



Database Compatibility for Oracle Developers Reference Guide

Release 14.1.0

EnterpriseDB

Dec 14, 2021

CONTENTS

1	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	7
1.2.1	Numeric Types	8
1.2.2	Character Types	10
1.2.3	Binary Data	12
1.2.4	Date/Time Types	12
1.2.5	Boolean Types	16
1.2.6	XML Type	16
1.3	Functions and Operators	17
1.3.1	Logical Operators	17
1.3.2	Comparison Operators	17
1.3.3	Mathematical Functions and Operators	18
1.3.4	String Functions and Operators	23
1.3.5	Pattern Matching String Functions	27
1.3.6	Pattern Matching Using the LIKE Operator	31
1.3.7	Data Type Formatting Functions	31
1.3.8	Date/Time Functions and Operators	45
1.3.9	Sequence Manipulation Functions	61
1.3.10	Conditional Expressions	62
1.3.11	Aggregate Functions	65
1.3.12	Subquery Expressions	76
1.3.13	Identifier Functions	78
1.3.14	Bitwise Functions	81
2	System Catalog Tables	83
2.1	dual	83
2.2	edb_dir	83
2.3	edb_password_history	84
2.4	edb_policy	84
2.5	edb_profile	85
2.6	edb_variable	86
2.7	pg_synonym	86
2.8	product_component_version	87

Database Compatibility for Oracle means that an application runs in an Oracle environment as well as in the EDB Postgres Advanced Server (Advanced Server) environment with minimal or no changes to the application code.

This guide provides reference material about the compatible data types supported by Advanced Server. Reference information about:

- Compatible SQL Language syntax **is** provided **in** the *Database Compatibility **for** [Oracle Developers SQL Guide](#).*.
- Compatible Catalog Views **is** provided **in** the *Database Compatibility **for** Oracle [Developers Catalog Views Guide](#).*.

Developing an application that is compatible with Oracle databases in the Advanced Server requires special attention to which features are used in the construction of the application. For example, developing a compatible application means selecting:

- Data types to define the application's database tables that are compatible with [Oracle databases](#)
- SQL statements that are compatible **with** Oracle SQL
- System **and** built-**in** functions **for** use **in** SQL statements **and** procedural logic that [are compatible with](#) Oracle databases
- Stored Procedure Language (SPL) to create database server-side application logic [for](#) stored procedures, functions, triggers, **and** packages
- System catalog views that are compatible with Oracle's data dictionary

For detailed information about Advanced Server's compatibility features and extended functionality, see the complete library of Advanced Server documentation, available at:

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

THE SQL LANGUAGE

The following sections describe the subset of the Advanced Server SQL language compatible with Oracle databases. The following SQL syntax, data types, and functions work in both EDB Postgres Advanced Server and Oracle.

The Advanced Server documentation set includes syntax and commands for extended functionality (functionality that does not provide database compatibility for Oracle or support Oracle-styled applications) that is not included in this guide.

This section is organized into the following sections:

- General discussion of Advanced Server SQL syntax **and** language elements
- Data types
- Built-**in** functions

1.1 SQL Syntax

This section describes the general syntax of SQL. It forms the foundation for understanding the following chapters that include detail about how the SQL commands are applied to define and modify data.

1.1.1 Lexical Structure

SQL input consists of a sequence of commands. A *command* is composed of a sequence of *tokens*, terminated by a semicolon (;). The end of the input stream also terminates a command. Which tokens are valid depends on the syntax of the particular command.

A token can be a *key word*, an *identifier*, a *quoted identifier*, a *literal* (or *constant*), or a special character symbol. Tokens are normally separated by *whitespace* (space, tab, new line), but need not be if there is no ambiguity (which is generally only the case if a special character is adjacent to some other token type).

Additionally, *comments* can occur in SQL input. They are not tokens -they are effectively equivalent to whitespace.

For example, the following is (syntactically) valid SQL input:

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

This is a sequence of three commands, one per line (although this is not required; more than one command can be on a line, and commands can usually be split across lines).

The SQL syntax is not very consistent regarding what tokens identify commands and which are operands or parameters. The first few tokens are generally the command name, so in the above example we would usually speak of a `SELECT`, an `UPDATE`, and an `INSERT` command. But for instance the `UPDATE` command always requires a `SET` token to appear in a certain position, and this particular variation of `INSERT` also requires a `VALUES` token in order to be complete. The precise syntax rules for each command are described in *Database Compatibility for Oracle Developers SQL Guide*.

1.1.2 Identifiers and Key Words

Tokens such as `SELECT`, `UPDATE`, or `VALUES` in the example above are examples of *key words*, that is, words that have a fixed meaning in the SQL language. The tokens `MY_TABLE` and `A` are examples of *identifiers*. They identify names of tables, columns, or other database objects, depending on the command they are used in. Therefore they are sometimes simply called, “names”. Key words and identifiers have the same *lexical structure*, meaning that one cannot know whether a token is an identifier or a key word without knowing the language.

SQL identifiers and key words must begin with a letter (a–z or A–Z). Subsequent characters in an identifier or key word can be letters, underscores, digits (0–9), dollar signs (\$), or number signs (#).

Identifier and key word names are case insensitive. Therefore

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

can equivalently be written as:

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

A convention often used is to write key words in upper case and names in lower case, e.g.,

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

There is a second kind of identifier: the *delimited identifier* or *quoted identifier*. It is formed by enclosing an arbitrary sequence of characters in double-quotes (“”). A delimited identifier is always an identifier, never a key word. So “select” could be used to refer to a column or table named “select”, whereas an unquoted select would be taken as a key word and would therefore provoke a parse error when used where a table or column name is expected. The example can be written with quoted identifiers like this:

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

Quoted identifiers can contain any character, except the character with the numeric code zero.

To include a double quote, use two double quotes. This allows you to construct table or column names that would otherwise not be possible (such as ones containing spaces or ampersands). The length limitation still applies.

Quoting an identifier also makes it case-sensitive, whereas unquoted names are always folded to lower case. For example, the identifiers `FOO`, `foo`, and `"foo"` are considered the same by Advanced Server, but `"Foo"` and `"FOO"` are different from these three and each other. The folding of unquoted names to lower case is not compatible with Oracle databases. In Oracle syntax, unquoted names are folded to upper case: for example, `foo` is equivalent to `"FOO"` not `"foo"`. If you want to write portable applications you are advised to always quote a particular name or never quote it.

1.1.3 Constants

The kinds of implicitly-typed constants in Advanced Server are *strings* and *numbers*. Constants can also be specified with explicit types, which can enable more accurate representation and more efficient handling by the system. These alternatives are discussed in the following subsections.

String Constants

A *string constant* in SQL is an arbitrary sequence of characters bounded by single quotes ('), for example 'This is a string'. To include a single-quote character within a string constant, write two adjacent single quotes, e.g. 'Dianne' 's horse'. Note that this is not the same as a double-quote character (").

Numeric Constants

Numeric constants are accepted in these general forms:

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

where *digits* is one or more decimal digits (0 through 9). At least one digit must be before or after the decimal point, if one is used. At least one digit must follow the exponent marker (*e*), if one is present. There may not be any spaces or other characters embedded in the constant. Note that any leading plus or minus sign is not actually considered part of the constant; it is an operator applied to the constant.

These are some examples of valid numeric constants:

```
42
3.5
4.
.001
5e2
1.925e-3
```

A numeric constant that contains neither a decimal point nor an exponent is initially presumed to be type **INTEGER** if its value fits in type **INTEGER** (32 bits); otherwise it is presumed to be type **BIGINT** if its value fits in type **BIGINT** (64 bits); otherwise it is taken to be type **NUMBER**. Constants that contain decimal points and/or exponents are always initially presumed to be type **NUMBER**.

The initially assigned data type of a numeric constant is just a starting point for the type resolution algorithms. In most cases the constant will be automatically coerced to the most appropriate type depending on context. When necessary, you can force a numeric value to be interpreted as a specific data type by casting it as described in the following section.

Constants of Other Types

CAST

A constant of an arbitrary type can be entered using the following notation:

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

The string constant's text is passed to the input conversion routine for the type called `type`. The result is a constant of the indicated type. The explicit type cast may be omitted if there is no ambiguity as to the type the constant must be (for example, when it is assigned directly to a table column), in which case it is automatically coerced.

CAST can also be used to specify runtime type conversions of arbitrary expressions.

CAST (MULTISET)

MULTISET is an extension to CAST that converts subquery results into a nested table type. The synopsis is:

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

Where `subquery` is a query returning one or more rows and `datatype` is a nested table type.

CAST(MULTISET) is used to store a collection of data in a table.

Example

The following example demonstrates using MULTISET:

```
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
```

(continues on next page)

(continued from previous page)

```

-----+-----
Morgan   | {Code,Teach}
Kolk     | {Code}
Scott    | {}
(3 rows)

```

1.1.4 Comments

A comment is an arbitrary sequence of characters beginning with double dashes and extending to the end of the line, e.g.:

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

Alternatively, C-style block comments can be used:

```

/* multiline comment
*  block
*  /

```

where the comment begins with `/*` and extends to the matching occurrence of `*/`.

1.2 Data Types

The following table shows the built-in general-purpose data types:

Name	Alias	Description
<i>BLOB</i>	<i>LONG RAW,</i> <i>RAW(n),</i> <i>BYTEA</i>	Binary data
<i>BOOLEAN</i>		Logical Boolean (true/false)
<i>CHAR [(n)]</i>	<i>CHARACTER [(n)]</i>	Fixed-length character string of <i>n</i> characters
<i>CLOB</i>	<i>LONG,</i> <i>LONG VAR-</i> <i>CHAR</i>	Long character string
<i>DATE</i>	<i>TIMESTAMP</i>	Date and time to the second
<i>DOUBLE PRECISION</i>	<i>FLOAT,</i> <i>FLOAT(25) –</i> <i>FLOAT(53)</i>	Double precision floating-point number
<i>INTEGER</i>	<i>INT, BI-</i> <i>NARY_INTEGER,</i> <i>PLS_INTEGER</i>	Signed four-byte integer

continues on next page

Table 1 – continued from previous page

Name	Alias	Description
<i>NUMBER</i>	<i>DEC</i> , <i>DECIMAL</i> , <i>NUMERIC</i>	Exact numeric with optional decimal places
<i>NUMBER(p [, s])</i>	<i>DEC(p [, s])</i> , <i>DECIMAL(p [, s])</i> , <i>NUMERIC(p [, s])</i>	Exact numeric of maximum precision, <i>p</i> , and optional scale, <i>s</i>
<i>REAL</i>	<i>FLOAT(1) – FLOAT(24)</i>	Single precision floating-point number
<i>TIMESTAMP [(p)]</i>		Date and time with optional, fractional second precision, <i>p</i>
<i>TIMESTAMP [(p)] WITH TIME ZONE</i>		Date and time with optional, fractional second precision, <i>p</i> , and with time zone
<i>VARCHAR2(n)</i>	<i>CHAR VARYING(n)</i> , <i>CHARACTER VARYING(n)</i> , <i>VARCHAR(n)</i>	Variable-length character string with a maximum length of <i>n</i> characters
<i>XMLTYPE</i>		XML data

1.2.1 Numeric Types

Numeric types consist of four-byte integers, four-byte and eight-byte floating-point numbers, and fixed-precision decimals. The following table lists the available types:

Name	Storage Size	Description	Range
<i>BINARY_INTEGER</i>	4 bytes	Signed integer, Alias for <i>INTEGER</i>	-2,147,483,648 to +2,147,483,647
<i>DOUBLE PRECISION</i>	8 bytes	Variable-precision, inexact	15 decimal digits precision

continues on next page

Table 2 – continued from previous page

Name	Storage Size	Description	Range
<i>INTEGER</i>	4 bytes	Usual choice for integer	-2,147,483,648 to +2,147,483,647
<i>NUMBER</i>	Variable	User-specified precision, exact	Up to 1000 digits of precision
<i>NUMBER(p [, s])</i>	Variable	Exact numeric of maximum precision, <i>p</i> , and optional scale, <i>s</i>	Up to 1000 digits of precision
<i>PLS_INTEGER</i>	4 bytes	Signed integer, Alias for <i>INTEGER</i>	-2,147,483,648 to +2,147,483,647
<i>REAL</i>	4 bytes	Variable-precision, inexact	6 decimal digits precision
<i>ROWID</i>	8 bytes	Signed 8 bit integer.	-9223372036854775808 to 9223372036854775807

The following sections describe the types in detail.

Integer Types

The *BINARY_INTEGER*, *INTEGER*, *PLS_INTEGER*, and *ROWID* types store whole numbers (without fractional components) as specified in table *Numeric Types*. Attempts to store values outside of the allowed range will result in an error.

Arbitrary Precision Numbers

The type, *NUMBER*, can store practically an unlimited number of digits of precision and perform calculations exactly. It is especially recommended for storing monetary amounts and other quantities where exactness is required. However, the *NUMBER* type is very slow compared to the floating-point types described in the next section.

In what follows we use these terms: The **scale** of a *NUMBER* is the count of decimal digits in the fractional part, to the right of the decimal point. The **precision** of a *NUMBER* is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. So the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero.

Both the precision and the scale of the *NUMBER* type can be configured. To declare a column of type *NUMBER* use the syntax

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

The precision must be positive, the scale zero or positive. Alternatively,

NUMBER(precision)

selects a scale of 0. Specifying **NUMBER** without any precision or scale creates a column in which numeric values of any precision and scale can be stored, up to the implementation limit on precision. A column of this kind will not coerce input values to any particular scale, whereas **NUMBER** columns with a declared scale will coerce input values to that scale. (The SQL standard requires a default scale of 0, i.e., coercion to integer precision. For maximum portability, it is best to specify the precision and scale explicitly.)

If the precision or scale of a value is greater than the declared precision or scale of a column, the system will attempt to round the value. If the value cannot be rounded so as to satisfy the declared limits, an error is raised.

Floating-Point Types

The data types **REAL** and **DOUBLE PRECISION** are *inexact*, variable-precision numeric types. In practice, these types are usually implementations of IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and printing back out a value may show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire branch of mathematics and computer science and will not be discussed further here, except for the following points:

If you require exact storage and calculations (such as for monetary amounts), use the **NUMBER** type instead.

If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), you should evaluate the implementation carefully.

Comparing two floating-point values for equality may or may not work as expected.

On most platforms, the **REAL** type has a range of at least $1\text{E}-37$ to $1\text{E}+37$ with a precision of at least 6 decimal digits. The **DOUBLE PRECISION** type typically has a range of around $1\text{E}-307$ to $1\text{E}+308$ with a precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding may take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow error.

Advanced Server also supports the SQL standard notations **FLOAT** and **FLOAT(p)** for specifying inexact numeric types. Here, *p* specifies the minimum acceptable precision in binary digits. Advanced Server accepts **FLOAT(1)** to **FLOAT(24)** as selecting the **REAL** type, while **FLOAT(25)** to **FLOAT(53)** as selecting **DOUBLE PRECISION**. Values of *p* outside the allowed range draw an error. **FLOAT** with no precision specified is taken to mean **DOUBLE PRECISION**.

1.2.2 Character Types

The following table lists the general-purpose character types available in Advanced Server:

Name	Description
<i>CHAR(n)</i>	Fixed-length character string, blank-padded to the size specified by <i>n</i>
<i>CLOB</i>	Large variable-length up to 1 GB
<i>LONG</i>	Variable unlimited length.
<i>NVARCHAR(n)</i>	Variable-length national character string, with limit

continues on next page

Table 3 – continued from previous page

Name	Description
<i>NVARCHAR2(n)</i>	Variable-length national character string, with limit
<i>STRING</i>	Alias for <i>VARCHAR2</i>
<i>VARCHAR(n)</i>	Variable-length character string, with limit (considered deprecated, but supported for compatibility)
<i>VARCHAR2(n)</i>	Variable-length character string, with limit

Where *n* is a positive integer; these types can store strings up to *n* characters in length. An attempt to assign a value that exceeds the length of *n* will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length.

The storage requirement for data of these types is the actual string plus 1 byte if the string is less than 127 bytes, or 4 bytes if the string is 127 bytes or greater. In the case of *CHAR*, the padding also requires storage. Long strings are compressed by the system automatically, so the physical requirement on disk may be less. Long values are stored in background tables so they do not interfere with rapid access to the shorter column values.

The database character set determines the character set used to store textual values.

CHAR

If you do not specify a value for *n*, *n* will default to 1. If the string to be assigned is shorter than *n*, values of type *CHAR* will be space-padded to the specified width (*n*), and will be stored and displayed that way.

Padding spaces are treated as semantically insignificant. That is, trailing spaces are disregarded when comparing two values of type *CHAR*, and they will be removed when converting a *CHAR* value to one of the other string types.

If you explicitly cast an over-length value to a *CHAR(n)* type, the value will be truncated to *n* characters without raising an error (as specified by the SQL standard).

<i>VARCHAR</i> , <i>VARCHAR2</i> , <i>NVARCHAR</i> and <i>NVARCHAR2</i>

If the string to be assigned is shorter than *n*, values of type *VARCHAR*, *VARCHAR2*, *NVARCHAR* and *NVARCHAR2* will store the shorter string without padding.

Note: The trailing spaces *are* semantically significant in *VARCHAR* values.

If you explicitly cast a value to a *VARCHAR* type, an over-length value will be truncated to *n* characters without raising an error (as specified by the SQL standard).

CLOB

You can store a large character string in a *CLOB* type. *CLOB* is semantically equivalent to *VARCHAR2* except no length limit is specified. Generally, you should use a *CLOB* type if the maximum string length is not known.

The longest possible character string that can be stored in a *CLOB* type is about 1 GB.

Note: The *CLOB* data type is actually a *DOMAIN* based on the PostgreSQL *TEXT* data type. For information on a *DOMAIN*, see the PostgreSQL core documentation at <https://www.postgresql.org/docs/current/static/sql-createdomain.html>

Thus, usage of the *CLOB* type is limited by what can be done for *TEXT* such as a maximum size of approximately 1 GB.

For usage of larger amounts of data, instead of using the CLOB data type, use the PostgreSQL *Large Objects* feature that relies on the `pg_largeobject` system catalog. For information on large objects, see the PostgreSQL core documentation at:

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

1.2.3 Binary Data

The following table shows data types that allow the storage of binary strings:

Name	Storage Size	Description
<i>BINARY</i>	The length of the binary string.	Fixed-length binary string, with a length between 1 and 8300.
<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.	Variable-length binary string, with a maximum size of 1 GB.
<i>VARBINARY</i>	The length of the binary string	Variable-length binary string, with a length between 1 and 8300.

A binary string is a sequence of octets (or bytes). Binary strings are distinguished from character strings by two characteristics: First, binary strings specifically allow storing octets of value zero and other “non-printable” octets (defined as octets outside the range 32 to 126). Second, operations on binary strings process the actual bytes, whereas the encoding and processing of character strings depends on locale settings.

1.2.4 Date/Time Types

The following discussion of the date/time types assumes that the configuration parameter, `edb_redwood_date`, has been set to `TRUE` whenever a table is created or altered.

Advanced Server supports the date/time types shown in the following table:

Name	Storage Size	Description	Low Value	High Value	Resolution
<i>DATE</i>	8 bytes	Date and time	4713 BC	5874897 AD	1 second
<i>INTERVAL DAY TO SECOND [(p)]</i>	12 bytes	Period of time	- 178000000 years	178000000 years	1 microsecond / 14 digits

continues on next page

Table 5 – continued from previous page

Name	Storage Size	Description	Low Value	High Value	Resolution
<i>INTERVAL YEAR TO MONTH</i>	12 bytes	Period of time	- 178000000 years	178000000 years	1 microsecond / 14 digits
<i>TIMESTAMP [(p)]</i>	8 bytes	Date and time	4713 BC	5874897 AD	1 microsecond
<i>TIMESTAMP [(p)] WITH TIME ZONE</i>	8 bytes	Date and time with time zone	4713 BC	5874897 AD	1 microsecond

When DATE appears as the data type of a column in the data definition language (DDL) commands, CREATE TABLE or ALTER TABLE, it is translated to TIMESTAMP at the time the table definition is stored in the database. Thus, a time component will also be stored in the column along with the date.

When DATE appears as a data type of a variable in an SPL declaration section, or the data type of a formal parameter in an SPL procedure or an SPL function, or the return type of an SPL function, it is always translated to TIMESTAMP and thus can handle a time component if present.

TIMESTAMP accepts an optional precision value p which specifies the number of fractional digits retained in the seconds field. The allowed range of p is from 0 to 6 with the default being 6.

When TIMESTAMP values are stored as double precision floating-point numbers (currently the default), the effective limit of precision may be less than 6. TIMESTAMP values are stored as seconds before or after midnight 2000-01-01. Microsecond precision is achieved for dates within a few years of 2000-01-01, but the precision degrades for dates further away. When TIMESTAMP values are stored as eight-byte integers (a compile-time option), microsecond precision is available over the full range of values. However eight-byte integer timestamps have a more limited range of dates than shown above: from 4713 BC up to 294276 AD.

TIMESTAMP (p) WITH TIME ZONE is similar to TIMESTAMP (p), but includes the time zone as well.

INTERVAL Types

INTERVAL values specify a period of time. Values of INTERVAL type are composed of fields that describe the value of the data. The following table lists the fields allowed in an INTERVAL type:

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(p) where 9(p) is the precision of fractional seconds

The fields must be presented in descending order – from YEARS to MONTHS, and from DAYS to HOURS, MINUTES and then SECONDS.

Advanced Server supports two INTERVAL types compatible with Oracle databases.

The first variation supported by Advanced Server is INTERVAL DAY TO SECOND [(p)]. INTERVAL DAY TO SECOND [(p)] stores a time interval in days, hours, minutes and seconds.

p specifies the precision of the second field.

Advanced Server interprets the value:

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

as 1 day, 2 hours, 34 minutes, 5 seconds and 678 thousandths of a second.

Advanced Server interprets the value:

```
INTERVAL '1 23' DAY TO HOUR
```

as 1 day and 23 hours.

Advanced Server interprets the value:

```
INTERVAL '2:34' HOUR TO MINUTE
```

as 2 hours and 34 minutes.

Advanced Server interprets the value:

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

as 2 hours, 34 minutes, 56 seconds and 13 thousandths of a second. Note that the fractional second is rounded up to 13 because of the specified precision.

The second variation supported by Advanced Server that is compatible with Oracle databases is `INTERVAL YEAR TO MONTH`. This variation stores a time interval in years and months.

Advanced Server interprets the value:

```
INTERVAL '12-3' YEAR TO MONTH
```

as 12 years and 3 months.

Advanced Server interprets the value:

```
INTERVAL '456' YEAR(2)
```

as 12 years and 3 months.

Advanced Server interprets the value:

```
INTERVAL '300' MONTH
```

as 25 years.

Date/Time Input

Date and time input is accepted in ISO 8601 SQL-compatible format, the Oracle default `dd-MON-yy` format, as well as a number of other formats provided that there is no ambiguity as to which component is the year, month, and day. However, use of the `TO_DATE` function is strongly recommended to avoid ambiguities.

Any date or time literal input needs to be enclosed in single quotes, like text strings. The following SQL standard syntax is also accepted:

<pre>type 'value'</pre>

`type` is either `DATE` or `TIMESTAMP`.

`value` is a date/time text string.

Dates

The following block shows some possible input formats for dates, all of which equate to January 8, 1999.

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

The date values can be assigned to a `DATE` or `TIMESTAMP` column or variable. The hour, minute, and seconds fields will be set to zero if the date value is not appended with a time value.

Times

Some examples of the time component of a date or time stamp are shown in the following table:

Example	Description
<i>04:05:06.789</i>	ISO 8601
<i>04:05:06</i>	ISO 8601
<i>04:05</i>	ISO 8601
<i>040506</i>	ISO 8601
<i>04:05 AM</i>	Same as 04:05; AM does not affect value
<i>04:05 PM</i>	Same as 16:05; input hour must be ≤ 12

Time Stamps

Valid input for time stamps consists of a concatenation of a date and a time. The date portion of the time stamp can be formatted according to any of the examples shown in the block under section **Dates**. The time portion of the time stamp can be formatted according to any of examples shown in table under section **Times**.

The following is an example of a time stamp which follows the Oracle default format.

```
08-JAN-99 04:05:06
```

The following is an example of a time stamp which follows the ISO 8601 standard.

```
1999-01-08 04:05:06
```

Date/Time Output

The default output format of the date/time types will be either (dd-MON-yy) referred to as the *Redwood date style*, compatible with Oracle databases, or (yyyy-mm-dd) referred to as the ISO 8601 format, depending upon the application interface to the database. Applications that use JDBC such as SQL Interactive always present the date in ISO 8601 form. Other applications such as PSQL present the date in Redwood form.

The following table shows examples of the output formats for the two styles, Redwood and ISO 8601:

Description	Example
Redwood style	<i>31-DEC-05 07:37:16</i>
ISO 8601/SQL standard	<i>1997-12-17 07:37:16</i>

Internals

Advanced Server uses Julian dates for all date/time calculations. Julian dates correctly predict or calculate any date after 4713 BC based on the assumption that the length of the year is 365.2425 days.

1.2.5 Boolean Types

Advanced Server provides the standard SQL type `BOOLEAN`. `BOOLEAN` can have one of only two states: `TRUE` or `FALSE`. A third state, `UNKNOWN`, is represented by the SQL `NULL` value.

Name	Storage Size	Description
<i>BOOLEAN</i>	1 byte	Logical Boolean (true/false)

The valid literal value for representing the true state is `TRUE`. The valid literal for representing the false state is `FALSE`.

1.2.6 XML Type

The `XMLTYPE` data type is used to store XML data. Its advantage over storing XML data in a character field is that it checks the input values for well-formedness, and there are support functions to perform type-safe operations on it.

The XML type can store well-formed “documents”, as defined by the XML standard, as well as “content” fragments, which are defined by the production `XMLDecl? content` in the XML standard. Roughly, this means that content fragments can have more than one top-level element or character node.

!!! Note * Oracle does not support the storage of content fragments in `XMLTYPE` columns.

The following example shows the creation and insertion of a row into a table with an `XMLTYPE` column.

```
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;
      *
```

(continues on next page)

(continued from previous page)

```
-----
<book><title>Manual</title><chapter>...</chapter></book>
(1 row)
```

1.3 Functions and Operators

Advanced Server provides a large number of functions and operators for the built-in data types.

1.3.1 Logical Operators

The usual logical operators are available: AND, OR, NOT

SQL uses a three-valued Boolean logic where the null value represents “unknown”. Observe the following truth tables:

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

The operators AND and OR are commutative, that is, you can switch the left and right operand without affecting the result.

1.3.2 Comparison Operators

The usual comparison operators are shown in the following table:

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
=	Equal
<>	Not equal

continues on next page

Table 12 – continued from previous page

Operator	Description
<code>!=</code>	Not equal

Comparison operators are available for all data types where this makes sense. All comparison operators are binary operators that return values of type `BOOLEAN`; expressions like `1 < 2 < 3` are not valid (because there is no `<` operator to compare a Boolean value with 3).

In addition to the comparison operators, the special **BETWEEN** construct is available.

`a BETWEEN x AND y`

is equivalent to

`a >= x AND a <= y`

Similarly,

`a NOT BETWEEN x AND y`

is equivalent to

`a < x OR a > y`

There is no difference between the two respective forms apart from the CPU cycles required to rewrite the first one into the second one internally.

To check whether a value is or is not null, use the constructs

`expression IS NULL`

`expression IS NOT NULL`

Do not write `expression = NULL` because `NULL` is not “equal to” `NULL`. (The null value represents an unknown value, and it is not known whether two unknown values are equal.) This behavior conforms to the SQL standard.

Some applications may expect that `expression = NULL` returns true if `expression` evaluates to the null value. It is highly recommended that these applications be modified to comply with the SQL standard.

1.3.3 Mathematical Functions and Operators

Mathematical operators are provided for many Advanced Server types. For types without common mathematical conventions for all possible permutations (e.g., date/time types) the actual behavior is described in subsequent sections.

The following table shows the available mathematical operators:

Operator	Description	Example	Result
<code>+</code>	Addition	<code>2 + 3</code>	<code>5</code>
<code>-</code>	Subtraction	<code>2 - 3</code>	<code>-1</code>
<code>*</code>	Multiplication	<code>2 * 3</code>	<code>6</code>
<code>/</code>	Division (See the following note)	<code>4 / 2</code>	<code>2</code>
<code>**</code>	Exponentiation Operator	<code>2 ** 3</code>	<code>8</code>

!!! Note * If the `db_dialect` configuration parameter in the `postgresql.conf` file is set to `redwood`, then division of a pair of `INTEGER` data types does not result in a truncated value. Any fractional result is retained as shown by the following example:

```

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)

```

This behavior is compatible with Oracle databases where there is no native `INTEGER` data type, and any `INTEGER` data type specification is internally converted to `NUMBER(38)`, which results in retaining any fractional result.

If the `db_dialect` configuration parameter is set to `postgres`, then division of a pair of `INTEGER` data types results in a truncated value as shown by the following example:

```

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)

```

This behavior is compatible with PostgreSQL databases where division involving any pair of `INTEGER`, `SMALLINT`, or `BIGINT` data types results in truncation of the result. The same truncated result is returned by Advanced Server when `db_dialect` is set to `postgres` as shown in the previous example.

Note however, that even when `db_dialect` is set to `redwood`, only division with a pair of `INTEGER` data types results in no truncation of the result. Division that includes only `SMALLINT` or `BIGINT` data types, with or without an `INTEGER` data type, does result in truncation in the PostgreSQL fashion without retaining the fractional portion as shown by the following where `INTEGER` and `SMALLINT` are involved in the division:

```

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)

```

The following table shows the available mathematical functions. Many of these functions are provided in multiple forms with different argument types. Except where noted, any given form of a function returns the same data type as its argument. The functions working with `DOUBLE PRECISION` data are mostly implemented on top of the host system's C library; accuracy and behavior in boundary cases may therefore vary depending on the host system.

Function	Return Type	Description	Example	Result
<i>ABS(x)</i>	Same as <i>x</i>	Absolute value	<i>ABS(-17.4)</i>	<i>17.4</i>
<i>CEIL(DOUBLE PRECISION or NUMBER)</i>	Same as input	Smallest integer not less than argument	<i>CEIL(-42.8)</i>	<i>-42</i>
<i>EXP(DOUBLE PRECISION or NUMBER)</i>	Same as input	Exponential	<i>EXP(1.0)</i>	<i>2.7182818284590452</i>
<i>FLOOR(DOUBLE PRECISION or NUMBER)</i>	Same as input	Largest integer not greater than argument	<i>FLOOR(-42.8)</i>	<i>43</i>
<i>LN(DOUBLE PRECISION or NUMBER)</i>	Same as input	Natural logarithm	<i>LN(2.0)</i>	<i>0.6931471805599453</i>
<i>LOG(b NUMBER, x NUMBER)</i>	<i>NUMBER</i>	Logarithm to base <i>b</i>	<i>LOG(2.0, 64.0)</i>	<i>6.0000000000000000</i>
<i>MOD(y, x)</i>	Same as argument types	Remainder of <i>y/x</i>	<i>MOD(9, 4)</i>	<i>1</i>
<i>NVL(x, y)</i>	Same as argument types; where both arguments are of the same data type	If <i>x</i> is null, then <i>NVL</i> returns <i>y</i>	<i>NVL(9, 0)</i>	<i>9</i>

continues on next page

Table 14 – continued from previous page

Function	Return Type	Description	Example	Result
<i>POWER(a DOU- BLE PRE- CI- SION, b DOU- BLE PRE- CI- SION)</i>	<i>DOUBLE PRECI- SION</i>	<i>a</i> raised to the power of <i>b</i>	<i>POWER(9.0, 3.0)</i>	729.000000000000000000
<i>POWER(a NUM- BER, b NUM- BER)</i>	<i>NUMBER</i>	<i>a</i> raised to the power of <i>b</i>	<i>POWER(9.0, 3.0)</i>	729.000000000000000000
<i>ROUND(DOUBLE PRECISION or NUMBER)</i>	Same as input	Round to nearest integer	<i>ROUND(42.4)</i>	42
<i>ROUND(v NUM- BER, s IN- TE- GER)</i>	<i>NUMBER</i>	Round to <i>s</i> decimal places	<i>ROUND(42.4382, 2)</i>	42.44
<i>SIGN(DOUBLE PRECISION or NUMBER)</i>	Same as input	Sign of the argument (-1, 0, +1)	<i>SIGN(-8.4)</i>	-1
<i>SQRT(DOUBLE PRECISION or NUMBER)</i>	Same as input	Square root	<i>SQRT(2.0)</i>	1.414213562373095
<i>TRUNC(DOUBLE PRECISION or NUMBER)</i>	Same as input	Truncate toward zero	<i>TRUNC(42.8)</i>	42

continues on next page

Table 14 – continued from previous page

Function	Return Type	Description	Example	Result
<i>TRUNC(v NUM- BER, s IN- TE- GER)</i>	<i>NUMBER</i>	Truncate to <i>s</i> decimal places	<i>TRUNC(42.4382, 2)</i>	<i>42.43</i>
<i>WIDTH_BUCKET(op NUM- BER, b1 NUM- BER, b2 NUM- BER, count IN- TE- GER)</i>	<i>INTEGER</i>	Return the bucket to which <i>op</i> would be assigned in an equidepth histogram with <i>count</i> buckets, in the range <i>b1</i> to <i>b2</i>	<i>WIDTH_BUCKET(5.35, 0.024, 10.06, 5)</i>	<i>3</i>

The following table shows the available trigonometric functions. All trigonometric functions take arguments and return values of type `DOUBLE PRECISION`.

Function	Description
<i>ACOS(x)</i>	Inverse cosine
<i>ASIN(x)</i>	Inverse sine
<i>ATAN(x)</i>	Inverse tangent
<i>ATAN2(x, y)</i>	Inverse tangent of <i>x/y</i>
<i>COS(x)</i>	Cosine
<i>SIN(x)</i>	Sine
<i>TAN(x)</i>	Tangent

1.3.4 String Functions and Operators

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of the types CHAR, VARCHAR2, and CLOB. Unless otherwise noted, all of the functions listed below work on all of these types, but be wary of potential effects of automatic padding when using the CHAR type. Generally, the functions described here also work on data of non-string types by converting that data to a string representation first.

Additional string manipulation functions are available and are listed in the following table. Some of them are used internally to implement the SQL-standard string functions listed in the above table.

Function	Return Type	Description	Example	Result
<i>ASCII(string)</i>	<i>INTEGER</i>	ASCII code of the first byte of the argument	<i>ASCII('x')</i>	<i>120</i>
<i>CHR(INTEGER)</i>	<i>CLOB</i>	Character with the given ASCII code	<i>CHR(65)</i>	<i>A</i>
DE-CODE(expr, expr1a, expr1b [, expr2a, expr2b]... [, default])	Same as argument types of expr1b, expr2b,..., default	Finds first match of <i>expr</i> with <i>expr1a</i> , <i>expr2a</i> , etc. When match found, returns corresponding parameter pair, <i>expr1b</i> , <i>expr2b</i> , etc. If no match found, returns <i>default</i> . If no match found and <i>default</i> not specified, returns null.	DE-CODE(3, 1,'One', 2,'Two', 3,'Three', 'Not found')	<i>Three</i>
<i>INITCAP(string)</i>	<i>CLOB</i>	Convert the first letter of each word to uppercase and the rest to lowercase. Words are sequences of alphanumeric characters separated by non-alphanumeric characters.	<i>INITCAP('hi THOMAS')</i>	<i>Hi Thomas</i>
<i>LENGTH</i>	<i>INTEGER</i>	Returns the number of characters in a string value.	<i>LENGTH('Côte d'Azur')</i>	<i>11</i>
<i>LENGTHC</i>	<i>INTEGER</i>	This function is identical in functionality to <i>LENGTH</i> ; the function name is supported for compatibility.	<i>LENGTHC('Côte d'Azur')</i>	<i>11</i>
<i>LENGTH2</i>	<i>INTEGER</i>	This function is identical in functionality to <i>LENGTH</i> ; the function name is supported for compatibility.	<i>LENGTH2('Côte d'Azur')</i>	<i>11</i>
<i>LENGTH4</i>	<i>INTEGER</i>	This function is identical in functionality to <i>LENGTH</i> ; the function name is supported for compatibility.	<i>LENGTH4('Côte d'Azur')</i>	<i>11</i>
<i>LENGTHB</i>	<i>INTEGER</i>	Returns the number of bytes required to hold the given value.	<i>LENGTHB('Côte d'Azur')</i>	<i>12</i>

continues on next page

Table 16 – continued from previous page

Function	Return Type	Description	Example	Result
<i>LPAD(string, length INTEGER [, fill])</i>	<i>CLOB</i>	Fill up <i>string</i> to size, <i>length</i> by prepending the characters, <i>fill</i> (a space by default). If <i>string</i> is already longer than <i>length</i> then it is truncated (on the right).	<i>LPAD('hi', 5, 'xy')</i>	<i>xyxhi</i>
<i>REPLACE(string, search_string [, replace_string])</i>	<i>CLOB</i>	Replaces one value in a string with another. If you do not specify a value for <i>replace_string</i> , the <i>search_string</i> value when found, is removed.	<i>REPLACE('GEORGE', 'GE', 'EG')</i>	<i>EGOREG</i>
<i>RPAD(string, length INTEGER [, fill])</i>	<i>CLOB</i>	Fill up <i>string</i> to size, <i>length</i> by appending the characters, <i>fill</i> (a space by default). If <i>string</i> is already longer than <i>length</i> then it is truncated.	<i>RPAD('hi', 5, 'xy')</i>	<i>hixyx</i>
<i>TRANSLATE(string, from, to)</i>	<i>CLOB</i>	Any character in <i>string</i> that matches a character in the <i>from</i> set is replaced by the corresponding character in the <i>to</i> set.	<i>TRANSLATE('12345', '14', 'ax')</i>	<i>a23x5</i>

Truncation of String Text Resulting from Concatenation with NULL

!!! Note * This section describes a functionality that is not compatible with Oracle databases, which may lead to some inconsistency when converting data from Oracle to Advanced Server.

For Advanced Server, when a column value is NULL, the concatenation of the column with a text string may result in either of the following:

- Return of the text string
- Disappearance of the text string (that is, a null result)

The result is dependent upon the data type of the NULL column and the way in which the concatenation is done.

If one uses the string concatenation operator '||', then the types that have implicit coercion to text as listed in Table Data Types with Implicit Coercion to Text will not truncate the string if one of the input parameters is NULL, whereas

for other types it will truncate the string unless the explicit type cast is used (that is, `::text`). Also, to see the consistent behavior in the presence of nulls, one can use the `CONCAT` function.

The following query lists the data types that have implicit coercion to text:

```
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';
```

The result of the query is listed in the following table:

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>
<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>

For information on the column output, see the `pg_cast` system catalog in the PostgreSQL core documentation available at:

<https://www.postgresql.org/docs/current/static/catalog-pg-cast.html>

So for example, data type `UUID` is not in this list and therefore does not have the implicit coercion to text. As a result,

certain concatenation attempts with a NULL UUID column results in a truncated text result.

The following table is created for this example with a single row with all NULL column values.

```
CREATE TABLE null_concat_types (  
  *   boolean_type    BOOLEAN,  
  *   uuid_type       UUID,  
  *   char_type       CHARACTER  
);  
  
INSERT INTO null_concat_types VALUES (NULL, NULL, NULL);
```

Columns `boolean_type` and `char_type` have the implicit coercion to text while column `uuid_type` does not.

Thus, string concatenation with the concatenation operator `'||'` against columns `boolean_type` or `char_type` results in the following:

```
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)
```

But concatenation with column `uuid_type` results in the loss of the `x=` string:

```
SELECT 'x=' || uuid_type || 'y' FROM null_concat_types;  
  * ?column?  
-----  
  y  
(1 row)
```

However, using explicit casting with `::text` prevents the loss of the `x=` string:

```
SELECT 'x=' || uuid_type::text || 'y' FROM null_concat_types;  
  * ?column?  
-----  
  x=y  
(1 row)
```

Using the `CONCAT` function also preserves the `x=` string:

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

Thus, depending upon the data type of a NULL column, explicit casting or the `CONCAT` function should be used to avoid loss of some text string.

1.3.5 Pattern Matching String Functions

Advanced Server offers support for the REGEXP_COUNT, REGEXP_INSTR and REGEXP_SUBSTR functions. These functions search a string for a pattern specified by a regular expression, and return information about occurrences of the pattern within the string. The pattern should be a POSIX-style regular expression; for more information about forming a POSIX-style regular expression, please refer to the core documentation at:

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

REGEXP_COUNT

REGEXP_COUNT searches a string for a regular expression, and returns a count of the times that the regular expression occurs. The signature is:

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

Parameters

srcstr

srcstr specifies the string to search.

pattern

pattern specifies the regular expression for which REGEXP_COUNT will search.

position

position is an integer value that indicates the position in the source string at which REGEXP_COUNT will begin searching. The default value is 1.

modifier

modifier specifies values that control the pattern matching behavior. The default value is NULL. For a complete list of the modifiers supported by Advanced Server, see the PostgreSQL core documentation available at:

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

Example

In the following simple example, REGEXP_COUNT returns a count of the number of times the letter i is used in the character string 'reinitializing':

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

In the first example, the command instructs REGEXP_COUNT begins counting in the first position; if we modify the command to start the count on the 6th position:

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

REGEXP_COUNT returns 3; the count now excludes any occurrences of the letter i that occur before the 6th position.

REGEXP_INSTR

REGEXP_INSTR searches a string for a POSIX-style regular expression. This function returns the position within the string where the match was located. The signature is:

```
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,
)
```

Parameters:

srcstr

srcstr specifies the string to search.

pattern

pattern specifies the regular expression for which REGEXP_INSTR will search.

position

position specifies an integer value that indicates the start position in a source string. The default value is 1.

occurrence

occurrence specifies which match is returned if more than one occurrence of the pattern occurs in the string that is searched. The default value is 1.

returnparam

returnparam is an integer value that specifies the location within the string that REGEXP_INSTR should return. The default value is 0. Specify:

- ``0`` to return the location within the string of the first character that matches `pattern``.
- A value greater than ``0`` to return the position of the first character following the end of the `pattern``.

modifier

modifier specifies values that control the pattern matching behavior. The default value is NULL. For a complete list of the modifiers supported by Advanced Server, see the PostgreSQL core documentation available at:

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

subexpression

subexpression is an integer value that identifies the portion of the **pattern** that will be returned by **REGEXP_INSTR**. The default value of **subexpression** is 0.

If you specify a value for **subexpression**, you must include one (or more) set of parentheses in the **pattern** that isolate a portion of the value being searched for. The value specified by **subexpression** indicates which set of parentheses should be returned; for example, if **subexpression** is 2, **REGEXP_INSTR** will return the position of the second set of parentheses.

Example

In the following simple example, **REGEXP_INSTR** searches a string that contains the a phone number for the first occurrence of a pattern that contains three consecutive digits:

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

The command instructs **REGEXP_INSTR** to return the position of the first occurrence. If we modify the command to return the start of the second occurrence of three consecutive digits:

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

REGEXP_INSTR returns 5; the second occurrence of three consecutive digits begins in the 5th position.

REGEXP_SUBSTR

The **REGEXP_SUBSTR** function searches a string for a pattern specified by a POSIX compliant regular expression. **REGEXP_SUBSTR** returns the string that matches the pattern specified in the call to the function. The signature of the function is:

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

Parameters:

srcstr

srcstr specifies the string to search.

pattern

pattern specifies the regular expression for which REGEXP_SUBSTR will search.

position

position specifies an integer value that indicates the start position in a source string. The default value is 1.

occurrence

occurrence specifies which match is returned if more than one occurrence of the pattern occurs in the string that is searched. The default value is 1.

modifier

modifier specifies values that control the pattern matching behavior. The default value is NULL. For a complete list of the modifiers supported by Advanced Server, see the PostgreSQL core documentation available at:

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

subexpression

subexpression is an integer value that identifies the portion of the **pattern** that will be returned by REGEXP_SUBSTR. The default value of **subexpression** is 0.

If you specify a value for **subexpression**, you must include one (or more) set of parentheses in the **pattern** that isolate a portion of the value being searched for. The value specified by **subexpression** indicates which set of parentheses should be returned; for example, if **subexpression** is 2, REGEXP_SUBSTR will return the value contained within the second set of parentheses.

Example

In the following simple example, REGEXP_SUBSTR searches a string that contains a phone number for the first set of three consecutive digits:

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

It locates the first occurrence of three digits and returns the string (800); if we modify the command to check for the second occurrence of three consecutive digits:

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

REGEXP_SUBSTR returns 555, the contents of the second substring.

1.3.6 Pattern Matching Using the LIKE Operator

Advanced Server provides pattern matching using the traditional SQL LIKE operator. The syntax for the LIKE operator is as follows.

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

Every **pattern** defines a set of strings. The LIKE expression returns TRUE if **string** is contained in the set of strings represented by **pattern**. As expected, the NOT LIKE expression returns FALSE if LIKE returns TRUE, and vice versa. An equivalent expression is NOT (string LIKE pattern).

If **pattern** does not contain percent signs or underscore, then the pattern only represents the string itself; in that case LIKE acts like the equals operator. An underscore (_) in **pattern** stands for (matches) any single character; a percent sign (%) matches any string of zero or more characters.

Some examples:

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

LIKE pattern matches always cover the entire string. To match a pattern anywhere within a string, the pattern must therefore start and end with a percent sign.

To match a literal underscore or percent sign without matching other characters, the respective character in **pattern** must be preceded by the escape character. The default escape character is the backslash but a different one may be selected by using the ESCAPE clause. To match the escape character itself, write two escape characters.

Note that the backslash already has a special meaning in string literals, so to write a pattern constant that contains a backslash you must write two backslashes in an SQL statement. Thus, writing a pattern that actually matches a literal backslash means writing four backslashes in the statement. You can avoid this by selecting a different escape character with ESCAPE; then a backslash is not special to LIKE anymore. (But it is still special to the string literal parser, so you still need two of them.)

It's also possible to select no escape character by writing ESCAPE ''. This effectively disables the escape mechanism, which makes it impossible to turn off the special meaning of underscore and percent signs in the pattern.

1.3.7 Data Type Formatting Functions

The Advanced Server formatting functions described in the following table provide a powerful set of tools for converting various data types (date/time, integer, floating point, numeric) to formatted strings and for converting from formatted strings to specific data types. These functions all follow a common calling convention: the first argument is the value to be formatted and the second argument is a string template that defines the output or input format.

Function	Return Type	Description	Example	Result
<i>TO_CHAR</i> (DATE [, <i>format</i>)	VAR- CHAR2	Convert a date/time to a string with output, <i>format</i> . If omitted default format is DD-MON-YY.	<i>TO_CHAR</i> (SYSDATE, 'MM/DD/YYYY HH12:MI:SS AM')	07/25/2007 09:43:02 AM
<i>TO_CHAR</i> (TIMESTAMP [, <i>format</i>)	VAR- CHAR2	Convert a timestamp to a string with output, <i>format</i> . If omitted default format is DD-MON-YY.	<i>TO_CHAR</i> (CURRENT_TIMESTAMP, 'MM/DD/YYYY HH12:MI:SS AM')	08/13/2015 08:55:22 PM
<i>TO_CHAR</i> (INTEGER [, <i>format</i>)	VAR- CHAR2	Convert an integer to a string with output, <i>format</i>	<i>TO_CHAR</i> (2412, '999,999S')	2,412+
<i>TO_CHAR</i> (NUMBER [, <i>format</i>)	VAR- CHAR2	Convert a decimal number to a string with output, <i>format</i>	<i>TO_CHAR</i> (10125.35, '999,999.99')	10,125.35
<i>TO_CHAR</i> (DOUBLE PRECISION, <i>format</i>)	VAR- CHAR2	Convert a floating-point number to a string with output, <i>format</i>	<i>TO_CHAR</i> (CAST(123.5282 AS REAL), '999.99')	123.53

continues on next page

Table 18 – continued from previous page

Function	Return Type	Description	Example	Result
<i>TO_DATE(string [, format])</i>	<i>TIMESTAMP</i>	Convert a date or timestamp formatted string to a <i>TIMESTAMP</i> data type	<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</i> <i>

JUL-07 00:00:00`</i>
<i>TO_NUMBER(string [, format])</i>	<i>NUMBER</i>	Convert a number formatted string to a <i>NUMBER</i> data type	<i>TO_NUMBER('2,412,999,999S')</i>	-2412
<i>TO_TIMESTAMP(string, format)</i>	<i>TIMESTAMPTZ</i>	Convert a timestamp formatted string to a <i>TIMESTAMPTZ</i> data type	<i>TO_TIMESTAMP('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, format)</i>	<i>TIMESTAMPTZ</i>	Convert a timestamp formatted string to a <i>TIMESTAMPTZ</i> data type	<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 23:47:18 +05:30</i>

TO_CHAR, TO_DATE, TO_TIMESTAMP, and TO_TIMESTAMP_TZ

In an output template string (for TO_CHAR), there are certain patterns that are recognized and replaced with appropriately-formatted data from the value to be formatted. Any text that is not a template pattern is simply copied verbatim. Similarly, in an input template string (for anything but TO_CHAR), template patterns identify the parts of the input data string to be looked at and the values to be found there.

If you do not specify a date, month, or year when calling TO_TIMESTAMP, TO_TIMESTAMP_TZ or TO_DATE, then by default the output format considers the first date of a current month or current year respectively. In the following example, date, month, and year is not specified in the input string; TO_TIMESTAMP, TO_TIMESTAMP_TZ, and TO_DATE returns a default value of the first date of a current month and current year.

```
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)
```

The following table shows the template patterns available for formatting date values using the TO_CHAR, TO_DATE, TO_TIMESTAMP, and TO_TIMESTAMP_TZ functions.

Pattern	Description
<i>HH</i>	Hour of day (01-12)
<i>HH12</i>	Hour of day (01-12)
<i>HH24</i>	Hour of day (00-23)
<i>MI</i>	Minute (00-59)
<i>SS</i>	Second (00-59)
<i>SSSSS</i>	Seconds past midnight (0-86399)
<i>FFn</i>	Fractional seconds where <i>n</i> is an optional integer from 1 to 9 for the number of digits to return. If omitted, the default is 6.
<i>AM or A.M. or PM or P.M.</i>	Meridian indicator (uppercase)
<i>am or a.m. or pm or p.m.</i>	Meridian indicator (lowercase)
<i>Y,YYY</i>	Year (4 and more digits) with comma
<i>YEAR</i>	Year (spelled out)
<i>SYEAR</i>	Year (spelled out) (BC dates prefixed by a minus sign)
<i>YYYY</i>	Year (4 and more digits)

continues on next page

Table 19 – continued from previous page

Pattern	Description
<i>YYYY</i>	Year (4 and more digits) (BC dates prefixed by a minus sign)
<i>YYY</i>	Last 3 digits of year
<i>YY</i>	Last 2 digits of year
<i>Y</i>	Last digit of year
<i>IYYY</i>	ISO year (4 and more digits)
<i>IYY</i>	Last 3 digits of ISO year
<i>IY</i>	Last 2 digits of ISO year
<i>I</i>	Last 1 digit of ISO year
<i>BC or B.C. or AD or A.D.</i>	Era indicator (uppercase)
<i>bc or b.c. or ad or a.d.</i>	Era indicator (lowercase)
<i>MONTH</i>	Full uppercase month name
<i>Month</i>	Full mixed-case month name
<i>month</i>	Full lowercase month name
<i>MON</i>	Abbreviated uppercase month name (3 chars in English, localized lengths vary)
<i>Mon</i>	Abbreviated mixed-case month name (3 chars in English, localized lengths vary)
<i>mon</i>	Abbreviated lowercase month name (3 chars in English, localized lengths vary)
<i>MM</i>	Month number (01-12)
<i>DAY</i>	Full uppercase day name
<i>Day</i>	Full mixed-case day name
<i>day</i>	Full lowercase day name
<i>DY</i>	Abbreviated uppercase day name (3 chars in English, localized lengths vary)
<i>Dy</i>	Abbreviated mixed-case day name (3 chars in English, localized lengths vary)
<i>dy</i>	Abbreviated lowercase day name (3 chars in English, localized lengths vary)
<i>DDD</i>	Day of year (001-366)
<i>DD</i>	Day of month (01-31)
<i>D</i>	Day of week (1-7; Sunday is 1)
<i>W</i>	Week of month (1-5) (The first week starts on the first day of the month)
<i>WW</i>	Week number of year (1-53) (The first week starts on the first day of the year)

continues on next page

Table 19 – continued from previous page

Pattern	Description
<i>IW</i>	ISO week number of year; the first Thursday of the new year is in week 1
<i>CC</i>	Century (2 digits); the 21st century starts on 2001-01-01
<i>SCC</i>	Same as CC except BC dates are prefixed by a minus sign
<i>J</i>	Julian Day (days since January 1, 4712 BC)
<i>Q</i>	Quarter
<i>RM</i>	Month in Roman numerals (I-XII; I=January) (uppercase)
<i>rm</i>	Month in Roman numerals (i-xii; i=January) (lowercase)
<i>RR</i>	First 2 digits of the year when given only the last 2 digits of the year. Result is based upon an algorithm using the current year and the given 2-digit year. The first 2 digits of the given 2-digit year will be the same as the first 2 digits of the current year with the following exceptions: If the given 2-digit year is <50 and the last 2 digits of the current year is >= 50, then the first 2 digits for the given year is 1 greater than the first 2 digits of the current year. If the given 2-digit year is >= 50 and the last 2 digits of the current year is <50, then the first 2 digits for the given year is 1 less than the first 2 digits of the current year.
<i>RRRR</i>	Only affects <i>TO_DATE</i> function. Allows specification of 2-digit or 4-digit year. If 2-digit year given, then returns first 2 digits of year like RR format. If 4-digit year given, returns the given 4-digit year.
<i>TZH</i>	Time-zone hours
<i>TZM</i>	Time-zone minutes

Date and Time Modifiers

Certain modifiers may be applied to any template pattern to alter its behavior. For example, *FM*Month is the Month pattern with the *FM* modifier. The following table shows the modifier patterns for date/time formatting.

Modifier	Description	Example
<i>FM</i> prefix	Fill mode (suppress padding blanks and zeros)	<i>FM</i> Month
<i>TH</i> suffix	Uppercase ordinal number suffix	<i>DDTH</i>
<i>th</i> suffix	Lowercase ordinal number suffix	<i>DDth</i>

continues on next page

Table 20 – continued from previous page

Modifier	Description	Example
<i>FX</i> prefix	Fixed format global option (see usage notes)	<i>FX Month DD Day</i>
<i>SP</i> suffix	Spell mode	<i>DDSP</i>

Usage notes for date/time formatting:

- ``FM`` suppresses leading zeroes and trailing blanks that would otherwise be added to `↵` make the output of a pattern fixed-width.
- ``TO_TIMESTAMP``, ``TO_TIMESTAMP_TZ``, and ``TO_DATE`` skip multiple blank spaces in the `↵` input string if the ``FX`` option is not used. ``FX`` must be specified as the first `↵` item in the template. For example:

```
* 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.
```

- Ordinary text is allowed in ``TO_CHAR`` templates and will be output literally.
- In conversions from string to ``timestamp``, ``timestampz``, or ``date``, the ``CC`` field `↵` is ignored if there is a ``YYY``, ``YYYY`` or ``Y,YYY`` field. If ``CC`` is used with `↵` ``YY`` or ``Y`` then the year is computed as ``(CC-1)*100+YY``.

The following table shows some examples of the use of the `TO_CHAR` and `TO_DATE` functions:

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

continues on next page

Table 21 – continued from previous page

Expression	Result
<i>TO_CHAR</i> (12, 'FM9990999.9')	'0012.'
<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'
<i>TO_CHAR</i> (-485, '999PR')	'<485>'
<i>TO_CHAR</i> (485, 'L999')	'\$ 485'

continues on next page

Table 21 – continued from previous page

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

The following table shows some examples of the use of the `TO_TIMESTAMP_TZ` function:

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

IMMUTABLE TO_CHAR(TIMESTAMP, format) Function

There are certain cases of the TO_CHAR function that can result in usage of an IMMUTABLE form of the function. Basically, a function is IMMUTABLE if the function does not modify the database, and the function returns the same, consistent value dependent upon only its input parameters. That is, the settings of configuration parameters, the locale, the content of the database, etc. do not affect the results returned by the function.

For more information about function volatility categories VOLATILE, STABLE, and IMMUTABLE, see the PostgreSQL Core documentation at:

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

A particular advantage of an IMMUTABLE function is that it can be used in the CREATE INDEX command to create an index based on that function.

In order for the TO_CHAR function to use the IMMUTABLE form the following conditions must be satisfied:

- The first parameter of the TO_CHAR function must be of data type TIMESTAMP.
- The format specified in the second parameter of the TO_CHAR function must not
 - ↪ affect the return value of the function based on factors such as language, locale, etc. For example a format of 'YYYY-MM-DD HH24:MI:SS' can be used for an IMMUTABLE form of the function since, regardless of locale settings, the result of the function is the date and time expressed solely in numeric form. However, a format of 'DD-MON-YYYY' cannot be used for an IMMUTABLE form of the function because the 3-character abbreviation of the month may return different results depending upon the locale setting.

Format patterns that result in a non-immutable function include any variations of spelled out or abbreviated months (MONTH, MON), days (DAY, DY), median indicators (AM, PM), or era indicators (BC, AD).

For the following example, a table with a TIMESTAMP column is created.

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

The following shows the successful creation of an index with the IMMUTABLE form of the TO_CHAR function.

```
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"
```

The following results in an error because the format specified in the TO_CHAR function prevents the use of the IMMUTABLE form since the 3-character month abbreviation, MON, may result in different return values based on the locale setting.

```
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

The following table lists the template patterns available for formatting numeric values:

Pattern	Description
<i>9</i>	Value with the specified number of digits
<i>0</i>	Value with leading zeroes
<i>. (period)</i>	Decimal point
<i>, (comma)</i>	Group (thousand) separator
<i>\$</i>	Dollar sign
<i>S</i>	Sign anchored to number (uses locale).
<i>L</i>	Currency symbol (uses locale)

Note that *9* results in a value with the same number of digits as there are *9*s. If a digit is not available, the server ignores the corresponding *9*s. The *S* pattern does not support + and *\$* pattern does not support decimal points in the expression.

The following table shows some examples of the use of the **TO_NUMBER** function:

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', 'FM99999999.99999')</i>	' 101.010'

Numeric Modifiers

The following table shows the modifier pattern for numeric formatting:

Pattern	Description	Example
<i>FM</i> prefix	Fill mode (suppress trailing zeroes and padding blanks)	<i>FM99.99</i>

Difference TO_CHAR, TO_NUMBER, and SUBSTR Functions

You can initialize the database using the INITDBOPTS variable to create clusters in a mode compatible with Oracle databases. The clusters created in PostgreSQL mode do not include compatibility features. To create a new cluster in PostgreSQL mode, use the "--no-redwood-compat" option, which offers a similar behavior to PostgreSQL.

A SELECT statement that contains one of the functions as shown in the following table returns the indicated result in Advanced Server and PostgreSQL. Listed are the following differences for TO_CHAR, TO_NUMBER, and SUBSTR functions.

TO_CHAR: Converts the timestamp to string according to a given format.

!!! Note * - The 'syear', 'syxxx', 'year', 'rrrr', 'sp', 'spt', 'scc', 'rr', 'ff', 'ff7', 'ff8', 'ff9', and lower case 'of', 'tzh', and 'tzm' pattern is not supported in PostgreSQL. - 'TH' modifier adds fewer padding zeros in Advanced Server. - The upper case 'TH' pattern, which displays the suffix in uppercase, is not supported in Advanced Server. If the 'TH/th' pattern is specified, the suffix is always displayed in lower case. - A modifier can appear in an 'FM' pattern, and each occurrence toggles the modifier's effect. To toggle the effect of the 'FM' pattern when specified an even number of times turns off the effect and turns on the effect of the 'FM' pattern when specified an odd number of times in PostgreSQL. However, in Advanced Server, if the 'FM' pattern is specified irrespective of an even or odd number of times, it always turns on the effect of the 'FM' pattern.

The following table demonstrates the examples for TO_CHAR(timestamp, text) function.

TO_NUMBER: Converts the string to numeric according to a given format.

!!! Note * - PostgreSQL does not strictly check the format options. In contrast, Advanced Server checks the format except in a few cases when the input string has '-' at the beginning or ',', you do not have to provide the pattern for these characters in the corresponding location in the format string. - In PostgreSQL, you do not have to match all the digits of the input string with the patterns. It outputs those many digits from the input string for which the valid patterns are present. Advanced Server throws an error if you do not provide patterns for all the digits present in the input string. - PostgreSQL ignores any invalid characters present in the input or the format string, whereas Advanced Server throws an error. This kind of behavior may lead to many differences with different types of patterns. - The patterns 'PL', 'PR', and 'SG' are not supported in Advanced Server.

The following table demonstrates the examples for TO_NUMBER(text, text) function.

Examples	Result	
Advanced Server	PostgreSQL	
<pre>SELECT TO_NUMBER('\$1', '00');</pre>	Invalid input syntax for type numeric: \$1	1

continues on next page

Table 26 – continued from previous page

Examples	Result	
<i>SELECT TO_NUMBER('\$\$1', '\$\$0');</i>	Invalid format string: \$\$0	<i>1</i>
<i>SELECT TO_NUMBER('1\$, '0\$');</i>	Invalid input syntax for type numeric: 1.	<i>1</i>
<i>SELECT TO_NUMBER('\$1\$, '\$0\$');</i>	Invalid format string: \$0\$	<i>1</i>
<i>SELECT TO_NUMBER('1\$\$1', '0\$L0');</i>	Invalid input syntax for type numeric: 1.\$1	<i>11</i>
<i>SELECT TO_NUMBER('1,11', '000');</i>	<i>111</i>	<i>11</i>
<i>SELECT TO_NUMBER('1,11', '0,0');</i>	Invalid input syntax for type numeric: 111	<i>11</i>
<i>SELECT TO_NUMBER('1- 1', '0S0');</i>	Invalid format string: 0S0	<i>-11</i>
<i>SELECT TO_NUMBER('1- 1', '000');</i>	Invalid input syntax for type numeric: 1-1	<i>11</i>
<i>SELECT TO_NUMBER('1.11', '000');</i>	Invalid input syntax for type numeric: 1.11	<i>11</i>
<i>SELECT TO_NUMBER('1..1', '0.0');</i>	Invalid input syntax for type numeric: 1..1	<i>1</i>

continues on next page

Table 26 – continued from previous page

Examples	Result	
<i>SELECT TO_NUMBER('11-' \$00S');</i>	Invalid input syntax for type numeric: -\$11	-11
<i>SELECT TO_NUMBER('11' '');</i>	Invalid input syntax for type numeric: 11	Returns blank
<i>SELECT TO_NUMBER(' ' '');</i>	Invalid input syntax for type numeric: **	Returns blank
<i>SELECT TO_NUMBER('\$12.345' L00.000');</i>	Invalid format string: L00.000	12.345
<i>SELECT TO_NUMBER('\$12.345' L00D000');</i>	Invalid format string: L00.000	12.345
<i>SELECT TO_NUMBER('11-' '9MMI');</i>	Invalid format string: 9MMI	-1
<i>SELECT TO_NUMBER('-11' MI99');</i>	Invalid input syntax for type numeric: -11	-11
<i>SELECT TO_NUMBER('000011' FMFM000000');</i>	Invalid format string: FMFM000000	11
<i>SELECT TO_NUMBER('+11' 'S99');

 SELECT TO_NUMBER('11+' '99S');</i>	Invalid input syntax for type numeric: +11 11	11 11

continues on next page

Table 26 – continued from previous page

Examples	Result	
<i>SELECT TO_NUMBER(' +11', 'PL99');</i>	Invalid format string: <i>PL99</i>	<i>11</i>
<i>SELECT TO_NUMBER(' -11', '99PR');</i>	Invalid format string: <i>99PR</i>	<i>-11</i>
<i>SELECT TO_NUMBER('1+1- 1', '9SG9SG9');</i>	Invalid format string: <i>9SG9SG9</i>	<i>-111</i>
<i>SELECT TO_NUMBER('01,000', '99G999'); <br
="" ><br="" select<br=""/>TO_NUMBER('01,000', 'FM99G999');</i>	<i>1000</i>	<i>1000

 100</i>

SUBSTR: Returns a substring of a character value.

!!! Note * - The index of the string starts from 1. In PostgreSQL, if zero as the start parameter to fetch only one character, the result returns blank. If you try to fetch two characters, the result shows one character since zero is specified in the start position. However, in Advanced Server, if zero as the start parameter, internally, it converts to 1 and starts the searching. - Negative values at the start parameter are not allowed in PostgreSQL, whereas in Advanced Server, a reverse search occurs. - PostgreSQL throws an error if a numeric value is passed for start and length parameter, whereas Advanced Server converts it to an integer.

The following table demonstrates the examples for SUBSTR(string, from, count), SUBSTR(string, from), SUBSTR(string, from, count)::bytea, and SUBSTR(string, from)::bytea function.

1.3.8 Date/Time Functions and Operators

The Date/Time Functions table shows the available functions for date/time value processing, with details appearing in the following subsections. The following table illustrates the behaviors of the basic arithmetic operators (+, -). For formatting functions, refer to [IMMUTABLE TO_CHAR\(TIMESTAMP, format\) Function](#). You should be familiar with the background information on date/time data types, see [Date/Time Types](#).

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

continues on next page

Table 27 – continued from previous page

Operator	Example	Result

In the date/time functions of the following table the use of the DATE and TIMESTAMP data types are interchangeable.

Function	Return Type	Description	Example	Result
<i>ADD_MONTHS</i> (<i>DATE</i> , <i>NUMBER</i>)	<i>DATE</i>	Add months to a date	<i>ADD_MONTHS</i> (<i>'28-FEB-97'</i> , <i>3.8</i>)	<i>31-MAY-97</i> <i>00:00:00</i>
<i>CURRENT_DATE</i>	<i>DATE</i>	Current date	<i>CURRENT_DATE</i>	<i>04-JUL-07</i>
<i>CURRENT_TIMESTAMP</i>	<i>TIMESTAMP</i>	Returns the current date and time	<i>CURRENT_TIMESTAMP</i>	<i>04-JUL-07</i> <i>15:33:23.484</i>
<i>EXTRACT</i> (<i>field</i> <i>FROM</i> <i>TIMESTAMP</i>)	<i>DOUBLE PRECISION</i>	Get subfield	<i>EXTRACT</i> (<i>hour</i> <i>FROM</i> <i>TIMESTAMP</i> <i>'2001-02-16 20:38:40'</i>)	<i>20</i>
<i>LAST_DAY</i> (<i>DATE</i>)	<i>DATE</i>	Returns the last day of the month represented by the given date. If the given date contains a time portion, it is carried forward to the result unchanged	<i>LAST_DAY</i> (<i>'14-APR-98'</i>)	<i>30-APR-98</i> <i>00:00:00</i>
<i>LOCALTIMESTAMP</i> [(<i>precision</i>)]	<i>TIMESTAMP</i>	Current date and time (start of current transaction)	<i>LOCALTIMESTAMP</i>	<i>04-JUL-07</i> <i>15:33:23.484</i>
<i>MONTHS_BETWEEN</i> (<i>DATE</i> , <i>DATE</i>)	<i>NUMBER</i>	Number of months between two dates	<i>MONTHS_BETWEEN</i> (<i>'28-FEB-07'</i> , <i>'30-NOV-06'</i>)	<i>3</i>

continues on next page

Table 28 – continued from previous page

Function	Return Type	Description	Example	Result
<i>NEXT_DAY</i> (<i>DATE</i> , <i>day-ofweek</i>)	<i>DATE</i>	Date falling on <i>dayofweek</i> following specified date	<i>NEXT_DAY</i> ('16- APR- 07', <i>FRI</i>)	20-APR-07 00:00:00
<i>NEW_TIME</i> (<i>DATE</i> , <i>VAR-CHAR</i> , <i>VAR-CHAR</i>)	<i>DATE</i>	Converts a date and time to an alternate time zone	<i>NEW_TIME</i> (<i>TO_DATE</i> '2005/05/29 01:45', 'AST', 'PST')	2005/05/29 21:45:00
<i>NUM-TODSIN-TER-VAL</i> (<i>NUMBER</i> , <i>INTER-VAL</i>)	<i>INTERVAL</i>	Converts a number to a specified day or second interval	<i>SE-LECT</i> <i>num-todsinter-val</i> (100, 'hour');	4 days 04:00:00
<i>NUM-TOYMINTER-VAL</i> (<i>NUMBER</i> , <i>INTER-VAL</i>)	<i>INTERVAL</i>	Converts a number to a specified year or month interval	<i>SE-LECT</i> <i>num-toyminter-val</i> (100, 'month');	8 years 4 mons
<i>ROUND</i> (<i>DATE</i> [, <i>format</i>)	<i>DATE</i>	Date rounded according to <i>format</i>	<i>ROUND</i> (<i>TO_DATE</i> ('29- MAY- 05'),'MON')	01-JUN-05 00:00:00
<i>SYS_EXTRACT_UTC</i> (<i>TIMESTAMP</i> <i>WITH TIME</i> <i>ZONE</i>)		Returns Coordinated Universal Time	<i>SYS_EXTRACT_UTC</i> (<i>CAST</i> 24-MAR-11 ('24-MAR-11 12:30:00PM - 04:00' AS <i>TIMESTAMP</i> <i>WITH</i> <i>TIME ZONE</i>))	16:30:00`
<i>SYSDATE</i>	<i>DATE</i>	Returns current date and time	<i>SYSDATE</i>	01-AUG-12 11:12:34
<i>SYSTIMESTAMP</i> (<i>)</i>	<i>TIMESTAMP</i>	Returns current date and time	<i>SYSTIMESTAMP</i>	01-AUG-12 11:11:23. 665229 -07:00

continues on next page

Table 28 – continued from previous page

Function	Return Type	Description	Example	Result
<i>TRUNC</i> (DATE [format])	DATE	Truncate according to <i>format</i>	<i>TRUNC</i> (TO_DATE ('29- MAY- 05'), 'MON')	01-MAY-05 00:00:00

ADD_MONTHS

The `ADD_MONTHS` functions adds (or subtracts if the second parameter is negative) the specified number of months to the given date. The resulting day of the month is the same as the day of the month of the given date except when the day is the last day of the month in which case the resulting date always falls on the last day of the month.

Any fractional portion of the number of months parameter is truncated before performing the calculation.

If the given date contains a time portion, it is carried forward to the result unchanged.

The following are examples of the `ADD_MONTHS` function.

```
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)
```

CURRENT DATE/TIME

Advanced Server provides a number of functions that return values related to the current date and time. These functions all return values based on the start time of the current transaction.

- ``CURRENT_DATE``
- ``CURRENT_TIMESTAMP``
- ``LOCALTIMESTAMP``

- `LOCALTIMESTAMP(precision)`

`CURRENT_DATE` returns the current date and time based on the start time of the current transaction. The value of `CURRENT_DATE` will not change if called multiple times within a transaction.

```
SELECT CURRENT_DATE FROM DUAL;
*      date
-----
06-AUG-07
```

`CURRENT_TIMESTAMP` returns the current date and time. When called from a single SQL statement, it will return the same value for each occurrence within the statement. If called from multiple statements within a transaction, may return different values for each occurrence. If called from a function, may return a different value than the value returned by `current_timestamp` in the caller.

```
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` can optionally be given a precision parameter which causes the result to be rounded to that many fractional digits in the seconds field. Without a precision parameter, the result is given to the full available precision.

```
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)
```

Since these functions return the start time of the current transaction, their values do not change during the transaction. This is considered a feature: the intent is to allow a single transaction to have a consistent notion of the “current” time, so that multiple modifications within the same transaction bear the same time stamp. Other database systems may advance these values more frequently.

EXTRACT

The `EXTRACT` function retrieves subfields such as year or hour from date/time values. The `EXTRACT` function returns values of type `DOUBLE PRECISION`. The following are valid field names:

YEAR

The year field

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

MONTH

The number of the month within the year (1 - 12)

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

DAY

The day (of the month) field (1 - 31)

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

HOURL

The hour field (0 - 23)

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

MINUTE

The minutes field (0 - 59)

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

SECOND

The seconds field, including fractional parts (0 - 59)

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

MONTHS_BETWEEN

The MONTHS_BETWEEN function returns the number of months between two dates. The result is a numeric value which is positive if the first date is greater than the second date or negative if the first date is less than the second date.

The result is always a whole number of months if the day of the month of both date parameters is the same, or both date parameters fall on the last day of their respective months.

The following are some examples of the MONTHS_BETWEEN function.

```
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

The NEXT_DAY function returns the first occurrence of the given weekday strictly greater than the given date. At least the first three letters of the weekday must be specified - e.g., SAT. If the given date contains a time portion, it is carried forward to the result unchanged.

The following are examples of the NEXT_DAY function.

```
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

The `NEW_TIME` function converts a date and time from one time zone to another. `NEW_TIME` returns a value of type `DATE`. The syntax is:

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

`time_zone1` and `time_zone2` must be string values from the Time Zone column of the following table:

Time Zone	Offset from UTC	Description
<i>AST</i>	UTC+4	Atlantic Standard Time
<i>ADT</i>	UTC+3	Atlantic Daylight Time
<i>BST</i>	UTC+11	Bering Standard Time
<i>BDT</i>	UTC+10	Bering Daylight Time
<i>CST</i>	UTC+6	Central Standard Time
<i>CDT</i>	UTC+5	Central Daylight Time
<i>EST</i>	UTC+5	Eastern Standard Time
<i>EDT</i>	UTC+4	Eastern Daylight Time
<i>GMT</i>	UTC	Greenwich Mean Time
<i>HST</i>	UTC+10	Alaska-Hawaii Standard Time
<i>HDT</i>	UTC+9	Alaska-Hawaii Daylight Time
<i>MST</i>	UTC+7	Mountain Standard Time
<i>MDT</i>	UTC+6	Mountain Daylight Time
<i>NST</i>	UTC+3:30	Newfoundland Standard Time
<i>PST</i>	UTC+8	Pacific Standard Time
<i>PDT</i>	UTC+7	Pacific Daylight Time
<i>YST</i>	UTC+9	Yukon Standard Time
<i>YDT</i>	UTC+8	Yukon Daylight Time

Following is an example of the `NEW_TIME` function:

```
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

The `NUMTODSINTERVAL` function converts a numeric value to a time interval that includes day through second interval units. When calling the function, specify the smallest fractional interval type to be included in the result set. The valid interval types are `DAY`, `HOURL`, `MINUTE`, and `SECOND`.

The following example converts a numeric value to a time interval that includes days and hours:

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

(continues on next page)

(continued from previous page)

```
4 days 04:00:00
(1 row)
```

The following example converts a numeric value to a time interval that includes minutes and seconds:

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

NUMTOYMINTERVAL

The NUMTOYMINTERVAL function converts a numeric value to a time interval that includes year through month interval units. When calling the function, specify the smallest fractional interval type to be included in the result set. The valid interval types are YEAR and MONTH.

The following example converts a numeric value to a time interval that includes years and months:

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

The following example converts a numeric value to a time interval that includes years only:

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

ROUND

The ROUND function returns a date rounded according to a specified template pattern. If the template pattern is omitted, the date is rounded to the nearest day. The following table shows the template patterns for the ROUND function.

Pattern	Description
CC, SCC	Returns January 1, <i>cc</i> '01 where `cc` is first 2 digits of the given year if last 2 digits <= 50, or 1 greater than the first 2 digits of the given year if last 2 digits > 50; (for AD years)
YYYY, YYYY, YEAR, SYEAR, YYY, YY, Y	Returns January 1, <i>yyyy</i> where <i>yyyy</i> is rounded to the nearest year; rounds down on June 30, rounds up on July 1

continues on next page

Table 30 – continued from previous page

Pattern	Description
<i>IYYY, IYY, IY, I</i>	Rounds to the beginning of the ISO year which is determined by rounding down if the month and day is on or before June 30th, or by rounding up if the month and day is July 1st or later
<i>Q</i>	Returns the first day of the quarter determined by rounding down if the month and day is on or before the 15th of the second month of the quarter, or by rounding up if the month and day is on the 16th of the second month or later of the quarter
<i>MONTH, MON, MM, RM</i>	Returns the first day of the specified month if the day of the month is on or prior to the 15th; returns the first day of the following month if the day of the month is on the 16th or later
<i>WW</i>	Round to the nearest date that corresponds to the same day of the week as the first day of the year
<i>IW</i>	Round to the nearest date that corresponds to the same day of the week as the first day of the ISO year
<i>W</i>	Round to the nearest date that corresponds to the same day of the week as the first day of the month
<i>DDD, DD, J</i>	Rounds to the start of the nearest day; 11:59:59 AM or earlier rounds to the start of the same day; 12:00:00 PM or later rounds to the start of the next day
<i>DAY, DY, D</i>	Rounds to the nearest Sunday
<i>HH, HH12, HH24</i>	Round to the nearest hour
<i>MI</i>	Round to the nearest minute

Following are examples of usage of the ROUND function.

The following examples round to the nearest hundred years.

```

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

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

```

The following examples round to the nearest year.

```

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)

```

The following examples round to the nearest ISO year. The first example rounds to 2004 and the ISO year for 2004 begins on December 29th of 2003. The second example rounds to 2005 and the ISO year for 2005 begins on January 3rd of that same year.

(An ISO year begins on the first Monday from which a 7 day span, Monday thru Sunday, contains at least 4 days of the new year. Thus, it is possible for the beginning of an ISO year to start in December of the prior year.)

```

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)

```

The following example round to the nearest quarter:

```

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

```

The following example round to the nearest month:

```

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

```

(continues on next page)

(continued from previous page)

```

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)

```

The following examples round to the nearest week. The first day of 2007 lands on a Monday so in the first example, January 18th is closest to the Monday that lands on January 15th. In the second example, January 19th is closer to the Monday that falls on January 22nd.

```

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)

```

The following examples round to the nearest ISO week. An ISO week begins on a Monday. In the first example, January 1, 2004 is closest to the Monday that lands on December 29, 2003. In the second example, January 2, 2004 is closer to the Monday that lands on January 5, 2004.

```

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)

```

The following examples round to the nearest week where a week is considered to start on the same day as the first day of the month.

```

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
-----

```

(continues on next page)

(continued from previous page)

```
01-MAR-07 00:00:00
(1 row)
```

The following examples round to the nearest day.

```
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)
```

The following examples round to the start of the nearest day of the week (Sunday).

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

The following examples round to the nearest hour.

```
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)
```

The following examples round to the nearest minute.

```
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;
```

(continues on next page)

(continued from previous page)

```

*      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

The SYSDATE function returns the current date and time (timestamp without timezone) of the operating system on which the database server resides. The function is STABLE and requires no arguments.

When called from a single SQL statement, it will return the same value for each occurrence within the statement. If called from multiple statements within a transaction, may return different values for each occurrence. If called from a function, may return a different value than the value returned by SYSDATE in the caller.

The following example demonstrates a call to SYSDATE:

```

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

TRUNC

The TRUNC function returns a date truncated according to a specified template pattern. If the template pattern is omitted, the date is truncated to the nearest day. The following table shows the template patterns for the TRUNC function.

Pattern	Description
<i>CC, SCC</i>	Returns January 1, <i>cc</i> 01 where <i>cc</i> is first 2 digits of the given year
<i>YYYY, YYYY, YEAR, SYEAR, YYY, YY, Y</i>	Returns January 1, <i>yyyy</i> where <i>yyyy</i> is the given year
<i>IYYY, IYY, IY, I</i>	Returns the start date of the ISO year containing the given date
<i>Q</i>	Returns the first day of the quarter containing the given date
<i>MONTH, MON, MM, RM</i>	Returns the first day of the specified month

continues on next page

Table 31 – continued from previous page

Pattern	Description
<i>WW</i>	Returns the largest date just prior to, or the same as the given date that corresponds to the same day of the week as the first day of the year
<i>IW</i>	Returns the start of the ISO week containing the given date
<i>W</i>	Returns the largest date just prior to, or the same as the given date that corresponds to the same day of the week as the first day of the month
<i>DDD, DD, J</i>	Returns the start of the day for the given date
<i>DAY, DY, D</i>	Returns the start of the week (Sunday) containing the given date
<i>HH, HH12, HH24</i>	Returns the start of the hour
<i>MI</i>	Returns the start of the minute

Following are examples of usage of the TRUNC function.

The following example truncates down to the hundred years unit.

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

The following example truncates down to the year.

```
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)
```

The following example truncates down to the beginning of the ISO year.

```
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)
```

The following example truncates down to the start date of the quarter.

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

The following example truncates to the start of the month.

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

The following example truncates down to the start of the week determined by the first day of the year. The first day of 2007 lands on a Monday so the Monday just prior to January 19th is January 15th.

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

The following example truncates to the start of an ISO week. An ISO week begins on a Monday. January 2, 2004 falls in the ISO week that starts on Monday, December 29, 2003.

```
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)
```

The following example truncates to the start of the week where a week is considered to start on the same day as the first day of the month.

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

The following example truncates to the start of the day.

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

The following example truncates to the start of the week (Sunday).

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

(continues on next page)

(continued from previous page)

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

The following example truncates to the start of the hour.

```
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)
```

The following example truncates to the minute.

```
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

This section describes Advanced Server's functions for operating on sequence objects. Sequence objects (also called sequence generators or just sequences) are special single-row tables created with the `CREATE SEQUENCE` command. A sequence object is usually used to generate unique identifiers for rows of a table. The sequence functions, listed below, provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

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

`sequence` is the identifier assigned to the sequence in the `CREATE SEQUENCE` command. The following describes the usage of these functions.

NEXTVAL

Advance the sequence object to its next value and return that value. This is done atomically: even if multiple sessions execute `NEXTVAL` concurrently, each will safely receive a distinct sequence value.

CURRVAL

Return the value most recently obtained by `NEXTVAL` for this sequence in the current session. (An error is reported if `NEXTVAL` has never been called for this sequence in this session.) Notice that because this is returning a session-local value, it gives a predictable answer whether or not other sessions have executed `NEXTVAL` since the current session did.

If a sequence object has been created with default parameters, `NEXTVAL` calls on it will return successive values beginning with 1. Other behaviors can be obtained by using special parameters in the `CREATE SEQUENCE` command.

Important: To avoid blocking of concurrent transactions that obtain numbers from the same sequence, a `NEXTVAL` operation is never rolled back; that is, once a value has been fetched it is considered used, even if the transaction that did the `NEXTVAL` later aborts. This means that aborted transactions may leave unused “holes” in the sequence of assigned values.

1.3.10 Conditional Expressions

The following section describes the SQL-compliant conditional expressions available in Advanced Server.

CASE

The SQL CASE expression is a generic conditional expression, similar to if/else statements in other languages:

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

CASE clauses can be used wherever an expression is valid. **condition** is an expression that returns a **BOOLEAN** result. If the result is **TRUE** then the value of the CASE expression is the **result** that follows the condition. If the result is **FALSE** any subsequent **WHEN** clauses are searched in the same manner. If no **WHEN condition** is **TRUE** then the value of the CASE expression is the **result** in the **ELSE** clause. If the **ELSE** clause is omitted and no condition matches, the result is **NULL**.

An example:

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

The data types of all the **result** expressions must be convertible to a single output type.

The following “simple” CASE expression is a specialized variant of the general form above:

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

The **expression** is computed and compared to all the **value** specifications in the **WHEN** clauses until one is found that is equal. If no match is found, the **result** in the **ELSE** clause (or a null value) is returned.

The example above can be written using the simple CASE syntax:

```
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)
```

A CASE expression does not evaluate any subexpressions that are not needed to determine the result. For example, this is a possible way of avoiding a division-by-zero failure:

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

COALESCE

The COALESCE function returns the first of its arguments that is not null. Null is returned only if all arguments are null.

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

It is often used to substitute a default value for null values when data is retrieved for display or further computation. For example:

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

Like a CASE expression, COALESCE will not evaluate arguments that are not needed to determine the result; that is, arguments to the right of the first non-null argument are not evaluated. This SQL-standard function provides capabilities similar to NVL and IFNULL, which are used in some other database systems.

NULLIF

The NULLIF function returns a null value if value1 and value2 are equal; otherwise it returns value1.

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

This can be used to perform the inverse operation of the COALESCE example given above:

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

If value1 is (none), return a null, otherwise return value1.

NVL

The NVL function returns the first of its arguments that is not null. NVL evaluates the first expression; if that expression evaluates to NULL, NVL returns the second expression.

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

The return type is the same as the argument types; all arguments must have the same data type (or be coercible to a common type). NVL returns NULL if all arguments are NULL.

The following example computes a bonus for non-commissioned employees. If an employee is a commissioned employee, this expression returns the employee's commission; if the employee is not a commissioned employee (that is, his commission is NULL), this expression returns a bonus that is 10% of his salary.

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

NVL2

NVL2 evaluates an expression, and returns either the second or third expression, depending on the value of the first expression. If the first expression is not NULL, NVL2 returns the value in `expr2`; if the first expression is NULL, NVL2 returns the value in `expr3`.

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

The return type is the same as the argument types; all arguments must have the same data type (or be coercible to a common type).

The following example computes a bonus for commissioned employees - if a given employee is a commissioned employee, this expression returns an amount equal to 110% of his commission; if the employee is not a commissioned employee (that is, his commission is NULL), this expression returns 0.

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

GREATEST and LEAST

The GREATEST and LEAST functions select the largest or smallest value from a list of any number of expressions.

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

The expressions must all be convertible to a common data type, which will be the type of the result. Null values in the list are ignored. The result will be null only if all the expressions evaluate to null.

!!! Note * The GREATEST and LEAST are not in the SQL standard, but are a common extension.

1.3.11 Aggregate Functions

Aggregate functions compute a single result value from a set of input values. The built-in aggregate functions are listed in the following tables.

Function	Argument Type	Return Type	Description
<i>AVG (expression)</i>	<i>INTEGER</i> , <i>REAL</i> , <i>DOUBLE</i> <i>PRECISION</i> , <i>NUMBER</i>	<i>NUMBER</i> for any integer type, <i>DOUBLE PRECISION</i> for a floating-point argument, otherwise the same as the argument data type	The average (arithmetic mean) of all input values
<i>COUNT(*)</i>		<i>BIGINT</i>	Number of input rows
<i>COUNT (expression)</i>	Any	<i>BIGINT</i>	Number of input rows for which the value of expression is not null
<i>MAX (expression)</i>	Any numeric, string, date/time, or bytea type	Same as argument type	Maximum value of expression across all input values
<i>MIN (expression)</i>	Any numeric, string, date/time, or bytea type	Same as argument type	Minimum value of expression across all input values

continues on next page

Table 32 – continued from previous page

Function	Argument Type	Return Type	Description
<i>SUM (expression)</i>	<i>INTEGER</i> , <i>REAL</i> , <i>DOUBLE</i> <i>PRECISION</i> , <i>NUMBER</i>	<i>BIGINT</i> for <i>SMALL-INT</i> or <i>INTEGER</i> arguments, <i>NUMBER</i> for <i>BIGINT</i> arguments, <i>DOUBLE</i> <i>PRECISION</i> for floating-point arguments, otherwise the same as the argument data type	Sum of expression across all input values

It should be noted that except for COUNT, these functions return a null value when no rows are selected. In particular, SUM of no rows returns null, not zero as one might expect. The COALESCE function may be used to substitute zero for null when necessary.

The following table shows the aggregate functions typically used in statistical analysis. (These are separated out merely to avoid cluttering the listing of more-commonly-used aggregates.) Where the description mentions N, it means the number of input rows for which all the input expressions are non-null. In all cases, null is returned if the computation is meaningless, for example when N is zero.

Function	Argument Type	Return Type	Description
<i>CORR(Y, X)</i>	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	Correlation coefficient
<i>COVAR_POP(Y, X)</i>	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	Population covariance
<i>COVAR_SAMP(Y, X)</i>	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	Sample covariance
<i>REGR_AVGX(Y, X)</i>	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	Average of the independent variable (sum`X) / N`)

continues on next page

Table 33 – continued from previous page

Function	Argument Type	Return Type	Description
<i>REGR_AVGY</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	Average of the dependent variable ($\sum(Y) / N$)
<i>REGR_COUNT</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	Number of input rows in which both expressions are nonnull
<i>REGR_INTERCEPT</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	y-intercept of the least-squares-fit linear equation determined by the (X, Y) pairs
<i>REGR_R2</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	Square of the correlation coefficient
<i>REGR_SLOPE</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	Slope of the least-squares-fit linear equation determined by the (X, Y) pairs
<i>REGR_SXX</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	$\sum(X^2) - \sum(X) * \sum(X) / N$ (“sum of squares” of the independent variable)
<i>REGR_SXY</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	$\sum(X*Y) - \sum(X) * \sum(Y) / N$ (“sum of products” of independent times dependent variable)
<i>REGR_SYY</i> (Y, X)	<i>DOUBLE PRECISION</i>	<i>DOUBLE PRECISION</i>	$\sum(Y^2) - \sum(Y) * \sum(Y) / N$ (“sum of squares” of the dependent variable)

continues on next page

Table 33 – continued from previous page

Function	Argument Type	Return Type	Description
<i>STDDEV(expression)</i>	<i>INTEGER, REAL, DOUBLE PRECISION, NUMBER</i>	<i>DOUBLE PRECISION</i> for floating-point arguments, otherwise <i>NUMBER</i>	Historic alias for <i>STDDEV_SAMP</i>
<i>STDDEV_POP(expression)</i>	<i>INTEGER, REAL, DOUBLE PRECISION, NUMBER</i>	<i>DOUBLE PRECISION</i> for floating-point arguments, otherwise <i>NUMBER</i>	Population standard deviation of the input values
<i>STDDEV_SAMP(expression)</i>	<i>INTEGER, REAL, DOUBLE PRECISION, NUMBER</i>	<i>DOUBLE PRECISION</i> for floating-point arguments, otherwise <i>NUMBER</i>	Sample standard deviation of the input values
<i>VARIANCE(expression)</i>	<i>INTEGER, REAL, DOUBLE PRECISION, NUMBER</i>	<i>DOUBLE PRECISION</i> for floating-point arguments, otherwise <i>NUMBER</i>	Historical alias for <i>VAR_SAMP</i>
<i>VAR_POP(expression)</i>	<i>INTEGER, REAL, DOUBLE PRECISION, NUMBER</i>	<i>DOUBLE PRECISION</i> for floating-point arguments, otherwise <i>NUMBER</i>	Population variance of the input values (square of the population standard deviation)

continues on next page

Table 33 – continued from previous page

Function	Argument Type	Return Type	Description
<i>VAR_SAMP(expression)</i>	<i>INTEGER,</i> <i>REAL,</i> <i>DOUBLE</i> <i>PRECISION,</i> <i>NUMBER</i>	<i>DOUBLE</i> <i>PRECISION</i> for floating-point arguments, otherwise <i>NUMBER</i>	Sample variance of the input values (square of the sample standard deviation)

LISTAGG

Advanced Server has added the LISTAGG function to support string aggregation. LISTAGG is an aggregate function that concatenates data from multiple rows into a single row in an ordered manner. You can optionally include a custom delimiter for your data.

The LISTAGG function mandates the use of an ORDER BY clause under a WITHIN GROUP clause to concatenate values of the measure column, and then generate the ordered aggregated data.

Objective

- ``LISTAGG`` can be used without any grouping. In this case, the ``LISTAGG`` function
 → operates on all rows in a table and returns a single row.
- ``LISTAGG`` can be used with the ``GROUP BY`` clause. In this case, the ``LISTAGG``
 → function operates on each group and returns an aggregated output for each group.
- ``LISTAGG`` can be used with the ``OVER`` clause. In this case, the ``LISTAGG`` function
 → partitions a query result set into groups based on the expression in the ``query_``
 → ``partition_by_clause`` and then aggregates data in each group.

Synopsis

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

Parameters

measure_expr

measure_expr (mandatory) specifies the column or expression that assigns a value to aggregate. NULL values are ignored.

delimiter

delimiter (optional) specifies a string that separates the concatenated values in the result row. The **delimiter** can be a NULL value, string, character literal, column name, or constant expression. If ignored, the LISTAGG function uses a NULL value by default.

order_by_clause

order_by_clause (mandatory) determines the sort order in which the concatenated values are returned.

query_partition_by_clause

query_partition_by_clause (optional) allows LISTAGG function to be used as an analytic function and sets the range of records for each group in the OVER clause.

Return Type

The LISTAGG function returns a string value.

Examples

The following example concatenates the values in the **emp** table and lists all the employees separated by a delimiter comma.

First, create a table named **emp** and then insert records into the **emp** table.

```

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)

```

The following example uses **PARTITION BY** clause with LISTAGG in **emp** table and generates output based on a partition by **deptno** that applies to each partition and not on the entire table.

```
edb=# SELECT DISTINCT DEPTNO, LISTAGG(ENAME, ',') WITHIN GROUP (ORDER BY
ENAME) OVER(PARTITION BY DEPTNO) FROM emp;
```

deptno	listagg
30	ALLEN, BLAKE, MARTIN, WARD
20	JONES

(2 rows)

The following example is identical to the previous example, except it includes the GROUP BY clause.

```
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)

MEDIAN

The **MEDIAN** function calculates the middle value of an expression from a given range of values; NULL values are ignored. The **MEDIAN** function returns an error if a query does not reference the user-defined table.

Objective

- **'MEDIAN'** can be used without any grouping. In this case, the **'MEDIAN'** function operates on all rows in a table and returns a single row.
- **'MEDIAN'** can be used with the **'OVER'** clause. In this case, the **'MEDIAN'** function partitions a query result set into groups based on the **'expression'** specified in the **'PARTITION BY'** clause and then aggregates data in each group.

Synopsis

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

Parameters

median_expression

median_expression (mandatory) is a target column or expression that the **MEDIAN** function operates on and returns a median value. It can be a numeric, datetime, or interval data type.

PARTITION BY

PARTITION BY clause (optional) allows a **MEDIAN** function to be used as an analytic function and sets the range of records for each group in the **OVER** clause.

Return Types

The return type is determined by the input data type of **expression**. The following table illustrates the return type for each input type.

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>

Examples

In the following example, a query returns the median salary for each department in the emp table:

```
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)
```

The following example uses `PARTITION BY` clause with `MEDIAN` in emp table and returns the median salary based on a partition by deptno:

```
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)
```

The `MEDIAN` function can be compared with `PERCENTILE_CONT`. In the following example, `MEDIAN` generates the same result as `PERCENTILE_CONT`:

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

(continues on next page)

(continued from previous page)

```
* 1250 |          1250
(1 row)
```

STATS_MODE

The STATS_MODE function takes a set of values as an argument and returns the value that occurs with the highest frequency. If multiple values are appearing with the same frequency, the STATS_MODE function arbitrarily chooses the first value and returns only that one value.

Objective

- `STATS\_MODE` function can be used without any grouping. In this case, the `STATS\_MODE` function operates on all the rows in a table and returns a single value.
- `STATS\_MODE` can be used as an ordered-set aggregate function using the `WITHIN GROUP` clause. In this case, the `STATS\_MODE` function operates on the ordered data set.
- `STATS\_MODE` can be used with the `GROUP BY` clause. In this case, the `STATS\_MODE` function operates on each group and returns the most frequent and aggregated output for each group.

Synopsis

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

OR

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

Parameters

expr

An expression or value to assign to the column.

Return Type

The STATS_MODE function returns a value that appears frequently. However, if all the values of a column are NULL, the STATS_MODE returns NULL.

Examples

The following example returns the mode of salary in the emp table:

```
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)
```

(continues on next page)

(continued from previous page)

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

The following example uses GROUP BY and ORDER BY clause with STATS_MODE in emp table and returns the salary based on a partition by deptno:

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

The following example uses the WITHIN GROUP clause with the STATS_MODE function to perform aggregation on the ordered data set.

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

BIT_AND_AGG

The BIT_AND_AGG is a bitwise aggregation function that performs a bitwise AND operation and returns a value based on the data type of input argument.

Objective

- The `BIT\_AND\_AGG` function can be used with the `GROUP BY` clause. In this case, the
 ↳ `BIT\_AND\_AGG` function operates on a group and returns the result of a bitwise
 ↳ `AND` operation.
- The `DISTINCT` or `UNIQUE` keywords can be used with the `BIT\_AND\_AGG` function to
 ↳ ensure that unique values in the `expr` are used for computation.

Synopsis

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

Parameters

expr

An expression or value to assign to the column.

Return Type

The BIT_AND_AGG function returns the same value as the data type of the input argument. However, if all the values of a column are NULL, the BIT_AND_AGG returns NULL.

Examples

The following example applies the BIT_AND_AGG function to the sal column in an emp table and groups the result by deptno:

```
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

The BIT_OR_AGG is a bitwise aggregation function that performs a bitwise OR operation and returns a value based on the data type of input argument.

Objective

- The `BIT\_OR\_AGG` function can be used with the `GROUP BY` clause. In this case, the
 ↪ `BIT\_OR\_AGG` function operates on a group and returns the result of a bitwise
 ↪ `OR` operation.
- The `DISTINCT` or `UNIQUE` keywords can be used with the `BIT\_OR\_AGG` function to
 ↪ ensure that unique values in the `expr` are used for computation.

Synopsis

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

Parameters

expr

An expression or value to assign to the column.

Return Type

The BIT_OR_AGG function returns the same value as the data type of the input argument. However, if all the values of a column are NULL, the BIT_OR_AGG returns NULL.

Examples

The following example applies BIT_OR_AGG function to the sal column in an emp table and groups the result by deptno:

```
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

This section describes the SQL-compliant subquery expressions available in Advanced Server. All of the expression forms documented in this section return Boolean (TRUE/FALSE) results.

EXISTS

The argument of **EXISTS** is an arbitrary **SELECT** statement, or subquery. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of **EXISTS** is **TRUE**; if the subquery returns no rows, the result of **EXISTS** is **FALSE**.

EXISTS(subquery)

The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery.

The subquery will generally only be executed far enough to determine whether at least one row is returned, not all the way to completion. It is unwise to write a subquery that has any side effects (such as calling sequence functions); whether the side effects occur or not may be difficult to predict.

Since the result depends only on whether any rows are returned, and not on the contents of those rows, the output list of the subquery is normally uninteresting. A common coding convention is to write all **EXISTS** tests in the form **EXISTS**(**SELECT** 1 **WHERE** . . .). There are exceptions to this rule however, such as subqueries that use **INTERSECT**.

This simple example is like an inner join on **deptno**, but it produces at most one output row for each **dept** row, even though there are multiple matching **emp** rows:

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

IN

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of **IN** is **TRUE** if any equal subquery row is found. The result is **FALSE** if no equal row is found (including the special case where the subquery returns no rows).

expression **IN** (subquery)

Note that if the left-hand expression yields **NULL**, or if there are no equal right-hand values and at least one right-hand row yields **NULL**, the result of the **IN** construct will be **NULL**, not **FALSE**. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with **EXISTS**, it's unwise to assume that the subquery will be evaluated completely.

NOT IN

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of **NOT IN** is **TRUE** if only unequal subquery rows are found (including the special case where the subquery returns no rows). The result is **FALSE** if any equal row is found.

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

Note that if the left-hand expression yields **NULL**, or if there are no equal right-hand values and at least one right-hand row yields **NULL**, the result of the **NOT IN** construct will be **NULL**, not **TRUE**. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with **EXISTS**, it's unwise to assume that the subquery will be evaluated completely.

ANY/SOME

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of **ANY** is **TRUE** if any true result is obtained. The result is **FALSE** if no true result is found (including the special case where the subquery returns no rows).

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

SOME is a synonym for **ANY**. **IN** is equivalent to **= ANY**.

Note that if there are no successes and at least one right-hand row yields **NULL** for the operator's result, the result of the **ANY** construct will be **NULL**, not **FALSE**. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with **EXISTS**, it's unwise to assume that the subquery will be evaluated completely.

ALL

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of **ALL** is **TRUE** if all rows yield true (including the special case where the subquery returns no rows). The result is **FALSE** if any false result is found. The result is **NULL** if the comparison does not return **FALSE** for any row, and it returns **NULL** for at least one row.

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

NOT IN is equivalent to **<> ALL**. As with **EXISTS**, it's unwise to assume that the subquery will be evaluated completely.

1.3.13 Identifier Functions

The following section describes identifier functions that provide information about the instance and session.

SYS_GUID

The SYS_GUID function generates and returns a globally unique identifier; the identifier takes the form of 16 bytes of RAW data. The SYS_GUID function is based on the uuid-oss module to generate universally unique identifiers. The signature is:

```
SYS_GUID()
```

Example

The following example adds a column to the table EMP, inserts a unique identifier, and returns a 16-byte RAW value:

```
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

The USERENV function retrieves information about the current session. The signature is:

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

The parameter specifies a value to return from the current session. The possible parameter values are:

Parameter	Description
<i>ISDBA</i>	Returns <i>TRUE</i> if the current user has DBA privileges, otherwise <i>FALSE</i> .
<i>LANGUAGE</i>	The language, territory, and character set of the current session in the following format: <i>language_territory.characterset</i>
<i>LANG</i>	The ISO abbreviation for the language name, a short name for the existing <i>LANGUAGE</i> parameter.
<i>SID</i>	The current session identifier.
<i>TERMINAL</i>	The current session's operating system terminal identifier.

Examples

The following example returns the ISDBA parameter of the current session.

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

The following example returns the LANG parameter of the current session.

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

The following example returns the LANGUAGE parameter of the current session.

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

The following example returns the TERMINAL identifier.

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

The following example returns the SID number of the current session.

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

SYS_CONTEXT

The SYS_CONTEXT function returns the value of a **parameter** associated with the context namespace in the current session. The signature is:

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

OR

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

Parameters

namespace

namespace can be any named context; USERENV is a built-in context that shows information about the current session.

parameter

The `parameter` is a defined attribute of a namespace. The following table lists predefined attributes of the `USERENV` namespace:

Parameter	Description
ISDBA	Returns TRUE if the current user has DBA privileges, otherwise FALSE.
LANGUAGE	The language, territory, and character set of the current session in the following format: <code>language_territory.characterset</code>
LANG	The ISO abbreviation for the language name, a short name for the existing <code>LANGUAGE</code> parameter.
SID	The current session identifier.
TERMINAL	The current session's operating system terminal identifier.

Examples

In the following examples, the built-in `USERENV` namespace is used with the `SYS_CONTEXT` function.

The following example returns the `ISDBA` parameter of the current session.

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

The following example returns the `LANG` parameter of the current session.

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

The following example returns the `LANGUAGE` parameter of the current session.

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

The following example returns the `TERMINAL` identifier.

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

The following example returns the `SID` number of the current session.

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

1.3.14 Bitwise Functions

The following section describes bitwise functions BITAND and BITOR.

BITAND

The BITAND function performs a bitwise AND operation and returns a value based on the data type of input argument. The signature is:

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

The input data type of `expr1` and `expr2` is a NUMBER on which the bitwise AND operation will be performed. The BITAND compares each bit of `expr1` value to the corresponding bit of `exp2` using a bitwise AND operation and returns the value based on the data type of the input argument. If either argument to BITAND is NULL, the result is a NULL value.

Return Types

The BITAND function returns the same value as the data type of the input argument.

Examples

The following example performs an AND operation on the numbers 10 and 11.

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

BITOR

The BITOR function performs a bitwise OR operation and returns a value based on the data type of input argument. The signature is:

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

The BITOR compares each bit of `expr1` value to the corresponding bit of `exp2` using a bitwise OR operation and returns the value based on the data type of the input argument.

Return Types

The BITOR function returns the same value as the data type of the input argument. However, if all the values of a column are NULL, the BITOR returns NULL.

Examples

The following example performs an OR operation on the numbers 10 and 11.

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

BITWISE OPERATORS

The bitwise operators perform the same operation as BITAND and BITOR. The bitwise operators are:

- `&`: `(Bitwise AND)` Return the result of a bitwise `AND` operation performed on the input type.
- `|`: `(Bitwise OR)` Return the result of a bitwise `OR` operation performed on the input type.

Examples

The following example uses `&` operator and returns the output as BITAND of the input.

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

The following example uses the `|` operator and returns the output as BITOR of the input.

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

SYSTEM CATALOG TABLES

The following system catalog tables contain definitions of database objects. The layout of the system tables is subject to change; if you are writing an application that depends on information stored in the system tables, it would be prudent to use an existing catalog view, or create a catalog view to isolate the application from changes to the system table.

2.1 dual

`dual` is a single-row, single-column table that is provided for compatibility with Oracle databases only.

Column	Type	Modifiers	Description
<i>dummy</i>	<i>VARCHAR2(1)</i>		Provided for compatibility only.

2.2 edb_dir

The `edb_dir` table contains one row for each alias that points to a directory created with the `CREATE DIRECTORY` command. A directory is an alias for a pathname that allows a user limited access to the host file system.

You can use a directory to fence a user into a specific directory tree within the file system. For example, the `UTL_FILE` package offers functions that permit a user to read and write files and directories in the host file system, but only allows access to paths that the database administrator has granted access to via a `CREATE DIRECTORY` command.

Column	Type	Modifiers	Description
<i>dirname</i>	<i>*name*</i>	<i>not null</i>	The name of the alias.
<i>diowner</i>	<i>oid</i>	<i>not null</i>	The <i>OID</i> of the user that owns the alias.
<i>dirpath</i>	<i>text</i>		The directory name to which access is granted.
<i>diracl</i>	<i>aclitem[]</i>		The access control list that determines which users may access the alias.

2.3 edb_password_history

The `edb_password_history` table contains one row for each password change. The table is shared across all databases within a cluster.

Column	Type	References	Description
<i>passhistroleid</i>	<i>oid</i>	<i>pg_authid.oid</i>	The <i>ID</i> of a role.
<i>passhistpassword</i>	<i>text</i>		Role password in md5 encrypted form.
<i>passhistpasswordsetat</i>	<i>timestampz</i>		The time the password was set.

2.4 edb_policy

The `edb_policy` table contains one row for each policy.

Column	Type	Modifiers	Description
<i>polycyname</i>	<i>name</i>	<i>not null</i>	The policy name.
<i>policygroup</i>	<i>oid</i>	<i>not null</i>	Currently unused.
<i>policyobject</i>	<i>oid</i>	<i>not null</i>	The <i>OID</i> of the table secured by this policy (the <i>object_schema</i> plus the <i>object_name</i>).
<i>policykind</i>	<i>char</i>	<i>not null</i>	The kind of object secured by this policy: 'r' for a table, 'v' for a view, = for a synonym. Currently always 'r'.
<i>policyproc</i>	<i>oid</i>	<i>not null</i>	The <i>OID</i> of the policy function (<i>function_schema</i> plus <i>policy_function</i>).
<i>policyinsert</i>	<i>boolean</i>	<i>not null</i>	True if the policy is enforced by <i>INSERT</i> statements.
<i>policyselect</i>	<i>boolean</i>	<i>not null</i>	True if the policy is enforced by <i>SELECT</i> statements.
<i>policydelete</i>	<i>boolean</i>	<i>not null</i>	True if the policy is enforced by <i>DELETE</i> statements.
<i>policyupdate</i>	<i>boolean</i>	<i>not null</i>	True if the policy is enforced by <i>UPDATE</i> statements.
<i>policyindex</i>	<i>boolean</i>	<i>not null</i>	Currently unused.
<i>policyenabled</i>	<i>boolean</i>	<i>not null</i>	True if the policy is enabled.
<i>policyupdate-check</i>	<i>boolean</i>	<i>not null</i>	True if rows updated by an <i>UPDATE</i> statement must satisfy the policy.
<i>polycystatic</i>	<i>boolean</i>	<i>not null</i>	Currently unused.
<i>policytype</i>	<i>integer</i>	<i>not null</i>	Currently unused.
<i>policyopts</i>	<i>integer</i>	<i>not null</i>	Currently unused.
<i>policyseccols</i>	<i>int2vector</i>	<i>not null</i>	The column numbers for columns listed in <i>sec_relevant_cols</i> .

2.5 edb_profile

The `edb_profile` table stores information about the available profiles. `edb_profiles` is shared across all databases within a cluster.

Column	Type	References	Description
<i>oid</i>	<i>oid</i>		Row identifier (hidden attribute; must be explicitly selected).
<i>prfname</i>	<i>name</i>		The name of the profile.
<i>prffaileddlogi-nattempts</i>	<i>integer</i>		The number of failed login attempts allowed by the profile. -1 indicates that the value from the default profile should be used. -2 indicates no limit on failed login attempts.
<i>prfpassword-locktime</i>	<i>integer</i>		The password lock time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the account should be locked permanently.
<i>prfpasswordlife-time</i>	<i>integer</i>		The password life time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the password never expires.
<i>prfpassword-gracetime</i>	<i>integer</i>		The password grace time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the password never expires.
<i>prfpassword-reuse-time</i>	<i>integer</i>		The number of seconds that a user must wait before reusing a password. -1 indicates that the value from the default profile should be used. -2 indicates that the old passwords can never be reused.
<i>prfpassword-reuse-max</i>	<i>integer</i>		The number of password changes that have to occur before a password can be reused. -1 indicates that the value from the default profile should be used. -2 indicates that the old passwords can never be reused.
<i>prfpasswordallowhashed</i>	<i>integer</i>		The password allow hashed parameter specifies whether an encrypted password to be allowed for use or not. The possible values can be <i>true/on/yes/1</i> , <i>false/off/no/0</i> , and <i>DEFAULT</i> .
<i>prfpasswordverifyfuncdb</i>	<i>oid</i>	<i>pg_database.oid</i>	The <i>OID</i> of the database in which the password verify function exists.
<i>prfpasswordverifyfunc</i>	<i>oid</i>	<i>pg_proc.oid</i>	The <i>OID</i> of the password verify function associated with the profile.

2.6 edb_variable

The `edb_variable` table contains one row for each package level variable (each variable declared within a package).

Column	Type	Modifiers	Description
<i>varname</i>	<i>*name*</i>	<i>not null</i>	The name of the variable.
<i>varpackage</i>	<i>oid</i>	<i>not null</i>	The <i>OID</i> of the <i>pg_namespace</i> row that stores the package.
<i>vartype</i>	<i>oid</i>	<i>not null</i>	The <i>OID</i> of the <i>pg_type</i> row that defines the type of the variable.
<i>varaccess</i>	<i>*char*</i>	<i>not null</i>	+ if the variable is visible outside of the package. - if the variable is only visible within the package. Note: Public variables are declared within the package header; private variables are declared within the package body.
<i>varsrc</i>	<i>text</i>		Contains the source of the variable declaration, including any default value expressions for the variable.
<i>vartseq</i>	<i>smallint</i>	<i>not null</i>	The order in which the variable was declared in the package.

2.7 pg_synonym

The `pg_synonym` table contains one row for each synonym created with the `CREATE SYNONYM` command or `CREATE PUBLIC SYNONYM` command.

Column	Type	Modifiers	Description
<i>synname</i>	<i>*name*</i>	<i>not null</i>	The name of the synonym.
<i>synnamespace</i>	<i>oid</i>	<i>not null</i>	Replaces <i>synowner</i> . Contains the <i>OID</i> of the <i>pg_namespace</i> row where the synonym is stored.
<i>synowner</i>	<i>oid</i>	<i>not null</i>	The <i>OID</i> of the user that owns the synonym.
<i>synobjschema</i>	<i>*name*</i>	<i>not null</i>	The schema in which the referenced object is defined.
<i>synobjname</i>	<i>*name*</i>	<i>not null</i>	The name of the referenced object.
<i>synlink</i>	<i>text</i>		The (optional) name of the database link in which the referenced object is defined.

2.8 product_component_version

The `product_component_version` table contains information about feature compatibility; an application can query this table at installation or run time to verify that features used by the application are available with this deployment.

Column	Type	Description
<i>product</i>	<i>character varying (74)</i>	The name of the product.
<i>version</i>	<i>character varying (74)</i>	The version number of the product.
<i>status</i>	<i>character varying (74)</i>	The status of the release.