



Database Compatibility for Oracle Developers Reference Guide

リリース *14.1.0*

EnterpriseDB

2021 年 12 月 14 日

目次

第 1 章	The SQL Language	3
1.1	SQL Syntax	3
1.1.1	Lexical Structure	3
1.1.2	Identifiers and Key Words	4
1.1.3	Constants	5
1.1.4	Comments	7
1.2	Data Types	8
1.2.1	Numeric Types	9
1.2.2	Character Types	11
1.2.3	Binary Data	12
1.2.4	Date/Time Types	13
1.2.5	Boolean Types	17
1.2.6	XML Type	17
1.3	Functions and Operators	18
1.3.1	Logical Operators	18
1.3.2	Comparison Operators	19
1.3.3	Mathematical Functions and Operators	20
1.3.4	String Functions and Operators	25
1.3.5	Pattern Matching String Functions	36
1.3.6	Pattern Matching Using the LIKE Operator	40
1.3.7	Data Type Formatting Functions	41
1.3.8	Date/Time Functions and Operators	62
1.3.9	Sequence Manipulation Functions	80
1.3.10	Conditional Expressions	81
1.3.11	Aggregate Functions	84
1.3.12	Subquery Expressions	98
1.3.13	Identifier Functions	100
1.3.14	Bitwise Functions	104
第 2 章	System Catalog Tables	107
2.1	デュアル	107
2.2	edb_dir	107
2.3	edb_password_history	108
2.4	edb_policy	108
2.5	edb_profile	110
2.6	edb_variable	112
2.7	pg_synonym	112
2.8	product_component_version	113

</div>

オラクルのデータベース互換性とは、アプリケーションコードをほとんどまたはまったく変更せずに、アプリケーションがオラクル環境および EDB Postgres Advanced Server（Advanced Server）環境で実行されることを意味します。

このガイドでは、Advanced Server でサポートされている互換性のあるデータ型に関するリファレンス資料を提供します。以下に関する参照情報：

- 互換性のある SQL 言語構文は、* オラクル Developers SQL ガイドのデータベース互換性*で提供されています。
- 互換性のあるカタログ Views は、「オラクル Developers カatalog Views のデータベース互換性ガイド」に記載されています。

Advanced Server でオラクルデータベースと互換性のあるアプリケーションを開発するには、アプリケーションの構築で使用される機能に特に注意する必要があります。例、互換性のあるアプリケーションを開発するということは、以下を選択することを意味します。

オラクルデータベースと互換性のあるアプリケーションのデータベーステーブルを定義するデータ型オラクル SQL と互換性のある SQL ステートメントオラクルデータベースと互換性のある SQL ステートメントおよび手続き型ロジックで使用するシステムおよびビルトイン関数- データベースサーバを作成するストアードプロシージャ言語（SPL）-ストアードプロシージャ、関数、トリガー、およびパッケージソフトのアプリケーションロジック- Oracle のデータディクショナリと互換性のあるシステムカタログビュー

Advanced Server の互換性機能と拡張機能の詳細については、次の場所にある Advanced Server 文書の完全なライブラリを参照してください。

<https://www.enterprisedb.com/docs>

第1章 The SQL Language

</ div>

以下のセクションでは、オラクルデータベースと互換性のある Advanced Server SQL 言語のサブセットについて説明します。次の SQL 構文、データ型、および関数は、EDB Postgres Advanced Server とオラクルの両方で機能します。

Advanced Server の文書セットには、このガイドに含まれていない拡張機能（オラクルのデータベース互換性を提供しない、または Oracle スタイルのアプリケーションをサポートしない機能）の構文とコマンドが含まれています。

このセクションは、次のセクションで構成されています。

Advanced Server SQL 構文と言語要素の一般的な説明 - データ型 - 組み込み関数

1.1 SQL Syntax

</ div>

このセクションでは、SQL の一般的な構文について説明します。これは、データを定義および変更するために SQL コマンドがどのように適用されるかについての詳細を含む次の章を理解するための基盤を形成します。

1.1.1 Lexical Structure

</ div>

SQL 入力はシーケンスのコマンドで構成されます。**コマンド**は、セミコロン (;) で終了するシーケンスの**トークン**で構成されます。入力ストリームの終わりのコマンドを終了します。有効なトークンは、特定のコマンドの構文によって異なります。

トークンは、**キーワード**、**識別子**、**引用符で囲まれた識別子**、**リテラル**（または**定数**）、または**特殊文字記号**です。トークンは通常**空白**（スペース、タブ、改行）で区切られますが、あいまいさがない場合は必要ありません（通常、特殊文字が他のトークンタイプに隣接している場合のみです）。

さらに、**コメント**が SQL 入力で発生する可能性があります。それらはトークンではありません-それらは事実上空白と同等です。

例、次は（構文的に）有効な SQL 入力です。

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

これは、1 行に 1 つずつの 3 つのコマンドのシーケンスです（ただし、これは必須ではありません。1 行に複数のコマンドを含めることができ、通常はコマンドを行に分割できます）。

SQL 構文は、どのトークンがコマンドを識別し、どのトークンがオペランドまたはパラメーターであるかに関してあまり一貫性がありません。通常、最初のいくつかのトークンはコマンド名前であるため、上記の例では通常、SELECT、UPDATE、および INSERT コマンドについて説明します。しかし、インスタンス、UPDATE コマンドでは、特定の位置に表示されるために SET トークンが常に必要であり、INSERT のこの特定のバリエーションは、完了オーダーに VALUES トークンも必要です。各コマンドの正確な構文規則は、* オラクル Developers SQL ガイドのデータベース互換性*で説明されています。

1.1.2 Identifiers and Key Words

</ div>

上記の例の SELECT、UPDATE、または VALUES などのトークンは、**キー**、つまり SQL 言語で固定された意味を持つ単語の例です。トークン MY_TABLE および A は* identifiers の例です。使用されるコマンドに応じて、テーブル、列、またはその他のデータベースオブジェクトの名前を識別します。したがって、単に「名前」と呼ばれることもあります。キーワードと識別子は同じ字句構造*を持っています。つまり、言語がわからないとトークンが識別子かキーワードかを知ることができません。

SQL 識別子とキーは、文字（a-z または A-Z）で始まる必要があります。識別子またはキーワードの後続の文字は、文字、アンダースコア、数字（0-9）、ドル記号（\$）、またはシャープ記号（#）です。

識別子とキーワード名は大文字と小文字を区別しません。だから

```
UPDATE MY_TABLE SET A = 5;
```

次のように書くこともできます。

```
uPDaTE my_Table SeT a = 5;
```

よく使用される規則は、キーを大文字で、名前を小文字で書くことです。たとえば、

```
UPDATE my_table SET a = 5;
```

2 番目の種類の識別子があります：**区切り識別子**または**引用符で囲まれた識別子**。これは、任意の文字シーケンスを二重引用符（"）で囲むことで形成されます。区切り識別子は常に識別子であり、キーワードではありません。キーワードとして解釈されるため、テーブル名または列名前が予想される場所で使用すると解析エラーが発生します。この例は、次のように引用符で囲まれた識別子で記述できます。

```
UPDATE "my_table" SET "a" = 5;
```

引用符付き識別子には、数字コードゼロの文字を除く、任意の文字を含めることができます。

二重引用符を含めるには、2つのダブルクォート符を使用します。これにより、他の方法では不可能なテーブル名または列名（スペースやアンパサンドを含む名前など）をコンストラクトできます。長さの制限は引き続き適用されます。

識別子を引用符で囲むと大文字と小文字が区別されますが、引用符で囲まれていない名前は常に小文字に変換されます。例、識別子 `F00`、`foo`、および `"foo"` は Advanced Server によって同じと見なされますが、`"Foo"` および `"F00"` はこれら 3 つとは異なります。引用符で囲まれていない名前の小文字への折りたたみは、オラクルデータベースと互換性がありません。オラクル構文では、引用符で囲まれていない名前は大文字に変換されます。例、`foo` は `"foo"` ではなく `"F00"` と同等です。ポータブルアプリケーションを作成する場合は、常に特定の名前をクォートするか、決してクォートしないことをお勧めします。

1.1.3 Constants

</div>

Advanced Server の暗黙に型付けされる定数の種類は、`* strings および numbers *` です。定数は明示的な型で指定することもできます。これにより、システムによるより正確な表現形式とより効率的なハンドリングが可能になります。これらの選択肢は以下のサブセクションで議論されます。

文字列定数

SQL の文字列定数は、`'This is a string'` 例のシングルクォート (') で区切られた任意の文字シーケンスです。文字列定数内に一重引用符を含めるには、2つの隣接するシングルクォートを記述します（例：`'Dianne''s horse'`）。これは、二重引用符の文字 (") と同じではないことに注意してください。

数値定数

数値定数は、次の一般的な形式で受け入れられます。

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

ここで、`digits` は 1 つ以上の 10 進数数 (0~9) です。小数点を使用する場合、少なくとも 1 桁は 10 進数の前または後になければなりません。指数マーカー (e) が存在する場合は、その後少なくとも 1 桁が必要です。定数にスペースやその他の文字を埋め込みことはできません。先行するプラス記号またはマイナス符号は、実際には定数のパートと見なされないことに注意してください。これは、定数に適用される演算子です。

有効な数値定数の例を次に示します。

```
42
3.5
4.
```

(次のページに続く)

(前のページからの続き)

```
.001
5e2
1.925e-3
```

10 進数も指数も含まない数値定数は、その値が **INTEGER** 型（32 ビット）に適合する場合、最初は **INTEGER** 型であると推定されます。それ以外の場合、値が **BIGINT** 型（64 ビット）に収まる場合、**BIGINT** 型であると見なされます。それ以外の場合、**NUMBER** 型と見なされます。10 進数および/または指数を含む定数は、常に最初は **NUMBER** 型であると想定されます。

数値定数の最初に割り当てられたデータタイプは、型解決アルゴリズムの出発点にすぎません。ほとんどの場合、定数はコンテキストに応じて自動的に最も適切な型に強制されます。必要に応じて、次のセクションで説明するようにキャストすることにより、数値を特定のデータタイプとして解釈させることができます。

他のタイプの定数

キャスト

次の表記法を使用して、任意のタイプの定数を入力できます。

```
CAST('string' AS type)
```

文字列定数のテキストは、**type** という型の入力変換ルーチンに渡されます。結果は、示されたタイプの定数です。明示的な型キャストは、定数が必要な型にあいまいさがない場合（例、テーブル・カラムに直接割り当てられる場合）、省略できます。その場合、自動的に強制されます。

CAST を使用して、任意の式のランタイム型変換を指定することもできます。

キャスト（マルチセット）

MULTISET は **CAST** の拡張機能で、サブクエリの結果をネストされたテーブルタイプに変換します。概要は次のとおりです。

```
CAST ( MULTISET ( < subquery > ) AS < datatype > )
```

subquery は 1 つ以上の行を返すクエリーであり、**datatype** はネストされたテーブルタイプです。

CAST(MULTISET) は、データのコレクションをテーブルに格納するために使用されます。

例

次の例は、**MULTISET** の使用方法を示しています。

```
edb=# CREATE OR REPLACE TYPE project_table_t AS TABLE OF VARCHAR2(25);
CREATE TYPE
edb=# CREATE TABLE projects (person_id NUMBER(10), project_name VARCHAR2(20));
CREATE TABLE
edb=# CREATE TABLE pers_short (person_id NUMBER(10), last_name VARCHAR2(25));
CREATE TABLE
```

```

edb=# INSERT INTO projects VALUES (1, 'Teach');
INSERT 0 1
edb=# INSERT INTO projects VALUES (1, 'Code');
INSERT 0 1
edb=# INSERT INTO projects VALUES (2, 'Code');
INSERT 0 1
edb=# INSERT INTO pers_short VALUES (1, 'Morgan');
INSERT 0 1
edb=# INSERT INTO pers_short VALUES (2, 'Kolk');
INSERT 0 1
edb=# INSERT INTO pers_short VALUES (3, 'Scott');
INSERT 0 1
edb=# COMMIT;
COMMIT

```

```

edb=# SELECT e.last_name, CAST(MULTISET(
edb(#   SELECT p.project_name
edb(#   FROM projects p
edb(#   WHERE p.person_id = e.person_id
edb(#   ORDER BY p.project_name) AS project_table_t)
edb=# FROM pers_short e;
  last_name | project_table_t
-----+-----
Morgan     | {Code,Teach}
Kolk       | {Code}
Scott      | {}
(3 rows)

```

1.1.4 Comments

</div>

コメントは、二重ダッシュで始まり、行の終わり拡張任意の文字シーケンスです。例：

```
-- This is a standard SQL comment
```

または、C スタイルのブロックコメントを使用できます。

```

/* multiline comment
 * block
 * /

```

ここで、コメントは/ *で始まり、マッチング */の出現まで拡張されます。

1.2 Data Types

</ div>

次の表に、ビルトインの汎用データ型を示します。

Name	Alias	Description
<i>BLOB</i>	<i>LONG RAW, RAW(n), BYTEA</i>	バイナリデータ
<i>BOOLEAN</i>		論理ブール (true / false)
<i>CHAR [(n)]</i>	<i>CHARACTER [(n)]</i>	`n` 文字の固定長文字の並び
<i>CLOB</i>	<i>LONG, LONG VAR- CHAR</i>	長い文字の並び
<i>DATE</i>	<i>TIMESTAMP</i>	秒までの日付と時刻
<i>DOUBLE PRECISION</i>	<i>FLOAT, FLOAT(25) – FLOAT(53)</i>	精度浮動小数点数
<i>INTEGER</i>	<i>INT, BI- NARY_INTEGER, PLS_INTEGER</i>	符号付き 4 バイト整数
<i>NUMBER</i>	<i>DEC, DECIMAL, NUMERIC</i>	オプションの 10 進数の桁数を含む正確な数値
<i>NUMBER(p [, s])</i>	<i>DEC(p [, s]), DEC- IMAL(p [, s]), NU- MERIC(p [, s])</i>	最大精度の正確な数値 `p`、およびオプションのスケール `s`
<i>REAL</i>	<i>FLOAT(1) – FLOAT(24)</i>	単一精度浮動小数点数
<i>TIMESTAMP [(p)]</i>		オプションの日付と時刻、分数秒精度、 <i>p</i>
<i>TIMESTAMP [(p)] WITH TIME ZONE</i>		オプションの分数秒精度 `p`、およびタイムゾーンを使用した日付と時刻
<i>VARCHAR2(n)</i>	<i>CHAR VARYING(n), CHARACTER VARY- ING(n), VARCHAR(n)</i>	最大長さが `n` 文字の可変長文字の並び
<i>XMLTYPE</i>		XML データ
		NA

1.2.1 Numeric Types

</ div>

数値型は、4 バイトの整数、4 バイトと 8 バイトの浮動小数点数、および固定精度の小数で構成されます。次の表に、使用可能なタイプを示します。

Name	Storage Size	Description	Range
<i>BINARY_INTEGER</i>	4 bytes	符号付き整数、`INTEGER` のエイリアス	- 2,147,483,648 to +2,147,483,647
<i>DOUBLE PRECISION</i>	8 bytes	可変精度、不正確	15 decimal digits precision
<i>INTEGER</i>	4 bytes	整数の通常の見方	- 2,147,483,648 to +2,147,483,647
<i>NUMBER</i>	Variable	ユーザー指定の精度、正確	Up to 1000 digits of precision
<i>NUMBER(p [, s])</i>	Variable	最大精度の正確な数値 `p`、およびオプションのスケール `s`	Up to 1000 digits of precision
<i>PLS_INTEGER</i>	4 bytes	符号付き整数、`INTEGER` のエイリアス	- 2,147,483,648 to +2,147,483,647
<i>REAL</i>	4 bytes	可変精度、不正確	6 decimal digits precision
<i>ROWID</i>	8 bytes	符号付き 8 ビット 整数。	-9223372036854775808 to 9223372036854775807
		NA	

次のセクションでは、タイプについて詳しく説明します。

整数型

BINARY_INTEGER、INTEGER、PLS_INTEGER、および ROWID 型は、表**数値型**で指定されているように、整数（小数部を含まない）を格納します。許容レンジ外の値を保存しようとすると、エラーが発生します。

任意の精度数

NUMBER 型は、事実上無制限の桁数の精度を格納し、正確に計算を実行できます。正確な金額が必要な金額やその他の数量を保存する場合に特にお勧めです。ただし、NUMBER 型は、次のセクションで説明する浮動小数点型に比べて非常に遅いです。

以下では、これらの用語を使用します。NUMBER の **scale** は、小数点の右側の 10 進数パートの 10 進数数のカウントです。NUMBER の **precision** は、整数の有効数字数の合計数、つまり、10 進数の両側の桁数です。したがって、数値 23.5141 の精度は 6、スケールは 4 です。整数のスケールはゼロと見なすことができます。

NUMBER タイプの精度とスケールの両方を構成できます。NUMBER 型の列を宣言するには、構文を使用します

```
NUMBER(precision, scale)
```

精度は正、スケールはゼロまたは正でなければなりません。あるいは、

```
NUMBER(precision)
```

スケール 0 を選択します。精度またはスケールなしで NUMBER を指定すると、精度の実装制限まで、精度およびスケールの数値を格納できる列が作成されます。この種類の列は、特定のスケールに入力値を強制しませんが、宣言されたスケールを持つ NUMBER 列は、そのスケールに入力値を強制します。（SQL 標準では、デフォルトのスケール 0、つまり整数精度への強制が必要です。移植性を最大限に高めるには、精度とスケールを明示的に指定するのが最善です。）

値の精度またはスケールが宣言された列の精度またはスケールより大きい、システムは値を丸めようとします。宣言された制限を満たすために値を丸めることができない場合、エラーが発生します。

浮動小数点型

データ型 REAL および DOUBLE PRECISION は、* inexact *、可変精度数値型です。実際には、これらの型は通常、基になるプロセッサ、オペレーティングシステム、およびコンパイラがサポートするエクステンツで、バイナリ浮動小数点演算（それぞれ単精度および倍精度）の IEEE 標準 754 の実装です。

不正確とは、一部の値を正確に内部フォーマットに変換できず、近似値として保存されることを意味します。そのため、値の保存と印刷はわずかな矛盾を示す場合があります。これらのエラーを管理し、計算を通じてどのように伝播するかは、数学およびコンピュータサイエンスの分野全体のサブジェクトであり、以下の点を除き、ここではこれ以上説明しません。

正確なストレージと計算が必要な場合（金額など）、代わりに NUMBER タイプを使用してください。

特に境界の場合の特定の動作（無限大、アンダーフロー）に依存している場合、これらの型を使用して重要な何かについて複雑な計算を行いたい場合は、実装を慎重に評価する必要があります。

2つの浮動小数点値が等しいかどうかを比較すると、期待どおりに機能する場合と機能しない場合があります。

ほとんどのプラットフォームでは、REAL 型のレンジは少なくとも 1E-37 から 1E+37 までで、精度は 10 進数で 6 桁以上です。DOUBLE PRECISION 型は通常、少なくとも 15 桁の精度で 1E-307 から 1E+308 のレンジです。値がラージすぎるか小さすぎると、エラーが発生します。入力数値の精度が高すぎる場合、丸めが行われる場合があります。ゼロとは異なるものとして表現できないゼロに近すぎる数値は、アンダーフローエラーを引き起こします。

Advanced Server は、不正確な数値型を指定するための SQL 標準表記 FLOAT および FLOAT(p) もサポートしています。ここで、p は最小許容精度を 2 進数で指定します。Advanced Server は、REAL タイプの選択として FLOAT(1)～FLOAT(24) を受け入れ、DOUBLE PRECISION の選択として FLOAT(25)～FLOAT(53) を受け入れます。許容レンジ外の p の値は、エラーを引き起こします。精度が指定されていない FLOAT は、DOUBLE PRECISION を意味します。

1.2.2 Character Types

</div>

次の表に、Advanced Server で使用可能な汎用文字タイプを示します。

Name	Description
<i>CHAR</i> (<i>n</i>)	固定長の文字の並び、`n`で指定されたサイズに空白が埋め込まれます
<i>CLOB</i>	最大ギガバイトの大型可変長
<i>LONG</i>	長さ無制限。
<i>NVAR-CHAR</i> (<i>n</i>)	可変長国文字の並び、制限付き
<i>NVAR-CHAR2</i> (<i>n</i>)	可変長国文字の並び、制限付き
<i>STRING</i>	`VARCHAR2`のエイリアス
<i>VARCHAR</i> (<i>n</i>)	制限付きの可変長文字の並び（非推奨と見なされますが、互換性のためにサポートされています）
<i>VAR-CHAR2</i> (<i>n</i>)	可変長文字の並び、制限付き
	NA

n は正の整数です。これらのタイプは、最大 n 文字の文字列を保存できる長さ。n の長さを超える値を割り当てようとすると、余分な文字がすべてスペースでない限りエラーになります。その場合、文字列は最大長さに切り捨てられます。

これらのタイプのデータのストレージ要件は、実際の文字列に文字列が 127 バイトより小さい場合は 1 バイト、文字列が 127 バイトプラスの場合は 4 バイトです。CHAR の場合、パディングにはストレージも必要です。長い文字列はシステムによって自動的に圧縮されるため、ディスク上の物理的要件は少なくなります。長い値はバックグラウンドテーブルに格納されるため、短い列値への迅速なアクセスを妨げません。

データベースのキャラクタセットは、テキスト値の保存に使用されるキャラクタセットを決定します。

CHAR

n の値を指定しない場合、n はデフォルトで 1 になります。割り当てられる文字列が n より短い場合、CHAR 型の値は指定された幅 (n) にスペースが埋め込まれ、そのように格納および表示されます。

パディングスペースは意味的に重要でないものとして扱われます。つまり、CHAR 型の 2 つの値を比較する場合、末尾のスペースは無視され、CHAR 値を他の文字列型のいずれかに変換するときに削除されます。

長すぎる値を CHAR(n) 型に明示的にキャストすると、エラーは発生せずに値が n 文字に切り捨てられます (SQL 標準で指定)。

VARCHAR, VARCHAR2, NVARCHAR and NVARCHAR2

割り当てる文字列が n より短い場合、VARCHAR、VARCHAR2、NVARCHAR、および NVARCHAR2 型の値は、パディングなしで短い文字列を格納します。

注：末尾のスペースは、VARCHAR 値で意味的に重要です。

値を明示的に VARCHAR 型にキャストすると、長すぎる値はエラーを発生させることなく n 文字に切り捨てられます (SQL 標準で指定されているとおり)。

CLOB

CLOB 型にラージ文字の並びを格納できます。CLOB は、長さの制限が指定されていないことを除いて、意味的に VARCHAR2 と同等です。通常、最大文字列長さがわからない場合は、CLOB 型を使用する必要があります。

CLOB 型に格納できる最長の文字の並びは約 1 ギガバイトです。

注：CLOB データタイプは、実際には PostgreSQL の TEXT データタイプに基づく DOMAIN です。DOMAIN については、< <https://www.postgresql.org/docs/current/ 静的/sql-createdomain.html> > の PostgreSQL コア文書を参照してください。

したがって、CLOB タイプの使用は、約 1 ギガバイトの最大サイズなど、TEXT に対して実行できる処理によって制限されます。

CLOB データタイプを使用する代わりに、大量のデータを使用するには、pg_largeobject システムカタログに依存する PostgreSQL * Large Objects * 機能を使用します。ラージオブジェクトの詳細については、次の PostgreSQL コア文書を参照してください。

< <https://www.postgresql.org/docs/current/ 静的/largeobjects.html> >

1.2.3 Binary Data

</div>

次の表に、バイナリ文字列をストレージできるデータ型を示します。

Name	Storage Size	Description
<i>BINARY</i>	The length of the binary string.	長さが 1～8300 の固定長さのバイナリ文字列。
<i>BLOB</i>	The actual binary string plus 1 byte if the binary string is less than 127 bytes, or 4 bytes if the binary string is 127 bytes or greater.	最大サイズがギガバイトの可変長バイナリ文字列。
<i>VARBINARY</i>	The length of the binary string	長さが 1～8300 の長さバイナリ文字列。
		NA

バイナリ文字列は、オクテット（またはバイト）のシーケンスです。バイナリ文字列は、次の 2 つの特性によって文字列と区別されます。1 つ目は、バイナリ文字列は値ゼロのオクテットとその他の「印刷不可」オクテット（32 から 126 のレンジ外のオクテットとして定義）を保存できることです。次に、バイナリ文字列の操作は実際のバイトを処理しますが、文字列のエンコーディングと処理はロケール設定に依存します。

1.2.4 Date/Time Types

</ div>

以下の日付/時刻型の説明では、テーブルが作成または変更されるたびに、設定パラメータ `edb_redwood_date` が `TRUE` に設定されていることを前提としています。

Advanced Server は、次の表に示す日付/時刻タイプをサポートしています。

Name	Storage Size	Description	Low Value	High Value	Resolution
<i>DATE</i>	8 bytes	日時	4713 BC	5874897 AD	1 second
<i>INTERVAL DAY TO SECOND [(p)]</i>	12 bytes	期間	- 178000000 years	178000000 years	1 microsecond / 14 digits
<i>INTERVAL YEAR TO MONTH</i>	12 bytes	期間	- 178000000 years	178000000 years	1 microsecond / 14 digits
<i>TIMESTAMP [(p)]</i>	8 bytes	日時	4713 BC	5874897 AD	1 microsecond
<i>TIMESTAMP [(p)] WITH TIME ZONE</i>	8 bytes	タイムゾーン付きの日付と時刻	4713 BC	5874897 AD	1 microsecond
		NA			

DATE がデータ定義言語 (DDL) コマンド CREATE TABLE または ALTER TABLE の列のデータタイプとして表示される場合、テーブル定義がデータベースに格納されるときに TIMESTAMP に変換されます。したがって、時間コンポーネントも日付とともに列に格納されます。

DATE が SPL 宣言セクションの変数のデータタイプ型、SPL プロシージャまたは SPL ファンクションの仮パラメータのデータタイプ、または SPL ファンクションの戻り値の型として表示される場合、常に TIMESTAMP に変換されるため、存在する場合、時間コンポーネントをハンドルできます。

TIMESTAMP は、秒フィールドに保持する小数桁数を指定するオプションの精度値 *p* を受け入れます。*p* の許容レンジは 0 から 6 で、デフォルトは 6 です。

TIMESTAMP 値が倍精度浮動小数点数（現在はデフォルト）として格納されている場合、有効な精度の制限は 6 より小さいになる場合があります。TIMESTAMP 値は、2000-01-01 の午前 0 時前後の秒として格納されます。マイクロ秒の精度は、2000-01-01 の数年以内の日付で達成されますが、それよりも遠い日付では精度が低下します。TIMESTAMP 値が 8 バイト整数（コンパイル時オプション）として格納されている場合、値の全レンジでマイクロ秒の精度を使用できます。ただし、8 バイトの整数のタイムスタンプには、上記の日付よりも限られた日付レンジ（紀元前 4713 年から西暦 294276 年まで）があります。

TIMESTAMP (*p*) WITH TIME ZONE は TIMESTAMP (*p*) に似ていますが、タイムゾーンも含まれます。

区間タイプ

INTERVAL 値は期間を指定します。INTERVAL タイプの値は、データの値を記述するフィールドで構成されます。次の表に、INTERVAL 型で許可されるフィールドを示します。

Field Name	INTERVAL Values Allowed
<i>YEAR</i>	Integer value (positive or negative)
<i>MONTH</i>	0 through 11
<i>DAY</i>	Integer value (positive or negative)
<i>HOURL</i>	0 through 23
<i>MINUTE</i>	0 through 59
<i>SECOND</i>	0 through 59.9(<i>p</i>) where 9(<i>p</i>) is the precision of fractional seconds

フィールドは、オーダー（YEARS から MONTHS、DAYS から HOURS、MINUTES、SECONDS）で表示する必要があります。

Advanced Server は、オラクルデータベースと互換性のある 2 つの INTERVAL タイプをサポートしています。

Advanced Server がサポートする最初のバリエーションは INTERVAL DAY TO SECOND [(*p*)] です。INTERVAL DAY TO SECOND [(*p*)] は、時間インターバルを日、時間、分、秒で保存します。

p は、second フィールドの精度を指定します。

Advanced Server は値を解釈します：

INTERVAL '1 2:34:5.678' DAY TO SECOND(3)

1 日、2 時間、34 分、5 秒、678 分の 1 秒。

Advanced Server は値を解釈します：

`INTERVAL '1 23' DAY TO HOUR`

1 日と 23 時間。

Advanced Server は値を解釈します：

`INTERVAL '2:34' HOUR TO MINUTE`

2 時間 34 分。

Advanced Server は値を解釈します：

`INTERVAL '2:34:56.129' HOUR TO SECOND(2)`

2 時間 34 分 56 秒 13 分の 1 秒。指定された精度のため、小数秒は 13 に切り上げられることに注意してください。

オラクルデータベースと互換性がある Advanced Server がサポートする 2 番目のバリエーションは `INTERVAL YEAR TO MONTH` です。このバリエーションは、年と月の時間インターバルを保存します。

Advanced Server は値を解釈します：

`INTERVAL '12-3' YEAR TO MONTH`

12 年 3 ヶ月。

Advanced Server は値を解釈します：

`INTERVAL '456' YEAR(2)`

12 年 3 ヶ月。

Advanced Server は値を解釈します：

`INTERVAL '300' MONTH`

25 年として。

日付/時刻入力

日付と時刻の入力は、ISO 8601 SQL 互換フォーマット、オラクルのデフォルトの `dd-MON-yy` フォーマット、および年、月、日のコンポーネントが曖昧でない限り、他の多くの形式で受け入れられます。ただし、あいまいさを避けるため、`TO_DATE` ファンクションの使用を強くお勧めします。

日付または時刻のリテラル入力は、テキスト文字列のようにシングルクォートで囲むニーズがあります。次の SQL 標準構文も使用できます。

```
type 'value'
```

`type` は `DATE` または `TIMESTAMP` のいずれかです。

`value` は、日付/時刻のテキスト文字列です。

日付

次のブロックは、1999 年 1 月 8 日に相当する日付のいくつかの可能な入力形式を示しています。

Example

January 8, 1999
1999-01-08
1999-Jan-08
Jan-08-1999
08-Jan-1999
08-Jan-99
Jan-08-99
19990108
990108

日付値は、DATE または TIMESTAMP の列または変数に割り当てることができます。日付の値に時刻の値が付加されていない場合、時間、分、および秒のフィールドはゼロに設定されます。

回

日付またはタイムスタンプの時間コンポーネントの例を、次の表に示します。

Example	Description
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	04:05 と同じ。AM は値に影響しません
04:05 PM	16:05 と同じ。入力時間は & lt; = 12 でなければなりません
	NA

タイムスタンプ

タイムスタンプの有効な入力は、日付と時刻の結合で構成されます。タイムスタンプの日付部分は、Dates セクションの下ブロックに示されている例のいずれかに従ってフォーマットできます。タイムスタンプの時間部分は、Times セクションの表に示されている例のいずれかに従ってフォーマットできます。

以下は、オラクルのデフォルトフォーマットに従うタイムスタンプの例です。

08-JAN-99 04:05:06

以下は、ISO 8601 標準に従うタイムスタンプの例です。

1999-01-08 04:05:06

日付/時刻出力

日付/時刻型のデフォルトの出力フォーマットは、オラクルデータベースと互換性のある* redwood 日付スタイル*と呼ばれる (dd-MON-yy)、または ISO 8601 フォーマットと呼ばれる (インタフェース) のいずれかです。データベース SQL Interactive などの JDBC を使用するアプリケーションは、常に日付を ISO 8601 フォームで提示します。PSQL などの他のアプリケーションは、日付を Redwood フォームで提示します。

次の表に、Redwood と ISO 8601 の 2 つのスタイルの出力形式の例を示します。

Description	Example
レッドウッドスタイル	31-DEC-05 07:37:16
ISO 8601 / SQL 標準	1997-12-17 07:37:16
NA	

内部

Advanced Server は、すべての日付/時刻の計算にユリウス日付を使用します。ユリウス日は、年の長さが 365.2425 日であるという仮定に基づいて、紀元前 4713 年以降の日付を正しく予測または計算します。

1.2.5 Boolean Types

</ div>

Advanced Server は、標準の SQL タイプ BOOLEAN を提供します。BOOLEAN には、TRUE または FALSE の 2 つの状態のいずれかのみを設定できます。3 番目の状態 UNKNOWN は、SQL の NULL 値で表されます。

Name	Storage Size	Description
BOOLEAN	1 byte	論理ブール (true / false)
		NA

真の状態を表すための有効なリテラル値は TRUE です。偽の状態を表すための有効なリテラルは FALSE です。

1.2.6 XML Type

</ div>

XMLTYPE データタイプは、XML データを格納するために使用されます。XML データを文字フィールドに格納することの利点は、入力値の整形形式をチェックすること、およびタイプセーフな操作を実行するためのサポート関数があることです。

XML タイプは、XML 標準で定義されている整形形式の「ドキュメント」と、XML 標準の稼動 XMLDecl? content で定義されている「コンテンツ」フラグメントを XML できます。大体、これは、コンテンツフラグメントが複数のトップレベル要素またはキャラクターノードを持つことができることを意味します。

!!! Note * オラクルは、XMLTYPE 列のコンテンツフラグメントのストレージをサポートしていません。

次の例は、XMLTYPE 列を持つテーブルへの行の作成と挿入を示しています。

```
CREATE TABLE books (  
  *      content      XMLTYPE  
);  
  
INSERT INTO books VALUES (XMLPARSE (DOCUMENT '<?xml  
version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>'));  
  
SELECT * FROM books;  
      *      content  
-----  
<book><title>Manual</title><chapter>...</chapter></book>  
(1 row)
```

1.3 Functions and Operators

</ div>

Advanced Server は、ビルトインデータ型用のラージの関数と演算子を提供します。

1.3.1 Logical Operators

</ div>

通常のロジカル演算子が利用可能です：AND、OR、NOT

SQL は、NULL 値が「不明」を表す 3 値のブールロジックを使用します。次の真理値表を確認してください。

AND/OR Truth Table

a	b	a AND b	a OR b
True	True	True	True
True	False	False	True
True	Null	Null	True
False	False	False	False
False	Null	False	Null
Null	Null	Null	Null

NOT Truth Table

a	NOT a
True	False
False	True
Null	Null

演算子 AND および OR は可換です。つまり、結果に影響を与えずに左右のオペランドをスイッチすることができます。

1.3.2 Comparison Operators

</ div>

通常の比較演算子を次の表に示します。

Operator	Description
<	未満
>	より大きい
<=	以下
>=	以上
=	等しい
<>	等しくない
!=	等しくない
	NA

比較演算子は、これが理にかなっているすべてのデータ型で使用できます。すべての比較演算子は、`1 < 2 < 3` などの `BOOLEAN`; 式の値を結果バイナリ演算子です（ブール値と 3 を比較する < 演算子がないため）。

比較演算子に加えて、特別な `BETWEEN` コンストラクトが利用可能です。

`a BETWEEN x AND y`

に等しい

`a >= x AND a <= y`

同様に、

`a NOT BETWEEN x AND y`

に等しい

`a < x OR a > y`

内部で最初の形式を 2 番目の形式に書き換えるのに必要な CPU サイクルを除いて、2 つのそれぞれの形式に違いはありません。

値が null かどうかを確認するには、構造を使用します

`expression IS NULL`

expression IS NOT NULL

NULL は NULL と「等しくない」ため、`expression = NULL` を書き込まないでください。(NULL 値は不明な値を表し、2 つの不明な値が等しいかどうかは不明です。) この動作は SQL 標準に準拠しています。

一部のアプリケーションでは、`expression` が NULL 値に評価される場合に `expression = NULL` が true を返すことを期待する場合があります。これらのアプリケーションは、SQL 標準に準拠するように変更することを強くお勧めします。

1.3.3 Mathematical Functions and Operators

</ div>

多くの Advanced Server タイプに数学演算子が用意されています。考えられるすべての順列に共通の数学的慣習がない型（日付/時刻型など）の場合、実際の動作については後続のセクションで説明します。

次の表に、使用可能な数学演算子を示します。

Operator	Description	Example	Result
+	追加	2 + 3	5
-	減算	2 - 3	-1
*	乗算	2 * 3	6
/	分割（次のノートを参照）	4 / 2	2
**	指数演算子	2 ** 3	8
	NA		

!!! Note * postgresql.conf ファイルの `db_dialect` 設定パラメータが `redwood` に設定されている場合、INTEGER データ型のペアの除算によって値が切り捨てられることはありません。次の例に示すように、小数の結果は保持されます。

```
edb=# SET db_dialect TO redwood;
SET
edb=# SHOW db_dialect;
db_dialect
-----
redwood
(1 row)

edb=# SELECT CAST('10' AS INTEGER) / CAST('3' AS INTEGER) FROM dual;
*      ?column?
-----
3.3333333333333333
(1 row)
```

この動作は、ネイティブの INTEGER データタイプがないオラクルデータベースと互換性があり、INTEGER データタイプの指定は内部的に NUMBER(38) に変換され、結果の小数部が保持されます。

db_dialect 設定パラメータが postgres に設定されている場合、INTEGER データ型のペアを分割すると、次の例に示すように値が切り捨てられます。

```
edb=# SET db_dialect TO postgres;
SET
edb=# SHOW db_dialect;
 db_dialect
-----
 postgres
(1 row)

edb=# SELECT CAST('10' AS INTEGER) / CAST('3' AS INTEGER) FROM dual;
?column?
-----
      *      3
(1 row)
```

この動作は、INTEGER、SMALLINT、または BIGINT データ型のペアを含む除算の結果が切り捨てられる PostgreSQL データベースと互換性があります。前の例に示すように、db_dialect が postgres に設定されている場合、同じ切り捨てられた結果が Advanced Server によって返されます。

ただし、db_dialect が redwood に設定されている場合でも、INTEGER データ型のペアによる除算のみで結果が切り捨てられないことに注意してください。INTEGER データデータタイプの有無にかかわらず、SMALLINT または BIGINT データ型のみを含む除算は、INTEGER および SMALLINT が除算に関与する次の図に示すように、小数部分を保持せずに PostgreSQL 形式で切り捨てられます。

```
edb=# SHOW db_dialect;
 db_dialect
-----
 redwood
(1 row)

edb=# SELECT CAST('10' AS INTEGER) / CAST('3' AS SMALLINT) FROM dual;
?column?
-----
      *      3
(1 row)
```

次の表に、使用可能な数学関数を示します。これらの関数の多くは、異なる引数タイプを持つマルチプルの形式で提供されます。特に明記されていない限り、ファンクションの特定のフォームは、その引数と同じデータタイプを返します。DOUBLE PRECISION データを処理する関数は、主にホストシステムの C ライブラリの上に実装されます。したがって、境界の場合の精度と振舞いはホストシステムによって異なるかもしれません。

Function	Return Type	Description	Example	Result
ABS(<i>x</i>)	Same as <i>x</i>	絶対値	ABS(-17.4)	17.4

次のページに続く

表 1 – 前のページからの続き

Function	Return Type	Description	Example	Result
<i>CEIL(DOUBLE PRECISION or NUMBER)</i>	Same as input	引数より小さいの最小整数	<i>CEIL(-42.8)</i>	-42
<i>EXP(DOUBLE PRECISION or NUMBER)</i>	Same as input	指数関数	<i>EXP(1.0)</i>	2.7182818284590452
<i>FLOOR(DOUBLE PRECISION or NUMBER)</i>	Same as input	引数より大きいの最大整数	<i>FLOOR(-42.8)</i>	43
<i>LN(DOUBLE PRECISION or NUMBER)</i>	Same as input	自然対数	<i>LN(2.0)</i>	0.6931471805599453
<i>LOG(b NUM- BER, x NUM- BER)</i>	NUMBER	ベース `b` への対数	<i>LOG(2.0, 64.0)</i>	6.0000000000000000
<i>MOD(y, x)</i>	Same as argument types	`y/x` の残り	<i>MOD(9, 4)</i>	1
<i>NVL(x, y)</i>	Same as argument types; where both arguments are of the same data type	`x` が null の場合、NVL は `y` を返します	<i>NVL(9, 0)</i>	9

次のページに続く

表 1 – 前のページからの続き

Function	Return Type	Description	Example	Result
<i>POWER(a, b)</i> <i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	`a`を`b`の累乗で累乗	<i>POWER(9.0, 3.0)</i>	729.0000000000000000
<i>POWER(a, b)</i> <i>NUMBER</i>	<i>NUMBER</i>	`a`を`b`の累乗で累乗	<i>POWER(9.0, 3.0)</i>	729.0000000000000000
<i>ROUND(DOUBLE PRECISION or NUMBER)</i>	Same as input	最も近い整数に丸める	<i>ROUND(42.4)</i>	42
<i>ROUND(v, s)</i> <i>NUMBER</i> <i>INTEGER</i>	<i>NUMBER</i>	`s`の 10 進数に丸めます	<i>ROUND(42.4382, 2)</i>	42.44
<i>SIGN(DOUBLE PRECISION or NUMBER)</i>	Same as input	引数の符号 (-1、0、+1)	<i>SIGN(-8.4)</i>	-1
<i>SQRT(DOUBLE PRECISION or NUMBER)</i>	Same as input	四角いルート	<i>SQRT(2.0)</i>	1.414213562373095

次のページに続く

表 1 – 前のページからの続き

Function	Return Type	Description	Example	Result
<i>TRUNC(DOUBLE PRECISION or NUMBER)</i>	Same as input	ゼロに向かって切り捨てます	<i>TRUNC(42.8)</i>	42
<i>TRUNC(number, NUM- BER, s IN- TE- GER)</i>	NUMBER	`s` の 10 進数に切り捨てます	<i>TRUNC(42.4382, 2)</i>	42.43
<i>WIDTH_BUCKET(NUM- BER, b1 NUM- BER, b2 NUM- BER, count IN- TE- GER)</i>	INTEGER	<i>b1</i> ~ <i>b2</i> のレンジで、`count` パケットと等深さのヒストグラムで `op` が割り当てられるパケットを返します	<i>WIDTH_BUCKET(5.35, 0.024, 10.06, 5)</i>	3
		NA		

次の表に、使用可能な三角関数を示します。すべての三角関数は引数を取り、DOUBLE PRECISION 型の値を結果ます。

Function	Description
<i>ACOS(x)</i>	逆コサイン
<i>ASIN(x)</i>	インバースサイン
<i>ATAN(x)</i>	逆接線
<i>ATAN2(x, y)</i>	`x/y` の逆接線
<i>COS(x)</i>	コサイン
<i>SIN(x)</i>	シネ
<i>TAN(x)</i>	タンジェント

1.3.4 String Functions and Operators

</ div>

このセクションでは、文字列値を調べて操作するための関数と演算子について説明します。このコンテキストの文字列には、CHAR、VARCHAR2、および CLOB 型の値が含まれます。特に明記されていない限り、以下にリストされている関数はすべてこれらのすべてのタイプで機能しますが、CHAR タイプを使用する場合の自動パディングの潜在的な影響には注意してください。一般に、ここで説明する関数は、最初にそのデータを文字列表現形式に変換することにより、非文字列型のデータでも機能します。

Function	Return Type	Description	Example	Result	
string		文字列	<i>CLOB</i>	String concatenation	'Enterprise'
<i>CONCAT(string, string)</i>	<i>CLOB</i>	文字列の結合	'a'		'b'
<i>HEXTORAW(varchar2)</i>	<i>RAW</i>	`VARCHAR2` 値を `RAW` value`に変換します	<i>HEXTORAW('303132')</i>	'012'	
<i>RAWTOHEX(raw)</i>	<i>VARCHAR2</i>	`RAW` 値を `HEXADECIMAL` 値に変換します	<i>RAWTOHEX('012')</i>	'303132'	
<i>INSTR(string, set, [start [, occurrence]])</i>	<i>INTEGER</i>	文字文字列 `string` の位置 `start` から開始して、文字列内の文字のセットの場所を検索し、セットの 1 番目、2 番目、3 番目などを検索します。セットが見つからない場合は `0` を返します。	<i>INSTR('PETER PIPER PICKED a PECK of PICKLED PEPERS', 'PI', 1, 3)</i>	30	

次のページに続く

表 2 – 前のページからの続き

Function	Return Type	Description	Example	Result	
<i>IN-STRB(string, set)</i>	<i>INTEGER</i>	`string`内の`set`の位置を返します。`set`が見つからない場合、`0`を返します。	<i>IN-STRB('PETER PIPER PICKED a PECK of PICK-LED PEP-PERS', 'PICK')</i>	<i>13</i>	
<i>IN-STRB(string, set, start)</i>	<i>INTEGER</i>	`start`で始まる`string`内の`set`の位置を返します。`set`が見つからない場合、`0`を返します。	<i>IN-STRB('PETER PIPER PICKED a PECK of PICK-LED PEP-PERS', 'PICK', 14)</i>	<i>30</i>	

次のページに続く

表 2 – 前のページからの続き

Function	Return Type	Description	Example	Result	
<i>IN-STRB(string, set, start, occurrence)</i>	<i>INTEGER</i>	<i>`start`</i> から始まる <i>`string`</i> 内の <i>`set`</i> の指定された <i>`occurrence`</i> の位置を返します。 <i>`set`</i> が見つからない場合、 <i>`0`</i> を返します。	<i>IN-STRB('PETER PIPER PICKED a PECK of PICKLED PEP-PERS', 'PICK', 1, 2)</i>	<i>30</i>	
<i>LOWER(string)</i>	<i>CLOB</i>	<i>`string`</i> を小文字に変換します	<i>LOWER('TOM')</i>	<i>tom</i>	
<i>SUB-STR(string, start [, count])</i>	<i>CLOB</i>	<i>`start`</i> から始まり、 <i>`count`</i> 文字までのサブストリングを抽出します。 <i>`count`</i> が指定されていない場合、文字列はスタートから最後までクリップされます。	<i>SUB-STR('This is a test', 6, 2)</i>	<i>is</i>	
<i>SUB-STRB(string, start [, count])</i>	<i>CLOB</i>	<i>`start`</i> と <i>`count`</i> がバイト数であることを除いて、 <i>`SUB-STR`</i> と同じです。	<i>SUB-STRB('abc', 3)</i> (assuming a double-byte character set)	<i>c</i>	

次のページに続く

表 2 – 前のページからの続き

Function	Return Type	Description	Example	Result	
<i>SUB-STR2(string, start [, count])</i>	<i>CLOB</i>	<i>`SUBSTR`</i> のエイリアス。	<i>SUB-STR2('This is a test',6,2)</i>	<i>is</i>	
<i>SUB-STR2(string, start [, count])</i>	<i>CLOB</i>	<i>`SUBSTRB`</i> のエイリアス。	<i>SUB-STR2('abc',3)</i> (assuming a double-byte character set)	<i>c</i>	
<i>SUB-STR4(string, start [, count])</i>	<i>CLOB</i>	<i>`SUBSTR`</i> のエイリアス。	<i>SUB-STR4('This is a test',6,2)</i>	<i>is</i>	

次のページに続く

表 2 – 前のページからの続き

Function	Return Type	Description	Example	Result	
<i>SUBSTR4</i> (string, start [, count)	<i>CLOB</i>	<i>⌘SUBSTRB⌘</i> のエイリアス。	<i>SUBSTR4</i> ('abc',3) (as- sum- ing a double- byte char- ac- ter set)	<i>c</i>	
<i>SUBSTRC</i> (string, start [, count)	<i>CLOB</i>	<i>⌘SUBSTR⌘</i> のエイリアス。	<i>SUBSTRC</i> ('This is a test',6,2)	<i>is</i>	
<i>SUBSTRC</i> (string, start [, count)	<i>CLOB</i>	<i>⌘SUBSTRB⌘</i> のエイリアス。	<i>SUBSTRC</i> ('abc',3) (as- sum- ing a double- byte char- ac- ter set)	<i>c</i>	

次のページに続く

表 2 – 前のページからの続き

Function	Return Type	Description	Example	Result	
TRIM([LEADING	TRAILING	BOTH] [文字] FROM 文字列)	<i>CLOB</i>	Remove the longest string containing only the characters (a space by default) from the start/end/both ends of the string.	<i>TRIM(BOTH 'x' FROM 'xTomxx')</i>
<i>LTRIM(string [, set])</i>	<i>CLOB</i>	指定された `string` の左側から `set` で指定されたすべての文字を削除します。`set` が指定されていない場合、デフォルトとしてスペースが使用されます。	<i>LTRIM('abcdefghi', 'abc')</i>	<i>defghi</i>	
<i>RTRIM(string [, set])</i>	<i>CLOB</i>	指定された `string` の右側から `set` で指定されたすべての文字を削除します。`set` が指定されていない場合、デフォルトとしてスペースが使用されます。	<i>RTRIM('abcdefghi', 'ghi')</i>	<i>abcdef</i>	
<i>UP-PER(string)</i>	<i>CLOB</i>	`string` を大文字に変換します	<i>UP-PER('tom')</i>	<i>TOM</i>	
		NA			

追加の文字列操作関数が利用可能であり、次の表にリストされています。それらのいくつかは、上記の表にリストされている SQL 標準の文字列関数を実装するために内部的に使用されます。

Function	Return Type	Description	Example	Result
<i>ASCII(string)</i>	<i>INTEGER</i>	引数の最初のバイトの ASCII コード	<i>ASCII('x')</i>	<i>120</i>
<i>CHR(INTEGER)</i>	<i>CLOB</i>	指定された ASCII コードの文字	<i>CHR(65)</i>	<i>A</i>

次のページに続く

表 3 – 前のページからの続き

Function	Return Type	Description	Example	Result
DE- CODE(expr, expr1a, expr1b [, expr2a, expr2b]... [, default])	Same as ar- gu- ment types of <i>expr1b</i> , <i>expr2b</i> ,..., <i>de- fault</i>	`expr`と `expr1a`, expr2a`などの最初 のマッチを検索し ます。マッチが見 つかった場合、対 応するパラメー タペア、`expr1b`, expr2b`などを返し ます。マッチが見 つからなかった場 合、`default`を返し ます。マッチする ものが見つからず、 `default`が指定され ていない場合、null を返します。	DE- CODE(3, 1,'One', 2,'Two', 3,'Three', 'Not found')	Three
INITCAP(string)	CLOB	各ワードの最初の文字を大文字 に変換し、残りを小文字に変換 します。単語は、英数字以外の 文字で区切られた英数字のシー ケンスです。	INITCAP('hi THOMAS')	Hi Thomas
LENGTH	INTEGER	文字列値の文字数を返します。	LENGTH('Côte d''Azur')	11
LENGTHC	INTEGER	この関クションの機能は `LENGTH`と同じです。ファン クション名前は互換性のために サポートされています。	LENGTHC('Côte d''Azur')	11
LENGTH2	INTEGER	この関クションの機能は `LENGTH`と同じです。ファン クション名前は互換性のために サポートされています。	LENGTH2('Côte d''Azur')	11
LENGTH4	INTEGER	この関クションの機能は `LENGTH`と同じです。ファン クション名前は互換性のために サポートされています。	LENGTH4('Côte d''Azur')	11
LENGTHB	INTEGER	指定された値を保持するために 必要なバイト数を返します。	LENGTHB('Côte d''Azur')	12

次のページに続く

表 3 – 前のページからの続き

Function	Return Type	Description	Example	Result
<i>LPAD(string, length INTEGER [, fill])</i>	<i>CLOB</i>	文字 'fill' (デフォルトではスペース) を先頭に追加して、'string' をサイズ 'length' に入力します。 'string' がすでに 'length' より長い場合は、切り捨てられます (右側)。	<i>LPAD('hi', 5, 'xy')</i>	<i>xyxhi</i>
<i>RE- PLACE(string, search_string [, re- place_string)</i>	<i>CLOB</i>	文字列内のある値を別の値に置き換えます。 'replace_string' に値を指定しない場合、見つかった 'search_string' 値は削除されます。	<i>RE- PLACE('GEORGE', 'GE', 'EG')</i>	<i>EGOREG</i>
<i>RPAD(string, length INTEGER [, fill])</i>	<i>CLOB</i>	文字 'fill' (デフォルトではスペース) を追加して、'string' をサイズ 'length' に入力します。 'string' がすでに 'length' より長い場合、切り捨てられます。	<i>RPAD('hi', 5, 'xy')</i>	<i>hixyx</i>
<i>TRANS- LATE(string, from, to)</i>	<i>CLOB</i>	'from' セットの文字と一致する 'string' の文字は、'to' セットの対応する文字に置き換えられます。	<i>TRANS- LATE('12345', '14', 'ax')</i>	<i>a23x5</i>
		NA		

NULL との連結から生じる文字列テキストの切り捨て

!!! Note * このセクションでは、オラクルデータベースと互換性のない機能について説明します。これにより、オラクルから Advanced Server にデータを変換する際に不整合が生じる可能性があります。

Advanced Server では、列の値が NULL の場合、列とテキスト文字列列の結合により、次のいずれかが発生する可能性があります。

- テキスト文字列の戻り
- テキスト文字列の消失 (つまり、null 結果)

結果は、NULL 列のデータタイプと結合の方法に依存します。

文字列結合演算子 '||' を使用する場合、入力パラメーターの 1 つが NULL である場合、テキストへの暗黙的強制を持つ表データ型にリストされているテキストへの暗黙的強制を持つ型は文字列を切り捨てませんが、他の型では切り捨てます明示的な型キャスト（つまり、::text）が使用されない限り、文字列。また、ヌルが存在する場合の一貫した動作を確認するには、CONCAT ファンクションを使用できます。

次のクエリーは、テキストへの暗黙的強制があるデータ型をリストします。

```
SELECT castsource::regtype, casttarget::regtype, castfunc::regproc,
*   CASE castcontext
*       WHEN 'e' THEN 'explicit'
*       WHEN 'a' THEN 'implicit in assignment'
*       WHEN 'i' THEN 'implicit in expressions'
*   END as castcontext,
*   CASE castmethod
*       WHEN 'f' THEN 'function'
*       WHEN 'i' THEN 'input/output function'
*       WHEN 'b' THEN 'binary-coercible'
*   END as castmethod
FROM pg_cast
*   WHERE casttarget::regtype::text = 'text'
*       AND castcontext='i';
```

次の表に、クエリーの結果を示します。

castsource	casttarget	castfunc	castcontext	castmethod
<i>character</i>	<i>text</i>	<i>pg_catalog.text</i>	<i>implicit in expressions</i>	<i>function</i>
<i>character varying</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>binary-coercible</i>
<i>*char*</i>	<i>text</i>	<i>pg_catalog.text</i>	<i>implicit in expressions</i>	<i>function</i>
<i>name</i>	<i>text</i>	<i>pg_catalog.text</i>	<i>implicit in expressions</i>	<i>function</i>
<i>pg_node_tree</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>binary-coercible</i>
<i>pg_ndistinct</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>
<i>pg_dependencies</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>
<i>integer</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>
<i>smallint</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>
<i>oid</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>
<i>date</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>
<i>double precision</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>
<i>real</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>

次のページに続く

表 4 – 前のページからの続き

castsource	casttarget	castfunc	castcontext	castmethod
<i>time with time zone</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>
<i>time without time zone</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>
<i>timestamp with time zone</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>
<i>interval</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>
<i>bigint</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>
<i>numeric</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>
<i>timestamp without time zone</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>
<i>record</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>
<i>boolean</i>	<i>text</i>	<i>pg_catalog.text</i>	<i>implicit in expressions</i>	<i>function</i>
<i>bytea</i>	<i>text</i>	-	<i>implicit in expressions</i>	<i>input/output function</i>

列の出力の詳細については、次の場所にある PostgreSQL コア文書の `pg_cast` システムカタログを参照してください。

< [https://www.postgresql.org/docs/current/ 静的/catalog-pg-cast.html](https://www.postgresql.org/docs/current/静的/catalog-pg-cast.html) >

したがって、例、データタイプ `UUID` はこのリストに含まれていないため、テキストに対する暗黙の強制はありません。その結果、`NULL UUID` 列を使用した特定の結合試行では、テキストが切り捨てられます。

この例では、すべての `NULL` 列の値を持つ単一の行で次の表が作成されます。

```
CREATE TABLE null_concat_types (
  *   boolean_type    BOOLEAN,
  *   uuid_type       UUID,
  *   char_type       CHARACTER
);

INSERT INTO null_concat_types VALUES (NULL, NULL, NULL);
```

列 `boolean_type` および `char_type` には、テキストに対する暗黙の強制がありますが、列 `uuid_type` にはありません。

したがって、文字列 `boolean_type` または `char_type` に対して結合演算子 `'||'` を使用して文字列を結合すると、次のようになります。

```

SELECT 'x=' || boolean_type || 'y' FROM null_concat_types;
      * ?column?

-----

x=y
(1 row)

SELECT 'x=' || char_type || 'y' FROM null_concat_types;
      * ?column?

-----

x=y
(1 row)

```

ただし、列 `uuid_type` と結合すると、`x`=文字列が失われます。

```

SELECT 'x=' || uuid_type || 'y' FROM null_concat_types;
      * ?column?

-----

y
(1 row)

```

ただし、`::text` で明示的なキャストを使用すると、`x`=文字列が失われなくなります。

```

SELECT 'x=' || uuid_type::text || 'y' FROM null_concat_types;
      * ?column?

-----

x=y
(1 row)

```

`CONCAT` ファンクションを使用すると、`x`=文字列も保持されます。

```

SELECT CONCAT('x=',uuid_type) || 'y' FROM null_concat_types;
      * ?column?

-----

x=y
(1 row)

```

したがって、`NULL` 列のデータタイプに応じて、明示的なキャストまたは `CONCAT` ファンクションを使用して、テキスト文字列が失われないようにする必要があります。

1.3.5 Pattern Matching String Functions

</ div>

Advanced Server は、REGEXP_COUNT、REGEXP_INSTR、および REGEXP_SUBSTR 関数のサポートを提供します。これらの関数は、正規表現で指定されたパターンの文字列を検索し、文字文字列内のパターンの出現に関する情報を結果ます。パターンは POSIX スタイルの正規表現である必要があります。POSIX スタイルの正規表現の作成の詳細については、次のコア文書を参照してください。

< <https://www.postgresql.org/docs/current/ 静的/functions-matching.html> >

REGEXP_COUNT

REGEXP_COUNT は、文字列で正規表現を検索し、正規表現が発生した回数を返します。シグネチャは次のとおりです。

```
INTEGER REGEXP_COUNT
(
    * srcstr      TEXT,
    * pattern     TEXT,
    * position    DEFAULT 1
    * modifier    DEFAULT NULL
)
```

パラメータ

srcstr

srcstr は、検索する文字列を指定します。

pattern

pattern は、REGEXP_COUNT が検索する正規表現を指定します。

position

position は、REGEXP_COUNT が検索を開始するソース文字列内の位置を示す整数値です。デフォルト値は 1 です。

modifier

modifier は、パターンマッチングの動作を制御する値を指定します。デフォルト値は NULL です。Advanced Server でサポートされている修飾子の完全なリストについては、次から入手可能な PostgreSQL コア文書を参照してください。

< <https://www.postgresql.org/docs/current/ 静的/functions-matching.html> >

例

次のシンプル例では、REGEXP_COUNT は、文字の並び 'reinitializing' で文字 i が使用された回数のカウントを返します。

```
edb=# SELECT REGEXP_COUNT('reinitializing', 'i', 1) FROM DUAL;
 regexp_count
-----
*           5
(1 row)
```

最初の例では、コマンドは REGEXP_COUNT が最初の位置でカウントを開始するよう指示します。コマンドを変更して、6 番目の位置からカウントをスタートする場合：

```
edb=# SELECT REGEXP_COUNT('reinitializing', 'i', 6) FROM DUAL;
 regexp_count
-----
*           3
(1 row)
```

REGEXP_COUNT は 3 を返します。カウントは、6 番目の位置の前にある文字 i の出現を除外するようになりました。

REGEXP_INSTR

REGEXP_INSTR は、POSIX スタイルの正規表現の文字列を検索します。このファンクションは、マッチした文字列内の位置を返します。シグネチャは次のとおりです。

```
INTEGER REGEXP_INSTR
(
  * srcstr          TEXT,
  * pattern          TEXT,
  * position         INT  DEFAULT 1,
  * occurrence       INT  DEFAULT 1,
  * returnparam      INT  DEFAULT 0,
  * modifier         TEXT DEFAULT NULL,
  * subexpression    INT  DEFAULT 0,
)
```

パラメータ：

srcstr

srcstr は、検索する文字列を指定します。

pattern

pattern は、REGEXP_INSTR が検索する正規表現を指定します。

position

position は、ソース文字列のスタート位置を示す整数値を指定します。デフォルト値は 1 です。

occurrence

`occurrence` は、検索される文字列でパターンが複数回出現した場合に返されるマッチを指定します。デフォルト値は 1 です。

`returnparam`

`returnparam` は、`REGEXP_INSTR` が結果文字列内の位置を指定する整数値です。デフォルト値は 0 です。指定してください：

- 0 は、`pattern` に一致する最初の文字の文字列内の位置を結果ます。
- `pattern` の終わりに続く最初の文字の位置を結果 0 より大きい値。

`modifier`

`modifier` は、パターンマッチングの動作を制御する値を指定します。デフォルト値は NULL です。Advanced Server でサポートされている修飾子の完全なリストについては、次から入手可能な PostgreSQL コア文書を参照してください。

< <https://www.postgresql.org/docs/current/ 静的/functions-matching.html> >

`subexpression`

`subexpression` は、`REGEXP_INSTR` によって返される `pattern` の部分を識別する整数値です。`subexpression` のデフォルト値は 0 です。

`subexpression` の値を指定する場合、検索対象の値の一部を分離する 1 つ（またはそれ以上）のカッコのセットを `pattern` に含める必要があります。`subexpression` で指定された値は、どのカッコのセットが返されるべきかを示します。例、`subexpression` が 2 の場合、`REGEXP_INSTR` は 2 番目のカッコのセットの位置を結果ます。

例

次のシンプル例では、`REGEXP_INSTR` は、3 つの連続した数字を含むパターンの最初の出現について、電話番号を含む文字列を検索します。

```
edb=# SELECT REGEXP_INSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 1) FROM DUAL;
 regexp_instr
-----
*              1
(1 row)
```

このコマンドは、`REGEXP_INSTR` に最初に出現した位置を結果ように指示します。コマンドを変更して、3 桁の連続した数字の 2 番目の出現のスタートを結果場合：

```
edb=# SELECT REGEXP_INSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 2) FROM DUAL;
 regexp_instr
-----
*              5
(1 row)
```

`REGEXP_INSTR` は 5 を返します。3 番目の連続した数字の 2 番目の出現は、5 th の位置から始まります。

REGEXP_SUBSTR

REGEXP_SUBSTR ファンクションは、POSIX 準拠の正規表現で指定されたパターンの文字列を検索します。REGEXP_SUBSTR は、ファンクションの呼び出しで指定されたパターンに一致する文字列を返します。ファンクションのシグネチャは次のとおりです。

```
TEXT REGEXP_SUBSTR
(
  * srcstr          TEXT,
  * pattern          TEXT,
  * position         INT  DEFAULT 1,
  * occurrence       INT  DEFAULT 1,
  * modifier         TEXT DEFAULT NULL,
  * subexpression    INT  DEFAULT 0
)
```

パラメータ：

srcstr

srcstr は、検索する文字列を指定します。

pattern

pattern は、REGEXP_SUBSTR が検索する正規表現を指定します。

position

position は、ソース文字列のスタート位置を示す整数値を指定します。デフォルト値は 1 です。

occurrence

occurrence は、検索される文字列でパターンが複数回出現した場合に返されるマッチを指定します。デフォルト値は 1 です。

modifier

modifier は、パターンマッチングの動作を制御する値を指定します。デフォルト値は NULL です。Advanced Server でサポートされている修飾子の完全なリストについては、次から入手可能な PostgreSQL コア文書を参照してください。

< <https://www.postgresql.org/docs/current/ 静的/functions-matching.html> >

subexpression

subexpression は、REGEXP_SUBSTR によって返される pattern の部分を識別する整数値です。subexpression のデフォルト値は 0 です。

subexpression の値を指定する場合、検索対象の値の一部を分離する 1 つ（またはそれ以上）のカッコのセットを pattern に含める必要があります。subexpression で指定された値は、どのカッコのセットが返されるべきかを示します。例、subexpression が 2 の場合、REGEXP_SUBSTR は 2 番目のカッコセットに含まれる値を結果ます。

例

次のシンプル例では、REGEXP_SUBSTR は電話番号を含む文字列を検索して、3 つの連続した数字の最初のセットを探します。

```
edb=# SELECT REGEXP_SUBSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 1) FROM DUAL;
 regexp_substr
-----
      800
(1 row)
```

3 桁の最初の出現箇所を見つけて、文字列 (800) を返します。コマンドを変更して、連続する 3 桁の数字の 2 番目の出現を確認する場合：

```
edb=# SELECT REGEXP_SUBSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 2) FROM DUAL;
 regexp_substr
-----
      555
(1 row)
```

REGEXP_SUBSTR は、2 番目のサブストリングの内容である 555 を返します。

1.3.6 Pattern Matching Using the LIKE Operator

</ div>

Advanced Server は、従来の SQL LIKE 演算子を使用したパターンマッチングを提供します。LIKE 演算子の構文は次のとおりです。

```
string LIKE pattern [ ESCAPE escape-character ]
string NOT LIKE pattern [ ESCAPE escape-character ]
```

すべての **pattern** は文字列のセットを定義します。**string** が **pattern** で表される文字列セットに含まれている場合、LIKE 式は TRUE を返します。予想どおり、LIKE が TRUE を返す場合、NOT LIKE 式は FALSE を返します。同等の式は NOT (string LIKE pattern) です。

pattern にパーセント記号またはアンダースコアが含まれていない場合、パターンは文字列自分自身のみを表します。その場合、LIKE は等号演算子のように機能します。**pattern** のアンダースコア () は、任意の 1 文字を表します (一致します)。パーセント記号 () は、ゼロ文字以上の任意の文字列に一致します。

いくつかの例：

```
'abc' LIKE 'abc'      true
'abc' LIKE 'a%'       true
'abc' LIKE '_b_'      true
'abc' LIKE 'c'        false
```

LIKE パターン一致は、常に文字列全体をカバーします。したがって、文字列内の任意の場所でパターンにマッチするには、パターンのスタートと末尾がパーセント記号である必要があります。

他の文字とマッチせずにリテラルのアンダースコアまたはパーセント記号とマッチングには、**pattern** の各文字の前にエスケープ文字が必要です。デフォルトのエスケープ文字はバックスラッシュですが、**ESCAPE** 句を使用して別のエスケープ文字を選択できます。エスケープ文字自分自身にマッチさせるには、2つのエスケープ文字を記述します。

バックスラッシュは文字列リテラルですでに特別な意味を持つため、バックスラッシュを含むパターン定数を記述するには、SQL ステートメントに2つのバックスラッシュを記述する必要があります。したがって、実際にリテラルのバックスラッシュに一致するパターンを記述することは、ステートメントに4つのバックスラッシュを記述することを意味します。これを回避するには、**ESCAPE** で別のエスケープ文字を選択します。その場合、バックスラッシュは **LIKE** にとって特別ではなくなりました。(ただし、文字列リテラルパーサにとっては特別なものなので、まだ2つ必要です。)

ESCAPE ' ' と記述することにより、エスケープ文字を選択しないこともできます。これにより、エスケープメカニズムが無効になり、パターン内のアンダースコアとパーセント記号の特別な意味をオフにすることができなくなります。

1.3.7 Data Type Formatting Functions

</ div>

次の表で説明する Advanced Server のフォーマット関数は、さまざまなデータ型（日付/時刻、整数、浮動小数点、数値）をフォーマットされた文字列に変換し、フォーマットされた文字列から特定のデータ型に変換するための強力なツールセットを提供します。これらの関数はすべて共通の呼び出し規則に従います。最初の引数はフォーマットされる値であり、2番目の引数は出力または入力フォーマットを定義する文字列テンプレートです。

Function	Return Type	Description	Example	Result
<i>TO_CHAR</i> (<i>DATE</i> [, <i>format</i>)	<i>VARCHAR2</i>	日付/時刻を出力 `format` の文字列に変換します。省略した場合、デフォルトのフォーマットは DD-MON-YY です。	<i>TO_CHAR</i> (SYSDATE, 'MM/DD/YYYY HH12:MI:SS AM')	07/25/2007 09:43:02 AM

次のページに続く

表 5 – 前のページからの続き

Function	Return Type	Description	Example	Result
<i>TO_CHAR</i> (<i>TIMESTAMP</i> [, <i>format</i>)	VARCHAR2	タイムスタンプを出力 `format`の文字 列に変換しま す。省略した 場合、デフォ ルトのフォー マットは DD- MON-YY で す。	<i>TO_CHAR</i> (<i>CUR- RENT_TIMESTAMP</i> , 'MM/DD/YYYY HH12:MI:SS AM')	08/13/2015 08:55:22 PM
<i>TO_CHAR</i> (<i>INTEGER</i> [, <i>format</i>)	VARCHAR2	整数を出力 `format`で文字 列に変換しま す	<i>TO_CHAR</i> (2412, '999,999S')	2,412+
<i>TO_CHAR</i> (<i>NUMBER</i> [, <i>format</i>)	VARCHAR2	10 進数数を出力 `format`で文 字列に変換し ます	<i>TO_CHAR</i> (10125.35, '999,999.99')	10,125.35
<i>TO_CHAR</i> (<i>DOUBLE PRECISION</i> , <i>format</i>)	VARCHAR2	浮動小数点数 を出力 `for- mat`で文字列 に変換します	<i>TO_CHAR</i> (<i>CAST</i> (123.5282 <i>AS REAL</i>), '999.99')	123.53

次のページに続く

表 5 – 前のページからの続き

Function	Return Type	Description	Example	Result
<i>TO_DATE(string [, for- mat])</i>	<i>TIMESTAMP</i>	日付またはタイムスタンプ形式の文字列を `TIMESTAMP` データタイプに変換します	<i>TO_DATE('2007-07-04 13:39:10', 'YYYY-MM-DD HH24:MI:SS')</i> <i>

TO_DATE('2007-07-04','YYYY-MM-DD')</i>	<i>04-JUL-07 13:39:10

04-JUL-07 00:00:00`</i>
<i>TO_NUMBER(string [, for- mat])</i>	<i>NUMBER</i>	数値形式の文字列を `NUMBER` データタイプに変換します	<i>TO_NUMBER('2,412', '999,999S')</i>	<i>-2412</i>
<i>TO_TIMESTAMP(string, for- mat)</i>	<i>TIMESTAMP</i>	タイムスタンプ形式の文字列を `TIMESTAMP` データタイプに変換します	<i>TO_TIMESTAMP('05 Dec 2000 08:30:25 pm', 'DD Mon YYYY hh12:mi:ss pm')</i>	<i>05-DEC-00 20:30:25 +05:30</i>
<i>TO_TIMESTAMP_TZ(string, for- mat)</i>	<i>TIMESTAMP</i>	タイムスタンプ形式の文字列を `TIMESTAMP` データタイプに変換します	<i>TO_TIMESTAMP_TZ('2003/12/13 10:13:18 -8:00', 'YYYY/MM/DD HH:MI:SS TZH:TZM')</i>	<i>13-DEC-03 23:43:18 +05:30</i>
		NA		

TO_CHAR、TO_DATE、TO_TIMESTAMP、および TO_TIMESTAMP_TZ

出力テンプレート文字列（TO_CHAR の場合）には、認識される特定のパターンがあり、フォーマットされる値から適切にフォーマットされたデータに置き換えられます。テンプレートパターンではないテキストは、そのままコピーされます。同様に、入力テンプレート文字列（TO_CHAR 以外の場合）で、テンプレートパターンは入力データ文字列の一部とそこにある値を特定します。

TO_TIMESTAMP、TO_TIMESTAMP_TZ、または TO_DATE の呼び出し時に日付、月、または年を指定しない場合、デフォルトでは、出力フォーマットはそれぞれ現在の月または現在の年の最初の日付を考慮します。

次の例では、日付、月、年が入力文字列に指定されていません。TO_TIMESTAMP、TO_TIMESTAMP_TZ、および TO_DATE は、現在の月と現在の年の最初の日付のデフォルト値を返します。

```
edb=# select to_timestamp('12', 'HH');
      *
      to_timestamp
-----
01-MAY-20 12:00:00 +05:30
(1 row)

edb=# select to_timestamp_tz('12', 'HH');
      *
      to_timestamp_tz
-----
01-MAY-20 12:00:00 +05:30
(1 row)

edb=# select to_date('12', 'HH');
      *
      to_date
-----
01-MAY-20 12:00:00
(1 row)
```

次の表は、TO_CHAR、TO_DATE、TO_TIMESTAMP、および TO_TIMESTAMP_TZ 関数を使用して日付値をフォーマットするために使用可能なテンプレートパターンを示しています。

Pattern	Description
<i>HH</i>	1 日の時間 (01-12)
<i>HH12</i>	1 日の時間 (01-12)
<i>HH24</i>	1 日の時間 (00-23)
<i>MI</i>	分 (00-59)
<i>SS</i>	2 番目 (00-59)
<i>SSSSS</i>	真夜中過ぎの秒数 (0-86399)
<i>FFn</i>	<i>n</i> が、結果桁数の 1~9 のオプションの整数である分数秒。省略した場合、デフォルトは 6 です。
<i>AM or A.M. or PM or P.M.</i>	子午線インジケータ (大文字)
<i>am or a.m. or pm or p.m.</i>	子午線インジケータ (小文字)
<i>Y,YYY</i>	カンマ付きの年 (4 桁以上)
<i>YEAR</i>	年 (スペルアウト)
<i>SYEAR</i>	年 (スペルアウト) (マイナス符号が前に付いた BC 日付)
<i>YYYY</i>	年 (4 桁以上)
<i>YYYYY</i>	年 (4 桁以上) (マイナス符号が前に付いた BC 日付)
<i>YYY</i>	年の下 3 桁

次のページに続く

表 6 – 前のページからの続き

Pattern	Description
<i>YY</i>	年の下 2 桁
<i>Y</i>	年の最後の桁
<i>YYYY</i>	ISO 年 (4 桁以上)
<i>IYY</i>	ISO 年の下 3 桁
<i>IY</i>	ISO 年の下 2 桁
<i>I</i>	ISO 年の下 1 桁
<i>BC or B.C. or AD or A.D.</i>	時代インジケーター (大文字)
<i>bc or b.c. or ad or a.d.</i>	時代インジケーター (小文字)
<i>MONTH</i>	完全な大文字の月名前
<i>Month</i>	大文字と小文字が完全に混在する月の名前
<i>month</i>	完全な小文字の月名前
<i>MON</i>	省略された大文字の月名前 (英語では 3 文字、ローカライズされた長さは異なります)
<i>Mon</i>	大文字と小文字が混在する月の省略名前 (英語では 3 文字、ローカライズされた長さは異なります)
<i>mon</i>	省略された小文字の月名前 (英語では 3 文字、ローカライズされた長さは異なります)
<i>MM</i>	月番号 (01-12)
<i>DAY</i>	完全な大文字の日の名前
<i>Day</i>	完全な大文字と小文字が混在する日の名前
<i>day</i>	完全な小文字の日の名前
<i>DY</i>	省略された大文字の日の名前 (英語では 3 文字、ローカライズされた長さは異なります)
<i>Dy</i>	大文字と小文字が混在する日の名前の省略形 (英語では 3 文字、ローカライズされた長さは異なります)
<i>dy</i>	省略された小文字の日の名前 (英語では 3 文字、ローカライズされた長さは異なります)
<i>DDD</i>	年の日 (001-366)
<i>DD</i>	日 (01-31)
<i>D</i>	曜日 (1-7; 日曜日は 1)
<i>W</i>	月の週 (1-5) (最初の週は月の最初の日から始まります)
<i>WW</i>	年の週番号 (1-53) (最初の週は年の最初の日から始まります)
<i>IW</i>	ISO 週数年; 新年の最初の木曜日は第 1 週です
<i>CC</i>	世紀 (2 桁) ; 21 世紀は 2001-01-01 に始まります
<i>SCC</i>	BC の日付の前にマイナス符号が付いていることを除いて CC と同じ
<i>J</i>	ジュリアンの日 (紀元前 4712 年 1 月 1 日からの日数)
<i>Q</i>	四半期
<i>RM</i>	ローマ数字の月 (I-XII; I = 1 月) (大文字)

次のページに続く

表 6 – 前のページからの続き

Pattern	Description
<i>rm</i>	ローマ数字の月 (i-xii; i = 1 月) (小文字)
<i>RR</i>	年の最後の 2 桁のみが指定された場合、年の最初の 2 桁。結果は、現在の年と指定された 2 桁の年を使用するアルゴリズムに基づいています。指定された 2 桁の年の最初の 2 桁は、次の例外を除いて、現在の年の最初の 2 桁と同じになります。指定された 2 桁の年が < 50 で、現在の年の最後の 2 桁が >= 50 の場合、指定された年の最初の 2 桁は、現在の年の最初の 2 桁より大きいです。指定された 2 桁の年が >= 50 で、現在の年の最後の 2 桁が < 50 の場合、指定された年の最初の 2 桁は、現在の年の最初の 2 桁より小さいです。
<i>RRRR</i>	TO_DATE フังก์ションにのみ影響します。2 桁または 4 桁の年を指定できます。2 桁の年が指定されている場合、RR フォーマットのように年の最初の 2 桁を返します。4 桁の年が指定されている場合、指定された 4 桁の年を返します。
<i>TZH</i>	タイムゾーン時間
<i>TZM</i>	タイムゾーン分
	NA

日付と時刻の修飾子

特定の修飾子をテンプレートパターンに適用して、その動作を変更できます。例、FMMonth は、FM 修飾子を持つ Month パターンです。次の表に、日付/時刻形式の修飾子パターンを示します。

Modifier	Description	Example
<i>FM</i> prefix	塗りつぶしモード (パディングブランクとゼロを抑制)	<i>FMMonth</i>
<i>TH</i> suffix	大文字の序数のサフィックス	<i>DDTH</i>
<i>th</i> suffix	小文字の序数のサフィックス	<i>DDth</i>
<i>FX</i> prefix	固定フォーマットのグローバルオプション (使用上の注意を参照)	<i>FX Month DD Day</i>
<i>SP</i> suffix	スペルモード	<i>DDSP</i>
	NA	

日付/時刻形式の使用上の注意：

- FM は、パターンの出力を固定幅に make に追加される先行ゼロと後続ブランクを抑制します。
- FX オプションが使用されていない場合、TO_TIMESTAMP、TO_TIMESTAMP_TZ、および TO_DATE は入力文字列のマルチプルの空白スペースをスキップします。FX は、テンプレートの最初のアイテムとして指定する必要があります。例：

```
* TO_TIMESTAMP('2000 - JUN', 'YYYY-MON') is correct, but
* TO_TIMESTAMP('2000    JUN', 'FXYYYY MON') and
* TO_TIMESTAMP_TZ('2000    JUN', 'FXYYYY MON') returns an error
* because TO_TIMESTAMP and TO_TIMESTAMP_TZ expects one space
* only.
```

- 通常のテキストは TO_CHAR テンプレートで許可され、そのまま出力されます。
- 文字列から timestamp、timestampz、または date への変換では、YYY、YYYY、または Y,YYY フィールドがある場合、CC フィールドは無視されます。CC または YY または Y を使用する場合、年は $(CC-1)*100+YY$ として計算されます。

次の表に、TO_CHAR および TO_DATE 関数の使用例をいくつか示します。

Expression	Result
<i>TO_CHAR(CURRENT_TIMESTAMP, 'Day, DD HH12:MI:SS')</i>	<i>'Tuesday , 06 05:39:18'</i>
<i>TO_CHAR(CURRENT_TIMESTAMP, 'FM-Day, FMDD HH12:MI:SS')</i>	<i>'Tuesday, 6 05:39:18'</i>
<i>TO_CHAR(-0.1, '99.99')</i>	<i>' -.10'</i>
<i>TO_CHAR(-0.1, 'FM9.99')</i>	<i>'-.1'</i>
<i>TO_CHAR(0.1, '0.9')</i>	<i>' 0.1'</i>
<i>TO_CHAR(12, '9990999.9')</i>	<i>' 0012.0'</i>
<i>TO_CHAR(12, 'FM9990999.9')</i>	<i>'0012.'</i>

次のページに続く

表 7 – 前のページからの続き

Expression	Result
<i>TO_CHAR</i> (485, '999')	' 485'
<i>TO_CHAR</i> (-485, '999')	'-485'
<i>TO_CHAR</i> (1485, '9,999')	' 1,485'
<i>TO_CHAR</i> (1485, '9G999')	' 1,485'
<i>TO_CHAR</i> (148.5, '999.999')	' 148.500'
<i>TO_CHAR</i> (148.5, 'FM999.999')	'148.5'
<i>TO_CHAR</i> (148.5, 'FM999.990')	'148.500'
<i>TO_CHAR</i> (148.5, '999D999')	' 148.500'
<i>TO_CHAR</i> (3148.5, '9G999D999')	' 3,148.500'
<i>TO_CHAR</i> (-485, '999S')	'485-'
<i>TO_CHAR</i> (-485, '999MI')	'485-'
<i>TO_CHAR</i> (485, '999MI')	'485 '
<i>TO_CHAR</i> (485, 'FM999MI')	'485'

次のページに続く

表 7 – 前のページからの続き

Expression	Result
<i>TO_CHAR</i> (-485, '999PR')	'<485>'
<i>TO_CHAR</i> (485, 'L999')	'\$ 485'
<i>TO_CHAR</i> (485, 'RN')	' <i>CDLXXXV</i> '
<i>TO_CHAR</i> (485, 'FMRN')	' <i>CDLXXXV</i> '
<i>TO_CHAR</i> (5.2, 'FMRN')	' <i>V</i> '
<i>TO_CHAR</i> (12, '99V999')	' <i>12000</i> '
<i>TO_CHAR</i> (12.4, '99V999')	' <i>12400</i> '
<i>TO_CHAR</i> (12.45, '99V9')	' <i>125</i> '

次の表に、TO_TIMESTAMP_TZ ファンクションの使用例をいくつか示します。

Expression	Result
<i>TO_TIMESTAMP_TZ('12-JAN-2010', 'DD-MONTH-YYYY')</i>	'12-JAN-10 00:00:00 +05:30'
<i>TO_TIMESTAMP_TZ('03-APR-07 09:12:21 P.M', 'DD-MON-YY HH12:MI:SS A.M')</i>	'03-APR-07 09:12:21 +05:30'
<i>TO_TIMESTAMP_TZ('2003/12/13 10:13:18 -8:00', 'YYYY/MM/DD HH:MI:SS TZH:TZM')</i>	'13-DEC-03 23:43:18 +05:30'
<i>TO_TIMESTAMP_TZ('20-MAR-20 04:30:00 +08:00', 'DD-MON-YY HH:MI:SS TZH:TZM')</i>	'20-MAR-20 02:00:00 +05:30'
<i>TO_TIMESTAMP_TZ('10-Sep-02 14:10:10.123000', 'DD-MON-RR HH24:MI:SS.FF')</i>	'10-SEP-02 14:10:10.123 +05:30'

</ div>

IMMUTABLE TO_CHAR (TIMESTAMP、フォーマット) ファンクション

関数の IMMUTABLE フォームを使用する可能性のある TO_CHAR ファンクションの特定のケースがありファンクション。基本的に、ファンクションがデータベースを変更しない場合、ファンクションは IMMUTABLE であり、ファンクションは入力パラメーターのみに依存して同じ一貫した値を返します。つまり、構成パラメーター、ロケール、データベースのコンテンツなどの設定は、ファンクションによって返される結果に影響しません。

ファンクションの変動性カテゴリ VOLATILE、STABLE、および IMMUTABLE の詳細については、次の PostgreSQL Core 文書を参照してください。

< <https://www.postgresql.org/docs/current/ 静的/xfunc-volatility.html> >

IMMUTABLE ファンクションの特定の利点は、CREATE INDEX コマンドで使用して、そのファンクションに基づいてインデックスを作成できることです。

オーダーファンクションが IMMUTABLE フォームを使用するには、次の条件が満たされている必要があります。

- TO_CHAR ファンクションの最初のパラメータは、TIMESTAMP のデータタイプである必要があります。
- TO_CHAR ファンクションの 2 番目のパラメータで指定されたフォーマットは、言語、ロケールなどの要因に基づいてファンクションの結果値に影響を与えてはなりません。例、'YYYY-MM-DD HH24:MI:SS' フォーマットは、IMMUTABLE フォームのファンクションに使用できます。ロケール設定に関係なく、ファンクションの結果は数値フォームのみで表された日付と時刻です。ただし、'DD-MON-YYYY' のフォーマットは、IMMUTABLE フォームのファンクションには使用できません。これは、3 文字の月の省略形がロケール設定に応じて異なる結果を結果可能性があるためです。

不変ではないファンクションを生成するフォーマットパターンには、スペルアウトまたは短縮された月 (MONTH, MON)、日 (DAY, DY)、中央値 (メジアン) インジケータ (AM, PM)、または時代インジケータ (BC, AD) のバリエーションが含まれます。

次の例では、TIMESTAMP 列を持つテーブルが作成されます。

```
CREATE TABLE ts_tbl (ts_col TIMESTAMP);
```

以下は、TO_CHAR ファンクションの IMMUTABLE フォームを使用したインデックスの正常な作成を示しています。

```
edb=# CREATE INDEX ts_idx ON ts_tbl (TO_CHAR(ts_col,'YYYY-MM-DD HH24:MI:SS'));
CREATE INDEX
edb=# \dS ts_idx
      *
      Index "public.ts_idx"
Column |      Type      | Definition
-----+-----+-----
to_char | character varying | to_char(ts_col, 'YYYY-MM-DD HH24:MI:SS'::character_varying)
btree, for table "public.ts_tbl"
```

3 文字の月の省略形 MON はロケール設定に基づいて異なる結果値になる可能性があるため、TO_CHAR ファンクションで指定されたフォーマットが IMMUTABLE フォームの使用を妨げるため、次の結果はエラーになります。

```
edb=# CREATE INDEX ts_idx_2 ON ts_tbl (TO_CHAR(ts_col, 'DD-MON-YYYY'));
ERROR: functions in index expression must be marked IMMUTABLE
```

TO_NUMBER

次の表に、数値の書式設定に使用できるテンプレートパターンを示します。

Pattern	Description
<i>9</i>	指定した桁数の値
<i>0</i>	先頭がゼロの値
<i>.</i> (<i>period</i>)	デシマルポイント
<i>,</i> (<i>comma</i>)	グループ（千）セパレータ
<i>\$</i>	ドル記号
<i>S</i>	番号に固定された記号（ロケールを使用）。
<i>L</i>	通貨記号（ロケールを使用）
	NA

9 は、9s と同じ桁数の値になります。数字が使用できない場合、サーバーは対応する 9s を無視します。S パターンは+をサポートせず、\$パターンは式の 10 進数をサポートしません。

次の表に、TO_NUMBER ファンクションの使用例をいくつか示します。

Expression	Result
<i>TO_NUMBER(' -65', 'S99')</i>	' -65'
<i>TO_NUMBER('\$65', 'L99')</i>	' 65'
<i>TO_NUMBER('9678584', '9999999')</i>	' 9678584'
<i>TO_NUMBER('123,456,789', '999,999,999')</i>	' 123456789'
<i>TO_NUMBER('1210.73', '9999.99')</i>	' 1210.73'
<i>TO_NUMBER('1210.73')</i>	' 1210.73'
<i>TO_NUMBER('0101.010', 'FM999999999.99999')</i>	' 101.010'

数値修飾子

次の表は、数値フォーマットの修飾子パターンを示しています。

Pattern	Description	Example
<i>FM</i> prefix	塗りつぶしモード（末尾のゼロとパディングブランクを抑制）	<i>FM99.99</i>
	NA	

TO_CHAR、TO_NUMBER、および SUBSTR Functions の違い

INITDBOPTS 変数を使用してデータベースを初期化し、オラクルデータベースと互換性のあるモードでクラスターを作成できます。PostgreSQL モードで作成されたクラスターには互換性機能が含まれていません。PostgreSQL モードで新しいクラスターを作成するには、"--no-redwood-compat" オプションを使用します。これは、PostgreSQL と同様の動作を提供します。

次の表に示す関数のいずれかを含む SELECT ステートメントは、Advanced Server および PostgreSQL で示された結果を返します。TO_CHAR、TO_NUMBER、および SUBSTR 関数の次の違いをリストします。

**** TO_CHAR ****: 指定されたフォーマットに従ってタイムスタンプを文字列に変換します。

!!! Note * - 'syear', 'syyy', 'year', 'rrrr', 'sp', 'spt', 'scc', 'rr', 'ff', 'ff7', 'ff8', tran_10, パターン, 'ff8', PostgreSQL-'TH' 修飾子は、Advanced Server のパディングゼロの追加を減らします。-大文字のサフィックスを表示する大文字の 'TH' パターンは、Advanced Server ではサポートされていません。'TH/th' パターンが指定されている場合、サフィックスは常に小文字で表示されます。-修飾子は 'FM' パターンで表示でき、出現するたびに修飾子の効果を切り替えます。偶数回指定されたときに 'FM' パターンの効果を切り替えるには、PostgreSQL で奇数回指定されたときに効果をオフにし、'FM' パターンの効果をオンにします。ただし、Advanced Server では、偶数回または奇数回に関係なく 'FM' パターンが指定されている場合、'FM' パターンの効果が常にオンになります。

次の表は、TO_CHAR(timestamp, text) ファンクションの例を示しています。

Examples	Result	
	Advanced Server	PostgreSQL
<pre>SELECT TO_CHAR(current_timestamp, 'of');

 SELECT TO_CHAR(current_timestamp, 'OF');</pre>	+05:30	of +05:30
<pre>SELECT TO_CHAR(current_timestamp, 'syear');</pre>	twenty twenty one	s1ear

次のページに続く

表 9 – 前のページからの続き

Examples	Result	
<i>SELECT TO_CHAR(current_timestamp, 'yyyy');</i>	<i>2021</i>	<i>s2021</i>
<i>SELECT TO_CHAR(current_timestamp, 'year');</i>	<i>twenty twenty one</i>	<i>1ear</i>
<i>SELECT TO_CHAR(current_timestamp, 'rrrr');</i>	<i>2021</i>	<i>rrrr</i>
<i>SELECT TO_CHAR(current_timestamp, 'hhsp');</i>	<i>five</i>	<i>05sp</i>
<i>SELECT TO_CHAR(current_timestamp, 'hh-spth');</i>	<i>fifth</i>	<i>05spth</i>
<i>SELECT TO_CHAR(current_timestamp, 'scc');</i>	<i>21</i>	<i>s21</i>
<i>SELECT TO_CHAR(current_timestamp, 'rr');</i>	<i>21</i>	<i>rr</i>
<i>SELECT TO_CHAR(current_timestamp, 'ff');</i>	<i>923202</i>	<i>ff</i>
<i>SELECT TO_CHAR(current_timestamp, 'ff7');</i>	<i>4192310</i>	<i>ff7</i>
<i>SELECT TO_CHAR(current_timestamp, 'ff8');</i>	<i>36181700</i>	<i>ff8</i>
<i>SELECT TO_CHAR(current_timestamp, 'ff9');</i>	<i>640405000</i>	<i>ff9</i>
<i>SELECT TO_CHAR(current_timestamp, 'tzh');</i> <i>

 SELECT TO_CHAR(current_timestamp, 'TZh');</i>	<i>+05</i>	<i>isth

 +05</i>

次のページに続く

表 9 – 前のページからの続き

Examples	Result	
<pre>SELECT TO_CHAR(current_timestamp, 'TZM');

 SELECT TO_CHAR(current_timestamp, 'tm');</pre>	30	30 istm
<pre>SELECT TO_CHAR(current_timestamp, 'iddd');</pre>	1134	131
<pre>SELECT TO_CHAR(current_timestamp, 'id');</pre>	16	5
<pre>SELECT TO_CHAR(current_timestamp, 'ssss');</pre>	5555	64490
<pre>SELECT TO_CHAR(current_timestamp, 'hhthh');</pre>	Invalid format	11thh
<pre>SELECT TO_CHAR(current_timestamp, 'rmth');</pre>	Invalid format	v
<pre>SELECT TO_CHAR(current_timestamp, 'FMOF');

 SELECT TO_CHAR(current_timestamp, 'FMF- MOF');

 SELECT TO_CHAR(current_timestamp, 'FMFMF- MOF');

 SELECT TO_CHAR(current_timestamp, 'FMFMFMF- MOF');</pre>	+5:30	+5:30 FM+05:30 FM+5:30 FMFM+05:30
<pre>SELECT TO_CHAR(timestampz'5-MAY-21 17:37:14 +05:30', 'DDTH');</pre>	05th	05TH

次のページに続く

表 9 – 前のページからの続き

Examples	Result	
<i>SELECT TO_CHAR(timestampz'5-JAN-0001 17:37:14 +05:30', 'FMY,YYY');</i>	<i>1</i>	<i>0,001</i>
<i>SELECT TO_CHAR(timestampz'5-JAN-0001 17:37:14 +05:30', 'Y,YYYTH');</i>	<i>01st</i>	<i>0,001ST</i>
<i>SELECT TO_CHAR(timestampz'5-JAN-0001 17:37:14 +05:30', 'YYYYTH');</i>	<i>01st</i>	<i>0001ST</i>
<i>SELECT TO_CHAR(timestampz'5-JAN-0001 17:37:14 +05:30', 'YYYYTH');</i>	<i>01st</i>	<i>0001ST</i>
<i>SELECT TO_CHAR(timestampz'5-JAN-0001 17:37:14 +05:30', 'YYTH');</i>	<i>01st</i>	<i>001ST</i>
<i>SELECT TO_CHAR(timestampz'5-JAN-0001 17:37:14 +05:30', 'YYTH');</i>	<i>01st</i>	<i>001ST</i>

**** TO_NUMBER ****：指定されたフォーマットに従って文字列を数値に変換します。

!!! Note * - PostgreSQL はフォーマットオプションを厳密にチェックしません。対照的に、Advanced Server は、入力文字列の先頭に '-' または ' ' があるいくつかの場合を除き、フォーマットをチェックします。これらの文字のパターンを、フォーマット文字列の対応する場所に指定する必要はありません。PostgreSQL では、入力文字列のすべての数字をパターンとマッチさせる必要はありません。有効なパターンが存在する入力文字列からこれらの数字を出力します。入力文字列に含まれるすべての数字のパターンを指定しないと、Advanced Server はエラーをスローします。PostgreSQL は、入力またはフォーマット文字列に存在する無効な文字を無視しますが、Advanced Server はエラーをスローします。この種の動作は、さまざまなタイプのパターンで多くの違いをもたらす可能性があります。-パターン 'PL'、'PR'、および 'SG' は Advanced Server ではサポートされていません。

次の表は、TO_NUMBER(text, text) ファンクションの例を示しています。

Examples	Result	
	Advanced Server	PostgreSQL

次のページに続く

表 10 – 前のページからの続き

Examples	Result	
<i>SELECT</i> <i>TO_NUMBER('\$1',</i> <i>'00');</i>	Invalid input syntax for type numeric: <i>\$1</i>	<i>1</i>
<i>SELECT</i> <i>TO_NUMBER('\$\$1',</i> <i>'\$\$0');</i>	Invalid format string: <i>\$\$0</i>	<i>1</i>
<i>SELECT</i> <i>TO_NUMBER('1\$',</i> <i>'0\$');</i>	Invalid input syntax for type numeric: <i>1.</i>	<i>1</i>
<i>SELECT</i> <i>TO_NUMBER('\$1\$',</i> <i>'\$0\$');</i>	Invalid format string: <i>\$0\$</i>	<i>1</i>
<i>SELECT</i> <i>TO_NUMBER('1\$\$1',</i> <i>'0\$L0');</i>	Invalid input syntax for type numeric: <i>1.\$1</i>	<i>11</i>
<i>SELECT</i> <i>TO_NUMBER('1,11',</i> <i>'000');</i>	<i>111</i>	<i>11</i>
<i>SELECT</i> <i>TO_NUMBER('1,11',</i> <i>'0,0');</i>	Invalid input syntax for type numeric: <i>111</i>	<i>11</i>
<i>SELECT</i> <i>TO_NUMBER('1-</i> <i>1', '0\$0');</i>	Invalid format string: <i>0\$0</i>	<i>-11</i>

次のページに続く

表 10 – 前のページからの続き

Examples	Result	
<i>SELECT</i> <i>TO_NUMBER('1-</i> <i>1', '000');</i>	Invalid input syntax for type numeric: <i>1-1</i>	<i>11</i>
<i>SELECT</i> <i>TO_NUMBER('1.11',</i> <i>'000');</i>	Invalid input syntax for type numeric: <i>1.11</i>	<i>11</i>
<i>SELECT</i> <i>TO_NUMBER('1..1',</i> <i>'0.0');</i>	Invalid input syntax for type numeric: <i>1..1</i>	<i>1</i>
<i>SELECT</i> <i>TO_NUMBER('\$11-',</i> <i>'\$00S');</i>	Invalid input syntax for type numeric: <i>-\$11</i>	<i>-11</i>
<i>SELECT</i> <i>TO_NUMBER('11',</i> <i>");</i>	Invalid input syntax for type numeric: <i>11</i>	Returns blank
<i>SELECT</i> <i>TO_NUMBER(' ',</i> <i>");</i>	Invalid input syntax for type numeric: <i>**</i>	Returns blank
<i>SELECT</i> <i>TO_NUMBER('\$12.345',</i> <i>'L00.000');</i>	Invalid format string: <i>L00.000</i>	<i>12.345</i>
<i>SELECT</i> <i>TO_NUMBER('\$12.345',</i> <i>'L00D000');</i>	Invalid format string: <i>L00.000</i>	<i>12.345</i>

次のページに続く

表 10 – 前のページからの続き

Examples	Result	
<i>SELECT</i> <i>TO_NUMBER('11-',</i> <i>'9MMI');</i>	Invalid format string: <i>9MMI</i>	<i>-1</i>
<i>SELECT</i> <i>TO_NUMBER('-</i> <i>11', 'MI99');</i>	Invalid input syntax for type numeric: <i>-11</i>	<i>-11</i>
<i>SELECT</i> <i>TO_NUMBER('000011',</i> <i>'FMFM000000');</i>	Invalid format string: <i>FMFM000000</i>	<i>11</i>
<i>SELECT</i> <i>TO_NUMBER('+11',</i> <i>'S99');</i> <i>SELECT</i> <i>TO_NUMBER('11+',</i> <i>'99S');</i>	Invalid input syntax for type numeric: <i>+11</i> <i>/> 11</i>	<i>11</i> <i>11</i>
<i>SELECT</i> <i>TO_NUMBER('+11',</i> <i>'PL99');</i>	Invalid format string: <i>PL99</i>	<i>11</i>
<i>SELECT</i> <i>TO_NUMBER('-</i> <i>11', '99PR');</i>	Invalid format string: <i>99PR</i>	<i>-11</i>
<i>SELECT</i> <i>TO_NUMBER('1+1-</i> <i>1', '9SG9SG9');</i>	Invalid format string: <i>9SG9SG9</i>	<i>-111</i>

次のページに続く

表 10 – 前のページからの続き

Examples	Result	
<pre>SELECT TO_NUMBER('01,000', '99G999');

 SELECT TO_NUMBER('01,000', 'FM99G999');</pre>	1000	1000
 100

**** SUBSTR ****：文字値の部分文字列を返します。

!!! Note * -文字列のインデックスは 1 から始まります。PostgreSQL では、1 つの文字のみをフェッチするスタートパラメータとしてゼロの場合、結果は空白を返します。2 文字をフェッチしようとする、スタート位置にゼロが指定されているため、結果には 1 文字が表示されます。ただし、Advanced Server では、スタートパラメータとしてゼロの場合、内部的に 1 に変換され、検索が開始されます。PostgreSQL では、スタートパラメータの負の値は許可されていませんが、Advanced Server では逆検索が発生します。-スタートおよび長さのパラメータに数値が渡されると、PostgreSQL はエラーをスローしますが、Advanced Server はそれを整数に変換します。

次 の 表 は 、SUBSTR(string, from, count)、SUBSTR(string, from)、SUBSTR(string, from, count)::bytea、および SUBSTR(string, from)::bytea ファンクションの例を示しています。

Examples	Result	
	Advanced Server	PostgreSQL
<i>SELECT (SUBSTR('ABCDEFGH', 0, 1));</i>	A	Returns blank
<i>SELECT (SUBSTR('ABCD', -1, 1));</i> <i>SELECT (SUBSTR('ABCD', -4, 1));</i>	D A	Returns blank
<i>SELECT (SUBSTR('ABCD', 1, -1));</i>	Returns blank	ERROR: negative substring length not allowed
<i>SELECT (SUBSTR('ABCD', 1.1, 1));</i>	A	ERROR: function substr(unknown, numeric, integer) does not exist
<i>SELECT (SUBSTR('ABCD', 1, 1.1));</i>	A	ERROR: function substr(unknown, integer, numeric) does not exist
<i>SELECT (SUBSTR('ABCD', -1));</i>	D	ABCD
<i>SELECT (SUBSTR('ABCD', 1.1));</i>	ABCD	ERROR: function substr(unknown, numeric, integer) does not exist
1.3. Functions and Operators		61

1.3.8 Date/Time Functions and Operators

</ div>

日付/時刻 Functions の表は、日付/時刻値の処理に使用可能な関数を示しています。詳細は、次のサブセクションに記載されています。次の表は、基本的な算術演算子 (+, -) の動作を示しています。書式設定関数については、を参照してください。日付/時刻データ型の背景情報に精通している必要があります。を参照してください。

Operator	Example	Result
plus (+)	<i>DATE</i> '2001-09-28' + 7	05-OCT-01 00:00:00
plus (+)	<i>TIMESTAMP</i> '2001-09-28 13:30:00' + 3	01-OCT-01 13:30:00
minus (-)	<i>DATE</i> '2001-10-01' - 7	24-SEP-01 00:00:00
minus (-)	<i>TIMESTAMP</i> '2001-09-28 13:30:00' - 3	25-SEP-01 13:30:00
minus (-)	<i>TIMESTAMP</i> '2001-09-29 03:00:00' - <i>TIMESTAMP</i> '2001-09-27 12:00:00'	@ 1 day 15 hours

次の表の日付/時刻関数では、DATE および TIMESTAMP データ型の使用は交換可能です。

Function	Return Type	Description	Example	Result
<i>ADD_MONTHS</i> (<i>DATE</i> , <i>NUMBER</i>)	<i>DATE</i>	日付に月を追加	<i>ADD_MONTHS</i> ('28-FEB-97', 3.8)	31-MAY-97 00:00:00
<i>CURRENT_DATE</i>	<i>DATE</i>	現在の日付	<i>CURRENT_DATE</i>	04-JUL-07
<i>CURRENT_TIMESTAMP</i>	<i>TIMESTAMP</i>	現在の日付と時刻を返します	<i>CURRENT_TIMESTAMP</i>	04-JUL-07 15:33:23.484
<i>EXTRACT(field FROM TIMESTAMP)</i>	<i>DOUBLE PRECISION</i>	サブフィールドを取得	<i>EXTRACT(hour FROM TIMESTAMP '2001-02-16 20:38:40')</i>	20

次のページに続く

表 12 – 前のページからの続き

Function	Return Type	Description	Example	Result
<i>LAST_DAY</i> (<i>DATE</i>)	<i>DATE</i>	指定された日付で表される月の最終日を返します。指定された日付に時間部分が含まれる場合、変更されずに結果に繰り越されます	<i>LAST_DAY</i> ('14-APR-98')	30-APR-98 00:00:00
<i>LOCAL-TIMESTAMP</i> [<i>(precision)</i>]	<i>TIMESTAMP</i>	現在の日時 (現在のトランザクションのスタート)	<i>LOCALTIMESTAMP</i>	04-JUL-07 15:33:23.484
<i>MONTHS_BETWEEN</i> (<i>DATE</i> , <i>DATE</i>)	<i>NUMBER</i>	2つの日付の間の月数	<i>MONTHS_BETWEEN</i> ('28-FEB-07', '30-NOV-06')	3
<i>NEXT_DAY</i> (<i>DATE</i> , <i>day-ofweek</i>)	<i>DATE</i>	指定された日付の次の `day-ofweek` に該当する日付	<i>NEXT_DAY</i> ('16-APR-07','FRI')	20-APR-07 00:00:00
<i>NEW_TIME</i> (<i>DATE</i> , <i>VAR-CHAR</i> , <i>VAR-CHAR</i>)	<i>DATE</i>	日付と時刻を別のタイムゾーンに変換します	<i>NEW_TIME</i> (<i>TO_DATE</i> '2005/05/29 01:45', 'AST', 'PST')	2005/05/29 21:45:00

次のページに続く

表 12 – 前のページからの続き

Function	Return Type	Description	Example	Result
<i>NUM-TODSIN-TER-VAL(NUMBER, IN-TER-VAL)</i>	<i>INTERVAL</i>	数値を指定された日または秒のインターバルに変換します	<i>SELECT numtodsinterval(100, 'hour');</i>	<i>4 days 04:00:00</i>
<i>NUM-TOYMINTER-VAL(NUMBER, IN-TER-VAL)</i>	<i>INTERVAL</i>	数値を指定された年または月のインターバルに変換します	<i>SELECT numtoyminterval(100, 'month');</i>	<i>8 years 4 mons</i>
<i>ROUND(DATE [, format])</i>	<i>DATE</i>	<i>'format'</i> に従って丸められた日付	<i>ROUND(TO_DATE('29-MAY-05'), 'MON')</i>	<i>01-JUN-05 00:00:00</i>
<i>SYS_EXTRACT_UTC_TIMESTAMP WITH TIME ZONE)</i>	<i>UTC_TIMESTAMP</i>	調整されたユニバーサルタイムを返します	<i>SYS_EXTRACT_UTC(CAST ('24-MAR-11 12:30:00PM -04:00' AS TIMESTAMP WITH TIME ZONE))</i>	<i>` 24-MAR-11 16:30:00`</i>
<i>SYSDATE</i>	<i>DATE</i>	現在の日付と時刻を返します	<i>SYSDATE</i>	<i>01-AUG-12 11:12:34</i>
<i>SYSTIMESTAMP()</i>	<i>TIMESTAMP</i>	現在の日付と時刻を返します	<i>SYSTIMESTAMP</i>	<i>01-AUG-12 11:11:23. 665229 -07:00</i>
<i>TRUNC(DATE [format])</i>	<i>DATE</i>	<i>'format'</i> に従って切り捨てます	<i>TRUNC(TO_DATE ('29-MAY-05'), 'MON')</i>	<i>01-MAY-05 00:00:00</i>
		NA		

ADD_MONTHS

ADD_MONTHS 関数は、指定された日付に指定された月数を加算（または、2 番目のパラメータが負の場合は減算）します。月の最終日は、特定の日付の月の日と同じです。ただし、日が月の最終日である場合は、結果の日付は常に月の最終日になります。

月数パラメータの小数部分は、計算を実行する前に切り捨てられます。

指定された日付に時刻部分が含まれる場合、変更されずに結果に繰り越されます。

以下は、ADD_MONTHS ファンクションの例です。

```
SELECT ADD_MONTHS('13-JUN-07',4) FROM DUAL;
*      add_months
-----
13-OCT-07 00:00:00
(1 row)

SELECT ADD_MONTHS('31-DEC-06',2) FROM DUAL;
*      add_months
-----
28-FEB-07 00:00:00
(1 row)

SELECT ADD_MONTHS('31-MAY-04',-3) FROM DUAL;
*      add_months
-----
29-FEB-04 00:00:00
(1 row)
```

現在の日付/時刻

Advanced Server は、現在の日付と時刻に関連する値を結果多くの機能を提供します。これらの関数はすべて、現在のトランザクションのスタート時間に基づいて値を結果ます。

- CURRENT_DATE
- CURRENT_TIMESTAMP
- LOCALTIMESTAMP
- LOCALTIMESTAMP(precision)

CURRENT_DATE は、現在のトランザクションのスタート時刻に基づいて現在の日付と時刻を返します。CURRENT_DATE の値は、トランザクション内でマルチプル回呼び出されても変更されません。

```
SELECT CURRENT_DATE FROM DUAL;
*      date
```

(次のページに続く)

(前のページからの続き)

```
-----
06-AUG-07
```

CURRENT_TIMESTAMP は現在の日付と時刻を返します。単一の SQL ステートメントから呼び出されると、ステートメント内の各出現に対して同じ値を結果ます。トランザクション内のマルチプルのステートメントから呼び出された場合、発生ごとに異なる値を結果することがあります。ファンクションから呼び出された場合、呼び出し元の current_timestamp によって返される値とは異なる値を結果することがあります。

```
SELECT CURRENT_TIMESTAMP, CURRENT_TIMESTAMP FROM DUAL;
*
current_timestamp | current_timestamp
-----+-----
02-SEP-13 17:52:29.261473 +05:00 | 02-SEP-13 17:52:29.261474 +05:00
```

LOCALTIMESTAMP には、オプションで精度パラメータを指定できます。これにより、秒フィールドの小数点以下の桁数に結果が丸められます。精度パラメータがない場合、結果は使用可能な完全な精度で提供されます。

```
SELECT LOCALTIMESTAMP FROM DUAL;
*
timestamp
-----
06-AUG-07 16:11:35.973
(1 row)

SELECT LOCALTIMESTAMP(2) FROM DUAL;
*
timestamp
-----
06-AUG-07 16:11:44.58
(1 row)
```

これらの関数は現在のトランザクションのスタート時間を結果ため、それらの値はトランザクション中に変更されません。これは機能と見なされます。その目的は、単一のトランザクションに「現在の」時間の一貫した概念を持たせることで、同じトランザクション内のマルチプルの変更と同じタイムスタンプを付けることです。他のデータベースシステムでは、これらの値をより頻繁に進める場合があります。

抽出物

EXTRACT ファンクションは、日付/時刻値から年や時間などのサブフィールドを取得します。EXTRACT ファンクションは、DOUBLE PRECISION 型の値を返します。有効なフィールド名は次のとおりです。

YEAR

年フィールド

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
*
date_part
```

(次のページに続く)

(前のページからの続き)

```

-----
*      2001
(1 row)

```

MONTH

年内の月の番号 (1~12)

```

SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
* date_part
-----
*      2
(1 row)

```

DAY

日 (月) フィールド (1-31)

```

SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
* date_part
-----
*      16
(1 row)

```

HOURL

時間フィールド (0-23)

```

SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
* date_part
-----
*      20
(1 row)

```

MINUTE

分フィールド (0-59)

```

SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
* date_part
-----
*      38
(1 row)

```

SECOND

秒のフィールド (小数部分を含む) (0-59)

```

SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
* date_part
-----
*          40
(1 row)

```

MONTHS_BETWEEN

MONTHS_BETWEEN ファンクションは、2つの日付間の月数を返します。結果は、最初の日付が2番目の日付より大きい場合は正、最初の日付が2番目の日付より小さい場合は負の数値です。

両方の日付パラメーターの月の日が同じである場合、または両方の日付パラメーターがそれぞれの月の最終日に該当する場合、結果は常に整数の月になります。

MONTHS_BETWEEN ファンクションの例を次に示します。

```

SELECT MONTHS_BETWEEN('15-DEC-06','15-OCT-06') FROM DUAL;
* months_between
-----
*          2
(1 row)

SELECT MONTHS_BETWEEN('15-OCT-06','15-DEC-06') FROM DUAL;
* months_between
-----
*         -2
(1 row)

SELECT MONTHS_BETWEEN('31-JUL-00','01-JUL-00') FROM DUAL;
* months_between
-----
*    0.967741935
(1 row)

SELECT MONTHS_BETWEEN('01-JAN-07','01-JAN-06') FROM DUAL;
* months_between
-----
*          12
(1 row)

```

NEXT_DAY

NEXT_DAY ファンクションは、指定された日付より大きい指定された曜日の最初の出現を返します。少なくとも曜日の最初の 3 文字を指定する必要があります-SAT など。指定された日付に時刻部分が含まれる場合、変更されずに結果に繰り越されます。

以下は、NEXT_DAY ファンクションの例です。

```
SELECT NEXT_DAY(TO_DATE('13-AUG-07','DD-MON-YY'),'SUNDAY') FROM DUAL;
*      next_day
-----
19-AUG-07 00:00:00
(1 row)

SELECT NEXT_DAY(TO_DATE('13-AUG-07','DD-MON-YY'),'MON') FROM DUAL;
*      next_day
-----
20-AUG-07 00:00:00
(1 row)
```

NEW_TIME

NEW_TIME ファンクションは、日付と時刻をあるタイムゾーンから別のタイムゾーンに変換します。NEW_TIME は、DATE 型の値を返します。構文は次のとおりです。

```
NEW_TIME(DATE, time_zone1, time_zone2)
```

time_zone1 および time_zone2 は、次の表の [タイムゾーン] 列の文字列値である必要があります。

Time Zone	Offset from UTC	Description
<i>AST</i>	UTC+4	大西洋標準時間
<i>ADT</i>	UTC+3	大西洋の昼光時間
<i>BST</i>	UTC+11	ベーリング標準時間
<i>BDT</i>	UTC+10	ベーリングデイトライトタイム
<i>CST</i>	UTC+6	中央標準時間
<i>CDT</i>	UTC+5	中央昼光時間
<i>EST</i>	UTC+5	東部標準時間
<i>EDT</i>	UTC+4	東部昼間時間
<i>GMT</i>	UTC	グリニッジ平均時間
<i>HST</i>	UTC+10	アラスカ-ハワイ標準時間
<i>HDT</i>	UTC+9	アラスカ-ハワイデイトライトタイム
<i>MST</i>	UTC+7	マウンテンスタンダードタイム
<i>MDT</i>	UTC+6	マウンテンデイトライトタイム
<i>NST</i>	UTC+3:30	ニューファウンドランド標準時間
<i>PST</i>	UTC+8	太平洋標準時間
<i>PDT</i>	UTC+7	パシフィックデイトライトタイム
<i>YST</i>	UTC+9	ユーコン標準時間
<i>YDT</i>	UTC+8	ユーコンデイトライトタイム
		NA

NEW_TIME ファンクションの例を次に示します。

```
SELECT NEW_TIME(TO_DATE('08-13-07 10:35:15', 'MM-DD-YY HH24:MI:SS'), 'AST',
'PST') "Pacific Standard Time" FROM DUAL;
```

Pacific Standard Time

13-AUG-07 06:35:15

(1 row)

NUMTODSINTERVAL

NUMTODSINTERVAL ファンクションは、数値を 1 日から 2 番目のインターバル単位を含む時間インターバルに変換します。ファンクションを呼び出すときに、結果セットに含める最小の非整数インターバルタイプを指定します。有効なインターバルタイプは、DAY、HOUR、MINUTE、および SECOND です。

次の例では、数値を日と時間を含む時間インターバルに変換します。

```
SELECT numtodsinterval(100, 'hour');
```

numtodsinterval

4 days 04:00:00

(1 row)

次の例では、数値を分と秒を含む時間インターバルに変換します。

```
SELECT numtodsinterval(100, 'second' );
numtodsinterval
-----
1 min 40 secs
(1 row)
```

NUMTOYMINTERVAL

NUMTOYMINTERVAL ファンクションは、数値を年から月までのインターバル単位を含む時間インターバルに変換します。ファンクションを呼び出すときに、結果セットに含める最小の非整数インターバルタイプを指定します。有効なインターバルタイプは **YEAR** および **MONTH** です。

次の例では、数値を年と月を含む時間インターバルに変換します。

```
SELECT numtoyminterval(100, 'month' );
numtoyminterval
-----
8 years 4 mons
(1 row)
```

次の例では、数値を年のみを含む時間インターバルに変換します。

```
SELECT numtoyminterval(100, 'year' );
numtoyminterval
-----
100 years
(1 row)
```

ラウンド

ROUND ファンクションは、指定されたテンプレートパターンに従って丸められた日付を返します。テンプレートパターンを省略すると、日付は最も近い日に丸められます。次の表に、ROUND ファンクションのテンプレートパターンを示します。

Pattern	Description
CC, SCC	1 月 1 日を返します。ここで、`cc` は、最後の 2 桁が 50 の場合は指定された年の最初の 2 桁、最後の 2 桁が 50 を超える場合は指定された年の最初の 2 桁より大きいです。（AD 年間）

[次のページに続く](#)

表 13 – 前のページからの続き

Pattern	Description
<i>SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y</i>	1 月 1 日を返します。`yyyy` ここで、`yyyy` は最も近い年に丸められます。6 月 30 日に切り下げ、7 月 1 日に切り上げ
<i>IYYY, IYY, IY, I</i>	月日が 6 月 30 日以前の場合は切り捨て、7 月 1 日以降の場合は切り上げて決定される ISO 年の初めに切り上げます
<i>Q</i>	月日がクォータの第 2 月の 15 日以前の場合は切り捨て、月日が第 2 月の 16 日以降の場合は切り上げて決定されたクォータの第 1 日を返します。クォータ
<i>MONTH, MON, MM, RM</i>	月の日が 15 日以前の場合、指定された月の最初の日を返します。日が 16 日以降の場合、翌月の初日を返します
<i>WW</i>	年の最初の日と同じ曜日に対応する最も近い日付に丸めます
<i>IW</i>	ISO 年の最初の日と同じ曜日に対応する最も近い日付に丸めます
<i>W</i>	月の最初の日と同じ曜日に対応する最も近い日付に丸めます
<i>DDD, DD, J</i>	最も近い日のスタートに丸めます。11:59:59 AM 以前は同じ日のスタートにラウンドします。12:00:00 PM 以降は、翌日のスタートに丸められます
<i>DAY, DY, D</i>	最も近い日曜日に丸めます
<i>HH, HH12, HH24</i>	最も近い時間に丸める
<i>MI</i>	最も近い分に丸める
	NA

ROUND ファンクションの使用例を次に示します。

次の例では、最も近い 100 年に丸めています。

```
SELECT TO_CHAR(ROUND(TO_DATE('1950', 'YYYY'), 'CC'), 'DD-MON-YYYY') "Century" FROM DUAL;
*    Century
-----
01-JAN-1901
(1 row)
```

(次のページに続く)

(前のページからの続き)

```
SELECT TO_CHAR(ROUND(TO_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM DUAL;
*    Century
-----
01-JAN-2001
(1 row)
```

次の例では、最も近い年に丸めます。

```
SELECT TO_CHAR(ROUND(TO_DATE('30-JUN-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY') "Year"
→FROM DUAL;
*    Year
-----
01-JAN-1999
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY') "Year"
→FROM DUAL;
*    Year
-----
01-JAN-2000
(1 row)
```

次の例では、最も近い ISO 年に丸めています。最初の例は 2004 年に丸められ、2004 年の ISO 年は 2003 年 12 月 29 日から始まります。2 番目の例は 2005 年に丸められ、2005 年の ISO 年は 1 月 3 日から始まります rd </ sup >その同じ年の。

(ISO 年は最初の月曜日から始まり、月曜日から日曜日までの 7 日間に新しい年の少なくとも 4 日間が含まれます。したがって、ISO 年のスタートは前年の 12 月に始まる可能性があります。)

```
SELECT TO_CHAR(ROUND(TO_DATE('30-JUN-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY') "ISO
→Year" FROM DUAL;
*    ISO Year
-----
29-DEC-2003
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY') "ISO
→Year" FROM DUAL;
*    ISO Year
-----
03-JAN-2005
(1 row)
```

次の例では、最も近いクォータに丸めます。

```
SELECT ROUND(TO_DATE('15-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;
```

(次のページに続く)

(前のページからの続き)

```

*      Quarter
-----
01-JAN-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;
*      Quarter
-----
01-APR-07 00:00:00
(1 row)

```

次の例では、最も近い月に丸めます。

```

SELECT ROUND(TO_DATE('15-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;
*      Month
-----
01-DEC-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;
*      Month
-----
01-JAN-08 00:00:00
(1 row)

```

次の例では、最も近い週に丸めます。2007 年の最初の日は月曜日に着陸するため、最初の例では、1 月 18 日 th </ sup>は 1 月 15 日 th </ sup>に着陸する月曜日に最も近いです。2 番目の例では、1 月 19 日 th </ sup>は、1 月 22 日 nd </ sup>に当たる月曜日により近いです。

```

SELECT ROUND(TO_DATE('18-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;
*      Week
-----
15-JAN-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('19-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;
*      Week
-----
22-JAN-07 00:00:00
(1 row)

```

次の例では、最も近い ISO 週に丸めます。ISO 週は月曜日に始まります。最初の例では、2004 年 1 月 1 日は 2003 年 12 月 29 日の月曜日に最も近い。2 番目の例では、2004 年 1 月 2 日は 2004 年 1 月 5 日の月曜日に近い。

```

SELECT ROUND(TO_DATE('01-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;
*          ISO Week
-----
29-DEC-03 00:00:00
(1 row)

SELECT ROUND(TO_DATE('02-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;
*          ISO Week
-----
05-JAN-04 00:00:00
(1 row)

```

次の例は、最も近い週に丸められ、週は月の最初の日と同じ日にスタートと見なされます。

```

SELECT ROUND(TO_DATE('05-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;
*          Week
-----
08-MAR-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('04-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;
*          Week
-----
01-MAR-07 00:00:00
(1 row)

```

次の例では、最も近い日に丸めます。

```

SELECT ROUND(TO_DATE('04-AUG-07 11:59:59 AM','DD-MON-YY HH:MI:SS AM'),'J')
"Day" FROM DUAL;
*          Day
-----
04-AUG-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('04-AUG-07 12:00:00 PM','DD-MON-YY HH:MI:SS AM'),'J')
"Day" FROM DUAL;
*          Day
-----
05-AUG-07 00:00:00
(1 row)

```

次の例では、最も近い曜日（日曜日）のスタート日に丸めます。

```

SELECT ROUND(TO_DATE('08-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;
*          Day of Week

```

(次のページに続く)

(前のページからの続き)

```

-----
05-AUG-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('09-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;
*      Day of Week
-----

12-AUG-07 00:00:00
(1 row)

```

次の例では、最も近い時間に丸めます。

```

SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:29','DD-MON-YY HH:MI'),'HH'),'DD-MON-YY,
→HH24:MI:SS') "Hour" FROM DUAL;
*      Hour
-----

09-AUG-07 08:00:00
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30','DD-MON-YY HH:MI'),'HH'),'DD-MON-YY,
→HH24:MI:SS') "Hour" FROM DUAL;
*      Hour
-----

09-AUG-07 09:00:00
(1 row)

```

次の例では、最も近い分に丸めます。

```

SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30:29','DD-MON-YY
HH:MI:SS'),'MI'),'DD-MON-YY HH24:MI:SS') "Minute" FROM DUAL;
*      Minute
-----

09-AUG-07 08:30:00
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30:30','DD-MON-YY
HH:MI:SS'),'MI'),'DD-MON-YY HH24:MI:SS') "Minute" FROM DUAL;
*      Minute
-----

09-AUG-07 08:31:00
(1 row)

```

SYSDATE

SYSDATE ファンクションは、データベースサーバが存在するオペレーティングシステムの現在の日時（タイムタイムゾーンなしのタイムタイムスタンプ）を返します。このファンクションは STABLE であり、引数は不要です。

単一の SQL ステートメントから呼び出されると、ステートメント内の各出現に対して同じ値を結果ます。トランザクション内のマルチプルのステートメントから呼び出された場合、発生ごとに異なる値を結果することがあります。ファンクションから呼び出された場合、呼び出し元の SYSDATE によって返される値とは異なる値を結果場合があります。

次の例は、SYSDATE の呼び出しを示しています。

```
SELECT SYSDATE, SYSDATE FROM DUAL;
*   sysdate          |          sysdate
-----+-----
28-APR-20 16:45:28 | 28-APR-20 16:45:28
(1 row)
```

TRUNC

TRUNC ファンクションは、指定されたテンプレートパターンに従って切り捨てられた日付を返します。テンプレートパターンを省略すると、日付は最も近い日に切り捨てられます。次の表に、TRUNC ファンクションのテンプレートパターンを示します。

Pattern	Description
<i>CC, SCC</i>	1 月 1 日を返しますここで、`cc`は指定された年の最初の 2 桁です
<i>YYYY, YYYY, YEAR, SYEAR, YYY, YY, Y</i>	1 月 1 日を返します`yyyy`ここで、`yyyy`は指定された年です
<i>IYYY, IYY, IY, I</i>	指定された日付を含む ISO 年のスタート日を返します
<i>Q</i>	指定された日付を含むクォータの最初の日を返します
<i>MONTH, MON, MM, RM</i>	指定された月の最初の日を返します
<i>WW</i>	年の最初の日と同じ曜日に対応する、指定された日付の直前または同じ日付を返します
<i>IW</i>	指定された日付を含む ISO 週のスタートを返します

次のページに続く

表 14 – 前のページからの続き

Pattern	Description
<i>W</i>	月の最初の日と同じ曜日に対応する、指定された日付の直前または同じ日付を返します
<i>DDD, DD, J</i>	指定された日付のスタート日を返します
<i>DAY, DY, D</i>	指定された日付を含む週のスタート（日曜日）を返します
<i>HH, HH12, HH24</i>	時間のスタートを返します
<i>MI</i>	分のスタートを返します
	NA

TRUNC ファンクションの使用例を次に示します。

次の例では、100 年単位に切り捨てられます。

```
SELECT TO_CHAR(TRUNC(TO_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM DUAL;
*      Century
-----
01-JAN-1901
(1 row)
```

次の例では、年まで切り捨てられます。

```
SELECT TO_CHAR(TRUNC(TO_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY') "Year"
↪FROM DUAL;
*      Year
-----
01-JAN-1999
(1 row)
```

次の例は、ISO 年の初めまで切り捨てます。

```
SELECT TO_CHAR(TRUNC(TO_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY') "ISO
↪Year" FROM DUAL;
*      ISO Year
-----
29-DEC-2003
(1 row)
```

次の例では、クォータのスタート日まで切り捨てられます。

```

SELECT TRUNC(TO_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;
*
Quarter
-----
01-JAN-07 00:00:00
(1 row)

```

次の例は、月のスタートに切り捨てます。

```

SELECT TRUNC(TO_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;
*
Month
-----
01-DEC-07 00:00:00
(1 row)

```

次の例では、年の最初の日によって決定される週のスタート日まで切り捨てられます。2007 年の最初の日
は月曜日なので、1 月 19 日の直前の月曜日は 1 月 15 日です。

```

SELECT TRUNC(TO_DATE('19-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;
*
Week
-----
15-JAN-07 00:00:00
(1 row)

```

次の例では、ISO 週のスタートに切り捨てます。ISO 週は月曜日に始まります。2004 年 1 月 2 日は、2003
年 12 月 29 日月曜日に始まる ISO 週に該当します。

```

SELECT TRUNC(TO_DATE('02-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;
*
ISO Week
-----
29-DEC-03 00:00:00
(1 row)

```

次の例では、週のスタートに切り捨てられます。この場合、週は月の最初の日と同じ日にスタートと見な
されます。

```

SELECT TRUNC(TO_DATE('21-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;
*
Week
-----
15-MAR-07 00:00:00
(1 row)

```

次の例は、一日のスタートに切り捨てます。

```

SELECT TRUNC(TO_DATE('04-AUG-07 12:00:00 PM','DD-MON-YY HH:MI:SS AM'),'J')
"Day" FROM DUAL;
*
Day
-----

```

(次のページに続く)

(前のページからの続き)

```
04-AUG-07 00:00:00
(1 row)
```

次の例は、週のスタート（日曜日）に切り捨てます。

```
SELECT TRUNC(TO_DATE('09-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;
*      Day of Week
-----
05-AUG-07 00:00:00
(1 row)
```

次の例では、時間のスタートまで切り捨てます。

```
SELECT TO_CHAR(TRUNC(TO_DATE('09-AUG-07 08:30','DD-MON-YY HH:MI'),'HH'),'DD-MON-YY_
↪HH24:MI:SS') "Hour" FROM DUAL;
*      Hour
-----
09-AUG-07 08:00:00
(1 row)
```

次の例では、分に切り捨てます。

```
SELECT TO_CHAR(TRUNC(TO_DATE('09-AUG-07 08:30:30','DD-MON-YY
HH:MI:SS'),'MI'),'DD-MON-YY HH24:MI:SS') "Minute" FROM DUAL;
*      Minute
-----
09-AUG-07 08:30:00
(1 row)
```

1.3.9 Sequence Manipulation Functions

</ div>

このセクションでは、シーケンスオブジェクトを操作するための Advanced Server の機能について説明します。シーケンスオブジェクト（シーケンスジェネレーターまたは単にシーケンスとも呼ばれます）は、CREATE SEQUENCE コマンドで作成された特別な単一行テーブルです。通常、シーケンスオブジェクトは、テーブルの行の一意の識別子を生成するために使用されます。以下にリストされているシーケンス関数は、シーケンスオブジェクトから連続したシーケンス値を取得するためのシンプルマルチユーザーセーフメソッドを提供します。

```
sequence.NEXTVAL
sequence.CURRVAL
```

sequence は、CREATE SEQUENCE コマンドでシーケンスに割り当てられた識別子です。以下に、これらの機能の使用法を説明します。

NEXTVAL

シーケンスオブジェクトを次の値に進め、その値を結果ます。これはアトミックに行われます。マルチプルのセッションが同時に NEXTVAL を実行場合でも、それぞれが個別のシーケンス値を安全に受け取ります。

CURRVAL

現在のセッションでこのシーケンスの NEXTVAL によって最後に取得された値を返します。（このセッションでこのシーケンスに対して NEXTVAL が呼び出されていない場合はエラーが報告されます。）これはセッションローカル値を返すため、現在のセッションが実行されてから他のセッションが NEXTVAL を実行したかどうかにかかわらず、予測可能な回答を提供します。

シーケンスオブジェクトがデフォルトパラメータで作成されている場合、NEXTVAL を呼び出すと、1 から始まる連続した値が結果れます。その他の動作は、CREATE SEQUENCE コマンドで特別なパラメーターを使用して取得できます。

重要：同じシーケンスから番号を取得する同時トランザクションのブロックを回避するため、NEXTVAL オペレーションはロールバックされません。つまり、値がフェッチされると、NEXTVAL を実行したトランザクションが後で中止された場合でも、その値は使用済みと見なされます。これは、中止されたトランザクションが、割り当てられた値のシーケンスに未使用の「穴」を残す可能性があることを意味します。

1.3.10 Conditional Expressions

</div>

次のセクションでは、Advanced Server で使用可能な SQL 準拠の条件式について説明します。

ケース

SQL CASE 式は、他の言語の if / else ステートメントに似た一般的な条件式です。

```
CASE WHEN condition THEN result
*   [ WHEN ... ]
*   [ ELSE result ]
END
```

CASE 句は、式が有効な場所であればどこでも使用できます。condition は、BOOLEAN の結果を返す式です。結果が TRUE の場合、CASE 式の値は条件に続く result です。結果が FALSE の場合、後続の WHEN 句はすべて同じ方法で検索されます。no WHEN condition が TRUE の場合、CASE 式の値は ELSE 句の result です。ELSE 句が省略され、一致する条件がない場合、結果は NULL になります。

例：

```
SELECT * FROM test;
* a
---
1
2
```

(次のページに続く)

(前のページからの続き)

```

3
(3 rows)

SELECT a,
      *   CASE WHEN a=1 THEN 'one'
      *           WHEN a=2 THEN 'two'
      *           ELSE 'other'
      *   END
FROM test;
      * a | case
-----+-----
1 | one
2 | two
3 | other
(3 rows)

```

すべての **result** 式のデータ型は、単一の出力型に変換可能である必要があります。

次の「シンプル」CASE 式は、上記の一般的なフォームの特殊なバリエーションです。

```

CASE expression
      *   WHEN value THEN result
      *   [ WHEN ... ]
      *   [ ELSE result ]
END

```

expression が計算され、等しいものが見つかるまで **WHEN** 句のすべての **value** 仕様と比較されます。マッチが見つからない場合、**ELSE** 句の **result**（または **NULL** 値）が返されます。

上記の例は、シンプル CASE 構文を使用して記述できます。

```

SELECT a,
      *   CASE a WHEN 1 THEN 'one'
      *           WHEN 2 THEN 'two'
      *           ELSE 'other'
      *   END
FROM test;
      * a | case
-----+-----
1 | one
2 | two
3 | other
(3 rows)

```

CASE 式は、結果を決定するために不要な部分式を評価しません。例、これはゼロ除算失敗を回避する可能な方法です。

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

合体

COALESCE ファンクションは、引数の最初の null ではないものを返します。Null は、すべての引数が null の場合にのみ返されます。

```
COALESCE(value [, value2 ] ... )
```

多くの場合、表示または追加の計算のためにデータを取得するときに、デフォルト値を null 値に置き換えるために使用されます。例：

```
SELECT COALESCE(description, short_description, '(none)') ...
```

CASE 式のように、COALESCE は結果を決定するのに必要ではない引数を評価しません。つまり、最初の非 null 引数の右側の引数は評価されません。この SQL 標準関数は、他のデータベースシステムで使用される NVL および IFNULL と同様のファンクションを提供します。

NULLIF

NULLIF と value2 が等しい場合、NULLIF ファンクションは NULL 値を返します。それ以外の場合は、value1 を返します。

```
NULLIF(value1, value2)
```

これを使用して、上記の COALESCE の例の逆オペレーションを実行できます。

```
SELECT NULLIF(value1, '(none)') ...
```

value1 が (なし) の場合は null を結果、そうでない場合は value1 を結果ます。

NVL

NVL ファンクションは、null ではない最初の引数を返します。NVL は最初の式を評価します。その式が NULL に評価される場合、NVL は 2 番目の式を返します。

```
NVL(expr1, expr2)
```

戻り値の型は引数の型と同じです。すべての引数は同じデータタイプでなければなりません（または共通の型に強制変換可能でなければなりません）。すべての引数が NULL の場合、NVL は NULL を返します。

次の例は、非委託従業員のボーナスを計算します。従業員が委託従業員である場合、この式は従業員の手数料を返します。従業員が委託された従業員でない場合（つまり、彼の手数料は NULL）、この式は給与の 10 % のボーナスを返します。

```
bonus = NVL(emp.commission, emp.salary * .10)
```

NVL2

NVL2 は式を評価し、最初の式の値に応じて 2 番目または 3 番目の式を返します。最初の式が NULL でない場合、NVL2 は `expr2` の値を返します。最初の式が NULL の場合、NVL2 は `expr3` の値を返します。

```
NVL2(expr1, expr2, expr3)
```

戻り値の型は引数の型と同じです。すべての引数は同じデータタイプでなければなりません（または共通の型に強制変換可能でなければなりません）。

次の例では、委託された従業員のボーナスを計算します。特定の従業員が委託された従業員である場合、この式はその手数料の 110 % に等しい金額を返します。従業員が委託された従業員でない場合（つまり、彼の委託は NULL）、この式は 0 を返します。

```
bonus = NVL2(emp.commission, emp.commission * 1.1, 0)
```

最大かつ最低

GREATEST および LEAST 関数は、任意の数の式のリストから最大値または最小値を選択します。

```
GREATEST(value [, value2 ] ... )
LEAST(value [, value2 ] ... )
```

式はすべて、結果の型となる共通のデータタイプに変換できる必要があります。リスト内のヌル値は無視されます。すべての式が null と評価される場合にのみ、結果は null になります。

!!! Note * GREATEST と LEAST は SQL 標準にはありませんが、一般的な拡張機能です。

1.3.11 Aggregate Functions

</ div>

Aggregate 関数は、入力値のセットから単一の結果値を計算します。ビルトインの集計関数を次の表にリストします。

Function	Argument Type	Return Type	Description
<i>AVG</i> (<i>expression</i>)	<i>INTEGER</i> , <i>REAL</i> , <i>DOUBLE</i> <i>PRESCISION</i> , <i>NUMBER</i>	<i>NUMBER</i> for any integer type, <i>DOUBLE</i> <i>PRESCISION</i> for a floating- point argument, otherwise the same as the argument data type	すべての入力値の平均（算術平均）
<i>COUNT</i> (*)		<i>BIGINT</i>	入力行数
<i>COUNT</i> (<i>expression</i>)	Any	<i>BIGINT</i>	式の値が null でない入力行の数
<i>MAX</i> (<i>expression</i>)	Any numeric, string, date/time, or bytea	Same as argument type	すべての入力値にわたる式の最大値
1.3. Functions and Operators			85

COUNT を除き、行が選択されていない場合、これらの関数は NULL 値を結果ことに注意してください。特に、行のない SUM は null を返しますが、期待どおりのゼロではありません。必要に応じて、COALESCE ファンクションを使用して、null をゼロに置き換えることができます。

次の表は、統計分析で通常使用される集計関数を示しています。（これらは、より一般的に使用される集約のリストが乱雑になるのを避けるために分離されています。）説明が **N** に言及している場合、それはすべての入力式が null でない入力行の数を意味します。すべての場合において、計算が無意味な場合、例 **N** がゼロの場合、null が返されます。

Function	Argument Type	Return Type	Description
<i>CORR</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	相関係数
<i>COVAR_POP</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	人口共分散
<i>COVAR_SAMP</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	サンプル共分散
<i>REGR_AVGX</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	独立変数の平均 ($\text{sum}(X) / N$)
<i>REGR_AVGY</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	従属変数の平均 ($\text{sum}(Y) / N$)
<i>REGR_COUNT</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	両方の式が非 NULL である入力行の数

次のページに続く

表 15 – 前のページからの続き

Function	Argument Type	Return Type	Description
<i>REGR_INTERCEPT</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	`(X, Y)`ペアによって決定される最小二乗適合線形方程式の y 切片
<i>REGR_R2</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	相関係数の 2 乗
<i>REGR_SLOPE</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	`(X, Y)`ペアによって決定される最小二乗適合線形方程式の勾配
<i>REGR_SXX</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	$\text{Sum}(X^2) - \text{総和}(X)^2 / N$ (独立変数の「二乗総和」)
<i>REGR_SXY</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	$\text{Sum}(X * Y) - \text{総和}(X) * \text{総和}(Y) / N$ (独立した時間従属変数の「積の総和」)
<i>REGR_SYY</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	$\text{Sum}(Y^2) - \text{総和}(Y)^2 / N$ (従属変数の「二乗総和」)

次のページに続く

表 15 – 前のページからの続き

Function	Argument Type	Return Type	Description
<i>STD-DEV(expression)</i>	<i>IN-TE-GER, REAL, DOU-BLE PRE-CI-SION, NUM-BER</i>	<i>DOU-BLE PRE-CI-SION</i> for floating-point arguments, otherwise <i>NUM-BER</i>	<i>STDDEV_SAMP</i> の履歴別名
<i>STD-DEV_POP(expression)</i>	<i>IN-TE-GER, REAL, DOU-BLE PRE-CI-SION, NUM-BER</i>	<i>DOU-BLE PRE-CI-SION</i> for floating-point arguments, otherwise <i>NUM-BER</i>	入力値の母集団標準偏差

次のページに続く

表 15 – 前のページからの続き

Function	Argument Type	Return Type	Description
<i>STD-DEV_SAMP(expression)</i>	<i>IN-TE-GER, REAL, DOU-BLE PRE-CI-SION, NUM-BER</i>	<i>DOU-BLE PRE-CI-SION</i> for floating-point arguments, otherwise <i>NUM-BER</i>	入力値のサンプル標準偏差
<i>VARI-ANCE(expression)</i>	<i>IN-TE-GER, REAL, DOU-BLE PRE-CI-SION, NUM-BER</i>	<i>DOU-BLE PRE-CI-SION</i> for floating-point arguments, otherwise <i>NUM-BER</i>	<i>VAR_SAMP`</i> の履歴別名

次のページに続く

表 15 – 前のページからの続き

Function	Argument Type	Return Type	Description
<i>VAR_POP(expression)</i>	<i>IN- TE- GER, REAL, DOU- BLE PRE- CI- SION, NUM- BER</i>	<i>DOU- BLE PRE- CI- SION for floating- point ar- gu- ments, oth- er- wise NUM- BER</i>	入力値の母集団分散（母集団標準偏差の 2 乗）
<i>VAR_SAMP(expression)</i>	<i>IN- TE- GER, REAL, DOU- BLE PRE- CI- SION, NUM- BER</i>	<i>DOU- BLE PRE- CI- SION for floating- point ar- gu- ments, oth- er- wise NUM- BER</i>	入力値のサンプル分散（サンプル標準偏差の 2 乗）
			NA

LISTAGG

Advanced Server は、文字列の集約をサポートする LISTAGG ファンクションを追加しました。LISTAGG は、マルチプルの行のデータを順序付けられた方法で単一の行に連結する集約ファンクションです。オプションで、データのカスタムデリミタを含めることができます。

LISTAGG ファンクションは、WITHIN GROUP 句の下で ORDER BY 句を使用してメジャー列の値を連結し、順序付けられた集計データを生成することを義務付けています。

目的

- LISTAGG は、グループ化せずに使用できます。この場合、LISTAGG ファンクションはテーブル内のすべての行で動作し、単一の行を返します。
- LISTAGG は、GROUP BY 句とともに使用できます。この場合、LISTAGG ファンクションは各グループで動作し、各グループの集約された出力を返します。
- LISTAGG を OVER 句とともに使用できます。この場合、LISTAGG ファンクションは、`query_partition_by_clause` の式に基づいてクエリー結果セットをグループに分割し、各グループのデータを集計します。

概要

```
LISTAGG( <measure_expr> [, <delimiter> ]) WITHIN GROUP( <order_by_clause> )
[ OVER <query_partition_by_clause> ]
```

パラメータ

measure_expr

`measure_expr` (必須) は、集約する値を割り当てる列または式を指定します。NULL 値は無視されます。

delimiter

`delimiter` (オプション) は、結果行の連結された値を区切る文字列を指定します。`delimiter` には、NULL 値、文字列、文字リテラル、列名前、または定数式を指定できます。無視した場合、LISTAGG ファンクションはデフォルトで NULL 値を使用します。

order_by_clause

`order_by_clause` (必須) は、連結された値が返されるソートオーダーを決定します。

query_partition_by_clause

`query_partition_by_clause` (オプション) は、LISTAGG ファンクションを分析ファンクションとして使用できるようにし、OVER 句で各グループのレコードのレンジを設定します。

返品タイプ

LISTAGG ファンクションは文字列値を返します。

例

次の例では、`emp` テーブルの値を連結し、`delimiter` コンマで区切られたすべての従業員をリストします。まず、`emp` 名前付けのテーブルを作成してから、`emp` テーブルにレコードを挿入します。

```

edb=# CREATE TABLE emp
edb-#      (EMPNO NUMBER(4) NOT NULL,
edb(#      ENAME VARCHAR2(10),
edb(#      JOB VARCHAR2(9),
edb(#      MGR NUMBER(4),
edb(#      HIREDATE DATE,
edb(#      SAL NUMBER(7, 2),
edb(#      COMM NUMBER(7, 2),
edb(#      DEPTNO NUMBER(2));
CREATE TABLE

```

```

edb=# INSERT INTO emp VALUES
edb-#      (7499, 'ALLEN', 'SALESMAN', 7698,
edb(#      TO_DATE('20-FEB-1981', 'DD-MON-YYYY'), 1600, 300, 30);
INSERT 0 1
edb=# INSERT INTO emp VALUES
edb-#      (7521, 'WARD', 'SALESMAN', 7698,
edb(#      TO_DATE('22-FEB-1981', 'DD-MON-YYYY'), 1250, 500, 30);
INSERT 0 1
edb=# INSERT INTO emp VALUES
edb-#      (7566, 'JONES', 'MANAGER', 7839,
edb(#      TO_DATE('2-APR-1981', 'DD-MON-YYYY'), 2975, NULL, 20);
INSERT 0 1
edb=# INSERT INTO emp VALUES
edb-#      (7654, 'MARTIN', 'SALESMAN', 7698,
edb(#      TO_DATE('28-SEP-1981', 'DD-MON-YYYY'), 1250, 1400, 30);
INSERT 0 1
edb=# INSERT INTO emp VALUES
edb-#      (7698, 'BLAKE', 'MANAGER', 7839,
edb(#      TO_DATE('1-MAY-1981', 'DD-MON-YYYY'), 2850, NULL, 30);
INSERT 0 1

edb=# SELECT LISTAGG(ENAME, ',') WITHIN GROUP (ORDER BY ENAME) FROM emp;
*
listagg
-----
ALLEN,BLAKE,JONES,MARTIN,WARD
(1 row)

```

次の例は、emp テーブルの LISTAGG で PARTITION BY 句を使用し、テーブル全体ではなく各パーティションに適用される deptno によるパーティションに基づいて出力を生成します。

```

edb=# SELECT DISTINCT DEPTNO, LISTAGG(ENAME, ',') WITHIN GROUP (ORDER BY
ENAME) OVER(PARTITION BY DEPTNO) FROM emp;
deptno | listagg
-----+-----

```

(次のページに続く)

(前のページからの続き)

```
*      30 | ALLEN, BLAKE, MARTIN, WARD
*      20 | JONES
(2 rows)
```

次の例は、GROUP BY 句が含まれていることを除いて、前の例と同じです。

```
edb=# SELECT DEPTNO, LISTAGG(ENAME, ',') WITHIN GROUP (ORDER BY ENAME) FROM
emp GROUP BY DEPTNO;
 deptno |          listagg
-----+-----
*      20 | JONES
*      30 | ALLEN, BLAKE, MARTIN, WARD
(2 rows)
```

中央値

MEDIAN ファンクションは、指定された値のレンジから式の間値を計算します。NULL 値は無視されます。クエリーがユーザー定義テーブルをリファレンスしない場合、MEDIAN ファンクションはエラーを返します。

目的

- MEDIAN は、グループ化せずに使用できます。この場合、MEDIAN ファンクションはテーブル内のすべての行で動作し、単一の行を返します。
- MEDIAN 句は OVER 句とともに使用できます。この場合、MEDIAN ファンクションは、PARTITION BY 句で指定された **expression** に基づいてクエリー結果セットをグループに分割し、各グループのデータを集計します。

概要

```
MEDIAN( <median_expression> ) [ OVER ( [ PARTITION BY... ] ) ]
```

パラメータ

median_expression

median_expression (必須) は、MEDIAN ファンクションが処理し、中央値(メジアン) 値を返すターゲット列または式です。数値、日時、またはインターバルのデータタイプにすることができます。

PARTITION BY

PARTITION BY 句 (オプション) を使用すると、MEDIAN ファンクションを分析ファンクションとして使用でき、OVER 句で各グループのレコードのレンジを設定できます。

返品タイプ

戻り値の型は、**expression** の入力データタイプによって決まります。次の表は、各入力タイプの戻り値の型を示しています。

Input Type	Return Type
<i>BIGINT</i>	<i>NUMERIC</i>
<i>FLOAT, DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>
<i>INTEGER</i>	<i>NUMERIC</i>
<i>INTERVAL</i>	<i>INTERVAL</i>
<i>NUMERIC</i>	<i>NUMERIC</i>
<i>REAL</i>	<i>REAL</i>
<i>SMALLINT</i>	<i>NUMERIC</i>
<i>TIMESTAMP</i>	<i>TIMESTAMP</i>
<i>TIMESTAMP TZ</i>	<i>TIMESTAMP TZ</i>

例

次の例では、クエリーは **emp** テーブルの各部門の給与の中央値 (メジアン) を返します。

```
edb=# SELECT * FROM emp;
 empno|  ename  |  job   | mgr  |      hiredate      |  sal  | comm  | deptno
-----+-----+-----+-----+-----+-----+-----+-----
 7369 | SMITH   | CLERK   | 7902 | 17-DEC-80 00:00:00 | 800.00 |      |    20
 7499 | ALLEN   | SALESMAN| 7698 | 20-FEB-81 00:00:00 | 1600.00 | 300.0 |    30
 7521 | WARD    | SALESMAN| 7698 | 22-FEB-81 00:00:00 | 1250.00 | 500.00 |    30
 7566 | JONES   | MANAGER | 7839 | 02-APR-81 00:00:00 | 2975.00 |      |    20
 7654 | MARTIN  | SALESMAN| 7698 | 28-SEP-81 00:00:00 | 1250.00 | 1400.00 |    30
(5 rows)

* edb=# SELECT MEDIAN (SAL) FROM emp;
* median
-----
*    1250
(1 row)
```

次の例では、**emp** テーブルの **MEDIAN** で **PARTITION BY** 句を使用し、**deptno** によるパーティションに基づいて給与の中央値 (メジアン) を返します。

```
edb=# SELECT EMPNO, ENAME, DEPTNO, MEDIAN (SAL) OVER (PARTITION BY DEPTNO)
FROM emp;
 empno |  ename  | deptno | median
-----+-----+-----+-----
* 7369 | SMITH   |    20 | 1887.5
* 7566 | JONES   |    20 | 1887.5
* 7499 | ALLEN   |    30 | 1250
* 7521 | WARD    |    30 | 1250
* 7654 | MARTIN  |    30 | 1250
(5 rows)
```

MEDIAN ファンクションは、PERCENTILE_CONT と比較できます。次の例では、MEDIAN は PERCENTILE_CONT と同じ結果を生成します。

```
edb=# SELECT MEDIAN (SAL), PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY SAL)
FROM emp;
 median | percentile_cont
-----+-----
*  1250 |          1250
(1 row)
```

STATS_MODE

STATS_MODE ファンクションは、一連の値を引数として取り、最も高い頻度で発生する値を返します。同じ頻度でマルチプル値が表示される場合、STATS_MODE ファンクションは最初の値を任意に選択し、その 1 つの値のみを返します。

目的

- STATS_MODE ファンクションは、グループ化せずに使用できます。この場合、STATS_MODE ファンクションはテーブル内のすべての行で動作し、単一の値を返します。
- STATS_MODE は、WITHIN GROUP 句を使用して、順序集合集約ファンクションとして使用できます。この場合、STATS_MODE ファンクションは、順序付けられたデータセットで動作します。
- STATS_MODE を GROUP BY 句とともに使用できます。この場合、STATS_MODE ファンクションは各グループで動作し、各グループの最も頻繁で集約された出力を返します。

概要

```
STATS_MODE( <expr> )
```

または

```
STATS_MODE() WITHIN GROUP ( ORDER BY sort_expression )
```

パラメータ

expr

列に割り当てる式または値。

返品タイプ

STATS_MODE ファンクションは、頻繁に表示される値を返します。ただし、列のすべての値が NULL である場合、STATS_MODE は NULL を返します。

例

次の例は、emp テーブルの給与モードを返します。

```
edb=# SELECT * FROM emp;
 empno|  ename  |   job   | mgr |      hiredate      |   sal   | comm   | deptno
-----+-----+-----+-----+-----+-----+-----+-----
 7369 | SMITH   | CLERK    | 7902 | 17-DEC-80 00:00:00 | 800.00 |        |    20
 7499 | ALLEN   | SALESMAN | 7698 | 20-FEB-81 00:00:00 | 1600.00 | 300.0  |    30
 7521 | WARD    | SALESMAN | 7698 | 22-FEB-81 00:00:00 | 1250.00 | 500.00 |    30
 7566 | JONES   | MANAGER  | 7839 | 02-APR-81 00:00:00 | 2975.00 |        |    20
 7654 | MARTIN  | SALESMAN | 7698 | 28-SEP-81 00:00:00 | 1250.00 | 1400.00 |    30
(5 rows)

edb=# SELECT STATS_MODE(SAL) FROM emp;
 stats_mode
-----
* 1250.00
(1 row)
```

次の例では、emp テーブルで STATS_MODE とともに GROUP BY および ORDER BY 句を使用し、deptno によるパーティションに基づいて給与を返します。

```
edb=# SELECT STATS_MODE(SAL) FROM emp GROUP BY DEPTNO ORDER BY DEPTNO;
 stats_mode
-----
* 800.00
* 1250.00
(2 rows)
```

次の例では、WITHIN GROUP 句を STATS_MODE ファンクションとともに使用して、順序付けられたデータセットで集計を実行します。

```
SELECT STATS_MODE() WITHIN GROUP(ORDER BY SAL) FROM emp;
 stats_mode
-----
* 1250.00
(1 row)
```

BIT_AND_AGG

BIT_AND_AGG は、ビット単位の AND オペレーションを実行し、入力引数のデータタイプに基づいて値を返すビット単位の集計ファンクションです。

目的

- BIT_AND_AGG ファンクションは、GROUP BY 句とともに使用できます。この場合、BIT_AND_AGG ファンクションはグループを操作し、ビット単位の AND オペレーションの結果を返します。
- DISTINCT または UNIQUE キーワードを BIT_AND_AGG ファンクションとともに使用して、保証の一意の値が計算に使用されるようにすることができます。

概要

```
BIT_AND_AGG ( [DISTINCT | ALL | UNIQUE] <expr> )
```

パラメータ

expr

列に割り当てる式または値。

返品タイプ

BIT_AND_AGG ファンクションは、入力引数のデータタイプと同じ値を返します。ただし、列のすべての値が NULL の場合、BIT_AND_AGG は NULL を返します。

例

次の例では、BIT_AND_AGG ファンクションを emp テーブルの sal 列に適用し、deptno で結果をグループ化します。

```
edb=# SELECT deptno,BIT_AND_AGG(DISTINCT sal) FROM emp GROUP BY deptno;
 deptno | bit_and_agg
-----+-----
      * | 20 |      2975.00
      * | 30 |           0
(2 rows)
```

BIT_OR_AGG

BIT_OR_AGG は、ビット単位の OR オペレーションを実行し、入力引数のデータタイプに基づいて値を返すビット単位の集計ファンクションです。

目的

- BIT_OR_AGG ファンクションは、GROUP BY 句とともに使用できます。この場合、BIT_OR_AGG ファンクションはグループを操作し、ビット単位の OR オペレーションの結果を返します。
- DISTINCT または UNIQUE キーワードを BIT_OR_AGG ファンクションとともに使用して、保証の一意の値が計算に使用されるようにすることができます。

概要

```
BIT_OR_AGG ( [DISTINCT | ALL | UNIQUE] <expr> )
```

パラメータ

expr

列に割り当てる式または値。

返品タイプ

BIT_OR_AGG ファンクションは、入力引数のデータタイプと同じ値を返します。ただし、列のすべての値が NULL の場合、BIT_OR_AGG は NULL を返します。

例

次の例では、BIT_OR_AGG ファンクションを emp テーブルの sal 列に適用し、deptno で結果をグループ化します。

```
edb=# SELECT deptno,BIT_OR_AGG(DISTINCT sal) FROM emp GROUP BY deptno;
 deptno | bit_or_agg
-----+-----
      * | 20 |      2975.00
      * | 30 |         4066
(2 rows)
```

1.3.12 Subquery Expressions

</ div>

このセクションでは、Advanced Server で使用可能な SQL 準拠のサブクエリ式について説明します。このセクションに記載されているすべての式フォームは、ブール値 (TRUE/FALSE) の結果を結果ます。

存在

EXISTS の引数は、任意の SELECT ステートメントまたはサブクエリです。サブクエリは、行を返すかどうかを判断するために評価されます。少なくとも 1 つの行を返す場合、EXISTS の結果は TRUE です。サブクエリが行を返さない場合、EXISTS の結果は FALSE です。

EXISTS(subquery)

サブクエリは、周囲のクエリーから変数を参照できます。変数は、サブクエリの評価の際に定数として機能します。

通常、サブクエリは、完了までではなく、少なくとも 1 つの行が返されるかどうかを判断するのに十分なだけ実行されます。副作用（シーケンス関数の呼び出しなど）を持つサブクエリを記述することは賢明ではありません。副作用が発生するかどうかを予測するのは難しいかもしれません。

結果は行が返されるかどうかのみ依存し、それらの行の内容には依存しないため、通常、サブクエリの出力リストは重要ではありません。一般的なコーディング規則は、すべての EXISTS テストを EXISTS(SELECT 1 WHERE ...) のフォームで記述することです。ただし、INTERSECT を使用するサブクエリなど、このルールには例外があります。

このシンプル例は deptno の内部結合に似ていますが、マルチプルのマッチング emp 行がある場合でも、各 dept 行に対して最大 1 つの出力行を生成します。

```

SELECT dname FROM dept WHERE EXISTS (SELECT 1 FROM emp WHERE emp.deptno =
dept.deptno);
      *      dname
-----
ACCOUNTING
RESEARCH
SALES
(3 rows)

```

IN

右側は括弧で囲まれたサブクエリで、正確に 1 つの列を結果必要があります。左側の式が評価され、サブクエリの結果の行と比較されます。等しいサブクエリ行が見つかった場合、**IN** の結果は **TRUE** になります。等しい行が見つからない場合、結果は **FALSE** です（サブクエリが行を返さない特別な場合を含む）。

```
expression IN (subquery)
```

左側の式が **NULL** を生成する場合、または右側の値が等しくなく、少なくとも 1 つの右側の行が **NULL** を生成する場合、**FALSE** ではなく **IN** コンストラクト体の結果になることに注意してください。これは、**null** 値のブールの組み合わせに関する SQL の通常の規則に従っています。

EXISTS と同様に、サブクエリが完全に評価されると仮定するのは賢明ではありません。

NOT IN

右側は括弧で囲まれたサブクエリで、正確に 1 つの列を結果必要があります。左側の式が評価され、サブクエリの結果の行と比較されます。等しくないサブクエリ行だけが見つかった場合、**NOT IN** の結果は **TRUE** になります（サブクエリが行を返さない特別な場合を含む）。等しい行が見つかった場合、結果は **FALSE** です。

```
expression NOT IN (subquery)
```

左側の式が **NULL** を生成する場合、または右側の値が等しくなく、少なくとも 1 つの右側の行が **NULL** を生成する場合、**TRUE** ではなく **NOT IN** コンストラクト体の結果になることに注意してください。これは、**null** 値のブールの組み合わせに関する SQL の通常の規則に従っています。

EXISTS と同様に、サブクエリが完全に評価されると仮定するのは賢明ではありません。

任意/一部

右側は括弧で囲まれたサブクエリで、正確に 1 つの列を結果必要があります。左側の式が評価され、指定された演算子を使用してサブクエリ結果の行と比較されます。演算子はブール値の結果を生成する必要があります。真の結果が得られた場合、**ANY** の結果は **TRUE** になります。真の結果が見つからない場合、結果は **FALSE** です（サブクエリが行を返さない特別な場合を含む）。

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

SOME は **ANY** の同義語です。IN は **ANY** と同等です。

成功せず、少なくとも 1 つの右側の行が演算子の結果として **NULL** を生成する場合、**ANY** コンストラクトの結果は **FALSE** ではなく **NULL** になることに注意してください。これは、**null** 値のブールの組み合わせに関する SQL の通常の規則に従っています。

EXISTS と同様に、サブクエリが完全に評価されると仮定するのは賢明ではありません。

すべて

右側は括弧で囲まれたサブクエリで、正確に 1 つの列を結果必要があります。左側の式が評価され、指定された演算子を使用してサブクエリ結果の行と比較されます。演算子はブール値の結果を生成する必要があります。すべての行が **true** になる場合、**ALL** の結果は **TRUE** になります（サブクエリが行を返さない特別な場合を含む）。偽の結果が見つかった場合、結果は **FALSE** です。比較がどの行に対しても **FALSE** を結果ない場合、結果は **NULL** になり、少なくとも 1 行に対して **NULL** を返します。

```
expression operator ALL (subquery)
```

NOT IN は **<> ALL** と同等です。**EXISTS** と同様に、サブクエリが完全に評価されると仮定するのは賢明ではありません。

1.3.13 Identifier Functions

</ div>

次のセクションでは、インスタンスとセッションに関する情報を提供する識別子関数について説明します。

SYS_GUID

SYS_GUID ファンクションは、グローバルに一意的識別子を生成して返します。識別子は、16 バイトの **RAW** データのフォームを取ります。**SYS_GUID** ファンクションは、**uuid-oss** モジュールに基づいて、普遍的に一意的識別子を生成します。シグネチャは次のとおりです。

```
SYS_GUID()
```

例

次の例では、テーブル **EMP** に列を追加し、一意の識別子を挿入して、16-byte RAW 値を返します。

```
edb=# CREATE TABLE EMP(C1 RAW (16) DEFAULT SYS_GUID() PRIMARY KEY, C2 INT);
CREATE TABLE
edb=# INSERT INTO EMP(C2) VALUES (1);
INSERT 0 1
edb=# SELECT * FROM EMP;
      *                  c1                  | c2
-----+-----
\xb944970d3a1b42a7a2119265c49cbb7f | 1
(1 row)
```

USERENV

USERENV ファンクションは、現在のセッションに関する情報を取得します。シグネチャは次のとおりです。

```
USERENV(<parameter>)
```

parameter は、現在のセッションから結果値を指定します。可能な **parameter** 値は次のとおりです。

Parameter	Description
	NA
<i>ISDBA</i>	現在のユーザが DBA 権限を持っている場合は `TRUE` を返し、そうでない場合は `FALSE` を返します。
	NA
<i>LANGUAGE</i>	現在のセッションの言語、地域、キャラクタセット（次のフォーマット）： <i>language_territory.characterset</i>
	NA
<i>LANG</i>	言語名前の ISO 略語、既存の `LANGUAGE` パラメータの短縮名前。
	NA
<i>SID</i>	現在のセッション識別子。
	NA
<i>TERMINAL</i>	現在のセッションのオペレーティングシステムのターミナル識別子。
	NA

例

次の例では、現在のセッションの **ISDBA** パラメータを返します。

```
edb=# SELECT USERENV('ISDBA') FROM DUAL;
userenv
-----
TRUE
(1 row)
```

次の例では、現在のセッションの LANG パラメータを返します。

```
edb=# SELECT USERENV('LANG') FROM DUAL;
userenv
-----
en
(1 row)
```

次の例では、現在のセッションの LANGUAGE パラメータを返します。

```
edb=# SELECT USERENV('LANGUAGE') FROM DUAL;
*      userenv
-----
English_USA.UTF-8
(1 row)
```

次の例では、TERMINAL 識別子を返します。

```
edb=# SELECT USERENV('TERMINAL') FROM DUAL;
userenv
-----
[local]
(1 row)
```

次の例では、現在のセッションの SID 番号を返します。

```
edb=# SELECT USERENV('SID') FROM DUAL;
userenv
-----
56867
(1 row)
```

SYS_CONTEXT

SYS_CONTEXT ファンクションは、現在のセッションのコンテキスト namespace に関連付けられた parameter の値を返します。シグネチャは次のとおりです。

```
SYS_CONTEXT(<namespace>, <parameter>)
```

または

```
SYS_CONTEXT(<userenv>, <parameter>)
```

パラメータ

namespace

namespace には、任意の名前付けコンテキストを指定できます。USERENV は、現在のセッションに関する情報を表示するビルトインコンテキストです。

parameter

parameter は、名前空間の定義済み属性です。次の表に、USERENV 名前空間の定義済み属性を示します。

パラメータ	説明
ISDBA	現在のユーザが DBA 特権を持っている場合は TRUE を返し、そうでない場合は FALSE を返します。
LANGUAGE	次のフォーマットの現在のセッションの言語、地域、およびキャラクタセット: language_territory_characterset
LANG	言語名前の ISO 略語、既存の LANGUAGE パラメータの短縮名前。
SID	現在のセッション識別子。
TERMINAL	現在のセッションのオペレーティングシステムターミナル識別子。

例

次の例では、ビルトインの USERENV 名前空間が SYS_CONTEXT ファンクションとともに使用されます。

次の例では、現在のセッションの ISDBA パラメータを返します。

```
edb=# SELECT SYS_CONTEXT('USERENV','ISDBA') AS ISDBA FROM DUAL;
isdba
-----
TRUE
(1 row)
```

次の例では、現在のセッションの LANG パラメータを返します。

```
edb=# SELECT SYS_CONTEXT('USERENV','LANG') AS LANG FROM DUAL;
lang
-----
en
(1 row)
```

次の例では、現在のセッションの LANGUAGE パラメータを返します。

```
edb=# SELECT SYS_CONTEXT('USERENV','LANGUAGE') AS LANGUAGE FROM DUAL;
*      language
-----
English_USA.UTF-8
(1 row)
```

次の例では、TERMINAL 識別子を返します。

```
edb=# SELECT SYS_CONTEXT('USERENV','TERMINAL') AS TERMINAL FROM DUAL;
terminal
-----
[local]
(1 row)
```

次の例では、現在のセッションの SID 番号を返します。

```
edb=# SELECT SYS_CONTEXT('USERENV','SID') AS SID FROM DUAL;
      * sid
-----
    56867
(1 row)
```

1.3.14 Bitwise Functions

</ div>

次のセクションでは、ビット単位関数 **BITAND** および **BITOR** について説明します。

ビット

BITAND ファンクションは、ビット単位の **AND** オペレーションを実行し、入力引数のデータタイプに基づいて値を返します。シグネチャは次のとおりです。

```
BITAND(<expr1>, <expr2>)
```

expr1 および **expr2** の入力データタイプは、ビット単位の **AND** オペレーションが実行される **NUMBER** です。**BITAND** は、ビット単位の **AND** オペレーションを使用して、**expr1** 値の各ビットを **expr2** の対応するビットと比較し、入力引数のデータタイプに基づいて値を返します。**BITAND** のいずれかの引数が **NULL** である場合、結果は **NULL** 値です。

返品タイプ

BITAND ファンクションは、入力引数のデータタイプと同じ値を返します。

例

次の例では、番号 10 および 11 で **AND** オペレーションを実行します。

```
edb=# SELECT BITAND(10,11) FROM DUAL;
      bitand
-----
      *    10
(1 row)
```

BITOR

BITOR ファンクションは、ビット単位の OR オペレーションを実行し、入力引数のデータタイプに基づいて値を返します。シグネチャは次のとおりです。

```
BITOR(<expr1>, <expr2>)
```

BITOR は、ビット単位の OR オペレーションを使用して、**expr1** 値の各ビットを **exp2** の対応するビットと比較し、入力引数のデータタイプに基づいて値を返します。

返品タイプ

BITOR ファンクションは、入力引数のデータタイプと同じ値を返します。ただし、列のすべての値が NULL の場合、BITOR は NULL を返します。

例

次の例では、番号 10 および 11 で OR オペレーションを実行します。

```
edb=# SELECT BITOR(10,11) FROM DUAL;
      bitor
-----
      *    11
(1 row)
```

ビットワイズ演算子

ビット演算子は、BITAND および BITOR と同じオペレーションを実行します。ビット演算子は次のとおりです。

- **& :** (Bitwise AND) 入力タイプで実行されたビット単位の AND オペレーションの結果を返します。
- **| :** (Bitwise OR) 入力タイプで実行されたビット単位の OR オペレーションの結果を返します。

例

次の例では、&演算子を使用し、入力 of BITAND として出力を返します。

```
edb=# SELECT 10&11 FROM DUAL;
      ?column?
-----
      *    10
(1 row)
```

次の例では、|演算子を使用して、出力を入力の BITOR として返します。

```
edb=# SELECT 10|11 FROM DUAL;
      ?column?
-----
```

(次のページに続く)

(前のページからの続き)

* 11

(1 row)

第2章 System Catalog Tables

</div>

次のシステムカタログテーブルには、データベースオブジェクトの定義が含まれています。システムテーブルのレイアウトは変更されるサブジェクト性があります。システムテーブルに格納されている情報に依存するアプリケーションを作成する場合は、既存のカタログビューを使用するか、カタログビューを作成してシステムテーブルの変更からアプリケーションを分離することをお勧めします。

2.1 デュアル

`dual` は、オラクルデータベースのみとの互換性のために提供される単一行、単一系列のテーブルです。

Column	Type	Modifiers	Description
<i>dummy</i>	<i>VARCHAR2(1)</i>		互換性のためにのみ提供されています。
			NA

2.2 edb_dir

`edb_dir` テーブルには、`CREATE DIRECTORY` コマンドで作成されたディレクトリを指す別名ごとに 1 行が含まれます。ディレクトリは、ホストファイルシステムへのユーザ限定アクセスを許可するパス名の別名です。

ディレクトリを使用して、ファイルシステム内の特定のディレクトリツリーにユーザをフェンスできます。例、`UTL_FILE` パッケージは、ユーザがホストファイルシステム内のファイルとディレクトリの読み取りと書き込みを許可する機能を提供しますが、データベース管理者が `CREATE DIRECTORY` コマンドを介してアクセスを許可した経路へのアクセスのみを許可します。

Column	Type	Modifiers	Description
<i>dirname</i>	<i>*name*</i>	<i>not null</i>	別名の名前。
<i>dirowner</i>	<i>oid</i>	<i>not null</i>	別名を所有するユーザの `OID`。
<i>dirpath</i>	<i>text</i>		アクセスが許可されるディレクトリ名前。
<i>diracl</i>	<i>aclitem[]</i>		別名にアクセスできるユーザーを決定するアクセス制御リスト。
			NA

2.3 edb_password_history

edb_password_history テーブルには、パスワードの変更ごとに 1 行が含まれます。テーブルは、クラスター内のすべてのデータベースで共有されます。

Column	Type	References	Description
<i>passhistroleid</i>	<i>oid</i>	<i>pg_authid.oid</i>	ロールの `ID`。
<i>passhistpassword</i>	<i>text</i>		md5 暗号化フォームのロールパスワード。
<i>passhistpasswordsetat</i>	<i>timestampz</i>		パスワードが設定された時刻。
			NA

2.4 edb_policy

edb_policy テーブルには、ポリシーごとに 1 つの行が含まれます。

Column	Type	Modifiers	Description
<i>policyname</i>	<i>name</i>	<i>not null</i>	ポリシー名前。
<i>policygroup</i>	<i>oid</i>	<i>not null</i>	現在未使用です。
<i>policyobject</i>	<i>oid</i>	<i>not null</i>	このポリシーによって保護されたテーブルの `OID` (<i>object_schema</i> と <i>object_name</i>)。
<i>policykind</i>	<i>char</i>	<i>not null</i>	このポリシーによって保護されるオブジェクトの種類：テーブルの場合は「r」、ビューの場合は「v」、シノニムの場合は=。現在、常に「r」。
<i>policyproc</i>	<i>oid</i>	<i>not null</i>	ポリシー関クションの `OID` (<i>function_schema</i> と <i>policy_function</i>)。
<i>policyinsert</i>	<i>boolean</i>	<i>not null</i>	ポリシーが `INSERT` ステートメントによって強制される場合は True。
<i>policyselect</i>	<i>boolean</i>	<i>not null</i>	ポリシーが `SELECT` ステートメントによって強制される場合は True。
<i>policydelete</i>	<i>boolean</i>	<i>not null</i>	ポリシーが `DELETE` ステートメントによって強制される場合は True。
<i>policyupdate</i>	<i>boolean</i>	<i>not null</i>	ポリシーが `UPDATE` ステートメントによって強制される場合は True。
<i>policyindex</i>	<i>boolean</i>	<i>not null</i>	現在未使用です。
<i>policyenabled</i>	<i>boolean</i>	<i>not null</i>	ポリシーが有効になっている場合は True。
<i>policyupdatecheck</i>	<i>boolean</i>	<i>not null</i>	`UPDATE` ステートメントによって更新された行がポリシーを満たす必要がある場合は True。
<i>policystatic</i>	<i>boolean</i>	<i>not null</i>	現在未使用です。
<i>policytype</i>	<i>integer</i>	<i>not null</i>	現在未使用です。
<i>policyopts</i>	<i>integer</i>	<i>not null</i>	現在未使用です。
<i>policyseccols</i>	<i>int2vector</i>	<i>not null</i>	`sec_relevant_cols` にリストされている列の列番号。
			NA

2.5 edb_profile

edb_profile テーブルには、使用可能なプロファイルに関する情報が格納されます。edb_profiles は、クラスター内のすべてのデータベースで共有されます。

Column	Type	References	Description
<i>oid</i>	<i>oid</i>		行識別子（非表示の属性; 明示的に選択する必要があります）。
<i>prfname</i>	<i>name</i>		プロファイルの名前。
<i>prffailedloginattempts</i>	<i>integer</i>		プロファイルで許可されたログイン試行の失敗回数。 -1 は、デフォルトプロファイルの値を使用する必要があることを示します。 -2 は、失敗したログイン試行に制限がないことを示します。
<i>prfpasswordlocktime</i>	<i>integer</i>		プロファイルに関連付けられたパスワードロック時間（秒単位）。 -1 は、デフォルトプロファイルの値を使用する必要があることを示します。 -2 は、アカウントを永続的にロックする必要があることを示します。
<i>prfpasswordlifetime</i>	<i>integer</i>		プロファイルに関連付けられたパスワードの有効期間（秒単位）。 -1 は、デフォルトプロファイルの値を使用する必要があることを示します。 -2 は、パスワードの有効期限が切れないことを示します。
<i>prfpasswordgracetime</i>	<i>integer</i>		プロファイルに関連付けられたパスワード猶予時間（秒単位）。 -1 は、デフォルトプロファイルの値を使用する必要があることを示します。 -2 は、パスワードの有効期限が切れないことを示します。
<i>prfpasswordreuse-time</i>	<i>integer</i>		ユーザがパスワードを再利用するまでに待機する必要がある秒数。 -1 は、デフォルトプロファイルの値を使用する必要があることを示します。 -2 は、古いパスワードを再利用できないことを示します。
<i>prfpasswordreuse-max</i>	<i>integer</i>		パスワードを再利用する前に発生する必要があるパスワード変更の数。 -1 は、デフォルトプロファイルの値を使用する必要があることを示します。 -2 は、古いパスワードを再利用できないことを示します。
<i>prfpasswordallow-lowhashed</i>	<i>integer</i>		パスワード <code>allow hashed</code> パラメータは、暗号化されたパスワードの使用を許可するかどうかを指定します。可能な値は、 <code>true/on/yes/1</code> 、 <code>false/off/no/0</code> 、および <code>DEFAULT</code> です。
<i>prfpasswordverify-funcdb</i>	<i>oid</i>	<i>pg_database.oid</i>	パスワード検証関数が存在するデータベースの <code>'OID'</code> 。
<i>prfpasswordverify-func</i>	<i>oid</i>	<i>pg_proc.oid</i>	プロファイルに関連付けられたパスワード検証関数の <code>'OID'</code> 。
			NA

2.6 edb_variable

edb_variable テーブルには、パッケージレベル変数（パッケージ内で宣言された各変数）ごとに 1 行が含まれます。

Column	Type	Modifiers	Description
<i>varname</i>	<i>*name*</i>	<i>not null</i>	変数の名前。
<i>varpackage</i>	<i>oid</i>	<i>not null</i>	パッケージを格納する `pg_namespace` 行の `OID`。
<i>vartype</i>	<i>oid</i>	<i>not null</i>	変数のタイプを定義する `pg_type` 行の `OID`。
<i>varaccess</i>	<i>*char*</i>	<i>not null</i>	変数がパッケージの外部に可視されている場合は `+`。変数がパッケージ内でのみ可視される場合は `-`。注: パブリック変数はパッケージヘッダー内で宣言されています; プライベート変数はパッケージ本体内で宣言されます。
<i>varsrc</i>	<i>text</i>		変数のデフォルト値式を含む、変数宣言のソースが含まれる変数。
<i>varseq</i>	<i>smallint</i>	<i>not null</i>	パッケージで変数が宣言されたオーダー。
			NA

2.7 pg_synonym

pg_synonym テーブルには、CREATE SYNONYM コマンドまたは CREATE PUBLIC SYNONYM コマンドで作成されたシノニムごとに 1 行が含まれます。

Column	Type	Modifiers	Description
<i>synname</i>	<i>*name*</i>	<i>not null</i>	同義語の名前。
<i>synnamespace</i>	<i>oid</i>	<i>not null</i>	`synowner` を置き換えます。同義語が格納されている `pg_namespace` 行の `OID` が含まれます。
<i>synowner</i>	<i>oid</i>	<i>not null</i>	同義語を所有するユーザの `OID`。
<i>synobjschema</i>	<i>*name*</i>	<i>not null</i>	被参照されるオブジェクトが定義されているスキーマ。
<i>synobjname</i>	<i>*name*</i>	<i>not null</i>	被参照オブジェクトの名前。
<i>synlink</i>	<i>text</i>		被参照されるオブジェクトが定義されているデータベースリンクの（オプションの）名前。
			NA

2.8 product_component_version

product_component_version テーブルには、機能の互換性に関する情報が含まれています。アプリケーションは、インストール時または実行時にこのテーブルをクエリーして、アプリケーションで使用される機能がこの展開で使用可能であることを確認できます。

Column	Type	Description
<i>product</i>	<i>character varying (74)</i>	製品の名前。
<i>version</i>	<i>character varying (74)</i>	製品のバージョン番号。
<i>status</i>	<i>character varying (74)</i>	リリースのステータス。