句を指定しないでオブジェクトを作り直します。同様に、信頼されないオブジェクトを信頼されるオブジェクトに変更しなければならない場合は、まずそのオブジェクトを削除してから、WITH EXECUTE AS 'MASTER' 句を指定してオブジェクトを作り直します。

オブジェクト	特性	注記
信頼されない ビューまたは信頼されない ストアド プロシージャ	任意のユーザーがオブジェクトを作成できます	セキュリティで保護されたデータベースでは、ユーザーに CREATEVIEW または CREATESP 権限が付与されている必要があります。備考を参照してください。
	任意のユーザーがオブジェクトを削除できます	ビューまたはストアド プロシージャが別のユーザーによって作成されている場合は、ユーザーに ALTER 権限が付与されている必要があります。GRANT を参照してください。
	オブジェクトを削除するには ALTER 権限が必要となります	ALTER 権限は、テーブルを削除する場合にも必要です。デフォルトでは、 Master ユーザーのみが信頼されるオブジェクトを削除することに注意してください。ビューまたはストアド プロシージャの作成者でない (Master 以外の) ユーザーがビューやストアド プロシージャを削除するには、 ALTER 権限が付与されている必要があります。
	デフォルトで、すべてのユーザーはオブジェクトに対するすべての権限を持っています	メタデータ バージョン 2 では、セキュリティで保護されていないデータベースに信頼されないオブジェクトが含まれている場合、データベースのセキュリティを有効にすれば、信頼されないオブジェクトのすべての権限は自動的に PUBLIC に付与されます。
	ビューまたはストアド プロシージャを実行するユーザーは、これらから参照されるオブジェクトの権限を必要とします	セキュリティで保護されたデータベースでは、 Master ユーザーは各オブジェクトに対してユーザー権限を付与する必要があります。
	GRANT および REVOKE ステートメントはオブジェクトに適用されます	GRANT および REVOKE を参照してください。
	オブジェクトは、セキュリティで保護されたデータベースまたは保護されていないデータベースに存在することができます	『Advanced Operations Guide』の Zen セキュリティ を参照してください。

例

このセクションでは、UNMASK を除くさまざまな権限に対する GRANT の使用例をいくつか示します。UNMASK 権限については、『*Advanced Operations Guide*』の [UNMASK 権限の付与と取り消し](#) を参照してください。

GRANT ALL ステートメントは、指定したユーザーまたはグループに INSERT、UPDATE、ALTER、SELECT、DELETE、および REFERENCES の権限を付与します。また、辞書の CREATE TABLE 権限も付与されます。次のステートメントによって、ユーザー *dannyd* にテーブル *Class* の上記のすべての権限が付与されます。

```
GRANT ALL ON Class TO dannyd
```

次のステートメントによって、ユーザー *debieg* にテーブル *Class* の ALTER 権限が付与されます。

```
GRANT ALTER ON Class TO debieg
```

次のステートメントによって、ユーザー *keithv* および *miked* に、テーブル *Class* の INSERT 権限が付与されます。このテーブルは *winsvr644AdminGrp* というオーナー ネームを持ちます。

```
GRANT INSERT ON Class winsvr644AdminGrp TO keithv, miked
```

次のステートメントによって、ユーザー *keithv* および *miked* に、テーブル *Class* の INSERT 権限が付与されます。

```
GRANT INSERT ON Class TO keithv, miked
```

次のステートメントによって、ユーザー *keithv* および *brendanb* に、*Person* テーブル内の 2 つの列 *First_name* と *Last_name* の INSERT 権限が付与されます。

```
GRANT INSERT(First_name,last_name) ON Person to keithv,brendanb
```

次のステートメントによって、ユーザー *aideenw* および *punitas* に CREATE TABLE 権限が付与されます。

```
GRANT CREATETAB TO aideenw, punitas
```

次の GRANT LOGIN TO ステートメントは *ravi* という名前のユーザーにログイン権限を与え、パスワード *password* を指定します。

```
GRANT LOGIN TO ravi:password
```

メモ： GRANT LOGIN TO ステートメントを使用してログイン権限を与えられたユーザー アカウントが現存しない場合、そのユーザー アカウントが作成されます。

ストアド プロシージャで GRANT LOGIN ステートメントを使用する場合は、ユーザー名とパスワードをコロンではなく空白文字で区切る必要があります。コロン文字は、ストアド プロシージャ内でローカル変数の識別に使用されます。

このユーザー名およびパスワードは Zen データベースのみを対象としたもので、オペレーティング システムまたはネットワークの認証に使用されるユーザー名およびパスワードとは無関係です。Zen ユーザー名、グループ、およびパスワードは Zen Control Center (ZenCC) を使用して設定することもできます。

次の例では、*dannyd* と *rgarcia* という名前のユーザーにログイン権限が与えられ、パスワードがそれぞれ *password*、*1234567* と指定されます。

```
GRANT LOGIN TO dannyd:password,rgarcia:1234567
```

名前に空白が含まれる場合は、次のように二重引用符を使用できます。このステートメントによって、*Jerry Gentry* と *Punita* という名前のユーザーにログイン権限が与えられ、パスワードがそれぞれ *sun*、*moon* と指定されます。

```
GRANT LOGIN TO 'Jerry Gentry':sun, Punita:moon
```

次の例では、パスワード *123456* を持つ *Jerry Gentry* という名前のユーザーと、パスワード *abcdef* を持つ *rgarcia* という名前のユーザーにログイン権限が与えられます。また、グループ *zen_dev* にこれらのユーザーが追加されます。

```
GRANT LOGIN TO "Jerry Gentry":123456, rgarcia:abcdef IN GROUP zen_dev
```

Master ユーザーは、オーナー ネームを持たないテーブルのすべての権限を持ちます。Btrieve オーナー ネームを持つテーブルの権限を付与するには、Master ユーザーは GRANT ステートメントで正しいオーナー ネームを指定しなければなりません。

次の例では、Master ユーザーに Btrieve オーナー ネーム *abcd* を持つテーブル *t1* の SELECT 権限が与えられます。

```
GRANT SELECT ON t1 'abcd' TO Master
```

テーブルのオーナー ネームは、ZenCC の [ツール] メニューにある Function Executor または Maintenance ユーティリティを使用して設定できます。詳細については、『*Advanced Operations Guide*』の[オーナー ネーム](#)を参照してください。

Master ユーザーによる次の SQL ステートメント セットの実行後、ユーザー jsmith は現在のデータベース内のすべてのテーブルに対する SELECT 権を持ちます。また、ユーザーは tab1 の DELETE 権および tab2 の UPDATE 権も持ちます。

```
GRANT DELETE ON tab1 TO jsmith
GRANT SELECT ON * TO jsmith
GRANT UPDATE ON tab2 TO jsmith
```

その後、CREATE TABLE 権を持つユーザーによって次のステートメントが実行された場合、ユーザー jsmith は新しく作成されたテーブルの SELECT 権を持ちます。

```
CREATE TABLE tab3 (col1 INT)
```

```
GRANT CREATETAB TO user1
```

```
GRANT CREATESP TO user1
```

次の例では、**すべての**ユーザーにストアド プロシージャ cal_rtrn_rate の EXECUTE 権限が付与されます。

```
GRANT EXECUTE ON PROCEDURE cal_rtrn_rate TO PUBLIC
```

次の例は、Accounting グループのメンバーは employee テーブルの salary 列だけを更新できることを示しています (employee は Demodata サンプル データベースの一部です)。

次のようなストアド プロシージャが存在するとします。

```
CREATE PROCEDURE employee_proc_upd(in :EmpID integer, in :Salary money) WITH EXECUTE AS 'Master';
BEGIN
UPDATE employee SET Salary = :Salary WHERE EmployeeID = :Empid;
END
GRANT EXECUTE ON PROCEDURE employee_proc_upd TO Accounting
```

グループ Accounting に属するユーザーは、Employee テーブル内の salary 以外の列を更新できないことに注目してください。これは、権限がストアド プロシージャに対してのみ付与されており、そのストアド プロシージャが salary 列のみを更新するためです。

次の例では、**Demodata** サンプル データベースのセキュリティが有効で、ユーザー名 **USAcctsMgr** が追加されているものとします。ここで、そのユーザーに対し、テーブル **Person** の ID 列への **SELECT** 権を付与します。以下のステートメントを使用します。

```
GRANT SELECT ( ID ) ON Person TO 'USAcctsMgr'
```

関連項目

CREATE GROUP	REVOKE
CREATE PROCEDURE	SET OWNER
CREATE VIEW	SET SECURITY
DROP GROUP	システム ストアド プロ シージャ

GROUP BY

SELECT ステートメント中の GROUP BY 構文に加えて、Zen では、ベンダー文字列を含むことができる拡張された GROUP BY 構文もサポートしています。

GROUP BY クエリを行うと、見つかったグループごとに 1 行の選択リストを含む結果セットが返されます（選択リストの構文については、[SELECT](#) を参照してください）。

次の例では、エスケープ シーケンス内にベンダー文字列を含む、拡張 GROUP BY を示します。

```
create table at1 (col1 integer, col2 char(10));
insert into at1 values (1, 'abc');
insert into at1 values (2, 'def');
insert into at1 values (3, 'aaa');

SELECT (--(*vendor(Microsoft), product(ODBC) fn left(at1.col2, 1) *)--) atv, count(*) Total FROM at1
GROUP BY atv
ORDER BY atv DESC
```

戻り値：

atv	Total
d	1
a	2

関連項目

[SELECT](#)

HAVING

HAVING 句は SELECT ステートメントの中で GROUP BY キーワードと一緒に使用して、集計値が特定の基準を満たしているグループのみを表示します。

HAVING 句内の式には、定数、セット関数、または GROUP BY 式リスト内の式の複製を含めることができます。

Zen データベース エンジン は、GROUP BY のない HAVING キーワードをサポートしていません。

HAVING キーワードでは、エイリアスを使用することができます。ただし、エイリアスはテーブル内のどの列名とも異なっていなければなりません。

例

次の例では、講座名が 5 つを超える学部名が返されます。

```
SELECT Dept_Name, COUNT(*) FROM Course GROUP BY Dept_Name HAVING COUNT(*) > 5
```

上と同じ例でエイリアス（ここでは **dn** と **ct**）を使用して、同様の結果を得られます。

```
SELECT Dept_Name dn, COUNT(*) ct FROM Course GROUP BY dn HAVING ct > 5
```

COUNT(式) によって、述部にある式の非ヌル値がすべてカウントされるということを覚えておいてください。COUNT(*) ではヌル値を含むすべての値がカウントされます。

次の例では、講座数が 5 つを超える **Accounting** という名前の学部名が返されます。

```
SELECT Dept_Name, COUNT(*) FROM Course GROUP BY Dept_Name HAVING COUNT(*) > 5 AND Dept_Name = 'Accounting'
```

関連項目

SELECT

IF

構文

```
IF ( ブール条件 )  
BEGIN  
SQL ステートメント  
END  
ELSE  
BEGIN  
SQL ステートメント  
END
```

備考

IF ステートメントにより、条件の値に基づく条件付き実行を提供します。IF ... THEN ... ELSE ... END IF 構造は、2 つのステートメント ブロックのどちらに基づいて実行するかをフロー制御します。IF ... ELSE 構文も使用できます。

IF ステートメントは、ストアド プロシージャとトリガーの本体で使用できます。

IF ステートメントをネストできる数に制限はありませんが、クエリは通常どおり、合計長の制限やその他のアプリケーション制限の影響を受けます。

メモ： Zen と T.SQL の構文の混合構文を使用することはできません。

IF...THEN...ELSE...END IF 構文または IF...ELSE 構文のいずれかを使用できます。IF または ELSE 条件で複数のステートメントを使用する場合は、BEGIN と END を使ってステートメント ブロックの始まりと終わりを示す必要があります。

例

次の例では、IF ステートメントを使用して、vInteger の値が正か負かによって変数 Negative を 1 または 0 に設定します。

```
IF (:vInteger < 0) THEN  
SET :Negative = '1';  
ELSE  
SET :Negative = '0';  
END IF;
```

次の例では、IF ステートメントを使用して、定義された条件 (SQLSTATE = '02000') のループをテストします。この条件が満たされた場合、WHILE ループが終了します。


```
FETCH_LOOP:
WHILE (:counter < :NumRooms) DO
FETCH NEXT FROM cRooms INTO :CurrentCapacity;
IF (SQLSTATE = '02000') THEN
LEAVE FETCH_LOOP;
END IF;
SET :counter = :counter + 1;
SET :TotalCapacity = :TotalCapacity +
:CurrentCapacity;
END WHILE;

IF(:vInteger >50)
BEGIN
SET :vInteger = :vInteger + 1;
INSERT INTO test VALUES('Test');
END;
ELSE
SET :vInteger = :vInteger - 1;
```

関連項目

[CREATE PROCEDURE](#)

[CREATE TRIGGER](#)

IN

備考

IN 演算子を使用して、外部クエリの結果が、サブクエリの結果に含まれるかどうかを検査します。ステートメントの結果テーブルには、外部クエリが返す行で、サブクエリ内に関連する行があるものだけが含まれます。

例

次の例は、Chemistry 408 を受講した全学生の名前をリストします。

```
SELECT p.First_Name + ' ' + p.Last_Name FROM Person p, Enrolls e WHERE (p.id = e.student_id) AND
(e.class_id IN
(SELECT c.ID FROM Class c WHERE c.Name = 'CHE 408'))
```

Zen では、まずサブクエリが評価され、Class テーブルから Chemistry 408 の ID が取得されます。次に外部クエリが実行され、結果は、その講座の Enrolls テーブルにエントリがある学生だけに制限されます。

一般に、IN クエリは、EXISTS キーワードまたは制限句を持つ簡単な結合条件を使用すると、より効率的に行うことができます。Zen では、サブクエリよりも結合の方が効率的に最適化されます。そのため、クエリの目的がデータベースのサブセットでの値の有無を判定することでない限り、簡単な結合条件を使用する方が効率的です。

関連項目

[SELECT](#)

INSERT

このステートメントにより、1つのテーブルに列の値を挿入します。

構文

```
INSERT INTO テーブル名
[ ( 列名 [ , 列名 ]... ) ] 挿入値
[ ON DUPLICATE KEY UPDATE 列名 = < NULL | DEFAULT | 式 | サブクエリ式 [ , 列名 = ... ] >
[ [ UNION [ ALL ] クエリ スペック ]...
[ ORDER BY order-by 式 [ , order-by 式 ]... ]
```

テーブル名 ::= ユーザー定義名

列名 ::= ユーザー定義名

挿入値 ::= values 句 | クエリ スペック

values 句 ::= VALUES (式 [, 式]...) | DEFAULT VALUES

式 ::= 式 - 式 | 式 + 式

サブクエリ式 ::= (クエリ スペック) [ORDER BY order-by 式
[, order-by 式]...] [limit 句]

クエリ スペック ::= (クエリ スペック)

| SELECT [ALL | DISTINCT] [top 句] 選択リスト

FROM テーブル参照 [, テーブル参照]...

[WHERE 検索条件]

[GROUP BY 式 [, 式]...

[HAVING 検索条件]]

order-by 式 ::= 式 [CASE (文字列) | COLLATE 照合順序名] [ASC | DESC]

備考

INSERT ステートメントは、DELETE および UPDATE と同様にアトミックな方法で動作します。つまり、複数の行の挿入に失敗した場合、同じステートメントによって実行された前の行の挿入がすべてロールバックされます。

INSERT ON DUPLICATE KEY UPDATE

Zen v13 R2 では、INSERT ON DUPLICATE KEY UPDATE により、INSERT が拡張されました。この挿入機能は、一意のキーについて、挿入または更新される値と、対象テーブル内の値を自動的に比較します。重複する主キーまたはインデックス キーのいずれかが見つかった場合は、それらの行の値が更新されます。重複する主キーもインデックス キーも見つからなかった場合は、新しい行が挿入されます。一般的な専門用語では、この動作は「upsert」と呼ばれます。

INSERT では、値リストまたは SELECT クエリのいずれかを使用できます。すべての INSERT コマンドと同様に、この動作はアトミックです。

この機能の実例については、[INSERT ON DUPLICATE KEY UPDATE の例](#)を参照してください。

リテラル文字列の最大長より長いデータの挿入

Zen でサポートされるリテラル文字列の最大長は 15,000 バイトです。これよりも長いデータを処理するには、直接の SQL ステートメントを使用し、挿入を複数の呼び出しに分割します。次のようなステートメントで開始します。

```
INSERT INTO table1 SET longfield = '15000 バイトのテキスト ' WHERE 制限
```

次に、それ以上のデータを追加する次のステートメントを発行します。

```
INSERT INTO table1 SET longfield = notefield + '15000 バイトを超えたテキスト ' WHERE 制限
```

例

- [INSERT の例](#)
- [INSERT ON DUPLICATE KEY UPDATE の例](#)
- [DEFAULT を使用した場合に発生するエラー](#)

INSERT の例

このトピックでは、単純な INSERT の例を示します。重複する一意キーを用いて、挿入ではなく更新する場合については、[INSERT ON DUPLICATE KEY UPDATE の例](#)を参照してください。

次のステートメントでは、VALUES 句で式を使ってテーブルにデータを追加しています。

```
CREATE TABLE t1 (c1 INT, c2 CHAR(20))
INSERT INTO t1 VALUES ((78 + 12)/3, 'This is' + CHAR(32) + 'a string')
SELECT * FROM t1
```

c1	c2
30	This is a string

次のステートメントでは、3 つの VALUES 句を使用して、Course テーブルにデータを直接追加しています。

```
INSERT INTO Course(Name, Description, Credit_Hours, Dept_Name)
VALUES ('CHE 308', 'Organic Chemistry II', 4, 'Chemistry')
INSERT INTO Course(Name, Description, Credit_Hours, Dept_Name)
VALUES ('ENG 409', 'Creative Writing II', 3, 'English')
INSERT INTO Course(Name, Description, Credit_Hours, Dept_Name)
VALUES ('MAT 307', 'Probability II', 4, 'Mathematics')
```

以下の INSERT ステートメントでは、SELECT 句を使用して、授業を受けた学生の ID 番号が Student テーブルから取り出されます。

次に、その ID 番号が Billing テーブルに挿入されます。

```
INSERT INTO Billing (Student_ID)
SELECT ID FROM Student WHERE Cumulative_Hours > 0
```

次の例は、CURTIME()、CURDATE()、NOW() の各変数を使用して、現在の現地時刻、日付、タイムスタンプの値を INSERT ステートメントに挿入する方法を示します。

```
CREATE TABLE Timetbl (c1 TIME, c2 DATE, c3 TIMESTAMP)
INSERT INTO Timetbl(c1, c2, c3) VALUES(CURTIME(), CURDATE(), NOW())
```

次の例は、INSERT および UPDATE ステートメントでのデフォルト値の基本的な使い方を示します。

```
CREATE TABLE t1 (c1 INT DEFAULT 10, c2 CHAR(10) DEFAULT 'abc')
INSERT INTO t1 DEFAULT VALUES
INSERT INTO t1 (c2) VALUES (DEFAULT)
INSERT INTO t1 VALUES (100, DEFAULT)
INSERT INTO t1 VALUES (DEFAULT, 'bcd')
INSERT INTO t1 VALUES (DEFAULT, DEFAULT)
SELECT * FROM t1
```

c1	c2
10	abc
10	abc
100	abc
10	bcd
10	abc

```
UPDATE t1 SET c1 = DEFAULT WHERE c1 = 100
UPDATE t1 SET c2 = DEFAULT WHERE c2 = 'bcd'
UPDATE t1 SET c1 = DEFAULT, c2 = DEFAULT
SELECT * FROM t1
```

c1	c2

```
10      abc
10      abc
10      abc
10      abc
10      abc
```

直前に記した CREATE TABLE ステートメントを基にした場合、次の 2 つの INSERT ステートメントはまったく同じです。

```
INSERT INTO t1 (c1,c2) VALUES (20,DEFAULT)
```

```
INSERT INTO t1 (c1) VALUES (20)
```

次の SQL コードは、複数の UPDATE 値での DEFAULT の使い方を示します。

```
CREATE TABLE t2 (c1 INT DEFAULT 10,
c2 INT DEFAULT 20 NOT NULL,
c3 INT DEFAULT 100 NOT NULL)
INSERT INTO t2 VALUES (1, 1, 1)
INSERT INTO t2 VALUES (2, 2, 2)
SELECT * FROM t2
```

c1	c2	c3
1	1	1
2	2	2

```
UPDATE t2 SET c1 = DEFAULT, c2 = DEFAULT, c3 = DEFAULT
WHERE c2 = 2
SELECT * FROM t2
```

c1	c2	c3
1	1	1
10	20	100

INSERT ON DUPLICATE KEY UPDATE の例

このトピックでは、INSERT ON DUPLICATE KEY UPDATE の例を示します。単純な INSERT については、[INSERT の例](#)を参照してください。

わかりやすくするために、以下のクエリ結果では、挿入された値を黒色で、更新された値を赤色で示しています。各例は前の例に基づいて進めているため、順番に実行していくと動作を確認できます。

VALUES 句があり、列リストがない INSERT INTO。一意のインデックス セグメント列の値を使用できます。

```
CREATE TABLE t1 (
```

```

a INT NOT NULL DEFAULT 10,
b INT,
c INT NOT NULL,
d INT DEFAULT 20,
e INT NOT NULL DEFAULT 1,
f INT NOT NULL DEFAULT 2,
g INT,
h INT,
PRIMARY KEY(e, f) );
CREATE UNIQUE INDEX t1_ab ON t1 ( a, b, c, d );
INSERT INTO t1 VALUES ( 1, 2, 3, 4, 5, 6, 7, 8 )
ON DUPLICATE KEY UPDATE t1.a = 10, t1.b = 20, t1.c = 30, t1.d = 40;
SELECT * FROM t1;

```

a	b	c	d	e	f	g	h
1	2	3	4	5	6	7	8

VALUES 句と完全な列リストがある INSERT INTO。行は更新されます。

```

INSERT INTO t1 ( a, b, c, d, e, f, g, h ) VALUES ( 1, 2, 3, 4, 5, 6, 7, 8 )
ON DUPLICATE KEY UPDATE t1.a = 10, t1.b = 20, t1.c = 30, t1.d = 40;
SELECT * FROM t1;

```

a	b	c	d	e	f	g	h
10	20	30	40	5	6	7	8

VALUES 句と部分的な列リストがある INSERT INTO。新しい行が挿入され、その行が更新されます。

```

INSERT INTO t1 ( a, b, c, d ) VALUES ( 1, 2, 3, 4 )
ON DUPLICATE KEY UPDATE t1.a = 11, t1.b = 12, t1.c = 13, t1.d = 14;
SELECT * FROM t1;

```

a	b	c	d	e	f	g	h
10	20	30	40	5	6	7	8
1	2	3	4	1	2	(Null)	(Null)

```

INSERT INTO t1 ( a, b, c ) VALUES ( -1, -2, -3 )
ON DUPLICATE KEY UPDATE t1.a = 11, t1.b = 12, t1.c = 13, t1.d = 14;
SELECT * FROM t1;

```

a	b	c	d	e	f	g	h
10	20	30	40	5	6	7	8
11	12	13	14	1	2	(Null)	(Null)

VALUES 句と DEFAULT がある INSERT INTO。行は、以前の状態に戻すように更新された後、重複キーに基づいて更新されます。

```

UPDATE t1 SET a = 1, b = 2, c = 3, d = 4, e = 11, f = 12 WHERE a = 11;
SELECT * FROM t1;

```

a	b	c	d	e	f	g	h
10	20	30	40	5	6	7	8
1	2	3	4	11	12	(Null)	(Null)

```

INSERT INTO t1 ( a, b, c, d, e, f ) VALUES ( 1, 2, 3, 4, DEFAULT, DEFAULT )
ON DUPLICATE KEY UPDATE g = VALUES ( a ) + VALUES ( b ) + VALUES ( c ), h = VALUES ( e ) + VALUES ( f );
SELECT * FROM t1;

```

a	b	c	d	e	f	g	h
10	20	30	40	5	6	7	8
1	2	3	4	11	12	6	3

UPDATE の SET 句でサブクエリ式を使用して、Demodata サンプル データベースの Person テーブルの値を基に更新します。

```

INSERT INTO t1 VALUES ( 1, 2, 3, 4, 5, 6, 7, 8 )
ON DUPLICATE KEY UPDATE t1.a = 10, t1.b = 20, t1.c = ( SELECT TOP 1 id FROM demodata.person ORDER BY id ), t1.d = ( SELECT TOP 1 id FROM demodata.person ORDER BY id DESC, last_name );
SELECT * FROM t1;

```

a	b	c	d	e	f	g	h
10	20	100062607	998332124	5	6	7	8
1	2	3	4	11	12	6	3

DEFAULT を使用した場合に発生するエラー

次の例は、列が NOT NULL として定義されており、デフォルト値が定義されていないことによって起こり得るエラー状況を示します。

```

CREATE TABLE t1 (c1 INT DEFAULT 10, c2 INT NOT NULL, c3 INT DEFAULT 100 NOT NULL)
INSERT INTO t1 DEFAULT VALUES -- エラー：列 <c2> にデフォルト値が割り当てられていません。
INSERT INTO t1 VALUES (DEFAULT, DEFAULT, 10) -- エラー：列 <c2> にデフォルト値が割り当てられていません。
INSERT INTO t1 (c1,c2,c3) VALUES (1, DEFAULT, DEFAULT) -- エラー：列 <c2> にデフォルト値が割り当てられていません。
INSERT INTO t1 (c1,c3) VALUES (1, 10) -- エラー：列 <c2> はヌル値を許可しません。

```

次の例は、IDENTITY 列およびデフォルト値を持つ列に対し INSERT を使用したとき、どのようになるかを示します。

```

CREATE TABLE t (id IDENTITY, c1 INTEGER DEFAULT 100)
INSERT INTO t (id) VALUES (0)
INSERT INTO t VALUES (0,1)
INSERT INTO t VALUES (10,10)
INSERT INTO t VALUES (0,2)
INSERT INTO t (c1) VALUES (3)
SELECT * FROM t

```

SELECT によって、テーブルに次の行が含まれていることが示されます。


```
1、100
2、1
10、10
11、2
12、3
```

最初の行は、IDENTITY 列に対して VALUES 句で 0 を指定すると、テーブルが空の場合に挿入される値は 1 であることを示します。

またこの行は、デフォルト値を持つ列に対して VALUES 句で値を指定しないと、指定されているデフォルト値が挿入されることも示します。

2 行目は、IDENTITY 列に対して VALUES 句で 0 を指定すると、挿入される値は、IDENTITY 列中で最も大きな値より 1 大きい値になることを示します。

この行はまた、デフォルト値を持つ列に対して VALUES 句で値を指定すると、指定した値によりデフォルト値が上書きされることを示します。

3 行目は、IDENTITY 列に対して VALUES 句で 0 以外の値を指定すると、指定した値が挿入されることを示します。IDENTITY 列に対して指定された値を含む行が既に存在する場合は、"レコードのキー フィールドに重複するキー値があります (Btrieve エラー 5)" というメッセージが返され、INSERT は失敗します。

4 行目では再び、IDENTITY 列に対して VALUES 句で 0 を指定すると、IDENTITY 列中で最も大きな値より 1 大きい値が挿入されることが示されています。これは、値の間に「ギャップ」がある場合でも当てはまります（つまり、IDENTITY 列が最も大きな値よりも小さい値の行が 1 行以上欠落するということです）。

5 行目は、IDENTITY 列に対して VALUES 句で値を指定しないと、挿入される値は、IDENTITY 列中で最も大きな値より 1 大きい値になることを示します。

関連項目

[CREATE TABLE](#)

[DEFAULT](#)

[SELECT](#)

[SET ANSI_PADDING](#)

JOIN

1 つのテーブルまたはビューを指定することも、複数のテーブルを指定することもできます。あるいは、1 つのビューと複数のテーブルを指定することもできます。複数のテーブルを指定する場合、テーブルを結合するといいます。

構文

```
結合定義 ::= テーブル参照 [ 結合タイプ ] JOIN テーブル参照 ON 検索条件
          | テーブル参照 CROSS JOIN テーブル参照
          | 外部結合定義
```

```
結合タイプ ::= INNER | LEFT [ OUTER ] | RIGHT [ OUTER ] | FULL [ OUTER ]
```

```
外部結合定義 ::= テーブル参照 外部結合タイプ JOIN テーブル参照
ON 検索条件
```

```
外部結合タイプ ::= LEFT [ OUTER ] | RIGHT [ OUTER ] | FULL [ OUTER ]
```

次の例は、2 つのテーブルの外部結合を示しています。

```
SELECT * FROM Person LEFT OUTER JOIN Faculty ON Person.ID = Faculty.ID
```

次の例は、ベンダー文字列に埋め込まれた外部結合を示しています。文字 "OJ" は大文字と小文字のどちらでもかまいません。

```
SELECT t1.deptno, ename FROM {OJ emp t2 LEFT OUTER JOIN dept t1 ON t2.deptno=t1.deptno}
```

Zen は、Microsoft の ODBC ドキュメントに明記されている 2 つのテーブルの外部結合をサポートしています。単純な 2 つのテーブルの外部結合のほかにも、Zen では *n* とおりのネストされた外部結合がサポートされています。

外部結合は、ベンダー文字列に埋め込んでも埋め込まなくてもかまいません。ベンダー文字列が使用された場合、Zen ではそれをストリップし、実際の外部結合テキストを解析します。

LEFT OUTER

Zen データベースは、SQL92 (SQL2) をモデルとする LEFT OUTER JOIN を使用しています。構文は、クロス結合、右外部結合、完全外部結合、および内部結合が組み込まれた全 SQL92 構文のサブセットです。以下の TableRefList は、SELECT ステートメント内の FROM キーワードと、後続の WHERE、HAVING、およびその他の句の間に置かれます。TableRef と LeftOuterJoin の再帰的な性質に注意してください。TableRef が TableRef を含む左外部結合で、その TableRef も左外部結合で、というように続けることができます。

```

TableRefList :
TableRef [, TableRefList]
| TableRef
| OuterJoinVendorString [, TableRefList]
TableRef :
TableName [CorrelationName]
| LeftOuterJoin
| (LeftOuterJoin )
LeftOuterJoin :
TableRef LEFT OUTER JOIN TableRef ON SearchCond

```

検索条件 (SearchCond) には結合条件が含まれます。この条件の通常の形式は、*LT.ColumnName = RT.ColumnName* のようになります。ここで、*LT* は左テーブル、*RT* は右テーブル、*ColumnName* は指定されたドメイン内の列を表します。検索条件内の各述部には、非リテラル式が含まれている必要があります。

左外部結合の実装は、Microsoft の ODBC ドキュメントに示されている構文の範囲を超えています。

ベンダー文字列

前のセクションの構文は、Microsoft ODBC ドキュメントに記載されていますが、詳しい説明は含まれていません。さらに、左外部結合の最初と最後のベンダー文字列エスケープシーケンスによって外部結合のコア構文は変わりません。

Zen データベースでは、ベンダー文字列を含まない外部結合構文を受け入れます。ただし、複数データベースにわたる ODBC 準拠を必要とするアプリケーションでは、ベンダーの文字列構成を使用する必要があります。ODBC のベンダー文字列外部結合では 3 つ以上のテーブルはサポートされないため、例に記載されている構文を使用しなければならない場合があります。

例

以下の例では、次の 4 つのテーブルを使用します。

Emp テーブル

FirstName	LastName	DeptID	EmpID
Franky	Avalon	D103	E1
Gordon	Lightfoot	D102	E2
Lawrence	Welk	D101	E3
Bruce	Cockburn	D102	E4

Dept テーブル

DeptID	LocID	Name
D101	L1	TV
D102	L2	Folk

Addr テーブル

EmpID	Street
E1	101 Mem Lane
E2	14 Young St.

Loc テーブル

LocID	Name
L1	PlanetX
L2	PlanetY

以下の例は、単純な 2 方向左外部結合を示しています。

```
SELECT * FROM Emp LEFT OUTER JOIN Dept ON Emp.DeptID = Dept.DeptID
```

この単純な 2 方向外部結合では、次の結果セットが作成されます。

Emp				Dept		
FirstName	LastName	DeptID	EmpID	DeptID	LocID	Name

Emp				Dept		
Franky	Avalon	D103	E1	NULL	NULL	NULL
Gordon	Lightfoot	D102	E2	D102	L2	Folk
Lawrence	Welk	D101	E3	D101	L1	TV
Bruce	Cockburn	D102	E4	D102	L2	Folk

テーブル内の Franky Avalon の NULL エントリに注目してください。これは、D103 の DeptID が Dept テーブル内に見つからなかったためです。標準 (INNER) 結合では、Franky Avalon はまとめて結合セットから削除されてしまうでしょう。

アルゴリズム

前の例で Zen データベース エンジンが使用しているアルゴリズムは、次のようになります。左テーブルを取り出して、右テーブルを走査します。現行の右テーブル行で ON 条件が TRUE になる場合ごとに、現行の左テーブル行に追加される適合した右テーブル行から成る結果セット行を返します。

ON 条件が TRUE の右テーブル行がない（現行の左テーブル行に対し、すべての右テーブル行で ON 条件が FALSE になる）場合は、すべての列の値を NULL にして右テーブルの行インスタンスを作成します。

その結果セットは、行ごとに現行の左テーブル行と結合され、返される結果セット内でインデックス付けされます。左テーブル行ごとにこのアルゴリズムが繰り返されて、完全な結果セットが構築されます。前述の単純な 2 方向左外部結合では、Emp が左テーブルで Dept が右テーブルです。

メモ： アルゴリズムとは関係ありませんが、左テーブルを右テーブルに追加する場合、クエリの選択リスト内に指定されたように適切に射影が行われることが前提となります。この射影の範囲は、すべての列（たとえば、SELECT * FROM ...）から、結果セット内の 1 つの列（たとえば、SELECT FirstName FROM ...）です。

放射状左外部結合では、1 つの中央テーブルにほかのすべてのテーブルが結合されます。以下の 3 方向の放射状左外部結合の例では、Emp が中央テーブルで、すべての結合はそのテーブルから射影したものです。

```
SELECT * FROM (Emp LEFT OUTER JOIN Dept ON Emp.DeptID = Dept.DeptID) LEFT OUTER JOIN Addr ON
Emp.EmpID = Addr.EmpID
```

Emp				Dept			Addr	
First Name	Last Name	Dept ID	Emp ID	Dept ID	Loc ID	Name	Emp ID	Street
Franky	Avalon	D103	E1	NULL	NULL	NULL	E1	101 Mem Lane
Gordon	Lightfoot	D102	E2	D102	L2	Folk	E2	14 Young St
Lawrence	Welk	D101	E3	D101	L1	TV	NULL	NULL
Bruce	Cockburn	D102	E4	D101	L1	TV	NULL	NULL

チェーン状の左外部結合では、1 つのテーブルが別のテーブルに結合され、そのテーブルがまた別のテーブルに結合されます。次の例は、3 方向のチェーン状左外部結合を示しています。

```
SELECT * FROM (Emp LEFT OUTER JOIN Dept ON Emp.DeptID = Dept.DeptID) LEFT OUTER JOIN Loc ON
Dept.LocID = Loc.LocID
```

Emp				Dept			Loc	
First Name	Last Name	Dept ID	Emp ID	Dept ID	Loc ID	Name	Loc ID	Name
Franky	Avalon	D103	E1	NULL	NULL	NULL	NULL	NULL
Gordon	Lightfoot	D102	E2	D102	L2	Folk	L2	PlanetY
Lawrence	Welk	D101	E3	D101	L1	TV	L1	PlanetX
Bruce	Cockburn	D102	E4	D101	L1	TV	L1	PlanetX

この結合は、次のように表すこともできます。

```
SELECT * FROM Emp LEFT OUTER JOIN (Dept LEFT OUTER JOIN Loc ON Dept.LocID = Loc.LocID) ON Emp.DeptID
= Dept.DeptID
```

最初の構文は放射状結合とチェーン状結合の両方に使用できるため、最初の構文をお勧めします。ネストされた左外部結合 ON 条件はネスト外部のテーブル内の列を参照できないため、この 2 番目の構文は放射状結合には使用できません。つまり、次のクエリでは Emp.EmpID への参照は不正です。

```
SELECT * FROM Emp LEFT OUTER JOIN (Dept LEFT OUTER JOIN Addr ON Emp.EmpID = Addr.EmpID) ON Emp.DeptID
= Dept.DeptID
```

次の例は、最適化が足りない 3 方向の放射状左外部結合を示しています。

```
SELECT * FROM Emp E1 LEFT OUTER JOIN Dept ON E1.DeptID = Dept.DeptID, Emp E2 LEFT OUTER JOIN Addr ON
E2.EmpID = Addr.EmpID WHERE E1.EmpID = E2.EmpID
```

Emp				Dept			Addr	
First Name	Last Name	Dept ID	Emp ID	Dept ID	Loc ID	Name	Emp ID	Street
Franky	Avalon	D103	E1	NULL	NULL	NULL	E1	101 Mem Lane
Gordon	Lightfoot	D102	E2	D102	L2	Folk	E2	14 Young St
Lawrence	Welk	D101	E3	D101	L1	TV	NULL	NULL
Bruce	Cockburn	D102	E4	D101	L1	TV	NULL	NULL

Emp 内の EmpID に NULL 値がなく、EmpID が重複値のない列である場合、このクエリでは、[Loc テーブル](#)の結果と同じ結果が返されます。ただし、このクエリは [Loc テーブル](#)で示される結果ほど最適化されておらず、はるかに遅くなる可能性があります。

関連項目

[SELECT](#)

LAG

LAG は、ウィンドウ関数としてのみ使用できるセット関数です。LAG は、結果セットの現在のパーティションにある前の行から値を取得する場合に使用されます。GROUP BY 集計としては使用できません。

構文

LAG (式[, lag オフセット式[, lag デフォルト式]] over 句)

各項目の説明は次のとおりです。

- **式**は、結果セット内の行の列または式です。
- **lag オフセット式**は、結果セット内の現在の行から何行前にある行の値を表示するかを示す整数式です。
- **lag デフォルト式**は、現在のパーティション内の **lag オフセット式** の行がまだ結果セットに累積されていない場合に返すための値です。

例

以下に示す LAG の例を実行するために、まず次の手順を実行してください。

1. ZenCC の Zen エクスプローラーで [データベース] を右クリックし、次に [新規作成] > [データベース] を選択して一時データベースを作成します。この場合は SAMPLEDB という名前にします。
2. SAMPLEDB を右クリックして [SQL ドキュメント] を選択します。
3. SQL Editor で次の **SQL スクリプト** を実行して、SAMPLEDB に checkout0 というテーブルを作成します。

```
create table checkout0 (StartTime TIMESTAMP, LaneNo INTEGER, Items INTEGER, Total MONEY(6,2),
Duration INTEGER);
insert into checkout0 values('2021-01-07 22:20:37.852', 1, 7, 18.72, 87);
insert into checkout0 values('2021-01-07 22:22:47.852', 1, 9, 23.45, 107);
insert into checkout0 values('2021-01-07 22:24:57.852', 1, 17, 68.22, 183);
insert into checkout0 values('2021-01-07 22:31:07.852', 1, 12, 48.17, 99);
insert into checkout0 values('2021-01-07 22:33:27.852', 1, 11, 37.77, 77);
insert into checkout0 values('2021-01-07 22:34:39.852', 1, 7, 23.32, 94);
insert into checkout0 values('2021-01-07 22:36:41.852', 1, 14, 56.61, 112);
insert into checkout0 values('2021-01-07 22:19:37.852', 2, 6, 16.72, 80);
insert into checkout0 values('2021-01-07 22:23:47.852', 2, 10, 25.45, 117);
insert into checkout0 values('2021-01-07 22:26:57.852', 2, 18, 72.22, 196);
insert into checkout0 values('2021-01-07 22:31:07.852', 2, 11, 58.17, 109);
insert into checkout0 values('2021-01-07 22:33:47.852', 2, 14, 47.77, 87);
insert into checkout0 values('2021-01-07 22:35:49.852', 2, 9, 27.32, 84);
```



```
insert into checkout0 values('2021-01-07 22:37:41.852', 2, 15, 46.16, 122);
insert into checkout0 values('2021-01-07 22:20:07.852', 3, 8, 18.82, 64);
insert into checkout0 values('2021-01-07 22:24:17.852', 3, 8, 19.54, 74);
insert into checkout0 values('2021-01-07 22:27:37.852', 3, 16, 62.44, 131);
insert into checkout0 values('2021-01-07 22:31:37.852', 3, 13, 51.87, 119);
insert into checkout0 values('2021-01-07 22:34:17.852', 3, 12, 37.65, 89);
insert into checkout0 values('2021-01-07 22:36:19.852', 3, 11, 28.23, 86);
insert into checkout0 values('2021-01-07 22:38:11.852', 3, 18, 65.26, 128);
```

4. これで、レジ精算システムのデータを使用する以下の例を実行する準備ができました。

すべてのウィンドウ関数と同様、結果のデフォルトの順序は、**OVER** 句で指定されている <partition 列>、<order 列> になります。この順序は、外部 **ORDER BY** を使用することによって変更できます。

次のクエリは、各 LaneNo が 7 行あることを示します。COUNT(*) を使用して、各行に 1 から 7 の番号を付けることができます。

```
SELECT LaneNo, StartTime, RowNumInPart FROM (select LaneNo, StartTime, COUNT(*) OVER (PARTITION BY LaneNo ORDER BY StartTime) FROM checkout0) AS T(LaneNo, StartTime, RowNumInPart);
```

次のクエリでは、LAG 関数を使用して、現在の行から 3 行前にある行の、RowNumInPart 列で示される値を取得する方法を示します。各パーティションが少なくとも 3 行になるまでは 0 が返されることに注目してください。

```
SELECT LaneNo, StartTime, RowNumInPart, LAG(RowNumInPart, 3, 0) OVER (PARTITION BY LaneNo ORDER BY StartTime) FROM (select LaneNo, StartTime, COUNT(*) OVER (PARTITION BY LaneNo ORDER BY StartTime) FROM checkout0) AS T(LaneNo, StartTime, RowNumInPart);
```

次のクエリは、LAG のデフォルトのオフセットである 1 行を使用して、各パーティション内の連続した 2 つの行ごとの時間差（秒単位）を表示します。

```
SELECT LaneNo, StartTime, DATEDIFF (SECOND, LAG(StartTime) OVER (PARTITION BY LaneNo ORDER BY StartTime), StartTime) AS Diff_in_secs FROM checkout0;
```

次のクエリは、レーン 1 の精算を前の精算と比較することで、精算ごとの品目数の変化を経時的に表示する方法を示します。

```
SELECT StartTime, LaneNo, Items, Items-LAG(Items) OVER (PARTITION BY LaneNo ORDER BY StartTime) AS delta_items FROM checkout0 WHERE LaneNo = 1;
```

LEAVE

備考

LEAVE ステートメントにより、ブロックまたはループ ステートメントを抜けて実行を継続します。LEAVE ステートメントは、ストアド プロシージャまたはトリガーの本体で使用できます。

例

次の例では、変数 `vInteger` の値が 1 ずつ増え、値が 11 になると、ループが LEAVE ステートメントで終了します。

```
TestLoop:
LOOP
IF (:vInteger > 10) THEN
LEAVE TestLoop;
END IF;
SET :vInteger = :vInteger + 1;
END LOOP;
```

関連項目

[IF](#)

[LOOP](#)

LIKE、ILIKE と ESCAPE の使用

LIKE および ILIKE はどちらも、文字ベースの列データ内でのパターン検索を可能にします。LIKE は大文字と小文字を区別するのに対し、ILIKE は大文字と小文字を区別しないという点が異なります。どちらも、ESCAPE 句を使用してエスケープ文字を指定できます。

構文

```
WHERE 式 [ NOT ] < LIKE | ILIKE > [ N ] 値 | ? [ エスケープ句 ]
エスケープ句 := ESCAPE [ N ] エスケープ文字
エスケープ文字 := ' 文字' | '' | ?
```

備考

LIKE 演算子または ILIKE 演算子の右辺のパターン値は、次のいずれかを指定する必要があります。

- 一致させる単純な文字列定数。
- 実行時に提供される動的パラメーター。これは、一致させる文字列を疑問符で示します。パラメーターはストアド プロシージャ内に配置できません。SQL Editor は動的パラメーターをサポートしていませんが、アプリケーション コードはサポートしています。
- USER キーワード。

パターン値では、パーセント記号ワイルドカードを使用して、列値内の 0 個以上の文字と一致させることができます。1 文字と一致させるには、アンダースコア ワイルドカードを使用します。どちらのワイルドカードも、パターン内で複数回使用できます。

次の表に、ワイルドカードやその他の特殊文字の使用に関する詳細を示します。

文字	用途
パーセント記号 "%"	0 個以上の文字と一致するワイルドカード。たとえば、Dan% は Dan、Daniel、Danny と一致します。
アンダースコア " _ "	1 つの文字と一致するワイルドカード。

文字	用途
円記号 "¥"	デフォルトのエスケープ文字。ワイルドカード文字または別のエスケープ文字が続く場合、その文字を文字自体と一致するリテラルとして扱います。たとえば、列値内の "%" と一致させるには、パーセント記号の前に円記号を使用します ("¥%")。ただし、値パターン内のエスケープ文字の後にワイルドカード文字または別のエスケープ文字以外の文字が続く場合、エスケープ文字の効果はありません。
2 つの一重引用符 "'"	間にスペースを入れない 2 つの一重引用符には、次の 2 つの用途があります。 - 結果文字列内の一重引用符を照合します。たとえば、データベースの列に <code>Jim's house</code> という値が含まれている場合、<code>WHERE</code> 句で <code>LIKE 'Jim''s house'</code> を指定すると、このパターンと一致させることができます。 <code>ESCAPE</code> キーワードの後に使用してヌル値を指定します。これにより、円記号がエスケープ文字として無効になり、他の文字と同じようにパターン値で使えるようになります。 メモ： 二重引用符はパターン文字列内では特殊文字でないため、すべての文字や数字と同様に使用できます。

`ESCAPE` 文字は次のいずれかになります。

- 単一文字。データベースで使われる任意のコード ページから指定できます。
- 2 つの一重引用符 `"`。ヌル エスケープ文字を指定します。
- 疑問符。アプリケーション コードによって提供される動的パラメーターを示します。

Unicode での `ESCAPE`

パターン値とエスケープ文字の前にオプションの `N` プレフィックスがある場合は、`N` で始まるデータ型 (`NCHAR`、`NLONGVARCHAR`、`NVARCHAR` など) に Unicode 文字を使用することを示します。`N` プレフィックスがない場合、`ESCAPE` を指定した `LIKE` および `ILIKE` でパターン値を Unicode 列と比較すると、結果が不完全または不正確になる可能性があります。

LIKE の例

このトピックでは LIKE の使用方法を説明します。ILIKE との比較のために、[ILIKE の例](#)を参照してください。

次の例は、長さ 5 文字、かつ中間の 3 文字が `abc` であるすべての列値と一致します。

```
SELECT Building_Name FROM Room WHERE Building_Name LIKE '_abc_'
```

次の例は、円記号を含むすべての列値と一致します。

```
SELECT Building_Name FROM Room where Building_Name LIKE '%¥%'
```

次の例は、パーセント記号で始まる列値**以外**のすべての列値と一致します。

```
SELECT Building_Name FROM Room where Building_Name NOT LIKE '%%%'
```

次の例は、1 つ以上の一重引用符を含むすべての列値と一致します。

```
SELECT Building_Name FROM Room where Building_Name LIKE '%''%'
```

次の例は、2 番目の文字が二重引用符であるすべての列値と一致します。

```
SELECT Building_Name FROM Room WHERE Building_Name LIKE '_%'
```

次の例では、`Building_Name` 列が入力変数 `:rname` に格納された文字列を含み、`Type` 列が入力変数 `:rtype` に格納された文字列を含む行を返す、ストアド プロシージャが作成されます。

```
CREATE PROCEDURE room_test(IN :rname CHAR(20), IN :rtype CHAR(20))
RETURNS(Building_Name CHAR(25), "Type" CHAR(20));
BEGIN
DECLARE :like1 CHAR(25);
DECLARE :like2 CHAR(25);
SET :like1 = '%' + :rname + '%';
SET :like2 = '%' + :rtype + '%';
SELECT Building_Name, "Type" FROM Room WHERE Building_Name LIKE :like1 AND "Type" LIKE :like2;
END;
```

次のステートメントを上記のストアド プロシージャ内に記述した場合、LIKE 演算子の右辺の式が単純な定数でないことが原因で、構文エラーが発生します。

次の構文は正しくないため、失敗します。

```
SELECT Building_Name, "Type" from Room WHERE Building_Name LIKE '%' + :rname + '%';
```

ILIKE の例

次の 2 つの LIKE と ILIKE の例は、Demodata サンプル データベースから同じ結果を返します。ILIKE ステートメントでは、UPPER 文字列関数を使用する必要はありません。

```
SELECT First_Name, Last_Name, Email_Address from Person
WHERE UPPER(Email_Address) LIKE '%@BTU.EDU%';
```

```
SELECT First_Name, Last_Name, Email_Address from Person
WHERE Email_Address ILIKE '%@btu.edu%';
```

メモ：最適化のために、ILIKE は大文字と小文字を区別する列に作成されたインデックスを使用できません。

ESCAPE を使用した LIKE または ILIKE の例

次の ESCAPE を使用した LIKE の例では、列値のパス名内の円記号をエスケープするためにエスケープ文字として感嘆符 "!" を指定し、円記号がデフォルトのエスケープ文字として扱われないようにしています。

```
CREATE TABLE backup_log (path VARCHAR(50), backup_file VARCHAR(50) );
INSERT INTO backup_log VALUES ('C:¥Data¥', 'Backup240420');
INSERT INTO backup_log VALUES ('D:¥Data¥', 'Backup240421');
SELECT path + backup_file as C_Files FROM backup_log WHERE path LIKE 'C:!¥%' ESCAPE '!';
```

結果：

```
C_Files
=====
C:¥Data¥Backup240420
```

関連項目

[SELECT](#)

LOOP

備考

LOOP ステートメントは、ステートメント ブロックの実行を繰り返します。

これはストアド プロシージャとトリガーでのみ使用できます。

Zen では、事後条件ループ REPEAT ... UNTIL はサポートされていません。

例

次の例では、変数 `vInteger` の値が 1 ずつ増え、値が 11 になるとループが終了します。

```
TestLoop:
LOOP
IF (:vInteger > 10) THEN
LEAVE TestLoop;
END IF;
SET :vInteger = :vInteger + 1;
END LOOP;
```

関連項目

[CREATE PROCEDURE](#)

[CREATE TRIGGER](#)

[IF](#)

NOT

備考

NOT キーワードと EXISTS キーワードを一緒に使用すると、サブクエリの結果に行が存在しないことを検査できます。外部クエリが評価するすべての行について、Zen はサブクエリ内に関連する行が存在するかどうかを検査します。Zen は、外部クエリの結果のうち、サブクエリ内の関連する行に対応する各行を、ステートメントの結果テーブルから除外します。

NOT と IN 演算子を組み合わせることにより、外部クエリの結果が、サブクエリの結果に含まれないかどうかを検査できます。結果テーブルには、外部クエリが返す行で、サブクエリ内に関連する行が存在しないものだけが含まれます。

例

次のステートメントは、どの講座にも登録されていない学生のリストを返します。

```
SELECT * FROM Person p WHERE NOT EXISTS
(SELECT * FROM Student s WHERE s.id = p.id
AND Cumulative_Hours = 0)
```

IN を使って、このステートメントを書き直すことができます。

```
SELECT * FROM Person p WHERE p.id NOT IN
(SELECT s.id FROM Student s WHERE Cumulative_Hours = 0)
```

関連項目

[SELECT](#)

[EXISTS](#)

[IN](#)

OPEN

構文

OPEN カーソル名

カーソル名 ::= ユーザー定義名

備考

OPEN ステートメントにより、カーソルが開きます。カーソルを開くには、事前にカーソルが定義されていなければなりません。

カーソルと変数はストアド プロシージャとトリガー内部でのみ許可されるため、このステートメントはストアド プロシージャまたはトリガーの内部でのみ使用できます。

例

次の例では、宣言されたカーソル BTUCursor が開きます。

```
DECLARE BTUCursor CURSOR
FOR SELECT Degree, Residency, Cost_Per_Credit FROM Tuition ORDER BY ID;
OPEN BTUCursor;
```

```
CREATE PROCEDURE MyProc(IN :CourseName CHAR(7)) AS
BEGIN
DECLARE cursor1 CURSOR
FOR SELECT Degree, Residency, Cost_Per_Credit FROM Tuition ORDER BY ID;
(追加コードはここに記述します)
OPEN cursor1;
FETCH cursor1 INTO :CourseName;
(追加コードはここに記述します)
CLOSE cursor1;
(追加コードはここに記述します)
END
```

関連項目

[CREATE PROCEDURE](#)

[CREATE TRIGGER](#)

[DECLARE CURSOR](#)

PARTIAL

備考

255 バイトを超える CHAR 列および VARCHAR 列でのインデックス作成を可能にするには、CREATE INDEX ステートメントに PARTIAL キーワードを記述します。部分インデックスを構成する列に格納されるバイト数が、オーバーヘッドも含めて 255 バイトより少なければ、PARTIAL は無視されます。

メモ： DROP INDEX ステートメントでは、部分インデックスを削除する場合に PARTIAL を必要としません。

UNIQUE と PARTIAL は相互に排他的であるため、1 つの CREATE INDEX 内で使用することはできません。

関連項目

[CREATE INDEX](#)

[DROP INDEX](#)

[UNIQUE](#)

PRIMARY KEY

備考

テーブル定義に主キーを追加するには、**ADD** 句で **PRIMARY KEY** を使用します。主キーは、ヌル値を含まない、重複のないインデックスです。主キーを指定すると、Zen によって、定義された列のグループに対して、指定された属性で重複のないインデックスが作成されます。

テーブルに複数の主キーがあってはならないので、既に主キーが定義されているテーブルに主キーを追加することはできません。テーブルの主キーを変更するには、**ALTER TABLE** ステートメントで **DROP** 句を使用して既存のキーを削除し、新しい主キーを追加します。

メモ： 主キーを追加したり、その他の参照整合性 (RI) 操作を行ったりするには、その前にデータベース名を使用してデータベースにログインする必要があります。

テーブル定義に主キーを追加するには、**ALTER TABLE** ステートメントの **ADD** 句に **PRIMARY KEY** を含めます。

主キーを追加する前に、主キー列リスト内の列が **NOT NULL** として定義されていることを確認する必要があります。主キーは重複のないインデックスであるため、ヌル値を許可しない列でのみ作成することができます。

ヌル値を許可しない列に基づく重複のないインデックスが既に存在する場合、**ADD PRIMARY KEY** によって別の重複のないインデックスは作成されません。代わりに、既存の重複のないインデックスが主キーになります。たとえば、次のステートメントでは、**T1_C1C2** という名前付きインデックスが主キーになります。

```
CREATE TABLE t1 (c1 INT NOT NULL, c2 CHAR(10) NOT NULL)
CREATE UNIQUE INDEX t1_c1c2 ON t1(c1,c2)
ALTER TABLE t1 ADD PRIMARY KEY(c1, c2)
```

このような主キーを削除した場合、その主キーは重複のないインデックスに切り替わります。

```
ALTER TABLE t1 DROP PRIMARY KEY
```

ヌル値を許可しない列に基づく重複のないインデックスがテーブル内に存在しない場合は、**ADD PRIMARY KEY** によって、ヌル値を許可しない列に基づいて重複のないインデックスが作成されます。**DROP PRIMARY KEY** によって、重複のないインデックスは完全に削除されます。

テーブル定義に主キーを追加するには、`CREATE TABLE` ステートメントに `PRIMARY KEY` 句を含めます。

データベースに参照制約を定義するには、`PRIMARY KEY` 句を使用して親テーブルの主キーを指定する必要があります。主キーは、1 つまたは複数の列で構成することができますが、ヌルではない列に対してのみ定義できます。指定する列は、`CREATE TABLE` ステートメントの列定義リストに含まれている必要もあります。

主キーを指定すると、Zen によって、定義された列のグループに対して、指定された属性でインデックスが作成されます。`CREATE TABLE` ステートメント内で、列が `NOT NULL` として明確に定義されていない場合、Zen はその列を強制的にヌル値を許可しない列にします。また、Zen はその列に基づいて重複のないインデックスを作成します。

たとえば、次の 2 つのステートメントは同じ結果になります。

```
CREATE TABLE t1 (c1 INT, c2 CHAR(10), PRIMARY KEY(c1,c2))
CREATE TABLE t1 (c1 INT NOT NULL, c2 CHAR(10) NOT NULL, PRIMARY KEY(c1,c2))
```

例

次のステートメントは、`Faculty` というテーブルに主キーを定義します。

```
ALTER TABLE Faculty ADD PRIMARY KEY (ID)
```

`ID` 列は、ヌル値を含まない重複のないインデックスとして定義されています。

関連項目

[ALTER TABLE](#)

[CREATE TABLE](#)

PRINT

備考

PRINT を使用して、変数の値または定数を印刷します。PRINT ステートメントは Windows ベースのプラットフォームにのみ適用されます。ほかのオペレーティングシステムのプラットフォームでは無視されます。

PRINT はストアド プロシージャ内でのみ使用できます。

例

次の例では、変数 :myvar の値が印刷されます。

```
PRINT( :myvar );
```

```
PRINT 'MYVAR = ' + :myvar;
```

次の例では、テキスト文字列の後に数値が印刷されます。数値はテキスト文字列に変換して印刷しなければなりません。

```
PRINT 'History 101 に登録されている生徒数 : ' + convert(:int_val, SQL_CHAR);
```

Windows Vista のリリース前は、[デスクトップとの対話をサービスに許可] を設定したローカル システム アカウントを使用して Zen がサービスとして実行されている場合には、ストアド プロシージャ内で PRINT を使用してダイアログ ボックスに出力を送ることができました。Windows Vista リリース以降、オペレーティング システムはこの出力を許可しなくなりました。次の回避策に示すように、値を文字列に変換して、SELECT ステートメントでそれを返すことができます。

```
DROP PROCEDURE varsub2;
CREATE PROCEDURE varsub2 ()
RETURNS (TestString CHAR(25));
DECLARE :vInteger INT;
DECLARE :tstring CHAR(25);
SET :vInteger = 0;
BEGIN
WHILE (:vInteger < 10) DO
SET :vInteger = :vInteger + 1;
END WHILE;
SET :tstring = 'The counter value is = ' + convert(:vInteger, SQL_CHAR);
SELECT :tstring;
END;
Call varsub2;
```

関連項目

[CREATE PROCEDURE](#)

PUBLIC

備考

PUBLIC キーワードを FROM に記述すると、Create Table 権が明示的に割り当てられていない全ユーザーからこの権限を取り消すことができます。

FROM 句には、アクセス権を取り消すグループまたはユーザーを指定します。単一の名前または名前のリストを指定することもできますが、PUBLIC キーワードを記述すれば、権限が明示的に割り当てられていない全ユーザーのアクセス権を取り消すことができます。

例

辞書内の全ユーザーにアクセス権を割り当てるには、次の例のように PUBLIC キーワードを使用して PUBLIC グループにアクセス権を付与します。

```
GRANT SELECT ON Course TO PUBLIC
```

このステートメントは、辞書内で定義されている全ユーザーに、Course テーブルの Select 権を割り当てます。後で PUBLIC グループの Select 権を取り消した場合、Select 権が明示的に付与されたユーザーだけがテーブルにアクセスできます。

次のステートメントでは、PUBLIC キーワードを使用して、辞書内で定義されている全ユーザーに Create Table 権を付与します。

```
GRANT CREATETAB TO PUBLIC
```

関連項目

[GRANT](#)

[REVOKE](#)

RELEASE SAVEPOINT

RELEASE SAVEPOINT ステートメントを使用して、セーブポイントを削除します。

構文

```
RELEASE SAVEPOINT セーブポイント名
```

```
セーブポイント名 ::= ユーザー定義名
```

備考

AUTOCOMMIT がオフの場合、RELEASE、ROLLBACK、および SAVEPOINT はセッションレベル（ストアードプロシージャの外部）でのみサポートされます。そうでない場合、RELEASE、ROLLBACK、および SAVEPOINT はストアードプロシージャ内で使用しなければなりません。

ストアードプロシージャ内でコミットされたステートメントは、呼び出し元の SQL アプリケーションの一番外側のトランザクションによって制御されます。

例

次の例では、SAVEPOINT を設定し、その後で条件を調べて SAVEPOINT を ROLLBACK するか RELEASE するかが判断されます。

```
CREATE PROCEDURE Enroll_student( IN :student ubigint, IN :classnum INTEGER);  
  
BEGIN  
    DECLARE :CurrentEnrollment INTEGER;  
    DECLARE :MaxEnrollment INTEGER;  
    SAVEPOINT SP1;  
    INSERT INTO Enrolls VALUES (:student, :classnum, 0.0);  
    SELECT COUNT(*) INTO :CurrentEnrollment FROM Enrolls WHERE class_id = :classnum;  
    SELECT Max_size INTO :MaxEnrollment FROM Class WHERE ID = :classnum;  
    IF :CurrentEnrollment >= :MaxEnrollment THEN  
        ROLLBACK TO SAVEPOINT SP1;  
    ELSE  
        RELEASE SAVEPOINT SP1;  
    END IF;  
END;
```

COUNT(式) によって、述部にある式の非ヌル値がすべてカウントされるということを覚えておいてください。COUNT(*) ではヌル値を含むすべての値がカウントされます。

関連項目

[CREATE PROCEDURE](#)

[ROLLBACK](#)

[SAVEPOINT](#)

RESTRICT

備考

RESTRICT を指定した場合、Zen は DELETE RESTRICT 規則を使用します。外部キー値が親テーブルの行を参照している場合、ユーザーはその行を削除できません。

削除規則を指定しない場合、Zen はデフォルトで RESTRICT 規則を適用します。

関連項目

[ALTER TABLE](#)

REVOKE

REVOKE ステートメントにより、ユーザー ID を削除し、セキュリティで保護されているデータベースにおける特定のユーザーの権限を取り消します。REVOKE ステートメントを使って、CREATE TABLE、CREATE VIEW、および CREATE PROCEDURE の権限を取り消すことができます。

構文

REVOKE CREATETAB | CREATEVIEW | CREATESP FROM *public* またはユーザー / グループ名 [, *public* またはユーザー / グループ名]...

REVOKE LOGIN FROM ユーザー名 [, ユーザー名]...

REVOKE 権限 **ON** < * | [**TABLE**] テーブル名 [オーナー ネーム] | **VIEW** ビュー名 | **PROCEDURE** ストアド プロシージャ名 > **FROM** ユーザー / グループ名 [, ユーザー / グループ名]...

* ::= すべてのオブジェクト（つまり、すべてのテーブル、ビュー、およびストアド プロシージャ）

権限 ::= **ALL**

```

| SELECT [ ( テーブルの列名 [ , テーブルの列名 ]... ) ]
| UPDATE [ ( テーブルの列名 [ , テーブルの列名 ]... ) ]
| INSERT [ ( テーブルの列名 [ , テーブルの列名 ]... ) ]
| UNMASK [ ( テーブルの列名 [ , テーブルの列名 ]... ) ]
| DELETE
| ALTER
| REFERENCES
| EXECUTE
```

テーブル名 ::= ユーザー定義のテーブル名

ビュー名 ::= ユーザー定義のビュー名

ストアド プロシージャ名 ::= ユーザー定義のストアド プロシージャ名

public またはユーザー / グループ名 ::= **PUBLIC** | ユーザー / グループ名

ユーザー / グループ名 ::= ユーザー名 | グループ名

ユーザー名 ::= ユーザー定義のユーザー名

グループ名 ::= ユーザー定義のグループ名

テーブルの列名 ::= ユーザー定義の列名（テーブルのみ）

次の表は、特定の操作に対する構文を示しています。

この操作の権限を取り消す場合	REVOKE で使用するキーワード
CREATE TABLE	CREATETAB
CREATE VIEW	CREATEVIEW

この操作の権限を取り消す場合	REVOKE で使用するキーワード
CREATE PROCEDURE	CREATESP

次の表は、ALL キーワードを使用した場合に取り消される権限を示しています。

ALL によって取り消される権限	テーブル	ビュー	ストアド プロシージャ
ALTER	X	X	X
DELETE	X	X	
INSERT	X	X	
REFERENCES	X		
SELECT	X	X	
UNMASK	X	X	
UPDATE	X	X	
EXECUTE			X

例

このセクションでは、UNMASK を除くさまざまな権限に対する REVOKE の使用例をいくつか示します。UNMASK 権限については、『*Advanced Operations Guide*』の [UNMASK 権限の付与と取り消し](#) を参照してください。

次のステートメントによって、dannyd からテーブル Class のすべての権限が取り消されます。

```
REVOKE ALL ON Class FROM 'dannyd'
```

次のステートメントでは、dannyd と rgarcia からテーブル Class のすべての権限が取り消されます。

```
REVOKE ALL ON Class FROM dannyd, rgarcia
```

次のステートメントによって、dannyd と rgarcia からテーブル Class の DELETE 権限が取り消されます。

```
REVOKE DELETE ON Class FROM dannyd, rgarcia
```

次の例では、keithv と miked からテーブル Class の INSERT 権限が取り消されます。

```
REVOKE INSERT ON Class FROM keithv, miked
```

次の例では、keithv と brendanb から、テーブル Person の列 First_name と Last_name の INSERT 権限が取り消されます。

```
REVOKE INSERT(First_name,Last_name) ON Person FROM keithv, brendanb
```

次のステートメントによって、dannyd からテーブル Class の ALTER 権限が取り消されます。

```
REVOKE ALTER ON Class FROM dannyd
```

次の例では、dannyd と rgarcia からテーブル Class の SELECT 権限が取り消されます。

```
REVOKE SELECT ON Class FROM dannyd, rgarcia
```

次のステートメントによって、dannyd と rgarcia から、テーブル Person の列 First_name と Last_name の SELECT 権限が取り消されます。

```
REVOKE SELECT(First_name, Last_name) ON Person FROM dannyd, rgarcia
```

次の例では、dannyd と rgarcia からテーブル Person の UPDATE 権限が取り消されます。

```
REVOKE UPDATE ON Person ON dannyd, rgarcia
```

次の例では、user1 から CREATE VIEW 権限が取り消されます。

```
REVOKE CREATEVIEW FROM user1;
```

次の例では、ストアド プロシージャ MyProc1 に対する user1 の EXECUTE 権限が取り消されます。

```
REVOKE EXECUTE ON PROCEDURE MyProc1 FROM user1;
```

次の例では、Demodata サンプルデータベースのセキュリティが有効になっており、ユーザー名 USAcctsMgr に対して、テーブル Person の ID 列に SELECT 権が付与されて

いるものとします。そのユーザーから、対象列の **SELECT** 権限を取り消します。以下のステートメントを使用します。

```
REVOKE SELECT ( ID ) ON Person FROM 'USAcctsMgr'
```

関連項目

[GRANT](#)

ROLLBACK

ROLLBACK は、データベースを現行トランザクションが開始される前の状態に戻します。このステートメントにより、最後に行った SAVEPOINT または START TRANSACTION 以降に取得したロックを解放します。

ROLLBACK TO SAVEPOINT は、指定されたセーブポイントまでトランザクションをロールバックします。

構文

```
ROLLBACK [ WORK ] [ TO SAVEPOINT セーブポイント名 ]
```

セーブポイント名 ::= ユーザー定義名

備考

AUTOCOMMIT がオフの場合、ROLLBACK、SAVEPOINT、および RELEASE はセッション レベル（ストアード プロシージャの外部）でのみサポートされます。そうでない場合、ROLLBACK、SAVEPOINT、および RELEASE はストアード プロシージャ内で使用しなければなりません。

ストアード プロシージャ内でコミットされたステートメントは、呼び出し元の SQL アプリケーションの一番外側のトランザクションによって制御されます。

ネストされたトランザクションの場合、ROLLBACK は一番近い START TRANSACTION までロールバックします。たとえば、トランザクションが 5 レベルまでネストされている場合、すべてのトランザクションを元に戻すには 5 つの ROLLBACK ステートメントが必要になります。トランザクションは、コミットされるかロールバックされるかのいずれかで、両方はされません。つまり、コミットされたトランザクションをロールバックすることはできません。

例

次のステートメントによって、トランザクションの開始以降に行われたデータベースの変更を元に戻します。

```
ROLLBACK WORK;
```

次のステートメントによって、Svpt1 と名付けられた最後のセーブポイント以降に行われたデータベースの変更を元に戻します。

```
ROLLBACK TO SAVEPOINT Svpt1;
```

関連項目

[COMMIT](#)

[RELEASE SAVEPOINT](#)

[SAVEPOINT](#)

SAVEPOINT

SAVEPOINT は、ロールバック可能なトランザクション内のポイントを定義します。

構文

SAVEPOINT セーブポイント名

セーブポイント名 ::= ユーザー定義名

備考

AUTOCOMMIT がオフの場合、ROLLBACK、SAVEPOINT、および RELEASE はセッションレベル（ストアド プロシージャの外部）でのみサポートされます。そうでない場合、ROLLBACK、SAVEPOINT、および RELEASE はストアド プロシージャ内で使用しなければなりません。

ストアド プロシージャ内でコミットされたステートメントは、呼び出し元の SQL アプリケーションの一番外側のトランザクションによって制御されます。

SAVEPOINT は、それが定義されているプロシージャにのみ適用されます。つまり、別のプロシージャ内に定義されている SAVEPOINT は参照できません。

例

次の例では、SAVEPOINT を設定し、その後で条件を調べて SAVEPOINT を ROLLBACK するか RELEASE するかが判断されます。

```
CREATE PROCEDURE Enroll_student( IN :student ubigint, IN :classnum INTEGER);
BEGIN
    DECLARE :CurrentEnrollment INTEGER;
    DECLARE :MaxEnrollment INTEGER;
    SAVEPOINT SP1;
    INSERT INTO Enrolls VALUES (:student, :classnum, 0.0);
    SELECT COUNT(*) INTO :CurrentEnrollment FROM Enrolls WHERE class_id = :classnum;
    SELECT Max_size INTO :MaxEnrollment FROM Class WHERE ID = :classnum;
    IF :CurrentEnrollment >= :MaxEnrollment THEN
        ROLLBACK TO SAVEPOINT SP1;
    ELSE
        RELEASE SAVEPOINT SP1;
    END IF;
END;
```

COUNT(式) によって、述部にある式の非ヌル値がすべてカウントされるということを覚えておいてください。COUNT(*) ではヌル値を含むすべての値がカウントされます。

関連項目

[COMMIT](#)

[CREATE PROCEDURE](#)

[RELEASE SAVEPOINT](#)

[ROLLBACK](#)

SELECT

指定された情報をデータベースから取り出します。SELECT ステートメントにより、テンポラリ ビューを作成します。

構文

```
クエリ スペック [ UNION [ ALL ] クエリ スペック ]...
[ ORDER BY order-by 式 [ , order-by 式 ]... ] [ Limit 句 ] [ FOR UPDATE ]
```

```
クエリ スペック ::= ( クエリ スペック )
| SELECT [ ALL | DISTINCT ] [ top 句 ] 選択リスト
FROM テーブル参照 [ , テーブル参照 ]...
[ WHERE 検索条件 ]
[ GROUP BY 式 [ , 式 ]...
[ HAVING 検索条件 ] ]
```

```
式またはサブクエリ ::= 式 | ( クエリ スペック ) [ ORDER BY order-by 式
[ , order-by 式 ]... ] [ Limit 句 ]
```

```
サブクエリ式 ::= ( クエリ スペック ) [ ORDER BY order-by 式
[ , order-by 式 ]... ] [ Limit 句 ]
```

```
order-by 式 ::= 式 [ CASE (文字列) | COLLATE 照合順序名 ] [ ASC | DESC ]
```

```
Limit 句 ::= [ LIMIT [ オフセット , ] 行数 | 行数 OFFSET オフセット | ALL [ OFFSET オフセット ] ]
```

```
オフセット ::= 数値 | ?
```

```
行数 ::= 数値 | ?
```

```
top 句 ::= TOP または LIMIT 個数
```

```
選択リスト ::= * | 選択項目 [ , 選択項目 ]...
```

```
選択項目 ::= 式 [ [ AS ] エイリアス名 ] | テーブル名 . *
```

```
テーブル参照 ::= { OJ 外部結合定義 }
| [ データベース名 . ] テーブル名 [ [ AS ] エイリアス名 ] [ WITH ( テーブル ヒント ) ]
| [ データベース名 . ] ビュー名 [ [ AS ] エイリアス名 ]
| dbo.f システム カタログ関数名 [ [ AS ] エイリアス名 ]
| 結合定義
| ( 結合定義 )
| ( テーブルサブクエリ ) [ AS ] エイリアス名 [ ( 列名 [ , 列名 ]... ) ]
```

```
外部結合定義 ::= テーブル参照 外部結合タイプ JOIN テーブル参照
```

```
ON 検索条件
```

```
外部結合タイプ ::= LEFT [ OUTER ] | RIGHT [ OUTER ] | FULL [ OUTER ]
```

```
テーブル ヒント ::= INDEX ( インデックス値 [ , インデックス値 ]... )
```

```
インデックス値 ::= 0 | インデックス名
```

```
インデックス名 ::= ユーザー定義名
```

結合定義 ::= テーブル参照 [結合タイプ] **JOIN** テーブル参照 **ON** 検索条件
 | テーブル参照 **CROSS JOIN** テーブル参照
 | 外部結合定義

結合タイプ ::= **INNER** | **LEFT** [**OUTER**] | **RIGHT** [**OUTER**] | **FULL** [**OUTER**]

テーブル サブクエリ ::= クエリ スペック [[**UNION** [**ALL**]
 クエリ スペック]...] [**ORDER BY** order-by 式 [, order-by 式]...]

検索条件 ::= 検索条件 **AND** 検索条件
 | 検索条件 **OR** 検索条件
 | **NOT** 検索条件
 | (検索条件)
 | 述部

述部 ::= 式 [**NOT**] **BETWEEN** 式 **AND** 式
 | 式またはサブクエリ 比較演算子 式またはサブクエリ
 | 式 [**NOT**] **IN** (クエリ スペック)
 | 式 [**NOT**] **IN** (値 [, 値]...)
 | 式 [**NOT**] **LIKE**、**ILIKE** と **ESCAPE** の使用 値
 | 式 **IS** [**NOT**] **NULL**
 | 式 比較演算子 **ANY** (クエリ スペック)
 | 式 比較演算子 **ALL** (クエリ スペック)
 | [**NOT**] **EXISTS** (クエリ スペック)

比較演算子 ::= < | > | <= | >= | = | <> | !=

式またはサブクエリ ::= 式 | (クエリ スペック)

値 ::= リテラル | **USER** | ?

式 ::= 式 - 式
 | 式 + 式
 | 式 * 式
 | 式 / 式
 | 式 & 式
 | 式 | 式
 | 式 ^ 式
 | (式)
 | - 式
 | + 式
 | 列名
 | ?
 | リテラル
 | セット関数
 | スカラー関数
 | { fn スカラー関数 }
 | ウィンドウ関数
 | **CASE** case 値式 **WHEN** when 式 **THEN** then 式 [...] [**ELSE**
 else 式] **END**
 | **COALESCE** (式, 式 [,...])
 | **IF** (検索条件, 式, 式)
 | **SQLSTATE**
 | サブクエリ式
 | **NULL**
 | : ユーザー定義名
 | **USER**
 | グローバル変数
 | 仮想列

case 値式 *when* 式, *then* 式 *else* 式 ::= **CASE (式)** を参照

サブクエリ式 ::= (クエリ スペック)

セット関数 ::= **AVG** ([**DISTINCT** | **ALL**] 式)
 | **COUNT** (< * | [**DISTINCT** | **ALL**] 式 >)
 | **COUNT_BIG** (< * | [**DISTINCT** | **ALL**] 式 >)
 | **LAG** (式 [, 式 [, 式]] *over* 句)
 | **MAX** ([**DISTINCT** | **ALL**] 式)
 | **MIN** ([**DISTINCT** | **ALL**] 式)
 | **STDEV** ([**DISTINCT** | **ALL**] 式)
 | **STDEVP** ([**DISTINCT** | **ALL**] 式)
 | **SUM** ([**DISTINCT** | **ALL**] 式)
 | **VAR** ([**DISTINCT** | **ALL**] 式)
 | **VARP** ([**DISTINCT** | **ALL**] 式)

スカラー関数 ::= **スカラー関数** を参照

グローバル変数 ::= **@:IDENTITY**
 | **@:ROWCOUNT**
 | **@@BIGIDENTITY**
 | **@@IDENTITY**
 | **@@ROWCOUNT**
 | **@@SPID**
 | **@@VERSION**

仮想列 ::= **SYS\$CREATE**
 | **SYS\$UPDATE**

ウィンドウ関数 ::= セット関数 *over* 句

over 句 ::= **OVER** ([*partition-by* 句] *order-by-in-over* 句 [*row* 句])

partition-by 句 ::= **PARTITION BY** 式 [, 式]...

order-by-in-over 句 ::= **ORDER BY** 式 [**ASC** | **DESC**] [, 式 [**ASC** | **DESC**]]...

row 句 ::= **ROWS** ウィンドウ フレームの範囲

ウィンドウ フレームの範囲 ::= { **UNBOUNDED PRECEDING**
 | 符号なし整数リテラル **PRECEDING**
 | **CURRENT ROW** }

メモ : **OVER** 句での **ORDER BY** の使用は、Zen SQL の他の場所での **ORDER BY** とは異なります。本リリースに適用される詳細および関連情報については、[SQL ウィンドウ関数](#)を参照してください。

備考

ここでは、SELECT の使用に関連する次のトピックについて説明します。

- [FOR UPDATE](#)
- [GROUP BY](#)
- [SQL ウィンドウ関数](#)
- [動的パラメーター](#)
- [エイリアス](#)
- [SUM と DECIMAL の精度](#)
- [サブクエリ](#)
- [テーブル ヒントの使用](#)
- [集計関数の DISTINCT](#)
- [TOP または LIMIT](#)
- [テーブル ヒントの例](#)
- [システム データ v2 のアクセス](#)

FOR UPDATE

SELECT FOR UPDATE は、クエリで選択されたテーブル内の 1 つまたは複数の行をロックします。レコード ロックは、次の COMMIT または ROLLBACK ステートメントが発行されたときに解放されます。

この競合を避けるため、SELECT FOR UPDATE は行を取得しているときロックします。

ステートメント レベル SQL_ATTR_CONCURRENCY が SQL_CONCUR_LOCK に設定されている場合、SELECT FOR UPDATE はトランザクション内で優先されます。

SQL_ATTR_CONCURRENCY が SQL_CONCUR_READ_ONLY に設定されている場合、データベース エンジンエラーを返しません。

SELECT FOR UPDATE は WAIT および NOWAIT キーワードをサポートしません。

SELECT FOR UPDATE は、短時間で行をロックできない場合（20 回再試行）、ステータス コード [84 : レコードまたはページはロックされています](#)。を返します。

制約

SELECT FOR UPDATE ステートメントには以下の制約があります。

- トランザクション内のみで有効です。トランザクション外で使用された場合、そのステートメントは無視されます。
- 単一のテーブルのみでサポートされます。SELECT FOR UPDATE は、JOIN や単純でないビュー、また、GROUP BY、DISTINCT、UNION キーワードと共に使用できません。
- CREATE VIEW ステートメント内では使用できません。

GROUP BY

Zen では、列リストの GROUP BY がサポートされているほか、式リストや、GROUP BY 式リスト内のあらゆる式に対する GROUP BY がサポートされています。GROUP BY 拡張の詳細については、[GROUP BY](#) を参照してください。GROUP BY のない HAVING はサポートされていません。

次の特性のいずれかを備えた SELECT ステートメントを実行することによって生成される結果セットおよびストアド ビューは、読み取り専用になります（更新できません）。つまり、位置付け UPDATE や位置付け DELETE、および SQLSetPos 呼び出しを使用して、結果セットあるいはストアド プロシージャでデータを追加、変更、削除することは許可されません。

- 選択リストに集計を含んでいる
`SELECT SUM(c1) FROM t1`
- 選択リストに DISTINCT を指定している
`SELECT DISTINCT c1 FROM t1`
- ビューに GROUP BY を使用している
`SELECT SUM(c1), c2 FROM t1 GROUP BY c2`
- ビューは結合である（複数のテーブルを参照する）
`SELECT * FROM t1, t2`
- ビューに UNION 演算子を使用しており、UNION ALL が指定されていないか、もしくはすべての SELECT ステートメントが同じテーブルを参照していない
`SELECT c1 FROM t1 UNION SELECT c1 FROM t1`
`SELECT c1 FROM t1 UNION ALL SELECT c1 FROM t2`
ストアド ビューでは UNION 演算子を使用できないので注意してください。
- ビューに、外部クエリ内のテーブル以外のテーブルを参照するサブクエリを含んでいる
`SELECT c1 FROM t1 WHERE c1 IN (SELECT c1 FROM t2)`

SQL ウィンドウ関数

Zen は、ANSI 標準の SQL ウィンドウ使用のサブセットを提供します。本リリースにおけるこの初期導入には、一定の制限事項および考慮事項があります。

制限事項

OVER 句には次の制限があります。

- **SELECT** ステートメント内のすべての **OVER** 句は、それらの **PARTITION BY**、**ORDER BY**、および **ROWS** 句と一致している必要があります。**PARTITION BY** の式は同一かつ同順である必要があります、**ORDER BY** の式も同一かつ同順、**ROWS** 句は同一である必要があります。これらのうち **OVER** 句に存在しない句は、他の句にも存在しない必要があります。
- **OVER** 句には **ORDER BY** 句を含める必要があります。**PARTITION BY** 句は省略可能です。**PARTITION BY** が存在する場合、この句で **ORDER BY** 句と同じ列を使用してはいけません。**PARTITION BY** がない場合は、結果セット全体が 1 つのパーティションとして扱われます。
- **PARTITION BY** 句の **ROWS** の指定では、次のキーワードがサポートされています。
 - **UNBOUNDED**
 - *n* **PRECEDING**
 - **CURRENT ROW**
- **PARTITION BY** 句の **ROWS** の指定では、次のキーワードはサポートされていません。
 - **BETWEEN**
 - **FOLLOWING**
- **RANGE** キーワードはサポートされていません。
- **DISTINCT** キーワードは、セット関数ではサポートされていません。
- **OVER** 句の **ORDER BY** 句では、**COLLATE** の指定はサポートされていません。
- ウィンドウ関数は、前方のみのカーソルでしか使用できません。

考慮事項

ANSI SQL 標準では、特定の構文の組み合わせはデフォルトの **RANGE** セマンティクスを意味しますが、現在の Zen リリースでは **RANGE** はサポートされていません。したがって、現在の Zen リリースでは、デフォルトの **RANGE** 指定が **RANGE UNBOUNDED PRECEDING** である場合には、このデフォルトは **ROWS UNBOUNDED PRECEDING** として実装されます。

通常、これら 2 つのデフォルトの違いが結果セットに影響するのは、**PARTITION BY** 句と **ORDER BY** 句で返される列の値の組み合わせが行ごとに一意でない場合だけです。そのため、現在の Zen リリースでは、そのような列の値の組み合わせが行ごとに一意

でない場合は、**ROWS UNBOUNDED PRECEDING** を明示的に指定することをお勧めします。これにより期待される結果が返ります。

動的パラメーター

動的パラメーター（疑問符 (?) で表されます）は、**SELECT** 項目としてはサポートされていません。動的パラメーターが述部の一部であれば、**SELECT** ステートメント内で使用することができます。たとえば、**SELECT * FROM faculty WHERE id = ?** は、動的パラメーターが述部の一部なので有効です。

Zen Control Center の SQL Editor では、述部に動的パラメーターを含む SQL ステートメントを実行できないことに留意してください。

変数は、ストアドプロシージャ内でのみ **SELECT** 項目として使用することができます。[CREATE PROCEDURE](#) を参照してください。

エイリアス

WHERE、**HAVING**、**ORDER BY**、または **GROUP BY** 句にエイリアスを含めることができます。エイリアス名はテーブル内のどの列名とも異なっていなければなりません。次のステートメントは、**WHERE** 句と **GROUP BY** 句でエイリアス **a** および **b** を使用する例を示しています。

```
SELECT Student_ID a, Transaction_Number b, SUM (Amount_Owed) FROM Billing WHERE a < 120492810 GROUP BY a, b UNION SELECT Student_ID a, Transaction_Number b, SUM (Amount_Paid) FROM Billing WHERE a > 888888888 GROUP BY a, b
```

SUM と DECIMAL の精度

DECIMAL 型のフィールドで **SUM** 集計関数を使用する場合は、次の規則が適用されます。

- 結果の有効桁数は 74 ですが、小数部桁数は列定義に依存します。
- 74 を超える桁数の数値（本当に、非常に大きな数値です）が計算された場合、結果でオーバーフローエラーが発生することがあります。オーバーフローが発生した場合、値は返らず、**SQLSTATE** に数値が範囲外であることを示す 22003 が設定されます。

サブクエリ

サブクエリは一種の `SELECT` ステートメントで、ステートメント内に 1 つ以上の `SELECT` ステートメントを含んでいます。サブクエリは、ステートメントの処理を進めるために値を生成します。一番上の `SELECT` ステートメントでサブクエリをネストできる最大数は 16 です。

サポートされるサブクエリのタイプは次のとおりです。

- `comparison`
- `quantified`
- `in`
- `exists`
- `correlated`
- `expression`
- `table`

相関サブクエリ述部は、グループ化された列を参照する `HAVING` 句ではサポートされていません。

`expression` サブクエリは、`SELECT` リストの中でサブクエリを使用できます。たとえば、`SELECT (SELECT SUM(c1) FROM t1 WHERE t1.c2 = t1.(c2) FROM t2` のように記述できます。サブクエリの `SELECT` リストに指定できる項目は 1 つのみです。たとえば、次のステートメントは、サブクエリの `SELECT` リストに 2 つ以上の項目が含まれているため、エラーが返ります。`SELECT p.id, (SELECT SUM(b.amount_owed), SUM(b.amount_paid) FROM billing b) FROM person p`

式としてのサブクエリは相関でも非相関でもかまいません。相関サブクエリは、一番上のステートメントに指定されているテーブル内の 1 つ以上の列を参照します。非相関サブクエリは、一番上のステートメントに指定されているテーブル内の列を参照しません。次の例は、`WHERE` 句の相関サブクエリを示しています。

```
SELECT * FROM student s WHERE s.Tuition_id IN
(SELECT t.ID FROM tuition t WHERE t.ID = s.Tuition_ID);
```

メモ： テーブル サブクエリでは、非相関サブクエリはサポートされますが、相関サブクエリはサポートされません。

演算子 **IN**、**EXISTS**、**ALL**、または **ANY** と関連付けられているサブクエリは式と見なされません。

相関および非相関のサブクエリは、どちらも単一の値のみ返すことができます。このため、相関サブクエリおよび非相関サブクエリはスカラー サブクエリとも呼ばれます。

スカラー サブクエリは、**DISTINCT**、**GROUP BY**、および **ORDER BY** 句に含むことができます。

サブクエリは式の左辺で 사용할 ことができます。

式またはサブクエリ 比較演算子 式またはサブクエリ

ここで、式は 1 つの式、比較演算子は次のうちのいずれかです。

<	>	<=	>=	=
(より小さい)	(より大きい)	(小さいかまたは等しい)	(大きいまたは等しい)	(等しい)
<>	!=	LIKE	IN	NOT IN
(等しくない)	(等しくない)			

このセクションの残り部分では、以下の項目について説明します。

- サブクエリの最適化
- サブクエリ内の UNION
- テーブル サブクエリ

サブクエリの最適化

左辺のサブクエリの動作は、サブクエリが相関サブクエリでなく、すべての結合条件が 1 つの外部結合である場合の、**IN**、**NOT IN**、および **=ANY** について最適化されています。それ以外の条件は最適化されないことがあります。次は、これらの条件を満たすクエリの例です。

```
SELECT count(*) FROM person WHERE id IN (SELECT faculty_id FROM class)
```

Zen はインデックスに基づいてサブクエリを最適化するため、サブクエリでインデックスを使用するとパフォーマンスが向上します。たとえば、次のステートメント内のサブクエリは、**student_id** が **Billing** テーブルのインデックスであるため、この列によって最適化されます。

```
SELECT (SELECT SUM(b.amount_owed) FROM billing b WHERE b.student_id = p.id) FROM person p
```

サブクエリ内の UNION

1 つのサブクエリの中で、複数の異なる UNION グループをカッコで囲むことは許可されていません。カッコは各 SELECT ステートメントの中で使用できます。

たとえば、次のステートメントで、IN の後のカッコと最後のカッコは**許可されません**。

```
SELECT c1 FROM t5 WHERE c1 IN ( (SELECT c1 FROM t1 UNION SELECT c1 FROM t2) UNION ALL (SELECT c1 FROM t3 UNION SELECT c1 from t4) )
```

テーブル サブクエリ

テーブル サブクエリを使用すると、複数のクエリを組み合わせで 1 つの詳細なクエリにすることができます。テーブル サブクエリは、データベースに残らない動的ビューです。一番上の SELECT クエリが完了したら、テーブル サブクエリに関連付けられたすべてのリソースが解放されます。

メモ： テーブル サブクエリでは、非相関サブクエリのみが使用できます。相関サブクエリは使用できません。

次の改ページ調整の例（1 ページあたり 100 行ずつの 1500 行）は、ORDER BY キーワードを使用したテーブル サブクエリの使用方法を示しています。

最初の 100 行

```
SELECT * FROM ( SELECT TOP 100 * FROM ( SELECT TOP 100 * FROM person ORDER BY last_name asc ) AS foo ORDER BY last_name desc ) AS bar ORDER BY last_name ASC
```

2 番目の 100 行

```
SELECT * FROM ( SELECT TOP 100 * FROM ( SELECT TOP 200 * FROM person ORDER BY last_name asc ) AS foo ORDER BY last_name DESC ) AS bar ORDER BY last_name ASC
```

...

15 番目の 100 行

```
SELECT * FROM ( SELECT TOP 100 * FROM ( SELECT TOP 1500 * FROM person ORDER BY last_name ASC ) AS foo ORDER BY last_name DESC ) AS bar ORDER BY last_name ASC
```

テーブル ヒントの使用

テーブル ヒント機能を使用すると、クエリの最適化のために、どのインデックスを使用するかを指定することができます。テーブル ヒントは、データベース エンジンが使用するデフォルトのクエリ オプティマイザーより優先されます。

テーブル ヒントで `INDEX(0)` が指定されていると、エンジンは関連するテーブルのテーブル スキャンを実行します。テーブル スキャンは、インデックスを使用して特定のデータ要素を見つけるのではなく、テーブル内の各行を読み取ります。

テーブル ヒントに `INDEX(インデックス名)` が指定されている場合、エンジンは**インデックス名**を使用し、JOIN 条件の制約や、`DISTINCT`、`GROUP BY`、`ORDER BY` の使用に基づいてテーブルを最適化します。指定されたインデックスでテーブルが最適化できない場合、エンジンは既存のインデックスに基づいて最適化を試みます。

複数のインデックス名を指定した場合、エンジンは最適なパフォーマンスを得られるインデックスを選択するか、**OR** 最適化のために複数のインデックスを使用します。わかりやすくするために例を示します。以下を想定します。

```
CREATE INDEX ndx1 on t1(c1)
CREATE INDEX ndx2 on t1(c2)
CREATE INDEX ndx3 on t1(c3)
SELECT * FROM t1 WITH (INDEX (ndx1, ndx2)) WHERE c1 = 1 AND c2 > 1 AND c3 = 1
```

データベース エンジンは、`ndx2` を使用するのではなく `ndx1` を使用して、`c1 = 1` を最適化します。`ndx3` は、テーブル ヒントに含まれていないため、考慮に入れません。

以下について考えてみましょう。

```
SELECT * FROM t1 WITH (INDEX (ndx1, ndx2)) WHERE (c1 = 1 OR c2 > 1) AND c3 = 1
```

エンジンは `ndx1` と `ndx2` の両方を使用して (`c1 = 1 OR c2 > 1`) を最適化します。

テーブル ヒントの中で複数のインデックス名が表れる順序は重要ではありません。データベース エンジンは、指定されたインデックスから最も優れた最適化を行えるインデックスを選択します。

テーブル ヒント内で重複するインデックス名は無視されます。

結合されたビューでは、`FROM` 句の最後ではなく、適切なテーブル名の後にテーブル ヒントを指定します。たとえば、次は正しいステートメントです。

```
SELECT * FROM person WITH (INDEX(Names)), student WHERE student.id = person.id AND last_name LIKE 'S%'
```

これに対し、次のステートメントは**正しくありません**。

```
SELECT * FROM person, student WITH (INDEX(Names)) WHERE student.id = person.id AND last_name LIKE 'S%'
```

メモ： テーブル ヒント機能は上級ユーザー向けです。一般的に、データベース クエリ オプティマイザーが最も優れた最適化方法を選択するので、テーブル ヒントは必要ではありません。

テーブル ヒントの制限事項

- テーブル ヒントで利用できるインデックス名の最大数は、SQL ステートメントの最大長（64 KB）によってのみ制限されます。
- テーブル ヒント内のインデックス名は、テーブル名で完全に修飾されては**なりません**。

間違った SQL :	SELECT * FROM t1 WITH (INDEX(t1.ndx1)) WHERE t1.c1 = 1
戻り値 :	SQL_ERROR
szSqlState :	37000
メッセージ :	構文エラー : SELECT * FROM t1 WITH (INDEX(t1.<< ???>>ndx1)) WHERE t1.c1 = 1

- テーブル ヒントは、SELECT ステートメント内でビューと共に使用された場合は無視されます。

間違った SQL :	SELECT * FROM myt1view WITH (INDEX(ndx1))
戻り値 :	SQL_SUCCESS_WITH_INFO
szSqlState :	01000
メッセージ :	ビューと共に指定されたインデックス ヒントは無視されます。

- インデックス名以外ではゼロが唯一の有効なヒントです。

間違った SQL :	SELECT * FROM t1 WITH (INDEX(85))
戻り値 :	SQL_ERROR
szSqlState :	S1000
メッセージ :	インデックス ヒントが無効です。

- テーブル ヒント内のインデックス名には既存のインデックスを指定する必要があります。

間違った SQL :	SELECT * FROM t1 WITH (INDEX(ndx4))
戻り値 :	SQL_ERROR
szSqlState :	S0012
メッセージ :	インデックス名が無効です。インデックスが見つかりません。

- テーブル ヒントは、サブクエリ AS テーブルには指定できません。

間違った SQL :	SELECT * FROM (SELECT c1, c2 FROM t1 WHERE c1 = 1) AS a WITH (INDEX(ndx2)) WHERE a.c2 = 10
戻り値 :	SQL_ERROR
szSqlState :	37000
メッセージ :	構文エラー : SELECT * FROM (SELECT c1, c2 FROM t1 WHERE c1 = 1) AS a WITH<< ??? >>(INDEX(ndx2)) WHERE a.c2 = 10

システム データ v2 のアクセス

13.0 および 16.0 形式を使用するデータ ファイルでは、システム データ v2 により、レコード作成およびレコード更新のタイム スタンプに基づく **Btrieve** キーが有効になります。これらの作成キーおよび更新キーには以下の特性があります。

- 作成キーは既存の **Btrieve** システム キー 125 に取って代わり、トランザクション ログで使用されます。
- 更新キーによって、キー 125 の作成時刻のタイム スタンプなど、特定の時点以降に変更された行を特定することができます。更新キーは **Btrieve** システム キー 124 です。
- どちらのキーも、セブタ秒精度を持つ **TIMESTAMP(7)** 形式、**YYYY-MM-DD HH:MM:SS.ssssss** です。
- SQL クエリでは、列名 **sys\$create** および **sys\$update** を使用してタイム スタンプにアクセスできます。作業例については、[Sys\\$create および Sys\\$update を使用したクエリ](#)を参照してください。

例

この単純な SELECT ステートメントによって、Faculty テーブルの全データが取り出されます。

```
SELECT * FROM Faculty
```

このステートメントによって、person および faculty テーブルから、person テーブル内の id 列と faculty テーブル内の id 列が同じデータが取り出されます。

```
SELECT Person.id, Faculty.salary FROM Person, Faculty WHERE Person.id = Faculty.id
```

このセクションの残り部分では、SELECT ステートメントのさまざまな例を示します。次の見出しのうちのいくつかは、SELECT の構文定義に示される変数に基づいています。

- [FOR UPDATE](#)
- [概算数値リテラル](#)
- [BETWEEN 述語](#)
- [関連名](#)
- [正確な数値リテラル](#)
- [IN 述語](#)
- [セット関数](#)
- [日付リテラル](#)
- [時刻リテラル](#)
- [タイムスタンプ リテラル](#)
- [文字列リテラル](#)
- [日付演算](#)
- [IF](#)
- [複数データベースの結合](#)
- [左外部結合](#)
- [右外部結合](#)
- [カルテシアン結合](#)
- [Sys\\$create および Sys\\$update を使用したクエリ](#)

FOR UPDATE

次の例ではテーブル t1 を使用して FOR UPDATE の使用法を示します。t1 は Demodata サンプル データベースの一部であるとしします。ストアド プロシージャは SELECT FOR UPDATE ステートメントのためのカーソルを作成します。ループで t1 から c1=2 の行の各レコードをフェッチし、c1 の値を 4 に設定します。

プロシージャは、IN パラメーターとして値 2 を渡して呼び出されます。

この例は、2 人のユーザー A と B は Demodata にログインしているものとしします。ユーザー A が以下の処理を行います。


```

DROP TABLE t1
CREATE TABLE t1 (c1 INTEGER, c2 INTEGER)
INSERT INTO t1 VALUES (1,1)
INSERT INTO t1 VALUES (2,1)
INSERT INTO t1 VALUES (1,1)
INSERT INTO t1 VALUES (2,1)
CREATE PROCEDURE p1 (IN :a INTEGER)
AS
BEGIN
    DECLARE :b INTEGER;
    DECLARE :i INTEGER;
    DECLARE c1Bulk CURSOR FOR SELECT * FROM t1 WHERE c1 = :a FOR UPDATE;
    START TRANSACTION;
    OPEN c1Bulk;
    BulkLinesLoop:
    LOOP
        FETCH NEXT FROM c1Bulk INTO :i;
        IF SQLSTATE = '02000' THEN
            LEAVE BulkLinesLoop;
        END IF;
        UPDATE SET c1 = 4 WHERE CURRENT OF c1Bulk;
    END LOOP;
    CLOSE c1Bulk;
    SET :b = 0;
    WHILE (:b < 100000) DO
    BEGIN
        SET :b = :b + 1;
    END;
    END WHILE;
    COMMIT WORK;
END;
CALL p1(2)

```

WHILE ループがトランザクションの COMMIT を遅延させることに注意してください。この遅延の間、ユーザー B は t1 の更新を試みるとします。これらの行はユーザー A が SELECT FOR UPDATE ステートメントでロックしているため、ユーザー B にはステータスコード 84 が返されます。

次の例ではテーブル t1 を使用して、SELECT FOR UPDATE がストアードプロシージャの外部で使用された場合のレコードのロック方法を示します。t1 は Demodata サンプルデータベースの一部であるとしています。

この例は、2 人のユーザー A と B は Demodata にログインしているものとします。ユーザー A が以下の処理を行います。

```

DROP TABLE t1
CREATE TABLE t1 (c1 INTEGER, c2 INTEGER)
INSERT INTO t1 VALUES (1,1)
INSERT INTO t1 VALUES (2,1)
INSERT INTO t1 VALUES (1,1)
INSERT INTO t1 VALUES (2,1)

```

(AUTOCOMMIT をオフにする)

(実行して取り出す) : "SELECT * FROM t1 WHERE c1 = 2 FOR UPDATE"

c1 = 2 である 2 つのレコードは、ユーザー A が COMMIT WORK または ROLLBACK WORK ステートメントを発行するまでロックされます。

(ユーザー B が t1 の更新を試みる) : "UPDATE t1 SET c1=3 WHERE c1=2" これらの行はユーザー A が SELECT FOR UPDATE ステートメントでロックしているため、ユーザー B にはステータス コード 84 が返されます。

(ここで、ユーザー A がトランザクションをコミットしたとします。) c1 = 2 である 2 つのレコードのロックは解除されます。

これで、ユーザー B は "UPDATE t1 SET c1=3 WHERE c1=2" を実行して、c1 の値を変更できます。

概算数値リテラル

```
SELECT * FROM results WHERE quotient =-4.5E-2
INSERT INTO results (quotient) VALUES (+5E7)
```

BETWEEN 述語

式 1 BETWEEN 式 2 and 式 3 という構文では、式 1 >= 式 2 かつ、式 1 <= 式 3 の場合に True が返されます。式 1 >= 式 3 または式 1 <= 式 2 の場合は、False が返されます。

式 2 と式 3 は動的パラメーター (たとえば、SELECT * FROM emp WHERE emp_id BETWEEN ? AND ?) にすることができます。

次の例では、ID が 10000 と 20000 の間にある名が Person テーブルから取り出されます。

```
SELECT First_name FROM Person WHERE ID BETWEEN 10000 AND 20000
```

相関名

テーブルと列の相関名はどちらもサポートされています。

次の例では、person テーブルと faculty テーブルからデータが選択されます。2 つのテーブルを区別するためにエイリアスの T1 と T2 を使用しています。

```
SELECT * FROM Person t1, Faculty t2 WHERE t1.id = t2.id
```

また、テーブルの相関名は、次の例のように FROM 句を使って指定することもできます。

```
SELECT a.Name, b.Capacity FROM Class a, Room b
WHERE a.Room_Number = b.Number
```

正確な数値リテラル

```
SELECT car_num, price FROM cars WHERE car_num = 49042 AND price = 49999.99
```

IN 述語

これにより、名が Bill と Roosevelt のレコードがテーブル Person から選択されます。

```
SELECT * FROM Person WHERE First_name IN ('Roosevelt', 'Bill')
```

セット関数

Zen ではセット関数 AVG、COUNT、COUNT_BIG、LAG、MAX、MIN、STDEV、STDEVP、SUM、VAR、および VARP がサポートされています。それぞれ以下の例で説明します。

LAG は、ウィンドウ関数としてのみ有効です。この詳細と例については、[LAG](#) キーワードを参照してください。

AVG、MAX、MIN、および SUM

集計関数の AVG、MAX、MIN、および SUM は、一般に予想されるとおりに動作します。次の例は、Faculty サンプル テーブルの Salary フィールドを用いてこれらの関数の使い方を示しています。

```
SELECT AVG(Salary) FROM Faculty
SELECT MAX(Salary) FROM Faculty
SELECT MIN(Salary) FROM Faculty
SELECT SUM(Salary) FROM Faculty
```

ほとんどの場合、これらの関数は GROUP BY と一緒に使用され、次の例で示すように、共通の列を持つ一連の行に適用されます。

```
SELECT AVG(Salary) FROM Faculty GROUP BY Dept_Name
```

次の例では、amount_paid の総額が 100 以上の場合に、student_id と amount_paid の総額が billing テーブルから取り出されます。次に、student_id 別にレコードが分類されます。

```
SELECT Student_ID, SUM(Amount_Paid)
FROM Billing
GROUP BY Student_ID
HAVING SUM(Amount_Paid) >=100.00
```

式が正の整数リテラルの場合、そのリテラルが結果セット内の列の番号として解釈され、その列について順序付けが行われます。セット関数またはセット関数を含む式については、順序付けは許可されません。

COUNT および COUNT_BIG

COUNT (式) および COUNT_BIG (式) によって、述部にある式の非ヌル値すべてがカウントされます。COUNT(*) および COUNT_BIG(*) では、ヌル値を含むすべての値がカウントされます。COUNT() は、最大値が 2,147,483,647 である INTEGER データ型を返します。COUNT_BIG() は、最大値が 9,223,372,036,854,775,807 である BIGINT データ型を返します。

次の例は、成績評価点平均が 3.5 以上ある（および、結果がヌルでない）化学専攻の学生の数を返します。

```
SELECT COUNT(*) FROM student WHERE (CUMULATIVE_GPA > 3.4 and MAJOR='Chemistry')
```

STDEV および STDEVP

STDEV 関数は、データのサンプルに基づいて、すべてのデータの標準偏差を返します。STDEVP 関数は、指定された式のすべての値の母集団に対する標準偏差を返します。各関数の方程式は以下のとおりです。

$$STDEV(x_n) = \sqrt{\frac{n\sum x^2 - (\sum x)^2}{n(n-1)}} \quad STDEVP(x_n) = \sqrt{\frac{n\sum x^2 - (\sum x)^2}{n^2}}$$

次の例は、Student サンプル テーブルから、専攻科目別に成績評価点平均の標準偏差を返します。

```
SELECT STDEV(Cumulative_GPA), Major FROM Student GROUP BY Major
```

次の例は、Student サンプル テーブルから、専攻科目別に成績評価点平均の母集団の標準偏差を返します。

```
SELECT STDEVP(Cumulative_GPA), Major FROM Student GROUP BY Major
```

VAR および VARP

VAR 関数は、データのサンプルに基づいて、すべての値の統計的分散を返します。VARP 関数は、指定された式のすべての値の母集団に対する統計的分散を返します。各関数の方程式は以下のとおりです。

$$VAR(x_n) = \frac{n\sum x^2 - (\sum x)^2}{n(n-1)} \quad VARP(x_n) = \frac{n\sum x^2 - (\sum x)^2}{n^2}$$

次の例は、Student サンプル テーブルから、専攻科目別に成績評価点平均の統計的分散を返します。

```
SELECT VAR(Cumulative_GPA), Major FROM Student GROUP BY Major
```

次の例は、Student サンプル テーブルから、専攻科目別に成績評価点平均の母集団の統計的分散を返します。

```
SELECT VARP(Cumulative_GPA), Major FROM Student GROUP BY Major
```

STDEV、STDEVP、VAR、および VARP の場合、式は数値データ型である必要があり、8 バイトの DOUBLE が返されるということに注意してください。式の最小値と最大値の差が範囲外である場合は、浮動小数点のオーバーフロー エラーになります。式に集計関数を含むことはできません。式フィールドに値が入っている行が少なくとも 2 つはある必要があります。そうでないと、結果は計算されず、NULL が返されます。

日付リテラル

日付値を参照してください。

時刻リテラル

時刻値を参照してください。

タイムスタンプ リテラル

タイムスタンプ値を参照してください。

文字列リテラル

文字列値を参照してください。

日付演算

日付演算を参照してください。

IF

IF システム スカラー関数によって、条件の真の値に基づく条件付き実行が提供されます。

次の式は、列ヘッダー Prime1 を出力し、amount_owed 列の値が 2000 の場合は 2000 を、2000 でない場合は 0 を出力します。

```
SELECT Student_ID, Amount_Owed,  
IF (Amount_Owed = 2000, Amount_Owed, Convert(0, SQL_DECIMAL)) "Prime1"  
FROM Billing
```

次の例では、Class テーブルから、Section 列が 001 の場合はその値を、001 でない場合は "xxx" を、列ヘッダー Prime1 の下に出力します。

さらに、列ヘッダー **Prime2** の下に、**Section** 列の値が **002** の場合はその値を、**002** でない場合は **"yyy"** を出力します。

```
SELECT ID, Name, Section,
IF (Section = '001', Section, 'xxx') "Prime1",
IF (Section = '002', Section, 'yyy') "Prime2"
FROM Class
```

ネストした IF 関数を使用して、ヘッダー **Prime1** とヘッダー **Prime2** を合体させることができます。次のクエリは、列ヘッダー **Prime** の下に、**Section** 列の値が **001** か **002** の場合は **Section** 列の値を出力し、それ以外の場合は **"xxx"** を出力します。

```
SELECT ID, Name, Section,
IF (Section = '001', Section, IF(Section = '002', Section, 'xxx')) Prime
FROM Class
```

複数データベースの結合

必要であれば、データベース名を **FROM** 句内のエイリアス付きのテーブル名の前に付加して、結合で使用する 2 つ以上の異なるデータベースにあるテーブルを区別することができます。

指定したデータベースはすべて同じデータベース エンジンで処理し、同じデータベース コード ページ設定である必要があります。これらのデータベースは同じ物理ボリューム上になくてもかまいません。現行データベースはセキュリティで保護されていてもいなくてもかまいませんが、結合内のそれ以外のデータベースはセキュリティで保護されていない必要があります。参照整合性に関しては、すべての **RI** キーが同一のデータベース内に存在している必要があります。(エンコードも参照してください。)

リテラル データベース名は選択リストまたは **WHERE** 句内で使用することはできません。選択リストまたは **WHERE** 句内で特定の列を参照する場合は、指定する各テーブルのエイリアスを使用する必要があります。例を参照してください。

2 つの異なるデータベース **accounting** と **customers** が同じサーバー上に存在していると仮定します。次の例のようなテーブル エイリアスと SQL 構文を使って 2 つのデータベースにあるテーブルを結合することができます。

```
SELECT ord.account, inf.account, ord.balance, inf.address
FROM accounting.orders ord, customers.info inf
WHERE ord.account = inf.account
```

次の例では、2 つのデータベースは **acctdb** と **loandb** です。テーブル エイリアスはそれぞれ **a** と **b** です。

```
SELECT a.loan_number_a, b.account_no, a.current_bal, b.balance
FROM acctdb.ds500_acct_master b LEFT OUTER JOIN loandb.m1502_loan_master a ON (a.loan_number_a =
```

```
b.loan_number)
WHERE a.current_bal <> (b.balance * -1)
ORDER BY a.loan_number_a
```

左外部結合

次の例は、Demodata データベースの **Person** テーブルと **Student** テーブルにアクセスして、学生の姓と、名の頭文字、および GPA を取得する方法を示します。LEFT OUTER JOIN を使用すると、**Person** テーブル内のすべての行がフェッチされます（LEFT OUTER JOIN の左側にあるテーブル）。すべての人が GPA を持っているわけではないため、列によっては結果にヌル値を持ちます。以下は、外部結合がどのように働き、両方のテーブルから一致しない行を返すかを示します。

```
SELECT Last_Name, Left(First_Name,1) AS First_Initial, Cumulative_GPA AS GPA FROM "Person"
LEFT OUTER JOIN "Student" ON Person.ID=Student.ID
ORDER BY Cumulative_GPA DESC, Last_Name
```

完全に整数の GPA を持つ人全員がわかり、彼らをその姓の長さに従って並べたいとします。MOD ステートメントと LENGTH スカラー関数を使用して、クエリに次の文を追加することにより、これを実現できます。

```
WHERE MOD(Cumulative_GPA,1)=0 ORDER BY LENGTH(Last_Name)
```

右外部結合

LEFT OUTER JOIN と RIGHT OUTER JOIN との違いは、一致しないすべての行を表示する対象が RIGHT OUTER JOIN の右側に定義されたテーブルである点です。クエリの LEFT OUTER JOIN の部分を RIGHT OUTER JOIN を使用するように変更します。GPA がない場合でも、右テーブル（この場合は **Student**）から不一致行がすべて表示される点が異なります。ただし、**Student** テーブル内の行はすべて GPA を持つため、すべての行がフェッチされます。

```
SELECT Last_Name, Left(First_Name,1) AS First_Initial, Cumulative_GPA AS GPA FROM "Person"
RIGHT OUTER JOIN "Student" ON Person.ID=Student.ID
ORDER BY Cumulative_GPA DESC, Last_Name
```

カルテシアン結合

カルテシアン結合は、両方のテーブルの行を可能な限りすべて組み合わせた行列です。カルテシアン内の行数は、1 番目のテーブル内の行数に 2 番目のテーブル内の行数を掛けた値に等しくなります。

データベース内に次のようなテーブルがあるとします。

Addr テーブル

EmpID	Street
E1	101 Mem Lane
E2	14 Young St.

Loc テーブル

LocID	Name
L1	PlanetX
L2	PlanetY

次のステートメントにより、上記のテーブルのカルテシアン結合を実行します。

```
SELECT * FROM Addr,Loc
```

これにより、次のデータ セットが生成されます。

EmpID	Street	LocID	Name
E1	101 Mem Lane	L1	PlanetX
E1	101 Mem Lane	L2	PlanetY
E2	14 Young St	L1	PlanetX
E2	14 Young St	L2	PlanetY

Sys\$create および Sys\$update を使用したクエリ

次の例は、作成時以降に更新されたレコードを検索する簡単な例を示します。

```
create table sensorData SYSDATA_KEY_2 (location varchar(20), temp real);

insert into sensorData values('Machine1', 77.3);
insert into sensorData values('Machine2', 79.8);
insert into sensorData values('Machine3', 65.4);
insert into sensorData values('Machine4', 90.0);

select "sys$create", "sys$update", sensorData.* from sensorData;

-- 行を更新します
update sensorData set temp = 90.1 where location = 'Machine1';

-- 更新された行を検索します
select "sys$create", "sys$update", sensorData.* from sensorData where sys$update > sys$create;
```


集計関数の DISTINCT

DISTINCT は集計関数で有用です。SUM、AVG、および COUNT と一緒に使用した場合は、重複する値を取り除いてから、総計、平均、件数を求めます。MIN および MAX と一緒に使用することはできますが、返される最大と最小の結果は変わりません。

たとえば、学部ごとの給与の最低額、最高額、平均額を知りたいと仮定します。このとき、同額の給与は除くことにします。次のステートメントは、**computer science** 学部以外の学部に対してこれを実行します。

```
SELECT dept_name, MIN(salary), MAX(salary), AVG(DISTINCT salary) FROM faculty WHERE dept_name <> 'computer science' GROUP BY dept_name
```

反対に、同額の給与も含む場合は DISTINCT を省きます。

```
SELECT dept_name, MIN(salary), MAX(salary), AVG(salary) FROM faculty WHERE dept_name <> 'computer science' GROUP BY dept_name
```

SELECT ステートメントでの DISTINCT の使用方法については、[DISTINCT](#) を参照してください。

TOP または LIMIT

TOP または LIMIT キーワードを使用することにより、SELECT ステートメントで返される行数を制限することができます。数値はリテラルの正の値でなければなりません。32 ビットの符号なし整数として定義します。たとえば、次のように指定します。

```
SELECT TOP 10 * FROM Person
```

Demodata の Person テーブルから最初の 10 行が返されます。

LIMIT は、OFFSET キーワードを提供する点を除けば、TOP とまったく同じです。OFFSET により、返されたレコード内で最初の行を選択することで、結果セット内を「スクロール」できるようになります。たとえば、オフセットを 5 にした場合、返される最初の行は行 6 になります。LIMIT には、オフセットを指定する方法が 2 つあります。次の例に示すように、OFFSET キーワードを使用する方法と使用しない方法で、どちらも同じ結果が返されます。

```
SELECT * FROM Person LIMIT 10 OFFSET 5
SELECT * FROM Person LIMIT 5,10
```

OFFSET キーワードを使用しない場合は、オフセット値を行数の前に、カンマで区切って置く必要があることに注意してください。

TOP または LIMIT を ORDER BY と一緒に使用することができます。その場合、データベース エンジン はテンポラリ テーブルを生成し、ORDER BY で使用できるインデック

スがない場合は、そこにクエリの結果セット全体を置きます。テンポラリ テーブル内の行は、結果セットにおいて、**ORDER BY** で指定した順序で並べられますが、**TOP** または **LIMIT** によって決められた行数のみがクエリで返されます。

TOP または **LIMIT** を使用するビューは、ほかのテーブルやビューと結合できます。

TOP や **LIMIT** と **SET ROWCOUNT** の主な違いは、**SET ROWCOUNT** が現在のデータベース セッション中に発行される後続のステートメントすべてに作用するのに対し、**TOP** や **LIMIT** は現在のステートメントにのみ作用する点です。

クエリで **SET ROWCOUNT** と、**TOP** または **LIMIT** を使用した場合は、2 つの値のうち小さい方の値に等しい行数を返します。

TOP と **LIMIT** は、1 つのクエリまたはサブクエリ内でいずれか一方を使用できますが、両方は使用できません。

カーソルのタイプと **TOP** または **LIMIT**

TOP 句または **LIMIT** 句を持ち、動的カーソルを使用する **SELECT** クエリは、カーソルのタイプを静的カーソルに変換します。前方のみのカーソルと静的カーソルは影響を受けません。

TOP または **LIMIT** の例

次の例では、**TOP** と **LIMIT** の両方の句を使用しています。これらは、キーワードとして互いに代替でき、同じ結果を得られますが、**LIMIT** の方が返される行に関してより多くの制御を提供します。

```
SELECT TOP 10 * FROM person; -- 10 行を返します
SELECT * FROM person LIMIT 10; -- 10 行を返します
SELECT * FROM person LIMIT 10 OFFSET 5; -- 行 6 から 10 行を返します
SELECT * FROM person LIMIT 5,10; -- 行 6 から 10 行を返します
SET ROWCOUNT = 5;
SELECT TOP 10 * FROM person; -- 5 行を返します
SELECT * FROM person LIMIT 10; -- 5 行を返します
SET ROWCOUNT = 12;
SELECT TOP 10 * FROM person ORDER BY id; -- id 列で順序付けされた全リストから最初の 10 行を返します
SELECT * FROM person LIMIT 20 ORDER BY id; -- id 列で順序付けされた全リストから最初の 12 行を返します
```

次の例では、**TOP** または **LIMIT** が **VIEW**、**UNION**、またはサブクエリで使用された場合のさまざまな動作を示します。

```
CREATE VIEW v1 (c1) AS SELECT TOP 10 id FROM person;
CREATE VIEW v2 (d1) AS SELECT TOP 5 c1 FROM v1;
SELECT * FROM v2 -- 5 行を返します
```

```

SELECT TOP 10 * FROM v2 -- 5 行を返します
SELECT TOP 2 * FROM v2 -- 2 行を返します
SELECT * FROM v2 LIMIT 10 -- 5 行を返します
SELECT * FROM v2 LIMIT 10 OFFSET 3 -- 行 4 から 2 行を返します
SELECT * FROM v2 LIMIT 3,10 -- 行 4 から 2 行を返します

SELECT TOP 10 id FROM person UNION SELECT TOP 13 faculty_id FROM class -- 17 行を返します
SELECT TOP 10 id FROM person UNION ALL SELECT TOP 13 faculty_id FROM class -- 23 行を返します
SELECT id FROM person WHERE id IN (SELECT TOP 10 faculty_id from class) -- 5 行を返します
SELECT id FROM person WHERE id >= any (SELECT TOP 10 faculty_id from class) -- 1040 行を返します

```

次の例では、ID が特定の番号より上の学生について、その姓と支払う義務のある金額を返します。

```

SELECT p_last_name, b_owed FROM
  (SELECT TOP 10 id, last_name FROM person ORDER BY id DESC) p (p_id, p_last_name),
  (SELECT TOP 10 student_id, SUM (amount_owed) FROM billing GROUP BY student_id ORDER BY student_id
  DESC) b (b_id, b_owed)
WHERE p.p_id = b.b_id AND p.p_id > 714662900
ORDER BY p_last_name ASC

```

テーブル ヒントの例

このトピックでは、テーブル ヒントの作業例を提供します。Zen で SQL ステートメントを使用して、それらを作成します。

```

DROP TABLE t1
CREATE TABLE t1 (c1 INTEGER, c2 INTEGER)
INSERT INTO t1 VALUES (1,10)
INSERT INTO t1 VALUES (1,10)
INSERT INTO t1 VALUES (2,20)
INSERT INTO t1 VALUES (2,20)
INSERT INTO t1 VALUES (3,30)
INSERT INTO t1 VALUES (3,30)
CREATE INDEX it1c1 ON t1 (c1)
CREATE INDEX it1c1c2 ON t1 (c1, c2)
CREATE INDEX it1c2 ON t1 (c2)
CREATE INDEX it1c2c1 ON t1 (c2, c1)
DROP TABLE t2
CREATE TABLE t2 (c1 INTEGER, c2 INTEGER)
INSERT INTO t2 VALUES (1,10)
INSERT INTO t2 VALUES (1,10)
INSERT INTO t2 VALUES (2,20)
INSERT INTO t2 VALUES (2,20)
INSERT INTO t2 VALUES (3,30)
INSERT INTO t2 VALUES (3,30)

```

テーブル ヒントの使用には、一定の制約が適用されます。[テーブル ヒントの制限事項](#)の例を参照してください。

次の例は、インデックス it1c1c2 で最適化します。

```

SELECT * FROM t1 WITH (INDEX(it1c1c2)) WHERE c1 = 1

```

これを次の例と対比してみます。制約は `c1 = 1` のみから成るため、次の例ではインデックス `it1c2` ではなく `it1c1` で最適化しています。クエリの最適化に使用できないインデックスがクエリに指定されている場合、ヒントは無視されます。

```
SELECT * FROM t1 WITH (INDEX(it1c2)) WHERE c1 = 1
```

次の例はテーブル `t1` のテーブル スキャンを実行します。

```
SELECT * FROM t1 WITH (INDEX(0)) WHERE c1 = 1
```

次の例は、インデックス `it1c1c2` および `it1c2c1` で最適化します。

```
SELECT * FROM t1 WITH (INDEX(it1c1c2, it1c2c1)) WHERE c1 = 1 OR c2 = 10
```

次の例は、ビューの作成でテーブル ヒントを使用します。すべてのレコードがビューから選択されると、`SELECT` ステートメントはインデックス `it1c1c2` で最適化します。

```
DROP VIEW v2  
CREATE VIEW v2 as SELECT * FROM t1 WITH (INDEX(it1c1c2)) WHERE c1 = 1  
SELECT * FROM v2
```

次の例はサブクエリでテーブル ヒントを使用し、インデックス `it1c1c2` で最適化します。

```
SELECT * FROM (SELECT c1, c2 FROM t1 WITH (INDEX(it1c1c2)) WHERE c1 = 1) AS a WHERE a.c2 = 10
```

次の例はサブクエリでテーブル ヒントを使用し、エイリアス名 `"a"` を使用します。このエイリアス名は必須です。

```
SELECT * FROM (SELECT Last_Name FROM Person AS P with (Index(Names)) ) a
```

次の例は、`c1 = 1` 制約に基づいてクエリを最適化し、インデックス `it1c1c2` に基づいて `GROUP BY` 句を最適化します。

```
SELECT c1, c2, count(*) FROM t1 WHERE c1 = 1 GROUP BY c1, c2
```

次の例はインデックス `it1c1` で最適化し、前の例とは異なり、制約のみで最適化して `GROUP BY` 句では最適化しません。

```
SELECT c1, c2, count(*) FROM t1 WITH (INDEX(it1c1)) WHERE c1 = 1 GROUP BY c1, c2
```

GROUP BY 句は指定したインデックス `it1c1` を使用して最適化できないため、データベース エンジン は GROUP BY を処理するのにテンポラリ テーブルを使用します。

次の例は JOIN 句でテーブル ヒントを使用し、インデックス `it1c1c2` で最適化します。

```
SELECT * FROM t2 INNER JOIN t1 WITH (INDEX(it1c1c2)) ON t1.c1 = t2.c1
```

これを、テーブル ヒントを使用せずインデックス `it1c1` で最適化する次のステートメントと対比してみます。

```
SELECT * FROM t2 INNER JOIN t1 ON t1.c1 = t2.c1
```

次の例は JOIN 句でテーブル ヒントを使用し、テーブル `t1` でテーブル スキャンを実行します。

```
SELECT * FROM t2 INNER JOIN t1 WITH (INDEX(0)) ON t1.c1 = t2.c1
```

これを、同じくテーブル `t1` のテーブル スキャンを実行する次の例と対比してみます。ただし、JOIN 句が使用されていないため、このステートメントはテンポラリ テーブル結合を使用します。

```
SELECT * FROM t2, t1 WITH (INDEX(0)) WHERE t1.c1 = t2.c1
```

関連項目

[グローバル変数](#)

SELECT (INTO 付き)

SELECT (INTO 付き) ステートメントを使用すると、指定したテーブルから列の値を選択して、変数に挿入したり、テーブルにデータを入れることができます。

構文

```
SELECT [ ALL | DISTINCT ] [ top 句 ] 選択リスト INTO 変数 | テーブル名 | テンポラリ テーブル名 [ , 変数 ]...
FROM テーブル参照 [ , テーブル参照 ]... [ WHERE 検索条件 ]
[ GROUP BY 式 [ , 式 ]... [ HAVING 検索条件 ] ] [ UNION [ ALL ] クエリ スペック ] [ ORDER BY
order-by 式 [ , order-by 式 ]... ]
```

```
クエリ スペック ::= ( クエリ スペック )
| SELECT [ ALL | DISTINCT ] [ top 句 ] 選択リスト
  FROM テーブル参照 [ , テーブル参照 ]...
  [ WHERE 検索条件 ]
  [ GROUP BY 式 [ , 式 ]...
  [ HAVING 検索条件 ] ]
```

```
変数 ::= ユーザー定義名
```

```
テーブル名 ::= テーブルのユーザー定義名
```

```
テンポラリ テーブル名 ::= テンポラリ テーブルのユーザー定義名
```

残りの構文定義については、[SELECT](#) を参照してください。

備考

変数は、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内で生じる必要があります。

SELECT INTO を使用してテーブルにデータを入れられるのは、SELECT INTO ステートメントがユーザー定義関数またはトリガーの外部で使用される場合のみです。ユーザー定義関数またはトリガーの内部で、SELECT INTO を使用してテーブルを作成したりデータを入れることは許可されていません。

SELECT INTO をストアド プロシージャ内で使用することは許可されています。

SELECT INTO ステートメントによって作成し、データを入れることができるテーブルは 1 つだけです。単独の SELECT INTO ステートメントで複数のテーブルの作成とデータ投入を行うことはできません。

SELECT INTO で作成された新規テーブルは、ソース テーブルからの CASE および NOT NULL 制約によってのみ保守することができます。DEFAULT および COLLATE などの

これ以外の制約では保守できません。さらに、新しいテーブルにはインデックスは作成されません。

例

SELECT INTO を使用してテンポラリ テーブルにデータを入れる方法については、[CREATE \(テンポラリ\) TABLE](#) の例を参照してください。

次の例では、Person テーブルの、名前が Bill であるデータの first_name と last_name の値を変数 :x、:y に割り当てます。

```
SELECT first_name, last_name INTO :x, :y FROM person WHERE first_name = 'Bill'
```

関連項目

[CREATE FUNCTION](#)

[CREATE PROCEDURE](#)

[CREATE \(テンポラリ\) TABLE](#)

[CREATE TABLE](#)

SET

SET ステートメントは、宣言された変数に値を割り当てます。

構文

```
SET 変数名 = プロシージャ式
```

備考

変数を設定するには、事前に変数が宣言されていなければなりません。SET は、ストアド プロシージャとトリガーでのみ使用できます。

例

次の例では、var1 に値 10 を割り当てます。

```
SET:var1 = 10;
```

関連項目

[CREATE PROCEDURE](#)

[DECLARE](#)

SET ANSI_PADDING

SET ANSI_PADDING ステートメントにより、NULL（バイナリ ゼロ）で埋められた CHAR データ型をリレーショナル エンジンで処理できるようになります。CHAR は固定長の文字データ型として定義されます。

Zen は 2 つのインターフェイス、トランザクショナルとリレーショナルをサポートしています。MicroKernel エンジンでは、CHAR を NULL で埋めることができます。一方、リレーショナル エンジンは ANSI 標準の埋め込み方法に従っているため、CHAR は空白で埋められます。たとえば、デフォルトでは、CREATE TABLE ステートメントで作成した CHAR 列は常に空白で埋められます。

両方のインターフェイスを使用しているアプリケーションの場合、NULL で埋められた文字列の処理が必要になる可能性があります。

構文

```
SET ANSI_PADDING = < ON | OFF >
```

備考

デフォルト値は ON です。これは、空白で埋められた文字列が CHAR に挿入されるということです。論理式の比較で、先頭の空白は**意味のないもの**と見なされます。先頭の NULL は**意味のあるもの**と見なされます。

このステートメントを OFF に設定すると、NULL で埋められた文字列が CHAR に挿入されるようになります。先頭の NULL と空白はどちらも論理式の比較で**意味のないもの**と判断されます。

Windows では、ANSI 埋め込みは、レジストリ設定によって DSN に対してオンまたはオフに設定できます。Actian Web サイトの Zen Knowledge Base で "ansipadding" を検索してください。

以下の文字列関数は NULL の埋め込みをサポートしています。

CHAR_LENGTH	CONCAT	LCASE または LOWER
LEFT	LENGTH	LOCATE
LTRIM	POSITION	REPLACE
REPLICATE	RIGHT	RTRIM

STUFF SUBSTRING UCASE または
UPPER

ANSI_PADDING が各関数にどのような影響を与えるかについては、対応するスカラー関数の記述を参照してください。

制限

SET ANSI_PADDING には次の制限が適用されます。

- このステートメントは固定長の文字データ型 **CHAR** にのみ適用され、NCHAR、VARCHAR、NVARCHAR、LONGVARCHAR、および NLONGVARCHAR には適用されません。
- ステートメントはセッションレベルに適用されます。

例

次の例は、SET ANSI_PADDING を ON にした状態と OFF にした状態で、INSERT ステートメントによる文字列の埋め込みがどのような結果になるかを示しています。

```
DROP TABLE t1
CREATE TABLE t1 (c1 CHAR(4))
SET ANSI_PADDING = ON
INSERT INTO t1 VALUES ('a') -- 文字列 a = a¥0x20¥0x20¥0x20
INSERT INTO t1 VALUES ('a' + CHAR(0) + CHAR(0) + CHAR(0)) -- 文字列 a = a¥0x00¥0x00¥0x00
DROP TABLE t1
CREATE TABLE t1 (c1 CHAR(4))
SET ANSI_PADDING = OFF
INSERT INTO t1 VALUES ('a') -- 文字列 a = a¥0x00¥0x00¥0x00
INSERT INTO t1 VALUES ('a' + CHAR(32) + CHAR(32) + CHAR(32)) -- 文字列 a = a¥0x20¥0x20¥0x20
```

次の例は、SET ANSI_PADDING を ON にした状態と OFF にした状態で、UPDATE ステートメントによる文字列の埋め込みがどのような結果になるかを示しています。

```
DROP TABLE t1
CREATE TABLE t1 (c1 CHAR(4))
SET ANSI_PADDING = ON
UPDATE t1 SET c1 = 'a' -- すべての行の c1 = a¥0x20¥0x20¥0x20
UPDATE t1 SET c1 = 'a' + CHAR(0) + CHAR(0) + CHAR(0) -- すべての行の c1 = a¥0x00¥0x00¥0x00
DROP TABLE t1
CREATE TABLE t1 (c1 CHAR(4))
SET ANSI_PADDING = OFF
UPDATE t1 SET c1 = 'a' -- すべての行の c1 = a¥0x00¥0x00¥0x00
UPDATE t1 SET c1 = 'a' + CHAR(32) + CHAR(32) + CHAR(32) -- すべての行の c1 = a¥0x20¥0x20¥0x20
```

次の例は、文字型の列 `c1` の内容をバイナリ形式で表示できるよう、`c1` を `BINARY` データ型にキャストする方法を示します。テーブル `t1` には次の 6 行のデータがあるとします。

```
a¥x00¥x00¥x00¥x00
a¥x00¥x00¥x00¥x00
a¥x00¥x20¥x00¥x00
a¥x00¥x20¥x00¥x00
a¥x20¥x20¥x20¥x20
a¥x20¥x20¥x20¥x20
```

次のステートメントにより、`c1` が `BINARY` データ型にキャストされます。

```
SELECT CAST(c1 AS BINARY(4)) FROM t1
```

`SELECT` ステートメントは以下の結果を返します。

```
0x61000000
0x61000000
0x61002000
0x61002000
0x61202020
0x61202020
```

関連項目

[INSERT](#)

[UPDATE](#)

[文字列関数](#)

[変換関数](#)

SET CACHED_PROCEURES

SET CACHED_PROCEURES ステートメントは、データベース エンジンが SQL セッションのメモリにキャッシュするストアド プロシージャの数を指定します。

構文

```
SET CACHED_PROCEURES = プロシージャ数
```

備考

プロシージャ数の値は、ゼロからおよそ 20 億までの整数です。データベース エンジン は、デフォルトを自動的に 50 とします。各セッションは、SET ステートメントを発行することにより、そのセッションでキャッシュするプロシージャ数を変更できます。

SET CACHED_PROCEURES の対となるステートメントは **SET PROCEDURES_CACHE** です。

- 両方の SET ステートメントをゼロに設定すると、データベース エンジン はストアド プロシージャをキャッシュ**しません**。さらに、エンジンはストアド プロシージャに使用される既存のキャッシュを削除します。つまり、エンジンは、両方のステートメントをゼロに設定する前に**キャッシュされた**すべてのストアド プロシージャをキャッシュから消去します。
- 一方**のステートメントだけをゼロまたはゼロ以外の値に設定した場合、もう一方のステートメントは暗黙的にゼロに設定されます。暗黙的にゼロに設定されたステートメントは無視されます。たとえば、70 プロシージャをキャッシュすることのみに関心があり、メモリの量に関心がなければ、CACHED_PROCEURES を 70 に設定します。データベース エンジン は暗黙的に PROCEDURES_CACHE をゼロに設定します。つまり、設定を無視します。

CACHED_PROCEURES をゼロ以外の値に設定すると、以下の条件が適用されます。プロシージャの実行によって、キャッシュされたプロシージャ数が CACHED_PROCEURES 値を超える場合、データベース エンジン は、最近最も使用頻度の低いプロシージャをキャッシュから削除します。

メモリ キャッシュを使用している場合、プロシージャの実行後もコンパイルしたストアド プロシージャを保持しています。一般的に、キャッシングによって、キャッシュされたプロシージャの次回からの呼び出しのパフォーマンスが向上します。キャッシュの設定やアプリケーションが実行する SQL によっては、過度のメモリ スワッピングや

スラッシングが発生するので注意してください。スラッシングはパフォーマンスの低下を招きます。

レジストリ設定

キャッシュするプロシージャ数は、SET ステートメントのほかにレジストリ設定でも指定できます。レジストリ設定はすべてのセッションに適用され、初期値を設定する便利な方法です。各セッションは SET ステートメントを使用して、そのセッション固有の値を指定できます。この値はレジストリ設定より優先されます。

レジストリ設定は、Zen Enterprise Server または Cloud Server がサポートされるすべてのサーバー プラットフォームに適用されます。レジストリ設定は手作業で変更する必要があります。Windows では、オペレーティング システムで提供されているレジストリ エディターを使用します。Linux では、**psregedit** ユーティリティを使用することができます。

レジストリ設定が指定されていない場合、データベース エンジンではデフォルトを自動的に 50 とします。

キャッシュされるプロシージャ数のレジストリ設定を Windows で指定するには

1. 以下のキーを見つけます。

HKEY_LOCAL_MACHINE¥SOFTWARE¥Actian¥Zen¥SQL Relational Engine

メモ：ほとんどの Windows システムで、このキーの場所は

HKEY_LOCAL_MACHINE¥SOFTWARE¥Actian¥Zen です。ただし、

HKEY_LOCAL_MACHINE¥SOFTWARE の下位以降の場所はオペレーティング システムによって異なる可能性があります。

2. このキーに、**CachedProcedures** という新しい文字列値を作成します。
3. **CachedProcedures** をキャッシュしたいプロシージャ数に設定します。

Linux の Zen レジストリでキャッシュ プロシージャ レジストリ キーを設定するには

1. 以下のキーを見つけます。

PS_HKEY_CONFIG¥SOFTWARE¥Actian¥Zen¥SQL Relational Engine

2. このキーに、**CachedProcedures** という新しい文字列値を作成します。

3. **CachedProcedures** をキャッシュしたいプロシージャ数に設定します。

キャッシングの除外

ストアド プロシージャは以下の条件に当てはまる場合、キャッシュ設定にかかわらず、**キャッシュされません**。

- ストアド プロシージャが、ローカルまたはグローバル テンポラリ テーブルを参照する。ローカル テンポラリ テーブルはポンド記号 (#) で始まる名前を持ちます。グローバル テンポラリ テーブルは 2 つのポンド記号 (##) で始まる名前を持ちます。[CREATE \(テンポラリ\) TABLE](#) を参照してください。
- ストアド プロシージャにデータ定義言語 (DDL) ステートメントが含まれている。[データ定義ステートメント](#) を参照してください。
- ストアド プロシージャに、文字列または文字列を返す式を実行するための EXEC[UTE] ステートメントが含まれている。たとえば、次のような例です。EXEC ('SELECT Student_ID FROM ' + :myinputvar)

例

次の例は、最大 20 のストアド プロシージャを格納する 2 MB のキャッシュ メモリを設定します。

```
SET CACHED_PROCEDURES = 20
SET PROCEDURES_CACHE = 2
```

次の例は、最大 500 のストアド プロシージャを格納する 1,000 MB のキャッシュ メモリを設定します。

```
SET CACHED_PROCEDURES = 500
SET PROCEDURES_CACHE = 1000
```

次の例は、ストアド プロシージャを**キャッシュせず**、既にキャッシュされているプロシージャを削除する指定をします。

```
SET CACHED_PROCEDURES = 0
SET PROCEDURES_CACHE = 0
```

次の例は、120 個のストアド プロシージャをキャッシュし、キャッシュで使用するメモリ量は設定しない指定をします。

```
SET CACHED_PROCEDURES = 120
```

データベース エンジン は暗黙的に `PROCEDURES_CACHE` をゼロに設定します。

関連項目

[CREATE PROCEDURE](#)

[SET PROCEDURES_CACHE](#)

SET DECIMALSEPARATORCOMMA

Zen データベース エンジン は、デフォルトでピリオド (.) を整数部と小数部の間の区切り文字として使い、小数值データを表示します (たとえば、100.95)。SET DECIMALSEPARATORCOMMA ステートメントを使用すれば、整数部と小数部をカンマで区切って結果を表示するよう指定することができます (たとえば、100,95)。

すべての SET ステートメントと同様に、このステートメントの効果は現在のデータベース セッションの間中、または別の SET DECIMALSEPARATORCOMMA ステートメントが発行されるまで適用されます。

構文

```
SET DECIMALSEPARATORCOMMA = < ON | OFF >
```

備考

デフォルト値は OFF です。これは、ピリオドをデフォルトの小数点の記号として使用します。

カンマを小数点の記号として使用する地域では、カンマまたはピリオドを区切り文字として使用して小数值データを入力することができます (カンマを区切り文字として使用したリテラル値は、'123,43' などのように一重引用符で囲む必要があります)。ただし、(SELECT ステートメントの結果のように) データが返される場合、SET DECIMALSEPARATORCOMMA=ON が指定されていない場合は、データは常にピリオドを使って表示されます。

また、小数点の記号にピリオドを使って入力したデータがデータベースに含まれる場合、このステートメントを使用して出力と表示でカンマを区切り文字にするよう指定することができます。

このカンマは出力と表示にのみ影響します。挿入、更新あるいは比較で使用する値にはまったく影響しません。

例

次の例では、ピリオドで区切られたデータの挿入方法と、SET DECIMALSEPARATORCOMMA ステートメントが SELECT の結果にどのように影響するかを示します。


```
CREATE TABLE t1 (c1 real, c2 real)
INSERT INTO t1 VALUES (102.34, 95.234)
SELECT * FROM t1
```

結果：

```
c1      c2
-----
102,34  95.234
```

```
SET DECIMALSEPARATORCOMMA=ON
SELECT * FROM t1
```

結果：

```
c1      c2
-----
102,34  95,234
```

次の例では、カンマで区切られたデータの挿入方法と、SET DECIMALSEPARATORCOMMA ステートメントが SELECT の結果にどのように影響するかを示します。

メモ： カンマを区切り文字として使用できるのは、クライアントとサーバーの両方またはどちらか一方のオペレーティングシステムの地域の設定がカンマを区切り文字として使用するロケールに設定されている場合のみです。たとえば、クライアントとサーバーの両方で U.S. のロケール設定をしていた場合、次の例を実行しようとするするとエラーになります。

```
CREATE TABLE t1 (c1 real, c2 real)
INSERT INTO t1 VALUES ('102,34', '95,234')
SELECT * FROM t1
```

結果：

```
c1      c2
-----
102,34  95.234
```

```
SET DECIMALSEPARATORCOMMA=ON
SELECT * FROM t1
```

結果：

```
c1      c2
-----
102,34  95,234
```

関連項目

[小数点の記号のカンマ](#)

SET DEFAULTCOLLATE

SET DEFAULTCOLLATE ステートメントは、CHAR、VARCHAR、LONGVARCHAR、NCHAR、NVARCHAR、または NLONGVARCHAR データ型のすべての列に使用する照合順序を指定します。ステートメントには次のオプションがあります。

- ヌル値。現在のコード ページの数値順をデフォルトとします
- オルタネート コレーティング シーケンス (ACS) 規則を含んでいるファイルへのパス
- インターナショナル ソート規則 (ISR) テーブル名
- ICU (International Components for Unicode) 照合順序名

構文

```
SET DEFAULTCOLLATE = < NULL | ' ソート順' >
```

ソート順 ::= ACS ファイルのパス名、ISR テーブルの名前、またはサポートされる ICU 照合順序名

備考

SET DEFAULTCOLLATE ステートメントは、便利なグローバル セッション設定を提供します。しかし、個々の列定義で COLLATE キーワードを使用すれば、その列特定の照合順序を設定できます。そうした場合、SET DEFAULTCOLLATE はその列には影響しません。

DEFAULTCOLLATE のデフォルト設定はヌルです。

ACS ファイルの使用

ソート順パラメーターとして ACS ファイルを指定した場合は、次のステートメントが適用されます。

- 呼び出し元アプリケーションではなく、データベース エンジンからアクセス可能なパスを指定する必要があります。
- パスは一重引用符で囲む必要があります。
- パスは、少なくとも 1 文字以上 255 文字以下でなければなりません。
- パスは既に存在しており、ACS ファイルの名前を含んでいる必要があります。ACS ファイルは、MicroKernel エンジンで使用される形式の 265 バイト イメージです。

Zen はデフォルトで、よく使われる ACS ファイル `upper.alt` を `C:\ProgramData\Actian\Zen\samples` にインストールします。カスタム ファイルを使用することもできます。カスタム ファイルの詳細については、『*Zen Programmer's Guide*』の[ユーザー定義 ACS](#) を参照してください。

- DDF ディレクトリに対して相対的である、相対パスを使用できます。相対パスには、1 つのピリオド（現在のディレクトリ）、2 つのピリオド（親ディレクトリ）、円記号、あるいはこれら 3 つのあらゆる組み合わせを含めることができます。相対パスの円記号は、スラッシュ（/）と円記号（¥）のどちらでも使用できます。同じパス内に円記号とスラッシュ文字を混在させることもできます。
- UNC（Universal Naming Convention）のパス名を使用できます。

ISR テーブル名の使用

ソート順パラメーターとして ISR テーブル名を指定した場合は、次のステートメントが適用されます。

- Zen は、本ドキュメントの[国際化](#)に挙げられているテーブル名をサポートしています。
- ISR テーブル名はファイルの名前ではなく、Zen でサポートする ISR の 1 つとして認識される文字列です。
- Zen は、ICU（International Components for Unicode）に基づいて、選択された Unicode 照合順序をサポートしています。単に、ISR テーブル名の代わりに ICU 照合順序名を使用するだけです。使用可能な照合順序は、[ICU Unicode 照合順序を使用した照合順序のサポート](#)に記載されています。

ACS、ISR、および ICU の例

この ACS の例では、Zen で提供される `upper.alt` ファイルを使用して照合順序を設定します。テーブル `t1` は、3 つのテキスト列と 3 つのテキスト以外の列で作成されます。SELECT ステートメントを Zen のシステム テーブルに対して実行し、`t1` 内の列の ID、データ型、サイズ、および属性を返します。結果は、3 つのテキスト列が UPPER 属性を持っていることを示しています。

```
SET DEFAULTCOLLATE = 'C:\ProgramData\Actian\Zen\samples\upper.alt'
DROP TABLE t1
CREATE TABLE t1 (c1 INT, c2 CHAR(10), c3 BINARY(10), c4 VARCHAR(10), c5 LONGVARBINARY, c6
LONGVARCHAR)
SELECT * FROM x$attrib WHERE x$id in (SELECT x$id FROM x$field WHERE x$file = (SELECT x$id FROM
x$file WHERE x$name = 't1'))
```

Xa\$Id	Xa\$Type	Xa\$ASize	Xa\$Attr
327	0	265	UPPER
329	0	265	UPPER
331	0	265	UPPER

3 行が影響を受けました。

次の ACS の例でもテーブル t1 を引き続き使用します。ALTER TABLE ステートメントによって、テキスト列 c2 を CHAR から INTEGER に変更します。SELECT ステートメントの結果は、2 列だけがデフォルトの照合順序の影響を受けていることを示しています。

```
ALTER TABLE t1 ALTER c2 INT
```

```
SELECT * FROM x$attrib WHERE xa$id in (SELECT xe$id FROM x$field WHERE xe$file = (SELECT xf$id FROM x$file WHERE xf$name = 't1'))
```

Xa\$Id	Xa\$Type	Xa\$ASize	Xa\$Attr
329	0	265	UPPER
331	0	265	UPPER

2 行が影響を受けました。

次の ACS の例では、ALTER TABLE ステートメントを使用して、テーブル t1 の列 c1 を INTEGER から CHAR に変更します。SELECT ステートメントの結果は、3 列がデフォルトの照合順序の影響を受けていることを示しています。

```
ALTER TABLE t1 ALTER c1 CHAR(10)
```

```
SELECT * FROM x$attrib WHERE xa$id in (SELECT xe$id FROM x$field WHERE xe$file = (SELECT xf$id FROM x$file WHERE xf$name = 't1'))
```

Xa\$Id	Xa\$Type	Xa\$ASize	Xa\$Attr
326	0	265	UPPER
329	0	265	UPPER
331	0	265	UPPER

3 行が影響を受けました。

次の ISR の例では、VARCHAR 列を持つテーブルを作成し、デフォルトの Windows エンコード CP1252 であるとして、ISR 照合順序 MSFT_ENUS01252_0 を使用します。

```
CREATE TABLE isrttest (ord INT, value VARCHAR(19) COLLATE 'MSFT_ENUS01252_0' NOT NULL, PRIMARY KEY(value));
```

次の ICU の例では、VARCHAR 列を持つテーブルを作成し、デフォルトの Linux エンコード UTF-8 であるとして、ICU 照合順序 u54-msft_enus_0 を使用します。

```
CREATE TABLE isrttest (ord INT, value VARCHAR(19) COLLATE 'u54-msft_enus_0' NOT NULL, PRIMARY KEY(value));
```

次の ICU の例は、NVARCHAR 列を持ち、ICU 照合順序 `u54-msft_enus_0` を使用するテーブルを作成します。

```
CREATE TABLE isrttest (ord INT, value NVARCHAR(19) COLLATE 'u54-msft_enus_0' NOT NULL, PRIMARY KEY(value));
```

関連項目

[ALTER TABLE](#)

[CREATE TABLE](#)

『*Advanced Operations Guide*』の[照合順序と並べ替えのサポート](#)

SET LEGACYTYPESALLOWED

SET LEGACYTYPESALLOWED ステートメントにより、本リリースの Zen ではサポートされなくなったデータ型との下位互換性を有効にすることができます。

構文

```
SET LEGACYTYPESALLOWED = < ON | OFF >
```

備考

SET LEGACYTYPESALLOWED ステートメントは SQL セッションで CREATE TABLE や ALTER TABLE ステートメントより先に実行され、以前のリリースの Zen でサポートされた旧データ型をこれらのステートメントで使えるようにします。

デフォルト値は OFF です。これは、旧データ型がサポートされないことを意味します。

詳細については、[旧データ型](#)を参照してください。

例

次の例では、CREATE TABLE ステートメントの前に LEGACYTYPESALLOWED をオンにして旧データ型が機能するようにし、テーブルの作成後、再びオフに戻しています。

```
SET LEGACYTYPESALLOWED=ON;  
CREATE TABLE notes (c1 INTEGER, c2 NOTE(20));  
SET LEGACYTYPESALLOWED=OFF;
```

メモ： 設定をオフにしなかった場合は、すべての SET コマンドと同様、その適用は SQL セッションで終わります。

SET OWNER

SET OWNER ステートメントは、現在のデータベース セッションで SQL コマンドがアクセスするファイルのオーナー ネームをリストアップします。このファイルレベルのセキュリティ機能の詳細については、[オーナー ネーム](#)を参照してください。

構文

```
SET OWNER = ['] オーナー ネーム ['] [ , ['] オーナー ネーム ['] ]...
```

備考

SET OWNER ステートメントでは、アルファベット以外の文字で始まるオーナー ネーム、およびスペースを含んでいる ASCII 形式のオーナー ネームは一重引用符で囲む必要があります。16 進数の長いオーナー ネームは 0x または 0X で始まるため、常に一重引用符が必要です。

SET OWNER ステートメントは、セッション内のデータ ファイルに必要とされるオーナー ネームの一覧を作成できます。リレーショナル エンジンでは、オーナー ネームをキャッシュしておき、MicroKernel エンジンからのファイル アクセス要求の必要に応じて使用します。

SET OWNER ステートメントは、現在の接続セッションでのみ有効です。ユーザーは、SET OWNER を発行後にログアウトした場合、次のログイン時にこのコマンドを再発行する必要があります。

SET OWNER ステートメントごとに、セッションの現在のオーナー ネーム リストはリセットされます。追加のステートメントにより、リストにオーナー ネームを追加することはできません。

セキュリティが無効になっているデータベースでは、SET OWNER ステートメントにより、このステートメントで指定したオーナー ネームと一致するオーナー ネームを持つすべてのデータ ファイルへのフル アクセスが許可されるようになります。

セキュリティが有効になっているデータベースでは、SET OWNER ステートメントは Master ユーザー以外のユーザーには何の影響もありません。Master ユーザーが自身に権限を付与していない場合は、SET OWNER を実行することにより、指定したオーナー ネームのいずれかを持つすべてのデータ ファイルへのフル アクセスが Master ユーザー

に与えられます。**Master** ユーザーは、他のユーザーについて、次の 2 つの方法のいずれかでアクセス権限を付与することができます。

- **SET OWNER** でオーナー ネームを指定し、続けて **GRANT** をオーナー ネームを指定しないで実行します。
- **GRANT** をオーナー ネームを指定して実行します。

これら 2 つのオプションについて、以下の例で説明します。

例

次の例では、オーナー ネームは数字で始まっているため、一重引用符で囲まれています。

```
SET OWNER = '1@1pha0m3gA'
```

次の例は、現在のセッションでアクセスするファイルによって使用されるオーナー ネームのリストを提供しています。

```
SET OWNER = 'server17 region5', '0x7374726f6e672050617373776f7264212425fe'
```

一重引用符が使用されている理由は、**ASCII** 文字列の場合はスペースが含まれているためであり、16 進数文字列の場合はプレフィックスの **0x** が数字で始まっているためです。

データベース セッション中、各 **SET OWNER** ステートメントは前の同ステートメントをオーバーライドします。次の例では、2 番目のコマンドを実行した後、最初の 3 つのオーナー ネームはファイル アクセスに使用できなくなります。

```
SET OWNER = judyann, krishna1, maxima  
SET OWNER = d3ltagamm@, V3rs10nXIII, m@X1mumSp33d
```

次の例は、**Master** ユーザーがセキュリティで保護されたデータベースで **SET OWNER** を使用する方法を示します。セキュリティは有効になっていますが、ユーザーには何の権限も付与されていません。**inventory1** というデータ ファイルはオーナー ネーム **admin** を持ちます。

Master ユーザーが自身に権限を付与する場合には、2 つのオプションがあります。1 目のオプションでは、**SET OWNER** を発行した後、**GRANT** をオーナー ネームなしで発行できます。


```
SET OWNER = admin  
GRANT ALL ON inventory1 TO MASTER
```

2 つ目のオプションでは、Master ユーザーは SET OWNER ステートメントを省略し、オーナー ネームを指定した GRANT を発行できます。

```
GRANT ALL ON inventory1 admin TO MASTER
```

どちらの方法も同じ結果になります。

関連項目

[GRANT](#)

[REVOKE](#)

SET PASSWORD

SET PASSWORD ステートメントは、セキュリティで保護されたデータベースに対して次の機能を提供します。

- Master ユーザーは、Master ユーザーまたはそれ以外のユーザーのパスワードを変更できます。
- 通常のユーザー (Master 以外のユーザー) は、データベースへのログオンパスワードを変更できます。

構文

```
SET PASSWORD [ FOR 'ユーザー名' ] = パスワード
```

ユーザー名 ::= データベースにログオンしているか、またはデータベースへのログオンを許可されているユーザーの名前

パスワード ::= パスワード文字列

備考

SET PASSWORD は、データベースでリレーショナルセキュリティが有効になっていることを必要とし、いつ発行してもかまいません対照的に、[SET SECURITY](#) は、Master ユーザーのセッションが現在の唯一のデータベース接続である場合にのみ発行されます。[SET SECURITY](#) を参照してください。

SET PASSWORD は、Master ユーザーでも Master 以外の通常のユーザーでも発行できます。Master ユーザーは、データベースへのログインを許可されているすべてのユーザーのパスワードを変更できます。通常のユーザーは自身のパスワードしか変更できません。パスワードの変更は、ユーザーが次回ログインしたときに有効になります。

SET PASSWORD ステートメントを発行するユーザー	FOR 句付き	FOR 句なし
Master	Master は、Master またはデータベースへのログインを許可されているすべてのユーザーのユーザー名を指定できます。 ¹ ユーザー名のパスワードが変更されます。	データベース全体のパスワードが変更されます (つまり、データベース全体に影響を与える、Master ユーザーのパスワードが変更されます)。

SET PASSWORD ステートメントを発行するユーザー	FOR 句付き	FOR 句なし
通常	通常ユーザーは、ユーザー名を指定できます。ユーザーはデータベースにログインする必要があります。当該ユーザーのパスワードのみ変更されます。	SET PASSWORD ステートメントを発行したユーザーのパスワードのみが変更されます。ユーザーはデータベースにログインする必要があります。

¹ ユーザー名は、Zen データベースにログオン可能なユーザーです。オペレーティングシステムのユーザー名とは異なる場合があります。たとえば、ユーザー名 **Yogine** はオペレーティングシステムにログオンできます。データベース **Demodata** でセキュリティが有効になっており、**Yogine** がユーザー名 **DeptMgr** (**Demodata** にログオンするユーザー名) として追加されています。

パスワードの特性

- パスワードの最大長と使用できる文字については、『*Advanced Operations Guide*』の **識別子の制限** を参照してください。
- パスワードでは大文字小文字が区別されます。パスワードがアルファベット以外の文字で始まる場合は、パスワードを一重引用符で囲む必要があります。
- 先頭文字以外であれば、空白文字をパスワードで使用することができます。パスワードに空白文字が含まれる場合は、パスワードを一重引用符で囲む必要があります。原則として、パスワードに空白文字を使用することは避けてください。
- "Password" は予約語ではありません。これは、テーブルまたは列の名前として使用することができます。ただし、SQL ステートメント内でテーブル名または列名として使用する場合は、"password" はキーワードであるため、二重引用符で囲む必要があります。
- リテラルの "null" をパスワードとして使用する場合は、単語を一重引用符で囲む必要があります ('null')。文字列を引用符で囲むことによって、データベースのセキュリティを無効にするための SET SECURITY = NULL ステートメントとの混乱を防ぎます。

例

次の例では以下のことを示します。Master ユーザーは、パスワード **bluesky** を使ってデータベースのセキュリティを有効にします。次に、Master ユーザーは **user45** という

ユーザーにログオン権限を与えてパスワードを `tmppword` と指定し、そのユーザーに `person` テーブルの `SELECT` 権限を与えます。その後で、Master ユーザーは Master のパスワードを `reddawn` に変更することで、データベース全体に対するパスワードを変更します。最後に、`user45` のパスワードを `newuser` に変更します。

```
SET SECURITY = bluesky
GRANT LOGIN TO user45:tmppword
GRANT SELECT ON person TO user45
SET PASSWORD = reddawn
SET PASSWORD FOR user45 = newuser
```

次の例では、`user45` はパスワード `newuser` でデータベースにログオンしているものとしてします。`user45` は自分のパスワードを `tomato` に変更し、その後、`person` テーブル内のすべてのレコードを選択しています。

```
SET PASSWORD FOR user45 = tomato
SELECT * FROM person
```

関連項目

[ALTER USER](#)

[CREATE USER](#)

[GRANT](#)

[SET SECURITY](#)

SET PROCEDURES_CACHE

SET PROCEDURES_CACHE ステートメントは、データベースエンジンがストアード プロシージャのキャッシュとして予約する SQL セッションのメモリ量を指定します。

構文

```
SET PROCEDURES_CACHE = メモリ量 (メガバイト)
```

備考

メモリ量（メガバイト）の値は、ゼロからおよそ 20 億までの整数です。データベースエンジンは、デフォルトで 5 MB に設定します。各セッションは、この SET ステートメントを発行することにより、そのセッションのキャッシュ メモリ量を変更できます。

SET PROCEDURES_CACHE の対となるステートメントは [SET CACHED_PROCEDURES](#) です。

- これら両方の SET ステートメントをゼロに設定すると、データベース エンジンはストアード プロシージャをキャッシュ**しません**。さらに、エンジンはストアード プロシージャに使用される既存のキャッシュを削除します。つまり、エンジンは、両方のステートメントをゼロに設定する前にキャッシュされたすべてのストアード プロシージャをキャッシュから消去します。
- 一方のステートメントだけをゼロまたはゼロ以外の値に設定した場合、もう一方のステートメントは暗黙的にゼロに設定されます。暗黙的にゼロに設定されたステートメントは無視されます。たとえば、キャッシュ メモリとして 30 MB を使用することのみに関心があり、キャッシュされるプロシージャ数に関心がなければ、PROCEDURES_CACHE を 30 に設定します。データベース エンジンは暗黙的に CACHED_PROCEDURES をゼロに設定します。つまり、設定を無視します。

PROCEDURES_CACHE をゼロ以外の値に設定すると、以下の条件が適用されます。プロシージャの実行によって割り当てられたメモリが PROCEDURES_CACHE 値を超える場合、データベース エンジンは最近最も使用頻度の低いプロシージャをキャッシュから削除します。

メモリ キャッシュを使用している場合、プロシージャの実行後もコンパイルしたストアード プロシージャを保持しています。一般的に、キャッシングによって、キャッシュされたプロシージャへの以降の呼び出しのパフォーマンスが向上します。キャッシュの設定やアプリケーションで実行される SQL ステートメントによっては、過度のメモリ

スワッピングやスラッシングが発生する可能性があるので注意してください。スラッシングはパフォーマンスを低下させる可能性があります。

レジストリ設定

キャッシュ用の予約メモリ量は、SET ステートメントのほかにレジストリ設定でも指定できます。レジストリ設定はすべてのセッションに適用され、初期値を設定する便利な方法です。各セッションは SET ステートメントを使用して、そのセッション固有の値を指定できます。この値はレジストリ設定より優先されます。

レジストリ設定は、Zen Enterprise Server または Cloud Server がサポートされるすべてのサーバー プラットフォームに適用されます。レジストリ設定は手作業で変更する必要があります。Windows では、オペレーティング システムで提供されているレジストリエディターを使用します。Linux では、**psregedit** ユーティリティを使用することができます。

レジストリ設定が指定されていない場合、データベース エンジンがデフォルトを自動的に 5 MB とします。

Windows のレジストリ設定でキャッシュ メモリ量を指定するには

1. 以下のキーを見つけます。

HKEY_LOCAL_MACHINE¥SOFTWARE¥Actian¥Zen¥SQL Relational Engine

メモ：ほとんどの Windows システムで、このキーの場所は

HKEY_LOCAL_MACHINE¥SOFTWARE¥Actian¥Zen です。ただし、

HKEY_LOCAL_MACHINE¥SOFTWARE の下位以降の場所はオペレーティング システムによって異なる可能性があります。

2. このキーに、**ProceduresCache** という新しい文字列値を作成します。
3. **ProceduresCache** にキャッシュに使用したいメモリ量を設定します。

Linux の Zen レジストリでキャッシュ メモリの量を設定するには

1. 以下のキーを見つけます。

PS_HKEY_CONFIG¥SOFTWARE¥Actian¥Zen¥SQL Relational Engine

2. このキーに、**ProceduresCache** という新しい文字列値を作成します。
3. **ProceduresCache** にキャッシュに使用したいメモリ量を設定します。

キャッシングの除外

ストアド プロシージャは、以下の条件に当てはまる場合、キャッシュ設定にかかわらず、キャッシュ**されません**。

- ストアド プロシージャが、ローカルまたはグローバル テンポラリ テーブルを参照する。ローカル テンポラリ テーブルはポンド記号 (#) で始まる名前を持ちます。グローバル テンポラリ テーブルは 2 つのポンド記号 (##) で始まる名前を持ちます。 **CREATE (テンポラリ) TABLE** を参照してください。
- ストアド プロシージャにデータ定義言語 (DDL) ステートメントが含まれている。 **データ定義ステートメント** を参照してください。
- ストアド プロシージャに、文字列または文字列を返す式を実行するための EXEC[UTE] ステートメントが含まれている。たとえば、次のような例です。 **EXEC ('SELECT Student_ID FROM ' + :myinputvar)**

例

次の例は、最大 20 のストアド プロシージャを格納する 2 MB のキャッシュ メモリを設定します。

```
SET CACHED_PROCEDURES = 20
SET PROCEDURES_CACHE = 2
```

次の例は、最大 500 のストアド プロシージャを格納する 1,000 MB のキャッシュ メモリを設定します。

```
SET CACHED_PROCEDURES = 500
SET PROCEDURES_CACHE = 1000
```

次の例は、ストアド プロシージャを**キャッシュせず**、既にキャッシュされているプロシージャを削除する指定をします。

```
SET CACHED_PROCEDURES = 0
SET PROCEDURES_CACHE = 0
```

次の例では、キャッシュ メモリを 80 MB に設定し、キャッシュされるプロシージャ数は設定しない指定をします。

```
SET PROCEDURES_CACHE = 80
```

データベース エンジン は暗黙的に CACHED_PROCEDURES をゼロに設定します。

関連項目

[CREATE PROCEDURE](#)

[SET CACHED_PROCEDES](#)

SET ROWCOUNT

SET ROWCOUNT キーワードを使用することによって、現在のセッション内にある後続のすべての SELECT ステートメントから返される行数を制限することができます。

SET ROWCOUNT と **TOP または LIMIT** の主な違いは、TOP が現在のステートメントにのみ作用するのに対し、SET ROWCOUNT は（次の SET ROWCOUNT またはセッションが終了するまでの）現在のデータベース セッション中に発行される後続のステートメントすべてに作用する点です。

構文

```
SET ROWCOUNT = 行数
```

備考

SET ROWCOUNT を条件とする SELECT ステートメントに ORDER BY キーワードが含まれており、その ORDER BY 句での最適化にインデックスを使用できない場合、Zen はテンポラリ テーブルを生成します。テンポラリ テーブルにはクエリの結果セット全体が置かれます。テンポラリ テーブル内の行は ORDER BY 値で指定した順序で並べられ、その順序付けされた結果セットから ROWCOUNT の n 行を返します。

ROWCOUNT をゼロに設定して ROWCOUNT 機能を無効にすることができます。

```
SET ROWCOUNT = 0
```

SET ROWCOUNT は動的カーソルを使用するときには無視されます。

SET ROWCOUNT と TOP の両方をクエリに適用した場合、2 つの値のうち小さい方の値に等しい行数を返します。

例

TOP または LIMIT の例も参照してください。

```
SET ROWCOUNT = 10;  
SELECT * FROM person;  
-- 10 行を返します
```

関連項目

TOP または LIMIT

SET SECURITY

SET SECURITY ステートメントを使用すると、Master ユーザーはログオンしているデータベースのセキュリティを有効または無効にできます。

構文

```
SET SECURITY [ USING 認証の種類 ] = < 'パスワード' | NULL >
```

備考

セキュリティを設定するには、まず Master としてログオンする必要があります。その後、SET SECURITY ステートメントを使用してパスワードを割り当てることができます。セキュリティで保護されていないデータベースにログオンする場合、Master はパスワードを要求されません。しかし、データベースにセキュリティを設定する場合は、Master ユーザーは割り当てられたパスワードが必要になります。

SET SECURITY は、Master ユーザーのセッションが現在の唯一のデータベース接続である場合のみ発行できます。セキュリティは Zen Control Center (ZenCC) から設定することもできます。『Zen User's Guide』の [Zen エクスプローラーを使用してセキュリティを有効にするには](#)を参照してください。

認証の種類の文字列は、local_db または domain です。USING 句が含まれていない場合、認証の種類は local_db に設定されます。

認証の種類が domain の場合に、ユーザーに関する SQL スクリプトを実行すると、エラー メッセージ（ステートメントがドメイン認証でサポートされていない）を返します。サポートされないステートメントは、ユーザー関連の ALTER USER、CREATE USER、DROP USER、GRANT や、非 Master ユーザー向けの SET PASSWORD、および REVOKE などです。

パスワード要件については、[パスワードの特性](#)を参照してください。

ユーザー権限

オブジェクト（テーブル、ビュー、およびストアド プロシージャなど）の権限は、SET SECURITY が NULL に設定された後もシステム テーブルに保持されます。次のシナリオを考えてみましょう。

- データベース mydbase のセキュリティを有効にし、ユーザー Master がログインします。

- Master が、ユーザー user1 と user2、およびデータベース mydbase のテーブル t1 を作成します。
- Master が user2 に t1 の SELECT 権限を付与します。
- mydbase のセキュリティを無効にします。
- テーブル t1 を削除します。

テーブル t1 が存在しなくなっても、t1 の権限はシステム テーブル内に残っています (t1 の ID は X\$Rights 内にまだあります)。以下について考えてみましょう。

- データベース mydbase のセキュリティを再び有効にします。
- user1 がデータベースにログインします。
- user1 が mydbase の新しいテーブル tbl1 を作成します。tbl1 には、t1 に割り当てられたオブジェクト ID と同じ ID を割り当てることができます。このシナリオでは、t1 と tbl1 に割り当てられるオブジェクト ID を同じにします。
- t1 の以前の権限が tbl1 に復元されます。つまり、user1 は、新しいテーブルに対する権限を明示的に付与されていなくても、tbl1 の SELECT 権限を持っているということです。

メモ： オブジェクトに対する権限を削除したい場合は、明示的にその権限を取り消す必要があります。このことは、テーブル、ビュー、およびストアド プロシージャに当てはまります。権限はオブジェクト ID と関連付けられており、データベースは新しいオブジェクトに対し、削除されたオブジェクトのオブジェクト ID を再使用するからです。

例

次の例では、データベースのセキュリティを有効にし、Master パスワードを "mypasswd" と設定しています。

```
SET SECURITY = 'mypasswd'
```

次の例では、データベースのドメイン認証を有効にし、Master パスワードを "123456" と設定しています。

```
SET SECURITY USING domain = '123456'
```

次の例ではセキュリティが無効になります。

```
SET SECURITY = NULL
```

関連項目

[ALTER USER](#)

[CREATE USER](#)

[GRANT](#)

[REVOKE](#)

[SET PASSWORD](#)

SET TIME ZONE

SET TIME ZONE キーワードを使用すると、ロケールの世界協定時刻（Coordinated Universal Time : UTC）を基に現在時刻のオフセットを指定することができ、これによって、データベース エンジンが配置されているオペレーティング システムのタイムゾーンの設定を無効にできます。

SET TIME ZONE ステートメントの効果は現在のデータベース セッションの間中、または別の SET TIME ZONE ステートメントが実行されるまで適用されます。

注意！ お使いのオペレーティング システムのタイムゾーンの設定を特に無効にする必要がなければ、常にデフォルトの動作を使用してください。DataExchange レプリケーションを使用している場合、またはアプリケーションに各レコードが挿入される時刻順序との依存関係がある場合は、SET TIME ZONE を使ってタイムゾーンのオフセットを変更することはお勧めしません。

構文

```
SET TIME ZONE < オフセット | LOCAL >
```

```
オフセット ::= <+|->hh:mm
```

hh の有効な範囲は 00 ～ 12 です。

mm の有効な範囲は 00 ～ 59 です。

プラス記号 (+) またはマイナス記号 (-) は、オフセット値の一部として必要です。

備考

デフォルトの動作 - SET TIME ZONE LOCAL がデフォルトの動作で、これは SET TIME ZONE コマンドをまったく使用しないのと同じです。デフォルトの動作の場合、データベース エンジンを実行しているオペレーティング システムに基づいてそのタイムゾーンを設定します。たとえば、SELECT CURTIME() では現在の現地時刻を返しますが、SELECT CURRENT_TIME() では、オペレーティング システムの地域のシステム時刻とタイムゾーンの設定の両方に基づいて現在の世界協定時刻（UTC）を返します。

LOCAL キーワードを使用すると、オフセット値を指定した後、データベース セッションを終了して再度開かなくても、デフォルトの動作に戻すことができます。

デフォルトの動作の場合、`1996-03-28` や `17:40:46` などの日付 / 時刻のリテラル値は、現在の現地日付 / 時刻として解釈されます。また、挿入中には、タイムスタンプリテラル値は指定した現在の現地時刻として解釈されます。タイムスタンプ値は常に UTC 時刻を使って調整されてから内部的に保存され、取得時に現地時刻に変換されます。詳細については、[タイムスタンプ値](#)を参照してください。

タイムゾーンが指定されていない場合、または TIME ZONE LOCAL が指定されている場合

<code>CURDATE()</code> 、 <code>CURTIME()</code> 、 <code>NOW()</code> 、 <code>SYSDATETIME()</code>	システム時計に基づいて現在の現地日付 / 時刻を返します。
<code>CURRENT_DATE()</code> 、 <code>CURRENT_TIME()</code> 、 <code>CURRENT_TIMESTAMP()</code> 、 <code>SYSUTCDATETIME()</code>	システム時計とオペレーティングシステムの地域の設定に基づいて、現在の UTC 日付 / 時刻を返します。

オフセットを指定した場合の動作 - 有効なオフセット値が指定されると、[CURDATE\(\)](#)、[CURTIME\(\)](#)、[NOW\(\)](#)、または [SYSDATETIME\(\)](#) の値を生成する場合に、オペレーティングシステムのタイムゾーンのオフセットではなくその値を使用します。たとえば、オフセットに `-02:00` を指定した場合、`CURDATE()` の現地時刻の値はオペレーティングシステムから返される UTC 時刻に `-02:00` が加算された値になります。

この動作の場合、日付と時刻のリテラルはそのままの値で現地時刻として解釈されます。タイムスタンプリテラルでは、オフセット値を減算した場合に結果が UTC となるような時刻を指定しているものと解釈されます。夏時刻はオフセットで明示的に設定するので、考慮しません。タイムスタンプ値は常に UTC 時刻を使って内部的に保存されます。

有効なオフセット値が指定された場合

<code>CURDATE()</code> 、 <code>CURTIME()</code> 、 <code>NOW()</code> 、 <code>SYSDATETIME()</code>	これらの関数は、現在の UTC 日付 / 時刻値に オフセット を加算して現在の現地日付 / 時刻値を返します。
<code>CURRENT_DATE()</code> 、 <code>CURRENT_TIME()</code> 、 <code>CURRENT_TIMESTAMP()</code> 、 <code>SYSUTCDATETIME()</code>	これらの関数は、常にシステム時計とオペレーティングシステムの地域の設定に基づいて現在の UTC 日付 / 時刻を返します。

指定した現地時刻値を UTC に変換するには、その現地時刻の値からタイムゾーンのオフセット値を減算する必要があります。つまり次のようになります。

UTC 時刻 = 現地時刻 - タイムゾーンのオフセット

この表は、変換の例を示しています。

現地時刻	オフセット	UTC
10:10:15 オースチン	US 中部標準時 -06:00	10:10:15-(-06:00)=16:10:15 UTC
16:10:15 ロンドン	グリニッジ標準時 +00:00	16:10:15-(+00:00)=16:10:15 UTC
22:10:15 ダッカ	+06:00	22:10:15-(+06:00)=16:10:15 UTC

タイムスタンプ データ型に関する注記

タイムスタンプ データは常に UTC として保存され、リテラル タイムスタンプ値はディスクに保存されている値も含め、取得するときに常に現地時刻へ変換されるため、[NOW\(\)](#) 値と [CURRENT_TIMESTAMP\(\)](#) 値の動作は混乱を招く可能性があります。たとえば、次の表ではデータベース エンジンが U.S. 中部標準時に設定されているとします。

ステートメント	値
SELECT NOW()	2001-10-01 12:05:00.123 が表示されます。
INSERT INTO t1 (c1) SELECT NOW()	2001-10-01 18:05:00.1234567 がディスクに保存されます。
SELECT * from t1	2001-10-01 12:05:00.123 が表示されます。
SELECT CURRENT_TIMESTAMP()	2001-10-01 18:05:00.123 が表示されます。
INSERT INTO t2 (c1) SELECT CURRENT_TIMESTAMP()	2001-10-01 18:05:00.1234567 がディスクに保存されます。
SELECT * from t2	2001-10-01 12:05:00.123 が表示されます。

直接の SELECT [NOW\(\)](#) で表示される値は、INSERT SELECT [NOW\(\)](#) 構文によってディスクに保存される値とは異なることに注意することが重要です。また、SELECT [CURRENT_TIMESTAMP\(\)](#) の表示値も、CURRENT_TIMESTAMP() の値を INSERT した後でそれを SELECT した場合 (INSERT SELECT CURRENT_TIMESTAMP() 構文) に表示される値とは異なることに注意してください。これは、データ ファイルに保存したリテラル値を取得するときに値が調整されるからです。

例

次の例では、SET TIME ZONE ステートメントがまだ発行されておらず、データベースエンジンが起動しているシステムのシステム時計は January 9, 2002, 16:35:03 CST (US) に設定されています。CURRENT_TIMESTAMP() およびその他の CURRENT_ 関数では、常にデータベースエンジンが起動しているシステムのシステム時計と地域の設定に基づく UTC 日付 / 時刻を返すことを思い出してください。

```
SELECT CURRENT_TIMESTAMP(), NOW(),
       CURRENT_TIME(), CURTIME(),
       CURRENT_DATE(), CURDATE()
```

結果：

```
2002-01-09 22:35:03.000 2002-01-09 16:35:03.000
22:35:03                16:35:03
01/09/2002              01/09/2002
```

CST は UTC の 6 時間前であることを注意してください。

```
SET TIME ZONE -10:00
```

これによって、上記の同じ SELECT ステートメントでは次の値を返します。

```
2002-01-09 22:35:03.000 2002-01-09 12:35:03.000
22:35:03                12:35:03
2002-01-09              2002-01-09
```

SET TIME ZONE ステートメントを発行後 NOW() の値は変わりましたが、CURRENT_TIMESTAMP() の値は変わらなかったことに注目してください。

次の例では、UTC 値として保存され取得時に現地時刻の値に変換されるタイムスタンプ値と、そのままの値で保存および取得される TIME または DATE 値との違いを示します。現在、システム時計は January 9, 2002, 16:35:03 CST (U.S.) を示しているものとして示します。また、SET TIME ZONE ステートメントはまだ発行されていないことを前提とします。

```
CREATE TABLE t1 (c1 TIMESTAMP, c2 TIMESTAMP, c3 TIME, c4 TIME, c5 DATE, c6 DATE)
INSERT INTO t1 SELECT CURRENT_TIMESTAMP(), NOW(), CURRENT_TIME(), CURTIME(), CURRENT_DATE(),
CURDATE()
SELECT * FROM t1
```

結果：

```
c1                c2
-----
2002-01-09 16:35:03.000 2002-01-09 16:35:03.000

c3      c4      c5      c6
-----
```

```
22:35:03 16:35:03 01/09/2002 01/09/2002
```

NOW() と CURRENT_TIMESTAMP() は、SELECT NOW()、CURRENT_TIMESTAMP() で画面に表示される場合には異なる値になりますが、いったんリテラル値をディスクに保存すると、両方の値に UTC 時刻が格納されることに注意してください。取得時に、両方の値が現地時刻に変換されます。

タイムゾーン オフセットをゼロに設定すると、取得時に +00:00 によって調整されるので、ファイルに保存された実際のデータを見ることができます。

```
SET TIME ZONE +00:00
SELECT * FROM t1
```

結果：

c1	c2
2002-01-09 22:35:03.000	2002-01-09 22:35:03.000

c3	c4	c5	c6
22:35:03	16:35:03	01/09/2002	01/09/2002

次の例では、現地日付が UTC 日付と異なる場合に予期される動作を示します（たとえば、UTC は午前 0 時を過ぎているが、現地時刻ではまだ午前 0 時前の場合。またはその逆の場合）。現在、システム時計は January 9, 2002, 16:35:03 CST (U.S.) を示しているものとします。

```
SET TIME ZONE +10:00
SELECT CURRENT_TIMESTAMP(), NOW(),
       CURRENT_TIME(), CURTIME(),
       CURRENT_DATE(), CURDATE()
```

結果：

2002-01-09 22:35:03.000	2002-01-10 08:35:03.000
22:35:03	08:35:03
01/09/2002	01/10/2002


```
INSERT INTO t1 SELECT CURRENT_TIMESTAMP(), NOW(), CURRENT_TIME(), CURTIME(), CURRENT_DATE(),
CURDATE()

SELECT * FROM t1
```

結果：

c1	c2
2002-01-10 08:35:03.000	2002-01-10 08:35:03.000

c3	c4	c5	c6
22:59:55	08:59:55	01/09/2002	01/10/2002

ご覧のとおり、`CURRENT_DATE()` および `CURRENT_TIME()` で返される UTC 時刻と日付はリテラル値として保存されます。これらはタイムスタンプ値ではないので、データベースから取得するときに調整が行われません。

関連項目

[TIMESTAMP データ型](#)

[時刻と日付の関数](#)

SET TRUEBITCREATE

SET TRUEBITCREATE ステートメントを使用すると、BIT データ型にインデックスを付けたり、LOGICAL トランザクショナル データ型にマップすることができます。

構文

```
SET TRUEBITCREATE = < ON | OFF >
```

備考

デフォルトは on です。これは、BIT データ型が 1 ビットで、インデックスを付けることができず、Zen 型コード 16 に割り当てられることを意味します。型コード 16 の場合、BIT をマップできる同等のトランザクショナル データ型はありません。

ほかの DBMS アプリケーションとの互換性などの一定の状況では、BIT を LOGICAL データ型にマップして BIT データ型にインデックスを付けたい場合があります。これを行うには、TRUEBITCREATE を off にします。これにより BIT は LOGICAL にマップされ、型コード 7 の 1 バイト データ型になります。

作成モードは、ステートメントを発行して変更されるか、データベース接続が切断されるまでは有効なままです。この設定は接続ごとに保持されるため、同一アプリケーション内であっても、個々のデータベース接続がそれぞれ異なる作成モードを保持することができます。どの接続も、BIT が Zen 型コード 16 で作成されるデフォルト モードの設定で開始されます。

この機能は既存の BIT には影響せず、SET ステートメントがアプリケーション適用された後に作成されたもののみに有効になります。

この設定は SQL ステートメントでのみ切り替えることができます。Zen Control Center では設定できません。Table Editor は列のリレーショナル データ型を表示することに留意してください（したがって、"BIT" 型と表示されます）。TRUEBITCREATE を off にすると、Table Editor で BIT 列にインデックスを付けることができます。

例

次のステートメントは、設定を切り替えて、新しい BIT を作成する必要があることを指定します。新しい BIT は、インデックス付けが可能で、LOGICAL トランザクショナル データ型にマップされ、型コード 7 を持ちます。

```
SET TRUEBITCREATE=OFF
```

SET TRUENULLCREATE

SET TRUENULLCREATE ステートメントにより、新規テーブルの作成時に真のヌルのオン/オフを切り替えます。

構文

```
SET TRUENULLCREATE = < ON | OFF >
```

備考

この設定が初めて登場したのは Pervasive.SQL 2000 (7.5) でした。デフォルトはオンで、これは、空のフィールドの先頭にヌル インジケータ バイトを付けてテーブルが作成されるようにします。SQL ステートメントによってこれをオフに設定すると、それ以降は、Pervasive.SQL 7 以前のリリースのレガシー ヌルを使用してテーブルが作成されます。レガシー ヌルの動作は、セッションが切断されるまで持続します。新しいセッションでは、設定は再びオンになります。

接続ごとに TRUENULLCREATE 設定を持つため、同一のアプリケーション内で他の接続と異なる設定にすることができます。

レガシー ヌルは真のヌルではありませんが、ヌル許容として動作するため、あらゆる列の型に INSERT NULL を実行できます。しかし、値を照会すると、次に示す非ヌルのバイナリ同値のいずれかが返されます。

- 0 - Binary 型の場合
- 空文字列 - STRING 型および BLOB 型 (LVAR、LSTRING などのレガシー型を含む) の場合

したがって、このバイナリ同値を WHERE 句で使用して、特定の値を取得する必要があります。

次の表は、デフォルト値とヌル値を許可する列との相互作用を示しています。

列のタイプ	列にリテラルのデフォルト値が定義されていない場合に使用するデフォルト値	リテラル値が定義されている場合のデフォルト値
ヌル値を許可	NULL	定義どおり

列のタイプ	列にリテラルのデフォルト値が定義されていない場合に使用するデフォルト値	リテラル値が定義されている場合のデフォルト値
NOT NULL	エラー - " 列にデフォルト値が割り当てられていません。 "	定義どおり
v7.5 以前のヌル許容	列のレガシー ヌル	定義どおり

ステートメントで、デフォルト値が定義されている NOT NULL 列に明示的なヌルを挿入しようとする、エラーが発生して失敗します。挿入を試みた場所でデフォルト値は使用されません。

デフォルト値が定義されている列については、INSERT ステートメントの中で、挿入する列リストからこの列を省略するか、または挿入値の場所に DEFAULT キーワードを使用することで、その値を呼び出すことができます。

テーブル内の列がすべて、ヌル値を許可するかデフォルト値が定義されているかのいずれかである場合は、values 句に DEFAULT VALUES を使用することで、すべてのデフォルト値を含むレコードを挿入できます。一部の列がヌル値を許可していないかデフォルト値が定義されていない場合、または列リストを指定しない場合には、この種の句は使用できません。

BLOB、CLOB、または BINARY データ型に対する DEFAULT VALUES の使用は、現在のところサポートされていません。

例

設定を切り替えて、現在のセッションではレガシー ヌル サポートを使って新規テーブルが作成されるようにするには、次のステートメントを使用します。

```
SET TRUENULLCREATE=OFF
```

エンジンをデフォルト設定に戻して、現在のセッションでは真のヌル サポートを使ってテーブルが作成されるようにするには、次のステートメントを使用します。

```
SET TRUENULLCREATE=ON
```

SIGNAL

備考

SIGNAL ステートメントを使って、例外状態や正常終了以外の終了状態を通知することができます。

SQLSTATE 値を通知すると、SQLSTATE が特定の値に設定されます。その後、この値はユーザーに返されるか、呼び出し元プロシージャで SQLSTATE 値を介して使用できるようになります。この値は、プロシージャを呼び出しているアプリケーションで使用できます。

SQLSTATE 値と一緒にエラー メッセージを指定することもできます。

メモ： SIGNAL はストアド プロシージャまたはユーザー定義関数の内部でのみ使用できます。

構文

SIGNAL SQLSTATE 値 [, エラー メッセージ]

SQLSTATE 値 ::= ユーザー定義値

エラー メッセージ ::= ユーザー定義メッセージ

例

次の例では、SQLSTATE の初期値 00000 を出力し、通知された後に "SQLSTATE 例外が見つかりました" を出力します。最後に出力される SQLSTATE は W9001 になります。

```
CREATE PROCEDURE GenerateSignal();
BEGIN
    SIGNAL 'W9001';
END;

CREATE PROCEDURE TestSignal() WITH DEFAULT HANDLER;
BEGIN
    PRINT SQLSTATE;
    CALL GenerateSignal();
    IF SQLSTATE <> '00000' THEN
        PRINT 'SQLSTATE 例外が見つかりました';
    END IF;
    PRINT SQLSTATE;
END;
```

```
CREATE PROCEDURE GenerateSignalWithErrorMsg();
```



```
BEGIN
    SIGNAL 'W9001', '構文が無効です';
END;
CALL GenerateSignalWithErrMsg();
```

関連項目

[CREATE PROCEDURE](#)

SQLSTATE

備考

SQLSTATE 値は、正常終了、警告、または例外の状態を表します。ODBC で定義される SQLSTATE 値の完全な一覧は、Microsoft ODBC のドキュメントに記載されています。

ハンドラーが実行されるとき、そのハンドラー内のステートメントは、複合ステートメントの本体内のステートメントと同じように SQLSTATE 値に影響します。ただし、特定の状態に対して特定の動作を行うハンドラーは、完了時にその状態を再割り当てして、状態に影響を与えないようにすることもできます。このとき、ハンドラーは再呼び出しされません。再呼び出しはループの原因となります。その代わりに Zen では、例外状態が未処理の例外として扱われ、実行が停止します。

関連項目

[CREATE PROCEDURE](#)

[SELECT](#)

[SIGNAL](#)

START TRANSACTION

START TRANSACTION は、論理トランザクションの開始を知らせるものであり、常に、COMMIT または ROLLBACK と組み合わせて使用する必要があります。

構文

```
START TRANSACTION
SQL ステートメント
COMMIT | ROLLBACK [WORK]
```

備考

START TRANSACTION はストアド プロシージャ内でのみサポートされます。SQL Editor では START TRANSACTION を使用できません。SQL Editor は AUTOCOMMIT をオンに設定します。

例

次の例では、ストアド プロシージャの中で、Billing テーブル内の Amount_Owed 列を更新するトランザクションが開始されます。この作業がコミットされると、別のトランザクションによって Amount_Paid 列が更新されてゼロに設定されます。最後の COMMIT WORK ステートメントにより、2 番目のトランザクションが終了します。

```
START TRANSACTION;
  UPDATE Billing B
  SET Amount_Owed = Amount_Owed - Amount_Paid
  WHERE Student_ID IN (SELECT DISTINCT E.Student_ID
  FROM Enrolls E, Billing B WHERE E.Student_ID = B.Student_ID);
  COMMIT WORK;
  START TRANSACTION;
  UPDATE Billing B
  SET Amount_Paid = 0
  WHERE Student_ID IN (SELECT DISTINCT E.Student_ID
  FROM Enrolls E, Billing B WHERE E.Student_ID = B.Student_ID);
  COMMIT WORK;
```

関連項目

[COMMIT](#)

[CREATE PROCEDURE](#)

[ROLLBACK](#)

UNION

備考

UNION または UNION ALL を使用する SELECT ステートメントを使って、複数の SELECT クエリから 1 つの結果テーブルを取得できます。UNION クエリは、2 つ以上のデータ ソースに含まれる類似した情報を結合して 1 つにするのに適しています。

UNION を使用すると、重複行が削除されます。UNION ALL を使用すると、重複行が保持されます。重複行を削除する必要がない限り、UNION ALL オプションの使用をお勧めします。

UNION では、Zen データベース エンジン は結果セット全体を整列させるため、テーブルが大きいと処理に数分かかることがあります。UNION ALL では、ソートは不要になります。

Zen データベースは、UNION ステートメントでの LONGVARBINARY 列の使用をサポートしていません。UNION ステートメントでは、LONGVARCHAR および NLONGVARCHAR は 65500 バイトまでに制限されています。演算子 UNION は、1 つ以上のビューを参照する SQL ステートメントに適用することはできません。

UNION に含まれる 2 つのクエリ スペックは互換性がなければなりません。それぞれのクエリは同数の列を持ち、それらの列は互換性のあるデータ型である必要があります。

UNION キーワードの後にくる SELECT ステートメントの ORDER BY 句で、最初の SELECT リストにある列名を使用できます。また、序数を使って目的の列を示すこともできます。たとえば、次のステートメントは有効です。

```
SELECT c1, c2, c3 FROM t1 UNION SELECT c4, c5, c6 FROM t2 ORDER BY t1.c1, t1.c2, t1.c3
```

```
SELECT c1, c2, c3 FROM t1 UNION SELECT c4, c5, c6 FROM t2 ORDER BY 1, 2, 3
```

また、列名のエイリアスも使用できます。

```
SELECT c1 x, c2 y, c3 z FROM t1 UNION SELECT c1, c2, c3 FROM t2 ORDER BY x, y, z
```

```
SELECT c1 x, c2 y, c3 z FROM t1 a UNION SELECT c1, c2, c3 FROM t1 b ORDER BY a.x, a.y, a.z
```

エイリアスは、クエリ内のどのテーブル名、列名とも異なっている必要があります。

例

次の例では、名字が "M" で始まるか、成績評価点平均が 4.0 の生徒の ID 番号の一覧が示されます。結果テーブルに重複する行は含まれません。

```
SELECT Person.ID FROM Person WHERE Last_name LIKE 'M%' UNION SELECT Student.ID FROM Student WHERE Cumulative_GPA = 4.0
```

次の例では、重複行を含めた **person** テーブルおよび **faculty** テーブルの列 **ID** が抽出されます。

```
SELECT person.id FROM person UNION ALL SELECT faculty.id from faculty
```

次の例では、名字が "M" で始まるか、成績評価点平均が 4.0 の生徒の ID 番号の一覧が示されます。結果テーブルには重複行が含まれず、最初の列に従って結果セットが並べられます。

```
SELECT Person.ID FROM Person WHERE Last_name LIKE 'M%' UNION SELECT Student.ID FROM Student WHERE Cumulative_GPA = 4.0 ORDER BY 1
```

NULL スカラー関数を使用すると、UNION 選択リストに親選択リストとは異なる数のエントリを持たせることができます。これを行うには、CONVERT 関数を使用して NULL を強制的に正しい型に変換する必要があります。

```
CREATE TABLE t1 (c1 INTEGER, c2 INTEGER)
INSERT INTO t1 VALUES (1,1)
CREATE TABLE t2 (c1 INTEGER)
INSERT INTO t2 VALUES (2)
SELECT c1, c2 FROM t1
UNION SELECT c1, CONVERT(NULL(),sql_integer)FROM t2
```

関連項目

SELECT

UNIQUE

備考

インデックスに重複する値を許可しないことを指定するには、UNIQUE キーワードを使用します。UNIQUE キーワードを使用して CREATE INDEX ステートメントを実行したときにインデックスを構成する 1 つまたは複数の列に重複する値があった場合、Zen はステータス コード 5 を返し、インデックスを作成しません。

メモ： UNIQUE キーワードは、インデックスの属性リストの中で、指定する列名の後に置かないでください。推奨される構文は CREATE UNIQUE INDEX です。

関連項目

[ALTER TABLE](#)

[CREATE INDEX](#)

[CREATE TABLE](#)

UNMASK

『*Advanced Operations Guide*』の [Zen データベースのデタマスキング](#) を参照してください。

UPDATE

UPDATE ステートメントを使用することにより、データベース内の列の値を変更することができます。

構文

```
UPDATE < テーブル名 | ビュー名 > [ エイリアス名 ]
  SET 列名 = < NULL | DEFAULT | 式 | サブクエリ式 > [ , 列名 = ... ]
  [ FROM テーブル参照 [ , テーブル参照 ]...
  [ WHERE 検索条件 ]
```

テーブル名 ::= ユーザー定義名

ビュー名 ::= ユーザー定義名

エイリアス名 ::= ユーザー定義名 （エイリアス名は、FROM 句が使用されている場合には使用できません。[FROM 句](#)を参照してください。）

```
テーブル参照 ::= { OJ 外部結合定義 }
| [ データベース名 . ] テーブル名 [ [ AS ] エイリアス名 ]
| [ データベース名 . ] ビュー名 [ [ AS ] エイリアス名 ]
| 結合定義
| ( 結合定義 )
| ( テーブルサブクエリ ) [ AS ] エイリアス名 [ ( 列名 [ , 列名 ]... ) ]
```

外部結合定義 ::= テーブル参照 外部結合タイプ JOIN テーブル参照 ON 検索条件

外部結合タイプ ::= LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER]

```
検索条件 ::= 検索条件 AND 検索条件
| 検索条件 OR 検索条件
| NOT 検索条件
| ( 検索条件 )
| 述部
```

データベース名 ::= ユーザー定義名

ビュー名 ::= ユーザー定義名

```
結合定義 ::= テーブル参照 [ 結合タイプ ] JOIN テーブル参照 ON 検索条件
| テーブル参照 CROSS JOIN テーブル参照
| 外部結合定義
```

結合タイプ ::= INNER | LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER]

```
テーブル サブクエリ ::= クエリ スペック [ [ UNION [ ALL ]  
クエリ スペック ]... ]
```

```
サブクエリ式 ::= ( クエリ スペック )
```

備考

UPDATE ステートメントは、DELETE および INSERT と同様にアトミックな方法で動作します。つまり、複数の行の更新に失敗した場合、同じステートメントによって実行された前の行の更新がすべてロールバックされます。

UPDATE ステートメントの SET 句にはサブクエリを指定できます。この機能を使用すると、テーブル内の情報を、別のテーブル内のデータまたは同じテーブル内の別の部分を基に更新することができます。

キーワード DEFAULT を使用すると、その値を、指定の列に定義されているデフォルト値に設定することができます。デフォルト値が定義されていないとき、その列がヌル値を許可する場合はヌルを用し、ヌル値を許可しない場合はエラーを返します。デフォルト値と、真のヌルおよび古いリリースのレガシーヌルの詳細については、[SET TRUENULLCREATE](#) を参照してください。

UPDATE ステートメントは一度に 1 つのテーブルしか更新できません。UPDATE は、SET 句内のサブクエリを使ってほかのテーブルと関連付けることができます。これは、更新するテーブルの内容の一部に依存する相関サブクエリか、または別のテーブルにのみ依存する非相関サブクエリにすることができます。

たとえば、次は相関サブクエリです。

```
UPDATE t1 SET t1.c2 = (SELECT t2.c2 FROM t2 WHERE t2.c1 = t1.c1)
```

非相関サブクエリと比較してみましょう。

```
UPDATE t1 SET t1.c2 = (SELECT SUM(t2.c2) FROM t2 WHERE t2.c1 = 10)
```

純粋な SELECT ステートメントとサブクエリは同じロジックを使って処理されるため、サブクエリを有効な SELECT ステートメントで構成することができます。サブクエリには特別な規則はありません。

UPDATE 内の SELECT が行を返さない場合、UPDATE はヌルを挿入します。指定した列がヌル値を許可しない場合、UPDATE は失敗します。また、SELECT が複数行を返した場合、UPDATE は失敗します。

UPDATE ステートメントは、ステートメント内でのテーブル結合の使用を許可しません。その代わりに、上の例で示したように SET 句で相関サブクエリを使用します。

真のヌルとレガシーヌルの詳細については、[SET TRUENULLCREATE](#) を参照してください。

リテラル文字列の最大長より長いデータの更新

Zen でサポートされるリテラル文字列の最大長は 15,000 バイトです。これよりも長いデータを処理するには、直接の SQL ステートメントを使用し、更新を複数の呼び出しに分割します。次のようなステートメントで開始します。

```
UPDATE table1 SET longfield = '15000 バイトのテキスト' WHERE 制限
```

次に、それ以上のデータを追加する次のステートメントを発行します。

```
UPDATE table1 SET longfield = notefield + '15000 バイトを超えるテキスト' WHERE 制限
```

FROM 句

オプションの FROM 句と、更新されるテーブル（「更新テーブル」と呼びます）への参照に関して混乱が生じる可能性があります。FROM 句に更新テーブルが現れる場合、その出現のうちの 1 つは更新されるテーブルと同じインスタンスになります。

たとえば、ステートメント `UPDATE t1 SET c1 = 1 FROM t1, t2 WHERE t1.c2 = t2.c2` の場合、UPDATE の直後の t1 と FROM の後の t1 は、テーブル t1 の同じインスタンスです。したがって、このステートメントは `UPDATE t1 SET c1 = 1 FROM t2 WHERE t1.c2 = t2.c2` と同じことになります。

FROM 句に更新テーブルが複数回現れる場合、その出現のうちの 1 つは更新テーブルと同じインスタンスであると識別される必要があります。FROM 句の参照のうち、更新テーブルと同じインスタンスであると見なされるのは、エイリアスが指定されていない参照です。

したがって、ステートメント `UPDATE t1 SET t1.c1 = 1 FROM t1 a, t1 b WHERE a.c2 = b.c2` の場合、FROM 句の t1 のインスタンスはどちらもエイリアスを持っているため、これは正しくありません。次であれば有効です。 `UPDATE t1 SET t1.c1 = 1 FROM t1, t1 b WHERE t1.c2 = b.c2`

FROM 句には次の条件が適用されます。

- UPDATE ステートメントにオプションの FROM 句を含める場合、FROM 句の前にあるテーブル参照にエイリアスを指定することはできません。たとえば、`UPDATE t1 a SET a.c1 = 1 FROM t2 WHERE a.c2 = t2.c2` とすると、次のエラーが返されます。

```
SQL_ERROR (-1)
```

```
SQLSTATE "37000"
```

"オプションの FROM を伴う UPDATE/DELETE ステートメントでは、テーブルエイリアスは使用できません。"

ステートメントの有効なバージョンは、`UPDATE t1 SET t1.c1 = 1 FROM t2 WHERE t1.c2 = t2.c2` または `UPDATE t1 SET t1.c1 = 1 FROM t1 a, t2 WHERE a.c2 = t2.c2` です。

- FROM 句に更新テーブルへの参照を 2 つ以上含める場合、それらの参照のうちの 1 つにだけエイリアスを指定できます。たとえば、`UPDATE t1 SET t1.c1 = 1 FROM t1 a, t1 b WHERE a.c2 = b.c2` とすると、次のエラーが返されます。

```
SQL_ERROR (-1)
```

```
SQLSTATE "37000"
```

"テーブル t1 があいまいです。"

誤りのあるステートメントでは、エイリアスを "a" とするテーブル t1 が更新テーブルと同じインスタンスであると仮定しています。正しいステートメントは、`UPDATE t1 SET t1.c1 = 1 FROM t1, t1 b WHERE t1.c2 = b.c2` です。

- UPDATE ステートメントにおける FROM 句はセッションレベルでのみサポートされます。UPDATE ステートメントがストアド プロシージャ内で発生する場合は、FROM 句はサポート **されません**。

例

次の例では、Faculty テーブルの ID 103657107 に給与として 95000 を設定し、レコードを更新します。

```
UPDATE Faculty SET salary = 95000.00 WHERE ID = 103657107
```

次の例では、DEFAULT キーワードの使い方を示します。

```
UPDATE t1 SET c2 = DEFAULT WHERE c2 = 'bcd'
UPDATE t1 SET c1 = DEFAULT, c2 = DEFAULT
```

次の例では、Course テーブルの ECO 305 の履修単位時間を 4 に変更します。

```
UPDATE Course SET Credit_Hours = 4 WHERE Name = 'ECO 305'
```

次の例は、Person テーブル中のある人物の住所を変更します。

```
UPDATE Person p
SET p.Street = '123 Lamar',
    p.zip = '78758',
    p.phone = 5123334444
WHERE p.ID = 131542520
```

サブクエリ例 A

2つのテーブルが作成され、行が挿入されます。最初のテーブル t5 は、列 c1 に値 2 を持つ各行が、2 番目のテーブル t6 の列の値で更新されます。テーブル t6 には列 c2 に値 3 を含む行が 2 行以上あるために、サブクエリから複数行が返されることから、最初の UPDATE は失敗します。たとえ結果値がどちらの場合も同じになるとしても、このような結果になります。2 番目の UPDATE で示されるように、サブクエリ内で DISTINCT キーワードを使用すると、重複する結果が取り除かれ、ステートメントは成功します。

```
CREATE TABLE t5 (c1 INT, c2 INT)
CREATE TABLE t6 (c1 INT, c2 INT)
INSERT INTO t5(c1, c2) VALUES (1,3)
INSERT INTO t5(c1, c2) VALUES (2,4)
INSERT INTO t6(c1, c2) VALUES (2,3)
INSERT INTO t6(c1, c2) VALUES (1,2)
INSERT INTO t6(c1, c2) VALUES (3,3)
SELECT * FROM t5
```

結果：

c1	c2
1	3
2	4

```
UPDATE t5 SET t5.c1=(SELECT c2 FROM t6 WHERE c2=3) WHERE t5.c1=2 -- 注意：クエリは失敗します
UPDATE t5 SET t5.c1=(SELECT DISTINCT c2 FROM t6 WHERE c2=3) WHERE t5.c1=2 -- 注意：クエリは成功します
SELECT * FROM t5
```

結果：

c1	c2
1	3
3	4

サブクエリ例 B

2つのテーブルが作成され、有効な構文のさまざまな例が示されます。サブクエリが複数行を返すことにより UPDATE が失敗する事例に注目してください。また、サブクエリが行を返さない場合に（そこでヌル値が許可される場合）、ヌルが挿入されて UPDATE は成功する点にも注目してください。

```
CREATE TABLE t1 (c1 INT, c2 INT)
CREATE TABLE t2 (c1 INT, c2 INT)
INSERT INTO t1 VALUES (1, 0)
INSERT INTO t1 VALUES (2, 0)
INSERT INTO t1 VALUES (3, 0)
INSERT INTO t2 VALUES (1, 100)
INSERT INTO t2 VALUES (2, 200)
```

```
UPDATE t1 SET t1.c2 = (SELECT SUM(t2.c2) FROM t2)
UPDATE t1 SET t1.c2 = 0
UPDATE t1 SET t1.c2 = (SELECT t2.c2 FROM t2 WHERE t2.c1 = t1.c1)
UPDATE t1 SET t1.c2 = @@IDENTITY
UPDATE t1 SET t1.c2 = @@ROWCOUNT
UPDATE t1 SET t1.c2 = (SELECT @@IDENTITY)
UPDATE t1 SET t1.c2 = (SELECT @@ROWCOUNT)
UPDATE t1 SET t1.c2 = (SELECT t2.c2 FROM t2) -- 更新は失敗します
INSERT INTO t2 VALUES (1, 150)
INSERT INTO t2 VALUES (2, 250)
UPDATE t1 SET t1.c2 = (SELECT t2.c2 FROM t2 WHERE t2.c1 = t1.c1) -- 更新は失敗します
UPDATE t1 SET t1.c2 = (SELECT t2.c2 FROM t2 WHERE t2.c1 = 5) -- 更新は成功し、t1.c2 の全行にヌルが挿入されます
UPDATE t1 SET t1.c2 = (SELECT SUM(t2.c2) FROM t2 WHERE t2.c1 = t1.c1)
```

次の例では、テーブル t1 と t2 を作成し、それらにデータを設定します。UPDATE ステートメントで FROM 句を使用して、新しい値を取得するための別のテーブルを指定します。

```
DROP table t1
CREATE table t1 (c1 integer, c2 integer)
INSERT INTO t1 VALUES (0, 10)
INSERT INTO t1 VALUES (0, 10)
INSERT INTO t1 VALUES (2, 20)
INSERT INTO t1 VALUES (2, 20)
DROP table t2
CREATE table t2 (c1 integer, c2 integer)
INSERT INTO t2 VALUES (2, 20)
INSERT INTO t2 VALUES (2, 20)
INSERT INTO t2 VALUES (3, 30)
INSERT INTO t2 VALUES (3, 30)
UPDATE t1 SET t1.c1 = t2.c1 FROM t2 WHERE t1.c2 = t2.c2
SELECT * FROM t1
```

関連項目

[ALTER TABLE](#)

[CREATE PROCEDURE](#)

[CREATE TRIGGER](#)

[DEFAULT](#)

[GRANT](#)

[INSERT](#)

UPDATE（位置付け）

UPDATE（位置付け）ステートメントにより、SQL カーソルに関連付けられた行セットの現在の行を更新します。

構文

```
UPDATE [ テーブル名 ] SET 列名 = プロシージャ式 [ , 列名 = プロシージャ式 ]...  
WHERE CURRENT OF カーソル名
```

テーブル名 ::= ユーザー定義名

カーソル名 ::= ユーザー定義名

備考

このステートメントは、ストアド プロシージャ、トリガー、およびセッション レベルでのみ使用できます。

メモ： セッション レベルでは位置付け UPDATE は使用できますが、DECLARE CURSOR ステートメントは使用できません。アクティブな結果セットのカーソル名を取得する方法は、アプリケーションが使用している Zen のアクセス方法によって決まります。アクセス方法については、Zen のドキュメントを参照してください。

位置付け UPDATE ステートメントをセッション レベルで使用する場合のみ、ステートメントに**テーブル名**を指定できます。**テーブル名**をストアド プロシージャやトリガーを使って指定することはできません。

例

次の一連のステートメントは、位置付け UPDATE ステートメントの設定を示します。位置付け UPDATE に必要なステートメントは、DECLARE CURSOR、OPEN CURSOR、および FETCH FROM **カーソル名**です。

この例の位置付け UPDATE は、HIS 305 という講座名を HIS 306 に変更します。

```
CREATE PROCEDURE UpdateClass();  
BEGIN  
    DECLARE :CourseName CHAR(7);  
    DECLARE :OldName CHAR(7);  
    DECLARE c1 CURSOR FOR SELECT name FROM course WHERE name = :CourseName FOR UPDATE;  
    SET :CourseName = 'HIS 305';  
    OPEN c1;  
    FETCH NEXT FROM c1 INTO :OldName;
```

```
UPDATE SET name = 'HIS 306' WHERE CURRENT OF c1;  
END;
```

関連項目

[CREATE PROCEDURE](#)

[CREATE TRIGGER](#)

USER

備考

USER キーワードは、SELECT 制限によって返される各行の現在のユーザー名 (Master など) を返します。

例

次の例では、course テーブルからユーザー名が返されます。

```
SELECT USER FROM course
-- Master のインスタンスを 145 返します (テーブルには 145 行含まれます)
SELECT DISTINCT USER FROM course
-- Master のインスタンスを 1 つ返します
```

関連項目

[SELECT](#)

[SET SECURITY](#)

WHILE

WHILE ステートメントを使用してフローを制御することができます。これによって、WHILE 条件が真である限りステータスを繰り返し実行することができます。

構文

```
[ ラベル名 : ] WHILE プロシージャ検索条件 DO [ プロシージャ ステートメント [ ; プロシージャ ステートメント ] ... END WHILE [ ラベル名 ]
```

備考

構文に示されているように、WHILE ステートメントは省略可能なラベル名を持つことができ、指定した場合には、ラベル付き WHILE ステートメントと呼ばれます。

例

次の例では、変数 `vInteger` の値が 10 になってループが終了するまで、変数の値が 1 ずつ増加します。

```
WHILE (:vInteger < 10) DO  
SET :vInteger = :vInteger + 1;  
END WHILE
```

関連項目

[CREATE PROCEDURE](#)

[CREATE TRIGGER](#)

文法要素の定義

以下は、文法構文で使用される要素定義のアルファベット順一覧です。

変更オプション ::= 変更オプション- リスト 1 | 変更オプション- リスト 2

変更オプション- リスト 1 ::= 変更オプション | (変更オプション [, 変更オプション]...)

変更オプション ::= **ADD** [**COLUMN**] 列定義

| **ADD** テーブル制約定義
| **ALTER** [**COLUMN**] 列定義
| **DROP** [**COLUMN**] 列名
| **DROP CONSTRAINT** 制約名
| **DROP PRIMARY KEY**
| **MODIFY** [**COLUMN**] 列定義

変更オプション- リスト 2 ::= **PSQL_MOVE** [**COLUMN**] 列名 **TO** [[**PSQL_PHYSICAL**] **PSQL_POSITION**]
新しい列位置 | **RENAME COLUMN** 列名 **TO** 新しい列名

as またはセミコロン ::= **AS** | ;

before/after ::= **BEFORE** | **AFTER**

呼び出し引数 ::= 位置引数 [, 位置引数]...

列制約 ::= **NOT NULL**

| **NOT MODIFIABLE**
| **UNIQUE**
| **PRIMARY KEY**
| **REFERENCES** テーブル名 [(列名)] [参照アクション]

照合順序名 ::= ' 文字列 '

列制約定義 ::= [**CONSTRAINT** 制約名] 列制約

列定義 ::= 列名 データ型 [**DEFAULT** 式] [列制約定義 [列制約定義]... [**CASE** | **COLLATE** 照合順序名]

列名 ::= ユーザー定義名

コミット ステートメント ::= **COMMIT** ステートメント参照

比較演算子 ::= < | > | <= | >= | = | <> | !=

制約名 ::= ユーザー定義名

相関名 ::= ユーザー定義名

カーソル名 ::= ユーザー定義名

データ型 ::= データ型名 [(桁数 [, 小数位])]

データ型名 ::= Zen で使用できるデータ型を参照

データベース名 ::= ユーザー定義名

式 ::= 式 - 式

| 式 + 式
| 式 * 式
| 式 / 式

	式 & 式
	式 式
	式 ^ 式
	(式)
	- 式
	+ 式
	~ 式
	?
	リテラル
	スカラー関数
	{ fn スカラー関数 }
	USER

リテラル ::= ' 文字列 ' | N' 文字列 '
 | 数字
 | { d ' 日付リテラル ' }
 | { t ' 時刻リテラル ' }
 | { ts ' タイムスタンプ リテラル ' }

スカラー関数 ::= [スカラー関数](#)を参照

式またはサブクエリ ::= 式 | (クエリ スペック)

フェッチ方向 ::= **NEXT**

グループ名 ::= ユーザー定義名

インデックス定義 ::= (インデックス セグメント定義 [, インデックス セグメント定義]...)

インデックス名 ::= ユーザー定義名

インデックス番号 ::= ユーザー定義値 -- 0 から 118 までの整数

インデックス セグメント定義 ::= 列名 [**ASC** | **DESC**]

挿入 / 更新 / 削除 ::= **INSERT** | **UPDATE** | **DELETE**

挿入値 ::= **values** 句
 | クエリ スペック

結合定義 ::= テーブル参照 [**INNER**] **JOIN** テーブル参照 **ON** 検索条件
 | テーブル参照 **CROSS JOIN** テーブル参照
 | 外部結合定義

ラベル名 ::= ユーザー定義名

リテラル ::= ' 文字列 ' | N' 文字列 '
 | 数字
 | { d ' 日付リテラル ' }
 | { t ' 時刻リテラル ' }
 | { ts ' タイムスタンプ リテラル ' }

order-by 式 ::= 式 [**CASE** | **COLLATE** 照合順序名] [**ASC** | **DESC**]

外部結合定義 ::= テーブル参照 外部結合タイプ **JOIN** テーブル参照 **ON** 検索条件

外部結合タイプ ::= **LEFT** [**OUTER**] | **RIGHT** [**OUTER**] | **FULL** [**OUTER**]

パラメーター ::= パラメーター タイプ名 [データ型](#) [**DEFAULT** プロシージャ式 | = プロシージャ式]
 | **SQLSTATE**

```

パラメーター タイプ名 ::= パラメーター名
| パラメーター タイプ パラメーター名
| パラメーター名 パラメーター タイプ

```

```

パラメーター タイプ ::= IN | OUT | INOUT | IN_OUT

```

```

パラメーター名 ::= [ : ] ユーザー定義名

```

```

パスワード ::= ユーザー定義名 | ' 文字列 '

```

```

位置引数 ::= 式

```

```

桁数 ::= 整数

```

```

述部 ::= 式 [ NOT ] BETWEEN 式 AND 式
| 式 比較演算子 式またはサブクエリ
| 式 [ NOT ] IN ( クエリ スペック )
| 式 [ NOT ] IN ( 値 [ , 値 ]... )
| 式 [ NOT ] LIKE 値
| 式 IS [ NOT ] NULL
| 式 比較演算子 ANY ( クエリ スペック )
| 式 比較演算子 ALL ( クエリ スペック )
| EXISTS ( クエリ スペック )

```

```

プロシージャ式 ::= 通常の式と同様。ただし、IF 式とスカラー関数は使用できない

```

```

プロシージャ検索条件 ::= 通常の検索条件と同様。ただし、サブクエリを含む式は使用できない

```

```

プロシージャ ステートメント ::= [ ラベル名 : ] BEGIN [ ATOMIC ] [ プロシージャ ステートメント [ ; プロ
シージャ ステートメント ]... ] END [ ラベル名 ]
| CALL プロシージャ名 ( プロシージャ式 [ , プロシージャ式 ]... )
| CLOSE カーソル名
| DECLARE カーソル名 CURSOR FOR 選択ステートメント [ FOR UPDATE | FOR READ ONLY ]
| DECLARE 変数名 データ型 [ DEFAULT プロシージャ式 | = プロシージャ式 ]
| DELETE WHERE CURRENT OF カーソル名
| 削除ステートメント
| FETCH [ フェッチ方向 [ FROM ] ] カーソル名 [ INTO 変数名 [ , 変数名 ] ]
| IF プロシージャ検索条件 THEN プロシージャ ステートメント [ ; プロシージャ ステートメント ]... [
ELSE プロシージャ ステートメント [ ; プロシージャ ステートメント ]... ] END IF
| IF プロシージャ検索条件 プロシージャ ステートメント [ ELSE プロシージャ ステートメント ]
| 挿入ステートメント
| LEAVE ラベル名
| [ ラベル名 : ] LOOP プロシージャ ステートメント [ ; プロシージャ ステートメント ]... END LOOP [ ラ
ベル名 ]
| OPEN カーソル名
| PRINT プロシージャ式 [ , ' 文字列 ' ]
| RETURN [ プロシージャ式 ]
| トランザクション ステートメント
| into 付き選択ステートメント
| 選択ステートメント
| SET 変数名 = プロシージャ式
| SIGNAL [ ABORT ] sqlstate 値
| START TRANSACTION [ トランザクション名 ]
| 更新ステートメント
| UPDATE SET 列名 = プロシージャ式 [ , 列名 = プロシージャ式 ]... WHERE CURRENT OF カーソル名
| [ ラベル名 : ] WHILE プロシージャ検索条件 DO [ プロシージャ ステートメント [ ; プロシージャ ステート
メント ] ]... END WHILE [ ラベル名 ]
| [ ラベル名 : ] WHILE プロシージャ検索条件 プロシージャ ステートメント
| テーブル変更ステートメント

```

	インデックス作成ステートメント
	テーブル作成ステートメント
	ビュー作成ステートメント
	インデックス削除ステートメント
	テーブル削除ステートメント
	ビュー削除ステートメント
	権限付与ステートメント
	権限取消ステートメント
	設定ステートメント

プロシージャ名 ::= ユーザー定義名

public またはユーザー / グループ名 ::= **PUBLIC** | ユーザー / グループ名

クエリ スペック [**UNION** [**ALL**] クエリ スペック]...

[**limit** 句] [**ORDER BY** *order-by* 式 [, *order-by* 式]...] [**FOR UPDATE**]

クエリ スペック ::= (クエリ スペック)
 | **SELECT** [**ALL** | **DISTINCT**] [*top* 句] 選択リスト
 | **FROM** テーブル参照 [, テーブル参照]...
 | [**WHERE** 検索条件]
 | [**GROUP BY** 式 [, 式]...
 | [**HAVING** 検索条件]]

参照エイリアス ::= **OLD** [**AS**] 関連名 [**NEW** [**AS**] 関連名]
 | **NEW** [**AS**] 関連名 [**OLD** [**AS**] 関連名]

参照アクション ::= 参照更新アクション [参照削除アクション]
 | 参照削除アクション [参照更新アクション]

参照更新アクション ::= **ON UPDATE RESTRICT**

参照削除アクション ::= **ON DELETE CASCADE**
 | **ON DELETE RESTRICT**

リリース ステートメント ::= **RELEASE** ステートメントを参照

結果 ::= ユーザー定義名 データ型

ロールバック ステートメント ::= **ROLLBACK WORK** ステートメントを参照

セーブポイント名 ::= ユーザー定義名

スカラー関数 ::= スカラー関数の一覧を参照

小数位 ::= 整数

検索条件 ::= 検索条件 **AND** 検索条件
 | 検索条件 **OR** 検索条件
 | **NOT** 検索条件
 | (検索条件)
 | 述部

選択項目 ::= 式 [[**AS**] エイリアス名] | テーブル名 . *

選択リスト ::= * | 選択項目 [, 選択項目]...

セット関数 ::= **AVG** ([**DISTINCT** | **ALL**] 式)
 | **COUNT** (< * | [**DISTINCT** | **ALL**] 式 >)

```

| COUNT_BIG ( < * | [ DISTINCT | ALL ] 式 > )
| MAX ( [ DISTINCT | ALL ] 式 )
| MIN ( [ DISTINCT | ALL ] 式 )
| STDEV ( [ DISTINCT | ALL ] 式 )
| STDEVP ( [ DISTINCT | ALL ] 式 )
| SUM ( [ DISTINCT | ALL ] 式 )
| VAR ( [ DISTINCT | ALL ] 式 )
| VARP ( [ DISTINCT | ALL ] 式 )

```

sqlstate 値 ::= ' 文字列 '

テーブル制約定義 ::= [CONSTRAINT 制約名] テーブル制約

```

テーブル制約 ::= UNIQUE ( 列名 [ , 列名 ]... )
| PRIMARY KEY ( 列名 [ , 列名 ]... )
| FOREIGN KEY ( 列名 [ , 列名 ] )
  REFERENCES テーブル名
  [ ( 列名 [ , 列名 ]... ) ]
  [ 参照アクション ]

```

テーブル要素 ::= 列定義
| テーブル制約定義

```

テーブル式 ::=
  FROM テーブル参照 [ , テーブル参照 ]...
  [ WHERE 検索条件 ]
  [ GROUP BY 式 [ , 式 ]... ]
  [ HAVING 検索条件 ]

```

テーブル名 ::= ユーザー定義名

```

テーブル権限 ::= ALL
| SELECT [ ( 列名 [ , 列名 ]... ) ]
| UPDATE [ ( 列名 [ , 列名 ]... ) ]
| INSERT [ ( 列名 [ , 列名 ]... ) ]
| DELETE
| ALTER
| REFERENCES

```

```

テーブル参照 ::= { OJ 外部結合定義 }
| [ データベース名 . ] テーブル名 [ [ AS ] エイリアス名 ]
| 結合定義
| ( 結合定義 )

```

テーブル サブクエリ ::= クエリ スペック [[UNION [ALL]
クエリ スペック]...] [limit 句] [ORDER BY order-by 式 [, order-by 式]...]

limit 句 ::= [LIMIT [オフセット ,] 行数 | 行数 OFFSET オフセット | ALL [OFFSET オフセット]]

オフセット ::= 数値 | ?

行数 ::= 数値 | ?

```

トランザクション ステートメント ::= コミット ステートメント
| ロールバック ステートメント
| リリース ステートメント

```

トリガー名 ::= ユーザー定義名

ユーザーおよびパスワード ::= ユーザー名 [:] パスワード

ユーザー / グループ名 ::= ユーザー名 | グループ名

ユーザー名 ::= ユーザー定義名

値 ::= リテラル | **USER** | **NULL** | ?

値リスト ::= (値 [, 値]...)

values 句 ::= **DEFAULT VALUES** | **VALUES** (式 [, 式]...)

変数名 ::= ユーザー定義名

ビュー名 ::= ユーザー定義名

SQL ステートメント リスト

SqlStatementList は次のように定義されます。

SqlStatementList

ステートメント ';' | SqlStatementList ';'

ステートメント ::= ステートメント ラベル ':' ステートメント

	BEGIN ... END	ブロック
	CALL	ステートメント
	CLOSE CURSOR	ステートメント
	COMMIT	ステートメント
	DECLARE CURSOR	ステートメント
	DECLARE	変数ステートメント
	DELETE	ステートメント
	FETCH	ステートメント
	IF	ステートメント
	INSERT	ステートメント
	LEAVE	ステートメント
	LOOP	ステートメント
	OPEN	ステートメント
	PRINT	ステートメント
	RELEASE SAVEPOINT	ステートメント
	RETURN	ステートメント
	ROLLBACK	ステートメント
	SAVEPOINT	ステートメント
	SELECT	ステートメント
	SET	ステートメント
	SIGNAL	ステートメント
	START TRANSACTION	ステートメント
	UPDATE	ステートメント
	WHILE	ステートメント

述部

述部は次のように定義されます。

式 比較演算子 式

	式 [NOT] BETWEEN	式 AND 式
	式 [NOT] LIKE	文字列リテラル
	式 IS [NOT]	NULL

	NOT	述部
	述部	AND 述部
	述部	OR 述部
	'(' 述部 ')' 比較演算子 ::= '=' '>=' '>' '<=' '<' '<>' '!='	
	[NOT] IN 値リスト	

式

式は次のように定義されます。

数字
文字列リテラル
列名
変数名
NULL
CONVERT '(' 式 ',' データ型 ')'
'-' 式
式 '+' 式
式 '-' 式
式 '*' 式
式 '/' 式
式 '%' 式
'~' 式
式 ' ' 式
式 '^' 式
関数名 '(' [式リスト] ')'
'(' 式 ')'
'{' D 文字列リテラル '}'
'{' T 文字列リテラル '}'
'{' TS 文字列リテラル '}'
@:IDENTITY
@:ROWCOUNT
@@BIGIDENTITY
@@IDENTITY
@@ROWCOUNT
@@VERSION

式リストは次のように定義されます。

式リスト ::= 式 [, 式 ...]

グローバル変数

Zen は次のグローバル変数をサポートしています。

- **@@IDENTITY** および **@@BIGIDENTITY**
- **@@ROWCOUNT**
- **@@SESSIONID**
- **@@SPID**

- **@@VERSION**

グローバル変数は 2 つの @ マーク (@@) で始まります。すべてのグローバル変数は、接続ごとの変数です。データベース接続ごとにそれぞれ固有の @@IDENTITY、@@BIGIDENTITY、@@ROWCOUNT、および @@SPID 値を持ちます。@@VERSION の値は、この変数が使用されているステートメントを実行するエンジンのバージョンに関する情報です。

@@IDENTITY および @@BIGIDENTITY

これらの変数は、一番最後に挿入された列値を返します。値は符号付き整数値になります。初期値はヌルです。

変数は単一行のみ参照することができます。ターゲット テーブルに 2 つ以上の IDENTITY 列が含まれている場合、この変数の値は、テーブルの主キーとなる IDENTITY 列を参照します。このような列が存在しない場合には、この変数の値はテーブル内の最初の IDENTITY 列を参照します。

一番最後の挿入が IDENTITY 列を含まないテーブルに対して行われたものであった場合、@@IDENTITY の値はヌルに設定されます。

BIGIDENTITY 値の場合は、@@BIGIDENTITY を使用します。SMALLIDENTITY 値の場合は、@@IDENTITY を使用します。

例

```
SELECT @@IDENTITY
```

現在の接続でレコードが挿入されていない場合はヌルを返し、そうでない場合は、一番最後に挿入された行の IDENTITY 列の値を返します。

```
SELECT * FROM t1 WHERE @@IDENTITY = 12
```

一番最後に挿入された行の IDENTITY 列の値が 12 の場合はその行を返し、そうでない場合は行を返しません。

```
INSERT INTO t1(c2) VALUES (@@IDENTITY)
```

最後に挿入された行の IDENTITY 値を新しい行の C2 列に挿入します。

```
UPDATE t1 SET t1.c1 = (SELECT @@IDENTITY) WHERE t1.c1 = @@IDENTITY + 10
```

C1 列の値が最後に挿入された行の IDENTITY 列の値より 10 大きい場合は、C1 列をこの IDENTITY 値で更新します。


```
UPDATE t1 SET t1.c1 = (SELECT NULL FROM t2 WHERE t2.c1 = @@IDENTITY)
```

C1 列の値が最後に挿入された行の IDENTITY 列の値と等しい場合は、C1 列をヌル値で更新します。

以下の例では、ストアド プロシージャを作成し、それを呼び出します。プロシージャは、最後に更新された行の IDENTITY 列の値と入力値の合計を変数 V1 に設定します。プロシージャは次に、C1 列が V1 と等しい行をテーブルから削除し、その後で何行削除されたかを示すメッセージを出力します。

```
CREATE PROCEDURE TEST (IN :P1 INTEGER);
BEGIN
    DECLARE :V1 INTEGER;
    SET :V1 = :P1 + @@IDENTITY;
    DELETE FROM t1 WHERE t1.c1 = :V1;
    IF (@@ROWCOUNT = 0) THEN
        PRINT ' 行は削除されませんでした ';
    ELSE
        PRINT CONVERT(@@ROWCOUNT, SQL_CHAR) + ' 行が削除されました ';
    END IF;
END;
CALL TEST (@@IDENTITY)
```

@@ROWCOUNT

この変数は、現在の接続で一番最後に行った操作により影響を受けた行の数を返します。値は符号なし整数になります。初期値はゼロです。

@@ROWCOUNT 変数は、INSERT、UPDATE、または DELETE ステートメント後に使用した場合にのみ有効になります。

例

```
SELECT @@ROWCOUNT
```

現在の接続で直前に行った操作により影響を受けたレコードがない場合はゼロを返し、そうでない場合は、直前の操作により影響を受けた行数を返します。

```
CREATE TABLE t1 (c1 INTEGER, c2 INTEGER)
INSERT INTO t1 (c1, c2) VALUES (100, 200)
INSERT INTO t1(c1, c2) VALUES (300, @@ROWCOUNT)
SELECT @@ROWCOUNT
```

結果：

```
1 (@@ROWCOUNT 変数の値)
```

4 番目の行で、@@ROWCOUNT 変数の値は 1 になります。これは、直前の INSERT 操作が影響を与えたのは 1 行だからです。

@@IDENTITY の例も参照してください。

@@SESSIONID

この変数は、Zen サーバー エンジン、レポート エンジン、またはワークグループ エンジンへの接続を識別する 8 バイトの整数値を返します。この整数は時刻の値と増分カウンターの組み合わせです。この変数は Zen 接続をそれぞれ一意に識別するために使用できます。

@@SESSIONID が値を返すには、データベース エンジンに接続されていることが必要です。データベース エンジンへの接続が失われた場合、変数は ID を返すことができません。

例

```
SELECT @@SESSIONID
```

この例は、26552653137523 などの整数の ID を返します。

@@SPID

この変数、サーバー プロセス ID は、Zen 接続のシステム スレッドの ID を示す整数値を返します。

データベース エンジンへの接続が失われた場合、SPID 変数は ID を返すことができません。代わりに、ステートメントが SqlState 08S01 を示すエラーを返します。

例

```
SELECT @@SPID
```

この例は、402 などの整数の ID を返します。

@@VERSION

この変数を使用する SQL ステートメントは、セッションが接続されている Zen サーバー エンジン、レポート エンジン、またはワークグループ エンジンに基づいて値を返します。

- Zen サーバーの場合、この値はローカル エンジンのバージョン、およびローカルオペレーティング システムのビット数、名前、およびバージョンになります。
- Zen Client Reporting Engine の場合は、ローカル レポート エンジンのバージョン、およびローカル オペレーティング システムのビット数、名前、およびバージョンになります。

- Zen Client の場合は、クライアントの接続先であるリモート Zen サーバー上のエンジンのバージョン、およびそのサーバーが実行されているオペレーティングシステムのビット数、名前、およびバージョンになります。
- Zen Workgroup Engine の場合、この値はローカル エンジンのバージョン、およびローカル オペレーティングシステムのビット数、名前、およびバージョンになります。

例

```
SELECT @@version
```

この例は、次のようなテキスト情報を返します。

```
Actian Zen - 14.10.020.000 (x86_64) Server Engine - Copyright (C) Actian Corporation 2019 on (64-bit)
Windows NT 6.2 7d
```

ほかの特性

このトピックでは、SQL 文法のその他の特性について説明します。この章は、以下のセクションに分かれています。

- テンポラリ ファイル
- NULL 値を使った作業
- バイナリ データを使った作業
- インデックスの作成
- 小数点の記号のカンマ

テンポラリ ファイル

Zen は、指定されたクエリを処理するためにテンポラリ テーブルを生成する必要がある場合は、次のいずれかの方法で決定された場所にそのファイルを作成します。

- 文字列キー値の `PervasiveEngineOptions¥TempFileDirectory` を ODBC.INI に手動で追加した場合、Zen は TempFileDirectory に設定されたパスを使用します。Windows における 32 ビットおよび 64 ビット Zen インストールに対するレジストリの場所は、どちらも `HKEY_LOCAL_MACHINE¥SOFTWARE¥ODBC¥ODBC.INI` です。Linux および Raspbian の場合、ODBC.INI ファイルは `/usr/local/actianzen/etc` にあります。

- ZenCC または **bcfg** を使用して Zen エンジンのテンポラリ ファイル ディレクトリのプロパティを設定した場合、Zen はこの場所を使用します。詳細については、『*Advanced Operations Guide*』の**テンポラリ ファイル**を参照してください。
- ここに挙げた最初の 2 つのオプションのいずれも使用していない場合、Zen は Windows プラットフォームでは以下の順序でファイルの場所を調べます。

1. TMP 環境変数で指定されたパス
2. TEMP 環境変数で指定されたパス
3. USERPROFILE 環境変数で指定されたパス
4. Windows ディレクトリ

たとえば、TMP 環境変数が定義されていない場合、Zen は TEMP 環境変数、その次の環境変数という順に指定されているパスを使用します。

Linux ディストリビューションでは、Zen はサーバー プロセスの現在のディレクトリを使用します。TMP の使用を試みることはありません。

Zen はクエリが完了すると、そのクエリの処理に使用されたすべてのテンポラリ ファイルを削除します。クエリが SELECT ステートメントの場合には、テンポラリ ファイルは結果セットがアクティブである間、つまり、結果セットが呼び出し元のアプリケーションによって解放されるまでは存在します。

テンポラリ ファイルはどのような場合に作成されるか

Zen は、メモリ内、ディスク上、および Btrieve (MicroKernel エンジン) の 3 種類のテンポラリ ファイルを使用します。

メモリ内のテンポラリ ファイル

メモリ内のテンポラリ ファイルは以下の状況で使用されます。

- 前方のみのカーソル
- テンポラリ ファイル内のバイト数は 250,000 未満
- インデックスを使用しない ORDER BY、GROUP BY、DISTINCT を伴う SELECT ステートメントで、ORDER BY、GROUP BY、DISTINCT 内に BLOB または CLOB を含まない。また、UNION を伴う選択リストに BLOB または CLOB を含まない。

ディスク上のテンポラリ ファイル

ディスク上のテンポラリ ファイルは以下の状況で使用されます。

- 前方のみのカーソル
- テンポラリ ファイル内のバイト数は 250,000 より大きい
- インデックスを使用しない ORDER BY、GROUP BY、DISTINCT を伴う SELECT ステートメントで、ORDER BY、GROUP BY、DISTINCT 内に BLOB または CLOB を含まない。また、UNION を伴う選択リストに BLOB または CLOB を含まない。

Btrieve テンポラリ ファイル

Btrieve テンポラリ ファイルは以下の状況で使用されます。

- ORDER BY、GROUP BY、DISTINCT 内に BLOB または CLOB を使用する前方のみのカーソルがあるか、UNION を使用する選択リストで BLOB または CLOB を使用。
- UNION クエリを伴う動的または静的カーソル、または、インデックスを使用しない ORDER BY、GROUP BY、DISTINCT を伴う SELECT ステートメント

Zen は、静的カーソル SELECT クエリで各ベース テーブルに Btrieve テンポラリ ファイルを作成しません。その代わりに、各ベース テーブルを MicroKernel を使用して開き、ファイルの静的な表現としてファイル内にページを予約します。静的カーソルを介して行われた変更は、そのカーソルからは見えません。

NULL 値を使った作業

Zen は NULL を未知の値として解釈します。したがって、2 つの NULL 値を比較すると、等価とは見なされません。

`WHERE NULL=NULL` という評価式は False を返します。

バイナリ データを使った作業

次のシナリオについて考えてみましょう。t1 テーブルで c1 という名前が付けられた BINARY (4) 列にリテラル値 '1' を挿入します。次に、`SELECT * FROM t1 WHERE c1='1'` というステートメントを入力します。

データベース エンジン は、データ入力に使用したのと同じバイナリ形式を用いてデータを取得できます。つまり、上記の `SELECT` の例は、リテラルの一致がなくても正しく動作し、値 `0x01000000` を返します。

メモ： データベース エンジン は、挿入する奇数桁のバイナリ値の前に必ずゼロ ('0') を追加します。たとえば、値 '010' を挿入すると、値 '0x00100000' がデータ ファイルに保存されます。

現在、Zen はバイナリ定数を示す接尾辞 `0x` をサポートしていません。バイナリ定数は、一重引用符で囲んだ 16 進数の文字列になります。

この動作は、Microsoft SQL Server の動作と同じです。

インデックスの作成

インデックス可能な `VARCHAR` 型の列の最大サイズは、ヌル値が許可されない列では 254 バイト、ヌル値が許可される列では 253 バイトです。

`CHAR` 型の列の最大サイズは、ヌル値が許可されない列では 255 バイト、ヌル値が許可される列では 254 バイトです。16.0 形式ファイルの場合、この制限は 1024 と 1023 です。

インデックス可能な `NVARCHAR` 型の列の最大サイズは `NVARCHAR(126)` です。この制限は、ヌル値が許可される列とヌルでない列の両方に適用されます。`NVARCHAR` のサイズは UCS-2 文字単位で指定されます。

`NCHAR` 型の列の最大サイズは `NCHAR(127)` です。この制限は、ヌル値が許可される列とヌルでない列の両方に適用されます。`NCHAR` のサイズは UCS-2 文字単位で指定されます。

`Btrieve` の最大キーサイズは、`Btrieve` ファイル形式に応じて 255 または 1024 です。列でヌル値が許可され、インデックスが作成されている場合、セグメント キーは、ヌル インジケータの 1 バイトと、インデックス列の最大 254 バイトまたは 1023 バイトを使って作成されます。`VARCHAR` 型の列が `CHAR` 型の列と異なるのは、長さバイト (`Btrieve lstring`) またはゼロ終端バイト (`Btrieve zstring`) のいずれかが予約されており、そのため有効な記憶域が 1 バイト増える点です。`NVARCHAR` (`Btrieve wzstring`) 型の列が `NCHAR` 型の列と異なるのは、ゼロ終端文字が予約されており、そのため有効な記憶域が 2 バイト増える点です。

小数点の記号のカンマ

多くの地域では、浮動小数点数フィールド内で整数部と小数部を区切るのにカンマを使用します。たとえば、1 と 2 分の 1 を数字で表すのに 1.5 ではなく 1,5 を使用します。

Zen は、ピリオドとカンマの両方を小数点の記号としてサポートしています。Zen はオペレーティング システムの地域の設定に基づいて、ピリオドまたはカンマを使用する入力値を受け入れます。デフォルトでは、データベース エンジンではピリオドを使って値を表示します。

メモ： 小数点の記号がピリオドでない場合、SQL ステートメントでは数値を引用符で囲む必要があります。

出力と表示のみについては、セッション レベルのコマンド **SET DECIMALSEPARATORCOMMA** を使用すれば、出力（たとえば、SELECT の結果）でカンマを小数点の記号として使用するよう指定できます。このコマンドは、データの入力や保存には影響しません。

クライアント / サーバーの考慮

小数点の記号としてカンマをサポートするかは、オペレーティング システムの地域の設定によります。クライアント オペレーティング システムとサーバー オペレーティング システムのどちらにも地域の設定があります。期待される動作は両方の設定に応じて変わります。

- サーバーとクライアントどちらか一方の地域設定で小数点の記号にカンマが使用されている場合、Zen はピリオド区切りの値と引用符で囲まれたカンマ区切りの値の両方を受け入れます。
- サーバーとクライアントどちらの地域設定でも小数点の記号にカンマが使用されていない場合、Zen はカンマ区切りの値を受け入れません。

地域の設定の変更

小数点の記号の情報は、Windows オペレーション システムが動作しているマシンでのみ取得または変更できます。Linux の場合、小数点の記号はピリオドに設定されており、設定を修正することはできません。Linux サーバー エンジンをお持ちで、小数点の記号にカンマを使用したい場合は、必ずすべてのクライアント コンピューターが小数点にカンマを使用する地域に設定されていなければなりません。

Windows オペレーティング システムの地域の設定を変更するには、コントロール パネルからこの設定にアクセスします。変更したら、Zen サービスを停止して再起動し、データベース エンジンが変更された設定を使用できるようにします。

例

例 A – サーバーの地域が小数点の記号にカンマを使用する

クライアントの地域が小数点の記号にカンマを使用する場合：

```
CREATE TABLE t1 (c1 DECIMAL(10,3), c2 DOUBLE)
INSERT INTO t1 VALUES (10.123, 1.232)
INSERT INTO t1 VALUES ('10,123', '1.232')
SELECT * FROM t1 WHERE c1 = 10.123
SELECT * FROM t1 FROM c1 = '10,123'
```

上記の 2 つの SELECT ステートメントがクライアントから実行された場合、次の値が返されます。

```
10.123, 1.232
10.123, 1.232
SET DECIMALSEPARATORCOMMA=ON
SELECT * FROM t1 FROM c1 = '10,123'
```

上記の SELECT ステートメントが、小数点の記号を設定した後でクライアントから実行された場合は、次の値が返されます。

```
10,123, 1,232
```

クライアントの地域が小数点の記号にピリオドを使用しており、次のステートメントが新しい接続から発行される場合（SET DECIMALSEPARATORCOMMA のデフォルトの動作）：

```
CREATE TABLE t1 (c1 DECIMAL(10,3), c2 DOUBLE)
INSERT INTO t1 VALUES (10.123, 1.232)
INSERT INTO t1 VALUES ('10,123', '1.232')
SELECT * FROM t1 WHERE c1 = 10.123
SELECT * FROM t1 WHERE c1 = '10,123'
```

上記の 2 つの SELECT ステートメントがクライアントから実行された場合、次の値が返されます。

```
10.123, 1.232
10.123, 1.232
```

例 B – サーバーの地域が小数点の記号にピリオドを使用する

クライアントの地域が小数点（DECIMAL）の記号にカンマを使用する場合：

例 A のクライアントがカンマを使用する場合と同様です。

クライアントの地域が小数点（DECIMAL）の記号にピリオドを使用する場合：

```
CREATE TABLE t1 (c1 DECIMAL(10,3), c2 DOUBLE)
INSERT INTO t1 VALUES (10.123, 1.232)
INSERT INTO t1 VALUES ('10,123', '1,232') -- 割り当てエラー
SELECT * FROM t1 WHERE c1 = 10.123
SELECT * FROM t1 WHERE c1 = '10,123' -- 割り当てエラー
```

上記の最初の SELECT ステートメントがクライアントから実行された場合、次の値が返されます。

```
10.123、1.232
SET DECIMALSEPARATORCOMMA=ON
SELECT * FROM t1 FROM c1 = 10.123
```

上記の SELECT ステートメントが、表示用の小数点の記号を設定した後で実行された場合は、次の値が返されます。

```
10,123、1,232
```


スカラー関数

以下のトピックでスカラー関数について説明します。

- [ビット演算子](#)
- [算術演算子](#)
- [文字列関数](#)
- [数値関数](#)
- [時刻と日付の関数](#)
- [システム関数](#)
- [論理関数](#)
- [変換関数](#)

Zen はスカラー関数をサポートしており、このようなスカラー関数を SQL ステートメントに一次式として含むことができます。

ビット演算子

ビット演算子を使用すると、1 つまたは複数のオペランドのビットを操作することができます。ビット演算子には次の種類があります。

演算子	説明
&	ビット演算 AND
~	ビット演算 NOT
	ビット演算 OR
^	ビット演算 排他的 OR

式の記憶域の長さは、ビット演算を実行するときに考慮すべき重要な要因です。ビット演算でサポートされるデータ型は次のとおりです。

BIT	TINYINT	SMALLINT
INTEGER	BIGINT	UTINYINT
USMALLINT	UINTeger	UBIGINT

次の表では、ビット演算子の説明、構文、返される値、および例を示します。各ビット演算子は、オペランドとして1つの数値のみを取ります。

ビット演算子	説明と構文	返される値	例
AND	ビット演算子 AND は、2つのオペランド間でビットごとに論理積演算を実行します。AND は、2つのビットを比較し、両方のビットの値が1の場合にのみ、値1を結果に代入します。そうでない場合、結果のビットは0にセットされます。 式 & 式 式は INTEGER データ型を含んでいる任意の有効な式です。ビット演算のためにバイナリ数値へ変換されます。	異なる INTEGER データ型のオペランドを伴うビット演算 AND では、2つのオペランドのうち、サイズの小さなデータ型の引数が大きなデータ型へ変換されるか、または大きい方の次に大きなデータ型へ変換されます。 ビット演算 AND にかかわるオペランドが符号付きの場合、結果の値も符号付きになります。	& 演算子を IF 関数と組み合わせて使用し、テーブルがシステムテーブルかユーザー定義のテーブルかを調べることができます。 <pre>select Xf\$Name, IF(Xf\$Flags & 16 = 16, 'System table','User table') from X\$File</pre>
NOT	ビット演算子 NOT は、あらゆる変数のビット値を反転し、その値を対応する結果のビットにセットします。 ~ 式 式は INTEGER データ型を含んでいる任意の有効な式です。ビット演算のためにバイナリ数値へ変換されます。チルダ (~) は、ユーザー定義名の一部として使用できません。	ビット演算子 NOT は、INTEGER データ型の単一オペランドの反転を返します。すべての1は0に変換され、すべての0は1に変換されます。	次のクエリは、数値リテラルに対して補数演算を実行します。 <pre>SELECT ~12</pre> 結果は -13 です。補数演算は符合ビットも補完するため、結果は負数になります。

ビット演算子	説明と構文	返される値	例
OR	ビット演算子 OR は、2 つのオペランド間でビットごとの論理和演算を実行します。OR は、2 つのビットを比較し、一方または両方のビットの値が 1 の場合は、値 1 を結果に代入します。入力式のどちらのビットの値も 1 でない場合、結果のビットは 0 にセットされます。OR 演算子は、オペランドとして数値のみを取ります。 式 式 式は INTEGER データ型を含んでいる任意の有効な式です。ビット演算のためにバイナリ数値へ変換されます。	異なる INTEGER データ型のオペランドを伴うビット演算 OR では、2 つのオペランドのうち、サイズの小さなデータ型の引数が大きなデータ型へ変換されるか、または大きい方の次に大きなデータ型へ変換されます。 ビット演算 OR にかかわるオペランドが符号付きの場合、結果の値も符号付きになります。	次の例では、外部キーと主キーの制約の一覧を取得することができます。 <pre>select B.Xf\$Name "Table name", C.Xe\$Name "Column name", IF (Xi\$Flags & 8192 = 0, 'Primary key', 'Foreign key') "Key type" from X\$Index A, X\$File B, X\$field C where (A.Xi\$Flags & (16384 8192)) > 0 AND A.Xi\$File = B.Xf\$Id AND A.Xi\$field = C.Xe\$Id</pre>
OR (排他的)	ビット演算子の排他的 OR は、2 つのオペランド間でビットごとの排他的論理和演算を実行します。排他的 OR は、2 つのビットを比較し、両方のビットの値が 0 または 1 の場合は、値 0 を結果に代入します。そうでない場合は、対応する結果のビットを 1 にセットします。 式 ^ 式 式は INTEGER データ型を含んでいる任意の有効な式です。ビット演算のためにバイナリ数値へ変換されます。キャレット記号 (^) は、ユーザー定義名の一部として使用できません。	異なる INTEGER データ型のオペランドを伴うビット演算排他的 OR では、2 つのオペランドのうち、サイズの小さなデータ型の引数が大きなデータ型へ変換されるか、または大きい方の次に大きなデータ型へ変換されます。 ビット演算排他的 OR にかかわるオペランドが符号付きの場合、結果の値も符号付きになります。	次の SQL クエリは、2 つの数値リテラルで排他的 OR を実行します。 <pre>SELECT 12 ^ 8</pre> 結果は 4 です。

真理値表

次の表は、ビット演算の真理値表です。

A	B	A & B	A B	A ^ B	~ A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

算術演算子

日付演算

Zen では、加算または減算する日数が整数で、日付がベンダー文字列に埋め込まれている場合、その整数を日付に加算したり日付から減算したりできます。これは、日付について変換を実行するのと同じです。

また、Zen では、ある日付を別の日付から減算して日数を計算することもできます。

例

```
SELECT * FROM person P, Class C WHERE p.Date_Of_Birth < ' 1973-09-05' AND c.Start_date >{d '1995-05-08'} + 30
```

文字列関数

文字列関数は、CHAR、NCHAR、VARCHAR、NVARCHAR、LONGVARCHAR、または NLONGVARCHAR データ型のようなテキスト情報から成る列の処理および操作に使用されます。

文字列関数ではマルチバイトの文字列がサポートされます。(ただし、**CASE (文字列)** はマルチバイト文字列を**サポートしない**ことに注意してください。**CASE (文字列)** キーワードは、文字列データがシングルバイトの ASCII であることを前提としています。**CASE (文字列)** を参照してください。)

string と示される引数には、列の名前、文字列リテラル、または別のスカラー関数の結果を指定できます。次の表は Zen の文字列関数の一覧を示します。

関数	説明
ASCII (<i>string</i>)	<i>string</i> の一番左の文字の数値を返します。値は、データベース コード ページ内の文字の位置です。UNICODE 関数も参照してください。
BIT_LENGTH (<i>string</i>)	<i>string</i> の長さをビット数で返します。
CHAR (<i>code</i>)	<i>code</i> 引数により選択されるデータベース コード ページの文字に対応する、単一文字の文字列を返します。引数は整数値でなければなりません。NCHAR 関数も参照してください。
CHAR_LENGTH (<i>string</i>)	<i>string</i> 内のバイト数を返します。CHAR および NCHAR <i>string</i> では、すべての埋め込みが意味を持ちます。
CHAR_LENGTH2 (<i>string</i>)	<i>string</i> 内の文字数を返します。CHAR および NCHAR <i>string</i> では、すべての埋め込みが意味を持ちます。データにダブルバイト文字が含まれている場合は、文字列のサイズより小さい値が返されることがあります。
CHARACTER_LENGTH (<i>string</i>)	CHAR_LENGTH と同じです。
CONCAT (<i>string1</i> , <i>string2</i>)	<i>string2</i> を <i>string1</i> に連結した結果の文字列を返します。
ISNUMERIC (<i>string</i>)	文字列値が数値として評価できる場合は 1 (TRUE) を返します。そうでない場合は 0 (FALSE) を返します。
LCASE または LOWER (<i>string</i>)	<i>string</i> の大文字がすべて小文字に変換された文字列を返します。
LEFT (<i>string</i> , <i>count</i>)	<i>string</i> の左端から <i>count</i> 分の文字を返します。 <i>count</i> の値は整数です。
LENGTH (<i>string</i>)	<i>string</i> 内の文字数を返します。VARCHAR、NVARCHAR、LONGVARCHAR、または NLONGVARCHAR の <i>string</i> の場合、後続の空白はカウントされます。CHAR、NCHAR、LONGVARCHAR、または NLONGVARCHAR の <i>string</i> の場合、後続のヌルはカウントされます。 <i>string</i> の終端文字はカウントされません。ANSI_PADDING = OFF の場合、後続のヌルは後続の空白と同様に扱われ、CHAR 列では長さにカウントされません。

関数	説明
LOCATE (<i>string1</i> , <i>string2</i> [, <i>start</i>])	<i>string2</i> の中で <i>string1</i> が最初に現れる位置を返します。 <i>string2</i> 内での検索は、開始位置 (<i>start</i>) が指定されていなければ、先頭の文字位置から始めます。検索は指定された開始位置から始まります。 <i>string2</i> の先頭文字位置は 1 です。 <i>string1</i> が見つからない場合は、値 0 が返されます。
LTRIM (<i>string</i>)	<i>string</i> から先頭の空白を除いた文字を返します。CHAR および NCHAR <i>string</i> では、すべての埋め込みが意味を持ちます。
NCHAR (<i>code</i>)	<i>code</i> 引数で指定される Unicode コードポイント値に対応する、単一文字のワイド文字列を返します。引数は整数値でなければなりません。CHAR 関数も参照してください。
OCTET_LENGTH (<i>string</i>)	<i>string</i> の長さを 8 進数 (バイト数) で返します。CHAR および NCHAR <i>string</i> では、すべての埋め込みが意味を持ちます。
POSITION (<i>string1</i> , <i>string2</i>)	<i>string2</i> の中にある <i>string1</i> の位置を返します。 <i>string1</i> が <i>string2</i> 内に存在しない場合は、ゼロが返されます。
REPLACE (<i>string1</i> , <i>string2</i> , <i>string3</i>)	<i>string1</i> を検索して <i>string2</i> が現れる箇所を探し、それをすべて <i>string3</i> に置き換えます。その結果が返されます。文字列が現れなかった場合は、 <i>string1</i> が返されます。
REPLICATE (<i>string</i> , <i>count</i>)	<i>string</i> を <i>count</i> の回数だけ繰り返して構成した <i>string</i> を返します。 <i>count</i> の値は整数です。
REVERSE (<i>string</i>)	文字の並びが反転された <i>string</i> を返します。文字列型の先頭のスペースは末尾のスペースとは異なり、意味のあるものと見なされることに注意してください。例を参照してください。
RIGHT (<i>string</i> , <i>count</i>)	<i>string</i> の右端から <i>count</i> 分の文字を返します。 <i>count</i> の値は整数です。
RTRIM (<i>string</i>)	<i>string</i> から後続の空白を除いた文字列を返します。ANSI_PADDING = OFF の場合、後続のヌルは後続の空白と同様に扱われ、CHAR 列の値から除かれます。

関数	説明
SOUNDEX (<i>string</i>)	アルファ文字列を 4 文字のコードに変換して類似した音声の語や名前を検索します。2 つの文字列の類似性を評価する 4 文字 (SOUNDEX) のコードを返します。通常は名前です。 メモ : 合衆国政府が使用する Soundex の公式な実装に設定される現在の規則に従います。
SPACE (<i>count</i>)	<i>count</i> で示された数の空白から成る <i>string</i> を返します。
STUFF (<i>string1</i> , <i>start</i> , <i>length</i> , <i>string2</i>)	<i>string1</i> の <i>start</i> 位置から <i>length</i> で示された数だけの文字を <i>string2</i> で置き換えた結果得られる文字列を返します。 <i>start</i> と <i>length</i> は整数です。
SUBSTRING (<i>string1</i> , <i>start</i> , <i>length</i>)	<i>string1</i> の中の <i>start</i> で指定された文字位置から、 <i>length</i> で指定された文字数を取り出し、その文字列を返します。 <i>start</i> 値には、任意の数値を指定できます。 <i>string1</i> の先頭位置は 1 です。0 または負数の <i>start</i> 値は、先頭位置の左と見なされます。 <i>length</i> は負の値にできません。
UCASE または UPPER (<i>string</i>)	<i>string</i> の小文字すべて大文字に変換された文字列を返します。
UNICODE (<i>string</i>)	<i>string</i> の一番左の文字の Unicode コードポイント値を返します。ASCII 関数も参照してください。

メモ : CHAR_LENGTH を除き、文字列関数は最大 65,500 バイトの長さの文字列に対してのみ動作します CHAR_LENGTH は、使用中のデータ型で許可されている文字列の長さ全体に対して動作します。

スカラー関数の RTRIM または LEFT を使用する WHERE 句を含むクエリは、最適化することができます。たとえば、次のようなクエリがあるとします。

```
SELECT * FROM T1, T2 WHERE T1.C1 = LEFT(T2.C1, 2)
```

この場合、T1.C1 と T2.C2 がインデックス列ならば、述部の両側が最適化されます。**述部**は、WHERE キーワードの後に続く完全な検索条件です。結合に含まれるテーブルのサイズに従って、オプティマイザーは最初に処理する適切なテーブルを選択します。

一方の述部で、LTRIM および RIGHT が複雑な式中に含まれている場合、これらを最適化することはできません。

例

次の例では、整数の列と文字の列を持つ新しいテーブルを作成します。文字の列だけに値を持つ行を4行挿入し、その後で、それらの行の整数列を各行に含まれる文字のASCII文字コードで更新します。

```
CREATE TABLE numchars(num INTEGER,chr CHAR(1) CASE);
INSERT INTO numchars (chr) VALUES('a');
INSERT INTO numchars (chr) VALUES('b');
INSERT INTO numchars (chr) VALUES('A');
INSERT INTO numchars (chr) VALUES('B');
UPDATE numchars SET num=ASCII(chr);
SELECT * FROM numchars;
```

SELECT の結果：

num	chr
97	a
98	b
65	A
66	B

```
SELECT num FROM numchars WHERE num=ASCII('a')
```

SELECT の結果：

num
97

次の例では、Person テーブルの名前と姓を連結しており、"RooseveltBora" という結果が出ます。

```
SELECT CONCAT(First_name, Last_name) FROM Person WHERE First_name = 'Roosevelt'
```

次の例では、名前を小文字に変換してから大文字に変換しているため、"roosevelt"、"ROOSEVELT" という結果が出ます。

```
SELECT LCASE(First_name), UCASE(First_name) FROM Person WHERE First_name = 'Roosevelt'
```

次の例では、左端から3文字に切り詰められた名前、長さ9、LOCATEの結果0、という結果が出ます。このクエリの結果は "Roo"、9、0 です。

```
SELECT LEFT(First_name, 3),LENGTH(First_name), LOCATE(First_name, 'a') FROM Person WHERE First_name = 'Roosevelt'
```

次は、文字列で LTRIM および RTRIM 関数を使用した例で、結果は "Roosevelt"、
"Roosevelt"、"elt" になります。

```
SELECT LTRIM(First_name), RTRIM(First_name), RIGHT(First_name, 3) FROM Person WHERE First_name = 'Roosevelt'
```

次は、SUBSTRING 関数を使用した例です。

この SUBSTRING は、指定された列の 2 番目の文字から最高 3 文字を返します。

```
SELECT SUBSTRING(First_name,2, 3) FROM Person WHERE First_name = 'Roosevelt'
```

結果セット：

```
'oos'
```

次の SUBSTRING は、開始位置が文字列の最後を越えているため、空文字列を返します。

```
SELECT substring('ABCDE',10,1);
```

次の SUBSTRING は、指定された値を返します。

```
SELECT substring('ABCDE',0,2);
```

 - 'A' を返します

```
SELECT substring('ABCDE',-5,10);
```

 - 'ABCD' を返します

```
SELECT substring('ABCDE',-1,4);
```

 - 'AB' を返します

Smith と Smythe の文字列に SOUNDEX を使用した例を次に示します。

```
SELECT SOUNDEX ('Smith'), SOUNDEX ('Smythe')
```

結果セット：

```
S530  
S530
```

次の例は、Person テーブルで SOUNDEX を使用して "Kennedy" のように聞こえるすべての姓を検索する方法を示します。

```
SELECT Last_Name FROM Person WHERE SOUNDEX(last_name) = SOUNDEX ('Kennedy')
```

SELECT の結果：

```
Last_Name
-----
Kandy
Kenady
Kennedy
Kennedy
```

以下は REVERSE 関数の使用例です。

```
SELECT REVERSE(dept_name) from COURSE where dept_name = 'Music'
```

結果セット：

```
      cisuM
      cisuM
      cisuM
      cisuM
      cisuM
```

5 行が影響を受けました。

先頭のスペースには意味があるため、次のクエリは 0 行を返します。

```
SELECT * from COURSE WHERE REVERSE(dept_name) = 'cisum'
```

これは、dept_name が 20 文字幅の CHAR フィールドとして定義されているからです。
次に示すどちらのクエリ ステートメントでも期待される結果を返します。

```
SELECT * from COURSE WHERE REVERSE(dept_name) = '          cisuM'
```

```
SELECT * from COURSE WHERE LTRIM(REVERSE(dept_name)) = 'cisum'
```

結果セット：

```
MUS 101  Hymnology      3  Music
MUS 102  Church         3  Music
MUS 203  Piano          3  Music
MUS 304  Music Theory    3  Music
MUS 405  Recital         3  Music
```

5 行が影響を受けました。

数値関数

数値関数は、decimal 値や integer 値のような数値情報だけから成る列の処理および操作に使用されます。次の表は Zen の数値関数の一覧を示します。

関数	説明
ABS (<i>numeric_exp</i>)	<i>numeric_exp</i> の絶対値（正数）を返します。
ACOS (<i>float_exp</i>)	ラジアンで示された角度として、 <i>float_exp</i> のアークコサインを返します。
ASIN (<i>float_exp</i>)	ラジアンで示された角度として、 <i>float_exp</i> のアークサインを返します。
ATAN (<i>float_exp</i>)	ラジアンで示された角度として、 <i>float_exp</i> のアークタンジェントを返します。
ATAN2 (<i>float_exp1</i> , <i>float_exp2</i>)	ラジアンで示された角度として、x (<i>float_exp1</i>) 座標と y (<i>float_exp2</i>) 座標のアークタンジェントを返します。
CEILING (<i>numeric_exp</i>)	<i>numeric_exp</i> で示された値以上の最小の整数を返します。
COS (<i>float_exp</i>)	<i>float_exp</i> がラジアンで示された角度である場合、 <i>float_exp</i> のコサインを返します。
COT (<i>float_exp</i>)	<i>float_exp</i> がラジアンで示された角度である場合、 <i>float_exp</i> のコタンジェントを返します。
DEGREES (<i>numeric_exp</i>)	<i>numeric_exp</i> のラジアンから変換された度数を返します。
EXP (<i>float_exp</i>)	<i>float_exp</i> の指数値を返します。
FLOOR (<i>numeric_exp</i>)	<i>numeric_exp</i> で示された値以下の最大の整数を返します。
LOG (<i>float_exp</i>)	<i>float_exp</i> の自然対数を返します。
LOG10 (<i>float_exp</i>)	<i>float_exp</i> の 10 を底とする対数を返します。
MOD (<i>integer_exp1</i> , <i>integer_exp2</i>)	<i>integer_exp1</i> を <i>integer_exp2</i> で割った余り（絶対値）を返します。
PI ()	π の定数値を浮動小数点値で返します。
POWER (<i>numeric_exp</i> , <i>integer_exp</i>)	<i>numeric_exp</i> に <i>integer_exp</i> で示されたべき乗を行った値を返します。
RADIANS (<i>numeric_exp</i>)	<i>numeric_exp</i> の度数から変換されたラジアン値を返します。

関数	説明
RAND (<i>integer_exp</i>)	<i>integer_exp</i> をオプションのシード値として使用する、ランダムな浮動小数点値を返します。
ROUND (<i>numeric_exp</i> , <i>integer_exp</i>)	小数点の右側の <i>integer_exp</i> で示される位置で丸められた <i>numeric_exp</i> を返します。 <i>integer_exp</i> が負数の場合、 <i>numeric_exp</i> は、小数点の左側の <i>integer_exp</i> (<i>integer_exp</i> の絶対値) で示される位置で丸められます。
SIGN (<i>numeric_exp</i>)	<i>numeric_exp</i> の符号を示す値を返します。 <i>numeric_exp</i> が 0 より小さい場合は -1 が返されます。 <i>numeric_exp</i> が 0 の場合は 0 が返されます。 <i>numeric_exp</i> が 0 より大きい場合は 1 が返されます。
SIN (<i>float_exp</i>)	<i>float_exp</i> がラジアンで示された角度である場合、 <i>float_exp</i> のサインを返します。
SQRT (<i>float_exp</i>)	<i>float_exp</i> の平方根を返します。
TAN (<i>float_exp</i>)	<i>float_exp</i> がラジアンで示された角度である場合、 <i>float_exp</i> のタンジェントを返します。
TRUNCATE (<i>numeric_exp</i> , <i>integer_exp</i>)	小数点の右側の <i>integer_exp</i> で示される位置で切り捨てられた <i>numeric_exp</i> を返します。 <i>integer_exp</i> が負数の場合、 <i>numeric_exp</i> は、小数点の左側の <i>integer_exp</i> (絶対値) で示される位置で切り捨てられます。

例

次の例では、Room テーブルの部屋番号の列と収容人数の列の MOD を一覧表示します。

```
SELECT Number, Capacity, MOD(Number, Capacity) FROM Room WHERE Building_Name = 'Faske Building' and Type = 'Classroom'
```

次の例では、Faculty テーブルから 100 で割り切れる給与すべてを選択します。

```
SELECT Salary FROM Faculty WHERE MOD(Salary, 100) = 0
```

時刻と日付の関数

時刻および日付の関数により、時刻や日付データ型のデータを生成、処理、および操作することができます。このトピックでは、これらの関数の使用について説明します。

CURTIME()、CURDATE()、NOW() を INSERT ステートメントで使用して、現在の現地日付、時刻、およびタイムスタンプの値を挿入できます。たとえば、次のように指定します。

```
CREATE TABLE table1 (col1 DATE)
INSERT INTO table1 VALUES (CURDATE())
```

すべての時刻と日付関数は、次に示すように INSERT ステートメント内の SELECT サブクエリをサポートしています。

```
INSERT INTO t1 (c1, c2) SELECT CURRENT_DATE(), CURRENT_TIME()
```

CURDATE()、CURTIME()、NOW() などのいくつかの関数では、次に示すような直接の挿入もサポートされます。

```
INSERT INTO t1 (c1) VALUES (CURDATE())
```

ほかの例については、[時刻と日付関数の例](#)を参照してください。

次の表は Zen の時刻と日付の関数の一覧を示します。

メモ：ここに記載されている日付書式および時刻書式は、ZenCC のテキストビューやグリッドビューで値の表示に使用される書式とは一致しない可能性があります。表示される書式は、アプリケーションによって修正されます。

関数	説明
CURDATE()	現在の現地日付を 'yyyy-mm-dd' 形式で返します。デフォルトで、ローカル時計の日付を使用します。 SET TIME ZONE を呼び出した場合、CURDATE() の値は、システム時計とオペレーティングシステムの地域の設定を基に UTC 日付と時刻を計算し、SET TIME ZONE のオフセット値を加算することによって決定されます。
CURRENT_DATE()	現在の UTC 日付を 'dd/mm/yyyy' 形式で返します。
CURTIME()	現在の現地時刻を 'hh:mm:ss' 形式で返します。デフォルトで、ローカル時計の時刻を使用します。 SET TIME ZONE を呼び出した場合、CURTIME() の値は、システム時計とオペレーティングシステムの地域の設定を基に UTC 日付と時刻を計算し、SET TIME ZONE のオフセット値を加算することによって決定されます。
CURRENT_TIME()	現在の UTC 時刻を 'hh:mm:ss' 形式で返します。
CURRENT_TIMESTAMP()	現在の UTC 日付と時刻を 'yyyy-mm-dd hh:mm:ss.mmm' 形式で返します。

関数	説明
DATEADD(<i>datepart</i> , <i>interval</i> , <i>date_exp</i>)	日付に <i>interval</i> を加算した新しい DATETIME 値を返します。たとえば、<i>datepart</i> が day、<i>interval</i> が 11、<i>date_exp</i> が 2020 年 1 月 26 日の場合は、2020 年 2 月 6 日を返します。<i>datepart</i> には <i>interval</i> を加算する日付の部分を指定します。値は以下のいずれかである必要があります。 • YEAR • QUARTER • MONTH • DAY • DAYOFYEAR • WEEK • HOUR • MINUTE • SECOND • MILLISECOND <i>interval</i> には、<i>datepart</i> を増加させるために用いる正または負の整数値を指定します。<i>interval</i> に小数部が含まれる場合、小数部は無視されます。 <i>date_exp</i> は DATETIME 値、暗黙的に DATETIME 値に変換される値、または DATE 形式の文字列を返す式です。 DATETIME を参照してください。

関数	説明
DATEDIFF(<i>datepart</i> , <i>start</i> , <i>end</i>)	2つの日付の差異を表す整数を返します。この整数は、2つの日付間の日付と時刻の境界の数です。 たとえば、テーブル <code>mytest</code> に <code>col1</code> および <code>col2</code> の2つの列があり、両方とも DATETIME だとします。<code>col1</code> の値は 2000-01-01 11:11:11.234 で、<code>col2</code> は 2004-09-11 10:10:10.211 です。次の SELECT ステートメントは 56 を返します。それが <code>col1</code> と <code>col2</code> の月の差異だからです。 <pre>SELECT DATEDIFF(month, col1, col2) as Month_Difference FROM mytest</pre> <i>datepart</i> には差異を計算する日付の部分を指定します。値は以下のいずれかである必要があります。 • YEAR • QUARTER • MONTH • DAY • DAYOFYEAR • WEEK • HOUR • MINUTE • SECOND • MILLISECOND <i>start</i> には差異を計算する始まりの日付を指定します。<i>start</i> は DATETIME 値または DATE 形式の Unicode 文字列を返す式です。 <i>end</i> には差異を計算する終わりの日付を指定します。<i>end</i> は DATETIME 値または DATE 形式の Unicode 文字列を返す式です。 <i>end</i> から <i>start</i> が減算されます。戻り値が整数値の範囲外の場合はエラーが返されます。データ型の範囲を参照してください。

関数	説明
DATEFLOOR(<i>timestamp_exp</i> , <i>interval_unit</i>)	<i>timestamp_exp</i> を最も近い <i>interval_unit</i> の境界に切り捨てることによって決定されるタイムスタンプを返します。この切り捨ては、<i>interval_unit</i> より右にあるすべてのタイムスタンプフィールドを最小値に設定することで行われます。たとえば、<i>interval_unit</i> が day である場合は、<i>timestamp_exp</i> の year、month、および day はすべて保持され、hour、minute、および second はゼロに設定されます。<i>interval_unit</i> に対して有効な値は次のとおりです。 • YEAR • MONTH • DAY • HOUR • MINUTE • SECOND
DATEFROMPARTS(<i>year</i> , <i>month</i> , <i>day</i>)	指定された年、月、日の日付値を返します。パラメーターのいずれかがヌルである場合は、ヌルが返されます。

関数	説明
DATENAME (<i>datepart</i> , <i>date_exp</i>)	<i>date_exp</i> の <i>datepart</i> を表す英語の文字列 (VARCHAR) を返します。たとえば、<i>datepart</i> month は January、February というように月名を返します。<i>datepart</i> weekday は Monday、Tuesday というように曜日を返します。 <i>datepart</i> には返される日付の部分を指定します。値は以下のいずれかである必要があります。 • YEAR • QUARTER • MONTH • DAY • DAYOFYEAR • WEEK • WEEKDAY • HOUR • MINUTE • SECOND • MILLISECOND <i>date_exp</i> は DATETIME 値、暗黙的に DATETIME 値に変換される値、または DATE 形式の文字列を返す式です。 DATETIME を参照してください。

関数	説明
DATEPART(<i>datepart</i> , <i>date_exp</i>)	<i>date_exp</i> の <i>datepart</i> を表す整数を返します。たとえば、<i>datepart</i> month は月を表す整数を返します（1 月 = 1、2 月 = 2）。<i>datepart</i> weekday は曜日を表す整数を返します（日曜日 = 1、土曜日 = 7）。 <i>datepart</i> には返される日付の部分指定します。値は以下のいずれかである必要があります。 • YEAR • QUARTER • MONTH • DAY • DAYOFYEAR • WEEK • WEEKDAY • HOUR • MINUTE • SECOND • MILLISECOND • TZOFFSET TZOFFSET 値は、タイムゾーンのオフセットを分数（符号付き）で返します。TZOFFSET を指定した DATEPART 関数は、SYSDATETIMEOFFSET() および、タイムゾーンのオフセットを含んでいる文字列リテラルでのみ機能します。タイムゾーンオフセットの範囲は -14:00 から +14:00 です。時刻と日付関数の例を参照してください。 <i>date_exp</i> は DATETIME 値、暗黙的に DATETIME 値に変換される値、または DATE 形式の文字列を返す式です。DATETIME を参照してください。
DAY(<i>date_exp</i>)	指定された <i>date_exp</i> の日にちを返します。これは DATEPART(day, <i>date_exp</i>) とまったく同じです。DATEPART(<i>datepart</i>, <i>date_exp</i>) を参照してください。

関数	説明
DAYNAME(<i>date_exp</i>)	<i>date_exp</i> の曜日部分に対して、曜日の名前（たとえば、Sunday から Saturday）を含んでいる英語の文字列を返します。 <i>date_exp</i> には、日付リテラル、SQL_TIMESTAMP リテラル、または DATE、DATETIME、タイムスタンプ データが格納されている列を指定できます。
DAYOFMONTH(<i>date_exp</i>)	<i>date_exp</i> の月の何日目かを 1 ～ 31 の範囲の整数値で返します。<i>date_exp</i> には、日付リテラル、SQL_TIMESTAMP リテラル、または DATE、DATETIME、タイムスタンプ データが格納されている列を指定できます。
DAYOFYEAR(<i>date_exp</i>)	<i>date_exp</i> の年の何日目かを 1 ～ 366 の範囲の整数値で返します。
DATETIMEFROMPARTS(<i>year, month, day, hour, minute, seconds, milliseconds</i>)	提供されたパラメーターで構成される値を返します。パラメーターのいずれかがヌルである場合は、ヌルが返されます。
DATETIMEOFFSETFROMPARTS(<i>year, month, day, hour, minute, seconds, fractions, hour_offset, minute_offset, scale</i>)	指定されたパラメーターを使用して、日付と時刻の文字列値を返します。 <i>scale</i> を除くパラメーターのいずれかがヌルである場合は、ヌルが返されます。<i>scale</i> がヌルの場合は、エラーが返されます。 <i>scale</i> には、分数値の精度を指定します。指定できる範囲は 0 から 7 です。<i>fractions</i> は <i>scale</i> によって決まり、範囲は 0 から 9999999 です。たとえば、<i>scale</i> が 3 の場合、各 <i>fractions</i> はミリ秒を表します。<i>fractions</i> に指定される桁数は、<i>scale</i> の値以下でなければなりません。 <i>hour_offset</i> には、タイムゾーンの時部分を指定します。範囲は -14 から +14 です。<i>minute_offset</i> には、タイムゾーンの分部分を指定します。範囲は 0 から 59 です。<i>hour_offset</i> と <i>minute_offset</i> の符号は、<i>hour_offset</i> が 0 でない限り、同じでなければなりません。 DATETIMEOFFSETFROMPARTS のデフォルトの文字列リテラル書式は、YYYY-MM-DD hh:mm:ss[.nnnnnnnn] [{+ -}hh:mm] です。

関数	説明
EVERYN(<i>interval_unit</i> , <i>timestamp_exp</i> , <i>rounding_unit</i> , <i>bucket_size</i>)	<i>timestamp_exp</i> を最も近いタイム バケットの開始に切り捨てることによって決定されるタイムスタンプを返します。タイム バケットは、<i>bucket_size</i> * <i>interval_unit</i> をバケットとし、最も近い <i>rounding_unit</i> の開始からカウントします。たとえば、1 分ごとに 10 個のデータのグループを持ちたいとします。これには 6 秒バケットが必要となります。EVERYN(SECOND, '2022-06-15 08:15:22.891', MINUTE, 6) は、指定されたタイムスタンプを、最も近い分 (MINUTE) の開始からカウントして最も近い 6 秒バケットの開始に切り捨てます。最も近い分は '2022-06-15 08:15:00.000' であり、この分の 4 番目の 6 秒バケットが '2022-06-15 08:15:18.000' から始まるため、この値が返されることになります。このバケット内のタイムスタンプはすべて、これと同じ結果を EVERYN から受け取ります。 <i>interval_unit</i> および <i>rounding_unit</i> に対して有効な値は次のとおりです。 • YEAR • MONTH • DAY • HOUR • MINUTE • SECOND メモ : <i>interval_unit</i> より <i>rounding_unit</i> の方が間隔が大きい場合にのみ、EVERYN は有用な結果をもたらします。
EXTRACT(<i>extract_field</i> , <i>extract_source</i>)	<i>extract_source</i> の <i>extract_field</i> 部分を返します。引数 <i>extract_source</i> は、日付、時刻、または間隔を表す式です。<i>extract_field</i> には次の値を指定でき、またこれらの値が対象の式から返されます。 • YEAR • MONTH • DAY • HOUR • MINUTE • SECOND

関数	説明
HOUR(<i>time_exp</i>)	時刻を 0 ～ 23 の範囲の整数値で返します。 <i>time_exp</i> には、日付リテラル、SQL_TIMESTAMP リテラル、または DATE、DATETIME、タイムスタンプ データが格納されている列を指定できます。
MINUTE(<i>time_exp</i>)	分を 0 ～ 59 の範囲の整数値で返します。 <i>time_exp</i> には、日付リテラル、SQL_TIMESTAMP リテラル、または DATE、DATETIME、タイムスタンプ データが格納されている列を指定できます。
MONTH(<i>date_exp</i>)	月を 1 ～ 12 の範囲の整数値で返します。 <i>date_exp</i> には、日付リテラル、SQL_TIMESTAMP リテラル、または DATE、DATETIME、タイムスタンプ データが格納されている列を指定できます。
MONTHNAME(<i>date_exp</i>)	<i>date_exp</i> の月の部分に対して、月名（たとえば、January から December）を含んでいる英語の文字列を返します。 <i>date_exp</i> には、日付リテラル、SQL_TIMESTAMP リテラル、または DATE、DATETIME、タイムスタンプ データが格納されている列を指定できます。
NOW()	現在の現地日付と時刻を 'yyyy-mm-dd hh:mm:ss.mmm' 形式で返します。 デフォルトで、ローカル時計の時刻を使用します。SET TIME ZONE を呼び出した場合、NOW() の値は、システム時計とオペレーティング システムの地域の設定を基に UTC 日付と時刻を計算し、SET TIME ZONE のオフセット値を加算することによって決定されます。
QUARTER(<i>date_exp</i>)	<i>date_exp</i> の四半期を 1 ～ 4 の範囲の整数値で返します。1 は 1 月 1 日～3 月 31 日を表します。 <i>date_exp</i> には、日付リテラル、SQL_TIMESTAMP リテラル、または DATE、DATETIME、タイムスタンプ データが格納されている列を指定できます。
SECOND(<i>time_exp</i>)	秒を 0 ～ 59 の範囲の整数値で返します。 <i>time_exp</i> には、日付リテラル、SQL_TIMESTAMP リテラル、または DATE、DATETIME、タイムスタンプ データが格納されている列を指定できます。

関数	説明
<code>SYSDATETIME()</code>	現在の現地日付と時刻を 'yyyy-mm-dd hh:mm:ss.nnnnnnnnn' 形式で返します。 デフォルトで、ローカル時計の時刻を使用します。小数位は、Windows 10 ではセプタ秒、Linux ではナノ秒、その他のすべてのプラットフォームではマイクロ秒です。値が返されない末尾の数字部分は、9 桁までゼロで埋められます。 SET TIME ZONE を呼び出した場合、<code>SYSDATETIME()</code> の値は、システム時計とオペレーティングシステムの地域の設定を基に UTC 日付と時刻を計算し、<code>SET TIME ZONE</code> のオフセット値を加算することによって決定されます。
<code>SYSDATETIMEOFFSET()</code>	現在の日付と時刻に加えて、Zen データベース エンジンが起動しているコンピューターの現在のタイムゾーンと UTC の間の時間と分のオフセットを返します。サマータイム (DST) が考慮されます。 返されるデフォルトの書式は、YYYY-MM-DD hh:mm:ss[.nnnnnnnnn] [<+ ->hh:mm] です。プラス記号は、現在のタイムゾーンが UTC より進んでいることを示します。マイナス記号は、現在のタイムゾーンが UTC より遅れていることを示します。
<code>SYSUTCDATETIME()</code>	現在の現地日付と時刻を 'yyyy-mm-dd hh:mm:ss.nnnnnnnnn' 形式で返します。 デフォルトで、ローカル時計の時刻を使用します。小数位は、Windows 10 ではセプタ秒、Linux ではナノ秒、その他のすべてのプラットフォームではマイクロ秒です。値が返されない末尾の数字部分は、9 桁までゼロで埋められます。 SET TIME ZONE を呼び出した場合、<code>SYSUTCDATETIME()</code> の値は、システム時計とオペレーティングシステムの地域の設定を基に UTC 日付と時刻を計算することによって決定されます。

関数	説明
TIMEFROMPARTS(<i>hour, minute, seconds, fractions, scale</i>)	指定された時刻パラメーターで構成される時刻値を返します。 <i>scale</i> を除くパラメーターのいずれかがヌルである場合は、ヌルが返されます。<i>scale</i> がヌルの場合は、エラーが返されます。 <i>scale</i> には、分数値の精度を指定します。指定できる範囲は 0 から 7 です。<i>fractions</i> は <i>scale</i> によって決まり、範囲は 0 から 9999999 です。たとえば、<i>scale</i> が 3 の場合、各 <i>fractions</i> はミリ秒を表します。<i>fractions</i> に指定される桁数は、<i>scale</i> の値以下でなければなりません。 TIMEFROMPARTS のデフォルトの書式は、hh:mm:ss[.nnnnnnn] です。
TIMESTAMPADD(<i>interval, integer_exp, timestamp_exp</i>)	<i>interval</i> タイプの <i>integer_exp</i> で示された間隔を <i>timestamp_exp</i> に加算したタイムスタンプを返します。 <i>interval</i> に指定できるキーワードは次のとおりです。 • SQL_TSI_YEAR • SQL_TSI_QUARTER • SQL_TSI_MONTH • SQL_TSI_WEEK • SQL_TSI_DAY • SQL_TSI_HOUR • SQL_TSI_MINUTE • SQL_TSI_SECOND
TIMESTAMPDIFF(<i>interval, timestamp_exp1, timestamp_exp2</i>)	<i>timestamp_exp2</i> が <i>timestamp_exp1</i> より大きい場合に、その差である <i>interval</i> の整数値を返します。 <i>interval</i> に指定できる値は、TIMESTAMPADD と同じです。
WEEK(<i>date_exp</i>)	<i>date_exp</i> が 1 年の第何週目かを 1 ～ 53 の範囲の整数値で返します。<i>date_exp</i> には、日付リテラル、SQL_TIMESTAMP リテラル、または DATE、DATETIME、タイムスタンプ データが格納されている列を指定できます。
WEEKDAY(<i>date_exp</i>)	指定された <i>date_exp</i> の曜日を返します。1 = 日曜日、7 = 土曜日です。これは DATEPART(<i>weekday, date_exp</i>) とまったく同じです。DATEPART(<i>datepart, date_exp</i>) を参照してください。

関数	説明
YEAR(<i>date_exp</i>)	年数を整数値で返します。範囲はデータソースに依存します。 <i>date_exp</i> には、日付リテラル、SQL_TIMESTAMP リテラル、または DATE、DATETIME、タイムスタンプデータが格納されている列を指定できます。

時刻と日付関数の例

次は HOUR の使用例です。

```
SELECT c.Name, c.Credit_Hours FROM Course c WHERE c.Name = ANY (SELECT cl.Name FROM Class cl WHERE c.Name = cl.Name AND c.Credit_Hours >(HOUR (Finish_Time - Start_Time) + 1))
```

これは MINUTE の使用例です。

```
SELECT MINUTE(log) FROM billing
```

次は SECOND の使用例です。

```
SELECT SECOND(log) FROM billing;
SELECT log FROM billing WHERE SECOND(log) = 31
```

次は NOW() の使用例です。

```
SELECT NOW() - log FROM billing
```

次は、MONTH、DAY、YEAR、HOUR、MINUTE を使用するより複雑な例です。

```
SELECT Name, Section, MONTH(Start_Date), DAY(Start_Date), YEAR(Start_Date), HOUR(Start_Time), MINUTE(Start_Time) FROM Class
```

次は CURDATE() の使用例です。

```
SELECT ID, Name, Section FROM Class WHERE (Start_Date - CURDATE()) <= 2 AND (Start_Date - CURDATE()) >= 0
```

次の例では、Class テーブルにあるクラスの開始日を基に、月の何日目かとその曜日を求めます。

```
SELECT DAYOFMONTH(Start_date), DAYOFWEEK(Start_date) FROM Class;
```

```
SELECT * FROM person WHERE YEAR(Date_Of_Birth) < 1970
```

次は、TZoffset パラメーターを指定した DATEPART の使用例です。

```
SELECT DATEPART(TZoffset, SYSDATETIMEOFFSET())
```

ステートメントが -360 を返すとしたら、現在のタイムゾーンは UTC より 360 分遅れています。

SELECT SYSDATETIMEOFFSET() が 2011-01-24 14:33:08.4650000 -06:00 を返すとし
ます。これを前提として、次のクエリは -360 を返します。

```
SELECT DATEPART(TZoffset, '2011-01-24 14:33:08.4650000 -06:00')
```

タイムゾーンの部分が文字列リテラルから省略されている場合は、0 が返されます。

```
SELECT DATEPART(TZoffset, '2011-01-24 14:33:08.4650000')
```

次の例は、DATEFROMPARTS を使用して、提供された値を基に日付を返します。

```
SELECT NOW(), DATEFROMPARTS(DATEPART(Year, NOW()), DATEPART(Month, NOW()), DATEPART(Day, NOW()))
```

戻り値 : 2013-05-09 14:33:34.835 PM 5/9/2013

次の例は、TIMEFROMPARTS を使用して、提供された値を基に時刻を返します。

```
SELECT NOW(), TIMEFROMPARTS(DATEPART(hour, NOW()), DATEPART(minute, NOW()), DATEPART(second, NOW()),  
DATEPART(millisecond, NOW()), 3)
```

戻り値 : 2013-05-09 15:04:11.425 PM 15:04:11.425

次の例は、DATETIMEFROMPARTS を使用して、提供された値を基にタイムスタンプ
を返します。

```
SELECT DATETIMEFROMPARTS(1962, 08, 12, 17, 45, 0, 0)
```

戻り値 : 1962-08-12 17:45:00.000 PM

次の例は、DATETIMEOFFSETFROMPARTS を使用して、タイムゾーン指定を加えたタ
イムスタンプの表示文字列を返します。

```
SELECT DATETIMEOFFSETFROMPARTS (1962, 08, 12, 17, 45, 0, 0, 5, 0, 0) + ' GMT'
```

戻り値 : 1962-08-12 17:45:00 +05:00 GMT

次の例は、EVERYN を使用して、Billing テーブルのレコードを TIMESTAMP 型の Log フィールドを基に、6 秒バケットにグループ化します。

```
SELECT COUNT(*) NumOfPayments, AVG(Amount_Paid) AveragePayment, EVERYN(SECOND, Log, MINUTE, 6)
TimePeriodStart FROM Billing GROUP BY TimePeriodStart
```

次の EVERYN の例では、丸め単位がバケット サイズで均等に分割されない場合の、丸め単位の指定方法を示します。たとえば、記録された値をすべて 7 秒バケットに割り当てる必要があるが、バケットが時間 (HOUR) の境界を越えるようにしたくないとします。次の測定値は、時間の境界付近に記録された一連の値を示しています。

Time Stamp	EVERYN(SECOND, <time stamp>, HOUR, 7)
2022-06-15 08:59:50.891	2022-06-15 08:59:44.000
2022-06-15 08:59:57.891	2022-06-15 08:59:51.000
2022-06-15 08:59:59.891	2022-06-15 08:59:58.000
2022-06-15 09:00:02.891	2022-06-15 09:00:00.000
2022-06-15 09:00:05.891	2022-06-15 09:00:00.000
2022-06-15 09:00:08.891	2022-06-15 09:00:07.000

時間の最後の間隔は 2022-06-15 08:59:58.000 から始まりますが、次の間隔は次の時間、つまり 2022-06-15 09:00:00.000 から始まる必要があるため、最後の間隔は 2 秒しかないことに留意してください。

EVERYN は機能上、複数の他の関数で構成したり、値を繰り返したりする必要のある式をパラメーター化することができます。たとえば、次の式では 7 つの関数呼び出しを使用しています。

```
DATEADD(SECOND, ((FLOOR (DATEDIFF(SECOND, DATEFLOOR(ts_val, HOUR), ts_val) / 8)) * 8),
DATEFLOOR(ts_val, HOUR))
```

EVERYN を呼び出すことで、これを簡略化できます。

```
EVERYN(SECOND, ts_val , HOUR, 8)
```

システム関数

システム関数は、システム レベルの情報を提供します。

関数	説明
DATABASE()	現在のデータベース名を返します。
NEWID()	データ型 uniqueidentifier の一意な値を作成します。
USER()	現在のユーザーのログイン名を返します。

システム関数の例

次の例は、現在のユーザーおよびデータベースの名前を取得する方法を示します。

```
SELECT USER();  
SELECT DATABASE();
```

以下の例は、新規テーブル `table1` の最初の列として、データ型 `UNIQUEIDENTIFIER` の列を作成しています。NEWID 関数を使用してデフォルト値を設定することによって、テーブル内の新しい行の `col1` にそれぞれ一意な値が与えられます。

```
CREATE TABLE table1 (col1 UNIQUEIDENTIFIER DEFAULT NEWID() NOT NULL, col2 INTEGER);  
INSERT INTO table1 (col2) VALUES (1);  
INSERT INTO table1 (col2) VALUES (2);  
INSERT INTO table1 (col2) VALUES (3);
```

論理関数

論理関数は、一定の条件に基づいたデータの操作に使用されます。

関数	説明
COALESCE(<i>expression1</i> , <i>expression2</i> [, ...])	式リストを左から調べて、最初の非ヌル引数を返します。詳細については、 COALESCE も参照してください。
IF (<i>predicate</i> , <i>expression1</i> , <i>expression2</i>)	述部 (<i>predicate</i>) が真の場合は <i>expression1</i> を返します。そうでない場合は <i>expression2</i> を返します。
NULL()	列をヌル値として設定します。
IFNULL(<i>exp</i> , <i>value</i>)	<i>exp</i> がヌルの場合は <i>value</i> が返されます。 <i>exp</i> がヌルでない場合は <i>exp</i> が返されます。 <i>value</i> に指定するデータ型は、 <i>exp</i> のデータ型と互換性がなければなりません。
ISNULL(<i>exp</i> , <i>value</i>)	ヌルを <i>value</i> で示された値に置き換えます。 <i>exp</i> はヌルをチェックする式です。 <i>value</i> は <i>exp</i> がヌルの場合に返される値です。ヌルでない場合は <i>exp</i> が返されます。 <i>value</i> に指定するデータ型は、 <i>exp</i> のデータ型と互換性がなければなりません。
NULLIF (<i>exp1</i> , <i>exp2</i>)	2 つの式が等しくない場合は、 <i>exp1</i> を返します。式が等しい場合、NULLIF はヌル値を返します。

論理関数の例

COALESCE スカラー関数は 2 つ以上の引数を取り、その式リストを左から調べて最初の非ヌル引数を返します。

```
select COALESCE(10, 'abc' + 'def')
```

10 は SMALLINT として扱われ、ResultType (SMALLINT, VARCHAR) は SMALLINT になります。したがって、結果の型は SMALLINT になります。

最初のパラメーターは 10 で、この値は結果の型の SMALLINT へ変換できます。そのため、この例の戻り値は 10 になります。

システム スカラー関数の IF と NULL は、SQL の拡張機能です。

IF を使用すると、条件が真か偽かによって異なる値を入力できます。たとえば、論理値を含む列をバイナリ表記ではなく "True" または "False" で表示するには、次の SQL ステートメントを使用します。

```
SELECT IF(logicalcol=1, 'True', 'False')
```

システム スカラー関数の NULL を使用すると、列をヌル値として設定できます。構文は次のとおりです。

```
NULL()
```

たとえば、次の SQL ステートメントは、Capacity をヌル値とする行を Room テーブルに挿入します。

```
INSERT INTO Room VALUES ('Young Building', 222, NULL(), 'Lab')
```

次の例は、ISNULL の値の返し方を示します。

```
CREATE TABLE t8 (c1 INT, c2 CHAR(10));  
INSERT INTO t8 VALUES (100, 'string1');  
SELECT c1, c2, ISNULL(c1, 1000), ISNULL(C2, 'a string') from t8;
```

SELECT は 100 と 'string1' を返します。これは、c1 と c2 のどちらにも値が入っており、ヌルではないからです。

```
INSERT INTO t8 VALUES (NULL, NULL);  
SELECT c1, c2, ISNULL(c1, 1000), ISNULL(C2, 'a string') from t8
```

SELECT は 1000 と 'a string' を返します。これは、c1 と c2 のどちらにもヌルが入っているからです。

次のステートメントは、IFNULL および NULLIF スカラー関数の使い方を示します。これらの関数はそれぞれ、ヌルが存在するかしないか、等しいかどうかに従って値の代入を行う場合に使用します。

```
CREATE TABLE Demo (col1 CHAR(3));
INSERT INTO Demo VALUES ('abc');
INSERT INTO Demo VALUES (NULL);
INSERT INTO Demo VALUES ('xyz');
```

2 番目の行がヌル値を含んでいるので、その場所に 'foo' が代入されます。

```
SELECT IFNULL(col1, 'foo') FROM Demo
```

この結果、1 つの列から 3 行がフェッチされます。

```
"abc"
"foo"
"xyz"
```

1 列から 3 行フェッチされました。

最初の行に "abc" が含まれており、これは次の NULLIF 呼び出しの第 2 引数と合致します。

```
SELECT NULLIF(col1, 'abc') FROM Demo
```

その場所にヌルが返されます。

```
<Null>
<Null>
"xyz"
```

1 列から 3 行フェッチされました。

変換関数

変換関数は、式を特定のデータ型に変換します。CONVERT 関数は、値とそのテキスト表現の間の変換で使用するのに最も適しています。CAST 関数は、データ型をより細かく制御できますが、文字書式の制御性は低下します。CONVERT 関数は、リレーショナル型のサブセットのみをサポートすることに注意してください。

CAST 関数は（変換できる場合に）式を Zen リレーショナルデータ型へ変換します。CAST 関数は文字列内のバイナリ ゼロを変換できます。たとえば、CAST(c1 AS BINARY(10)) となります。ここで、c1 はバイナリ ゼロ (NULL) を含んでいる文字型の列です。

入力と出力が共に文字列の場合、CAST または CONVERT の出力は入力文字列と同じ照合順序になります。

CHAR、VARCHAR、または LONGVARCHAR と、NCHAR、NVARCHAR、または NLONGVARCHAR とにおける変換では、CHAR 値はデータベース コード ページを使用してエンコードされていることを前提とします。

TRY_CAST および TRY_CONVERT は、変換できないデータ値の扱いが異なることを除けば、それぞれ CAST および CONVERT と同じです。CAST および CONVERT の場合はクエリ全体が失敗しますが、TRY_CAST および TRY_CONVERT では、クエリ結果のうちクエリに失敗した列にヌル値が入ります。次の表は Zen の変換関数の一覧を示します。[変換関数の例](#)も参照してください。

関数	説明
CAST(<i>exp</i> AS <i>type</i>) TRY_CAST (<i>exp</i> AS <i>type</i>)	<i>exp</i> (式) を <i>type</i> へ変換します。この <i>type</i> は Zen で使用できるデータ型 に表示されるいずれかのデータ型です。桁数と小数位パラメーターも含めます。
CONVERT (<i>exp</i> , <i>type</i> [, <i>style</i>]) TRY_CONVERT (<i>exp</i> , <i>type</i> [, <i>style</i>])	以下の型の引数を使用して、<i>exp</i> を、指定された <i>type</i> に変換します。 SQL_BIGINT SQL_BINARY SQL_BIT SQL_CHAR SQL_DATE SQL_DECIMAL SQL_DOUBLE SQL_FLOAT SQL_GUID SQL_INTEGER SQL_LONGVARBINARY SQL_LONGVARCHAR SQL_NUMERIC SQL_REAL SQL_SMALLINT SQL_TIME SQL_TIMESTAMP SQL_TINYINT SQL_VARCHAR SQL_WCHAR SQL_WLONGVARCHAR SQL_WVARCHAR CONVERT の引数は、データ型のプレフィックスとして SQL_ を使用します。 Zen リレーショナルデータ型には SQL_ プレフィックスが含まれていません。 桁数と小数位はデフォルト値を取ります。 オプション パラメーターの <i>style</i> は、DATETIME データ型のみに適用されます。このパラメーターを使用すると、DATETIME データ型のミリ秒部分が切り捨てられます。<i>style</i> の値は 20 または 120 です。 どちらの値も正規の形式である yyyy-mm-dd hh:mm:ss を指定します。変換関数の例を参照してください。

変換関数の例

次の例では、DATE を CHAR にキャストしています。

```
CREATE TABLE u1(cdata DATE);
INSERT INTO u1 VALUES(curdate());
SELECT CAST(cdate as (CHAR20)) FROM u1;
```

現在の日付が 2004 年 1 月 1 日の場合、SELECT は 2004-01-01 を返します。

次の例は、それぞれ UBIGINT を CHAR へ、文字列データを DATE、TIME、および TIMESTAMP へ変換します。

```
SELECT CONVERT(id, SQL_CHAR), CONVERT('1995-06-05', SQL_DATE), CONVERT('10:10:10', SQL_TIME),
CONVERT('1990-10-10 10:10:10', SQL_TIMESTAMP) FROM Faculty
```

次の例は、文字列を DATE へ変換した後、DATE に 31 を加算します。

```
SELECT Name FROM Class WHERE Start_date > CONVERT ('1995-05-07', SQL_DATE) + 31
```

以下の例で、UNIQUEIDENTIFIER データ型をキャストおよび変換する方法を示します。

```
CREATE TABLE table1(col1 CHAR(36), col2 UNIQUEIDENTIFIER DEFAULT NEWID());
INSERT INTO table1 (col1) VALUES ('1129619D-772C-AAAB-B221-00FF00FF0099');
SELECT CAST(col1 AS UNIQUEIDENTIFIER) FROM table1;
SELECT CAST(col2 AS LONGVARCHAR) FROM table1;
SELECT CONVERT(col2 , SQL_CHAR) FROM table1;
SELECT CONVERT('1129619D-772C-AAAB-B221-00FF00FF0099' , SQL_GUID) FROM table1;
```

以下の例では、DATETIME データ型を *style* パラメーターを使用して、または使用しないで変換する方法を示します。

```
CREATE TABLE table2(col1 DATETIME);
INSERT INTO table2 (col1) VALUES ('2006-12-25 10:10:10.987');
SELECT CONVERT(col1 , SQL_CHAR, 20) FROM table2;
```

これは、2006-12-25 10:10:10 を返します。

```
SELECT CONVERT(col1, SQL_CHAR, 120) FROM table2
```

これは、2006-12-25 10:10:10 を返します。

```
SELECT CONVERT(col1, SQL_CHAR) FROM table2
```

これは、`2006-12-25 10:10:10.987` を返します。

DATETIME にミリ秒を含めたい場合は、`style` パラメーターを省略します。

`style` パラメーターを使用する場合は、以下の要件に注意してください。

- `type` パラメーターは `SQL_CHAR` である必要があります。それ以外のデータ型は無視されます。
- 式の列データ型は `DATETIME` である必要があります。
- 許可される `style` 値は 20 と 120 だけです。それ以外の値ではエラーを返します。値は 20 でも 120 でも正規の形式である `yyyy-mm-dd hh:mm:ss` を指定します。

次の例では、`CAST` と `TRY_CAST` を使用した場合の結果の違いを示しています。

`CONVERT` と `TRY_CONVERT` でも同様の違いが発生します。

```
SELECT CAST ( '10' AS numeric(10,2)); - 成功 : 10.00 を返します
SELECT CAST( 'test' AS float ); - エラー : 式の評価エラーを返します
SELECT TRY_CAST ( '10' AS numeric(10,2)); - 成功 : 10.00 を返します
SELECT TRY_CAST ( 'test' AS float ); - 成功 : ヌルを返します
```

システム ストアド プロシージャ

システム ストアド プロシージャは、データ定義言語（DDL）では取り扱われない管理タスクや情報タスクの遂行に利用できます。

Zen システム ストアド プロシージャ

システム ストアド プロシージャには **psp_** プレフィックスが付きます。次の表は、Zen で現在サポートされているシステム ストアド プロシージャの一覧を示します。

psp_columns	psp_column_attributes	psp_column_rights
psp_dbnames	psp_fkeys	psp_groups
psp_help_sp	psp_help_trigger	psp_help_udf
psp_help_view	psp_indexes	psp_pkeys
psp_procedure_rights	psp_rename	psp_stored_procedures
psp_tables	psp_table_rights	psp_triggers
psp_udfs	psp_users	psp_view_rights
psp_views		

特段の記述がない限り、システム ストアド プロシージャの例は、**Demodata** サンプルデータベースを使用しているか、または **Zen** システム テーブルを参照しています。

データベース内のシステム ストアド プロシージャを実行して、セキュリティで保護されたデータベースから情報を取得しようとする、エラーが発生します。セキュリティで保護されたデータベース内の情報にほかのデータベースからアクセスすることはできません。

メモ： **psp_** プレフィックスを付けた名前のストアド プロシージャを作成しないでください。システム ストアド プロシージャと同じ名前を持つユーザー作成のストアド プロシージャの実行は失敗します。

psp_columns

現在のデータベースまたは指定されたデータベースから、特定テーブルの列の一覧とそれに関連する情報を返します。

構文

```
call psp_columns(['database_qualifier'], 'table_name', ['column_name'])
```

引数

パラメーター	データ型	デフォルト値	説明
database_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
table_name	VARCHAR(255)	(デフォルト値なし)	列情報を要求するテーブルの名前
column_name	VARCHAR(255)	テーブルのすべての列	指定したテーブルの列名

返される結果セット

列名	データ型	説明
TABLE_QUALIFIER	VARCHAR(20)	データベースの名前
TABLE_OWNER	VARCHAR(20)	テーブル所有者の名前。テーブル所有者は、将来使用するために予約されています。現在は空（ヌル）が返ります。
TABLE_NAME	VARCHAR(255)	テーブルの名前
COLUMN_NAME	VARCHAR(255)	テーブルの列名
DATA_TYPE	SMALLINT	列のデータ型コード。 Zen で使用できるデータ型 を参照してください。
TYPE_NAME	VARCHAR(32)	列のデータ型の、DATA_TYPE 値に対応する名前
PRECISION	INTEGER	列の精度（列のデータ型が Decimal、Numeric などである場合）。 Decimal データ型の精度と小数位 を参照してください。
LENGTH	INTEGER	列の長さ。
SCALE	SMALLINT	列の小数点以下の桁数（列のデータ型が Decimal、Numeric などである場合）。
RADIX	SMALLINT	数値データ型の基数
NULLABLE	SMALLINT	ヌル値の許可を示します。 1 - ヌルを許可する 0 - ヌルを許可しない
REMARKS	VARCHAR(255)	備考フィールド

例

```
create table tx (c_binary BINARY(10),
  c_char CHAR(10),
  c_tinyint TINYINT,
  c_smallint SMALLINT,
  c_int INT,
  c_bigint BIGINT,
  c_utinyint UTINYINT);
call psp_columns(, 'tx',);
```

結果セット

Table_qualified	Table_owner	Table_name	Column_name	Data_type	Type_name	P	L	S	R	N	R
'demodata'	Null	tx	C_binary	-2	Binary	10	10	Null	Null	1	Null
'demodata'	Null	tx	C_char	-1	Char	10	10	Null	Null	1	Null
'demodata'	Null	tx	C_tinyint	-6	Tinyint	Null	1	0	10	1	Null

.....

凡例 : **P** = Precision; **L** = Length; **S** = Scale; **R** = Radix; **N** = Nullable; **R** = Remarks

mydatabase という名前のデータベースに tx というテーブルが含まれているとします。

```
call psp_columns('mydatabase', 'tx', )
```

結果セット

Table_qualifier	Table_owner	Table_name	Column_name	Data_type	Type_name	P	L	S	R	N	R
'wsrde'	Null	tx	C_binary	-2	Binary	10	10	Null	Null	1	Null
'wsrde'	Null	tx	C_char	-1	Char	10	10	Null	Null	1	Null
'wsrde'	Null	tx	C_tinyint	-6	Tinyint	Null	1	0	10	1	Null
.....											
凡例 : P = Precision; L = Length; S = Scale; R = Radix; N = Nullable; R = Remarks											

```
call psp_columns('mydatabase', 'tx', 'c_binary')
```

結果セット

Table_qualifier	Table_owner	Table_name	Column_name	Data_type	Type_name	P	L	S	R	N	R
'wsrde'	Null	tx	C_binary	-2	Binary	10	10	Null	Null	1	Null
凡例 : P = Precision; L = Length; S = Scale; R = Radix; N = Nullable; R = Remarks											

エラー状況

条件	エラー メッセージ
<i>database_qualifier</i> が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<i>database_qualifier</i> が未定義のデータベースである	テーブルを開けません : X\$File
<i>table_name</i> がデータベース内に定義されていない	エラーは返されず、結果も返されません
<i>table_name</i> がヌルである	テーブル名はヌルにできません
<i>table_name</i> が空文字列である	テーブル名は空文字列にできません。

条件	エラー メッセージ
<i>column_name</i> が空文字列である	列名は空文字列にできません
<i>column_name</i> がテーブル内に定義されていない	エラーは返されず、結果も返されません

psp_column_attributes

現在のデータベースまたは指定されたデータベースから、列属性の一覧とそれに関連する情報を返します。

構文

```
call psp_column_attributes(['database_qualifier'], ['table_name'], ['column_name'])
```


引数

パラメーター	データ型	デフォルト値	説明
database_qualifier	VARCHAR(20)	ログインしている現在のデータベース	詳細を取得するデータベースの名前
table_name	VARCHAR(255)	指定したデータベースのすべてのテーブル	列情報を要求するテーブルの名前
column_name	VARCHAR(255)	指定したテーブルのすべての列	指定したテーブルの列名

返される結果セット

列名	データ型	説明
TABLE_QUALIFIER	VARCHAR(20)	データベースの名前
TABLE_OWNER	VARCHAR(20)	テーブル所有者の名前
TABLE_NAME	VARCHAR(255)	テーブルの名前
COLUMN_NAME	VARCHAR(255)	テーブルの列名
ATTRIB_TYPE	CHAR(10)	列にデフォルト値が割り当てられている場合は "Default" 列に照合順序が使用されている場合は "Collate" 列に論理位置がある場合は "L" これ以外のすべての属性はヌルまたはブランク
ATTRIB_SIZE	USMALLINT	列属性のサイズ
ATTRIB_VALUE	LONGVARCHAR	列属性の値

例

```
create table tx (c_binary binary (10) default 01,  
  c_char char (11) default 'thisisatest',  
  c_tinyint TINYINT,  
  c_SMALLINT SMALLINT,  
  c_int INT,  
  c_bigint BIGINT,  
  c_utinyint uTINYINT);  
call psp_column_attributes(, , );
```

結果セット

Table_qualifier	Table_owner	Table_name	Column_name	Attrib_Type	Attrib_Size	Attrib_Value
'demodata'	Null	tx	C_binary	Default	2	01
'demodata'	Null	tx	C_char	Default	11	'Thisisatest'

```
create table tlogicalmv (col1 integer, col2 char(20))
alter table tlogicalmv psql_move col1 to 2
call psp_column_attributes(, 'tlogicalmv' , )
```

結果セット

Table_qualifier	Table_owner	Table_name	Column_name	Attrib_Type	Attrib_Size	Attrib_Value
'demodata'	Null	tlogicalmv	col2	L	1	1
'demodata'	Null	tlogicalmv	col1	L	1	2

```
call psp_column_attributes(, 'tx', 'c_binary')
```

結果セット

Table_qualifier	Table_owner	Table_name	Column_name	Attrib_Type	Attrib_Size	Attrib_Value
'demodata'	Null	tx	C_binary	Default	2	01

エラー状況

条件	エラー メッセージ
<i>database_qualifier</i> が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<i>database_qualifier</i> が未定義のデータベースである	テーブルを開けません：X\$File
<i>table_name</i> がデータベース内に定義されていない	エラーは返されず、結果も返されません
<i>table_name</i> がヌルである	テーブル名はヌルにできません
<i>table_name</i> が空文字列である	テーブル名は空文字列にできません。
<i>column_name</i> が空文字列である	列名は空文字列にできません
<i>column_name</i> がテーブル内に定義されていない	エラーは返されず、結果も返されません

psp_column_rights

現在のデータベースまたは指定されたデータベースから、特定テーブルの列権限の一覧とそれに関連する情報を返します。

メモ： このシステム ストアド プロシージャは、GRANT 構文によって明示的に指定されている列のアクセス権についてののみ一覧を返します。

構文

```
call psp_column_rights(['database_qualifier'], 'table_name', ['column_name'], ['user_name'])
```

引数

パラメーター	データ型	デフォルト値	説明
database_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
table_name	VARCHAR(255)	(デフォルト値なし)	アクセス権が指定されているテーブルの名前
column_name	VARCHAR(255)	指定したテーブルのすべての列	アクセス権を取得する列の名前

パラメーター	データ型	デフォルト値	説明
user_name	VARCHAR(255)	現在のユーザー	列のアクセス権の一覧を取得する必要があるユーザーの名前。パターン検索がサポートされます。

返される結果セット

列名	データ型	説明
TABLE_QUALIFIER	VARCHAR(20)	データベースの名前
TABLE_OWNER	VARCHAR(20)	テーブルの所有者の名前
USER_NAME (GRANTEE)	VARCHAR(255)	ユーザーの名前
TABLE_NAME	VARCHAR(255)	テーブルの名前
COLUMN_NAME	VARCHAR(255)	異なるアクセス権が付与されている列の名前
RIGHTS	VARCHAR(12)	次のいずれかの値です。 SELECT UPDATE INSERT

例

Demodata データベースの Dept テーブルに対して次の権限を付与した後、列権限を取得します。

```
GRANT SELECT(Name, Building_Name) ON Dept TO John;
GRANT UPDATE(Name) ON Dept TO Mary;
GRANT INSERT(Building_Name) ON Dept TO John;
Call psp_column_rights('Dept', , '%');
```

結果セット

Table_qualifier	Table_owner	User_name	Table_name	Column_name	Rights
Demodata	Null	John	Dept	Name	SELECT
Demodata	Null	John	Dept	Building_name	SELECT
Demodata	Null	John	Dept	Building_name	INSERT
Demodata	Null	Mary	Dept	Name	UPDATE

メモ： Master ユーザーには列のアクセス権が明示的に定義されていないため、`psp_column_rights` はそのユーザーの結果を返しません。

ユーザー John がデータベースにログオンしているとします。次のステートメントは、ユーザー John が Dept テーブルに対して持っている列権限を出力します。

```
call psp_column_rights ('demodata', 'Dept', ,)
```

結果セット

Table_qualifier	Table_owner	User_name	Table_name	Column_name	Rights
Demodata	Null	John	Dept	Building_name	INSERT
Demodata	Null	John	Dept	Building_name	SELECT
Demodata	Null	John	Dept	Name	SELECT

メモ： ユーザーにテーブルレベルの権限が付与されている場合は（例：`GRANT SELECT ON Dept TO Mary`）、`psp_column_rights` を呼び出してもアクセス権は何も返されません。アクセス権は特定の列ではなくテーブルに付与されています。

次のステートメントは、現在のユーザーを示す列 Name が Dept テーブルに対して持っている列権限を出力します。

```
call psp_column_rights ('demodata', 'dept', 'name',)
```

結果セット

Table_qualifier	Table_owner	User_name	Table_name	Column_name	Rights
Demodata	Null	John	Dept	Name	SELECT

次のステートメントは、ユーザー Mary が Dept テーブルに対して持っている列権限を出力します。

```
call psp_column_rights('demodata', 'dept', , 'Mary')
```

結果セット

Table_qualifier	Table_owner	User_name	Table_name	Column_name	Rights
Demodata	Null	Mary	Dept	Name	UPDATE

エラー状況

条件	エラー メッセージ
<i>database_qualifier</i> が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<i>table_name</i> がヌルである	テーブル名はヌルにできません。
<i>table_name</i> が空文字列である	テーブル名は空文字列にできません。
<i>column_name</i> が空文字列である	列名は空文字列にできません。
<i>user_name</i> が空文字列である	ユーザー名は空文字列にできません。

psp_dbnames

サーバー上にあるデータベース名の一覧を返します。

構文

```
call psp_dbnames(['dbnames.cfg_location' | null], ['dbname' | partial_dbname with wild card | null])
```

引数

パラメーター	データ型	デフォルト値	説明
dbnames.cfg_location	VARCHAR(1000)	(デフォルト値なし)	サーバー上の dbnames.cfg の場所、または辞書内の x\$dbnames を削除する場合はヌル。
dbname	VARCHAR(21)	(デフォルト値なし)	正確なデータベース名、または LIKE 演算子のワイルドカードを使用した部分的なデータベース名。ヌルを受け入れます。

メモ： dbnames.cfg_location がヌルの場合、返される結果セットはヌルになります。

返される結果セット

名前	データ型	説明
dbname	VARCHAR(21)	データベースの名前

例

dbnames.cfg ファイルは C:\ProgramData\Actian\Zen にあると仮定します。psp_dbnames の動作を示すには、以下を実行します。

1. 次のステートメントは、サーバー上にあるデータベース名の一覧を返します。

```
call psp_dbnames ( 'C:\ProgramData\Actian\Zen', null );
```

2. dbnames.cfg に対して間違った場所が指定されている場合は、エラーが発生します。たとえば、次のように指定します。

```
call psp_dbnames ( 'C:\ProgramData\Actian\Zen2', null );
```

データベース エンジン はエラー メッセージ "Btrieve ファイル ディレクトリが無効です (Btrieve エラー 35) " を返します。

3. エラーから回復するには、次のステートメントを実行して psp_dbnames をリセットします。

```
call psp_dbnames ( null, null );
```

ステートメントはヌルを返します。

psp_fkeys

現在のデータベース内の特定テーブルの外部キー情報を返します。

構文

```
call psp_fkeys(['table_qualifier'], 'pkey_table_name', ['fkey_table_name'])
```

引数

パラメーター	データ型	デフォルト値	説明
table_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
pkey_table_name	VARCHAR(255)	(デフォルト値なし)	主キー列に外部キーが関連付けられているテーブルの名前
fkey_table_name	VARCHAR(255)	(デフォルト値なし)	外部キー情報を取得する必要があるテーブルの名前

返される結果セット

列名	データ型	説明
PKTABLE_QUALIFIER	VARCHAR(20)	主キー テーブルのデータベース名
PKTABLE_OWNER	VARCHAR(20)	主キー テーブルの所有者の名前
PKTABLE_NAME	VARCHAR(255)	主キー テーブルの名前
PKCOLUMN_NAME	VARCHAR(255)	主キー列の列名
KEY_SEQ	USMALLINT	キーの順序。この列の値は、X\$Index の Xi\$Part に対応します。X\$Index を参照してください。
FKTABLE_QUALIFIER	VARCHAR(20)	外部キー テーブルのデータベース名
FKTABLE_OWNER	VARCHAR(20)	外部キー テーブルの所有者の名前
FKTABLE_NAME	VARCHAR(255)	外部キー テーブルの名前
FKCOLUMN_NAME	VARCHAR(255)	外部キー列の列名
UPDATE_RULE	Utinyint	更新規則
DELETE_RULE	Utinyint	削除規則

列名	データ型	説明
PK_NAME	VARCHAR(255)	主キーの名前
FK_NAME	VARCHAR(255)	外部キーの名前

例

```
CREATE TABLE Employee
(
  Id INTEGER NOT NULL,
  Name VARCHAR(50) NOT NULL,
  SupId INTEGER NOT NULL
);
ALTER TABLE Employee
  ADD CONSTRAINT EmpPkey
  PRIMARY KEY(Id);
ALTER TABLE Employee
  ADD CONSTRAINT ForgnKey
  FOREIGN KEY(SupId) REFERENCES
  Employee(Id) ON DELETE CASCADE;
call psp_fkeys('Employee',);
```

結果セット

PkQ	PkO	PkT	PkCol	Seq	FkQ	FkO	FkT	FkCol	UR	DR	PK	FK
Demo data	Null	Emplo yee	Id	0	Demo data	Null	Employ ee	Supid	1	2	EmpP key	Forgn Key

凡例 : **PkQ** = Pkey_table_qualifier; **PkO** = Pkey_table_owner; **PkT** = Pktable_name; **PkCol** = Pk_column_name; **Seq** = Key_seq; **FkQ** = Fktable_qualifier; **FkO** = Fktable_owner; **FkT** = Fktable_name; **FkCol** = Fkcolumn_name; **UR** = Update_rule; **DR** = Delete_rule; **Pk** = Pk_name; **FK** = Fk_name

エラー状況

条件	エラー メッセージ
<i>table_qualifier</i> が空文字列である	テーブル名は空文字列にできません。
<i>pKey_table_name</i> が空文字列である	主キー テーブル名は空文字列にできません。
<i>pKey_table_name</i> がヌルである	引数またはデフォルト値が指定されていません。引数 2.
<i>fKey_table_name</i> が空文字列である	外部キー テーブル名は空文字列にできません。

psp_groups

現在のデータベースまたは指定されたデータベースから、グループの一覧とそれに関連する情報を返します。

構文

```
call psp_groups(['database_qualifier'], ['group_name'])
```

引数

パラメーター	データ型	デフォルト値	説明
database_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前

パラメーター	データ型	デフォルト値	説明
group_name	VARCHAR(255)	(デフォルト値なし)	グループ情報の取得に使用するグループの名前。パターン検索がサポートされます。

返される結果セット

列名	データ型	説明
DATABASE_QUALIFIER	VARCHAR(20)	データベースの名前
GROUP_ID	USMALLINT	グループ ID
GROUP_NAME	VARCHAR(255)	グループの名前

例

Demodata サンプル データベースに 2 つのグループ、DevGrp1 と DevGrp2 が定義されているとします。

```
call psp_groups(,)
```

結果セット

database_qualifier	Group_Id	Group_Name
Demodata	1	PUBLIC
Demodata	2	DevGrp1
Demodata	3	DevGrp2

```
call psp_groups('Demodata', 'D%')
```

結果セット

database_qualifier	Group_Id	Group_Name
Demodata	2	DevGrp1

database_qualifier	Group_Id	Group_Name
Demodata	3	DevGrp2

エラー状況

条件	エラー メッセージ
<i>database_qualifier</i> が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<i>group_name</i> が空文字列である	グループ名は空文字列にできません。

psp_help_sp

現在のデータベースまたは指定されたデータベースから、特定のストアド プロシージャの定義テキストを返します。

構文

```
call psp_help_sp('[database_qualifier]', 'procedure_name')
```

引数

パラメーター	データ型	デフォルト値	説明
database_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
procedure_name	CHAR(255)	(デフォルト値なし)	定義テキストを要求するプロシージャの名前。パターン検索はサポートされません。

返される結果セット

列名	データ型	説明
DATABASE_QUALIFIER	VARCHAR(20)	データベースの名前
SP_TEXT	LONGVARCHAR	ストアド プロシージャの定義テキスト

例

Demodata サンプル データベースには、"Myproc" として保存された次のようなストアド プロシージャが含まれているとします。

```
Create procedure Myproc(:a integer, OUT :b integer) as
Begin
  Set :a = :a + 10;
  Set :b = :a;
End
```

次のステートメントは、現在のデータベース内にあるストアド プロシージャ "Myproc" の定義テキストを出力します。

```
call psp_help_sp(, 'Myproc')
```

結果セット

Database_Qualifier	SP_TEXT
Demodata	Create procedure Myproc(:a integer, OUT :b integer) as Begin Set :a = :a + 10; Set :b = :a; End

"wsrde" という名前のデータベースに、"Myproc1" として保存された次のようなストアド プロシージャが含まれているとします。

```
Create procedure Myproc1(:a integer) returns (name char(20))
as
Begin
    Select name from employee where Id =:a;
End
```

次のステートメントは、データベース "wsrde" 内にあるストアド プロシージャ "Myproc1" の定義テキストを出力します。

```
call psp_help_sp('wsrde', 'Myproc1')
```

結果セット

Database_Qualifier	SP_TEXT
wsrde	Create procedure Myproc1(:a integer) returns (name char(20)) as Begin Select name from employee where Id =:a; End

エラー状況

条件	エラー メッセージ
<i>database_qualifier</i> が空文字列またはヌルである	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<i>procedure_name</i> がヌルである	引数またはデフォルト値が指定されていません。
<i>procedure_name</i> が空文字列である	プロシージャ名は空文字列にできません。

psp_help_trigger

現在のデータベースまたは指定されたデータベースから、トリガーの定義テキストを返します。

構文

```
call psp_help_trigger (['database_qualifier'], 'trigger_name')
```

引数

パラメーター	データ型	デフォルト値	説明
database_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
trigger_name	VARCHAR(255)	(デフォルト値なし)	定義テキストを取得するトリガーの名前パターン検索がサポートされません。

返される結果セット

列名	データ型	説明
DATABASE_QUALIFIER	VARCHAR(20)	データベースの名前
TRIGGER_TEXT	LONGVARCHAR	トリガーの定義テキスト

例

次のステートメントにより、'MyInsert' トリガーの定義が出力されます。

```
CREATE TABLE A
(
col1 INTEGER,
col2 CHAR(255)
);
CREATE TABLE B
(
col1 INTEGER,
col2 CHAR(255)
);
CREATE TRIGGER MyInsert
AFTER INSERT ON A
FOR EACH ROW
INSERT INTO B VALUES
(NEW.col1, NEW.col2);
call psp_help_trigger('MyIns%');
```

結果セット

Database_Qualifier	TRIGGER_TEXT
Demodata	<pre>CREATE TRIGGER MyInsert AFTER INSERT ON A FOR EACH ROW INSERT INTO B VALUES (NEW.col1, NEW.col2);</pre>

エラー状況

条件	エラー メッセージ
<i>database_qualifier</i> が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<i>trigger_name</i> がヌルである	引数またはデフォルト値が指定されていません。
<i>trigger_name</i> が空文字列である	トリガー名は空文字列にできません。

psp_help_udf

現在のデータベースまたは指定されたデータベースから、特定のユーザー定義関数 (UDF) のテキストを返します。

構文

```
call psp_help_udf (['database_qualifier'], 'udf_name')
```


引数

パラメーター	データ型	デフォルト値	説明
database_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
udf_name	VARCHAR(255)	(デフォルト値なし)	関数テキストを要求するユーザー定義関数の名前。パターン検索がサポートされます。

返される結果セット

列名	データ型	説明
DATABASE_QUALIFIER	VARCHAR(20)	データベースの名前
UDF_TEXT	LONGVARCHAR	ユーザー定義関数のテキスト

例

```
call psp_help_udf(, 'Myfunction')
```

結果セット

Database_Qualifier	UDF_TEXT
Demodata	Create function Myfunction(:a integer) Returns integer as Begin Return :a * :a; End

```
call psp_help_udf('mydbase', 'Get%')
```

結果セット

Database_Qualifier	UDF_TEXT
wsrde	<pre>CREATE FUNCTION GetSmallest(:A integer, :B Integer) RETURNS Integer AS BEGIN DECLARE :smallest INTEGER IF (:A < :B) THEN SET :smallest = :A; ELSE SET :smallest = :B; END IF; RETURN :smallest; END</pre>

エラー状況

条件	エラー メッセージ
<i>database_qualifier</i> が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<i>udf_name</i> がヌルである	引数またはデフォルト値が指定されていません。
<i>udf_name</i> が空文字列である	ユーザー定義関数名は空文字列にできません。

psp_help_view

現在のデータベースまたは指定されたデータベースから、ビューの定義テキストを返します。

構文

```
call psp_help_view(['database_qualifier'], 'view_name')
```

引数

パラメーター	データ型	デフォルト値	説明
database_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前

パラメーター	データ型	デフォルト値	説明
view_name	VARCHAR(255)	(デフォルト値なし)	定義テキストを要求するビューの名前。パターン検索がサポートされます。

返される結果セット

列名	データ型	説明
DATABASE_QUALIFIER	VARCHAR(20)	データベースの名前
VIEW_TEXT	LONGVARCHAR	ビューの定義テキスト

例

Demodata サンプル データベースに次のビューが存在するとします。

```
CREATE VIEW vw_Person (lastn,firstn,phone) AS
SELECT Last_Name, First_Name, Phone
FROM Person;
```

次のステートメントは、Demodata データベース内のビュー "vw_Person" の定義テキストを返します。

```
call psp_help_view('vw_Person')
```

または

```
call psp_help_view('vw_%')
```

結果セット

Database_Qualifier	VIEW_TEXT
Demodata	SELECT "T1" ."Last_Name" ,"T1" ."First_Name" ,"T1" ."Phone" FROM "Person" "T1"

エラー状況

条件	エラー メッセージ
database_qualifier が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
view_name がヌルである	引数またはデフォルト値が指定されていません。

条件	エラー メッセージ
<code>view_name</code> が空文字列である	ビュー名は空文字列にできません。

psp_indexes

指定したテーブルに定義されているインデックスの一覧を返します。また、各インデックスについて、X\$Index テーブルに保存されているとおりにインデックスプロパティを列挙します。

構文

```
call psp_indexes(['table_qualifier'], ['table_name'])
```

引数

パラメーター	データ型	デフォルト値	説明
<code>table_qualifier</code>	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
<code>table_name</code>	VARCHAR(255)	(デフォルト値なし)	インデックスを取得するテーブルの名前パターン検索がサポートされます。

返される結果セット

列名	データ型	説明
TABLE_QUALIFIER	VARCHAR(20)	データベースの名前
TABLE_OWNER	VARCHAR(20)	主キー テーブルの所有者の名前
TABLE_NAME	VARCHAR(255)	主キー テーブルの名前
INDEX_NAME	VARCHAR(255)	インデックスの名前
INDEX_TYPE	VARCHAR(20)	インデックスの種類 : primary (主)、 foreign (外部)、 normal (標準)
COLUMN_NAME	VARCHAR(255)	インデックスが定義されている列の名前
ORDINAL_POSITION	USMALLINT	インデックスの位置を表す序数

列名	データ型	説明
DUPLICATES_ALLOWED	CHAR(3)	Yes - 重複インデックスの場合 No - 重複インデックスでない場合
UPDATABLE	CHAR(3)	Yes - インデックスが更新可能な場合 No - インデックスが更新可能でない場合
CASE_SENSITIVE	CHAR(3)	Yes - インデックスが大文字小文字を区別する場合 No - インデックスが大文字小文字を区別しない場合
ASC_DESC	CHAR(1)	D - 降順 A - 昇順
NAMED_INDEX	CHAR(3)	Yes - 名前付きインデックスの場合 No - 名前付きインデックスでない場合

例

```
call psp_indexes(,)
```

結果セット

Qual	TO	TN	IN	IT	CN	Opos	Dup	Up	Case	A/D	NI
Demo data	Null	Billing	Student_Trans	Normal Index	Student_ID	0	No	Yes	No	A	Yes
Demo data	Null	Billing	Student_Trans	Normal Index	Transaction_Number	1	No	Yes	No	A	Yes
Demo data	Null	Billing	Student_Trans	Normal Index	Log	2	No	Yes	No	A	Yes
.....											

凡例 : **Qual** = Table_qualifier; **TO** = Table_owner; **TN** = Table_name; **IN** = Index_name; **IT** = Index_type; **CN** = Column_name; **Opos** = Ordinal_position; **Dup** = Duplicates_allowed; **UP** = Updatable; **Case** = Case_sensitive; **A/D** = Asc_desc; **NI** = Named_index

```
call psp_indexes('demodata', 'Dep%')
```

結果セット

Qual	TO	TN	IN	IT	CN	Opos	Dup	Up	Case	A/D	NI
Demo data	Null	Dept	Building_Room	Normal Index	Building_Name	0	Yes	Yes	Yes	A	Yes
Demo data	Null	Dept	Building_Room	Normal Index	Room_Number	1	Yes	Yes	No	A	Yes
Demo data	Null	Dept	Dept_Head	Normal Index	Head_Of_Dept	0	No	Yes	No	A	Yes
Demo data	Null	Dept	Dept_Name	Normal Index	Name	0	No	Yes	Yes	A	Yes

凡例: **Qual** = Table_qualifier; **TO** = Table_owner; **TN** = Table_name; **IN** = Index_name; **IT** = Index_type; **CN** = Column_name; **Opos** = Ordinal_position; **Dup** = Duplicates_allowed; **UP** = Updatable; **Case** = Case_sensitive; **A/D** = Asc_desc; **NI** = Named_index

エラー状況

条件	エラー メッセージ
<i>database_qualifier</i> が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<i>table_name</i> が空文字列である	ビュー名は空文字列にできません。

psp_pkeys

現在のデータベースまたは指定されたデータベースから、特定テーブルの主キー情報を返します。

構文

```
call psp_pkeys(['pkey_table_qualifier'] 'table_name')
```

引数

パラメーター	データ型	デフォルト値	説明
pkey_table_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
table_name	VARCHAR(255)	(デフォルト値なし)	主キー情報を要求するテーブルの名前。パターン検索がサポートされます。

返される結果セット

列名	データ型	説明
TABLE_QUALIFIER	VARCHAR(20)	データベースの名前
TABLE_OWNER	VARCHAR(20)	主キー テーブルの所有者の名前
TABLE_NAME	VARCHAR(255)	主キー テーブルの名前
COLUMN_NAME	VARCHAR(255)	主キー列の名前
COLUMN_SEQ	USMALLINT	列の順序 (セグメント インデックス)
PK_NAME	VARCHAR(255)	主キーの名前

例

次のステートメントは、'pkeytest1' テーブルに定義されている主キーに関する情報を返します。

```
CREATE TABLE pkeytest1
(
  col1 int NOT NULL,
  col2 int NOT NULL,
  col3 VARCHAR(20) NOT NULL,
  PRIMARY KEY(col1, col2),
  UNIQUE(col3)
);

call psp_pkeys('pkeytest1');
```

結果セット

Table_qualifier	Table_owner	Table_name	Column_name	Column_Seq	PK_name
'demodata'	Null	Pkeytest1	Col1	0	PK_col1
'demodata'	Null	Pkeytest1	Col2	1	PK_col1

エラー状況

条件	エラー メッセージ
<i>pkey_table_qualifier</i> が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<i>table_name</i> がヌルである	引数またはデフォルト値が指定されていません。
<i>table_name</i> が空文字列である	テーブル名は空文字列にできません。

psp_procedure_rights

現在のデータベースまたは指定されたデータベースから、特定ストアド プロシージャのプロシージャ権限の一覧とそれに関連する情報を返します。ストアド プロシージャは、信頼されるものでも信頼されないものでもかまいません。[信頼されるオブジェクト](#)と[信頼されないオブジェクト](#)を参照してください。

構文

```
call psp_procedure_rights(['database_qualifier'], ['procedure_name'], ['user_name'])
```

引数

パラメーター	データ型	デフォルト値	説明
database_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
procedure_name	VARCHAR(255)	(デフォルト値なし)	アクセス権が指定されているプロシージャの名前。パターン検索がサポートされます。

パラメーター	データ型	デフォルト値	説明
user_name	VARCHAR(255)	現在のユーザー	プロシージャのアクセス権の一覧を取得する必要があるユーザーの名前。パターン検索がサポートされます。

返される結果セット

列名	データ型	説明
PROCEDURE_QUALIFIER	VARCHAR(20)	データベースの名前
PROCEDURE_OWNER	VARCHAR(20)	プロシージャの所有者の名前
USER_NAME (GRANTEE)	VARCHAR(255)	ユーザーの名前
PROCEDURE_NAME	VARCHAR(255)	プロシージャの名前
RIGHTS	VARCHAR(12)	次のいずれかの値です。 ALTER EXECUTE RIGHTS は、メタデータ バージョン 2 を使用するデータベースのプロシ ージャにのみ付随します。

例

Demodata サンプル データベースに対して次の権限が存在するとします。

```
GRANT EXECUTE ON PROCEDURE Dept1_Proc TO John;
GRANT ALTER ON PROCEDURE Dept1_Proc TO Mary;
GRANT ALTER ON PROCEDURE Dept1_Proc TO John;
GRANT EXECUTE ON PROCEDURE Proc2 TO Mary;
GRANT ALTER ON PROCEDURE Proc2 TO Mary;
GRANT ALTER ON PROCEDURE MyProc TO Mary;
```

次のステートメントは、ユーザー "John" が "Dept1_Proc" ストアド プロシージャに対して持っている権限を出力します。

```
call psp_procedure_rights(, 'Dept1_Proc', 'John');
```

結果セット

Procedure_Qualifier	Procedure_owner	User_name	Procedure_name	Rights
Demodata	Null	John	Dept1_Proc	ALTER
Demodata	Null	John	Dept1_Proc	EXECUTE

次のステートメントは、ユーザー "Mary" が "Proc2" ストアド プロシージャに対して持っている権限を出力します。

```
call psp_procedure_rights('demodata', '%Pr%', 'M%')
```

結果セット

Procedure_Qualifier	Procedure_owner	User_name	Procedure_name	Rights
Demodata	Null	Mary	MyProc	ALTER
Demodata	Null	Mary	Proc2	ALTER
Demodata	Null	Mary	Proc2	EXECUTE

エラー状況

条件	エラー メッセージ
<i>database_qualifier</i> が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<i>procedure_name</i> が空文字列である	プロシージャ名は空文字列にできません。
<i>user_name</i> が空文字列である	ユーザー名は空文字列にできません。
メタデータ バージョン 1 を使用するデータベースで <code>psp_procedure_rights</code> を呼び出す	メタデータ バージョン 1 では、ビューおよびストアド プロシージャの権限はサポートされていません。

psp_rename

お使いのマシンが現在接続されているデータベース内の列、インデックス、関数、プロシージャ、テーブル、トリガー、またはビューの名前を変更します。

構文

```
call psp_rename('object_name','new_name','object_type')
```

引数

パラメーター	データ型	説明
object_name	VARCHAR(776)	列、インデックス、ユーザー定義関数、ストアド プロシージャ、テーブル、トリガー、またはビューの現在の名前 object_name は、オブジェクトの種類に応じて特有の形式で指定する必要があります。 • 列：テーブル名 . 列名 • インデックス：テーブル名 . インデックス名 • 関数：関数名 • プロシージャ：プロシージャ名 • テーブル：テーブル名 • トリガ：テーブル名 . トリガ名 • ビュー：ビュー名
new_name	VARCHAR(776)	オブジェクトのユーザー定義名。名前は、オブジェクトの種類に合った名前付け規則に従っている必要があります。『 <i>Zen Programmer's Guide</i> 』の 名前付け規則 を参照してください。
object_type	VARCHAR(13)	名前を変更するオブジェクトの種類。object_type は次のいずれかでなければなりません。列、インデックス、関数、プロシージャ、テーブル、トリガー、またはビュー。

例

次のステートメントは、現在のデータベース内のストアド プロシージャ "checkstatus" の名前を "eligibility" に変更します。

```
call psp_rename('checkstatus', 'eligibility', 'PROCEDURE')
```

エラー状況

psp_rename から返されるすべてのエラーはステータス コード -5099 を使用します。
『Status Codes and Messages』の [-5099 : ストアド プロシージャに関連するエラー状況](#) を参照してください。

psp_stored_procedures

現在のデータベースまたは指定されたデータベースから、ストアド プロシージャの一覧とそれに関連する情報を返します。

構文

```
call psp_stored_procedures(['database_qualifier'], ['procedure_name'], ['procedure_type'])
```

引数

パラメーター	データ型	デフォルト値	説明
database_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
procedure_name	VARCHAR(255)	(デフォルト値なし)	情報を要求するストアド プロシージャの名前。パターン検索がサポートされます。
procedure_type	VARCHAR(5)	(デフォルト値なし)	'SP' - ストアド プロシージャを返します 'SSP' - システム ストアド プロシージャを返します (この種類は現在サポートされていません)

メモ : システム ストアド プロシージャは、Zen Control Center に表示されない内部データベース PERVASIVESYSDB に定義されています。

返される結果セット

列名	データ型	説明
PROCEDURE_QUALIFIER	VARCHAR(20)	データベースの名前

列名	データ型	説明
PROCEDURE_OWNER	VARCHAR(20)	プロシージャの所有者の名前
PROCEDURE_NAME	VARCHAR(255)	プロシージャの名前
PROCEDURE_TYPE	VARCHAR(25)	プロシージャの種類。STORED PROCEDURE と SYSTEM STORED PROCEDURE があります。
NUM_INPUT_PARAMS	INT	ヌルを返します。これは、Zen DSN に対して SQLPROCEDURES を実行した場合にはヌルが返されるからです。
NUM_OUTPUT_PARAMS	INT	ヌルを返します。これは、Zen DSN に対して SQLPROCEDURES を実行した場合にはヌルが返されるからです。
NUM_RESULT_SETS	INT	ヌルを返します。これは、Zen DSN に対して SQLPROCEDURES を実行した場合にはヌルが返されるからです。
REMARKS	VARCHAR(255)	備考
TRUSTEE	INTEGER	メタデータバージョン 2 の場合、信頼されるストアード プロシージャでは 0、信頼されないストアード プロシージャでは -1 を返します。メタデータバージョン 1 の場合、TRUSTEE 列は空になります。

例

現在のデータベース mydbase には 2 つのストアード プロシージャ、myproc1 と myproc2 が含まれているとします。次のステートメントは、それらに関する情報を一覧表示します。

```
Call psp_stored_procedures(, ,)
```

結果セット

Qualifier 1	Owner 1	Name ¹	Type ¹	Num_ input _params	Num_ output_ _params	Num_ result_ _sets	Remarks	Trustee
mydbase	Null	Myproc1	Stored Procedure	Null	Null	Null	Null	

Qualifier 1	Owner 1	Name ¹	Type ¹	Num_ input_ _params	Num_ output_ _params	Num_ result_ _sets	Remarks	Trustee
mydbase	Null	Myproc2	Stored Procedure	Null	Null	Null	Null	

¹ 完全な列名は、この名前の前に "procedure_" が付きます。たとえば、Procedure_qualifier や procedure_owner などのようになります。

次のステートメントは、PERVASIVESYSDB 内部データベースに含まれるストアードプロシージャに関する情報を一覧表示します。

```
call psp_stored_procedures('PERVASIVESYSDB', 'psp_u%', 'SP')
```

結果セット

Qualifier	Owner	Name	Type	Num_ input_ _params	Num_ output_ _params	Num_ result_ _sets	Remarks	Trustee
pervasivesystdb	Null	psp_udfs	Stored Procedure	Null	Null	Null	Null	
pervasivesystdb	Null	psp_users	Stored Procedure	Null	Null	Null	Null	

メモ : qualifier、owner、name、および type の完全な列名は、この名前の前に "procedure_" が付きます。たとえば、Procedure_qualifier や procedure_owner などのようになります。

エラー状況

条件	エラー メッセージ
<i>database_qualifier</i> が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<i>table_name</i> が空文字列である	テーブル名は空文字列にできません。
<i>procedure_type</i> が空文字列である	プロシージャの種類は空文字列にできません。
<i>procedure_type</i> が SP、SSP、またはヌル以外の値である	プロシージャの種類には SP、SSP、またはヌルを指定できます

psp_tables

現在のデータベースまたは指定されたデータベースから、テーブルの一覧とそれに関連する情報を返します。

構文

```
call psp_tables(['database_qualifier'], ['table_name'], ['table_type'])
```

引数

パラメーター	データ型	デフォルト値	説明
database_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
table_name	VARCHAR(255)	(デフォルト値なし)	情報を取得する必要があるテーブルの名前パターン検索がサポートされます。
table_type	VARCHAR(20)	(デフォルト値なし)	次のいずれかでなければなりません。 'User table' - ユーザー テーブルのみ返します 'System table' - すべてのシステム テーブルを返します ヌル - すべてのテーブルを返します

返される結果セット

列名	データ型	説明
TABLE_QUALIFIER	VARCHAR(20)	データベースの名前
TABLE_OWNER	VARCHAR(20)	テーブル所有者の名前
TABLE_NAME	VARCHAR(255)	テーブルの名前
TABLE_TYPE	VARCHAR(15)	System table - テーブルがシステム テーブルの場合 User table - 任意のユーザーによって作成されたテーブルの場合
REMARKS	VARCHAR(255)	備考
FILE_LOCATION	VARCHAR(255)	ファイルが保存されている場所

例

```
call psp_tables(,,)
```

結果セット

Table_ Qualifier	Table_ owner	Table_ name	Table_ Type	Remarks	File_ location
Demodata	Null	X\$file	System table	Null	File.ddf
Demodata	Null	X\$field	System table	Null	Field.ddf
Demodata	Null	X\$Attrib	System table	Null	Attrib.ddf
Demodata	Null	Billing	User table	Null	Billing.mkd
.....					

```
call psp_tables(, , 'user table')
```


結果セット

Table_ Qualifier	Table_ owner	Table_ name	Table_ Type	Remarks	File_ location
Demodata	Null	Class	User table	Null	class.mkd
Demodata	Null	Billing	User table	Null	Billing.mkd
.....					

```
call psp_tables(, , 'system table')
```

結果セット

Table_ Qualifier	Table_ owner	Table_ name	Table_ Type	Remarks	File_ location
Demodata	Null	X\$file	System table	Null	File.ddf
Demodata	Null	X\$field	System table	Null	Field.ddf
Demodata	Null	X\$Attrib	System table	Null	Attrib.ddf
.....					

エラー状況

条件	エラー メッセージ
<i>database_qualifier</i> が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<i>table_name</i> が空文字列である	テーブル名は空文字列にできません。
<i>table_type</i> が空文字列である	テーブルの種類は空文字列にできません。
<i>table_type</i> が 'system table'、'user table'、またはヌル以外の値である	テーブルの種類には system table、user table、またはヌルを指定できます。

psp_table_rights

現在のデータベースまたは指定されたデータベースから、特定テーブルのテーブル権限の一覧とそれに関連する情報を返します。

構文

```
call psp_table_rights(['database_qualifier'], ['table_name'], ['user_name'])
```

引数

パラメーター	データ型	デフォルト値	説明
database_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
table_name	VARCHAR(255)	すべてのテーブル	アクセス権が指定されているテーブルの名前パターン検索がサポートされます。
user_name	VARCHAR(255)	現在のユーザー	テーブルのアクセス権の一覧を取得する必要があるユーザーの名前。パターン検索がサポートされます。

返される結果セット

列名	データ型	説明
TABLE_QUALIFIER	VARCHAR(20)	データベースの名前
TABLE_OWNER	VARCHAR(20)	テーブルの所有者の名前
USER_NAME (GRANTEE)	VARCHAR(255)	ユーザーの名前
TABLE_NAME	VARCHAR(255)	テーブルの名前
RIGHTS	VARCHAR(12)	次のいずれかの値です。 SELECT ALTER DELETE INSERT REFERENCES SELECT UPDATE

例

Demodata サンプル データベースに対して次の権限が存在するとします。

```
GRANT SELECT ON Dept TO John;  
GRANT ALTER ON Dept TO John;  
GRANT DELETE ON Dept TO John;  
GRANT SELECT ON Class TO Mary;  
GRANT ALTER ON Class TO Mary;
```

次のステートメントは、ユーザー "John" が現在のデータベース（Demodata）の "Dept" テーブルに対して持っているテーブル権限を出力します。

```
call psp_table_rights(, 'Dept', 'John');
```

結果セット

Table_qualifier	Table_owner	User_name	Table_name	Rights
Demodata	Null	John	Dept	ALTER
Demodata	Null	John	Dept	DELETE
Demodata	Null	John	Dept	SELECT

ユーザー "Mary" がデータベースにログオンしているとします。次のステートメントは、現在のユーザー（Mary）が Demodata データベースの "Class" テーブルに対して持っているテーブル権限を出力します。

```
call psp_table_rights('demodata', 'cl%', )
```

結果セット

Table_qualifier	Table_owner	User_name	Table_name	Rights
Demodata	Null	Mary	Class	SELECT
Demodata	Null	Mary	Class	ALTER

エラー状況

条件	エラー メッセージ
<i>database_qualifier</i> が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<i>table_name</i> が空文字列である	テーブル名は空文字列にできません。
<i>user_name</i> が空文字列である	ユーザー名は空文字列にできません。

psp_triggers

現在のデータベースまたは指定されたデータベースから、トリガーの一覧とそれに関連する情報を返します。

構文

```
call psp_triggers(['database_qualifier'], ['table_name'])
```

引数

パラメーター	データ型	デフォルト値	説明
database_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
table_name	VARCHAR(255)	すべてのテーブル	トリガーが定義されているテーブルの名前。パターン検索がサポートされます。

返される結果セット

列名	データ型	説明
TRIGGER_QUALIFIER	VARCHAR(20)	データベースの名前
TRIGGER_OWNER	VARCHAR(20)	トリガーの所有者の名前
TABLE_NAME	VARCHAR(255)	トリガーが定義されているテーブルの名前。
TRIGGER_NAME	VARCHAR(255)	トリガーの名前
ISUPDATE	UTINYINT	UPDATE トリガーである場合に設定されます

列名	データ型	説明
ISDELETE	UTINYINT	DELETE トリガーである場合に設定されます
ISINSERT	UTINYINT	INSERT トリガーである場合に設定されます
ISAFTER	UTINYINT	トリガーの動作時間が "AFTER" である場合に設定されます
ISBEFORE	UTINYINT	トリガーの動作時間が "BEFORE" である場合に設定されます
REMARKS	VARCHAR(255)	備考

例

現在のデータベースは mydbase であるとします。次のステートメントは、このデータベースに定義されているトリガーの一覧を返します。

```
CREATE TABLE A
(
  col1 INTEGER,
  col2 CHAR(255)
);

CREATE TABLE B
(
  col1 INTEGER,
  col2 CHAR(255)
);

CREATE TRIGGER Insert
AFTER INSERT ON A
FOR EACH ROW
INSERT INTO B VALUES
(NEW.col1, NEW.col2);

call psp_triggers(,);
```

結果セット

Trigger qualifier	Trigger owner	Table name	Trigger name	isupdate	isdelete	isinsert	isafter	isbefore	Remarks
mydbase	Null	A	Insert	0	0	1	0	0	Null

エラー状況

条件	エラー メッセージ
<code>database_qualifier</code> が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<code>table_name</code> が空文字列である	テーブル名は空文字列にできません。

psp_udfs

現在のデータベースまたは指定されたデータベースから、ユーザー定義関数（UDF）の一覧とそれに関連する情報を返します。

構文

```
call psp_udfs(['database_qualifier'], ['udf_name'])
```

引数

パラメーター	データ型	デフォルト値	説明
<code>database_qualifier</code>	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
<code>udf_name</code>	VARCHAR(255)	すべてのユーザー定義関数	詳細を要求する UDF の名前。パターン検索がサポートされます。

返される結果セット

列名	データ型	説明
UDF_QUALIFIER	VARCHAR(20)	データベースの名前
UDF_OWNER	VARCHAR(20)	UDF の所有者の名前
UDF_NAME	VARCHAR(255)	UDF の名前
UDF_TYPE	VARCHAR(25)	UDF の種類（常に 1 に設定されます） 特別な種類の UDF タイプは現在サポートされていません。

列名	データ型	説明
NUM_INPUT_PARAMS	INT	ヌルを返します。これは、Zen DSN に対して SQLPROCEDURES を実行した場合にはヌルが返されるからです。
NUM_OUTPUT_PARAMS	INT	1 を返します。UDF はスカラー値のみを返します。
NUM_RESULT_SETS	INT	0 を返します。UDF は結果セットを返しません。
REMARKS	VARCHAR(255)	備考

例

現在のデータベース mydbase には 2 つのユーザー定義関数、calcinterest と factorial が含まれているとします。

```
call psp_udfs(, )
```

結果セット

UDF_qualifier	UDF_owner	UDF_name	Udf_type	Num_input_params	Num_output_params	Num_result_sets	Remarks
mydbase	Null	CalcInterest	1	Null	1	0	Null
mydbase	Null	Factorial	1	Null	1	0	Null

エラー状況

条件	エラー メッセージ
<i>database_qualifier</i> が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<i>udf_name</i> が空文字列である	ユーザー定義関数名は空文字列にできません。

psp_users

現在のデータベースまたは指定されたデータベースから、ユーザーの一覧とそれに関連する情報を返します。

構文

```
call psp_users(['database_qualifier'], ['group_name'], ['user_name'])
```

引数

パラメーター	データ型	デフォルト値	説明
database_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
group_name	VARCHAR(255)	すべてのグループ (group_name がヌルの場合)	ユーザー情報の取得に使用するグループの名前。パターン検索がサポートされます。グループ名が指定されている（つまり、ヌルでない）場合は、その同じグループに属しているユーザーのみが返されます。
user_name	VARCHAR(255)	すべてのユーザー (user_name がヌルの場合)	ユーザーの名前。パターン検索がサポートされます。

返される結果セット

列名	データ型	説明
DATABASE_QUALIFIER	VARCHAR(20)	データベースの名前
GROUP_ID	USMALLINT	ユーザーが属しているグループのグループ ID
GROUP_NAME	VARCHAR(255)	ユーザーが属しているグループの名前
USER_ID	USMALLINT	ユーザーの ID
USER_NAME	VARCHAR(255)	ユーザーの名前

例

現在のデータベース mydbase は、ユーザー John、Mary、および Michael と、グループ DevGrp および DevGrp1 を所有しています。

```
call psp_users(, , )
```


結果セット

database_qualifier	Group_Id	Group_Name	User_Id	User_Name
Demodata	1	DevGrp	3	John
Demodata	2	DevGrp1	1	Mary
Demodata	1	DevGrp	4	Michael

```
call psp_users(, 'Devgrp', )
```

結果セット

database_qualifier	Group_Id	Group_Name	User_Id	User_Name
Demodata	1	DevGrp	3	John
Demodata	2	DevGrp	4	Michael

エラー状況

条件	エラー メッセージ
<i>database_qualifier</i> が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<i>user_name</i> がヌルである	ユーザー名はヌルにできません。
<i>group_name</i> が空文字列である	グループ名は空文字列にできません。

psp_view_rights

現在のデータベースまたは指定されたデータベースから、特定ビューのビュー権限の一覧とそれに関連する情報を返します。ビューは、信頼されるものでも信頼されないものでもかまいません。[信頼されるオブジェクトと信頼されないオブジェクト](#)を参照してください。

`psp_view_rights` は、メタデータ バージョン 2 を使用するデータベースにのみ適用されます。

構文

```
call psp_view_rights(['database_qualifier'], ['view_name'], ['user_name'])
```

引数

パラメーター	データ型	デフォルト値	説明
database_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
view_name	VARCHAR(255)	すべてのビュー (view_name がヌルの場合)	アクセス権が指定されているビューの名前。パターン検索がサポートされます。
user_name	VARCHAR(255)	現在のユーザー (user_name がヌルの場合)	ビューのアクセス権の一覧を取得する必要があるユーザーの名前。パターン検索がサポートされます。

返される結果セット

列名	データ型	説明
VIEW_QUALIFIER	VARCHAR(20)	データベースの名前
VIEW_OWNER	VARCHAR(20)	ビューの所有者の名前
USER_NAME (GRANTEE)	VARCHAR(255)	ユーザーの名前
VIEW_NAME	VARCHAR(255)	ビューの名前
RIGHTS	VARCHAR(12)	次のいずれかの値です。 ALTER DELETE INSERT SELECT UPDATE

例

Demodata サンプル データベースに対して次の権限が存在するとします。

```
GRANT SELECT ON VIEW vw_Dept TO John;
GRANT ALTER ON VIEW vw_Dept TO John;
GRANT DELETE ON VIEW vw_Dept TO John;
GRANT SELECT ON VIEW vw_Class TO Mary;
GRANT ALTER ON VIEW vw_Class TO Mary;
GRANT SELECT ON VIEW vw_Class TO Prakesh;
```

次のステートメントは、ユーザー "John" が "vw_Dept" ビューに対して持っているビュー権限を出力します。

```
call psp_view_rights('vw_Dept', 'John');
```

結果セット

View_Qualifier	View_owner	User_name	View_name	Rights
Demodata	Null	John	vw_Dept	ALTER
Demodata	Null	John	vw_Dept	DELETE
Demodata	Null	John	vw_Dept	SELECT

ユーザー "Mary" がデータベースにログオンしているとします。次のステートメントは、現在のユーザー (Mary) がサンプル データベース Demodata 内のすべてのビューに対して持っているビュー権限を出力します。

```
call psp_view_rights('demodata', , )
```

結果セット

View_Qualifier	View_owner	User_name	View_name	Rights
Demodata	Null	Mary	vw_Class	ALTER
Demodata	Null	Mary	vw_Class	SELECT

次のステートメントは、ユーザー "Mary" が "vw_Class" ビューに対して持っているビュー権限を出力します。

```
call psp_view_rights('demodata', 'vw_C%', 'Mary')
```

結果セット

View_Qualifier	View_owner	User_name	View_name	Rights
Demodata	Null	Mary	vw_Class	ALTER
Demodata	Null	Mary	vw_Class	SELECT

エラー状況

条件	エラー メッセージ
<i>database_qualifier</i> が空文字列である	有効なデータベース名を入力してください。 データベース名は空文字列にできません。
<i>view_name</i> が空文字列である	ビュー名は空文字列にできません。
<i>user_name</i> が空文字列である	ユーザー名は空文字列にできません。
メタデータ バージョン 1 を使用するデータベースで <i>psp_procedure_rights</i> を呼び出す	メタデータ バージョン 1 では、ビューおよびストアド プロシージャの権限はサポートされていません。

psp_views

現在のデータベースまたは指定されたデータベースから、ビューの一覧とそれに関連する情報を返します。

構文

```
call psp_views(['database_qualifier'], ['view_name'])
```

引数

パラメーター	データ型	デフォルト値	説明
database_qualifier	VARCHAR(20)	現在のデータベース	詳細を取得するデータベースの名前
view_name	VARCHAR(255)	(デフォルト値なし)	情報を要求するビューの名前。パターン検索がサポートされます。

返される結果セット

列名	データ型	説明
VIEW_QUALIFIER	VARCHAR(20)	データベースの名前
VIEW_OWNER	VARCHAR(20)	ビューの所有者の名前
VIEW_NAME	VARCHAR(255)	ビューの名前
REMARKS	VARCHAR(255)	備考
TRUSTEE	INTEGER	メタデータバージョン 2 の場合、信頼されるビューでは 0、信頼されないビューでは -1 を返します。メタデータバージョン 1 の場合、TRUSTEE 列は空になります。

例

メタデータバージョン 2 の Demodata2 というデータベースに次のようなビューが存在するとします。

```
CREATE VIEW vw_Person (lastn,firstn,phone) WITH EXECUTE AS 'Master' AS  
SELECT Last_Name, First_Name,Phone FROM Person;
```

次のステートメントは、現在のデータベースである Demodata2 内のビューの一覧を出力します。

```
call psp_views(, ,);
```

結果セット

View_Qualifier	View_Owner	View_Name	Remarks	Trustee
Demodata	Null	Vw_Person	Null	0

次のステートメントは、Demodata データベース内にあるビュー "vw_Person" のビュー情報を出力します。

```
call psp_views('demodata', 'vw_P%')
```

結果セット

View_Qualifier	View_Owner	View_Name	Remarks	Trustee
Demodata	Null	Vw_Person	Null	

エラー状況

条件	エラー メッセージ
<i>database_qualifier</i> が空文字列である	有効なデータベース名を入力してください。データベース名は空文字列にできません。
<i>view_name</i> が空文字列である	テーブル名は空文字列にできません。

パフォーマンスの最適化リファレンス

Zen データベース エンジンでは、多数の最適化を使用します。以下のトピックでは、SQL ステートメントでこれを活用する方法について説明します。この技術的題材は熟練した SQL ユーザー向けです。使用されている用語の定義については、[用語](#)を参照してください。

- [制限の分析](#)
- [制限の最適化](#)
- [プッシュダウン フィルター](#)
- [インデックスの有効な使用法](#)
- [テンポラリ テーブルのパフォーマンス](#)
- [行の事前取得](#)
- [用語](#)

データベースのパフォーマンスの向上に関する詳細については、『*Advanced Operations Guide*』の[パフォーマンス](#)を参照してください。

制限の分析

このトピックでは、データベース エンジンが**制限**を分析し、最適化するために使用する 1 方法について説明します。使用されている技術用語の定義については、**用語**を参照してください。

変形 CNF 変換

SQL ステートメントの実行中に、データベース エンジンは制限を**変形論理積標準形 (変形 CNF)**に変換しようとします。変形 CNF への変換は、論理式を同一構造に置いて、クエリ処理の最適化のための制限の分析を容易にします。制限を変形 CNF に変換できる場合、データベース エンジンはクエリを順序立てて完全に分析し、利用可能な**インデックス**を有効に利用できるように最適化します。データベース エンジンが制限を変形 CNF に変換できない場合でも、制限を分析してできる限りの最適化を行います。ただし、この場合データベース エンジンは、既に変形 CNF にある制限や内部的に変形 CNF に変換することができる制限のように、利用可能なインデックスを有効利用できないことがしばしばあります。

変換できない制限

制限の内容によっては、データベース エンジンが変形 CNF に変換できない制限があります。次の項目のいずれかが真になる場合、制限は変形 CNF に変換されません。

- 制限にサブクエリが含まれている。
- 制限に NOT 演算子が含まれている。
- 制限に動的パラメーターが含まれている（動的パラメーターは SQL ステートメント内の疑問符（"?"）で、ステートメントが実行されるときに入力要求进行される）。

変換が行われない条件

データベース エンジンが制限を変形 CNF に変換できるのにそうしない場合がいくつかあります。変形 CNF 変換後に最適化を適用するよりも元の形式に適用した方が最適化による利益をより多く受けられると判断した場合、データベース エンジンは制限を変形 CNF に変換しないという選択をします。

次の項目のいずれかが真になる場合、制限は変形 CNF に変換されません。

-
- 制限が**論理和標準形 (DNF)** になっており、すべての**述部**が等号 (=)、LIKE または IN 比較演算子のみを含む。

たとえば、データベース エンジンでは次の制限を変形 CNF に変換しません。

```
(c1 = 1 AND c2 = 1) OR (c1 = 1 AND c2 = 2) OR (c1 = 2)
```

- 制限が次のすべての条件に合致する。
 - 論理和標準形 (DNF)** の式を含み、制限の残りの部分と AND で結合されている。
 - 指定した DNF 式は、**述部**のみで、等号 (=)、LIKE または IN 比較演算子を含む。
 - DNF 式内の各**論理積**で同じ位置にある述部が同一列を参照する。

たとえば、次の**式**を含む**制限**は変形 CNF に変換されません。

```
(c1 = 1 AND c2 = 1) OR (c1 = 1 AND c2 = 2)
```

制限の最適化

ここでは、制限内の式を利用するためにデータベース エンジンが使用する基本技術を詳細に説明します。データベース エンジンが実行する最適化のタイプを単純なものから複雑なもの順に下に示します。

データベース エンジンが使用する最適化技術を完全に理解することは、データベース エンジンでの最適なパフォーマンスを実現するクエリを構築するのに役立ちます。さらに、データベース エンジンがクエリを最適化するためにインデックスをどのように使用するかを理解することにより、一般的な使用で最高のパフォーマンスを提供するインデックスの構築方法を決定することができます。

単純化のために、下の説明ではまず 1 つのテーブルの列を参照する式について述べます。述部で 2 つの異なるテーブルの列を比較する結合条件を使用する最適化は単一テーブルの最適化の次に説明します。

使用されている技術用語の定義については、[用語](#)を参照してください。

単一述部の最適化

制限の最適化の最も単純な形式は単一の**述部**を使用するものです。述部は、次の条件をすべて満たす場合、最適化に使用されます。

- 述部は AND 演算子で制限の残りの部分と結合されている。
- 述部の一方のオペランドは先頭セグメントである列の参照で構成され、もう一方のオペランドは列の参照を含まない式で構成されている（つまりリテラル値または直接パラメーターしか含まない）。
- 比較演算子は <、<=、=、>=、>、LIKE、または IN のいずれかである。

たとえば、最初のセグメントが列 `c1` であるインデックスがあるとします。次の述部は最適化に使用されます。

```
c1 = 1  
c1 IN (1,2)  
c1 > 1
```

LIKE 演算子は、2 番目のオペランドがワイルドカード以外の文字で始まる場合のみ最適化されます。たとえば、`C2 LIKE 'ABC%'` は最適化できますが、`C2 LIKE '%ABC'` は最適化されません。

閉じた範囲の最適化

閉じた範囲は、**単一述部の最適化**の条件をすべて満たす場合、最適化に使用されます。

たとえば、最初のセグメントが列 **c1** であるインデックスがあるとしします。次の閉じた範囲は最適化に使用されます。

```
c1 >= 1 AND c1 < 10
```

変形論理和の最適化

変形論理和は、次の条件にすべて合致する場合、最適化に使用されます。

- **制限**の残りの部分と **AND** 演算子で結合されている。
- 論理和内のそれぞれの**述部**および**閉じた範囲**は**単一述部の最適化**および**閉じた範囲の最適化**の要件を満たしている。
- 述部または閉じた範囲がそれぞれもう一方と同じ列を参照している。

たとえば、最初のセグメントが列 **c1** であるインデックスがあるとしします。次の変形論理和は最適化に使用されます。

```
c1 = 1 OR (c1 > 5 AND c1 < 10) OR c1 > 20
```

次の変形論理和は、すべての述部と閉じた範囲内で同一列が参照されていないため、このタイプの最適化には使用されません。

```
c1 = 1 OR (c1 > 5 AND c1 < 10) OR c2 = 1
```

論理積の最適化

論理積は、次の条件にすべて合致する場合、最適化に使用されます。

- **制限**の残りの部分と **AND** 演算子で結合されている。
- 論理積内のそれぞれの**述部**は、**単一述部の最適化**の要件を満たしている。
- それぞれの述部はインデックスの先頭セグメントで最適化を行っており、セグメントごとに1つの述部のみを使用している（つまり、同じ先頭セグメントのセットを使用する2つの異なる述部はない）。
- 最適化に使用される最後のセグメントを参照する述部以外のすべての述部は、等号(=) 比較演算子を使用している。

たとえば、最初の3つのセグメントとして列 **c1**、**c2**、**c3** をこの順序で持つインデックスがあるとしします。次の論理積の配置は最適化に使用されます。

```
c1 = 1 AND c2 = 1 AND c3 = 1  
c1 = 1 AND c2 = 1 AND c3 >= 1  
c1 = 1 AND c2 > 1
```

述部の順序は関係ありません。たとえば、次の論理積は最適化に使用されます。

```
c2 = 1 AND c3 = 1 AND c1 = 1
```

次の論理積は、インデックスの2番目のセグメントが飛ばされているため（列 c2 への参照がない）、最適化には使用されません。

```
c1 = 1 AND c3 = 1
```

この場合、単一の述部 `c1 = 1` は最適化に使用されます。

論理和標準形の最適化

論理和標準形（DNF）の式は、次の条件をすべて満たす場合、最適化に使用されます。

- 制限の残りの部分と AND 演算子で結合されている。
- 式のそれぞれの論理積は、**論理積の最適化**の要件を満たし、さらにすべての述部が等号（=）比較演算子を含む必要がある。
- すべての論理積は、最適化に、同じインデックスと同じ数のセグメントを使用する必要がある。

データベースエンジンは、元から DNF にあった制限を変形 CNF に変換しません。これは DNF 上で最適化されるためです。

たとえば、最初の3つのセグメントとして列 c1、c2、c3 をこの順序で持つインデックスがあるとします。DNF の次の式は最適化に使用されます。

```
(c1 = 1 AND c2 = 1 AND c3 = 1) OR (c1 = 1 AND c2 = 1 AND c3 = 2) OR (c1 = 2 AND c2 = 2 AND c3 = 2)
```

DNF の次の式は、両方の論理積が同数のセグメントを参照していないため、最適化に使用されません。

```
(c1 = 1 AND c2 = 1 AND c3 = 1) OR (c1 = 1 AND c2 = 2)
```

変形論理積標準形の最適化

変形論理積標準形（変形 CNF）の式は、次の条件をすべて満たす場合、最適化に使用されます。

- 制限の残りの部分と AND 演算子で結合されている。

- それぞれの**変形論理和**は**変形論理和の最適化**の要件を満たしている。ただし、それぞれの**変形論理和**は、**先頭セグメント**を共に構成するインデックス セグメントの別のセグメントを参照している必要がある（つまり、すべての論理和をまとめると、どのセグメントも抜けていない）
- すべての**変形論理和**は、最後のセグメントを参照するものを除き、等号 (=) 比較演算子を使用する述部を少なくとも 1 つ含む。

変形 CNF 最適化は DNF 最適化と似ていますが、DNF 最適化ではサポートされない、異なる比較演算子を含む述部の組み合わせを使用することができます。

たとえば、最初の 3 つのセグメントとして列 c1、c2、c3 をこの順序で持つインデックスがあるとします。変形 CNF の次の式は最適化に使用されます。

```
(c1 = 1 OR c1 = 2) AND (c2 = 1 OR (c2 > 2 AND c2 < 5)) AND (c3 > 1)
```

変形 DNF 内の等号式に注目すると、データベース エンジンがこの式を最適化に使用する方法を簡単に理解できます。

```
(c1 = 1 AND c2 = 1 AND c3 > 1) OR (c1 = 1 AND (c2 > 2 AND c2 < 5) AND c3 > 1) OR (c1 = 2 AND c2 = 1 AND c3 > 1) OR (c1 = 2 AND (c2 > 2 AND c2 < 5) AND c3 > 1)
```

変形 CNF 最適化で開いた範囲を閉じる

2 つの**変形論理和**は、次の条件を満たす場合、結合されて 1 つまたは複数の**閉じた範囲**を形成します。

- 変形論理和は**変形論理和の最適化**の要件を満たしている。
- 両方の**変形論理和**が同一インデックスの同一セグメントを使用している。
- どちらの**変形論理和**も開いた範囲を含んでおり、それらを結合して 1 つまたは複数の閉じた範囲を形成できる。

たとえば、最初のセグメントが列 c1 であるインデックスがあるとします。変形 CNF の次の式は最適化に使用されます。

```
(c1 = 1 OR c1 > 2) AND (c1 < 5 OR c1 = 10)
```

単純な**変形論理和**である等号式に注目すると、データベース エンジンがこの式を最適化に使用する方法が簡単に理解できます。

```
c1 = 1 OR (c1 > 2 AND c1 < 5) OR c1 = 10
```

単一結合条件の最適化

2つのテーブルにかかわる最適化の最も単純な形式は、単一の**結合条件**を利用するものです。単一の結合条件の最適化は**単一述部の最適化**に似ています。結合条件は、単一述部の最適化の条件をすべて満たす場合、最適化に使用されます。結合条件を使用して最適化されるテーブルは、結合条件内で参照されるほかのテーブルの後で処理されます。結合条件を使用して最適化されるテーブルは、結合条件内で参照されるほかのテーブルの行から取得した最適化値を使用します。

たとえば、テーブル t1 に最初のセグメントが列 c1 であるインデックスがあるとしします。次の結合条件は最適化に使用されます。

```
t1.c1 = t2.c2  
t1.c1 > t2.c2
```

最適化中、行はテーブル t2 から取得されます。この行から、列 c2 の値を使用して、結合条件に従ってテーブル t1 を最適化します。

t1.c1 のインデックスの代わりに t2.c2 のインデックスがある場合には、**t1.c1=t2.c2** によってテーブル t2 を最適化できます。この場合、テーブル t1 が最初に処理され、t1.c1 の値を基に、結合条件に従ってテーブル t2 が最適化されます。

t1.c1 のインデックスだけでなく t2.c2 のインデックスも存在する場合には、データベースエンジンのクエリ オプティマイザーは両方のテーブルのサイズに加え、2つのインデックスの特性も調べて、全体で最高のクエリ パフォーマンスを提供するテーブルを選択します。

結合条件を持つ論理積の最適化

結合条件とほかの**述部**の混成である**論理積**は、次の条件をすべて満たす場合、最適化に使用されます。

- すべての結合条件が同じ 2 つのテーブルの列を比較する。
- 論理積は、2 つのテーブルのいずれかの通常の**論理積の最適化**の要件を満たしている。

論理積を使用して最適化されるテーブルは、結合条件内で参照されるほかのテーブルの後で処理されます。

たとえば、テーブル t1 に最初の 3 つのセグメントとして列 c1、c2、c3 をこの順序で持つインデックスがあるとしします。次の論理積は最適化に使用されます。

```
t1.c1 = t2.c1 AND t1.c2 = t2.c2 AND t1.c3 = t2.c3
```

```
t1.c1 = t2.c1 AND t1.c2 > t2.c2
t1.c1 = t2.c1 AND t1.c2 = 1
t1.c1 = 1 AND t1.c2 = t2.c2
```

結合条件を持つ変形論理積標準形の最適化

結合条件を含む**変形論理積標準形（変形 CNF）**の式は、次の条件をすべて満たす場合、最適化に使用されます。

- **変形論理積標準形の最適化**の条件を満たしている。
- さらに、使用されている先頭セグメントの最後の部分で最適化する論理和以外のすべての論理和は、単一の結合条件または単一の述部のみを含み、少なくともその中の 1 つは単一結合条件である必要がある。

たとえば、テーブル t1 に最初の 3 つのセグメントとして列 c1、c2、c3 をこの順序で持つインデックスがあるとします。変形 CNF の次の式は最適化に使用されます。

```
(t1.c1 = t2.c1) AND (t1.c2 = t2.c2 OR
(t1.c2 > 2 AND t1.c2 < 5))

(t1.c1 = 1) AND (t1.c2 = t2.c2) AND
(t1.c3 > 2 AND t1.c3 < 5)
```

変形 CNF 最適化で結合条件の開いた範囲を閉じる

このタイプの最適化は、閉じられる範囲が**結合条件**であることを除けば、まさに**変形 CNF 最適化で開いた範囲を閉じる**とよく似ています。

たとえば、テーブル t1 に最初の 2 つのセグメントとして列 c1、c2 をこの順序で持つインデックスがあるとします。変形 CNF の次の式は最適化に使用されます。

```
(t1.c1 > t2.c1) AND (t1.c1 < t2.c2 OR t1.c1 = 10)

(t1.c1 = t2.c1) AND (t1.c2 > t2.c2) AND (t1.c2 < 10 OR t1.c2 = 100)
```

複数インデックスの変形論理和の最適化

変形論理和は、次の条件にすべて合致する場合、複数のインデックスを使用して最適化に使用されます。

- 制限の残りの部分と AND 演算子で結合されている。
- 論理和内のそれぞれの**述部**および**閉じた範囲**は、**単一述部の最適化**または**閉じた範囲の最適化**の要件をそれぞれ満たしている。

-
- それぞれの述部または閉じた範囲は、インデックスの最初のセグメントである列を参照する。すべての述部と閉じた範囲が同一列を参照する場合、このシナリオは単に、前述の**変形論理和の最適化**のとおりとなる。

たとえば、最初のセグメントが列 **c1** であるインデックスと、最初のセグメントが列 **c2** であるもう 1 つのインデックスがあるとします。次の変形論理和は最適化に使用されます。

```
c1 = 1 OR (c1 > 5 AND c1 < 10) OR c2 = 1
```

プッシュダウン フィルター

プッシュダウン フィルターは厳密には内部的な最適化技術です。高速なフィルター処理能力を利用することにより、データベース エンジンが、制限の特性に応じて結果セットから拒否する特定の行を効率的に識別することができます。結果セットから行を排斥してから結果セットを返すため、データベース エンジンが分析する必要のある行数が減り、プッシュダウン フィルターを使用しない場合よりも速く処理を完了します。

データベース エンジンが、次の条件が満たされる場合、式または式の組み合わせをプッシュダウン フィルターとして使用できます。

- **述部**は、**制限**の残りの部分と AND 演算子で結合されている場合、プッシュダウン フィルターで使用する。
- **述部**は、1 つのオペランドが列参照から成り、ほかのオペランドがリテラル値または動的パラメーター ("?") から成る場合、プッシュダウン フィルターで使用する。また、参照される列のデータ型は、bit、float、double、real、longvarchar、longvarbinary または binary のいずれでもない必要がある。
- **述部**は、比較演算子が <、<=、=、>=、>、または <> のいずれかである場合、プッシュダウン フィルターで使用する。
- **論理和**は、制限の残りの部分と AND 演算子で結合されており、論理和内のすべての述部がプッシュダウン フィルターで使用する述部の要件を満たしている（ただし、述部が制限の残りの部分と AND 演算子で結合されている必要があるという要件を除く）場合、プッシュダウン フィルターで使用する。プッシュダウン フィルターに含めることができる論理和は 1 つだけである。
- プッシュダウン フィルターは単一の論理和およびプッシュダウン フィルターで使用する述部の要件を満たす述部を組み合わせることができる。

使用されている技術用語の定義については、**用語**を参照してください。

インデックスの有効な使用法

インデックスは、**制限**以外の **DISTINCT** や **ORDER BY** 句などのクエリ特性を最適化することができます。

使用されている技術用語の定義については、**用語**を参照してください。

集計関数の **DISTINCT**

DISTINCT キーワード付きの**集計関数**から成る選択リストが指定されているクエリでは、インデックスの使用によって取得する行数を減らすことができます。このタイプの最適化の対象となるには、**DISTINCT** キーワードが作用する式は単一の列参照で構成されている必要があります。さらに、その列はインデックスの先頭セグメントである必要があります。

たとえば、テーブル **t1** に最初のセグメントが列 **c1** であるインデックスがあるとしします。インデックスを使用すると、列 **c1** の値が重複している行の取得を回避することができます。

```
SELECT COUNT(DISTINCT c1) FROM t1 WHERE c2 = 1
```

選択リストに先立つ **DISTINCT**

選択リストの前に **DISTINCT** キーワードが指定されているクエリでは、インデックスの使用によって取得する行数を減らすことができます。このタイプの最適化の対象となるには、選択リストは列参照のみから成り（算術式やスカラー関数のような複合式を含まない）、参照列は単一インデックスの先頭セグメントである必要があります。

たとえば、テーブル **t1** に最初の 3 つのセグメントとして列 **c1**、**c2**、**c3** を任意の順序で持つインデックスがあるとしします。インデックスを使用すると、選択リストの各項目の値が重複している行の取得を回避することができます。

```
SELECT DISTINCT c1, c2, c3 FROM t1 WHERE c2 = 1
```

インデックスのセグメント順序

ORDER BY 句で最適化のためにインデックスが使用されるかどうかは、インデックスのセグメントとして列がどのような順序で現れるかによります。具体的には、このタイプの最適化の対象となるには、**ORDER BY** 句内の列はインデックスの先頭セグメント

である必要があり、インデックス内でセグメントとして現れる順序と同じ順序で **ORDER BY** 句に現れる必要があります。

これとは対照的に、選択リストや **GROUP BY** 句の場合は、インデックスの先頭セグメントの列で構成されてさえいれば、選択リストの前にある **DISTINCT** または **GROUP BY** 句の最適化にインデックスを使用できます。このステートメントは、インデックス内のセグメントとして列が現れる順序にかかわらず真です。

たとえば、テーブル **t1** に最初の 3 つのセグメントとして列 **c1**、**c2**、**c3** を任意の順序で持つインデックスがあるとします。次のクエリでは、**DISTINCT** で最適化を行うためにインデックスが使用されます。

```
SELECT DISTINCT c1, c2, c3 FROM t1
SELECT DISTINCT c2, c3, c1 FROM t1 WHERE c3 > 1
```

次のクエリでは、**GROUP BY** で最適化を行うためにインデックスが使用されます。

```
SELECT c1, c2, c3, count(*) FROM t1 GROUP BY c2, c1, c3
SELECT c2, c3, c1, count(*) FROM t1 GROUP BY c3, c2, c1
```

ただし、**ORDER BY** での最適化に使用されるインデックスは、**c2**、**c1**、**c3** の順序である必要があります。

```
SELECT c1, c2, c3 FROM t1 ORDER BY c2, c1, c3
```

セグメントの昇順属性

ORDER BY 句で最適化にインデックスが使用されるかどうかは、いくつかの条件によって決まります。

具体的には、インデックスは次の条件をすべて満たす場合、最適化に使用されます。

- **ORDER BY** 句で列名に続いて **DESC** キーワードがある。
- 対応するインデックス セグメントが降順で定義されている。
- 指定した列がヌル可能列ではない。

さらに、インデックスは次の逆の条件をすべて満たす場合、最適化に使用されます（ヌル可能列は昇順の **ORDER BY** では使用できません）。

- **ORDER BY** ステートメントで列名の後に **ASC** キーワードまたは **ASC** でも **DESC** でもないものが続く。
- 対応するインデックス セグメントが昇順で定義されている。

同様に、関連するすべてのセグメントの昇順または降順属性が **ORDER BY** 句で指定された **ASC** または **DESC** キーワードと完全に反する場合、**ORDER BY** の最適化にインデックスが使用されます。さらに、降順に定義されたセグメントは、関連する列がヌル可能列でない場合にのみ使用されます。

インデックスは、いずれかのセグメントの属性が昇順であるか降順であるかに関係なく、制限の最適化、**DISTINCT** の最適化または **GROUP BY** 句の最適化に使用されます。

たとえば、テーブル **t1** に最初の 2 つのセグメントとして列 **c1**、**c2** をこの順序で持つインデックスがあり、セグメントは 2 つとも昇順であるとします。このインデックスは、次のクエリで最適化を行うために使用することができます。

```
SELECT c1, c2, c3 FROM t1 ORDER BY c1, c2
SELECT c1, c2, c3 FROM t1 ORDER BY c1 DESC, c2 DESC
SELECT DISTINCT c1, c2 FROM t1
SELECT DISTINCT c2, c1 FROM t1
SELECT * FROM t1 WHERE c1 = 1
```

テーブル **t1** に最初の 2 つのセグメントとして列 **c1**、**c2** をこの順序で持つインデックスがあり、**c1** のセグメントは昇順、**c2** のセグメントは降順に定義されているとします。また **c2** はヌル可能列であるとします。2 番目のセグメントは降順かつヌル可能列であるため、**ORDER BY** の最適化には使用できません。このインデックスは、次のクエリで最適化を行うために使用することができます。

```
SELECT c1, c2, c3 FROM t1 ORDER BY c1
SELECT c1, c2, c3 FROM t1 ORDER BY c1 DESC
SELECT DISTINCT c1, c2 FROM t1
SELECT DISTINCT c2, c1 FROM t1
SELECT * FROM t1 WHERE c1 = 1
```

c2 列がヌル可能列でない場合、インデックスは次のクエリで最適化を行うために使用することができます。

```
SELECT c1, c2, c3 FROM t1 ORDER BY c1, c2 DESC
SELECT c1, c2, c3 FROM t1 ORDER BY c1 DESC, c2
```

検索更新の最適化

インデックスの先頭セグメントを更新するとき、検索のための **WHERE** 句で同じインデックスを使うことによって、検索の最適化を利用することができます。データベースエンジンは、**UPDATE** 用のセッション（クライアント ID）と、もう 1 つ検索用のセッションを使用します。

次のステートメントは、検索の最適化によって利益を得られます。

```
CREATE TABLE t1 (c1 INT)
CREATE INDEX t1_c1 ON t1(c1)
INSERT INTO t1 VALUES(1)
```

```
INSERT INTO t1 VALUES(1)
INSERT INTO t1 VALUES(9)
INSERT INTO t1 VALUES(10)
INSERT INTO t1 VALUES(10)
UPDATE t1 SET c1 = 2 WHERE c1 = 10
UPDATE t1 SET c1 = c1 + 1 WHERE c1 >= 1
```

テンポラリ テーブルのパフォーマンス

パフォーマンスの改善は、このリリースでテンポラリ ソート テーブルの実装に対しても行われました。クエリを処理するには、データベース エンジンが内部的な使用目的でテンポラリ テーブルを生成する必要があります。これらのクエリの多くに対するパフォーマンスが改善されました。

一般的に、次の条件のいずれかが真の場合、データベース エンジンが少なくとも 1 つのテンポラリ テーブルを生成してクエリを処理します。

- 選択リストの前に **DISTINCT** キーワードがあり、選択リストの項目がインデックスの先頭セグメントの列ではない。

たとえば次の場合、列 **c1** および **c2** を先頭セグメントとするインデックスが存在しなければ、クエリを処理するためにテンポラリ テーブルが生成されます。

```
SELECT DISTINCT c1, c2 FROM t1
```

- **GROUP BY** 句が使用されており、**GROUP BY** 句内の項目がインデックスの先頭セグメントの列ではない。

たとえば次の場合、列 **c1** および **c2** を先頭セグメントとするインデックスが存在しなければ、クエリを処理するためにテンポラリ テーブルが生成されます。

```
SELECT c1, c2, COUNT(*) FROM t1 GROUP BY c1, c2
```

- 静的カーソルが使用されている。

たとえば、アプリケーションが結果セットを作成する前に **SQL_CURSOR_TYPE** オプションと **SQL_CURSOR_STATIC** 値を指定して ODBC API の **SQLSetStmtOption** を呼び出す場合、テンポラリ テーブルが生成されます。

- 結果セットにブックマークが含まれる。

たとえば、結果セットを生成する前に **SQL_USE_BOOKMARKS** オプションと **SQL_UB_ON** 値を指定して ODBC API の **SQLSetStmtOption** を呼び出す場合です。

- クエリは、**IN** または **=ANY** キーワードの右側に非相関サブクエリを含む。

たとえば、次のような例です。

```
SELECT c1 FROM t1 WHERE c2 IN (SELECT c2 FROM t2)
```

行の事前取得

特定の状況下で、**SELECT** ステートメントを実行した場合、このリリースのデータベースエンジンはクライアントに対し、結果の行セットの最初の 2 行をあらかじめ取得しようとしています。この事前取得により、ゼロ行または 1 行の結果セットからデータを取得するパフォーマンスが著しく改善されます。

結果セットが複数の行から成り、最初のデータ取得操作で要求される行が結果セットの最初の行でない場合、たとえば最後の行である場合には、行の事前取得は著しく時間を無駄にします。このため、事前取得がもたらす利点が意味を成さないケースを回避する一方、利点をもたらすケースでのパフォーマンスの改善を目指すため、事前取得は最大 2 行に制限されています。

事前取得は、クライアント DSN 用の詳細な接続属性で「配列フェッチ」が有効になっている場合にのみ行われます（『*ODBC Guide*』の[詳細な接続属性](#)を参照してください）。配列フェッチは事前取得と似ていますが、配列フェッチは最初の明示的なデータ取得操作が実行されるまでは行われない点が異なります。この違いは、最初の明示的なデータ取得操作により、データベースエンジンが残りの結果セットを取得する方法を推定できるだけの情報が提供されるためです。たとえば、最初のデータ取得操作が ODBC API の **SQLFetch** 呼び出しであった場合、これにより、データベースエンジンは結果セット全体を一度に 1 レコードずつ前方へ取得していくことが確実に推定できます。このように推定できるのは、ODBC の仕様によると、**SQLFetch** を使用した場合は残りの結果セットも同様に **SQLFetch** で取得されることが必要となるためです。一方、**SQLExtendedFetch** が呼び出され、行セットのサイズが 1 より大きい場合には、クライアントは開発者が最適な行セット サイズを指定したものと推定し、配列フェッチによってその設定を無効にしません。

事前取得は、次の条件のすべてが満たされた場合に起こります。

- 配列フェッチが有効である。
- 結果セットは大きな可変長データを含まない。たとえば、選択リストに **LONGVARCHAR** または **LONGVARBINARY** 型の列を含まない。
- 結果セットにブックマークが含まれない。
たとえば、結果セットの生成前に **SQL_USE_BOOKMARKS** オプションおよび **SQL_UB_ON** 値を指定して ODBC API の **SQLSetStmtOption** が呼び出された場合、事前取得は行われません。
- 読み取り専用の並行カーソルが使用されている。

たとえば、結果セットの生成前に、SQL_CONCURRENCY オプションおよび SQL_CONCUR_READ_ONLY 以外の値を指定して ODBC API の SQLSetStmtOption が呼び出された場合、事前取得は行われません。デフォルトで、並行カーソルは読み取り専用です。

用語

このトピックでは、ここで述べた複雑な技術的題材についての理解を助けるための定義や例を示します。

集計関数

集計関数は、クエリの SELECT または HAVING 句の値のグループを使用して単一の値を生成します。集計関数には、COUNT、AVG、SUM、STDEV、MAX、MIN、および DISTINCT があります。

閉じた範囲

閉じた範囲とは、AND 演算子で結合された 1 組の**開いた範囲**です。開いた範囲は両方とも同じ列を参照し、一方は < または <= 比較演算子を含み、もう一方は >= または > 演算子を含む必要があります。BETWEEN 句も閉じた範囲を定義します。

たとえば、次の式は閉じた範囲です。

```
c1 > 1 AND c1 <= 10  
c1 BETWEEN 1 AND 10
```

論理積

論理積は、2 つ以上の**述部**が AND 演算子で結合された式です。たとえば、次の**制限**は論理積です。

```
c1=2 AND c2<5  
c1>2 AND c1<5 AND c2= 'abc'
```

論理積標準形 (CNF)

ある**式**が AND 演算子で結合された 2 つ以上の**論理和**を含む場合、それは論理積標準形です。たとえば、次の式は CNF です。

```
c1 = 2 AND c2 < 5  
(c1 = 2 OR c1 = 5) AND (c2 < 5 OR c2 > 20) AND (c3 = 'abc' OR c3 = 'efg')
```

論理和

論理和は、2 つ以上の**述部**が OR 演算子で結合された**式**です。たとえば、次の式は論理和です。

```
c1 = 2 OR c2 = 5  
c1 = 2 OR c1 > 5 OR c2 = 'abc'
```

論理和標準形 (DNF)

ある**式**が OR 演算子で結合された 2 つ以上の**論理積**を含む場合、それは論理和標準形です。たとえば、次の式は DNF です。

```
c1 = 2 OR c2 < 5  
(c1 = 2 AND c2 = 5) OR (c2 > 5 AND c2 < 10) OR c3 = 'abc'
```

式

式は、**制限**で利用できるブール代数で構成されます。少なくとも 1 つ以上の完全な**述部**を含む制限全体または制限の一部が式です。

インデックス

インデックスは、テーブル内の 1 つ以上の列に関連付けられた構造で、データベースエンジンが効率的に検索と並べ替えを実行できるようにするものです。インデックスを利用すると、データベースエンジンはテーブル内のすべての行を調べる代わりに検索条件を満たす特定の行のみを読み取ることができ、検索パフォーマンスを向上させることができます。また、インデックスを利用すれば、行を取得した後で効率の悪い技術を使って並べ替えしなくても、SQL クエリで指定した順序で行を取得できます。

結合条件

結合条件は <、<=、=、>=、> の比較演算子を使用して、あるテーブルの列を別のテーブルの列と比較する**述部**です。

たとえば、次の述部は結合条件です。

```
t1.c1 = t2.c1  
t1.c1 > t2.c2
```

先頭セグメント

インデックス セグメントのグループが、**インデックス**の最初の n 列 (n はインデックスの総セグメント数以下の任意の数値) で構成されている場合、このグループは先頭セグメントです。たとえば、インデックスが列 **c1**、**c2**、**c4** をセグメントとして定義されている場合、**c1** は先頭セグメントと呼ばれます。また、**c1** と **c2** を合わせても先頭セグメント、3 つすべて合わせても先頭セグメントと言いますが、**c2** 単独では先頭セグメントと言いません。これは、**c2** の前にあるセグメント **c1** が除外されているためです。列 **c1** と **c4** を合わせても先頭セグメントとは言いません。**c4** の前にある **c2** が除外されているためです。

変形論理積標準形 (変形 CNF)

変形論理積標準形内の**式**は**論理積標準形 (CNF)** 内の式と似ていますが、各**論理和**が**閉じた範囲**に加えて**述部**を含む点が異なります。

たとえば、次の式は変形 CNF です。

```
c1 = 2 AND c2 < 5
(c1 = 2 OR (c1 > 4 AND c1 < 6) OR c1 = 10) AND (c2 = 1 OR c3 = 'efg')
```

変形論理和

変形論理和は**論理和**と似ていますが、**閉じた範囲**に加えて**述部**を含む点が異なります。

たとえば、次の式は変形論理和です。

```
c1 = 2 OR (c1 > 4 AND c1 < 5)
(c1 = 2 OR (c1 > 4 AND c1 < 5)) OR c2 = 'abc'
```

開いた範囲

開いた範囲は、**<**、**<=**、**>=**、または**>**のいずれかの比較演算子を持つ述部です。さらに、述部のオペランドの 1 つは単一の列で構成され、もう一方のオペランドはほかのテーブルの単一の列またはリテラルで構成されている必要があります。

たとえば、次の式は開いた範囲です。

```
c1 > 1
c1 <= 10
t1.c1 > t2.c1
```

述部

述部は、AND または OR の論理演算子を含まない論理式です（BETWEEN は除く）。

たとえば、次の式は述部です。

```
(c1 = 1)
(c1 LIKE 'abc')
(c1 BETWEEN 1 AND 2)
```

次の例は述部では**ありません**。

```
(c1 > 1 AND c1 < 5)
(c1 = 1 OR c1 = 2)
```

制限

制限は SQL クエリの WHERE 句全体として定義されます。

システム カタログ関数

以下のトピックでは、メタデータを取得するためのシステム カタログ関数について説明します。

- [Zen システム カタログ関数](#)
- [dbo.fSQLColumns](#)
- [dbo.fSQLForeignKeys](#)
- [dbo.fSQLPrimaryKeys](#)
- [dbo.fSQLProcedures](#)
- [dbo.fSQLProcedureColumns](#)
- [dbo.fSQLSpecialColumns](#)
- [dbo.fSQLStatistics](#)
- [dbo.fSQLTables](#)
- [dbo.fSQLDBTableStat](#)
- [文字列検索パターン](#)

Zen システム カタログ関数

システム カタログ関数を使用すると、データ辞書ファイル（「カタログ」とも呼ばれます）からデータベース メタデータを取得することができます。システム カタログ関数は FROM 句でのみ使用できます。

Zen では、適切なカタログ API を呼び出したり、システム ストアド プロシージャ（[システム ストアド プロシージャ](#)を参照）を使用したりすることによって、メタデータを返すこともできます。しかし、これら 2 つの方法はメタデータをビューに格納しません。ビューでは、ほかのテーブルとの JOIN や UNION が行えます。ほかのテーブルとの JOIN および UNION 機能を提供するには、システム カタログ関数が必要です。

ADO.NET など一部のアクセス方法では、JOIN および UNION の機能を利用できるように、エンティティ サポートのためのシステム カタログ関数を必要とします。

各システム カタログ関数のテンポラリ ビュー スキーマは、準備フェーズで作成されます。実行フェーズで対応するカタログ API を呼び出すことにより、そのビューにデータが格納されます。

次の表は、利用可能なシステム カタログ関数を示しています。

Zen 関数 ¹	戻り値
<code>dbo.fSQLColumns</code>	現在のデータベースまたは指定されたデータベースの、特定テーブルの列の一覧とそれに関連する情報。
<code>dbo.fSQLForeignKeys</code>	現在のデータベース内の特定テーブルの外部キー情報。
<code>dbo.fSQLPrimaryKeys</code>	現在のデータベースまたは指定されたデータベースの、特定テーブルの主キー情報。
<code>dbo.fSQLProcedures</code>	現在のデータベースまたは指定されたデータベースの、ストアドプロシージャの名前。
<code>dbo.fSQLProcedureColumns</code>	入力パラメーターおよび出力パラメーターのリストと、指定されたプロシージャの結果セットを構成する列。
<code>dbo.fSQLSpecialColumns</code>	指定されたテーブル内の行を一意に識別する最適な列セットに関する情報、または、トランザクションによって行内の任意の値が更新されたときに自動的に更新される列に関する情報。
<code>dbo.fSQLStatistics</code>	現在のデータベースまたは指定されたデータベースの、ある単一テーブルに関する統計情報と、そのテーブルに関連付けられているインデックスの一覧。
<code>dbo.fSQLTables</code>	現在のデータベースまたは指定されたデータベースの、テーブルの一覧とそれに関連する情報。

¹ Zen のカタログ関数は ODBC に基づいているので、追加情報については ODBC のドキュメントを参照してください。ここに記述されている内容は、Zen カタログ関数を理解し使用するのに十分な情報を提供していますが、完全な技術的詳細は含まれていません。

リターン ステータス

各システム カタログ関数は、実行結果に応じて以下のステータス値のいずれかを返します。

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

-
- SQL_STILL_EXECUTING
 - SQL_ERROR
 - SQL_INVALID_HANDLE

まとめ

システム カタログ関数には以下の特性があります。

- メタデータを返す。
- ビューと同じように動作する。
- SELECT ステートメントの FROM 句でのみ参照できる。
- パラメーターは、定数または動的パラメーターの形式でのみ指定できる。

メモ： よく使われる SQL エディターのほとんどは、複数のステートメントを実行するためにステートメント区切り文字を使用しません。しかし、ZenCC の SQL Editor は区切り文字を必要とします。ほかの環境で例を実行する場合は、シャープ記号またはセミコロンの区切り文字を取り除く必要があります。

dbo.fsSQLColumns

この関数は、指定されたテーブルの列名の一覧を返します。

構文

```
dbo.fsSQLColumns <'database_qualifier' | null>, <'table_name' | null>, <'column_name' | null>)
```

引数

パラメーター	データ型	デフォルト値	説明
<i>database_qualifier</i>	VARCHAR	現在のデータベース	詳細を取得するデータベースの名前。
<i>table_name</i>	VARCHAR	(デフォルト値なし)	列情報を要求するテーブルの名前
<i>column_name</i>	VARCHAR	指定したテーブルのすべての列	指定したテーブルの列名。

返される結果セット

列名	データ型	説明
TABLE_QUALIFIER	VARCHAR	データベース名。データベースに適用されない場合は NULL。
TABLE_OWNER	VARCHAR	テーブルのスキーマ名。データベースに適用されない場合は NULL。
TABLE_NAME	NULL でない VARCHAR	テーブルの名前。
COLUMN_NAME	VARCHAR	テーブルの列名、または、名前のない列の場合は空文字列。
DATA_TYPE	NULL でない SMALLINT	列の SQL データ型コード。『 <i>ODBC Guide</i> 』の 使用できるデータ型 を参照してください。
TYPE_NAME	VARCHAR	列のデータ型の、DATA_TYPE 値に対応する名前

列名	データ型	説明
PRECISION	INTEGER	列の精度（列のデータ型が Decimal 、 Numeric などである場合）。 Decimal データ型の精度と小数位 を参照してください。DATA_TYPE が CHAR または VARCHAR である場合、この列には列の最大長（文字単位）が格納されます。これは、日付時刻データ型の場合は、値が文字列に変換されたときに、その値を表示するために必要な文字の総数になります。数値データ型の場合は、 RADIX 列に応じて、列で許可される桁数の合計またはビット数の合計のいずれかになります。
LENGTH	INTEGER	SQL_C_DEFAULT が指定された場合は、 SQLGetData 、 SQLFetch 、または SQLFetchScroll 操作で転送されるデータの長さ（バイト単位）。 数値データの場合、このサイズは、データベースに格納されているデータのサイズとは異なる場合があります。この値は、文字データの COLUMN_SIZE 列とは異なる場合があります。
SCALE	SMALLINT	小数点の右側にある有効桁の総数。 TIME 、 TIMESTAMP 、および TIMESTAMP2 の場合、この列には小数の秒部分の桁数が格納されます。 それ以外のデータ型では、これはデータベース内の列の小数点以下の桁数です。 Decimal データ型の精度と小数位 を参照してください。
RADIX	SMALLINT	数値データ型の基数 数値データ型の場合は、10 または 2 のいずれかです。 10 – COLUMN_SIZE および DECIMAL_DIGITS の値は、列で許可される小数点以下の桁数を示します。 2 – COLUMN_SIZE および DECIMAL_DIGITS の値は、列で許可されるビット数を示します。 RADIX が適用されないデータ型では、 NULL が返されます。

列名	データ型	説明
NULLABLE	SMALLINT	プロシージャ列がヌル値を受け入れるかどうかを示します。 0 = NO_NULLS – プロシージャ列はヌル値を受け入れません。 1 = NULLABLE – プロシージャ列はヌル値を受け入れます。 2 = NULLABLE_UNKNOWN – プロシージャ列がヌル値を受け入れるかどうかわかりません。
REMARKS	VARCHAR	備考フィールド
COLUMN_DEF	VARCHAR	列のデフォルト値。 デフォルト値として NULL が指定された場合、この列は、引用符で囲まれていない単語の NULL になります。デフォルト値を切り捨てなしで表すことができない場合は、一重引用符で囲まれていない TRUNCATED が格納されます。デフォルト値が指定されていない場合は、NULL になります。
SQL_DATA_TYPE	NULL でない SMALLINT	記述子の SQL_DESC_TYPE フィールドに表示される SQL データ型の値。この列は次のデータ型を除き、TYPE_NAME 列と同じです。例外とするのは AUTOTIMESTAMP、DATE、DATETIME、TIME、TIMESTAMP、および TIMESTAMP2 です。 例外とするデータ型について、結果セットの SQL_DATA_TYPE フィールドは次のように値を返します。DATE は SQL_DATE、TIME は SQL_TIME、AUTOTIMESTAMP、DATETIME、TIMESTAMP、および TIMESTAMP2 は SQL_TIMESTAMP です。
SQL_DATETIME_SUB	SMALLINT	AUTOTIMESTAMP、DATE、DATETIME、TIME、TIMESTAMP、および TIMESTAMP2 のサブタイプコード。その他のデータ型について、この列は NULL を返します。 1 = DATE (SQL_CODE_DATE) 2 = TIME (SQL_CODE_TIME) 3 = AUTOTIMESTAMP、DATETIME、TIMESTAMP、および TIMESTAMP2 (SQL_CODE_TIMESTAMP)

列名	データ型	説明
CHAR_OCTET_LENGTH	INTEGER	文字またはバイナリ データ型列の最大長（バイト単位）。その他すべてのデータ型について、この列は NULL を返します。
ORDINAL_POSITION	NULL でない INTEGER	入力および出力パラメーターについては、プロシージャ定義におけるパラメーターの位置を表す序数（1 から始まり、増加するパラメーター順）。 戻り値がある場合、戻り値については 0 が返されます。結果セット列については、結果セットにおける列の位置を表す序数。この場合、結果セット列の最初の列は 1 から始まります。
IS_NULLABLE	VARCHAR	列に NULL が含まれていない場合は "NO"。 列に NULL が含まれている場合は "YES"。 ヌル値を許可するかどうかわからない場合、この列は長さゼロの文字列を返します。この列に返される値は、 NULLABLE 列に返される値とは異なります。

例

次の例は、デフォルトの Demodata サンプル データベースに含まれる、Room テーブルのすべての列に関する情報を返します。

```
SELECT * FROM dbo.fsSQLColumns ('Demodata', 'room', null);
```

結果セット（スペースを考慮して短縮されています）：

COLUMN_NAME	DATA_TYPE	LENGTH	ORDINAL_POSITION
=====	=====	=====	=====
Building_Name	1	25	1
Number	4	4	2
Capacity	5	2	3
Type	1	20	4

4 行が影響を受けました。

dbo.fsSQLForeignKeys

この関数は、現在のデータベース内の特定テーブルの外部キー情報を返します。dbo.fsSQLForeignKeys は、外部キーの一覧を次のいずれかの結果セットとして返すことができます。

- 特定テーブル（他のテーブルの主キーを参照する特定テーブル内の列）

- 特定テーブルの主キーを参照する他のテーブル

構文

```
dbo.fSQLForeignKeys (<'table_qualifier' | null>, 'pkey_table_name' | null>, <'fkey_table_name' | null>)
```

引数

パラメーター	データ型	デフォルト値	説明
<i>table_qualifier</i>	VARCHAR	現在のデータベース	詳細を取得するデータベースの名前。
<i>pkey_table_name</i>	VARCHAR	(デフォルト値なし)	主キー列に外部キーが関連付けられているテーブルの名前。パターン検索がサポートされます (文字列検索パターン を参照してください)。
<i>fkey_table_name</i>	VARCHAR	(デフォルト値なし)	外部キー情報を取得する必要があるテーブルの名前。パターン検索がサポートされます (文字列検索パターン を参照してください)。

返される結果セット

列名	データ型	説明
PKTABLE_QUALIFIER	VARCHAR	主キー テーブルのデータベース名。データベースに適用されない場合は NULL。
PKTABLE_OWNER	VARCHAR	主キー テーブルのオーナーの名前。データベースに適用されない場合は NULL。
PKTABLE_NAME	NULL でない VARCHAR	主キー テーブルの名前
PKCOLUMN_NAME	NULL でない VARCHAR	主キー列の列名名前のない列については、空文字列が返されます。
FKTABLE_QUALIFIER	VARCHAR	外部キー テーブルのデータベース名。データベースに適用されない場合は NULL。
FKTABLE_OWNER	VARCHAR	外部キー テーブルのオーナーの名前。データベースに適用されない場合は NULL。
FKTABLE_NAME	NULL でない VARCHAR	外部キー テーブルの名前。
FKCOLUMN_NAME	NULL でない VARCHAR	外部キー列の列名名前のない列については、空文字列が返されます。
KEY_SEQ	SMALLINT	キーにおける列のシーケンス番号 (1 から始まります)。この列の値は、X\$Index の Xi\$Part に対応します。X\$Index を参照してください。
UPDATE_RULE	SMALLINT	SQL 操作が UPDATE であるとき、外部キーに適用される動作。次のいずれかの値になります。 • 0 = CASCADE • 1 = RESTRICT
DELETE_RULE	SMALLINT	SQL 操作が DELETE であるとき、外部キーに適用される動作。次のいずれかの値になります。 • 0 = CASCADE • 1 = RESTRICT
FK_NAME	VARCHAR	外部キーの名前。データベースに適用されない場合は NULL。
PK_NAME	VARCHAR	主キーの名前。データベースに適用されない場合は NULL。

列名	データ型	説明
DEFERRABILITY	SMALLINT	次のいずれかの値です。 • 5 = INITIALLY_DEFERRED • 6 = INITIALLY_IMMEDIATE • 7 = NOT_DEFERRABLE

例

次の例では、Demodata サンプル データベースに 3 つのテーブルを作成します。それらのテーブルに外部キーと主キーが割り当てられます。dbo.fSQLForeignKeys 関数は、文字列検索パターンを使って 2 つの主キー テーブルを参照します。[文字列検索パターン](#) も参照してください。

```
CREATE TABLE primarykey1 (pk1col1 INT, pk1col2 INT, pk1col3 INT, pk1col4 INT, PRIMARY KEY (pk1col1, pk1col2));
ALTER TABLE primarykey1 ADD FOREIGN KEY (pk1col3, pk1col4) REFERENCES primarykey1 ON DELETE CASCADE;

CREATE TABLE primarykey2 (pk2col1 INT, pk2col2 INT, pk2col3 INT, pk2col4 INT, PRIMARY KEY (pk2col1, pk2col2));
ALTER TABLE primarykey2 ADD FOREIGN KEY (pk2col3, pk2col4) REFERENCES primarykey2 ON DELETE CASCADE;

CREATE TABLE foreignkey1 (fkcol1 INT, fkcol2 INT, fkcol3 INT, fkcol4 INT);
ALTER TABLE foreignkey1 ADD FOREIGN KEY (fkcol1, fkcol2) REFERENCES PRIMARYKEY1;
ALTER TABLE foreignkey1 ADD FOREIGN KEY (fkcol3, fkcol4) REFERENCES PRIMARYKEY2;

SELECT * FROM dbo.fSQLForeignKeys ('Demodata', 'primarykey%', 'foreignkey1');
```

結果セット（スペースを考慮して短縮されています）：

FKCOLUMN_NAME	DELETE_RULE	FK_NAME	PK_NAME
fkcol1	1	FK_0fkcol1	PK_pk1col1
fkcol2	1	FK_0fkcol1	PK_pk1col1
fkcol3	1	FK_0fkcol3	PK_pk2col1
fkcol4	1	FK_0fkcol3	PK_pk2col1

4 行が影響を受けました。

dbo.fSQLPrimaryKeys

この関数は、テーブルの主キーを構成する列名を結果セットとして返します。dbo.fSQLPrimaryKeys では、1 回の呼び出しで複数のテーブルから主キーを返すことはできません。

構文

```
dbo.fsSQLPrimaryKeys (<'pkey_table_qualifier' | null>, <'table_name' | null>)
```

引数

パラメーター	データ型	デフォルト値	説明
<i>pkey_table_qualifier</i>	VARCHAR	現在のデータベース	詳細を取得するデータベースの名前。
<i>table_name</i>	VARCHAR	(デフォルト値なし)	主キー情報を要求するテーブルの名前。パターン検索がサポートされます (文字列検索パターン を参照してください)。

返される結果セット

列名	データ型	説明
TABLE_QUALIFIER	VARCHAR	データベース名。データベースに適用されない場合は NULL。
TABLE_OWNER	VARCHAR	主キー テーブルのオーナーの名前。データベースに適用されない場合は NULL。
TABLE_NAME	NULL でない VARCHAR	主キー テーブルの名前
COLUMN_NAME	NULL でない VARCHAR	主キー列の名前。名前のない列については、空文字列が返されます。
COLUMN_SEQ	NULL でない SMALLINT	キーにおける列のシーケンス番号 (1 から始まります)。
PK_NAME	VARCHAR	主キーの名前。データベースに適用されない場合は NULL。

例

次の例では、Demodata サンプル データベースに 2 つのテーブルを作成します。それらのテーブルに外部キーと主キーが割り当てられます。dbo.fsSQLPrimaryKeys 関数は、文字列検索パターンを使って 2 つのテーブルを参照します。[文字列検索パターン](#)も参照してください。

```
CREATE TABLE tblprimarykey3 ( tblpk3col1 INT, tblpk3col2 INT, tblpk3col3 INT, tblpk3col4 INT, PRIMARY KEY (tblpk3col1, tblpk3col2) );
ALTER TABLE tblprimarykey3 ADD FOREIGN KEY (tblpk3col3, tblpk3col4) REFERENCES tblprimarykey3 ON DELETE CASCADE;
```

```
CREATE TABLE tblprimarykey4 ( tblpk4col1 INT, tblpk4col2 INT, tblpk4col3 INT, tblpk4col4 INT, PRIMARY KEY (tblpk4col1, tblpk4col2) );
ALTER TABLE tblprimarykey4 ADD FOREIGN KEY (tblpk4col3, tblpk4col4) REFERENCES tblprimarykey4 ON DELETE CASCADE;
```

```
SELECT * FROM dbo.fsSQLprimarykeys('Demodata', 'tbl%');
```

結果セット（スペースを考慮して短縮されています）：

TABLE_NAME	COLUMN_NAME	KEY_SEQ	PK_NAME
tblprimarykey3	tblpk3col1	1	PK_tblpk3col1
tblprimarykey3	tblpk3col2	2	PK_tblpk3col1
tblprimarykey4	tblpk4col1	1	PK_tblpk4col1
tblprimarykey4	tblpk4col2	2	PK_tblpk4col1

4 行が影響を受けました。

dbo.fsSQLProcedures

この関数は、現在のデータベースまたは指定されたデータベースの、ストアド プロシージャおよびユーザー定義関数の名前を結果セットとして返します。[CREATE PROCEDURE](#) および [CREATE FUNCTION](#) も参照してください。

構文

```
dbo.fsSQLProcedures (<'database_qualifier' | null>, <'procedure_name' | null>)
```


引数

パラメーター	データ型	デフォルト 値	説明
<i>database_qualifier</i>	VARCHAR	現在のデータベース	詳細を取得するデータベースの名前。
<i>procedure_name</i>	VARCHAR	(デフォルト 値なし)	情報を要求するストアド プロシージャの名前。

メモ： システム ストアド プロシージャは、Zen Control Center に表示されない内部データベース PERVASIVESYSDB に定義されています。

返される結果セット

列名	データ型	説明
PROCEDURE_QUALIFIER	VARCHAR	プロシージャが作成されたデータベースの名前。データベースに適用されない場合は NULL。
PROCEDURE_OWNER	VARCHAR	プロシージャ スキーマ識別子。データベースに適用されない場合は NULL。
PROCEDURE_NAME	NULL でない VARCHAR	プロシージャ識別子
NUM_INPUT_PARAMS	なし	今後の使用に備えて予約されています。アプリケーションには使用しないでください。
NUM_OUTPUT_PARAMS	なし	今後の使用に備えて予約されています。アプリケーションには使用しないでください。
NUM_RESULT_SETS	なし	今後の使用に備えて予約されています。アプリケーションには使用しないでください。
REMARKS	VARCHAR	プロシージャの解説。
PROCEDURE_TYPE	SMALLINT	プロシージャの種類を定義します。 • 0 = PT_UNKNOWN - プロシージャが値を返すかどうかを判断できません。 • 1 = PT_PROCEDURE - 返されるオブジェクトはプロシージャで、戻り値はありません。 • 2 = PT_FUNCTION - 返されるオブジェクトは関数で、戻り値があります。

例

デフォルトで、Demodata データベースにはストアド プロシージャとユーザー定義関数何れも含まれていません。dbo.fSQLProcedures 関数（および [dbo.fSQLProcedureColumns](#)）の出力を提供するために、次のようなストアド プロシージャとユーザー定義関数を作成してください。それらはすべて、必要なテーブルおよびパラメーター バインドが与えられれば呼び出すことができます。しかし、この例の目的はそれらの実行を示すことではなく、それらをデータベース オブジェクトとして含めることです。

[CREATE PROCEDURE](#) および [CREATE FUNCTION](#) も参照してください。

```

CREATE PROCEDURE curs1 (IN :Arg1 CHAR(4), IN :Arg2 INTEGER) AS BEGIN
DECLARE :alpha char(10) DEFAULT 'BA';
DECLARE :beta INTEGER DEFAULT 100;
DECLARE degdel CURSOR FOR
SELECT degree, cost_per_credit FROM tuition WHERE Degree = :Arg1 AND cost_per_credit = :arg2
FOR UPDATE;
OPEN degdel;
FETCH NEXT FROM degdel INTO :alpha, :beta
DELETE WHERE CURRENT OF degdel;
CLOSE degdel;
END

CREATE PROCEDURE EnrollStudent2 (IN :Stud_id INTEGER, IN
:Class_Id INTEGER);
BEGIN
INSERT INTO Enrolls VALUES (:Stud_id, :Class_Id, 0.0);
END

CREATE PROCEDURE AInsert
(IN :AGUID BINARY(16),
IN :APeriod INT,
IN :BBal UTINYINT,
IN :BDr DECIMAL(23,9),
IN :BCr DECIMAL(23,9),
IN :BNet DECIMAL(23,9),
IN :HTrx UTINYINT,
IN :PDr DECIMAL(23,9),
IN :PCr DECIMAL(23,9),
IN :PNet DECIMAL(23,9))
AS BEGIN
INSERT INTO "ASum" ("AID", "APeriod", "IBal", "BDr", "BCr", "BNet", "HTrx", "PDr", "PCr", "PNet")
VALUES (:AGUID,:APeriod,:BBal,:BDr,:BCr,:BNet,:HTrx, :PDr,:PCr,:PNet);
END

CREATE PROCEDURE AR (OUT :BIID SMALLINT, IN :BName CHAR(30))
AS BEGIN
SELECT MAX(BID) + 1 INTO :BIID FROM Br;
INSERT INTO Br (BID, FName) VALUES (:BIID, :BName);
END

CREATE FUNCTION CalInterest (IN :principle FLOAT,
IN :period REAL, IN :rate DOUBLE)
RETURNS DOUBLE
AS BEGIN
DECLARE :interest DOUBLE;
SET :interest = ((:principle * :period * :rate) /
100);
RETURN (:interest);
END;

SELECT * FROM dbo.fsQLProcedures ('Demodata', null);

```

結果セット（スペースを考慮して短縮されています）:

PROCEDURE_QUALIFIER	PROCEDURE_NAME	PROCEDURE_TYPE
=====	=====	=====
Demodata	curs1	1
Demodata	Enrollstudent2	1
Demodata	AInsert	1
Demodata	AR	1
Demodata	CalInterest	2

5 行が影響を受けました。

dbo.fSQLProcedureColumns

この関数は、入力パラメーターおよび出力パラメーターのリストと、指定されたストアード プロシージャまたはユーザー定義関数の結果セットを構成する列を返します。

[CREATE PROCEDURE](#) および [CREATE FUNCTION](#) も参照してください。

構文

```
dbo.fSQLProcedureColumns (<'database_qualifier' | null>, <'procedure_name' | null>,  
<'procedure_column_name' | null>)
```

引数

パラメーター	データ型	デフォルト 値	説明
<i>database_qualifier</i>	VARCHAR	現在のデータベース	詳細を取得するデータベースの名前。
<i>procedure_name</i>	VARCHAR	(デフォルト 値なし)	情報を要求するストアード プロシージャの名前。
<i>procedure_column_name</i>	VARCHAR	(デフォルト 値なし)	プロシージャの列の名前。

メモ： システム ストアド プロシージャは、Zen Control Center に表示されない内部データベース PERVASIVESYSDB に定義されています。

返される結果セット

列名	データ型	説明
PROCEDURE_QUALIFIER	VARCHAR	プロシージャが作成されたデータベースの名前。データベースに適用されない場合は NULL。
PROCEDURE_OWNER	VARCHAR	プロシージャ スキーマ識別子。データベースに適用されない場合は NULL。
PROCEDURE_NAME	NULL でない VARCHAR	プロシージャ識別子
COLUMN_TYPE	NULL でない SMALLINT	プロシージャ列をパラメーターまたは結果セット列として定義します。 • 0 = PARAM_TYPE_UNKNOWN - プロシージャ列は種類が不明なパラメーターです。 • 1 = PARAM_INPUT - プロシージャ列は入力パラメーターです。 • 2 = PARAM_INPUT_OUTPUT - プロシージャ列は入力 / 出力パラメーターです。 • 3 = RESULT_COL - プロシージャ列は結果セット列です。 • 4 = PARAM_OUTPUT - プロシージャ列は出力パラメーターです。 • 5 = RETURN_VALUE - プロシージャ列はプロシージャの戻り値です。
DATA_TYPE	NULL でない SMALLINT	SQL データ型。『 <i>ODBC Guide</i> 』の 使用できるデータ型 も参照してください。
TYPE_NAME	NULL でない VARCHAR	リレーショナル データ型名。 Zen で使用できるデータ型 も参照してください。
PRECISION	INTEGER	データベース内のプロシージャ列のサイズ。列サイズが適用されないデータ型では、NULL が返されます。 Decimal データ型の精度と小数位 も参照してください。

列名	データ型	説明
LENGTH	INTEGER	SQL_C_DEFAULT が指定された場合は、SQLGetData または SQLFetch 操作で転送されるデータの長さ（バイト単位）。数値データの場合、このサイズは、データベースに格納されているデータのサイズとは異なる場合があります。Zen で使用できるデータ型も参照してください。
SCALE	SMALLINT	データベース内のプロシージャ列の小数点以下の桁数。小数点以下の桁数が適用されないデータ型では、NULL が返されます。Decimal データ型の精度と小数位も参照してください。
RADIX	SMALLINT	数値データ型の場合は、10 または 2 のいずれかです。 10 - COLUMN_SIZE および DECIMAL_DIGITS の値は、列で許可される小数点以下の桁数を示します。 2 - COLUMN_SIZE および DECIMAL_DIGITS の値は、列で許可されるビット数を示します。 RADIX が適用されないデータ型では、NULL が返されます。
NULLABLE	NULL ではない SMALLINT	プロシージャ列がヌル値を受け入れるかどうかを示します。 0 = NO_NULLS - プロシージャ列はヌル値を受け入れません。 1 = NULLABLE - プロシージャ列はヌル値を受け入れます。 2 = NULLABLE_UNKNOWN - プロシージャ列がヌル値を受け入れるかどうかわかりません。
REMARKS	VARCHAR	プロシージャ列の解説。

列名	データ型	説明
COLUMN_DEF	VARCHAR	列のデフォルト値。 デフォルト値として NULL が指定された場合、この列は、引用符で囲まれていない単語の NULL になります。デフォルト値を切り捨てなしで表すことができない場合は、一重引用符で囲まれていない TRUNCATED が格納されます。デフォルト値が指定されていない場合は、NULL になります。
SQL_DATA_TYPE	NULL でない SMALLINT	記述子の SQL_DESC_TYPE フィールドに表示される SQL データ型の値。この列は次のデータ型を除き、TYPE_NAME 列と同じです。例外とするのは AUTOTIMESTAMP、DATE、DATETIME、TIME、TIMESTAMP、および TIMESTAMP2 です。 例外とするデータ型について、結果セットの SQL_DATA_TYPE フィールドは次のように値を返します。DATE は SQL_DATE、TIME は SQL_TIME、AUTOTIMESTAMP、DATETIME、TIMESTAMP、および TIMESTAMP2 は SQL_TIMESTAMP です。
SQL_DATETIME_SUB	SMALLINT	AUTOTIMESTAMP、DATE、DATETIME、TIME、TIMESTAMP、および TIMESTAMP2 のサブタイプコード。その他のデータ型について、この列は NULL を返します。 • 1 = DATE (SQL_CODE_DATE) • 2 = TIME (SQL_CODE_TIME) • 3 = AUTOTIMESTAMP、DATETIME、TIMESTAMP、および TIMESTAMP2 (SQL_CODE_TIMESTAMP)
CHAR_OCTET_LENGTH	INTEGER	文字またはバイナリ データ型列の最大長 (バイト単位)。その他すべてのデータ型について、この列は NULL を返します。

列名	データ型	説明
ORDINAL_POSITION	NULL でない INTEGER	入力および出力パラメーターについては、プロシージャ定義におけるパラメーターの位置を表す序数（1 から始まり、増加するパラメーター順）。 戻り値がある場合、戻り値については 0 が返されます。結果セット列については、結果セットにおける列の位置を表す序数。この場合、結果セット列の最初の列は 1 から始まります。
IS_NULLABLE	VARCHAR	列に NULL が含まれていない場合は "NO"。 列に NULL が含まれている場合は "YES"。 ヌル値を許可するかどうかわからない場合、この列は長さゼロの文字列を返します。この列に返される値は、 NULLABLE 列に返される値とは異なります。

例

デフォルトで、Demodata サンプル データベースにはストアド プロシージャとユーザー定義関数が何も含まれていません。dbo.fsQSQLProcedureColumns 関数の出力を提供するために、[dbo.fsQSQLProcedures](#) の例で示されているストアド プロシージャとユーザー定義関数を作成してください。この例は、Demodata にストアド プロシージャ curs1、Enrollstudent2、AInsert、および AR と、ユーザー定義関数 CalInterest が含まれていることを前提とします。

次のステートメントは、Demodata サンプル データベース内のすべてのストアド プロシージャおよびユーザー定義関数の、すべての列に関する情報を返します。

```
SELECT * FROM dbo.fsqsqlprocedurecolumns ('Demodata', null, null)
```

結果セット（スペースを考慮して短縮されています）：

PROCEDURE_NAME	COLUMN_NAME	COLUMN_TYPE	DATA_TYPE
=====	=====	=====	=====
AInsert	:AGUID	1	-2
AInsert	:APeriod	1	4
AInsert	:BBal	1	-6
AInsert	:BCr	1	3
AInsert	:BDr	1	3
AInsert	:BNet	1	3
AInsert	:HTrx	1	-6
AInsert	:PCr	1	3
AInsert	:PDr	1	3
AInsert	:PNet	1	3
AR	:BIID	4	5

AR	:BName	1	1
CalInterest	:period	1	7
CalInterest	:principle	1	8
CalInterest	:rate	1	8
CalInterest	:RETURN_VALUE	5	8
curs1	:Arg1	1	1
curs1	:Arg2	1	4
Enrollstudent2	:Class_Id	1	4
Enrollstudent2	:Stud_id	1	4

20 行が影響を受けました。

dbo.fsQLSpecialColumns

この関数は、指定されたテーブルについて次のような列情報を取得します。対象となるのは、テーブル内の行を一意に識別する最適な列のセット、およびトランザクションによって行内の任意の値が更新されたときに自動的に更新される列です。

構文

```
dbo.fsQLSpecialColumns (<'database_qualifier' | null>, <'table_name' | null>, <'nullable' | null>)
```

引数

パラメーター	データ型	デフォルト値	説明
<i>database_qualifier</i>	VARCHAR	現在のデータベース	詳細を取得するデータベースの名前
<i>table_name</i>	VARCHAR	(デフォルト値なし)	列情報を要求するテーブルの名前
<i>nullable</i>	SMALLINT	(デフォルト値なし)	ヌル値を持つことができる特殊な列を返すかどうかを判断します。次のいずれかでなければなりません。 • 0 = NO_NULLS - ヌル値を持つことができる特殊な列を除外します。 • 1 = NULLABLE - ヌル値を持つことができる特殊な列を返します。

返される結果セット

列名	データ型	説明
SCOPE	SMALLINT	rowid の適用範囲。次のいずれかの値を含みます。 • 0 = SCOPE_CURROW • 1 = SCOPE_TRANSACTION • 2 = SCOPE_SESSION IdentifierType が SQL_ROWVER である場合は、NULL が返されます。
COLUMN_NAME	NULL でない VARCHAR	列の名前。名前のない列については、空文字列が返されます。
DATA_TYPE	NULL でない SMALLINT	SQL データ型。『 <i>ODBC Guide</i> 』の 使用できるデータ型 も参照してください。
PRECISION	INTEGER	データベース内のプロシージャ列のサイズ。 Decimal データ型の精度と小数位 も参照してください。

列名	データ型	説明
LENGTH	INTEGER	SQL_C_DEFAULT が指定された場合は、SQLGetData または SQLFetch 操作で転送されるデータの長さ（バイト単位）。数値データの場合、このサイズは、データベースに格納されているデータのサイズとは異なる場合があります。 Zen で使用できるデータ型 も参照してください。
SCALE	SMALLINT	データベース内のプロシージャ列の小数点以下の桁数。小数点以下の桁数が適用されないデータ型では、NULL が返されます。 Decimal データ型の精度と小数位 も参照してください。
PSEUDO_COLUMN	SMALLINT	列が擬似列であるかどうかを示します。 0 = PC_UNKNOWN Zen は擬似列をサポートしません。

例

この例では次のようなテーブルを作成します。テーブルは行を一意に識別する 2 つの列を持ち、トランザクションが行内の任意の値を更新したときにそれらの列が自動的に更新されます。

```
CREATE TABLE t2 (c1 IDENTITY, c2 INTEGER, c3 SMALLINT NOT NULL, c4 TIMESTAMP NOT NULL)
ALTER TABLE t2 ADD PRIMARY KEY (c1, c4);

SELECT * FROM dbo.fsSQLSpecialColumns ('Demodata' , 't2' , 'null')
```

結果セット（スペースを考慮して短縮されています）：

COLUMN_NAME	DATA_TYPE	TYPE_NAME	PRECISION	LENGTH
c1	4	INTEGER	4	4
c4	11	DATETIME	16	16

2 行が影響を受けました。

dbo.fsSQLStatistics

この関数は、テーブルに関する統計情報の一覧と、そのテーブルに関連付けられているインデックスを 1 つの結果セットとして返します。

構文

```
dbo.fSQLStatistics (<'database_qualifier' | null>, <'table_name' | null>, <'unique' | null>)
```

引数

パラメーター	データ型	デフォルト値	説明
<i>database_qualifier</i>	VARCHAR	現在のデータベース	詳細を取得するデータベースの名前
<i>table_name</i>	VARCHAR	(デフォルト値なし)	列情報を要求するテーブルの名前。パターン検索がサポートされます (文字列検索パターン を参照してください)。
<i>unique</i>	SMALLINT	(デフォルト値なし)	インデックスの種類： 0 = INDEX_UNIQUE 1 = INDEX_ALL

返される結果セット

列名	データ型	説明
TABLE_QUALIFIER	VARCHAR	統計情報またはインデックスが適用されるテーブルを含んでいるデータベースの名前。データベースに適用されない場合は NULL。
TABLE_OWNER	VARCHAR	統計情報またはインデックスが適用されるテーブルのスキーマ名。データベースに適用されない場合は NULL。
TABLE_NAME	NULL でない VARCHAR	統計情報またはインデックスが適用されるテーブルの名前。
NON_UNIQUE	SMALLINT	インデックスが重複値を許可しないかどうかを示します。 • 0 = FALSE - インデックス値が一意である必要がある場合。 • 1 = TRUE - インデックス値が一意でなくてもよい場合。 TYPE が TABLE_STAT である場合は、NULL が返されます。

列名	データ型	説明
INDEX_QUALIFIER	VARCHAR	DROP INDEX を行うインデックス名を修飾するために使用されている識別子。インデックス修飾子がデータベースでサポートされない場合、または TYPE が TABLE_STAT である場合は、NULL が返されます。
INDEX_NAME	VARCHAR	インデックス名。TYPE が TABLE_STAT である場合は、NULL が返されます。
TYPE	NULL でない SMALLINT	返される情報の種類： • 0 = TABLE_STAT • 3 = INDEX_OTHER
SEQ_IN_INDEX	SMALLINT	インデックスにおける列のシーケンス番号（1 から始まります）。TYPE が TABLE_STAT である場合は、NULL が返されます。
COLUMN_NAME	VARCHAR	列名。列が PERSONID + NAME などのような式に基づいている場合は、その式が返されます。式を判断できない場合は、空文字列が返されます。 TYPE が TABLE_STAT である場合は、NULL が返されます。
COLLATION	CHAR	列の並べ替え順。 A = 昇順 D = 降順 列の並べ替え順がデータベースでサポートされない場合、または TYPE が TABLE_STAT である場合は、NULL が返されます。
CARDINALITY	INTEGER	テーブルまたはインデックスの基数。 TYPE が TABLE_STAT である場合は、テーブル内の行数。TYPE が TABLE_STAT でない場合は、インデックスに含まれる一意な値の数。 この値をデータベースから入手できない場合は、NULL が返されます。

列名	データ型	説明
PAGES	INTEGER	インデックスまたはテーブルの格納に使用されるページ数。 TYPE が TABLE_STAT である場合は、テーブルのページ数。TYPE が TABLE_STAT でない場合は、インデックスのページ数。 この値をデータベースから入手できない場合、またはデータベースに適用されない場合は、NULL が返されます。
FILTER_CONDITION	VARCHAR	インデックスがフィルター選択されたインデックスである場合は、これは CLASSID > 150 などのようなフィルター条件になります。フィルター条件を判断できない場合、これは空文字列になります。 インデックスがフィルターされたインデックスでない場合、フィルターされたインデックスであるかどうかを判断できない場合、あるいは TYPE が TABLE_STAT である場合は、NULL になります。

例

次の例は、デフォルトの Demodata サンプル データベース中、文字 *c* で始まるすべてのテーブルの全インデックスに関する統計情報を返します。INDEX_NAME の NULL は除いてあります。[文字列検索パターン](#)も参照してください。

```
SELECT * FROM dbo.fsQLStatistics ('Demodata', 'c%', 1) WHERE INDEX_NAME IS NOT NULL
```

結果セット（スペースを考慮して短縮されています）：

TABLE_NAME	INDEX_NAME	COLUMN_NAME
=====	=====	=====
Class	UK_ID	ID
Class	Class_Name	Name
Class	Class_Name	Section
Class	Class_seg_Faculty	Faculty_ID
Class	Class_seg_Faculty	Start_Date
Class	Class_seg_Faculty	Start_Time
Class	Building_Room	Building_Name
Class	Building_Room	Room_Number
Class	Building_Room	Start_Date
Class	Building_Room	Start_Time
Course	Course_Name	Name
Course	DeptName	Dept_Name

12 行が影響を受けました。

dbo.fSQLTables

この関数は、データベースに格納されているテーブル、カタログ、またはスキーマの名前と、テーブルの種類の一覧を返します。

構文

```
dbo.fSQLTables (<'database_qualifier' | null>, <'table_name' | null>, <['type' | null]>)
```

引数

パラメーター	データ型	デフォルト値	説明
<i>database_qualifier</i>	VARCHAR	現在のデータベース	詳細を取得するデータベースの名前
<i>table_name</i>	VARCHAR	(デフォルト値なし)	情報を取得する必要があるテーブルの名前
<i>type</i>	VARCHAR	(デフォルト値なし)	次のいずれかでなければなりません。 • TABLE - ユーザー テーブルのみ返します • SYSTEM TABLE - すべてのシステム テーブルを返します • VIEW - ビューのみ返します • ヌル - すべてのテーブルを返します

返される結果セット

列名	データ型	説明
TABLE_QUALIFIER	VARCHAR	データベース名。データベースに適用されない場合は NULL。
TABLE_OWNER	VARCHAR	テーブル所有者の名前。データベースに適用されない場合は NULL。
TABLE_NAME	VARCHAR	テーブルの名前
TABLE_TYPE	VARCHAR	次のいずれかです。 • TABLE • VIEW • SYSTEM TABLE
REMARKS	VARCHAR	テーブルの記述。

例

次の例は、デフォルトの Demodata サンプル データベース中の、ユーザー テーブルおよびシステム テーブルの一覧を返します。

```
SELECT * FROM dbo.fsQLTables ('Demodata', null, null)
```

結果セット（スペースを考慮して短縮されています）：

TABLE_NAME	TABLE_TYPE
=====	=====
X\$File	SYSTEM TABLE
X\$Field	SYSTEM TABLE
X\$Index	SYSTEM TABLE
X\$View	SYSTEM TABLE
X\$Proc	SYSTEM TABLE
X\$Relate	SYSTEM TABLE
X\$Trigger	SYSTEM TABLE
X\$Attrib	SYSTEM TABLE
X\$Occurs	SYSTEM TABLE
X\$Variant	SYSTEM TABLE
Billing	TABLE
Class	TABLE
Course	TABLE
Dept	TABLE
Enrolls	TABLE
Faculty	TABLE
Person	TABLE
Room	TABLE
Student	TABLE
Tuition	TABLE
X\$User	SYSTEM TABLE
X\$Rights	SYSTEM TABLE

22 行が影響を受けました。

dbo.fsQLDBTableStat

この関数は、Btrieve Stat（15）オペレーションから返される詳細など、現在のデータベース内のテーブルやファイルに関する基本情報を結果セットとして返します。

構文

```
dbo.fsQLDBTableStat ('table_name')
```

引数

パラメーター	データ型	デフォルト値	説明
<i>table_name</i>	VARCHAR	（デフォルト値なし）	情報を取得する、現在のデータベース内のテーブルの名前。

返される結果セット

列名	データ型	説明
Table Name	VARCHAR	テーブルの名前。
Table Location	VARCHAR	テーブルのデータ ファイルの絶対パス名。
Dictionary Path	VARCHAR	データベースの辞書パス。
File Version	VARCHAR	16 進数のファイルの Btrieve バージョン（例：バージョン 13 の場合は "13.0"）。
Record Length	SMALLINT	Stat（15）によって返される、ファイルの固定レコード長。
Page Size	SMALLINT	Stat（15）によって返される、ファイルのページ サイズ。
Number of Records	BIGINT	Stat（15）によって返される、ファイル内のレコード数。
Number of Indexes	SMALLINT	Stat（15）によって返される、インデックス数。

列名	データ型	説明
Unused LinkedDup Ptr	SMALLINT	Stat (15) によって返される、未使用の重複ポインター数。
Unused PreAlloc Pages	SMALLINT	Stat (15) によって返される、ファイル内の未使用の空ページ数。
Variable Len Records	VARCHAR	ファイルに可変長レコードがあるかどうかを示す、YES または NO。
Blank Truncation	VARCHAR	可変長レコードでブランク トランケーションを使用しているかどうかを示す、YES または NO。
Record Compression	VARCHAR	Btrieve のデータ圧縮かどうかを示す、YES または NO。
Page Compression	VARCHAR	Btrieve のページ圧縮かどうかを示す、YES または NO。
Key Only File	VARCHAR	ファイルはキー オンリー ファイルであるかどうかを示す、YES または NO。
Index Balancing	VARCHAR	Btrieve のインデックス バランスかどうかを示す、YES または NO。
Freespace Threshold	VARCHAR	空きスペース スレッシュホールドが指定されている場合、そのスレッシュホールドを示すパーセンテージ。
Uses ACS	VARCHAR	ファイルはオルタネート コレーティング シーケンスを使用しているかどうかを示す、YES または NO。
System Data	VARCHAR	ファイルはシステム データ キーを有効にしているかどうかを示す、YES または NO。
Used LinkedDup Ptr	SMALLINT	リンク重複を使用するキー数。

上記のフィールドの詳細については、『*Btrieve API Guide*』の Create (14) および Stat (15) を参照してください。

例

次の例は、デフォルトの Demodata サンプル データベースに含まれるテーブルの情報を返します。

```
SELECT * FROM dbo.fsQLDBTableStat ('student')
```

結果セット (スペースを考慮して再編成されています) :

Table Name	Student
Table Location	C:\PROGRAMDATA\ACTIAN\ZEN\DEMODATA\Student.mkd
Dictionary Path	C:\PROGRAMDATA\ACTIAN\ZEN\DEMODATA
File Version	9.5
Record Length	76
Page Size	4096
Number of Records	1288
Number of Indexes	2
Unused LinkedDup Ptr	0
Unused PreAlloc Pages	0
Variable Len Records	NO
Blank Truncation	NO
Record Compression	NO
Page Compression	NO
Key Only File	NO
Index Balancing	NO
Freespace Threshold	0%
Uses ACS	NO
System Data	YES
Used LinkedDup Ptr	0

文字列検索パターン

以下のシステム カタログ関数は、文字列検索パターンをサポートしています。

- [dbo.fSQLForeignKeys](#)
- [dbo.fSQLPrimaryKeys](#)
- [dbo.fSQLStatistics](#)

検索パターンで2つのワイルドカード文字を使用できます。

- パーセント記号 (%) は、任意の連続する n 個の文字を表します。
- アンダースコア (_) は、単一文字を表します。

例

次の表は、文字列検索パターンの使用例を示します。

ステートメントの例	戻り値
SELECT * FROM dbo.fSQLStatistics (null, '%', 0)	現在のデータベースにある、一意のインデックスを持つすべてのテーブル
SELECT * FROM dbo.fSQLStatistics (null, 't%', 1)	現在のデータベースにある、名前が 't' で始まり、インデックスを持つすべてのテーブル

ステートメントの例	戻り値
SELECT * FROM dbo.fSQLPrimaryKeys (null, '%')	現在のデータベースにある、主キーを持つすべてのテーブル
SELECT * FROM dbo.fSQLPrimaryKeys (null, 't%')	現在のデータベースにある、名前が 't' で始まり、主キーを持つすべてのテーブル
SELECT * FROM dbo.fSQLForeignKeys (null, '%', '%')	現在のデータベースにある、主キーを持つすべてのテーブルと、それに関連する外部キー テーブル

A. データ型

ここでは、MicroKernel およびリレーショナル エンジンを通じて Zen によって提供される、データ型およびデータ型マッピングについて説明します。

- [Zen で使用できるデータ型](#)
- [データ型に関する注意事項](#)
- [旧データ型](#)
- [Btrieve キーのデータ型](#)
- [キーでないデータ型](#)

Zen で使用できるデータ型

次の表は、Zen によってサポートされるトランザクショナル データ型とリレーショナル データ型のマッピングです。Btrieve データ ファイルのデータにアクセスする SQL アプリケーションの開発者に有用です。

トランザクショナル型 (サイズ)	リレーショナル型	メタデー タの型 コード値	サイズ (バイ ト数)	作成 / 追加 パラメー ター ¹	データ 型に関 する注 記
AUTOINCREMENT(2)	SMALLIDENTITY	15	2		
AUTOINCREMENT(4)	IDENTITY	15	4		
AUTOINCREMENT(8)	BIGIDENTITY	15	8		
AUTOTIMESTAMP	AUTOTIMESTAMP	32	8		11
BFLOAT(4)	BFLOAT4	9	4	ヌルでない	4
BFLOAT(8)	BFLOAT8	9	8	ヌルでない	4
BLOB	LONGVARBINARY	21	N/A ²	ヌルでない	2、3、 6
BLOB(2)	NLONGVARCHAR	21	N/A ²	ヌルでない 大小文字無 視	7

トランザクショナル型 (サイズ)	リレーショナル型	メタデー タの型 コード値	サイズ (バイト数)	作成 / 追加 パラメー ター ¹	データ 型に関 する注 記
CLOB	LONGVARCHAR	21	N/A ²	ヌルでない 大小文字無 視	5、6
CURRENCY	CURRENCY	19	8	ヌルでない	
DATE	DATE	3	4	ヌルでない	
なし	DATETIME	30	8	ヌルでない	10
DECIMAL	DECIMAL	5	1 - 64	桁数 小数位 ヌルでない	
FLOAT(4)	REAL	2	4	ヌルでない	
FLOAT(8)	DOUBLE	2	8	ヌルでない	
GUID	UNIQUEIDENTIFIER	27	16	ヌルでない	
INTEGER(1)	TINYINT	1	1	ヌルでない	
INTEGER(2)	SMALLINT	1	2	ヌルでない	
INTEGER(4)	INTEGER	1	4	ヌルでない	
INTEGER(8)	BIGINT	1	8	ヌルでない	
MONEY	DECIMAL	6	1 - 64	桁数 小数位 ヌルでない	
NUMERIC	NUMERIC	8	1 - 37	桁数 小数位 ヌルでない	4
NUMERICSA	NUMERICSA	18	1 - 37	桁数 小数位 ヌルでない	4
NUMERICSLB	NUMERICSLB	28	1 - 37	桁数 小数位 ヌルでない	4
NUMERICSLS	NUMERICSLS	29	1 - 37	桁数 小数位 ヌルでない	4

トランザクショナル型 (サイズ)	リレーショナル型	メタデー タの型 コード値	サイズ (バイト数)	作成 / 追加 パラメー ター ¹	データ 型に関 する注 記
NUMERICSTB	NUMERICSTB	31	1 - 37	桁数 小数位 ヌルでない	4
NUMERICSTS	NUMERICSTS	17	1 - 37	桁数 小数位 ヌルでない	4
STRING	BINARY	0	1 - 8,000	サイズ ヌルでない 大小文字無 視	2、3
STRING	CHAR	0	1 - 8,000	サイズ ヌルでない 大小文字無 視	1
TIME	TIME	4	4	ヌルでない	
TIMESTAMP	TIMESTAMP	20	8	ヌルでない	
TIMESTAMP2	TIMESTAMP2	34	8	ヌルでない	11
UNSIGNED(1) BINARY	UTINYINT	14	1	ヌルでない	
UNSIGNED(2) BINARY	USMALLINT	14	2	ヌルでない	
UNSIGNED(4) BINARY	UINTINTEGER	14	4	ヌルでない	
UNSIGNED(8) BINARY	UBIGINT	14	8	ヌルでない	
WSTRING	NCHAR	25	2 - 8,000	サイズ 1 - 4,000 ヌルでない 大小文字無 視	12、13
WZSTRING	NVARCHAR	26	2 - 8,000	サイズ 1 - 4,000 ヌルでない 大小文字無 視	12、14

トランザクショナル型 (サイズ)	リレーショナル型	メタデータ の型 コード値	サイズ (バイト 数)	作成 / 追加 パラメー ター ¹	データ 型に関 する注 記
ZSTRING	VARCHAR	11	1 - 8,000	サイズ ヌルでない 大小文字無 視	5
なし	BIT	16	1 ビッ ト		6、8
LOGICAL(1)	BIT	7	1 ビッ ト		9
LOGICAL(2)	SMALLINT	1	2	ヌルでない	

¹ 必須パラメーターは「桁数」と「サイズ」です。オプションパラメーターは「大小文字無視」、「ヌルでない」、および「小数位」です。

² "N/A" は「適用外」を意味します。

データ型に関する注記

1. 空白で埋められます
2. FIELD.DDF で、バイナリの使用を SQL に知らせるフラグ セット。『*Distributed Tuning Interface Guide*』の **COLUMNMAP フラグ** および『*Distributed Tuning Objects Guide*』の **列フラグ** を参照してください。
3. バイナリ ゼロで埋められます
4. 変数としても、またストアド プロシージャ内でも使用できません
5. 埋め込みなし
6. インデックスを作成できません
7. FIELD.DDF で、NLONGVARCHAR の使用を SQL に知らせるフラグ セット。『*Distributed Tuning Interface Guide*』の **COLUMNMAP フラグ** および『*Distributed Tuning Objects Guide*』の **列フラグ** を参照してください。
8. TRUEBITCREATE は on (デフォルト) に設定する必要があります。
9. TRUEBITCREATE は off に設定する必要があります。
10. 型コード 30 は MicroKernel エンジンの型コードではありません。リレーショナル エンジン メタデータ内における DATETIME の識別子です。
11. UBIGINT と同じようにソートされます。
12. Unicode 型の場合、列のサイズは 2 バイトの UCS-2 単位の数を表します。
13. Unicode の空白 (2 バイト) で埋められます。
14. Unicode の NUL 文字 (2 バイト、バイナリ ゼロ) で埋められます。

データ型の範囲

次の表は、Zen データ型で有効な値の範囲と、必要に応じてそれらに増分される値を示したものです。

リレーショナル データ型	有効値の範囲
AUTOTIMESTAMP	1970-01-01 00:00:00.000000000 ~ 2554-07-21 23:34:33.709551615 ゼロで初期化すると、挿入または次の更新で現在の日時が使用されます。
BFLOAT4	-1.70141172e+38 ~ +1.70141173e+38 BFLOAT4 をインクリメントまたはデクリメントできる最小値は 2.938736e-39 です。
BFLOAT8	-1.70141173e+38 ~ +1.70141173e+38 BFLOAT8 をインクリメントまたはデクリメントできる最小値は 2.93873588e-39 です。
BIGIDENTITY	-9223372036854775808 ~ +9223372036854775807
BIGINT	-9223372036854775808 ~ +9223372036854775807
BINARY	範囲は適用されません
BIT	範囲は適用されません
CHAR	範囲は適用されません
CURRENCY	-922337203685477.5808 ~ +922337203685477.5807
DATE	01-01-0001 ~ 12-31-9999 メモ : 00-00-0000 は有効な値ではありません。DATE 型の値に 00-00-0000 データが入っているレガシー データがある場合は、クエリに "is null" を使用することでそれを照会できます。
DATETIME	1753-01-01 00:00:00.000 ~ 9999-12-31 23:59:59.999。1 ミリ秒の精度
DECIMAL	長さと小数点以下の桁数によって異なります
DOUBLE	-1.7976931348623157e+308 ~ +1.7976931348623157e+308 DOUBLE をインクリメントまたはデクリメントする最小値は 2.2250738585072014e-308 です。

リレーショナル データ型	有効値の範囲
FLOAT	-1.7976931348623157E+308 ～ +1.7976931348623157E+308 FLOAT をインクリメントまたはデクリメントできる最小値は 2.2250738585072014e-308 です。
IDENTITY	-2147483648 ～ +2147483647
INTEGER	-2147483648 ～ +2147483647
LOGICAL	範囲は適用されません
LONGVARBINARY	範囲は適用されません
LONGVARCHAR	範囲は適用されません
MONEY	-9999999999999999.99 ～ +9999999999999999.99
NCHAR	範囲は適用されません
NLONGVARCHAR	範囲は適用されません
NUMERIC	長さと小数点以下の桁数に基づきます。 Decimal データ型の精度と小数位 を参照してください。
NUMERICSA	長さと小数点以下の桁数に基づきます。 Decimal データ型の精度と小数位 を参照してください。
NUMERICSLB	長さと小数点以下の桁数に基づきます。 Decimal データ型の精度と小数位 を参照してください。
NUMERICSLS	長さと小数点以下の桁数に基づきます。 Decimal データ型の精度と小数位 を参照してください。
NUMERICSTB	長さと小数点以下の桁数に基づきます。 Decimal データ型の精度と小数位 を参照してください。
NUMERICSTS	長さと小数点以下の桁数に基づきます。 Decimal データ型の精度と小数位 を参照してください。
NVARCHAR	範囲は適用されません
REAL	-3.4028234E+38 ～ +3.4028234e+38 REAL 値をインクリメントまたはデクリメントできる最小値は 1.4E-45 です。
SMALLIDENTITY	-32768 ～ +32767
SMALLINT	-32768 ～ +32767

リレーショナル データ型	有効値の範囲
TIME	00:00:00 ～ 23:59:59
TIMESTAMP	0001-01-01 00:00:00.00000000 ～ 9999-12-31 23:59:59.99999999 UTC 小数位は変わります。タイムスタンプ データ型の小数位と返される関数値を参照してください。
TIMESTAMP2	1970-01-01 00:00:00.0000000000 ～ 2554-07-21 23:34:33.709551615 UTC 小数位は変わります。タイムスタンプ データ型の小数位と返される関数値を参照してください。
TINYINT	-128 ～ +127
UBIGINT	0 ～ 18446744073709551615
INTEGER	0 ～ 4294967295
UNIQUEIDENTIFIER	範囲は適用されません
USMALLINT	0 ～ 65535
UTINYINT	0 ～ 255
VARCHAR	範囲は適用されません

演算子の優先順位

式には複数の演算子が含まれることがあり、それらは優先順位に従って実行されます。Zen で使用される順序は次のとおりで、レベル 1 が最高、レベル 9 が最低です。高レベルの演算子は低レベルの演算子より先に評価されます。

1. + (正)、- (負、否定)、~ (ビット演算 NOT)
2. * (乗算)、/ (除算)、% (剰余)
3. + (加算)、+ 連結)、- (減算)、& (ビット演算 AND)
4. =、>、<、>=、<=、<>、!= (これらの比較演算子はそれぞれ、等しい、より大きい、より小さい、以上、以下、等しくない、を表します)
5. ^ (ビット演算 排他的 OR)、| (ビット演算 OR)
6. NOT

7. AND

8. ALL、ANY、BETWEEN、IN、LIKE、OR、SOME

9. = (代入)

式内にある同じ優先レベルの演算子は、左から右へ評価されます。たとえば、`SET :Counter = 12 / 4 * 7`では、除算は乗算より先に評価され、結果として21が返されます。

かっこ

かっこを使用すると、演算子の優先順位より優先させることができます。最初にかっこ内のものがすべて評価され、1つの値がもたらされます。その後、この値がかっこの外にある演算子で使用されます。たとえば、次のステートメントの場合、除算演算子は通常は加算演算子より先に評価されます。結果は12になります（つまり、 $8 + 4$ ）。しかし、加算はかっこで囲まれているため最初に実行されるので、プロシージャは結果の4を返します。

```
SET :Counter = 32 / (4 + 4)
```

式にネストされたかっこがある場合は、一番深くネストされている式が最初に評価され、次に深くネストされている式の順に評価されます。たとえば、次のステートメントでは、加算が最初に実行され、次に乗算、減算、最後に除算が実行されます。結果は値5になります。

```
SET :Counter = 100 / (40 - (2 * (5 + 5)));
```

データ型の優先順位

データ型の優先順位は、異なる型の2つの式が演算子によって結合されているときに、その結果の型を決定します。優先順位の低いデータ型は優先順位の高いデータ型に変換されます。

メモ： 互換性のないデータ型で演算を行う、たとえば INTEGER を CHAR に加算すると、エラーが返されます。

数値データ型

リレーショナルの数値データ型では次の優先順位を使用します。

1. DOUBLE、FLOAT、BFLOAT8（最高）

-
2. REAL、BFLOAT4
 3. DECIMAL、NUMERIC、NUMERICSA、NUMERICSTS
 4. NUMERICSLS、NUMERICSTB、NUMERICSLB
 5. CURRENCY、MONEY
 6. BIGINT、UBIGINT、BIGIDENTITY
 7. INTEGER、UINTeger、IDENTITY
 8. SMALLINT、USMALLINT、SMALLIDENTITY
 9. TINYINT、UTINYINT
 10. BIT（最低）

文字データ型

リレーショナルの文字データ型では次の優先順位を使用します。

1. NLONGVARCHAR
2. NCHAR、NVARCHAR
3. LONGVARCHAR
4. CHAR、VARCHAR

NCHAR または NVARCHAR と NLONGVARCHAR を連結させた場合、結果は NLONGVARCHAR になります。

NCHAR と LONGVARCHAR を連結させた場合、結果は NLONGVARCHAR になります。

CHAR または VARCHAR と LONGVARCHAR を連結させた場合、結果は LONGVARCHAR になります。

CHAR と VARCHAR を連結させた場合、結果の型はその連結で最初にくるデータ型になります。つまり左から右に移動します。たとえば、c1 が CHAR で c2 が VARCHAR の場合、(c1 + c2) の結果は CHAR になり、(c2 + c1) の結果は VARCHAR になります。

優先順位のないデータ型

BINARY、LONGVARBINARY、および UNIQUEIDENTIFIER データ型の優先順位はありません。これらのデータ型を組み合わせる演算は許可されていません。

日付と時刻のデータ型は、他のどの日付または時刻のデータ型とも組み合わせることができません。

Decimal データ型の精度と小数位

精度は、数値内の数字の数（桁数）です。小数位は、数値内の小数点より右側の数字の数です。たとえば、909.777 という数では精度が 6 で小数位が 3 となります。

NUMERIC、NUMERICSA、および DECIMAL データ型の最大精度は 64 です。NUMERICSTS および NUMERICSLs では、プラスまたはマイナス符号のために 1 バイトが予約されているため、最大精度は 63 です。

DECIMAL を除くすべての数値データ型で、精度と小数位は固定です。同じデータ型の 2 つの式で算術演算を行った場合、結果は同じデータ型となり、そのデータ型の精度と小数位を持ちます。データ型が異なる式で演算を行った場合は、優先順位のルールによって結果のデータ型が決定されます。結果の精度と小数位は、そのデータ型に定義されている桁数になります。

次の条件における演算の場合、結果は DECIMAL になります。

- 両方の式が DECIMAL。
- 一方の式が DECIMAL で、他方の式が DECIMAL よりも優先順位の低いデータ型。

次の表は、演算結果のデータ型が DECIMAL になる場合に精度と小数位を導き出す方法の定義を示しています。*exp* は「式」を表し、*s* は「小数位」、*p* は「精度」を表します。

演算	精度	小数位
Addition (<i>exp1</i> + <i>exp2</i>)	$\max(s1, s2) + \max(p1 - s1, p2 - s2) + 1$	$\max(s1, s2)$
Subtraction (<i>exp1</i> - <i>exp2</i>)	$\max(s1, s2) + \max(p1 - s1, p2 - s2) + 1$	$\max(s1, s2)$
Multiplication (<i>exp1</i> * <i>exp2</i>)	$p1 + p2 + 1$	$s1 + s2$
Division (<i>exp1</i> / <i>exp2</i>)	$p1 - s1 + s2 + \max(6, s1 + p2 + 1)$	$\max(6, s1 + p2 + 1)$
UNION (<i>exp1</i> UNION <i>exp2</i>)	$\max(s1, s2) + \max(p1 - s1, p2 - s2) + 1$	$\max(s1, s2)$

たとえば、DECIMAL (8,2) および DECIMAL (7,4) と定義されている 2 つのフィールドを加算または減算した場合、結果フィールドは DECIMAL (11,4) になります。

タイムスタンプ データ型的小数位と返される関数値

タイムスタンプ データ型では、小数位はタイムスタンプ内の小数の秒部分の小数点より右側の桁数です。たとえば、2019-12-31 23:59:59.782 では小数位は 3 桁、つまりミリ秒となります。

Zen 14.10 から、TIMESTAMP データ型および TIMESTAMP2 データ型的小数位を選択できるようになりました。たとえば、次の SQL スクリプトは 4 つの列を持つテーブルを作成します。各列は TIMESTAMP または TIMESTAMP2 とし、最初の 2 列はデフォルトの小数位を持ち、次の 2 列は小数位を小数点以下 1 桁に設定しています。

```
create table times
(ts timestamp default sysdatetime(),
 ts2 timestamp2 default sysdatetime(),
 ts_1 timestamp(1) default sysdatetime(),
 ts2_1 timestamp2(1) default sysdatetime());
insert into times default values;
select * from times;
```

SELECT ステートメントは次の行を返します。

ts	ts2	ts-1	ts2-1
2019/12/10 10:25:39.555	2019/12/10 10:25:39.555080200	2019/12/10 10:25:39.5	2019/12/10 10:25:39.5

小数位を短くした場合、小数の秒は丸められないことに留意してください。

次の表は、小数位を持つ日付およびタイムスタンプをサポートする Zen データ型の一覧を示します。

データ型	デフォルトの小数位を含む形式	小数位
AUTOTIMESTAMP	yyyy-mm-dd hh:mm:ss.nnnnnnnnn (ナノ秒)	9
DATETIME	yyyy-mm-dd hh:mm:ss.nnn (ミリ秒)	3
TIMESTAMP	yyyy-mm-dd hh:mm:ss.nnn (ミリ秒)	3
TIMESTAMP(n)	yyyy-mm-dd hh:mm:ss.nnnnnnn (なし～セプタ秒)	0 – 7
TIMESTAMP2	yyyy-mm-dd hh:mm:ss.nnnnnnnnn (ナノ秒)	9
TIMESTAMP2(n)	yyyy-mm-dd hh:mm:ss.nnnnnnnnn (なし～ナノ秒)	0 – 9

次の表は、小数位を持つ日付値およびタイムスタンプ値を返す Zen スカラー関数の一覧を示します。

関数	形式	小数位
CURRENT_TIMESTAMP()	yyyy-mm-dd hh:mm:ss.nnn (ミリ秒)	3
NOW()	yyyy-mm-dd hh:mm:ss.nnn (ミリ秒)	3
SYSDATETIME()	yyyy-mm-dd hh:mm:ss.nnnnnnnnn (ナノ秒)	9
SYSUTCDATETIME()	yyyy-mm-dd hh:mm:ss.nnnnnnnnn (ナノ秒)	9

関数から返されるタイムスタンプの小数位が、その値が書き込まれる列のデータ型の小数位より小さい場合、末尾の小数点以下桁数はゼロで埋められます。たとえば、CURRENT_TIMESTAMP() の値を TIMESTAMP2 列に入れるとした場合は、値 2019/12/10 14:23:46 292000000 が返されます。

切り捨て

アプリケーションが異なる SQL DBMS 製品に対して実行される場合は、切り捨てに関する以下の問題が発生する可能性があります。

特定の状況で、一部の SQL DBMS 製品では切り捨てのためにデータを挿入できないのに、Zen ではその同じデータを挿入できることがあります。さらに、Zen の SQL_SUCCESS_WITH_INFO のレポートと切り捨てられる情報は、メッセージが報告されるタイミングに基づく特定のシナリオにおいて、一部の SQL DBMS 製品と異なります。

数値文字列データおよび数値データは、Zen では常に切り捨てられるのに対し、SQL DBMS 製品では丸められます。たとえば、数値文字列または数値の 123.457 があり、これを 6 バイトの文字列の列または小数部 2 桁の数値列に挿入する場合、Zen は常に 123.45 を挿入します。これに対して、ほかの DBMS 製品は 123.46 という値を挿入することがあります。

データ型に関する注意事項

このトピックでは、使用可能なデータ型に関するさまざまな動作およびキー情報について説明します。

CHAR、NCHAR、VARCHAR、NVARCHAR、LONGVARCHAR、および NLONGVARCHAR

- CHAR および NCHAR 列には、末尾の空白が埋め込まれます。比較演算（LIKE および =）では、このような空白は演算の対象になりません。ただし、LIKE の場合、クエリに空白が明示的に入力されているとき（'abc %' など）は、ワイルドカードの前の空白は演算の対象になります。この例の場合、'abc< 空白 >< 任意の文字 >'を検索します。
- CHAR 型は、データベース コード ページを使用して、つまり、1 文字につき 1 バイト以上を使用して文字を格納します。NCHAR 型は、UCS-2 の 2 バイト値として文字を格納します。
- VARCHAR、NVARCHAR、LONGVARCHAR、および NLONGVARCHAR 値は末尾の空白は埋め込まれません。意味のあるデータはヌル文字で終わります。
- VARCHAR および NVARCHAR の比較演算子では、末尾の空白は意味を持ちます。たとえば c1 = 'Test' とした場合、c1 が VARCHAR 型であるとき、値 'Test' を含む列は検出されません。

[LONGVARCHAR、NLONGVARCHAR、および LONGVARBINARY の制約](#)も参照してください。

BINARY および LONGVARBINARY

- BINARY 列には、末尾のゼロが埋め込まれます。
- LONGVARBINARY 列には、末尾のゼロは埋め込まれません。
- データベース エンジン は、LONGVARBINARY 列の比較はできません。データベース エンジン は、固定長の BINARY データの比較はできます。

Zen では、1 つのテーブルで複数の LONGVARCHAR 列および LONGVARBINARY 列がサポートされます。データは、オフセットに応じてレコードの可変長部分に格納されます。データの可変長部分は、データの操作方法に応じて、データの列順とは異なるものにすることができます。次の例で考えてみましょう。

```
CREATE TABLE BlobDataTest
(
  Nbr    UINT,           // 固定レコード (型 14)
  Clob1  LONGVARCHAR,    // 固定レコード (型 21)
  Clob2  LONGVARCHAR,    // 固定レコード (型 21)
  Blob1  LONGVARBINARY,  // 固定レコード (型 21)
)
```

ディスク上では、物理レコードは通常次のように見えます。

```
[ 固定データ (Nbr, Clob1header, Clob2header, Blob1header)][ClobData1][ClobData2][BlobData1]
```

列 Nbr を LONGVARCHAR 列に変更します。

```
ALTER TABLE BlobDataTest ALTER Nbr LONGVARCHAR
```

これで、物理レコードはディスク上で次のように見えるようになりました。

```
[ 固定データ (Nbrheader, Clob1header, Clob2header, Blob1header)][ClobData1][ClobData2][BlobData1]
[NbrClobData]
```

見たとおり、データの可変長部分は既存のデータの列順には入りません。

しかし、新しく挿入したレコードについては、データの可変長部分は既存のデータの列順に入ります。これは、すべての列にデータが割り当てられている（列がヌルでない）ことを前提としています。

```
[ 固定データ (Nbrheader, Clob1header, Clob2header, Blob1header)][NbrClobData][ClobData1][ClobData2]
[BlobData1]
```

[LONGVARCHAR、NLONGVARCHAR、および LONGVARBINARY の制約](#)も参照してください。

LONGVARCHAR、NLONGVARCHAR、および LONGVARBINARY の制約

LONGVARCHAR および LONGVARBINARY データ型には次の制約が適用されます。

- LIKE 述部は、列データの最初の 65500 バイトに適用されます。
- その他の述部はすべて、列データの最初の 256 バイトに適用されます。
- GROUP BY、DISTINCT、および ORDER BY を伴った SELECT ステートメントはすべてのデータを返しますが、列データの最初の 256 バイトにのみ指示を行います。
- LONGVARCHAR および LONGVARBINARY 型の列には最大 2 GB までデータを挿入できますが、INSERT ステートメントでリテラル値を使用すると、挿入可能なバイト数は 15000 に減ります。ただし、パラメーター化された INSERT を使用することによって、15000 バイト以上挿入することは可能です。
- Zen による 1 回の呼び出しで、LONGVARCHAR、NLONGVARCHAR、または LONGVARBINARY 列に対して返される最大のバイト数は、アプリケーションが使用するアクセス方法によって異なります。ほとんどの場合、この制限は 65500 バイトです。詳細については、特定の開発環境用のドキュメントを参照してください。

DATETIME

DATETIME データ型は日付と時刻の値を表します。この型は、内部的に 2 つの 4 バイト整数として格納されます。最初の 4 バイトは基準日である 1900 年 1 月 1 日より前または後の日数を格納します。あとの 4 バイトは、深夜 0 時からのミリ秒数で表すその日の時刻を格納します。

DATETIME データ型はインデック付けすることができます。DATETIME の精度は 1 ミリ秒です。

DATETIME はリレーショナル データ型のみです。対応する Btrieve データ型はありません。

DATETIME の書式

DATETIME の書式は YYYY-MM-DD HH:MM:SS.mmm のみです。ミリ秒部分を切り捨てる必要がある場合は、CONVERT 関数にそれを行うためのパラメーターがあります。次の表に、DATETIME のデータ コンポーネントとそれらの値の範囲を示します。

コンポーネント	有効な値
Year (YYYY)	1753 ~ 9999
Month (MM)	01 ~ 12
Day (DD)	01 ~ 31
Hour (HH)	00 ~ 23
Minute (MM)	00 ~ 59
Second (SS)	00 ~ 59
Millisecond (mmm)	000 ~ 999

日付と時刻のデータ型の互換性

日付や時刻のデータ型を含む加算または減算を実行する必要がある場合は、スカラー関数の TIMESTAMPADD()、DATEADD()、TIMESTAMPDIFF()、および DATEDIFF() を使用することをお勧めします。式に新しい AUTOTIMESTAMP データ型および TIMESTAMP2 データ型が含まれている場合には、これらの関数の使用が必要になります。他のデータ型は、場合によっては演算子と共に式で直接使用できます。たとえば、次のステートメントは有効です。

```
SELECT "Start_Date" + 5 FROM "Class"
SELECT "Finish_Time" - "Start_Time" FROM "Class"
SELECT current_timestamp() - "Log" FROM "Billing"
```

クエリによっては、" 互換性のない型 " または " 式にエラーがあります " というメッセージを返す可能性があります。たとえば、互換性のない値を加算または減算しようとするクエリや、結果が有効にならないクエリがこれに当たります。たとえば、次のステートメントはそのようなエラーを返します。

```
SELECT "Start_Date" + 5.0 FROM "Class"
SELECT "Start_Time" + "Finish_Time" FROM "Class"
SELECT current_timestamp() + "Log" FROM "Billing"
```

CONVERT 関数および CAST 関数は、次の表に示すように DATE、DATETIME、TIME、および TIMESTAMP と使用できます。

変換元	許可される結果のデータ型（変換先）
AUTOTIMESTAMP	SQL_CHAR、SQL_DATE、SQL_TIME、SQL_TIMESTAMP、SQL_VARCHAR
DATE	SQL_CHAR、SQL_DATE、SQL_TIMESTAMP、SQL_VARCHAR
DATETIME	GUID、BINARY、および LONGVARBINARY 以外 の、サポートされる CONVERT データ型。CONVERT の <i>type</i> パラメーターには、プレフィックス "SQL_" が必要です。 CONVERT (exp, type [, style]) を参照してください。
TIME	SQL_CHAR、SQL_TIME、SQL_TIMESTAMP、SQL_VARCHAR
TIMESTAMP	SQL_CHAR、SQL_DATE、SQL_TIME、SQL_TIMESTAMP、SQL_VARCHAR
TIMESTAMP2	SQL_CHAR、SQL_DATE、SQL_TIME、SQL_TIMESTAMP、SQL_VARCHAR
VARCHAR	SQL_CHAR、SQL_DATE、SQL_TIME、SQL_TIMESTAMP、SQL_VARCHAR

メモ： CONVERT 関数には DATETIME のミリ秒部分を切り捨てることのできるオプション パラメーターがあります。[変換関数](#)の Convert 関数を参照してください。

キャスト元	許可される結果のデータ型（変換先）
AUTOTIMESTAMP	DATE、DATETIME、TIME、TIMESTAMP、TIMESTAMP2、VARCHAR

キャスト元	許可される結果のデータ型（変換先）
DATE	DATE、DATETIME、TIMESTAMP、VARCHAR
DATETIME	任意のリレーショナル データ型
TIME	TIME、DATETIME、TIMESTAMP、TIMESTAMP2、VARCHAR
TIMESTAMP	DATE、DATETIME、TIME、TIMESTAMP、TIMESTAMP2、VARCHAR
TIMESTAMP2	DATE、DATETIME、TIME、TIMESTAMP、TIMESTAMP2、VARCHAR
VARCHAR	DATE、DATETIME、TIME、TIMESTAMP、TIMESTAMP2

UNIQUEIDENTIFIER

UNIQUEIDENTIFIER データ型は GUID（Globally Unique Identifier：グローバル一意識別子）として知られている 16 バイトのバイナリ値です。GUID は、行がほかの行と重複しない場合に有用です。

UNIQUEIDENTIFIER は、9.5 以降のファイル形式を必要とします。

UNIQUEIDENTIFIER の列またはローカル変数は、以下の方法で初期化できます。

- NEWID() スカラー関数を使用します。NEWID() を参照してください。
- 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx' 形式の引用符で囲んだ文字列を指定します。x は 0-9 または A-F の範囲の 16 進数です。たとえば、'1129619D-772C-AAAB-B221-00FF00FF0099' は有効な UNIQUEIDENTIFIER 値です。

引用符で囲んだ文字列を指定する場合、32 桁全部を指定する必要があります。データベース エンジン は部分文字列の埋め込み処理を行いません。

以下の比較演算子のみが UNIQUEIDENTIFIER に使用できます。

演算子	説明
=	等しい
<> または !=	等しくない
<	より小さい
>	より大きい

演算子	説明
<=	小さいかまたは等しい
>=	大きいかまたは等しい
IS NULL	値はヌルです
IS NOT NULL	値はヌルではありません

2つの値のビット パターンの比較によっては並べ替えは行われません。

変数を宣言する

SET ステートメントを使用して UNIQUEIDENTIFIER データ型の変数を宣言することができます。

```
DECLARE :Cust_ID UNIQUEIDENTIFIER DEFAULT NEWID()
DECLARE :ISO_ID uniqueidentifier
SET :ISO_ID = '1129619D-772C-AAAB-B221-00FF00FF0099'
```

UNIQUEIDENTIFIER を別のデータ型に変換する

UNIQUEIDENTIFIER は CAST または CONVERT スカラー関数を使用して以下のデータ型のいずれかに変換できます。

- CHAR
- LONGVARCHAR
- VARCHAR

変換の例については、[変換関数](#)を参照してください。

無限の表現

Zen で無限を表すには、次の表のように、4 バイト（C 言語の float 型）または 8 バイト（C 言語の double 型）の形式で、16 進数または文字として表現できます。

値	Float 16 進数	Float 文字	Double 16 進数	Double 文字
正の最大数			0x7FEFFFFFFFFFFFFFFF	
負の最大数			0xFFEFFFFFFFFFFFFFFF	

値	Float 16 進数	Float 文字	Double 16 進数	Double 文字
正の無限数	0x7F800000	1E999	0x7FF0000000000000	1E999
負の無限数	0xFF800000	-1E999	0xFFF0000000000000	-1E999

旧データ型

いくつかの古い（レガシー）データ型は、現行リリースの **Zen** ではサポートされていません。次の表は、旧データ型に代わって使用する新データ型を示します。

旧データ型	型コード	代替データ型	型コード
LOGICAL(1)	7	BIT	16
LOGICAL(2)	7	SMALLINT	14
LSTRING	10	VARCHAR	11
LVAR	13	LONGVARCHAR	21
NOTE	12	LONGVARCHAR	21

上記のデータ型を使用している既存のデータベースはサポートされており、正しく機能します。新しいデータベースでこれらのデータ型をサポートするには、**CREATE TABLE** または **ALTER TABLE** ステートメントを実行する前に **SET LEGACYTYPESALLOWED=ON** ステートメントを実行します。その後、**SET LEGACYTYPESALLOWED=OFF** ステートメントを実行するか、または **SQL** セッションによって **SET** ステートメントを終了させます。詳細については、[SET LEGACYTYPESALLOWED](#) を参照してください。

Btrieve キーのデータ型

このトピックでは、インデックスを作成できる **Btrieve** データ型（キー タイプ）について説明します。**MicroKernel** は内部的に、文字列キーを 1 バイトずつ左から右へ比較します。デフォルトで、**MicroKernel** は **ASCII** 値に基づいて文字列キーをソートします。しかし、文字列キーは、大文字小文字を無視したり、オルタネート コレーティング シーケンス（**ACS**）を使用するように定義することもできます。

MicroKernel は、符号なしバイナリ キーを一度に 1 WORD ずつ比較します。Intel 8086 プロセッサ ファミリは Integer の上位バイトと下位バイトを反転させるため、MicroKernel はこれらのキーを右から左へ比較します。

特定のデータ型が複数のサイズで利用できる（たとえば、4 バイトと 8 バイトの FLOAT 値が利用できる）場合は、（新しいキーの作成に使用される）キー長パラメーターにより、そのキーのすべての値に適用されるサイズが定義されます。許可されていないキーの長さを使ってキーを定義しようとすると、ステータス 29（キー長が無効）になります。

次の表は、キー タイプとそれに関連付けられたコードの一覧を示します。表に続いて、各キー タイプの内部記憶形式について説明します。

データ型	型コード	データ型	型コード
AUTOINCREMENT	15	NUMERIC	8
AUTOTIMESTAMP	32	NUMERICSA	18
BFLOAT	9	NUMERICSLB	28
STRING	0	NUMERICSLS	29
CURRENCY	19	NUMERICSTB	31
DATE	3	NUMERICSTS	17
DECIMAL	5	TIME	4
FLOAT	2	TIMESTAMP	20
GUID	27	TIMESTAMP2	34
INTEGER	1	UNSIGNED BINARY	14
LOGICAL	7	WSTRING	25
LSTRING	10	WZSTRING	26
MONEY	6	ZSTRING	11

AUTOINCREMENT

AUTOINCREMENT キー タイプは、2 バイト、4 バイト、または 8 バイト長の符号付き Intel 整数です。autoincrement キーは、内部的に Intel バイナリ整数形式で格納され、

WORD 内で上位バイトと下位バイトが反転されています。MicroKernel は **autoincrement** キーをその絶対値を基に、右から左へと一度に 1 WORD ずつ別のレコードに保存されている値と比較してソートします。**autoincrement** キーを使用すると、ファイルにレコードを挿入するとき、既存の最大の値より 1 大きい値を自動的に割り当てることができます。値は絶対値を基にソートされるため、データ型が符号付きと考えた場合、可能性のあるレコード数は期待するレコード数のおおよそ半分になります。

ファイルから削除された値は、自動的に再使用されません。挿入または更新でゼロ (0) 値を入力したら次に来る値を割り当てるようにデータベース エンジンに指示している場合、データベースは単純に最も大きい値を調べ、その値に 1 を足した結果を挿入します。

すべてのレコードまたは一部のレコードで 1 つのフィールドの値を 0 に初期化し、後から **AUTOINCREMENT** タイプのインデックスを追加できます。この機能により、必要になるまで実際にインデックスを作成しなくても **autoincrement** キーの準備をすることができます。

インデックスを追加すると、MicroKernel は各フィールドの 0 値を適切に変更します。値の番号付けは、そのフィールドで現在定義されている最大値に 1 を足した値から始まります。フィールドに 0 以外の値が存在する場合、MicroKernel はそれらを変更しません。ただし、0 以外の重複する値がフィールドに存在する場合は、MicroKernel はエラー ステータス コードを返します。

MicroKernel は、**autoincrement** キーを含んでいる各オープン ファイルに関連付けられる自動インクリメント値の、以前に使用した最大値を保持します。この値は、**autoincrement** フィールドに ASCII ゼロを含むレコードについて **INSERT** オペレーションが発生した場合にのみ、確立されて増加します。キー ページの並行性を利用して同時変更が行えるように、値はすべてのクライアントから使用されます。

ファイルの次の **autoincrement** 値は、前の **autoincrement** 値を使った **INSERT** が発生するたびに増加されます。これは、**INSERT** がトランザクション内にあるかどうか、また変更がコミットされるかどうかに関係なく起こります。

ただし、以下の項目がすべて真の場合には、**INSERT** 中にこの値を小さくすることができます。

- キー内にある最大の **autoincrement** 値が、ファイルに対する次の **autoincrement** 値よりも小さい。
- ほかのどのクライアントも、最大の **autoincrement** 値を含んでいるページに影響を与えるトランザクションを保留にしていない。

-
- 最大の `autoincrement` 値を含むキー ページが、`INSERT` を行っているクライアントによって保留にされていない。

つまり、1 トランザクション内の最初の `INSERT` のみが、次回使用可能な `autoincrement` 値を小さくすることができます。その後、次回使用可能な `autoincrement` 値は増加し続けます。

例を示して、`autoincrement` 値を小さくできる方法をわかりやすくしましょう。レコード 1、2、3 および 4 を含む自動インクリメント ファイルがあるとします。次回使用可能な `autoincrement` 値は 5 です。

クライアント 1 がトランザクションを開始し、新しいレコードを 2 件挿入すると、次回使用可能な `autoincrement` 値は 7 に増えます（クライアント 1 は値 5 と 6 を取得します）。クライアント 2 がトランザクションを開始し、また新しいレコードを 2 件挿入します。これにより、次回使用可能な `autoincrement` 値は 9 に増えます（クライアント 2 は値 7 と 8 を取得します）。

クライアント 1 がレコード 4、5、6 を削除します。次回使用可能な `autoincrement` 値は `INSERT` でしか調整されないなので、同じ値のままです。次に、クライアント 1 がコミットを行います。コミットされたバージョンのファイルには、現在レコード 1、2 および 3 が含まれています。

クライアント 2 の場合、ファイルにはレコード 1、2、3、7 および 8 が含まれます（7 と 8 はまだコミットされていません）。次にクライアント 2 がもう 1 件レコードを挿入すると、レコード 9 になり、次回使用可能な `autoincrement` 値は 10 に増えます。クライアント 2 がレコード 3、7、8、9 を削除します。クライアント 2 では、現在ファイルにはコミット済みのレコード 1 と 2 だけが含まれています。

次にクライアント 2 がもう 1 件レコードを挿入すると、レコード 10 になります。次回使用可能な `autoincrement` 値は 11 に増えます。この値は、変更を含んでいるページに保留状態のほかの変更があるため、3 に**減らされません**。

次に、クライアント 2 がトランザクションを中止します。コミットされたバージョンのファイルには、現在レコード 1、2 および 3 が含まれていますが、次回使用可能な `autoincrement` 値は 11 のままです。

どちらかのクライアントが、トランザクションの内外を問わずもう 1 件レコードを挿入すると、次回使用可能な `autoincrement` 値は 4 に**減少されます**。これは、値を小さくするのに必要な条件がすべて真であるために起こります。

自動インクリメントした値が範囲外になる場合は、ステータス コード 5 が返されます。データベース エンジン は、値の「折り返し」を試みることも、再度ゼロから始めることもしません。ただし、以前に挿入した値が削除されており、自動インクリメントのシーケンス中で欠番になっている箇所がわかる場合は、未使用値を直接挿入することができます。

制限

AUTOINCREMENT 型のキーには次の制限が適用されます。

- 重複のないキーとして定義する必要があります。
- キーをセグメント キーにすることはできません。ただし、`autoincrement` キーが個別の単一のキーとして最初に定義されており、`autoincrement` キー番号がセグメント キーのキー番号よりも小さい場合に限り、`autoincrement` キーを別のキーの整数セグメントとして含めることができます。
- ほかのキーとオーバーラップすることはできません。
- キーはすべて昇順でなければなりません。

ファイルにレコードを挿入すると、MicroKernel は `autoincrement` キー値を次のように扱います。

- `autoincrement` キーにバイナリ 0 の値を指定した場合、MicroKernel は以下の基準に沿ってキーに値を割り当てます。
 - ファイルに最初のレコードを挿入する場合、MicroKernel は `autoincrement` キーに値 1 を割り当てます。
 - ファイル内に既にレコードが存在する場合、MicroKernel は `autoincrement` キーに、ファイル内の既存の最も大きい絶対値より 1 大きい値を割り当てます。
- `autoincrement` キーに 0 以外の正の値を指定した場合、MicroKernel はファイルにレコードを挿入し、指定した値をキー値として使用します。その値を含むレコードがファイルに既に存在する場合、MicroKernel はエラー ステータス コードを返し、そのレコードを挿入しません。

AUTOTIMESTAMP

AUTOTIMESTAMP キー タイプは 8 バイトの符号なし整数で、Unix エポックに基づいてナノ秒単位で時間を追跡します。値 0 は、新しいレコードが挿入されたとき、または既存のレコードが初めて更新されたときに、この値を自動的に現在の時刻に置き換える

よう、データベース エンジンに指示します。0 以外の値も指定でき、その場合は 1970 UTC からのナノ秒数として解釈されます。

このキー タイプを利用できるのは Zen v14 からで、対象となるファイル形式は 9.5 から 13.0 です。古いデータベース エンジンで、このタイプを使用するレコードを持つファイルを開こうとすると、認識されない Microkernel ファイルに対するステータス コード 30 が返されます。

AUTOTIMESTAMP 値の範囲は、1970-01-01 00:00:00.0000000000 ～ 2554-07-21 23:34:33.709551615 です。

現在の Linux および Android のシステム時計は、真のナノ秒の精度を備えています。Windows では、最も高い精度はセプタ秒 (10^{-7} 秒) です。ナノ秒の精度に対応していないシステム上で AUTOTIMESTAMP キーが書き込まれた場合、値はゼロで埋められます。その結果、ナノ秒精度でないシステムで挿入または更新を行うと、値が重複する可能性があります。しかし、インデックスが一意となるよう設定されている場合には、データベース エンジンは、以前に生成された最新のタイムスタンプとの一致を検出すると、1 ナノ秒を加算します。重複は、手動で挿入されたタイムスタンプ値や、システム時計のリセットによっても生じる可能性があります。どちらの場合も、Insert (2) または Update (3) はステータス コード 5 で失敗します。

AUTOTIMESTAMP を使用した挿入と更新

Insert (2) および Update (3) オペレーションは、AUTOTIMESTAMP キーのゼロ値を処理する場合、データベース エンジン サーバーのシステム時計から現在のタイムスタンプを取得します。エンジンは、現在のレコード内で、ゼロを含むすべての AUTOTIMESTAMP キーに対してこの値を使用します。

Insert Extended (40) の場合は、エンジンはオペレーション内で指定されたレコードごとに新しいタイムスタンプ値を取得します。AUTOTIMESTAMP キーを一意とした場合、その値が同一オペレーション内で以前に生成されたタイムスタンプ値と一致する場合には、生成された値を 1 ナノ秒ずつ加算することで、レコード間でタイムスタンプが重複しないようにします。Insert (2) と同様に、レコードごとに生成されるタイムスタンプは、当該レコード内のすべての AUTOTIMESTAMP キーに使用されます。

Update Chunk (53) オペレーションで、AUTOTIMESTAMP キーを含むレコードの固定部分に書き込みを行う場合、エンジンは新しいタイムスタンプを取得しません。代わりに、提供されたキー値をそのまま受け入れ、レコードにセットします。したがって、Update Chunk オペレーションを使用するときに、AUTOTIMESTAMP キーを更新するた

めにゼロ値を指定すると、レコードにゼロ値が格納され、キーは 1970 UTC と解釈されることになります。自動生成されたタイムスタンプでキーを更新したい場合は、Update (3) オペレーションを使用してレコードの固定部分を更新します。

制限

AUTOTIMESTAMP 型のキーを作成する場合には、次の制限が適用されます。

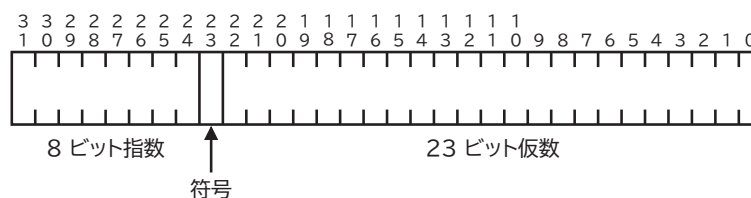
- NOCASE フラグをキーに適用することはできません。
- NULL_KEY および MANUAL_KEY は指定できません。タイムスタンプをインデックスから除外することはできません。

Function Executor および Maintenance ツールでの使い方

Function Executor ツールまたは Maintenance ツールでの AUTOTIMESTAMP キーの使い方は、AUTOINCREMENT キーと似ています。このキーを使用するファイルの場合、キータイプは Atstamp として一覧に表示されます。また、Atstamp は `butil <filename> -stat` の出力に表示され、`butil -create` コマンドのディスクリプションファイルのキータイプに使用されます。

BFLOAT

BFLOAT キータイプは、単精度実数または倍精度実数です。単精度実数は、23 ビットの仮数、128 でバイアスされた 8 ビットの指数、および符号ビットで格納されます。4 バイト float の内部レイアウトは次のとおりです。



倍精度実数の表現法は、仮数が 23 ビットではなく 55 ビットである点を除いては、単精度実数の表現法と同じです。最小有効数字の 32 ビットは、バイト 0 から 3 に格納されます。

BFLOAT 型は、一般に古い BASIC アプリケーションで使用されます。Microsoft はこのデータ型を MBF (Microsoft Binary Format) と呼びますが、Visual Basic 環境でこの型は

サポートしていません。新しいデータベース定義では、BFLOAT ではなく FLOAT を使用する必要があります。

STRING

STRING キータイプは、左から右へ並んだ文字の連なりです。キー値がヌルであるかどうかを MicroKernel が判定する場合を除いて、各文字は単一バイトに ASCII 形式で表されます。STRING データは、キーのサイズいっぱいになるまで空白を埋め込む必要があります。

CURRENCY

CURRENCY ?? タイプは、8 バイトの符号付き数値を表し、Intel バイナリ整数形式で並べ替えられ格納されています。このため、その内部表現法は 8 バイトの INTEGER データ型と同じです。CURRENCY データ型には、小数点以下に 4 桁が想定されており、通貨のデータ値の分数部分を表します。

DATE

DATE キータイプは、内部的に 4 バイト値として格納されます。日と月はそれぞれ 1 バイトのバイナリ形式で格納されます。年は、年の値全体を表す 2 バイトのバイナリ数です。MicroKernel は、日を 1 番目のバイトに、月を 2 番目のバイトに、年を月に続く 2 バイトの WORD に置きます。



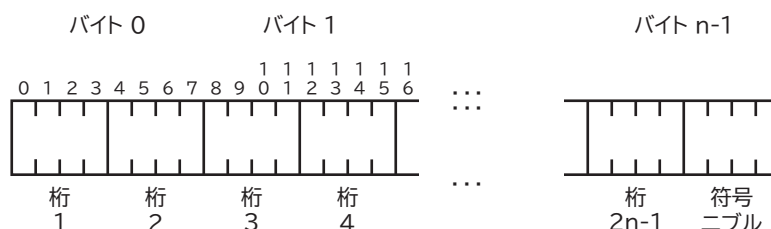
日付フィールドに使用する C 言語の構造体の例を示します。

```
TYPE dateField {  
  char day;  
  char month;  
  integer year;  
}
```

日付フィールドの year 部には、年全体の整数表現が設定されるようにする必要があります。たとえば、2001 年の場合は 2,001 です。

DECIMAL

DECIMAL キータイプは、内部的に、1 バイトごとに 2 つの 10 進数を含む、パックされた 10 進数として格納されます。 n バイトの DECIMAL フィールドの内部表現法は次のとおりです。



DECIMAL の小数点は暗黙に設定されます。DECIMAL フィールドに小数点は格納されません。DECIMAL フィールドの値の小数点位置のトラッキングはアプリケーションが行います。データベース エンジンがキーを正しく照合するために、DECIMAL キータイプの値はすべて、小数点以下の桁数を同じにする必要があります。DECIMAL 型は、一般に COBOL アプリケーションで使用されます。

8 バイトの decimal は、15 桁の数字と符号を保持することができます。10 バイトの decimal では、19 桁の数字と符号を保持できます。decimal 値は、左側の桁をゼロで埋める必要があります。

符号ニブルは、正の数の場合は 0xF または 0xC で、負の数の場合は 0xD です。デフォルトで、リレーショナル エンジンおよびそれを使用する SDK アクセス方法の場合、DECIMAL 用の正符号ニブルには常に 0xF を書き込みます。これらは読み取りオペレーションに対し 0xF および 0xC の両方を正として解釈することができます。

レジストリ (Windows レジストリおよび Zen レジストリ) の設定によって、DECIMAL 用の正符号のニブルに対してどのデータベース エンジンを使用するか制御します。デフォルトの正符号ニブルを **0xC** に変更する必要がある場合、下記のようにレジストリを編集します。

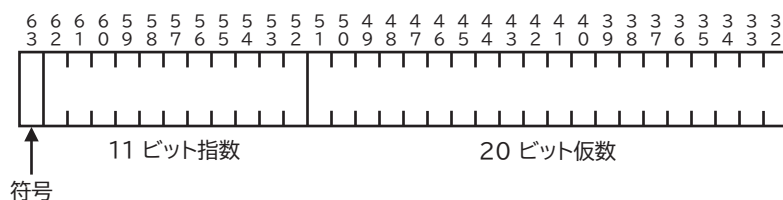
Windows

レジストリ エディターで、次のキーの CommonCOBOLDecimalSign の値を "YES" に変更します。

HKEY_LOCAL_MACHINE\SOFTWARE\Actian\Zen\SQL Relational Engine

8 バイトの FLOAT キーは、52 ビットの仮数、1023 でバイアスされた 11 ビットの指数、および符号ビットを持ちます。内部形式は次のとおりです。

バイト 7-4:



バイト 3-0:



GUID

GUID キー タイプは 16 バイトの数字で、内部的には 16 バイトのバイナリ値として格納されます。拡張データ型の値は 27 です。

通常、GUID はグローバルな一意識別子として使用されます。これはリレーショナル エンジンの **UNIQUEIDENTIFIER** データ型に相当します。

GUID は 9.5 以降のファイル形式を必要とすることに注意してください。

GUID キー

GUID を構成するバイトのソート順序は、10、11、12、13、14、15、8、9、6、7、4、5、0、1、2、3 という順序で比較されます。

GUID のキー セグメント長は 16 バイトにする必要があります。『*Btrieve API Guide*』の **キー仕様ブロック**を参照してください。

INTEGER

INTEGER キー タイプは、符号付き整数で、偶数バイトを保持する必要があります (1 は例外)。何桁の数字でも含むことができます。INTEGER フィールドは、内部的に

Intel バイナリ 整数形式で格納され、WORD 内で上位バイトと下位バイトが反転されています。MicroKernel は、キーを右から左へと一度に 1 WORD ずつ評価します。符号は、右端のバイトの上位ビットに格納しなければなりません。INTEGER 型は、ほとんどの開発環境でサポートされています。

長さ (バイト単位)	値の範囲
1	0 ~ 255
2	-32768 ~ 32767
4	-2147483648 ~ 2147483647
8	-9223372036854775808 ~ 9223372036854775807

LOGICAL

LOGICAL キー タイプは、1 バイト値または 2 バイト値として格納されます。MicroKernel は LOGICAL キー タイプを文字列として照合します。これにより、アプリケーションは、格納されている真または偽を表す値を判定することができます。

LSTRING

LSTRING キー タイプは、文字列の最初のバイトに文字列の長さのバイナリ表現を含む点を除いて、通常の STRING 型と同じ特性を持ちます。LSTRING キー タイプのサイズは、最大 255 バイトに制限されています。LSTRING キーのバイト 0 に格納されている長さにより、有効バイト数が判断されます。データベース エンジンでは、値をソートまたは検索する際、指定された文字列の長さを超える値はすべて無視します。LSTRING 型は、一般に古い Pascal アプリケーションで使用されます。

MONEY

MONEY キー タイプの内部表現法は DECIMAL 型と同じですが、小数点以下が 2 桁に想定されます。

NUMERIC

NUMERIC キー タイプの数字はそれぞれ 1 バイトを占めます。NUMERIC 値は、先頭に 0 を付けて右揃えされ、ASCII 文字列として格納されます。右端のバイトには、埋め込み符号が EBCDIC 値で含まれます。デフォルトで、正の NUMERIC データ型の符号値は符号なし数値です。

オプションで、正の NUMERIC データ型の符号の値をシフトするよう指定できます。次の表は、符合値のデフォルト（シフトされていない）状態とシフトされた状態の比較を示します。

桁	デフォルト（シフトなし）の符合値		シフトされた符合値	
	正	負	正	負
1	1	J	A	J
2	2	K	B	K
3	3	L	C	L
4	4	M	D	M
5	5	N	E	N
6	6	O	F	O
7	7	P	G	P
8	8	Q	H	Q
9	9	R	I	R
0	0	}	{	}

シフト形式の有効化

シフト形式を有効にするには、Zen データベース エンジンが実行されているマシンで、ある設定を手動で指定しなければなりません。設定 **DBCobolNumeric** を yes にセットします。このトピックでは、Windows 32 ビット、Linux プラットフォームでのこの設定の使用について概説します。

Windows 32 ビット

レジストリ エディターを使用して、DBCobolNumeric 設定を次のキーの文字列値として追加します。

HKEY_LOCAL_MACHINE¥SOFTWARE¥Actian¥Zen¥Database Names

DBCobolNumeric の文字列値を **yes** に設定します。

ほとんどの Windows システムで、このキーの場所は HKEY_LOCAL_MACHINE¥SOFTWARE¥Actian¥Zen ですが、HKEY_LOCAL_MACHINE¥SOFTWARE の下位以降の場所はオペレーティング システムによって異なります。

注意！ Windows レジストリを誤って編集すると、Windows を起動できなくなる可能性があります。編集は、訓練を受けた IT 担当者のみが行ってください。Actian Corporation は Windows レジストリの破損に対して責任を負いません。

データベース エンジンまたはエンジン サービスを停止して、再起動します。

Linux

bt.ini の [Database Names] エントリの下に DBCobolNumeric 設定を追加します。

```
[Database Names]
DBCobolNumeric=yes
```

デフォルトでは、**bt.ini** は /usr/local/actianzen/etc ディレクトリにあります。

データベース エンジンを停止して、再起動します。

正の NUMERIC データの符合値の整合

符合値がデフォルト（シフトされていない）形式で含まれている正の NUMERIC データが既にあるかもしれません。DBCobolNumeric を **yes** に設定した後、引き続き同じテーブルヘデータを追加すると、形式が混在することになります。データの符合値の形式を混在させたままにしておくことはお勧めできません。

形式が混在した状態を解消する、あるいは防ぐには、UPDATE ステートメントを使用して、NUMERIC 列をその列自身で更新します。たとえば、テーブル **t1** には NUMERIC データ型の列 **c1** があるとします。DBCobolNumeric を **yes** に設定後、**c1** を UPDATE TABLE **t1** SET **c1** = **c1** のように更新します。

NUMERICSA

NUMERICSA キー タイプ (NUMERIC SIGNED ASCII と呼ばれることもあります) は、COBOL データ型で、埋め込み符号が EBCDIC 値ではなく ASCII 値である点を除けば、NUMERIC と同じです。

桁	デフォルトの符号値	
	正	負
1	1 または Q	q
2	2 または R	r
3	3 または S	s
4	4 または T	t
5	5 または U	u
6	6 または V	v
7	7 または W	w
8	8 または X	x
9	9 または Y	y
0	0 または P	p

NUMERICSLB

NUMERICSLB キー タイプ (SIGN LEADING と呼ばれることもあります。COBOL コンパイラ オプション -dcb を使用します) は、COBOL データ型で、NUMERIC データ型の値と似た値を持ちます。NUMERICSLB 値は、先頭に 0 を付けて右揃えされ、ASCII 文字列として格納されます。

桁	デフォルトの符号値	
	正	負
1	1	A
2	2	B
3	3	C

桁	デフォルトの符号値	
	正	負
4	4	D
5	5	E
6	6	F
7	7	G
8	8	H
9	9	I
0	0	@

NUMERICSLS

NUMERICSLS キー タイプ (SIGN LEADING SEPARATE と呼ばれることもあります) は、COBOL データ型で、NUMERIC データ型の値と似た値を持ちます。NUMERICSLS 値は、先頭に 0 を付けて左揃えされ、ASCII 文字列として格納されます。ただし、NUMERICSLS 文字列の左端のバイトは、「+」(ASCII 0x2B) か「-」(ASCII 0x2D) のいずれかになります。この点が、右端のバイトにそのバイトの値と共に符号を埋め込む NUMERIC 値とは異なります。

NUMERICSTB

NUMERICSTB キー タイプ (SIGN TRAILING と呼ばれることもあります。COBOL コンパイラ オプション `-dcb` を使用します) は、COBOL データ型で、NUMERIC データ型の値と似た値を持ちます。NUMERICSTB 値は、先頭に 0 を付けて右揃えされ、ASCII 文字列として格納されます。

桁	デフォルトの符号値	
	正	負
1	1	A
2	2	B
3	3	C
4	4	D

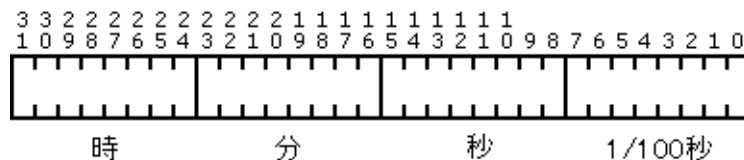
桁	デフォルトの符号値	
	正	負
5	5	E
6	6	F
7	7	G
8	8	H
9	9	I
0	0	@

NUMERICSTS

NUMERICSTS キー タイプ (SIGN TRAILING SEPARATE と呼ばれることもあります) は、COBOL データ型で、NUMERIC データ型の値と似た値を持ちます。NUMERICSTS 値は、先頭に 0 を付けて右揃えされ、ASCII 文字列として格納されます。ただし、NUMERICSTS 文字列の右端のバイトは、「+」(ASCII 0x2B) か「-」(ASCII 0x2D) のいずれかになります。この点が、右端のバイトにそのバイトの値と共に符号を埋め込む NUMERIC 値とは異なります。

TIME

TIME キー タイプは、内部的に 4 バイト値として格納されます。100 分の 1 秒、秒、分、時の値が、それぞれ 1 バイトのバイナリ形式で格納されます。MicroKernel は、100 分の 1 秒の値を最初のバイトに、それ以降のバイトにそれぞれ秒、分、時の値を置きます。データ形式は hh:mm:ss.nn です。サポートされる値の範囲は 00:00:00.00 ~ 23:59:59.99 です。



TIMESTAMP

TIMESTAMP キー タイプは日付と時刻の値を表します。SQL アプリケーションでは、このデータ型を使用して、レコードを最後に更新した現在の日時をレコードにスタンプ

します。TIMESTAMP 値は、グレゴリオ暦 0001 年 1 月 1 日の世界協定時刻 (UTC) から経過したセプタ秒 (10^{-7} 秒) を表す、8 バイトの符号なし値として格納されます。サポートされる値の範囲は、0001-01-01 00:00:00.0000000 ~ 9999-12-31 23:59:59.9999999 です。

AUTOTIMESTAMP とは異なり、値 0 は、新しいレコードが挿入されたとき、または既存のレコードが初めて更新されたときの現在の時刻に自動的に置き換えられません。

メモ： ODBC 標準に従って、CURRENT_TIMESTAMP() や NOW() などのスカラー関数は、データ型のうち小数の秒を表す部分は無視します。これらの関数を使用する際、Zen は小数の秒を無視しないで、3 桁のミリ秒を表示するという点に注意することが重要です。

TIMESTAMP は、構成要素、年、月、日、時、分、秒、およびミリ秒から成る日付と時刻の値をサポートします。次の表は、これらの各構成要素で有効な値の範囲を示しています。

YEAR	0001 ~ 9999
MONTH	01 ~ 12
DAY	01 ~ 31、グレゴリオ暦の年と月の値によって決められます。
HOURL	00 ~ 23
MINUTE	00 ~ 59
SECOND	00 ~ 59
MILLISECOND	000 ~ 999 デフォルトの設定。小数位は、0 ~ 7 (セプタ秒) の値に設定できます。

各 TIMESTAMP 値は完全な日付時刻の値を含んでいます。小数位はローカルオペレーティングシステムでサポートされる最大の桁数を持ち、必要に応じて末尾のゼロで埋められます。値が返されるとき、この値はタイムスタンプに対して設定された小数位を使用します。たとえば、小数位が 3 の場合はマイクロ秒で返され、小数位が 6 の場合はミリ秒で返されます。

日付データ型および時刻データ型の小数位の詳細については、[タイムスタンプデータ型の小数位と返される関数値](#)を参照してください。

現地時刻で **TIMESTAMP** の値を指定すると、リレーショナル エンジン はそれを UTC (Coordinated Universal Time) に変換してからレコードに格納します。**TIMESTAMP** 値を要求すると、リレーショナル エンジン は値を現地時刻に変換して返します。

注意！ データベース エンジン を実行するコンピュータのタイムゾーン情報が正しく設定されていることが重要です。タイムゾーンを移動するか、タイムゾーン情報を変更した場合、返されるデータは UTC から現地時刻に変換されるときに変わります。現地時刻および UTC の変換は、リレーショナル エンジン 内で、リレーショナル エンジン が動作している場所のタイムゾーン情報を使用して発生します。リレーショナル エンジン とタイムゾーンが異なるセッションのタイムゾーン情報は、現地時刻および UTC の変換には使用されません。

タイムスタンプ データは格納する前に変換されるため、**TIMESTAMP** 型は、データベース本体の外にあるイベントを参照する現地時刻データや現地日付データでの使用には向きません。特に、季節時間の変更が行われるタイムゾーン (米国のサマー タイム など) ではそうです。

たとえば、10 月 15 日に、11 月 15 日午前 10 時の予定を記録するタイムスタンプ値を入力したとします。タイムゾーンは U.S. 中部であるとして。リレーショナル エンジン は値を格納するとき、現在の現地時刻情報を使って値を UTC に変換します (CDT の場合、UTC - 5 時間)。したがって、時間値 15 が格納されます。11 月 1 日に予定の時刻を確認するとします。現在、お使いのコンピュータは標準時間になっています。これは、10 月にサマー タイムの切り替えが発生したためです。これにより、変換は (UTC - 6 時間) になります。予定時刻を抽出すると、現地時刻の午前 9 時 (15 UTC - 6 CST) が表示されますが、これは正しい予定時刻ではありません。

データベース エンジン をあるタイムゾーンから別のタイムゾーンに移動させた場合も、同種の問題が発生します。

リレーショナル エンジン は、**DATE** 値および **TIME** 値を UTC に変換しないので、外部データを記録する場合は、できる限りいつも **DATE** 列および **TIME** 列を使用することをお勧めします。**TIMESTAMP** 列を使用する唯一の理由は、データベースに入力したレコードの時間順を判定する固有の機能に必要なからです。

Function Executor および Maintenance ツールでの使い方

Function Executor ツールまたは Maintenance ツールでの **TIMESTAMP** キーの使い方は、**AUTOINCREMENT** キーと似ています。このキーを使用するファイルの場合、キータイプは **Tstamp** として一覧に表示されます。また、**Tstamp** は `butil <filename> -stat` の出

力に表示され、`butil -create` コマンドのディスクリプション ファイルのキー タイプに使用されます。

TIMESTAMP2

TIMESTAMP2 キー タイプは、Unix エポックに基づいてナノ秒単位で時間を追跡します。SQL アプリケーションでは、このデータ型を使用して、レコードを最後に更新した現在の日時をレコードにスタンプします。値は、グレゴリオ暦 1970 年 1 月 1 日の世界協定時刻 (UTC) から経過したナノ秒 (10^9 秒) を表す、8 バイトの符号なし値として格納されます。サポートされる値の範囲は、1970-01-01 00:00:00.000000000 ~ 2554-07-21 23:34:33.709551615 です。

AUTOTIMESTAMP とは異なり、値 0 は、新しいレコードが挿入されたとき、または既存のレコードが初めて更新されたときの現在の時刻に自動的に置き換えられません。

このキー タイプを利用できるのは Zen v14 SP1 からで、対象となるファイル形式は 9.5 から 13.0 です。古いデータベース エンジンで、このタイプを使用するレコードを持つファイルを開こうとすると、認識されない Microkernel ファイルに対するステータス コード 30 が返されます。

メモ： ODBC 標準に従って、`CURRENT_TIMESTAMP()` や `NOW()` などのスカラー関数は、データ型のうち小数の秒を表す部分は無視します。これらの関数を使用する際、Zen は小数の秒を無視しないで、9 桁のナノ秒を表示するという点に注意することが重要です。

TIMESTAMP2 は、構成要素、年、月、日、時、分、秒、およびナノ秒から成る日付と時刻の値をサポートします。次の表は、これらの各構成要素で有効な値の範囲を示しています。

YEAR	1970 ~ 2554
MONTH	01 ~ 12
DAY	01 ~ 31、グレゴリオ暦の年と月の値によって決められます。
HOURL	00 ~ 23
MINUTE	00 ~ 59
SECOND	00 ~ 59

NANOSEC 000000000 ~ 999999999 デフォルトの設定。
OND

各 TIMESTAMP2 値は完全な日付時刻の値を含んでいます。小数位はローカル オペレーティング システムでサポートされる最大の桁数を持ち、必要に応じて末尾のゼロで埋められます。値が返される時、この値はタイムスタンプに対して設定された小数位を使用します。たとえば、小数位が 3 の場合はマイクロ秒で返され、小数位が 6 の場合はミリ秒で返されます。

日付データ型および時刻データ型の小数位の詳細については、[タイムスタンプ データ型の小数位と返される関数値](#)を参照してください。

現地時刻で TIMESTAMP2 の値を指定すると、リレーショナル エンジンはその値を UTC (Coordinated Universal Time) に変換してからレコードに格納します。TIMESTAMP2 値を要求すると、リレーショナル エンジンはその値を現地時刻に変換して返します。

注意！ データベース エンジンを実行するコンピューターのタイムゾーン情報が正しく設定されていることが重要です。タイムゾーンを移動するか、タイムゾーン情報を変更した場合、返されるデータは UTC から現地時刻に変換されるときに変わります。現地時刻および UTC の変換は、リレーショナル エンジン内で、リレーショナル エンジンが動作している場所のタイムゾーン情報を使用して発生します。リレーショナル エンジンとタイムゾーンが異なるセッションのタイムゾーン情報は、現地時刻および UTC の変換には使用されません。

タイムスタンプ データは格納する前に変換されるため、TIMESTAMP2 型は、データベース本体の外にあるイベントを参照する現地時刻データや現地日付データでの使用には向きません。特に、季節時間の変更が行われるタイムゾーン（米国のサマー タイムなど）ではそうです。

たとえば、10 月 15 日に、11 月 15 日午前 10 時の予定を記録するタイムスタンプ値を入力したとします。タイムゾーンは U.S. 中部であるとして、リレーショナル エンジンはその値を格納するとき、現在の現地時刻情報を使ってその値を UTC に変換します (CDT の場合、UTC - 5 時間)。したがって、時間値 15 が格納されます。11 月 1 日に予定の時刻を確認するとします。現在、お使いのコンピューターは標準時間になっています。これは、10 月にサマー タイムの切り替えが発生したためです。これにより、変換は (UTC - 6 時間) になります。予定時刻を抽出すると、現地時刻の午前 9 時 (15 UTC - 6 CST) が表示されますが、これは正しい予定時刻ではありません。

データベース エンジンがあるタイム ゾーンから別のタイム ゾーンに移動させた場合も、同種の問題が発生します。

リレーショナル エンジンは、DATE 値および TIME 値を UTC に変換しないので、外部データを記録する場合は、できる限りいつも DATE 列および TIME 列を使用することをお勧めします。TIMESTAMP2 列を使用する唯一の理由は、データベースに入力したレコードの時間順を判定する固有の機能に必要なからです。

Function Executor および Maintenance ツールでの使い方

Function Executor ツールまたは Maintenance ツールでの TIMESTAMP2 キーの使い方は、AUTOINCREMENT キーと似ています。このキーを使用するファイルの場合、キー タイプは TS2 として一覧に表示されます。また、TS2 は `butil <filename> -stat` の出力に表示され、`butil -create` コマンドのディスクリプション ファイルのキー タイプに使用されます。

UNSIGNED BINARY

UNSIGNED BINARY キーは、キーの最大長である 255 バイトまでであれば何バイトにでもできます。UNSIGNED キーは、上位バイトから下位バイトへと 1 バイトごとに比較されます。キーの最初のバイトが下位バイトです。キーの最後のバイトが上位バイトです。

データベース エンジンは、UNSIGNED BINARY キーを符号なしの INTEGER キーとしてソートします。これらのキーの違いは、INTEGER には符号ビットがあるのに対し、UNSIGNED BINARY タイプにはないという点と、UNSIGNED BINARY キーは 4 バイトより長くすることができるという点です。

WSTRING

WSTRING は、ヌルで終わらない Unicode 文字列です。文字列の長さはフィールドの長さで決まります。

WZSTRING

WZSTRING は、2 つのヌルで終わる Unicode 文字列です。文字列の長さは、フィールド内の Unicode NULL (2 ヌル バイト) の位置で決まります。これは、Btrieve でサポートされる ZSTRING 型に対応します。

ZSTRING

ZSTRING キー タイプは、C 文字列に対応します。ZSTRING 型は、バイナリ 0 を含むバイトで終わるという点以外は、通常の文字列型と同じ特性を持ちます。MicroKernel は、キー値がヌルかどうかを判定する場合を除き、ZSTRING 内で見つけた最初のバイナリ 0 より後の値をすべて無視します。

ZSTRING 型の最大長は、ヌル終端文字を含めて、255 バイトです。ヌル値を許可する列のキーとして使用する場合、文字列の先頭 254 バイトのみがキーに使用されます。このわずかな制限は、キーの合計長が 255 バイトに制限されていることにより生じるもので、1 バイトは列のヌル インジケータが占めるため、残り 254 バイトだけがキー値になります。

キーでないデータ型

このトピックでは、インデックスを設定できない (Btrieve キーとして使用できない) データ型の内部記憶形式について説明します。

BLOB

バイナリ ラージ オブジェクト (BLOB) タイプは、最大 2 GB までのサイズのバイナリ データ フィールドのサポートを提供します。このタイプは 2 つの部分から成ります。

- レコードの固定長部分での 8 バイト ヘッダー。ヘッダーには、レコードの可変長部分におけるデータの開始位置のオフセットを示す 4 バイト整数と、そのデータのサイズを示す 4 バイト整数が含まれます。
- レコードの可変長部分内に格納されているバイナリ データ自体。すべての BLOB および CLOB フィールドのサイズの合計は 2 GB 以下である必要があります。これは、レコードの可変長部分へのオフセット ポインターが、最大 2 GB オフセットに制限されているためです。最大サイズ 2 GB の BLOB を格納するには、レコード中に BLOB または CLOB フィールドは 1 つしか定義できません。

詳細については、[BINARY](#) および [LONGVARBINARY](#)、[LONGVARCHAR](#)、[NLONGVARCHAR](#)、および [LONGVARBINARY](#) の制約を参照してください。

CLOB

文字ラージオブジェクト (CLOB) タイプは、最大 2 GB までのサイズの文字列データフィールドのサポートを提供します。このタイプは 2 つの部分から成ります。

- レコードの固定長部分での 8 バイト ヘッダー。ヘッダーには、レコードの可変長部分におけるデータの開始位置のオフセットを示す 4 バイト整数と、そのデータのサイズ (バイト単位) を示す 4 バイト整数が含まれます。
- レコードの可変長部分内に格納されている文字データ自体。すべての BLOB および CLOB フィールドのサイズの合計は 2 GB 以下である必要があります。これは、レコードの可変長部分へのオフセット ポインターが、最大 2 GB オフセットに制限されているためです。最大サイズ 2 GB の BLOB を格納するには、レコード中に BLOB または CLOB フィールドは 1 つしか定義できません。

詳細については、[CHAR](#)、[NCHAR](#)、[VARCHAR](#)、[NVARCHAR](#)、[LONGVARCHAR](#)、および [NLONGVARCHAR](#)、[LONGVARCHAR](#)、[NLONGVARCHAR](#)、および [LONGVARBINARY](#) の制約を参照してください。

B. SQL の予約語

予約語には SQL キーワードと、それ以外の、リレーショナル エンジンによって処理されるときに特別な意味を持つ記号があります。予約語をデータベースや、テーブル、列、変数、またはその他のオブジェクトの名前に使用することはお勧めしません。予約語をオブジェクト名に使用する場合は、その語句が指定した文脈中でキーワードとして使用されていないことをリレーショナル エンジンに知らせるために、二重引用符で囲む必要があります。

ユーザー定義のオブジェクト名を常に二重引用符で囲むことで、予約語のことを心配しなくても済むようになります。

この付録には次のセクションがあります。

- [予約語](#)
- [使用を避ける語句](#)

予約語

以下に挙げるそれぞれの記号または語句は、二重引用符で区切られていない限り、リレーショナル エンジンによって処理される際に特別な意味を持ちます。これらの語句の 1 つを引用符を付けずにテーブル名またはオブジェクト名に使用すると、エラーが発生します。

次のトピック、[使用を避ける語句](#)も参照してください。

記号

#	;	:	@

A

ABORT	ACCELERATED	ADD
AFTER	ALL	ALTER
AND	ANSI_PADDING	ANY
AS	ASC	ATOMIC
AVG		

B

BEFORE	BEGIN	BETWEEN
BORDER	BY	

C

CALL	CACHED_PROCEDURES	CASCADE
CASE	CAST	CHECK
CLOSE	COALESCE	COLLATE
COLUMN	COMMIT	COMMITTED
CONSTRAINT	CONVERT	COUNT
CREATE	CREATE SP	CREATETAB
CREATEVIEW	CROSS	CS
CURDATE	CURRENT	CURSOR
CURTIME		

D

DATA_PATH	DATABASE	DATETIMEINMILLISECONDS
DBO	DBSEC_AUTHENTICATION	DBSEC_AUTHORIZATION

DCOMPRESS	DDF	DECIMALSEPARATORCOMMA
DECLARE	DEFAULT	DEFAULTCOLLATE
DELETE	DENY	DESC
DIAGNOSTICS	DICTIONARY	DICTIONARY_PATH
DISTINCT	DO	DROP
DSN		

E

EACH	ELSE	ENCODING
END	ENFORCED	EX
EXCLUSIVE	EXEC	EXECUTE
EXISTING	EXISTS	EXPR

F

FETCH	FILES	FN
FOR	FOREIGN	FROM
FULL	FUNCTION	

G

GLOBAL_QRYPLAN	GRANT	GROUP

H

HANDLER	HAVING

I

IF	IN	INDEX
INNER	INOUT	INSERT
INTEGRITY	INTERNAL	INTO
IS	ISOLATION	

J

JOIN

K

KEY

L

LEAVE	LEFT	LEGACYOWNERNAME
LEVEL	LIKE	LIMIT
LINKDUP	LOGIN	LOOP

M

MAX	MIN	MODE
MODIFIABLE	MODIFY	

N

NEW	NEXT	NO
NO_REFERENTIAL_INTEGRITY	NORMAL	NOT
NOW	NULL	

O

OF	OFF	OFFSET
OLD	ON	ONLY
OPEN	OPTINNERJOIN	OR
ORDER	OUT	OUTER
OVER	OWNER	

P

PAGESIZE	PARTIAL	PARTITION
PASSWORD	PCOMPRESS	PRECEDING
PRED	PRIMARY	PRINT
PROCEDURE	PROCEDURES_CACHE	PSQL_MOVE
PSQL_PHYSICAL	PSQL_POSITION	PUBLIC

Q

QRYPLAN	QRYPLANOUTPUT

R

READ	REFERENCES	REFERENCING
RELATIONAL	RELEASE	RENAME
REPEAT	REPEATABLE	REPLACE
RESTRICT	RETURN	RETURNS
REUSE_DDF	REVERSE	REVOKE
RIGHT	ROLLBACK	ROW
ROWS	ROWCOUNT	ROWCOUNT2

S

SAVEPOINT	SECURITY	SELECT
SERIALIZABLE	SESSIONID	SET
SIGNAL	SIZE	SPID
SQLSTATE	SSP_EXPR	SSP_PRED
START	STDEV	SUM
SVBEGIN	SVEND	

T

T	TABLE	THEN
TO	TOP	TRANSACTION
TRIGGER	TRIGGERSTAMPMISC	TRUEBITCREATE
TRUENULLCREATE	TRY_CAST	TS

U

UNBOUNDED	UNCOMMITTED	UNION

UNIQUE	UNIQUEIDENTIFIER	UNMASK
UNTIL	UPDATE	USER
USING		

V

V1_METADATA	V2_METADATA	VALUES
VIEW		

W

WHEN	WHERE	WHILE
WITH	WORK	WRITE

使用を避ける語句

次の表は、SQL-92 および SQL-99 ANSI 標準のキーワードに加え、Zen で認識される追加のキーワードの一覧を示しています。これらの語句は、二重引用符で囲まない限り、テーブルや列、その他のオブジェクトの名前での使用を避けることをお勧めします。Actian Corporation は、以下のキーワードだけでなく、将来のリリースで今後追加される ANSI SQL キーワードのサポートも追加する権利を有しています。追加されたら、それらのキーワードもこの一覧に含まれるようになります。

二重引用符を使ってすべてのテーブル、列、およびユーザー定義オブジェクトの名前を区切っておけば、将来、予約語と競合する可能性について心配する必要はありません。

[予約語](#) トピックも参照してください。

ABSOLUTE	ACTION	ADD
ALL	ALLOCATE	ALTER
AND	ANY	ARE
AS	ASC	ASSERTION
AT	AUTHORIZATION	AVG
BEGIN	BETWEEN	BIGIDENTITY
BIT	BIT_LENGTH	BOTH
BY	CASCADE	CASCADE

CASE	CAST	CATALOG
CHAR	CHARACTER	CHAR_LENGTH
CHARACTER_LENGTH	CHECK	CLOSE
COALESCE	COLLATE	COLLATION
COLUMN	COMMIT	CONNECT
CONNECTION	CONSTRAINT	CONSTRAINTS
CONTINUE	CONVERT	CORRESPONDING
COUNT	CREATE	CROSS
CURRENT	CURRENT_DATE	CURRENT_TIME
CURRENT_TIMESTAMP	CURRENT_USER	CURSOR
DATE	DAY	DEALLOCATE
DEC	DECIMAL	DECLARE
DEFAULT	DEFERRABLE	DEFERRED
DELETE	DESC	DESCRIBE
DESCRIPTOR	DIAGNOSTICS	DISCONNECT
DISTINCT	DOMAIN	DOUBLE
DROP	ELSE	END
END-EXEC	ESCAPE	EXCEPT
EXCEPTION	EXEC	EXECUTE
EXISTS	EXTERNAL	EXTRACT
FALSE	FETCH	FIRST
FLOAT	FOR	FOREIGN
FOUND	FROM	FULL
FUNCTION	GET	GLOBAL
GO	GOTO	GRANT
GROUP	HAVING	HOURL
IDENTITY	IMMEDIATE	IN
INDICATOR	INITIALLY	INNER
INPUT	INSENSITIVE	INSERT
INT	INTEGER	INTERSECT
INTERVAL	INTO	IS
ISOLATION	JOIN	KEY
LANGUAGE	LAST	LEADING
LEFT	LEVEL	LIKE
LIMIT	LOCAL	LOWER

MASK	MATCH	MAX
MIN	MINUTE	MODULE
MONTH	NAMES	NATIONAL
NATURAL	NCHAR	NEXT
NO	NOT	NLONGVARCHAR
NULL	NULLIF	NUMERIC
NVARCHAR	OCTET_LENGTH	OF
OFFSET	ON	ONLY
OPEN	OPTION	OR
ORDER	OUTER	OUTPUT
OVERLAPS	PAD	PARTIAL
PASSWORD	POSITION	PRECISION
PREPARE	PRESERVE	PRIMARY
PRIOR	PRIVILEGES	PROCEDURE
PUBLIC	READ	REAL
REFERENCES	RELATIVE	RESTRICT
REVERSE	REVOKE	RIGHT
ROLLBACK	ROWS	SCHEMA
SCROLL	SECOND	SECTION
SELECT	SESSION	SESSION_USER
SET	SIZE	SMALLIDENTITY
SMALLINT	SOME	SPACE
SQL	SQLCODE	SQLERROR
SQLSTATE	STDEV	SUBSTRING
SUM	SYSDATETIME	SYSUTCDATETIME
SYSTEM_USER	TABLE	TEMPORARY
THEN	TIME	TIMESTAMP
TIMESTAMP2	TIMEZONE_HOUR	TIMEZONE_MINUTE
TO	TRAILING	TRANSACTION
TRANSLATE	TRANSLATION	TRIM
TRUE	TRY_CAST	UNION
UNIQUE	UNKNOWN	UPDATE
UPPER	USAGE	USER
USING	VALUE	VALUES
VARCHAR	VARYING	VIEW

WHEN
WITH
YEAR

WHENEVER
WORK
ZONE

WHERE
WRITE

C. システム テーブル

以下のトピックでは、Zen のシステム テーブルについて説明します。

- [概要](#)
- [システム テーブルの構造](#)

概要

Zen およびそのコンポーネントが使用する情報は、システム テーブルと呼ばれる特殊なテーブルに保管されます。

注意！ DELETE、UPDATE や INSERT ステートメント、またはユーザー定義トリガーを使って、システム テーブルの変更を試みないでください。システム テーブルを直接変更してはいけません。

システム テーブルを直接照会するアプリケーションを記述しないでください。システム テーブルの一部の列が記録されない可能性があります。次のいずれかの方法を使用すれば、システム テーブルに保管されている情報をアプリケーションで取得できます。

- [システム ストアド プロシージャ](#)
- トランザクション SQL ステートメントおよび関数
- Zen API で提供される関数

Zen API は開発者向けドキュメントに含まれています。開発コンポーネントは、リリースごとにデータベース エンジンとの互換性を保つように設計されています。システム テーブルの形式は、データベース エンジンの内部アーキテクチャ次第であり、リリースごとに変わる可能性があります。公開されていないシステム テーブルの列に直接アクセスしているアプリケーションでは、Zen の内部アーキテクチャが変更されたときにアプリケーションの修正が必要になる場合があります。

次のシステム テーブルの一覧は、関連付けられるファイルの名前が示され、辞書ファイルがメタデータ バージョン 1 の名前を使用するか、またはメタデータ バージョン 2 の名前を使用するかなど、システム テーブルの内容が識別されます。

メモ： システム テーブルに含まれる一部のデータは表示できません。たとえば、ユーザー パスワードは暗号化形式で表示されます。

システム テーブル	辞書ファイル		内容
	V1 ¹	V2 ²	
X\$Attrib	ATTRIB.DDF	PVATTRIB.DDF	列属性の定義
X\$Depend	DEPEND.DDF	PVDEPEND.DDF	テーブル、ビュー、プロシージャとトリガーとの依存関係
X\$Field	FIELD.DDF	PVFIELD.DDF	列と名前付きインデックスの定義
X\$File	FILE.DDF	PVFILE.DDF	データベース内のテーブルの名前と場所
X\$Index	INDEX.DDF	PVINDEKX.DDF	インデックスの定義
X\$Proc	PROC.DDF	PVPROC.DDF	ストアド プロシージャの定義
X\$Relate	RELATE.DDF	PVRELATE.DDF	参照整合性 (RI) の情報
X\$Rights	RIGHTS.DDF	PVRIGHTS.DDF	ユーザーとグループのアクセス権の定義
X\$Trigger	TRIGGER.DDF	PVTRIG.DDF	トリガーの情報
X\$User	USER.DDF	PVUSER.DDF	ユーザー名、グループ名、およびパスワード
X\$View	VIEW.DDF	PVVIEW.DDF	ビューの定義

¹ バージョン 1 (V1) メタデータに適用。 [Zen メタデータ](#)を参照してください。

² バージョン 2 (V2) メタデータに適用。 [Zen メタデータ](#)を参照してください。

データベースを作成すると、Zen によりすべてのシステム テーブルが作成されます。

これら以外に遭遇する可能性のあるシステム テーブルが 2 つあります。VARIANT.DDF と OCCURS.DDF (V1 データベースの場合) または PVVARIANT.DDF と PVOCCURS.DDF (V2 データベースの場合) です。これら 2 つのシステム ファイルは COBOL サポート用に使われるため、ユーザーの直接介入は一切必要としません。COBOL 用ユーティリティの将来のバージョンは、これらのシステム テーブルが必要でなくなった場合には、異なるアーキテクチャを実装するかもしれません。 [COBOL アプリケーションの SQL アクセス](#)も参照してください。

システム テーブルの構造

このトピックでは、システム テーブルの構造について説明します。

- [メタデータ バージョン 1 のシステム テーブル](#)
- [メタデータ バージョン 2 のシステム テーブル](#)

メタデータ バージョン 1 のシステム テーブル

X\$Attrib

X\$Attrib システム テーブルは、ATTRIB.DDF ファイルに関連付けられています。X\$Attrib には、データベース内の各列の列属性に関する情報が含まれています。つまり、定義した列属性ごとに 1 つのエントリがあります。メタデータ バージョン 1 用の X\$Attrib の構造を次の表に示します。

列名	データ型	サイズ	大小文字 無視	説明
Xa\$Id	USMALLINT	2	適用外	X\$Field の Xe\$Id に対応
Xa\$Type	CHAR	1	No	D (デフォルト) L (論理位置) O (列の照合順序) C (文字)、H (ヘッダー)、M (マスク)、R (範囲)、V (値) ¹
Xa\$ASize	USMALLINT	2	適用外	Xa\$Attr\$ に含まれるテキストの長さ
Xa\$Attr\$	LONGVARCHAR (NOTE)	<=2048	適用外	列属性を定義するテキスト

¹ 属性タイプ、C、H、M、R および V は旧形式の検証タイプで、Pervasive.SQL v7 または Scalable SQL 環境のみで有効です。Pervasive.SQL 7 より新しい Zen リリースは、D (デフォルト)、L (論理位置) および O (列の照合順序) 属性のみを使用します。

1 つの列に複数の属性を定義すると、X\$Attrib システム テーブルには、その列 ID に複数のエントリが含まれます。つまり、定義した属性ごとに 1 つのエントリが対応します。特定の列に列属性を定義しないと、X\$Attrib テーブルにその列のエントリは含まれ

ません。Xa\$Attr\$列に含まれるテキストは、Zen で定義したとおりに表示されます。前の表で説明したように、X\$Attrib テーブルにはインデックスが 1 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xa\$Id	No	適用外	Yes
0	1	Xa\$Type	No	No	No

X\$Depend

X\$Depend システム テーブルは、DEPEND.DDF ファイルに関連付けられています。X\$Depend には、テーブル、ビュー、およびプロシージャとトリガーとの依存関係に関する情報が含まれています。X\$Depend の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xd\$Trigger	CHAR	30	Yes	トリガー名。X\$Trigger 内の Xt\$Name に対応
Xd\$DependType	UNSIGNED	1	適用外	テーブルは 1、ビューは 2、プロシージャは 3
Xd\$DependName	CHAR	30	Yes	トリガーと関連付けられている依存関係の名前。X\$File の Xf\$Name、X\$View の Xv\$Name、X\$Proc の Xp\$Name のいずれかに対応しています。

X\$Depend テーブルでは、次のようにインデックスが 2 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xd\$Trigger	No	Yes	Yes
0	1	Xd\$DependType	No	適用外	Yes
0	2	Xd\$DependName	No	Yes	No
1	0	Xd\$DependType	Yes	適用外	Yes
1	1	Xd\$DependName	Yes	Yes	No

インデックス番号は、X\$Index システム テーブルの Xi\$Number 列に保管されている値に対応します。また、セグメント番号は、X\$Index システム テーブルの Xi\$Part 列に保管されている値に対応します。

X\$Field

X\$Field システム テーブルは、FIELD.DDF ファイルに関連付けられています。X\$Field には、データベースで定義されているすべての列と名前付きインデックスに関する情報が含まれています。X\$Field の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xe\$Id	USMALLINT	2	適用外	Zen によって割り当てられた、データベース内の各フィールドに固有の内部 ID
Xe\$File	USMALLINT	2	適用外	この列または名前付きインデックスが属するテーブルの ID。これは X\$File 内の Xf\$Id に対応しています。
Xe\$Name	CHAR	20	Yes	列名またはインデックス名
Xe\$DataType	UTINYINT	1	適用外	制御フィールド： 0 ～ 26：列のデータ型 227：制約名 255：インデックス名
Xe\$Offset	USMALLINT	2	適用外	テーブル内での列のオフセット。名前付きインデックスの場合はインデックス番号。オフセットは、ゼロを基準にした相対です。 インデックス番号は、X\$Index システム テーブルの Xi\$Number 列に保管されている値に対応します。
Xe\$Size	USMALLINT	2	適用外	列の長さ。フィールドに必要な内部格納領域をバイト単位で表します。 TRUE NULL フィールドの場合、ヌルバイトはサイズに含まれません。

列名	データ型	サイズ	大小文字無視	説明
Xe\$Dec	UTINYINT	1	適用外	列の小数点位置 (DECIMAL、NUMERIC、NUMERICSA、NUMERICSTS、MONEY、または CURRENCY 型の場合)。連続ビット列の場合は相対的なビット位置。AUTOTIMESTAMP、TIMESTAMP、および TIMESTAMP2 データ型の場合は小数の秒。

列名	データ型	サイズ	大小文字無視	説明
Xe\$Flags	USMALLINT	2	適用外	フラグを表す WORD。 ビット 0 は文字列データ型の大文字と小文字を区別するフラグです。 ビット 0 = 1 の場合、フィールドの大文字と小文字は区別されません。 ビット 2 = 1 の場合、フィールドでヌル値が許可されます。 Xe\$Flags のビット 3 は、Pervasive.SQL v7 の 1 バイト TINYINT (B_TYPE_INTEGER 符号なし) とリレーショナル エンジンの 1 バイト TINYINT (B_TYPE_INTEGER 符号付き) を区別するために使用されます。 ビット 3 = 1 で、Xe\$DataType = 1 かつ Xe\$Size = 1 の場合、TINYINT 列は、リレーショナル エンジンで作成された符号付きの 1 バイト TINYINT であることを示します。 ビット 3 = 0 で、Xe\$DataType = 1 かつ Xe\$Size = 1 の場合、TINYINT 列は、古い SQL エンジンで作成された符号なしの 1 バイト TINYINT であることを示します。 ビット 11 = 1 の場合、フィールドは文字の LONGVARCHAR フィールドではなく、ワイド文字の NLONGVARCHAR フィールドとして解釈されます。 ビット 12 = 1 の場合、フィールドは BINARY として解釈されます。 ビット 13 = 1 の場合、フィールドは偶数桁の精度の DECIMAL として解釈されます。

Xe\$File 列は、X\$File システム テーブルの Xf\$Id 列に対応し、テーブルとテーブル内の列のリンクになります。たとえば次のクエリは、Billing テーブルのフィールド定義をすべて順番に返します。

```
SELECT "X$Field".*
  FROM X$File,X$Field
 WHERE Xf$Id=Xe$File AND Xf$Name = 'Billing' AND Xe$DataType <= 26
ORDER BY Xe$Offset
```

Xe\$DataType 列の整数値は Zen のデータ型を表すコードです。コードについては、[Zen で使用できるデータ型](#)を参照してください。

X\$Field テーブルでは、次のようにインデックスが 5 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xe\$Id	No	適用外	No
1	0	Xe\$File	Yes	適用外	No
2	0	Xe\$Name	Yes	Yes	No
3	0	Xe\$File	No	適用外	Yes
3	1	Xe\$Name	No	Yes	No
4	0	Xe\$File	Yes	適用外	Yes
4	1	Xe\$Offset	Yes	適用外	Yes
4	2	Xe\$Dec	Yes	適用外	No

X\$File

X\$File システム テーブルは、FILE.DDF ファイルに関連付けられています。X\$File には、データベースで定義されている各テーブルについて、そのテーブル名、関連テーブルの場所、および Zen が割り当てた固有の内部 ID 番号が含まれています。X\$File の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xf\$Id	USMALLINT	2	適用外	Zen によって割り当てられた内部 ID
Xf\$Name	CHAR	20	Yes	テーブル名
Xf\$Loc	CHAR	64	No	ファイルの場所 (パス名)

列名	データ型	サイズ	大小文字無視	説明
Xf\$Flags	UTINYINT	1	適用外	ファイルフラグ。ビット 4 = 1 の場合、ファイルは辞書ファイルです。ビット 4 = 0 の場合、ファイルはユーザー定義です。ビット 6 = 1 の場合、テーブルは真のヌル値を許可する列をサポートします。
Xf\$Reserved	CHAR	10	No	予約済み

X\$File テーブルには、2 つのインデックスが定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xf\$Id	No	適用外	No
1	0	Xf\$Name	No	Yes	No

X\$Index

X\$Index システム テーブルは、INDEX.DDF ファイルに関連付けられています。X\$Index には、データベース内のテーブルで定義されているすべてのインデックスに関する情報が含まれています。X\$Index の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xi\$File	USMALLINT	2	適用外	インデックスが属するテーブルの固有の ID。これは X\$File 内の Xf\$Id に対応しています。
Xi\$Field	USMALLINT	2	適用外	インデックス列の固有の ID。これは X\$Field 内の Xe\$Id に対応しています。
Xi\$Number	USMALLINT	2	適用外	インデックス番号 (範囲 0 ~ 119)
Xi\$Part	USMALLINT	2	適用外	セグメント番号 (範囲 0 ~ 119)
Xi\$Flags	USMALLINT	2	適用外	インデックス属性フラグ

Xi\$File 列は、X\$File システム テーブルの Xf\$Id 列に対応します。また、Xi\$Field 列は、X\$Field システム テーブルの Xe\$Id 列に対応します。したがって、インデックス セグメント エントリは 1 つのファイル、1 つのフィールドにリンクされます。

Xi\$Flags 列には整数値が含まれ、インデックス属性を定義します。次の表は、ビットがバイナリ値 1 であるとき、Zen が各ビット位置を解釈する方法を示したものです。ビット位置 0 が整数の右端のビットになります。

ビット位置	対応する 10 進数の値	説明
0	1	インデックスの重複が可能
1	2	インデックスの変更が可能です。
2	4	オルタネート コレーティング シーケンス
3	8	ヌル値にはインデックスなし (SQL の真のヌルではなく、Btrieve のヌルを参照)
4	16	インデックス内でこのセグメントに別のセグメントが連結
5	32	インデックスの大小文字の区別なし
6	64	インデックスを降順に照合
7	128	ビット 0=0 の場合、インデックスは名前付きインデックスです。ビット 0=1 かつビット 7=1 の場合、インデックスは繰り返し重複キーの方法を使用します。ビット 0=1 かつビット 7=0 の場合、インデックスはリンク重複キーの方法を使用します。 LINKDUP も参照してください。リンク重複方法および繰り返し重複方法の詳細な説明については、『 <i>Advanced Operations Guide</i> 』の 重複キーの操作方法 を参照してください。
8	256	インデックスは Btrieve の拡張キー タイプ
9	512	インデックスは部分的です。
13	8192	インデックスは外部キー
14	16384	インデックスは一部の外部キーが参照する主キー

特定のインデックスの Xi\$Flags 列の値は、インデックス属性に対応する 10 進数の値の合計になります。X\$Index テーブルでは、次のようにインデックスが 3 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xi\$File	Yes	適用外	No
1	0	Xi\$Field	Yes	適用外	No
2	0	Xi\$File	No	適用外	Yes
2	1	Xi\$Number	No	適用外	Yes
2	2	Xi\$Part	No	適用外	No

インデックス番号は、X\$Index システム テーブルの Xi\$Number 列に保管されている値に対応します。インデックス番号はゼロから始まります。また、セグメント番号は、X\$Index システム テーブルの Xi\$Part 列に保管されている値に対応します。

たとえば、Billing テーブルに定義されているインデックス セグメントに関する情報を見るには、次のようなクエリを発行します。

```
SELECT Xe$Name,Xe$Offset, "X$Index".*
  FROM X$File,X$Index,X$Field
 WHERE Xf$Id=Xi$File and Xi$Field=Xe$Id and Xf$Name = 'Billing'
ORDER BY Xi$Number,Xi$Part
```

X\$Proc

X\$Proc システム テーブルは、PROC.DDF ファイルに関連付けられています。X\$Proc には、定義されているすべてのストアードプロシージャのコンパイルされた構造情報が含まれています。X\$Proc の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xp\$Name	CHAR	30	Yes	ストアード プロシージャ名
Xp\$Ver	UTINYINT	1	適用外	バージョン ID これは将来使用するために予約されています。
Xp\$Id	USMALLINT	2	適用外	0 を基数とするシーケンス番号

列名	データ型	サイズ	大小文字無視	説明
Xp\$Flags	UTINYINT	1	適用外	ストアド ステートメントは 1、ストアド プロシージャは 2、外部プロシージャは 3
Xp\$Misc	LONGVARCHAR (LVAR)	<=990	適用外	ストアド プロシージャの内部表現

X\$Proc テーブルでは、次のようにインデックスが 1 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xp\$Name	No	Yes	Yes
0	1	Xp\$Id	No	適用外	No

単一のストアド プロシージャを X\$Proc の複数のエントリに保管し、Xp\$Name によってリンクすることができます。

X\$Relate

X\$Relate システム テーブルは、RELATE.DDF ファイルに関連付けられています。

X\$Relate には、データベースで定義されている参照整合性 (RI) 制約に関する情報が含まれています。X\$Relate は最初の外部キーが作成されるときに自動的に作成され、関係が定義されます。

X\$Relate の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xr\$PIId	USMALLINT	2	適用外	主テーブル ID
Xr\$Index	USMALLINT	2	適用外	主テーブルに含まれる主キーのインデックス番号
Xr\$FIId	USMALLINT	2	適用外	従属テーブル ID
Xr\$FIndex	USMALLINT	2	適用外	従属テーブルに含まれる外部キーのインデックス番号
Xr\$Name	CHAR	20	Yes	外部キー名

列名	データ型	サイズ	大小文字無視	説明
Xr\$updateRule	UTINYINT	1	適用外	制限は 1
Xr\$deleteRule	UTINYINT	1	適用外	制限は 1、カスケードは 2
Xr\$Reserved	CHAR	30	No	予約済み

X\$Relate テーブルでは、次のようにインデックスが 5 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xr\$PId	Yes	適用外	No
1	0	Xr\$FId	Yes	適用外	No
2	0	Xr\$Name	No	Yes	No
3	0	Xr\$PId	No	適用外	Yes
3	1	Xr\$Name	No	Yes	No
4	0	Xr\$FId	No	適用外	Yes
4	1	Xr\$Name	No	Yes	No

X\$Rights

X\$Rights システム テーブルは、RIGHTS.DDF ファイルに関連付けられています。X\$Rights には、各ユーザーのアクセス権情報が含まれています。Zen は、セキュリティオプションが有効になっているときにのみ、このテーブルを使用します。X\$Rights の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xr\$user	USMALLINT	2	適用外	User ID
Xr\$table	USMALLINT	2	適用外	テーブル ID
Xr\$column	USMALLINT	2	適用外	列 ID
Xr\$Rights	UTINYINT	1	適用外	テーブルまたは列のアクセス権フラグ

Xr\$User 列は、X\$User テーブルの Xu\$Id 列に対応します。また、Xr\$Table 列は X\$File テーブルの Xf\$Id 列に、Xr\$Column 列は X\$Field テーブルの Xe\$Id 列に対応します。

メモ： テーブル アクセス権を記述するシステム テーブル内の各行では、Xr\$Column 列の値はヌルになります。

Xr\$Rights 列には整数値が含まれ、その右端 8 ビットがユーザー アクセス権を定義します。次の表は、Zen が値を解釈する方法を説明したものです。この表の値は、単一の Xr\$Rights 値に結合される場合があります。

16 進数の値	対応する 10 進数の値	説明
1	1	再編成権
0x90	144	テーブルの参照権
0xA0	160	テーブルの変更権
0x40	64	テーブルまたは列の選択権
0x82	130	テーブルまたは列の更新権
0x84	132	テーブルまたは列の挿入権
0x88	136	テーブルまたは列の削除権

10 進数の 0 はアクセス権がないことを示します。

特定のユーザーの Xr\$Rights 列の値は、そのユーザーに適用されるアクセス権に対応する 16 進数のビット単位の共通部分になります。10 進数の値の合計ではありません。

たとえば、すべてのアクセス権が割り当てられているユーザーの Xr\$Rights の値は、次のように表されます。

$$144 | 160 | 64 | 130 | 132 | 136 = 254$$

X\$Rights テーブルでは、次のようにインデックスが 3 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xr\$User	Yes	適用外	No
1	0	Xr\$User	No	適用外	Yes
1	1	Xr\$Table	No	適用外	Yes

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
1	2	Xr\$Column	No	適用外	No
2	0	Xr\$Table	Yes	適用外	Yes
2	1	Xr\$Column	Yes	適用外	No

X\$Trigger

X\$Trigger システム テーブルは、TRIGGER.DDF ファイルに関連付けられています。X\$Trigger には、データベースで定義されているトリガーに関する情報が含まれています。X\$Trigger の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xt\$Name	CHAR	30	Yes	トリガー名
Xt\$Version	USMALLINT	2	適用外	トリガーのバージョン。 Scalable SQL バージョン 4 の場合は 4
Xt\$File	USMALLINT	2	適用外	トリガーが定義されている ファイル。X\$File 内の Xf\$Id に対応
Xt\$Event	UNSIGNED	1	適用外	INSERT は 0、DELETE は 1、UPDATE は 2
Xt\$ActionTime	UTINYINT	1	適用外	BEFORE は 0、AFTER は 1
Xt\$ForEach	UTINYINT	1	適用外	ROW は 0（デフォルト）、 STATEMENT は 1
Xt\$Order	USMALLINT	2	適用外	トリガーの実行順序
Xt\$Sequence	USMALLINT	2	適用外	0 を基数とするシーケンス 番号
Xt\$Misc	LONGVARCHAR (LVAR)	<=4054	適用外	トリガーの内部表現

あまり長いトリガーは、TRIGGER.DDF 内に複数のエントリを必要とすることがあります。各エントリは Xt\$Name フィールドに同一のトリガー名を持ち、Xt\$Sequence フィールドで指定された順序で使用されます。

X\$Trigger テーブルでは、次のようにインデックスが 3 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xt\$Name	No	Yes	Yes
0	1	Xt\$Sequence	No	適用外	No
1	0	Xt\$File	No	適用外	Yes
1	1	Xt\$Name	No	Yes	Yes
1	2	Xt\$Sequence	No	適用外	No
2	0	Xt\$File	Yes	適用外	Yes
2	1	Xt\$Event	Yes	適用外	Yes
2	2	Xt\$ActionTime	Yes	適用外	Yes
2	3	Xt\$ForEach	Yes	適用外	Yes
2	4	Xt\$Order	Yes	適用外	Yes
2	5	Xt\$Sequence	Yes	適用外	No

トリガーを X\$Trigger の複数のエントリに保管して、Xt\$Name によってリンクし、Xt\$Sequence 順に並べることができます。

X\$User

X\$User システム テーブルは、USER.DDF ファイルに関連付けられています。X\$User には、各ユーザーの名前とパスワード、および各ユーザー グループの名前が含まれてい

ます。Zen は、セキュリティ オプションが有効になっているときにのみ、このテーブルを使用します。次の表は X\$User の構造を示しています。

列名	データ型	サイズ	大小文字無視	説明
Xu\$Id	USMALLINT	2	適用外	ユーザーまたはグループに割り当てられている内部 ID
Xu\$Name	CHAR	30	Yes	ユーザーまたはグループ名
Xu\$Password	CHAR	9	No	ユーザー パスワード (暗号化)
Xu\$Flags	USMALLINT	2	適用外	ユーザーまたはグループ フラグ

メモ： グループを記述する X\$User システム テーブル内の各行では、Xu\$Password の列の値はヌルになります。

Xu\$Flags 列には整数値が含まれ、その右端の 8 ビットはユーザーまたはグループ属性を定義します。次の表は、ビットがバイナリ値 1 であるとき、Zen が各ビット位置を解釈する方法を示したものです。ビット位置 0 が整数の右端のビットになります。

ビット位置	対応する 10 進数の値	説明
0	1	予約済み
1	2	予約済み
2	4	予約済み
3	8	予約済み
4	16	予約済み
5	32	予約済み
6	64	名前はグループ名
7	128	ユーザーまたはグループは、辞書内のテーブルを定義する権限を持っています。

特定のユーザーやグループの Xu\$Flags 列の値は、そのユーザーまたはグループに適用される属性に対応する 10 進数の値の合計になります。

X\$User テーブルでは、次の表に示すようにインデックスが 2 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xu\$Id	Yes	適用外	No
1	0	Xu\$Name	No	Yes	No

X\$View

X\$View システム テーブルは、VIEW.DDF ファイルに関連付けられています。X\$View には、結合テーブルの情報やビューを定義する制約条件などのビュー定義が含まれています。X\$View テーブルにクエリを実行して、辞書に定義されているビューの名前を取得することができます。

X\$View テーブルの先頭列にはビュー名が含まれます。第 2 列と第 3 列には、LVAR 列の Xv\$Misc で検出される情報の説明が含まれています。X\$View の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xv\$Name	CHAR	20	Yes	ビュー名
Xv\$Ver	UTINYINT	1	適用外	バージョン ID これは将来使用するために予約されています。
Xv\$Id	UTINYINT	1	適用外	シーケンス番号
Xv\$Misc	LONGVARCHAR (LVAR)	<=2000	適用外	Zen 内部定義

X\$View テーブルでは、次のようにインデックスが 2 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xv\$Name	Yes	Yes	No
1	0	Xv\$Name	No	Yes	Yes
1	1	Xv\$Ver	No	適用外	Yes
1	2	Xv\$Id	No	適用外	No

単一のビューを X\$View の複数のエントリに保管して、Xv\$Name によってリンクし、Xv\$Id 順に並べることができます。

メタデータ バージョン 2 のシステム テーブル

X\$Attrib

X\$Attrib システム テーブルは、PVATTRIB.DDF ファイルに関連付けられています。X\$Attrib には、データベース内の各列の列属性に関する情報が含まれています。つまり、定義した列属性ごとに 1 つのエントリがあります。次の表は X\$Attrib の構造を示しています。

列名	データ型	サイズ	大小文字無視	説明
Xa\$Id	INTEGER	4	適用外	X\$Field の Xe\$Id に対応
Xa\$Type	CHAR	4	No	D (デフォルト) L (論理位置) O (列の照合順序)
Xa\$ASize	USMALLINT	2	適用外	Xa\$Attr\$ に含まれるテキストの長さ
Xa\$Attr\$	LONGVARCHAR (NOTE)	32763	適用外	列属性を定義するテキスト

1 つの列に複数の属性を定義すると、X\$Attrib システム テーブルには、その列 ID に複数のエントリが含まれます。つまり、定義した属性ごとに 1 つのエントリが対応します。特定の列に列属性を定義しないと、X\$Attrib テーブルにその列のエントリは含まれません。Xa\$Attr\$ 列に含まれるテキストは、Zen で定義したとおりに表示されます。Xa\$Attr\$ テーブルでは、次の表に示すようにインデックスが 1 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xa\$Id	No	適用外	Yes
0	1	Xa\$Type	No	No	No

X\$Depend

X\$Depend システム テーブルは、PVDEPEND.DDF ファイルに関連付けられています。X\$Depend には、テーブル、ビュー、およびプロシージャなどのオブジェクトとトリガーとの依存関係に関する情報が含まれています。メタデータ バージョン 2 用の X\$Depend の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xd\$Trigger	CHAR	128	Yes	トリガー名。X\$Trigger 内の Xt\$Name に対応
Xd\$DependType	UTINYINT	1	適用外	テーブルは 1、ビューは 2、プロシージャは 3
Xd\$DependName	CHAR	128	Yes	トリガーと関連付けられている依存関係の名前。X\$File の Xf\$Name、X\$View の Xv\$Name、X\$Proc の Xp\$Name のいずれかに対応しています。

メタデータ バージョン 2 用の X\$Depend テーブルでは、次のようにインデックスが 2 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xd\$Trigger	No	Yes	Yes
0	1	Xd\$DependType	No	適用外	No
1	0	Xd\$DependType	Yes	適用外	Yes
1	1	Xd\$DependName	Yes	Yes	No

X\$Field

X\$Field システム テーブルは、PVFIELD.DDF ファイルに関連付けられています。X\$Field には、データベースで定義されているすべての列と名前付きインデックスに関

する情報が含まれています。メタデータバージョン 2 用の X\$Field の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xe\$Id	INTEGER	4	適用外	Zen によって割り当てられた、データベース内の各フィールドに固有の内部 ID
Xe\$File	INTEGER	4	適用外	この列または名前付きインデックスが属するテーブルの ID。これは X\$File 内の Xf\$Id に対応しています。
Xe\$Name	CHAR	128	Yes	列名またはインデックス名
Xe\$Datatype	UTINYINT	1	適用外	0 ～ 26：列のデータ型 227：制約名 255：インデックス名
Xe\$Offset	INTEGER	4	適用外	テーブル内での列のオフセット。名前付きインデックスの場合はインデックス番号。オフセットは、ゼロを基準にした相対です。 インデックス番号は、X\$Index システム テーブルの Xi\$Number 列に保管されている値に対応します。
Xe\$Size	INTEGER	4	適用外	列の長さ。フィールドに必要な内部格納領域をバイト単位で表します。
Xe\$Dec	USMALLINT	2	適用外	列の小数点位置 (DECIMAL、NUMERIC、NUMERICSA、NUMERICSTS、MONEY、または CURRENCY 型の場合)。連続ビット列の場合は相対的なビット位置。AUTOTIMESTAMP、TIMESTAMP、および TIMESTAMP2 データ型の場合は小数の秒。

列名	データ型	サイズ	大小文字無視	説明
Xe\$Flags	INTEGER	4	適用外	フラグを表す WORD。 ビット 0 は文字列データ型の大文字と小文字を区別するフラグです。 ビット 0 = 1 の場合、フィールドの大文字と小文字は区別されません。 ビット 2 = 1 の場合、フィールドでヌル値が許可されます。 Xe\$Flags のビット 3 は、Pervasive.SQL v7 の 1 バイト TINYINT (B_TYPE_INTEGER 符号なし) とリレーショナル エンジンの 1 バイト TINYINT (B_TYPE_INTEGER 符号付き) を区別するために使用されます。 ビット 3 = 1 で、Xe\$DataType = 1 かつ Xe\$Size = 1 の場合、TINYINT 列は、リレーショナル エンジンで作成された符号付きの 1 バイト TINYINT であることを示します。 ビット 3 = 0 で、Xe\$DataType = 1 かつ Xe\$Size = 1 の場合、TINYINT 列は、古い SQL エンジンで作成された符号なしの 1 バイト TINYINT であることを示します。 ビット 11 = 1 の場合、フィールドは文字の LONGVARCHAR フィールドではなく、ワイド文字の NLONGVARCHAR フィールドとして解釈されます。 ビット 12 = 1 の場合、フィールドは BINARY として解釈されます。 ビット 13 = 1 の場合、フィールドは偶数バイト精度の DECIMAL として解釈されます。

Xe\$File 列は、X\$File システム テーブルの Xf\$Id 列に対応し、テーブルとテーブル内の列のリンクになります。たとえば次のクエリは、Billing テーブルのフィールド定義をすべて順番に返します。

```
SELECT "X$Field".*
  FROM X$File,X$Field
 WHERE Xf$Id=Xe$File AND Xf$Name = 'Billing' AND Xe$DataType <= 26
ORDER BY Xe$Offset
```

Xe\$DataType 列の整数値は Zen のデータ型を表すコードです。コードについては、[Zen で使用できるデータ型](#)を参照してください。

X\$Field テーブルでは、次の表に示すようにインデックスが 5 つ定義されています。

インデックス 番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xe\$Id	No	適用外	No
1	0	Xe\$File	Yes	適用外	No
2	0	Xe\$Name	Yes	Yes	No
3	0	Xe\$File	No	適用外	Yes
3	1	Xe\$Name	No	Yes	No
4	0	Xe\$File	Yes	適用外	Yes
4	1	Xe\$Offset	Yes	適用外	Yes
4	2	Xe\$Dec	Yes	適用外	No

X\$File

X\$File システム テーブルは、PVFILE.DDF ファイルに関連付けられています。X\$File には、データベースで定義されている各テーブルについて、そのテーブル名、関連テーブルの場所、および Zen が割り当てた固有の内部 ID 番号が含まれています。メタデータバージョン 2 用の X\$File の構造を次の表に示します。

列名	データ型	サイズ	大小文字無視	説明
Xf\$Id	UINTeger	4	適用外	Zen によって割り当てられた内部 ID
Xf\$Name	CHAR	128	Yes	テーブル名
Xf\$Loc	CHAR	250	No	ファイルの場所 (パス名)

列名	データ型	サイズ	大小文字無視	説明
Xf\$Flags	INTEGER	4	適用外	ファイルフラグ。ビット 4 = 1 の場合、ファイルは辞書ファイルです。ビット 4 = 0 の場合、ファイルはユーザー定義です。ビット 6 = 1 の場合、テーブルは真のヌル値を許可する列をサポートします。
Xf\$Reserved	CHAR	16	No	予約済み

メタデータ バージョン 2 用の X\$File テーブルでは、インデックスが 2 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xf\$Id	No	適用外	No
1	0	Xf\$Name	No	Yes	No

X\$Index

X\$Index システム テーブルは、PVINDEX.DDF ファイルに関連付けられています。X\$Index には、データベース内のテーブルで定義されているすべてのインデックスに関する情報が含まれています。メタデータ バージョン 2 用の X\$Index の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xi\$File	INTEGER	4	適用外	インデックスが属するテーブルの固有の ID。これは X\$File 内の Xf\$Id に対応しています。
Xi\$Field	INTEGER	4	適用外	インデックス列の固有の ID。これは X\$Field 内の Xe\$Id に対応しています。
Xi\$Number	INTEGER	4	適用外	インデックス番号（範囲 0 ～ 119）
Xi\$Part	INTEGER	4	適用外	セグメント番号（範囲 0 ～ 119）
Xi\$Flags	INTEGER	4	適用外	インデックス属性フラグ

Xi\$File 列は、**X\$File** システム テーブルの **Xf\$Id** 列に対応します。また、**Xi\$Field** 列は、**X\$Field** システム テーブルの **Xe\$Id** 列に対応します。したがって、インデックス セグメント エントリは 1 つのファイル、1 つのフィールドにリンクされます。

Xi\$Flags 列には整数値が含まれ、インデックス属性を定義します。次の表は、ビットがバイナリ値 1 であるとき、Zen が各ビット位置を解釈する方法を示したものです。ビット位置 0 が整数の右端のビットになります。

ビット位置	対応する 10 進数の値	説明
0	1	インデックスの重複が可能
1	2	インデックスの変更が可能です。
2	4	オルタネート コレーティング シーケンス
3	8	ヌル値にはインデックスなし (SQL の真のヌルではなく、Btrieve のレガシー ヌルを参照)
4	16	インデックス内でこのセグメントに別のセグメントが連結
5	32	インデックスの大小文字の区別なし
6	64	インデックスを降順に照合
7	128	ビット 0=0 の場合、インデックスは名前付きインデックスです。ビット 0=1 かつビット 7=1 の場合、インデックスは繰り返し重複キーの方法を使用します。ビット 0=1 かつビット 7=0 の場合、インデックスはリンク重複キーの方法を使用します。 LINKDUP も参照してください。リンク重複方法および繰り返し重複方法の詳細な説明については、『 <i>Advanced Operations Guide</i> 』の 重複キーの操作方法 を参照してください。
8	256	インデックスは Btrieve の拡張キー タイプ
13	8192	インデックスは外部キー
14	16384	インデックスは一部の外部キーが参照する主キー

特定のインデックスの **Xi\$Flags** 列の値は、インデックス属性に対応する 10 進数の値の合計になります。メタデータ バージョン 1 用の **X\$Index** テーブルでは、次のようにインデックスが 3 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xi\$File	Yes	適用外	No
1	0	Xi\$Field	Yes	適用外	No
2	0	Xi\$File	No	適用外	Yes
2	1	Xi\$Number	No	適用外	Yes
2	2	Xi\$Part	No	適用外	No

インデックス番号は、**X\$Index** システム テーブルの **Xi\$Number** 列に保管されている値に対応します。インデックス番号はゼロから始まります。また、セグメント番号は、**X\$Index** システム テーブルの **Xi\$Part** 列に保管されている値に対応します。

たとえば、**Billing** テーブルに定義されているインデックス セグメントに関する情報を見るには、次のようなクエリを発行します。

```
SELECT Xe$Name,Xe$Offset, "X$Index".*
FROM X$File,X$Index,X$Field
WHERE Xf$Id=Xi$File and Xi$Field=Xe$Id and Xf$Name = 'Billing'
ORDER BY Xi$Number,Xi$Part
```

X\$Proc

X\$Proc システム テーブルは、**PVPROC.DDF** ファイルに関連付けられています。**X\$Proc** には、定義されているすべてのストアード プロシージャのコンパイルされた構造情報が含まれています。メタデータ バージョン 1 用の **X\$Proc** の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xp\$Name	CHAR	128	Yes	ストアード プロシージャ名
Xp\$Ver	UTINYINT	1	適用外	バージョン ID これは将来使用するために予約されています。
Xp\$Id	INTEGER	4	適用外	Zen によって割り当てられた内部 ID

列名	データ型	サイズ	大小文字無視	説明
Xp\$Flags	INTEGER	4	適用外	ストアド ステートメントは 1、ストアド プロシージャは 2、外部プロシージャは 3
Xp\$Trustee	INTEGER	4	適用外	信頼されるストアド プロシージャは 0、信頼されないストアド プロシージャは -1。信頼されるオブジェクトと信頼されないオブジェクトを参照してください。
Xp\$Sequence	USMALLINT	2	適用外	シーケンス番号。32,765 バイトを超えるプロシージャは、オーバーフローを処理するために PVPROC.DDF 内に複数のエントリを必要とします。各エントリは Xp\$Name フィールドに同一のプロシージャ名を持ち、シーケンス番号が割り当てられます。複数のエントリは、Xp\$Sequence フィールドを使用して正しく並べられます。 シーケンスはゼロから始まります（最初のシーケンス番号は 0 です）。
Xp\$Misc	LONGVARCHAR (LVAR)	32765	適用外	ストアド プロシージャの内部表現

メタデータ バージョン 2 用の X\$Proc テーブルでは、次のようにインデックスが 4 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xp\$Name	Yes	Yes	No
1	0	Xp\$Name	No	Yes	Yes
1	1	Xp\$Ver	No	適用外	Yes
1	2	Xp\$Sequence	No	適用外	No

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
2	0	Xp\$Id	Yes	適用外	No
3	0	Xp\$Id	No	適用外	Yes
3	1	Xp\$sequence	No	適用外	No

X\$Relate

X\$Relate システム テーブルは、PVRELATE.DDF ファイルに関連付けられています。X\$Relate には、データベースで定義されている参照整合性 (RI) 制約に関する情報が含まれています。X\$Relate は最初の外部キーが作成されるときに自動的に作成され、これにより関係が定義されます。

メタデータ バージョン 2 用の X\$Relate の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xr\$PIId	INTEGER	4	適用外	主テーブル ID
Xr\$Index	INTEGER	4	適用外	主テーブルに含まれる主キーのインデックス番号
Xr\$FId	INTEGER	4	適用外	従属テーブル ID
Xr\$FIndex	INTEGER	4	適用外	従属テーブルに含まれる外部キーのインデックス番号
Xr\$Name	CHAR	128	Yes	外部キー名
Xr\$updateRule	UTINYINT	1	適用外	制限は 1
Xr\$DeleteRule	UTINYINT	1	適用外	制限は 1、カスケードは 2
Xr\$Reserved	CHAR	250	No	予約済み

メタデータ バージョン 2 用の X\$Relate テーブルでは、次のようにインデックスが 5 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	変更可能	大小文字無視	セグメント
0	0	Xr\$PIId	Yes	No	適用外	No

インデックス番号	セグメント番号	列名	重複可能	変更可能	大小文字無視	セグメント
1	0	Xr\$FId	Yes	No	適用外	No
2	0	Xr\$Name	No	No	Yes	No
3	0	Xr\$PId	No	Yes	適用外	Yes
3	1	Xr\$Name	No	Yes	Yes	No
4	0	Xr\$FId	No	Yes	適用外	Yes
4	1	Xr\$Name	No	Yes	Yes	No

X\$Rights

X\$Rights システム テーブルは、PVRIGHTS.DDF ファイルに関連付けられています。X\$Rights には、各ユーザーのアクセス権情報が含まれています。Zen は、セキュリティオプションが有効になっているときにのみ、このテーブルを使用します。メタデータバージョン 2 用の X\$Rights の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xr\$User	INTEGER	4	適用外	User ID
Xr\$Object	INTEGER	4	適用外	Xf\$Id に対応するテーブル ID、Xv\$Id に対応するビュー ID、または Xp\$Id に対応するストアード プロシージャ ID
Xr\$Type	INTEGER	4	適用外	テーブルは 1、プロシージャは 3、ビューは 4
Xr\$Column	INTEGER	4	適用外	列 ID
Xr\$Rights	INTEGER	4	適用外	テーブル、列、ビュー、またはストアード プロシージャのアクセス権フラグ

Xr\$User 列は、X\$User テーブルの Xu\$Id 列に対応します。Xr\$Object 列は、以下のいずれかに対応します。

- X\$File テーブルの Xf\$Id 列

- X\$Views テーブルの Xv\$Id 列
- X\$Proc テーブルの Xp\$Id 列

Xr\$Column 列は X\$Field テーブルの Xe\$Id 列に対応します。

メモ： テーブル、ビュー、ストアド プロシージャのアクセス権を記述するシステムテーブル内の各行では、Xr\$Column 列の値はヌルになります。

Xr\$Rights 列には整数値が含まれ、その右端 8 ビットがユーザー アクセス権を定義します。次の表は、Zen が値を解釈する方法を説明したものです。この表の値は、メタデータバージョン 2 用の単一の Xr\$Rights 値に結合される場合があります。

16 進数の 値	対応する 10 進数の値	説明
1	1	オブジェクト所有者のアクセス権
0x90	144	テーブルの参照権
0xA0	160	テーブルの変更権
0x40	64	ビュー、テーブル、または列の選択権
0x82	130	ビュー、テーブル、または列の更新権
0x84	132	ビュー、テーブル、または列の挿入権
0x88	136	ビュー、テーブル、または列の削除権
0xC0	192	ストアド プロシージャの実行権および呼び出し権

10 進数の 0 はアクセス権がないことを示します。

特定のユーザーの Xr\$Rights 列の値は、そのユーザーに適用されるアクセス権に対応する 16 進数のビット単位の共通部分になります。10 進数の値の合計ではありません。

たとえば、すべてのアクセス権が割り当てられているユーザーの Xr\$Rights の値は次のように表されます。

144 | 160 | 64 | 130 | 132 | 136 = 254

ビューに関するすべてのアクセス権が割り当てられているユーザーの Xr\$Rights の値は次のように表されます。

64 | 130 | 132 | 136 = 206

ストアド プロシージャに関するすべてのアクセス権が割り当てられているユーザーの Xr\$Rights の値は次のように表されます。

192 = 192

メタデータ バージョン 2 用の X\$Rights テーブルでは、次のようにインデックスが 3 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xr\$User	Yes	適用外	No
1	0	Xr\$User	No	適用外	Yes
1	1	Xr\$Object	No	適用外	Yes
1	2	Xr\$Type	No	適用外	Yes
1	3	Xr\$Column	No	適用外	No
2	0	Xr\$Object	Yes	適用外	Yes
2	1	Xr\$Type	Yes	適用外	Yes
2	2	Xr\$Column	Yes	適用外	No

X\$Trigger

X\$Trigger システム テーブルは、PVTRIG.DDF ファイルに関連付けられています。X\$Trigger には、データベースで定義されているトリガーに関する情報が含まれています。メタデータ バージョン 2 用の X\$Trigger の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xt\$Name	CHAR	128	Yes	トリガー名
Xt\$Version	UTINYINT	1	適用外	トリガーのバージョン。Scalable SQL バージョン 4 の場合は 4
Xt\$File	INTEGER	4	適用外	トリガーが定義されているファイル。X\$File 内の Xf\$Id に対応
Xt\$Event	UTINYINT	1	適用外	INSERT は 0、DELETE は 1、UPDATE は 2
Xt\$ActionTime	UTINYINT	1	適用外	BEFORE は 0、AFTER は 1

列名	データ型	サイズ	大小文字無視	説明
Xt\$ForEach	UTINYINT	1	適用外	ROW は 0 (デフォルト)、STATEMENT は 1
Xt\$Order	USMALLINT	2	適用外	トリガーの実行順序
Xt\$Sequence	USMALLINT	2	適用外	シーケンス番号。4,054 バイトを超えるトリガーは、オーバーフローを処理するために PVTRIG.DDF 内に複数のエントリを必要とします。各エントリは Xt\$Name フィールドに同一のトリガー名を持ち、シーケンス番号が割り当てられます。複数のエントリは、Xt\$Sequence フィールドを使用して正しく並べられます。 シーケンスはゼロから始まります (最初のシーケンス番号は 0 です)。
Xt\$Misc	LONGVARCHAR (LVAR)	4054	適用外	トリガーの内部表現

メタデータ バージョン 2 用の X\$Trigger テーブルでは、インデックスが 3 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xt\$Name	No	Yes	Yes
0	1	Xt\$Sequence	No	適用外	No
1	0	Xt\$Name	No	Yes	Yes
1	1	Xt\$File	No	適用外	Yes
1	2	Xt\$Sequence	No	適用外	No
2	0	Xt\$File	Yes	適用外	Yes
2	1	Xt\$Event	Yes	適用外	Yes
2	2	Xt\$ActionTime	Yes	適用外	Yes
2	3	Xt\$ForEach	Yes	適用外	Yes
2	4	Xt\$Order	Yes	適用外	Yes
2	5	Xt\$Sequence	Yes	適用外	No

X\$User

X\$User システム テーブルは、PVUSER.DDF ファイルに関連付けられています。X\$User には、各ユーザーの名前とパスワード、および各ユーザー グループの名前が含まれています。Zen は、セキュリティ オプションが有効になっているときにのみ、このテーブルを使用します。X\$User の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xu\$Id	INTEGER	4	適用外	ユーザーまたはグループに割り当てられている内部 ID
Xu\$Name	CHAR	128	Yes	ユーザーまたはグループ名
Xu\$Password	CHAR	153	No	ユーザー パスワード (暗号化)
Xu\$Flags	INTEGER	4	適用外	ユーザーまたはグループ フラグ

メモ： グループを記述する X\$User システム テーブル内の各行では、Xu\$Password の列の値はヌルになります。

Xu\$Flags 列には整数値が含まれ、その右端の 8 ビットはユーザーまたはグループ属性を定義します。次の表は、ビットがバイナリ値 1 であるとき、Zen が各ビット位置を解釈する方法を示したものです。ビット位置 0 が整数の右端のビットになります。

ビット位置	対応する 10 進数の値	説明
0	1	予約済み
1	2	予約済み
2	4	予約済み
3	8	予約済み
4	16	予約済み
5	32	予約済み
6	64	名前はグループ名
7	128	ユーザーまたはグループは、辞書内のテーブルを定義する権限を持っています。
8	256	ユーザーまたはグループは、辞書内のビューを定義する権限を持っています。
9	512	ユーザーまたはグループは、辞書内のストアード プロシージャを定義する権限を持っています。

特定のユーザーやグループの Xu\$Flags 列の値は、そのユーザーまたはグループに適用される属性に対応する 10 進数の値の合計になります。

メタデータ バージョン 2 用の X\$User テーブルでは、次のようにインデックスが 2 つ定義されています。

インデックス番号	セグメント番号	列名	重複可能	大小文字無視	セグメント
0	0	Xu\$Id	Yes	適用外	No
1	0	Xu\$Name	No	Yes	No

X\$View

X\$View システム テーブルは、PVVIEW.DDF ファイルに関連付けられています。X\$View には、結合テーブルの情報やビューを定義する制約条件などのビュー定義が含

まれています。X\$View テーブルにクエリを実行して、辞書に定義されているビューの名前を取得することができます。

X\$View テーブルの先頭列にはビュー名が含まれます。第 2 列と第 3 列には、LVAR 列の Xv\$Misc で検出される情報の説明が含まれています。メタデータ バージョン 2 用の X\$View の構造は次のとおりです。

列名	データ型	サイズ	大小文字無視	説明
Xv\$Name	CHAR	128	Yes	ビュー名
Xv\$Version	UTINYINT	1	適用外	バージョン ID 今後の使用に備えて予約されています。
Xv\$Id	INTEGER	4	適用外	Zen によって割り当てられた内部 ID
Xv\$Trustee	INTEGER	4	適用外	信頼されるビューは 0、信頼されないビューは -1。信頼されるオブジェクトと信頼されないオブジェクトを参照してください。
Xv\$Sequence	USMALLINT	2	適用外	シーケンス番号。32,765 バイトを超えるビューは、オーバーフローを処理するために PVVIEW.DDF 内に複数のエントリを必要とします。各エントリは Xv\$Name フィールドに同一のビュー名を持ち、シーケンス番号が割り当てられます。複数のエントリは、Xv\$Sequence フィールドを使用して正しく並べられます。 シーケンスはゼロから始まります（最初のシーケンス番号は 0 です）。
Xv\$Misc	LONGVARCHAR (LVAR)	32765	適用外	Zen 内部定義

メタデータバージョン2用の X\$View テーブルでは、次のようにインデックスが3つ定義されています。

インデックス 番号	セグメント番号	列名	重複可能	大小文字無視	セグメン ト
0	0	Xv\$Name	Yes	Yes	No
1	0	Xv\$Name	No	Yes	Yes
1	1	Xv\$Version	No	適用外	Yes
1	2	Xv\$Sequence	No	適用外	No
2	0	Xv\$Id	Yes	適用外	No
3	0	Xv\$Id	No	適用外	Yes
3	1	Xv\$Sequence	No	適用外	No

D. COBOL アプリケーションの SQL アクセス

この付録では、以下の項目について説明します。

- COBOL の Zen サポートの概要
- コンポーネント
- SQL アクセスの使用
- サンプル XML ファイルの実行例

COBOL の Zen サポートの概要

Zen リレーショナル エンジンには、COBOL の OCCURS 句、部分的 REDEFINES、および可変レコード レイアウトのサポートが含まれています。

部分的 REDEFINES 句は、レコード内のデータの一部（たとえば、01 レベル内の 05 レベルなど）を識別します。可変レコード レイアウトもまた REDEFINES と呼ばれます。これは、レコード全体を再定義するものだからです。用語の混乱を避けるため、このトピックでは部分的 REDEFINES および可変レコード レイアウトと呼びます。

SQL アクセスを利用するために COBOL アプリケーションを変更する必要はありません。

アプリケーションでのデータの処理を Zen リレーショナル エンジンに示すことによって、SQL アクセスは有効になります。開発者の言葉で言えば、リレーショナル エンジンにメタデータを定義します。

このトピックは、OCCURS、部分的 REDEFINES、または可変レコード レイアウトを含んでいる COBOL アプリケーションにのみ適用されることに注意してください。

制限

現在、COBOL アプリケーションに SQL アクセスを提供する場合には以下の制限が適用されます。

- OCCURS の中で OCCURS をネストすることはできません。
- 部分的 REDEFINES の中で OCCURS をネストすることはできません。

- 部分的 REDEFINES の中で部分的 REDEFINES をネストすることはできません。
- OCCURS の中で部分的 REDEFINES をネストすることはできません。
- OCCURS に定義できるのは 1 インデックスのみです。OCCURS の中で、項目に追加インデックスを定義することはできません。
- サポートされるデータ型は、MicroKernel エンジンおよびリレーショナル エンジンに定義されているデータ型のみです。データ型は、トランザクショナル データ型のみを使って、XML 制御ファイルに記述されます。XML 制御ファイルの解説については、[コンポーネント](#) セクションの表で [cobolschemaexec.xsd](#) を参照してください。データ型の説明については、[Zen で使用できるデータ型](#) を参照してください。

SQL ステートメント

次の表は、OCCURS、部分的 REDEFINES、または可変レコード レイアウトに基づいて作成されたデータ テーブルでの SQL ステートメントの使用を示します。

ステートメント	OCCURS、部分的 REDEFINES での使用	可変レコード レ イアウトでの使 用	注記
ALTER TABLE	No	No	
CREATE INDEX	No	No	
CREATE INDEX IN DICTIONARY	No	Yes	
CREATE TRIGGER	No	No	
DELETE	No	Yes	

ステートメント	OCCURS、部分的 REDEFINES での使用	可変レコード レ イアウトでの使 用	注記
DROP TABLE	Yes	Yes	DROP TABLE ステートメントは、システム テーブルからすべてのエントリを削除します。データ ファイル自体は削除も変更もされません。また、メイン テーブルを削除する場合、従属テーブルが検出されると、それらをすべて削除するよう通知するメッセージが表示されます。従属テーブルはメイン テーブルに依存するものであり、これはインデックスを含んでいる OCCURS や部分的 REDEFINES などの条件によって生じます。従属テーブルを削除すれば、メイン テーブルの削除が可能になります。
INSERT INTO	No	No	
UPDATE	Yes	Yes	UPDATE ステートメントで、テーブル フィルターが定義されている列は更新できません。テーブル フィルターとは、テーブルに関連付けられている論理式のことです。テーブル フィルターは XML ファイル内のメタデータの一部として定義されます。
『SQL Engine Reference』に列挙されているその他のすべての SQL ステートメント	Yes	Yes	Zen の SQL 文法 を参照してください。

コンポーネント

Zen は、COBOL アプリケーションの SQL アクセスを提供するために以下のコンポーネントをインストールします。

コンポーネント	用途	場所 ¹
w3cobolschemaexec100.dll	Schema Executor が使用する ルーチンの 32 ビット ライ ブラリ	Windows サーバー : <i>file_path¥Zen¥bin¥</i>
w64cobolschemaexec.dll	Schema Executor が使用する ルーチンの 64 ビット ライ ブラリ	Windows サーバー : <i>file_path¥Zen¥bin¥</i>
Linux : libpsqlcobolschemaexec100.so	Schema Executor が使用する ルーチンの 32 ビットおよび 64 ビット ライブラリ	Linux サーバー : /usr/local/ actianzen/lib
cobolschemexecmsgsr.dll	ルーチンの 32 ビット ライ ブラリで使用されるメッ セージリソースバンドル	Windows サーバー : <i>file_path¥Zen¥bin¥</i>
w64cobolschemexecmsgsr.dll	ルーチンの 64 ビット ライ ブラリで使用されるメッ セージリソースバンドル	Windows サーバー : <i>file_path¥Zen¥bin¥</i>
Linux : libpsqlcobolschemexecmsgsr.so	ルーチンの 32 ビットおよび 64 ビット ライブラリで使用 されるメッセージリソース バンドル	Linux サーバー : /usr/local/ actianzen/lib
cobolschemaexec.xsd	XML ファイルを処理する際 に Schema Executor が使用す る制御ファイル (ドキュメ ント型定義)	Windows サーバー : <i>file_path¥Zen¥schemas</i> Linux サーバー : /user/local/actianzen/ schemas/
cobolschemaexec.log	XML ファイルを処理する際 に Schema Executor が生成し たメッセージを記録するデ フォルトのログファイル	Windows サーバー : <i>file_path¥Zen¥logs</i> Linux サーバー : /usr/local/actianzen/logs/

コンポーネント	用途	場所 ¹
cobolschemaexec.exe	ISAM データを正規化された SQL テーブルと解釈するために、リレーショナルエンジンが使用するシステムテーブルを作成するユーティリティ Schema Executor と呼ばれます。	<i>file_path</i> ¥Zen¥bin¥
cobolschemaexec	ISAM データを正規化された SQL テーブルと解釈するために、リレーショナルエンジンが使用するシステムテーブルを作成するユーティリティ Schema Executor と呼ばれます。	Linux インストール : /usr/local/actianzen/bin/
SampleMainTable.xml	シンプルなテーブルのデータを定義する、サンプルの XML テンプレート サンプル XML ファイルの実行例 も参照してください。	Windows サーバーおよびクライアント : <i>file_path</i> ¥Zen¥samples¥cobolschemaexec Linux インストール : /usr/local/actianzen/samples/cobolschemaexec
SampleMainWithOccurs.xml	OCCURS 構造を含んでいるデータの定義に使用する、サンプルの XML テンプレート サンプル XML ファイルの実行例 も参照してください。	Windows サーバーおよびクライアント : <i>file_path</i> ¥Zen¥samples¥cobolschemaexec Linux インストール : /usr/local/actianzen/samples/cobolschemaexec
SampleMainWithRedefines.xml	REDEFINES 構造を含んでいるデータの定義に使用する、サンプルの XML テンプレート サンプル XML ファイルの実行例 も参照してください。	Windows サーバーおよびクライアント : <i>file_path</i> ¥Zen¥samples¥cobolschemaexec Linux インストール : /usr/local/actianzen/samples/cobolschemaexec

コンポーネント	用途	場所 ¹
SampleVariantRecord.xml	可変レコード レイアウトを 含んでいるデータの定義に 使用する、サンプルの XML テンプレート サンプル XML ファイルの実 行例 も参照してください。	Windows サーバーおよび クライアント： <code>file_path¥Zen¥samples¥ cobolschemaexec</code> Linux インストール： <code>/usr/local/actianzen/samples/ cobolschemaexec</code>
Windows および Linux クライアントでは、ログ ファイルはオプションです。Schema Executor を起動するときに指定できます。	Schema Executor コマンドの形式 を参照	パスが指定されていない場合は、現在のディレクトリと同じです。そうでない場合は、ユーザー指定のパスによって決まります。

¹ Zen ファイルのデフォルトの保存場所については、『*Getting Started with Zen*』の**ファイルはどこにインストールされますか？**を参照してください。

SQL アクセスの使用

SQL アクセスを利用するには次の作業を行います。

1. 適切な XML テンプレートを手で編集して、データ レイアウトを記述します。
2. XML テンプレートで指定したデータ ファイルをデータベース フォルダにコピーします。
3. ユーティリティを実行して、リレーショナル エンジンが使用するシステム テーブルを作成します (XML を使って、正規化されたデータを作成します)。
4. COBOL アプリケーションの開発者である場合は、任意で、Schema Executor によって作成された新しいシステム テーブルを配布します。

手順 1 : サンプル XML テンプレートの変更

Zen にはサンプルの XML テンプレートが含まれており、これを利用すると、COBOL アプリケーションで必要とされたとおりにデータのレイアウトを定義できます。**コンポーネント**の表を参照してください。開発者の言葉で言えば、XML ファイルにメタデータを記述します。

XML テンプレートを変更するには

1. テキスト エディターで XML テンプレートを開きます。
2. ファイルに記述されているコメントに従って XML を変更します。
3. 選択したパスとファイル名で、変更したテンプレートを保存します。

手順 2 : XML テンプレートで指定したデータ ファイルのコピー

Schema Executor を起動する前に、XML ファイルで指定したデータ ファイルをデータベースのデータ ファイルの場所へコピーします。データベースは、テーブルを追加する必要のあるデータベースです。

たとえば、XML で mytable.mkd と指定したテーブルをデータベース test に追加するとします。このとき、データベースのデータ ファイルは c:¥data¥test にあるとします。

Schema Executor を起動する前に、データ ファイル mytable.mkd を c:¥data¥test へコピーします。

手順 3 : Schema Executor ユーティリティの実行

Zen には Schema Executor というコマンド ライン ユーティリティが付属しています。このユーティリティは SchemaExec とも言います。

Schema Executor は以下の操作を実行します。

- 手で編集された XML ファイルを解析する
- リレーショナル エンジンがデータを正規化された SQL テーブルと解釈するために使用する、既存のシステム テーブルを配置する（Zen で作成されるデータベースには、SQL アクセスのサポートに必要なシステム テーブルがすべて含まれています）
- Zen の現在のバージョンより前のバージョンで作成されたデータベースに対してこのユーティリティを実行している場合は、追加のシステム テーブルを作成して配置する

Schema Executor で XML ファイルを処理するには

[サンプル XML ファイルの実行例](#)も参照してください。

1. オペレーティング システムでコマンド プロンプトにアクセスします。

2. コマンドラインで **Schema Executor** を実行します（この実行可能プログラムがデフォルトでインストールされる場所は、[コンポーネント](#)の表を参照してください）。

必須オプションの **XMLfilename** および **databasename** と、必要な任意のオプションを指定します。[Schema Executor コマンドの形式](#)を参照してください。

XML コンテンツの処理中にエラーが発生した場合は、**Schema Executor** ログ ファイルに報告されているエラーを検討します。[ログ メッセージ](#)を参照してください。処理でエラーが発生しなくなるまで、繰り返し XML を修正してユーティリティを実行します。

Schema Executor コマンドの形式

```
cobolschemaexec XMLfilename databasename [-s servername] [-u login_id] [-p password] [-i svr_loginid] [-c svr_password] [-l log_file] [-h | -?]
```

次の表に **Schema Executor** ユーティリティのオプションを示します。

オプション	説明
<i>XMLfilename</i>	データのレイアウトを定義する XML スキーマのファイル名。必須オプションです。 手順 1：サンプル XML テンプレートの変更 を参照してください。
<i>databasename</i>	アプリケーションがアクセスする Zen データベースの名前。必須オプションです。指定したデータベースが存在しない場合は、ユーティリティからパスとファイル名の入力を求められます。 Schema Executor による新規データベースの作成 も参照してください。
<i>-s servername</i>	Zen データベース エンジンが実行されているサーバーの名前、または IP アドレス。データベース エンジンと同じマシン上で SchemaExec を実行している場合は、名前として "localhost" を使用できます。 <i>servername</i> が指定されていない場合は、ローカル マシンがサーバーであると見なします。
<i>-u login_id</i>	セキュリティで保護されたデータベースへのアクセスに必要なユーザー名。 Zen セキュリティ モデルの解説については、『 <i>Advanced Operations Guide</i> 』の Zen セキュリティ を参照してください。
<i>-p password</i>	セキュリティで保護されたデータベースへのアクセスに必要なパスワード。 Zen セキュリティ モデルの解説については、『 <i>Advanced Operations Guide</i> 』の Zen セキュリティ を参照してください。
<i>-i svr_loginid</i>	リモート マシンのオペレーティング システムへのアクセスに必要なログイン名。 SchemaExec がリモート サーバー上にある XML ファイルを処理している場合には、このオプションは必須です。

オプション	説明
<code>-c svr_password</code>	リモート マシンのオペレーティング システムへのアクセスに必要なパスワード。SchemaExec がリモート サーバー上にある XML ファイルを処理している場合には、このオプションは必須です。
<code>-l log_file</code>	XML ファイルの処理中に生成されるメッセージの記録に使用するログファイル。 Zen データベース エンジンが実行されているマシンで SchemaExec を実行する場合は、自動的にデフォルト ログが作成されます。 <code>-l log_file</code> オプションを使用する必要はありません。デフォルト ログの名前は cobolschemaexec.log です。 クライアント マシン (Zen データベース エンジンが実行されていないマシン) で SchemaExec を実行する場合は、クライアント マシン用のログ ファイルを指定できます。 ログ メッセージ を参照してください。
<code>-h</code> または <code>-?</code>	コマンドの使い方を表示します。その他すべてのオプションは無視されます。

メモ： 必須オプションの *XMLfilename* と *databasename* は位置決めされており、この順序で指定する必要があります。

使い方の例

次の例は、Schema Executor の使い方を示しています。

Zen ファイルのデフォルトの保存場所については、『*Getting Started with Zen*』の[ファイルはどこにインストールされますか？](#)を参照してください。

- データベースが既に存在する場合 (サーバーはローカル ホストで動作している) :

```
cobolschemaexec file_path%Zen%samples%cobolschemaexec%test.xml demodata%
```

- データベースが存在しない場合 (サーバーはローカル ホストで動作している) :

```
cobolschemaexec file_path%Zen%samples%cobolschemaexec%test.xml mytest
```

ユーティリティは次のようなプロンプトを表示します。

```
CB103 : mytest に接続できません
データベースを作成しますか (y/n)?
```

y を押すと、ユーティリティはデータベース パスを入力するプロンプトを表示します。

データベース パスを入力してください。

既存のパスを入力します。そうしないと、エラーが返されます。必ず、XML ファイルで使用されているデータベース ファイル（たとえば .MKD ファイル）がそのパスで使用可能であるようにしてください。

- データベースがリモート サーバーに存在する場合：

```
cobolschemaexec file_path%Zen%samples%cobolschemaexec%test.xml demodata -s TestMachine -i testuser -c testuser
```

この例では、管理者権限を持つユーザー `testuser`（パスワード "`testuser`"）がリモート マシン（`TestMachine`）に存在し、XML ファイルで使用されるデータベース ファイルは、リモート マシン上のデータベースのデータ ファイル ディレクトリにあるものとします。

- データベースがリモート サーバーに存在しない場合：

```
cobolschemaexec file_path%Zen%samples%cobolschemaexec%test.xml mytest -s RemoteMachine -i testuser -c testuser
```

この例では、ユーザー `testuser`（パスワード "`testuser`"）はリモート マシン（`TestMachine`）に存在し、管理者権限を持っているものとします。

ユーティリティは次のようなプロンプトを表示します。

```
CB103 : mytest に接続できません  
データベースを作成しますか (y/n)?
```

`y` を押すと、ユーティリティはデータベース パスを入力するプロンプトを表示します。

```
データベース パスを入力してください。
```

既存のパスを入力します。そうしないと、エラーが返されます。必ず、XML ファイルで使用されているデータベース ファイル（たとえば .MKD ファイル）がそのパスで使用可能であるようにしてください。

Schema Executor による新規データベースの作成

ユーティリティ オプション `databasename` に存在しないデータベースを指定すると、Schema Executor は新しいデータベースを作成するかどうかを尋ねます。"yes" と答えると、新しいデータベースの場所を入力するプロンプトが表示されます。Schema Executor がデータベースを作成する場所（パスとフォルダー名）は既存でなければなりません。

また、Schema Executor は、デフォルト フォルダーに `databasename` のデータ ファイルがあることを期待するので注意してください。データ ファイルが見つからない場合は、

その旨が通知されます。データ ファイルを自分でデフォルト フォルダへコピーしてから、Schema Executor を再実行して XML を処理します。

ログ メッセージ

このセクションは、Schema Executor による XML ファイルの処理後、ログ ファイルに記載されている可能性のあるコードの一覧を示します。

成功を表すコードは、「**CB100 : Schemaexec は正常に完了しました**」です。

次の表は、エラー コードの一覧を示します。

エラー コード	説明
CB001	不明なエラーです
CB002	プロパティ名属性がありません
CB003	MAINTABLE および VARIANTRECORDTABLES はサポートされていません
CB004	指定された Occurs テーブル名は無効です。
CB005	指定された Occurs Count は無効です。
CB006	指定された Occurs マッピング インデックスは親テーブルのインデックスではありません。
CB007	<i>TableName</i> - <i>TableName</i> が重複しています
CB008	<i>FieldName</i> - <i>FieldName</i> が重複しています
CB009	<i>IndexName</i> - <i>IndexName</i> が重複しています
CB010	<i>TableName</i> パラメーターが指定されていません。
CB012	<i>Identifier</i> - 無効な文字が含まれています。
CB013	オフセットは負でない整数でなければなりません。
CB014	識別子の長さは 20 文字を超えてはいけません。
CB016	精度はゼロより大きくなければなりません。
CB017	指定された精度が無効です - <i>FieldName</i>
CB018	小数位を精度よりも大きくすることはできません - <i>FieldName</i>
CB019	ログ ファイルと XML ファイルは別にしなければなりません。

エラー コード	説明
CB022	<i>TableFilter</i> - テーブル フィルターは空にできません。また、255 文字を超えてもいけません。
CB023	<i>FieldName</i> は <i>TableName</i> のフィールドではありません。
CB024	テーブル フィルターは、すべての REDEFINES テーブルに対して、または NONE に対して定義する必要があります。
CB025	親要素が正しくありません
CB028	識別子名 <i>identifiername</i> は英文字で始まっている必要があります。
CB029	識別子名 <i>identifiername</i> にキーワードは使用できません。
CB050	<i>DataFile</i> - <i>DatabasePath</i> に存在しません。
CB051	OCCURS/REDEFINES の長さは負でない整数でなければなりません。
CB052	<i>Btrievefilename</i> の長さが 64 文字を超えてはいけません。
CB057	親テーブルにインデックスが指定されていません - <i>ParentTableName</i>
CB099	パーサー エラー
CB100	Schemaexec は正常に完了しました。
CB101	コマンド ライン引数 <i>argument</i> の値が無効です
CB102	<i>Login</i> を指定しないで、 <i>Password</i> の値を指定することはできません。
CB103	<i>DatabaseName</i> に接続できませんでした。
CB105	ディレクトリ パス <i>Databasepath</i> にデータベース <i>DatabaseName</i> を作成できませんでした。
CB106	指定された DSN を作成できませんでした。
CB108	データベース <i>databasename</i> を閉じられませんでした。
CB109	XML ファイルからデータを読み取れませんでした。
CB110	データベースを削除できませんでした。

手順 4 : システム テーブルの配布 (任意)

COBOL アプリケーションの開発者である場合は、作成したアプリケーションと共にすべてのシステム アプリケーションを配布するようにしてください。配布自体に違いは

ありませんが、追加するシステム テーブルが生じる可能性があるので、ここで述べています。たとえば、Zen の現在のバージョンより前のバージョンで作成されたデータベースに対して **Schema Executor** を実行した場合は、追加のシステム テーブルが作成され配置されます。その結果、配布するシステム テーブル (DDF ファイル) がいくつか追加されることになります。

サンプル XML ファイルの実行例

サンプル XML ファイルおよびデータ ファイルは、`file_path¥Zen¥samples¥cobolschemaexec` フォルダー内に提供されています。Zen ファイルのデフォルトの保存場所については、『*Getting Started with Zen*』の[ファイルはどこにインストールされますか？](#)を参照してください。

Schema Executor を使って XML ファイル `SampleMainTable.xml` を実行するには、次の手順に従います。

1. 接続するデータベースのデータ ファイル フォルダーに `maintbl.mkd` をコピーします。
たとえば、`test` という名前のデータベースがあり、データ ファイルの場所は `c:¥data¥test` であるとします。`c:¥data¥test` へ `maintbl.mkd` をコピーします。
2. `Zen¥bin¥` ディレクトリでコマンド プロンプトを開きます。
3. DOS プロンプトで次のコマンドを実行します。

```
cobolschemaexec file_path¥Zen¥samples¥cobolschemaexec¥samplemaintable.xml test
```

4. **Schema Executor** の実行が成功したら、(XML ファイルに指定されている) テーブル `maintbl` が `test` データベースに作成されます。
5. これで、ZenCC を使ってテーブル `maintbl` で SQL 操作を実行できます。

補注

このセクションでは、SELECT ステートメントおよびテーブル フィルターに関する注意事項を提供します。

SELECT ステートメント

OCCURS テーブルで SELECT クエリを実行すると、次の内容が返されます。

-
- OCCURS テーブルの列
 - マッピング インデックスを構成するメイン テーブルの列
 - OCCURS 句の出現回数を示す OCCURS カウンター

たとえば、次のサンプル XML ファイルについて Schema Executor を実行した結果作成されるテーブルに対し、SELECT * FROM FIELD というクエリを実行するとします。

file_path¥Zen¥samples¥cobolschemaexec¥SampleMainWithOccurs.xml

ユーティリティは、列 Id、OccursCounter、Field_1、Field_2、および Field_3 を返します。

REDEFINES テーブルで SELECT を実行すると、親テーブルの列と REDEFINES テーブルの列がすべて返されます。

たとえば、次のサンプル XML ファイルについて Schema Executor を実行した結果作成されるテーブルに対し、SELECT * FROM Redefined_group というクエリを実行するとします。

file_path¥Zen¥samples¥cobolschemaexec¥SampleMainWithRedefines.xml

ユーティリティは、列 Id、Account_Num、Category、Redef_Struct_Num（親テーブルのすべての列）、および Redefined_field_1（REDEFINES テーブルの列）を返します。

テーブル フィルター

テーブル フィルターとは、特定のテーブルのフィルター条件のことです。サンプル XML ファイルでは TABLEFILTER として参照されます。

- TABLEFILTER の式は、左右両方のオペランドが列名であってもかまいません。たとえば、Cust_Num = My_Cust_Num とした場合、Cust_Num と My_Cust_Num はどちらも列名です。

オペランドと演算子の間にスペースを入れてください。

- TABLEFILTER の式で定数を使用する場合は、値を一重引用符で囲んで指定する必要があります。

例：Cust_Num = '100'（Cust_Num は列名です）

- XML ファイルで TABLEFILTER を指定する場合は、次の XML エンティティを使用します。

XML エンティティ	使用目的
<	より小さい (<)
>	より大きい (>)
&	アンパサンド (&) (AND)
"	二重引用符 (")
'	一重引用符 (')

有効な TABLEFILTER の使用例

Cust_Num = '100' (Cust_Num = 100 と同等)

Cust_Num < '100' (Cust_Num < 100 と同等)

Cust_Num > '100' (Cust_Num > 100 と同等)

Cust_Num <>'100' (Cust_Num <> 100 と同等)

'a' = Category | Account_Num <= 'a123' ('a'=category OR account_num <= 'a123' と同等)

'a' = Category & Account_Num = 'a123' ('a' = category AND account_num = 'a123' と同等)

Cust_Num = My_cust_Num (両オペランドが列名)

E. Query Plan Viewer

おそらく、SQL パフォーマンスの最も複雑な面は、クエリの最適化です。データベースエンジンはクエリの最適化を自動的に実行しますが、クエリ構造自体が全体的な性能やエンジンによる最適化方法に影響する可能性があります。

ほとんどすべてのクエリは、2 とおり以上の書き方をしても、同じ結果セットを返すことができます。たとえば、単純な `SELECT * FROM table1` というクエリについて考えてみましょう。`table1` には 5 つの列があり、`col1`、`col2`、(以下同様) という名前が付けられているとします。クエリを `SELECT col1, col2, col3, col4, col5 FROM table1` と記述しても、同様の結果セットをもたらすことができます。

これら 2 つのクエリを視覚的に比較した場合、`SELECT *` の方が、各列を名前で列挙するよりも単純なように見えます。しかし、クエリ内に各列を名前で列挙することによって、ごくわずかながらパフォーマンスが引き上げられます。これは、`SELECT *` の場合には、アスタリスク記号を列名に解析しなければならないからです。クエリ自体でそのような解析を前もって実行しておけば、その作業は必要でなくなります。

パフォーマンスを向上させる最良の方法は、データベースに対してクエリを実行するのに必要な時間を最小限にすることです。クエリは非常に複雑にすることも、途方もなく大きい構造にすることもできるため、考え得るすべてのクエリの最適化について、このトピックで説明することはできません。

Query Plan Viewer を使用して、クエリを最適化する方法を決定できます。Query Plan Viewer は、データベースエンジンによって選択されたクエリ プランを表示するためのグラフィカル ユーティリティです。クエリ プランでは `SELECT`、`INSERT`、`UPDATE` または `DELETE` ステートメントについて表示できます。Query Plan Viewer はワイド文字データに対応しています。

メモ： Query Plan Viewer を [SQL クエリ ログ](#) と一緒に使用すると便利です。

Query Plan の設定

2 つの SQL ステートメントによって、クエリ プランを作成するかどうか、また、プランにどのような名前を付けるかを制御できます。どちらのステートメントも SQL セッションにのみ適用されます。

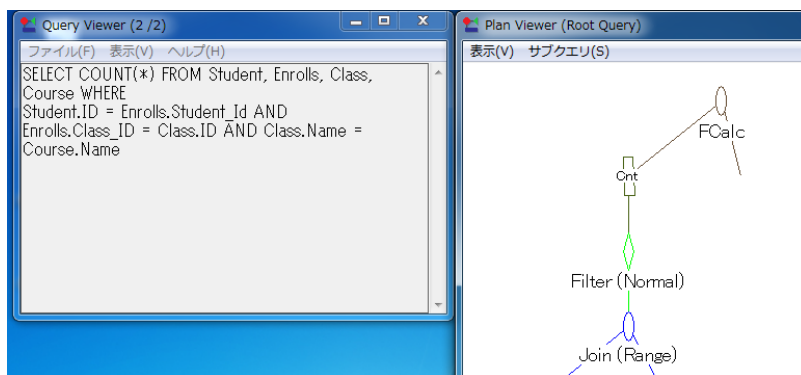
以下のステートメントは、Zen Control Center 内で、または Zen データベース エンジンに SQL ステートメントを送信できるユーティリティから実行できます。

SQL ステートメント	説明
SET QRYPLAN=<on off>	Query Plan Viewer を使用してクエリ プランを作成するかどうかを、データベース エンジンに指示します。
SET QRYPLANOUTPUT=<NULL ファイル名 >	クエリ プラン ファイルの場所と名前を設定します。NULL は、クエリ プラン ファイルを作成しないように指示します。1 つのクエリ プラン ファイルに複数のクエリのプラン結果を入れることができます。参考までに、クエリ プラン ファイルには、クエリごとに使用するエンコードのコード ページ識別子が含まれています。クエリでどのデータベース エンコードが使用されているかに関係なく、Query Plan Viewer はワイド文字データを正しく表示します。 デフォルトでは、Query Plan Viewer は拡張子が .qpf のファイル名を探します。ファイル名の拡張子はどんなものでも使用できますし、省略してもかまいません。 たとえば、select_salary という名前のクエリ プランを作成し、そのクエリ プランをドライブ D のルートにあるディレクトリ mydirectory に保存します。 SET QRYPLANOUTPUT='d:¥mydirectory¥select_salary.qpf' データベース エンジンがクエリ プランの出力ファイルを作成するため、パスは、データベース エンジンが実行されているマシン上の場所にする必要があります。パスがクライアント側の場所やクライアントのドライブ割り当てを参照してはいけません。

クエリ プランの検査およびクエリパフォーマンスの評価も参照してください。

グラフィカル ユーザー インターフェイス

Query Plan Viewer には、**Query Viewer** と **Plan Viewer** の 2 つのウィンドウが含まれています。



Query Viewer

Query Viewer では SQL クエリを表示します。一度に表示できるクエリ プランは 1 つですが、複数のプランを同時に開いておくことができます。また Query Viewer には、クエリ プラン ファイルを開く、2 つ以上のプランを同時に開いているときに目的のクエリ プランへ移動する、およびクエリ プランをエクスポートするためのメニュー コマンドが含まれています。

Query Viewer は必要に応じて表示サイズを変更することができます。SQL クエリ の表示を容易にする垂直スクロールバーも備えています。

Query Viewer ウィンドウは、ワイド文字データに適したフォントを使用します。Query Plan Viewer は、使用可能なシステム フォントをチェックし、次のうちから使用可能な最初の 1 つを選択します。

- Consolas
- Lucida Console
- Andale Mono
- Courier New

Plan Viewer

Plan Viewer には、クエリ プランのグラフィック表現がノードを持つツリー形式で含まれます。各ノードはクエリのさまざまなコンポーネントを表します。

Plan Viewer は必要に応じて表示サイズを変更することができます。さまざまなサイズでクエリ プラン ツリーを表示するための、垂直および水平スクロールバーを備えています。ツリーのサイズを変更したり、拡大/縮小したりするためのメニュー コマンドやキーボード ショートカットがあります。

Query Viewer には、クエリに応じてさまざまなタイプのノードが表示されます。各ノードは、クエリの実行のステップを表します。たとえば、ベース テーブルから選択する、2 つのテーブルの結果を結合して 1 つの結果セットに入れる、集計値を計算する、グループの変わり目を判断する、グループ結果を組み合わせて 1 つの結果セットに入れるなどを表すことができます。

ノード

次の表はノードの種類を示しています。

ノードの記号	ノードの意味
 T1 (ndx2)	データベース内のテーブルからデータを得ることを表します。四角形の下に表示される名前は、テーブルの名前です。テーブル名の下に名前が表示されている場合、その名前は、テーブルからデータを取得する際に使用したインデックスを示します。インデックス名の右にあるアスタリスクは、そのインデックスに一意の値が含まれていることを示します。
 Filter (Normal)	行の選択操作を表します。カッコ内の単語は、"Normal" か "Range:" のいずれかになります。 • Normal フィルターは、ダウンストリーム ノードから行が返された後に適用されます。 • Range フィルターはベース テーブルの真上にのみ現れます。Range フィルターによって、テーブルからインデックス値を基に制限されたレコードが取得されます。
 Distinct	DISTINCT（重複値の削除）操作を実行します。このノードは通常、プラン ツリーの先頭か先頭付近に現れます。クエリから行を返す前に、結果セットから重複を取り除きます。

ノードの記号	ノードの意味
 Group Break	GROUP BY 句を基に、グループの変わり目を検出します。
 Group	SELECT および HAVING 句で集計値を正しく累算するために、Group Break ノードと一緒に働きます。
 Join (Outer)	2つのノード間の JOIN を実行します。カッコ内の値は、実行される結合 (JOIN) の種類を示します。 • "Outer" は、インデックスを使用しない、左外部結合または右外部結合を示します。 • "OuterRange" は、インデックスを使用する、左外部結合または右外部結合を示します。 • "Normal" は、カルテシアン結合を示します。 • "Range" は、インデックスを使用する内部結合を示します。
 Sum	集計値の累算を実行します。このノードは MIN、MAX、AVG、COUNT、SUM、および STDEV に対して、これらの集計関数で DISTINCT 句が使われている場合に使用されます。ノードの中の単語は、累算する集計値の種類を示します。 GROUP BY 句と併用して集計値が累算される場合、集計関数ノードはグループ分け (Group Break) ノードとグループ (Group) ノードの間に現れます。
 FCalc	GROUP BY 句が存在しない場合に、集計値の累算を処理します。
 SubQuery	メインクエリの単一のサブクエリによるデータ取得を処理します。このノードは、root のクエリプランを表示している場合には現れません。サブクエリを表示している場合にのみ現れます。
 Ordered Temp Table	テンポラリテーブルの作成、およびテンポラリテーブルからのデータ取得を処理します。このノード下のベーステーブルへの参照は、テンポラリテーブルの列の参照に変更されます。

ノードの記号	ノードの意味
	UNION および UNION ALL 操作を処理します。基になるクエリ実行プランを繰り返し実行し、UNION 結果セットのデータを取得します。 Plan Viewer は、最初のクエリをルート クエリ、2 番目のクエリをサブクエリ 1、以下同様として、UNION を表示します。

メモ： Plan Viewer のサイズの縦横比を変更すると、それに応じてノードを表す記号の縦横比も変わります。そのため、ノードの表示は上記の表の例と同じように見えるとは限りません。

ノードの詳細

以下のクエリ プラン ノードをダブルクリックすると、ポップアップ ウィンドウが現れ、詳細な情報が表示されます。

- テーブル
- フィルター
- サブクエリ
- 順序付きテンポラリ テーブル

上記以外のノードをダブルクリックしても、ノードに関する詳しい情報は提供されません。詳細情報を持つノード上にマウス カーソルが置かれると、カーソルが手の形に変わります。

次の表では、ノードに表示される詳細情報の種類について説明します。

ノードの種類	詳細情報
テーブル	• テーブルの名前 • テーブル内の合計行数 • 読み取る予測行数 • 範囲の情報。範囲情報は、ベース テーブルが JOIN の右側にあり、テーブルからのデータの取得をインデックスの使用によって最適化できる場合にのみ使用されます。範囲情報には次のものが含まれます。 • 取得された列 • 範囲の取得の開始に使用された値（通常は、別のテーブルおよび列の値になります）と取得の終了に使用された値 • 実行する最初の演算。たとえば、より大きい（GT）、以上（GE）、より小さい（LT）、など。 • 中止する場合の判断に用いる比較方法（GT、GE など）
フィルター (Normal)	行の評価に使用される条件のテキスト表現。ワイド文字データが存在する場合、そのデータは正しく表示されます。
フィルター (Range)	• 使用されるインデックスに関する情報 • フィルターが返される行のセットを絞り込む方法 • テーブルからの最初の読み取りタイプ（GT、GE など） • これ以上のレコードの読み取りを停止するために、TRUE と評価されなければならない条件（GT、GE など）
サブクエリ	サブクエリの種類と、サブクエリに対して実行される最適化
順序付きテンポラ リ テーブル	• テンポラリ テーブルに含まれている列の一覧 • その列がテンポラリ テーブルの行の順序付けに使用されている列（キー）かどうか、あるいはツリーに渡す値であるかどうかを示す印

Plan Viewer にも、プランをさまざまなズーム レベルで表示したり、サブクエリがある場合にはそれを表示したりするためのメニュー コマンドが含まれています。

Query Plan Viewer のタスク

ここでは、以下の作業について説明します。

- [クエリ プランを作成するには](#)
- [Query Plan Viewer を起動するには](#)

-
- クエリ プランを表示するには
 - クエリ プランを XML ファイルへエクスポートするには
 - Plan Viewer でクエリ プランの表示サイズを調整するには
 - Plan Viewer でクエリ プランをスクロールするには
 - 変更されたクエリ プランを再読み込みするには
 - クエリ プラン ノードの詳細を表示するには
 - クエリ プランのサブクエリを表示するには

クエリ プランを作成するには

クエリ プランは、次の 2 つの方法で作成することができます。SET QRYPLAN コマンドを使用するか、またはカーソルを使用してクエリ ステートメントを選択し、右クリックしてメニューから [ログ クエリ プラン] を選択します。

SET QRYPLAN の使用方法

1. **SET QRYPLAN = on** を実行して、クエリ プランの作成を有効にします。
2. SET QRYPLANOUTPUT ステートメントを実行して、クエリ プラン ファイルの場所と名前を指定します。

[Query Plan の設定](#)を参照してください。

3. SELECT、INSERT、UPDATE、または DELETE ステートメントを実行します（対応するクエリ プランを作成します）。
4. **SET QRYPLAN = off** を実行して、クエリ プランの作成を無効にします。

ログ クエリ プランの使用方法

1. 次のいずれかを実行します。
 - SELECT、INSERT、UPDATE、または DELETE ステートメントを右クリックします。
 - カーソルを使用して SELECT、INSERT、UPDATE、または DELETE ステートメントを 1 つ以上選択し、選択範囲を右クリックします。
2. 開いたメニューで [ログ クエリ プラン] を選択します。

Query Plan Viewer を起動するには

1. 以下の操作のいずれかを実行します。
 - Zen Control Center で、[ツール] > [Query Plan Viewer] をクリックします。
 - Zen¥bin ディレクトリにある **w3sqlqpv.exe** ファイルを実行します。

クエリ プランを表示するには

1. Query Viewer で、[ファイル] > [開く] メニューをクリックします。
2. 目的のクエリ プラン ファイルの場所へ移動してそのファイルを選択し、[開く] ボタンをクリックします。

Query Viewer のタイトル バーには、開いているクエリ プランの数と現在表示されているプランの情報が示されます。
3. 2 つ以上のクエリ プランが開いている場合は、[表示] メニューから以下のコマンドを使用してウィンドウ間を移動できます。
 - [先頭] (または Ctrl+F) : 最初に読み込んだクエリ プランを表示します。
 - [末尾] (または Ctrl+L) : 最後に読み込んだクエリ プランを表示します。
 - [次] (または Ctrl+N) : 次に新しいクエリ プランを表示します。
 - [前] (または Ctrl+P) : 次に古いクエリ プランを表示します。
 - [ジャンプ] (または Ctrl+G) : 読み込んだプランの序数を基に、クエリ プランを表示します。

クエリ プランを XML ファイルへエクスポートするには

1. Query Viewer で、[ファイル] > [XML へエクスポート] をクリックします。
2. 目的のクエリ プラン XML ファイルの場所へ移動してそのファイルを選択し、[開く] ボタンをクリックします。

メモ : 同じダイアログ ボックスで、新しい XML ファイルの名前を指定することもできます。

ヒント : [XML へエクスポート] メニュー項目が有効になるのは、クエリ プランがビューアーに読み込まれている場合のみです。

次の表では、SQL クエリから派生した XML ファイルのスキーマについて説明します。

要素および属性	説明	親要素	子要素
<QPF filename= <i>filename</i> > filename : QPF ファイル のパスと名前。	XML ファイルのエク ポート元となる QPF ファ イル。1 つの XML ファイル に 1 つの QPF ファイル が対応します。	ヘッダー情報	<Query>
<Query number= <i>number</i> > number : Query Plan Viewer で表示されるク エリ番号。1 番目は 1、2 番目は 2 というように番 号が付けられます。	<QPF> ファイルのクエリ。 1 つの XML ファイルにつ き、少なくとも 1 つのク エリが対応します。	<QPF>	<SQL> <TreeRoot>
<SQL>	プランの生成に使用する SQL ステートメント。 SQL スクリプトが、大文 字 N で始まる Unicode 文 字の文字列リテラルを宣 言している場合、そのプ レフィックスは <SQL> 要 素に表示されます。 <Filter> の子要素 <Properties> も参照して ください。	<Query>	

要素および属性	説明	親要素	子要素
<code><TreeRoot name=<i>name</i>></code> <code>name</code> : "Root Query" または "Subquery <i>X</i> ".	ルート クエリまたはサブクエリのどちらかを示します。	<code><Query></code>	以下のすべてのノード要素: <code><Join></code> <code><Filter></code> <code><Base></code> <code><Distinct></code> <code><Set></code> <code><FCalc></code> <code><Group></code> <code><GroupBreak></code> <code><OrderedTempTable></code> <code><Union></code> <code><Subquery></code>
ノードの要素	それぞれの要素がクエリプラン ツリー内の 1 つのノードです (SQL ステートメントの一部)。	<code><TreeRoot></code> 、または別のノード要素の <code><Child></code> 、 <code><LeftChild></code> 、 <code><RightChild></code>	
<code><Join></code>		<code><TreeRoot></code>	<code><Text></code> <code><Properties></code> <code><LeftChild></code> <code><RightChild></code>
<code><Filter></code>	SQL スクリプトが、大文字 N で始まる Unicode 文字の文字列リテラルを宣言している場合、そのプレフィックスは <code><Properties></code> 子要素には表示 されません 。 <code><SQL></code> も参照してください。	<code><TreeRoot></code>	<code><Text></code> <code><Properties></code> <code><Child></code>
<code><Base></code>	ダイアグラム ツリー (樹形図) における葉を表します。	<code><TreeRoot></code>	<code><Text></code> <code><Properties></code>
<code><Distinct></code>		<code><TreeRoot></code>	<code><Properties></code> <code><Child></code>

要素および属性	説明	親要素	子要素
<Set>		<TreeRoot>	<Text> <Properties> <SetString> <Child>
<FCalc>		<TreeRoot>	<Properties> <LeftChild> <RightChild>
<Group>		<TreeRoot>	<Properties> <Child>
<GroupBreak>		<TreeRoot>	<Properties> <Child>
<OrderedTempTable>		<TreeRoot>	<Properties> <Child>
<Union>		<TreeRoot>	<Properties> <Child>
<Subquery>		<TreeRoot>	<Properties> <Child>
ノードの要素の子	多様。ノードに関する情報の追加、またはノードタグの子へのリンク。	多様	
<Text>		<Join> <Filter> <Set> <Base> (オプション)	
<Properties>		すべてのノード要素	
<SetString>		<Set>	

要素および属性	説明	親要素	子要素
<Child>		<Filter> <Distinct> <Set> <Group> <GroupBreak> <OrderedTempTable> > <Union> <Subquery>	
<LeftChild>		<Join> <FCalc>	
<RightChild>		<Join> <FCalc> (オプション)	

Plan Viewer でクエリ プランの表示サイズを調整するには

〔表示〕をクリックしてから、変更したいサイズ変更コマンドをクリックします。

- 〔自動調整〕：クエリ プランのサイズを変更して、Plan Viewer でプラン全体が表示できるようにします。ウィンドウのサイズを変更するとビューのサイズも変更されます。クエリ プランを表示するときのデフォルトは〔自動調整〕です。
- 〔100%〕、〔50%〕、または〔25%〕：クエリ プランのサイズを指定したパーセンテージに変更します。
- 〔拡大率〕：クエリ プランのサイズを、指定したパーセンテージに変更します。
- 〔拡大 (-)〕または〔縮小 (+)〕：クエリ プランのサイズを大きくする（拡大）か、または、小さく（縮小）します。拡大率は 5% から 500% の間で指定できます。

Plan Viewer でクエリ プランをスクロールするには

〔表示〕をクリックしてから、希望するスクロールコマンドをクリックします。

- 〔右へスクロール〕（または右矢印）：ウィンドウの右側へ向かってスクロールします。
- 〔左へスクロール〕（または左矢印）：ウィンドウの左側へ向かってスクロールします。

-
- [上へスクロール] (または上矢印) : ウィンドウの上側へ向かってスクロールします。
 - [下へスクロール] (または下矢印) : ウィンドウの下側へ向かってスクロールします。

変更されたクエリ プランを再読み込みするには

Query Viewer で [ファイル] > [更新] をクリックすると、現在読み込まれているクエリ プラン ファイルを再読み取りします。

クエリ プラン ノードの詳細を表示するには

Plan Viewer で、次のノードのいずれかをダブルクリックします。

- テーブル
- フィルター
- サブクエリ
- 順序付きテンポラリ テーブル

[ノードの詳細](#)を参照してください。

クエリ プランのサブクエリを表示するには

Plan Viewer で、[サブクエリ] をクリックしてからサブクエリの番号をクリックします (メイン クエリ内の最初のサブクエリは「サブクエリ 1」に対応し、2 番目のクエリは「サブクエリ 2」、以下同様に対応します)。クエリ プランには、いくつものサブクエリが含まれることもあれば、サブクエリなしの場合もあります。クエリ プランに対するすべてのサブクエリが [サブクエリ] メニューに現れます。サブクエリを選択すると、その名前が Plan Viewer のタイトル バーのかっこ内に表示されます。

クエリ プランの検査およびクエリ パフォーマンスの評価

Query Plan Viewer は特に、作成したクエリをテストして、そのクエリをデータベース エンジンがどのように実行するかを調べる、プロジェクトの開発段階で役に立ちます。それぞれのクエリを作成し、クエリ プラン ファイルを 1 つ生成してから、各プランを検査することができます。各クエリの情報を基に、インデックスを追加したり削除した

りして、その変更による影響を調べることができます。また、クエリを変更して、ステートメントの構文に加えた変更がパフォーマンスに影響を及ぼすかどうかを調べることができます。

比較するクエリ プランの例を作成する

例として、以下の手順に従って、Zen で提供されるサンプル データベース Demodata のいくつかのテーブルを使用することにより、Query Plan Viewer の使い方を実演できます。

比較のために、Enrolls テーブルからインデックスを 1 つ削除し、クエリを実行してクエリ プラン ファイルを作成した後、Enrolls にインデックスを追加し直し、クエリをもう一度実行して比較クエリ プラン ファイルを作成します。

1. Zen Control Center (ZenCC) で、クエリ プランの作成を有効にする SQL ステートメントと、クエリ プラン ファイルの名前を **example1.qpf** と指定する SQL ステートメントを実行します。 [Query Plan の設定](#) を参照してください。
2. ZenCC で、Demodata データベースに対して次のクエリを実行します。

```
DROP INDEX Enrolls.ClassID
```

Demodata はインストールするときに最適化されるため、Enrolls テーブルの Class_ID 列からインデックスを削除する必要があります。

3. ZenCC で、Demodata データベースに対して次のクエリを実行します。

```
SELECT Student.ID, Class.Name, Course.Credit_Hours FROM Student, Enrolls, Class, Course WHERE  
Student.ID = Enrolls.Student_Id AND Enrolls.Class_ID = Class.ID AND Class.Name = Course.Name
```

このクエリは、在籍するすべての学生と、学生が登録している講座、および各コースの履修単位時間を取得します。

4. ZenCC で、クエリ プラン ファイルの名前を **example2.qpf** と指定します。 [Query Plan の設定](#) を参照してください。
5. ZenCC で、Demodata データベースに対して次のクエリを実行します。

```
CREATE INDEX ClassID ON Enrolls(Class_ID)
```

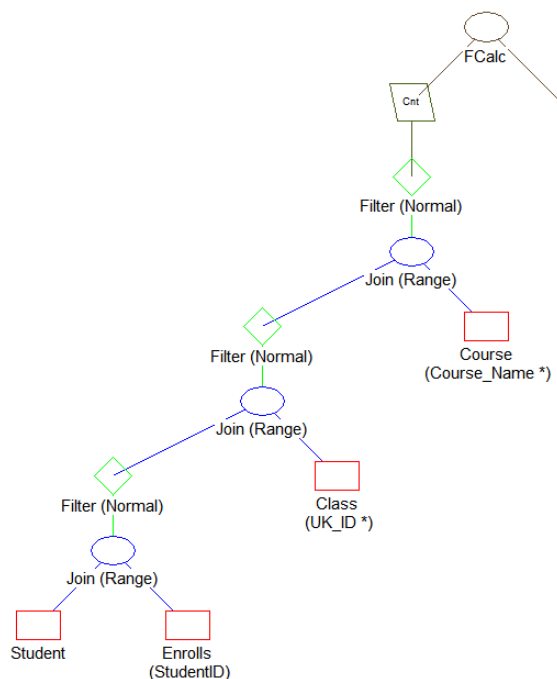
6. ZenCC で、Demodata データベースに対して次のクエリを実行します。

```
SELECT Student.ID, Class.Name, Course.Credit_Hours FROM Student, Enrolls, Class, Course WHERE  
Student.ID = Enrolls.Student_Id AND Enrolls.Class_ID = Class.ID AND Class.Name = Course.Name
```

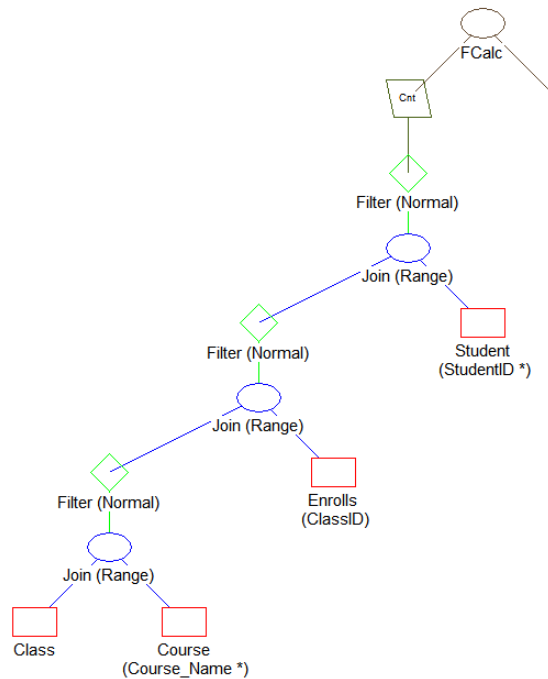
クエリの実行が速くなったことに気付いてください。

クエリ プランの例を表示する

Query Plan Viewer で、[ファイル] > [開く] を使用して example1.qpf() を開きます。次のようなものが表示されます。



比較のために、Query Plan Viewer で example2.qpf を開きます。次のようなものが表示されます。



このプランについて、次の点に注目してください。

- Class テーブルからレコードがスキャンされます。
- Course.Course_Name インデックスを使用する Class.Name 値に基づいて、Course テーブルからレコードが取得されます。
- Class.ID 値を基に Enrolls テーブルをスキャンして、Enrolls テーブルからレコードが取得されます。
- Student.ID インデックスを使用して、Enrolls.Student_Id 値を基に Student テーブルからレコードが取得されます。
- Enrolls からのデータの選択には、新しく作成されたインデックス ClassID を使用します。

この例と同じように自分で作成したクエリを比較すれば、どのような構文および構造が自分の要求に適合しているかを判断することができます。

