EDB Postgres™ Advanced Server

Release 13

Database Compatibility for Oracle® Developers Reference Guide

Oct 20, 2020

Contents

CHAPTER 1

Introduction

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 View 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/edb-docs

CHAPTER 2

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

2.1 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.

2.1.1 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*.

2.1.2 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.

2.1.3 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.

2.1.3.1 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 `(")`.

2.1.3.2 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.

2.1.3.3 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
-----------+-----------------
 Morgan    | {Code,Teach}
 Kolk      | {Code}
 Scott     | {}
(3 rows)
```

2.1.4 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 */.

2.2 Data Types

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

Name	Alias	Description
BLOB	LONG RAW, RAW(*n*), BYTEA	Binary data
BOOLEAN		Logical Boolean (true/false)
CHAR [(n)]	CHARACTER [(n)]	Fixed-length character string of n characters
CLOB	LONG, LONG VARCHAR	Long character string
DATE	TIMESTAMP	Date and time to the second
DOUBLE PRECISION	FLOAT, FLOAT(25) – FLOAT(53)	Double precision floating-point number
INTEGER	INT, BINARY_INTEGER, PLS_INTEGER	Signed four-byte integer
NUMBER	DEC, DECIMAL, NUMERIC	Exact numeric with optional decimal places
NUMBER(p [, s])	DEC(p [, s]), DECIMAL(p [, s]), NUMERIC(p [, s])	Exact numeric of maximum precision, p, and optional scale, s
REAL	FLOAT(1) – FLOAT(24)	Single precision floating-point number
TIMESTAMP [(p)]		Date and time with optional, fractional second precision, p
TIMESTAMP [(p)] WITH TIME ZONE		Date and time with optional, fractional second precision, p, and with time zone
VARCHAR2(n)	CHAR VARYING(n), CHARACTER VARYING(n), VARCHAR(n)	Variable-length character string with a maximum length of n characters
XMLTYPE		XML data

2.2.1 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
BINARY_INTEGER	4 bytes	Signed integer, Alias for INTEGER	-2,147,483,648 to +2,147,483,647
DOUBLE PRECISION	8 bytes	Variable-precision, inexact	15 decimal digits precision
INTEGER	4 bytes	Usual choice for integer	-2,147,483,648 to +2,147,483,647
NUMBER	Variable	User-specified precision, exact	Up to 1000 digits of precision
NUMBER(p [, s])	Variable	Exact numeric of maximum precision, p, and optional scale, s	Up to 1000 digits of precision
PLS_INTEGER	4 bytes	Signed integer, Alias for INTEGER	-2,147,483,648 to +2,147,483,647
REAL	4 bytes	Variable-precision, inexact	6 decimal digits precision
ROWID	8 bytes	Signed 8 bit integer.	-9223372036854775808 to 9223372036854775807

The following sections describe the types in detail.

2.2.1.1 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.

2.2.1.2 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.

2.2.1.3 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 1E-37 to 1E+37 with a precision of at least 6 decimal digits. The DOUBLE PRECISION type typically has a range of around 1E-307 to 1E+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.

2.2.2 Character Types

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

Name	Description
CHAR[(n)]	Fixed-length character string, blank-padded to the size specified by *n*
CLOB	Large variable-length up to 1 GB
LONG	Variable unlimited length.
NVARCHAR(n)	Variable-length national character string, with limit.
NVARCHAR2(n)	Variable-length national character string, with limit.
STRING	Alias for VARCHAR2.
VARCHAR(n)	Variable-length character string, with limit (considered deprecated, but supported for compatibility)
VARCHAR2(n)	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).
>
> `VARCHAR, VARCHAR2, NVARCHAR and NVARCHAR2`
>
> 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 that 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

2.2.3 Binary Data

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

Name	Storage Size	Description
BINARY	The length of the binary string.	Fixed-length binary string, with a length between 1 and 8300.
BLOB	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.
VARBINARY	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 characters 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.

2.2.4 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
DATE	8 bytes	Date and time	4713 BC	5874897 AD	1 second
INTERVAL DAY TO SECOND [(p)]	12 bytes	Period of time	-178000000 years	178000000 years	1 microsecond / 14 digits
INTERVAL YEAR TO MONTH	12 bytes	Period of time	-178000000 years	178000000 years	1 microsecond / 14 digits
TIMESTAMP [(p)]	8 bytes	Date and time	4713 BC	5874897 AD	1 microsecond
TIMESTAMP [(p)] WITH TIME ZONE	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.

2.2.4.1 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
YEAR	Integer value (positive or negative)
MONTH	0 through 11
DAY	Integer value (positive or negative)
HOUR	0 through 23
MINUTE	0 through 59
SECOND	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.

2.2.4.2 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:

```
type 'value'
```

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

`value` is a date/time text string.

2.2.4.2.1 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.

2.2.4.2.2 Times

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

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

2.2.4.2.3 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
```

2.2.4.3 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	31-DEC-05 07:37:16
ISO 8601/SQL standard	1997-12-17 07:37:16

2.2.4.4 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.

2.2.5 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
BOOLEAN	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`.

2.2.6 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;

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

2.3 Functions and Operators

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

2.3.1 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.

2.3.2 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
!=	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.

2.3.3 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
+	Addition	2 + 3	5
–	Subtraction	2 – 3	-1
*	Multiplication	2 * 3	6
/	Division (See the following note.)	4 / 2	2
**	Exponentiation Operator	2 ** 3	8

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
------------
```

(continues on next page)

(continued from previous page)

```
 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
ABS(x)	Same as x	Absolute value	ABS(-17.4)	17.4
CEIL(DOUBLE PRECISION or NUMBER)	Same as input	Smallest integer not less than argument	CEIL(-42.8)	-42
EXP(DOUBLE PRECISION or NUMBER)	Same as input	Exponential	EXP(1.0)	2.7182818284590452
FLOOR(DOUBLE PRECISION or NUMBER)	Same as input	Largest integer not greater than argument	FLOOR(-42.8)	43
LN(DOUBLE PRECISION or NUMBER)	Same as input	Natural logarithm	LN(2.0)	0.6931471805599453
LOG(b NUMBER, x NUMBER)	NUMBER	Logarithm to base b	LOG(2.0, 64.0)	6.0000000000000000
MOD(y, x)	Same as argument types	Remainder of y/x	MOD(9, 4)	1

continues on next page

Table 1 – continued from previous page

NVL(x, y)	Same as argument types; where both arguments are of the same data type	If x is null, then NVL returns y	NVL(9, 0)	9
POWER(a DOUBLE PRECISION, b DOUBLE PRECISION)	DOUBLE PRECISION	a raised to the power of b	POWER(9.0, 3.0)	729.0000000000000000
POWER(a NUMBER, b NUMBER)	NUMBER	a raised to the power of b	POWER(9.0, 3.0)	729.0000000000000000
ROUND(DOUBLE PRECISION or NUMBER)	Same as input	Round to nearest integer	ROUND(42.4)	42
ROUND(v NUMBER, s INTEGER)	NUMBER	Round to s decimal places	ROUND(42.4382, 2)	42.44
SIGN(DOUBLE PRECISION or NUMBER)	Same as input	Sign of the argument (-1, 0, +1)	SIGN(-8.4)	-1
SQRT(DOUBLE PRECISION or NUMBER)	Same as input	Square root	SQRT(2.0)	1.414213562373095
TRUNC(DOUBLE PRECISION or NUMBER)	Same as input	Truncate toward zero	TRUNC(42.8)	42
TRUNC(v NUMBER, s INTEGER)	NUMBER	Truncate to s decimal places	TRUNC(42.4382, 2)	42.43
WIDTH_BUCKET(op NUMBER, b1 NUMBER, b2 NUMBER, count INTEGER)	INTEGER	Return the bucket to which op would be assigned in an equidepth histogram with count buckets, in the range b1 to b2	WIDTH_BUCKET(5.35, 0.024, 10.06, 5)	3

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

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

2.3.4 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.

Function	Return Type	Description	Example	Result
string ‖ string	CLOB	String concatenation	‘Enterprise’ ‖ ‘DB’	EnterpriseDB
CONCAT(string, string)	CLOB	String concatenation	‘a’ ‖ ‘b’	ab
HEXTORAW(varchar2)	RAW	Converts a VARCHAR2 value to a RAW value	HEXTORAW(‘303132’)	‘012’
RAWTOHEX(raw)	VARCHAR2	Converts a RAW value to a HEXADECIMAL value	RAWTOHEX(‘012’)	‘303132’

continues on next page

Table 2 – continued from previous page

INSTR(string, set, [start [, occurrence]])	INTEGER	Finds the location of a set of characters in a string, starting at position start in the string, string, and looking for the first, second, third and so on occurrences of the set. Returns 0 if the set is not found.	INSTR('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PI',1,3)	30
INSTRB(string, set)	INTEGER	Returns the position of the set within the string. Returns 0 if set is not found.	INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS', 'PICK')	13
INSTRB(string, set, start)	INTEGER	Returns the position of the set within the string, beginning at start. Returns 0 if set is not found.	INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PICK', 14)	30

continues on next page

Table 2 – continued from previous page

INSTRB(string, set, start, occurrence)	INTEGER	Returns the position of the specified occurrence of set within the string, beginning at start. Returns 0 if set is not found.	INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PICK', 1, 2)	30
LOWER(string)	CLOB	Convert string to lower case	LOWER('TOM')	tom
SUBSTR(string, start [, count])	CLOB	Extract substring starting from start and going for count characters. If count is not specified, the string is clipped from the start till the end.	SUBSTR('This is a test',6,2)	is
SUBSTRB(string, start [, count])	CLOB	Same as SUBSTR except start and count are in number of bytes.	SUBSTRB('abc',3) (assuming a double-byte character set)	c
SUBSTR2(string, start [, count])	CLOB	Alias for SUBSTR.	SUBSTR2('This is a test',6,2)	is
SUBSTR2(string, start [, count])	CLOB	Alias for SUBSTRB.	SUBSTR2('abc',3) (assuming a double-byte character set)	c

continues on next page

Table 2 – continued from previous page

SUBSTR4(string, start [, count])	CLOB	Alias for SUBSTR.	SUBSTR4('This is a test',6,2)	is
SUBSTR4(string, start [, count])	CLOB	Alias for SUBSTRB.	SUBSTR4('abc',3) (assuming a double-byte character set)	c
SUBSTRC(string, start [, count])	CLOB	Alias for SUBSTR.	SUBSTRC('This is a test',6,2)	is
SUBSTRC(string, start [, count])	CLOB	Alias for SUBSTRB.	SUBSTRC('abc',3) (assuming a double-byte character set)	c
TRIM([LEADING | TRAILING | BOTH] [characters] FROM string)	CLOB	Remove the longest string containing only the characters (a space by default) from the start/end/both ends of the string.	TRIM(BOTH 'x' FROM 'xTomxx')	Tom
LTRIM(string [, set])	CLOB	Removes all the characters specified in set from the left of a given string. If set is not specified, a blank space is used as default.	LTRIM('abcdefghi', 'abc')	defghi

continues on next page

Table 2 – continued from previous page

RTRIM(string [, set])	CLOB	Removes all the characters specified in set from the right of a given string. If set is not specified, a blank space is used as default.	RTRIM('abcdefghi', 'ghi')	abcdef
UPPER(string)	CLOB	Convert string to upper case	UPPER('tom')	TOM

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
ASCII(string)	INTEGER	ASCII code of the first byte of the argument	ASCII('x')	120
CHR(INTEGER)	CLOB	Character with the given ASCII code	CHR(65)	A
DECODE(expr, expr1a, expr1b [, expr2a, expr2b]... [, default])	Same as argument types of expr1b, expr2b,..., default	Finds first match of expr with expr1a, expr2a, etc. When match found, returns corresponding parameter pair, expr1b, expr2b, etc. If no match found, returns default. If no match found and default not specified, returns null.	DECODE(3, 1,'One', 2,'Two', 3,'Three', 'Not found')	Three

continues on next page

Table 3 – continued from previous page

INITCAP(string)	CLOB	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.	INITCAP('hi THOMAS')	Hi Thomas
LENGTH	INTEGER	Returns the number of characters in a string value.	LENGTH('Côte d''Azur')	11
LENGTHC	INTEGER	This function is identical in functionality to LENGTH; the function name is supported for compatibility.	LENGTHC('Côte d''Azur')	11
LENGTH2	INTEGER	This function is identical in functionality to LENGTH; the function name is supported for compatibility.	LENGTH2('Côte d''Azur')	11
LENGTH4	INTEGER	This function is identical in functionality to LENGTH; the function name is supported for compatibility.	LENGTH4('Côte d''Azur')	11
LENGTHB	INTEGER	Returns the number of bytes required to hold the given value.	LENGTHB('Côte d''Azur')	12
LPAD(string, length INTEGER [, fill])	CLOB	Fill up string to size, length by prepending the characters, fill (a space by default). If string is already longer than length then it is truncated (on the right).	LPAD('hi', 5, 'xy')	xyxhi

continues on next page

Table 3 – continued from previous page

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

2.3.4.1 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
character	text	pg_catalog.text	implicit in expressions	function
character varying	text	–	implicit in expressions	binary-coercible
“char”	text	pg_catalog.text	implicit in expressions	function
name	text	pg_catalog.text	implicit in expressions	function
pg_node_tree	text	–	implicit in expressions	binary-coercible
pg_ndistinct	text	–	implicit in expressions	input/output function
pg_dependencies	text	–	implicit in expressions	input/output function
integer	text	–	implicit in expressions	input/output function
smallint	text	–	implicit in expressions	input/output function
oid	text	–	implicit in expressions	input/output function
date	text	–	implicit in expressions	input/output function
double precision	text	–	implicit in expressions	input/output function
real	text	–	implicit in expressions	input/output function
time with time zone	text	–	implicit in expressions	input/output function
time without time zone	text	–	implicit in expressions	input/output function
timestamp with time zone	text	–	implicit in expressions	input/output function
interval	text	–	implicit in expressions	input/output function
bigint	text	–	implicit in expressions	input/output function
numeric	text	–	implicit in expressions	input/output function
timestamp without time zone	text	–	implicit in expressions	input/output function
record	text	–	implicit in expressions	input/output function
boolean	text	pg_catalog.text	implicit in expressions	function
bytea	text	–	implicit in expressions	input/output function

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.

2.3.4.2 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-ossp module to generate universally unique identifiers. The synopsis 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)
```

2.3.5 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

2.3.5.1 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
--------------
```

(continues on next page)

(continued from previous page)

```
             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.

2.3.5.2 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.

2.3.5.3 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.

2.3.6 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.

2.3.7 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
TO_CHAR(DATE [, format])	VARCHAR2	Convert a date/time to a string with output, format. If omitted default format is DD-MON-YY.	TO_CHAR(SYSDATE, 'MM/DD/YYYY HH12:MI:SS AM')	07/25/2007 09:43:02 AM
TO_CHAR(TIMESTAMP [, format])	VARCHAR2	Convert a timestamp to a string with output, format. If omitted default format is DD-MON-YY.	TO_CHAR (CURRENT_TIMESTAMP, 'MM/DD/YYYY HH12:MI:SS AM')	08/13/2015 08:55:22 PM
TO_CHAR(INTEGER [, format])	VARCHAR2	Convert an integer to a string with output, format	TO_CHAR(2412, '999,999S')	2,412+

continues on next page

Table 5 – continued from previous page

TO_CHAR(NUMBER [, format])	VARCHAR2	Convert a decimal number to a string with output, format	TO_CHAR(10125.35, '999,999.99')	10,125.35
TO_CHAR(DOUBLE PRECISION, format)	VARCHAR2	Convert a floating-point number to a string with output, format	TO_CHAR (CAST(123.5282 AS REAL), '999.99')	123.53
TO_DATE(string [, format])	DATE	Convert a date formatted string to a DATE data type	TO_DATE('2007-07-04 13:39:10', 'YYYY-MM-DD HH24:MI:SS')	04-JUL-07
TO_NUMBER(string [, format])	NUMBER	Convert a number formatted string to a NUMBER data type	TO_NUMBER('2,412- ', '999,999S')	-2412
TO_TIMESTAMP(string, format)	TIMESTAMPTZ	Convert a timestamp formatted string to a TIMESTAMPTZ data type	TO_TIMESTAMP('05 Dec 2000 08:30:25 pm', 'DD Mon YYYY hh12:mi:ss pm')	05-DEC-00 20:30:25 +05:30

continues on next page

Table 5 – continued from previous page

TO_TIMESTAMP_TZ(string, format)	TIMESTAMPTZ	Convert a timestamp formatted string to a TIMESTAMPTZ data type	TO_TIMESTAMP_TZ (‘2003/12/13 10:13:18 -8:00’, ‘YYYY/MM/DD HH:MI:SS TZH:TZM’)	13-DEC-03 23:43:18 +05:30

2.3.7.1 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
HH	Hour of day (01-12)

continues on next page

Table 6 – continued from previous page

HH12	Hour of day (01-12)
HH24	Hour of day (00-23)
MI	Minute (00-59)
SS	Second (00-59)
SSSSS	Seconds past midnight (0-86399)
FFn	Fractional seconds where n is an optional integer from 1 to 9 for the number of digits to return. If omitted, the default is 6.
AM or A.M. or PM or P.M.	Meridian indicator (uppercase)
am or a.m. or pm or p.m.	Meridian indicator (lowercase)
Y,YYY	Year (4 and more digits) with comma
YEAR	Year (spelled out)
SYEAR	Year (spelled out) (BC dates prefixed by a minus sign)
YYYY	Year (4 and more digits)
SYYYY	Year (4 and more digits) (BC dates prefixed by a minus sign)
YYY	Last 3 digits of year
YY	Last 2 digits of year
Y	Last digit of year
IYYY	ISO year (4 and more digits)
IYY	Last 3 digits of ISO year
IY	Last 2 digits of ISO year
I	Last 1 digit of ISO year
BC or B.C. or AD or A.D.	Era indicator (uppercase)
bc or b.c. or ad or a.d.	Era indicator (lowercase)
MONTH	Full uppercase month name
Month	Full mixed-case month name
month	Full lowercase month name
MON	Abbreviated uppercase month name (3 chars in English, localized lengths vary)
Mon	Abbreviated mixed-case month name (3 chars in English, localized lengths vary)
mon	Abbreviated lowercase month name (3 chars in English, localized lengths vary)
MM	Month number (01-12)
DAY	Full uppercase day name
Day	Full mixed-case day name
day	Full lowercase day name
DY	Abbreviated uppercase day name (3 chars in English, localized lengths vary)
Dy	Abbreviated mixed-case day name (3 chars in English, localized lengths vary)
dy	Abbreviated lowercase day name (3 chars in English, localized lengths vary)
DDD	Day of year (001-366)
DD	Day of month (01-31)

continues on next page

Table 6 – continued from previous page

D	Day of week (1-7; Sunday is 1)
W	Week of month (1-5) (The first week starts on the first day of the month)
WW	Week number of year (1-53) (The first week starts on the first day of the year)
IW	ISO week number of year; the first Thursday of the new year is in week 1
CC	Century (2 digits); the 21st century starts on 2001-01-01
SCC	Same as CC except BC dates are prefixed by a minus sign
J	Julian Day (days since January 1, 4712 BC)
Q	Quarter
RM	Month in Roman numerals (I-XII; I=January) (uppercase)
rm	Month in Roman numerals (i-xii; i=January) (lowercase)
RR	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.
RRRR	Only affects TO_DATE 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.
TZH	Time-zone hours
TZM	Time-zone minutes

2.3.7.1.1 Date and Time Modifiers

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

Modifier	Description	Example
FM prefix	Fill mode (suppress padding blanks and zeros)	FMMonth
TH suffix	Uppercase ordinal number suffix	DDTH
th suffix	Lowercase ordinal number suffix	DDth
FX prefix	Fixed format global option (see usage notes)	FX Month DD Day
SP suffix	Spell mode	DDSP

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, timestamptz, 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
TO_CHAR(CURRENT_TIMESTAMP, 'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'
TO_CHAR(CURRENT_TIMESTAMP, 'FM-Day, FMDD HH12:MI:SS')	'Tuesday, 6 05:39:18'
TO_CHAR(-0.1, '99.99')	' -.10'
TO_CHAR(-0.1, 'FM9.99')	'-.1'
TO_CHAR(0.1, '0.9')	' 0.1'
TO_CHAR(12, '9990999.9')	' 0012.0'
TO_CHAR(12, 'FM9990999.9')	'0012.'
TO_CHAR(485, '999')	' 485'
TO_CHAR(-485, '999')	'-485'
TO_CHAR(1485, '9,999')	' 1,485'
TO_CHAR(1485, '9G999')	' 1,485'
TO_CHAR(148.5, '999.999')	' 148.500'
TO_CHAR(148.5, 'FM999.999')	'148.5'
TO_CHAR(148.5, 'FM999.990')	'148.500'
TO_CHAR(148.5, '999D999')	' 148.500'
TO_CHAR(3148.5, '9G999D999')	' 3,148.500'
TO_CHAR(-485, '999S')	'485-'
TO_CHAR(-485, '999MI')	'485-'
TO_CHAR(485, '999MI')	'485 '
TO_CHAR(485, 'FM999MI')	'485'
TO_CHAR(-485, '999PR')	'<485>'
TO_CHAR(485, 'L999')	'$ 485'
TO_CHAR(485, 'RN')	' CDLXXXV'
TO_CHAR(485, 'FMRN')	'CDLXXXV'
TO_CHAR(5.2, 'FMRN')	'V'
TO_CHAR(12, '99V999')	' 12000'

continues on next page

Table 7 – continued from previous page

TO_CHAR(12.4, '99V999')	' 12400'
TO_CHAR(12.45, '99V9')	' 125'

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

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

2.3.7.2 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
```

2.3.7.3 TO_NUMBER

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

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

Note that 9 results in a value with the same number of digits as there are 9s. If a digit is not available, the server ignores the corresponding 9s. 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
TO_NUMBER('-65', 'S99')	' -65'
TO_NUMBER('$65', 'L99')	' 65'
TO_NUMBER('9678584', '9999999')	' 9678584'
TO_NUMBER('123,456,789', '999,999,999')	' 123456789'
TO_NUMBER('1210.73', '9999.99')	' 1210.73'
TO_NUMBER('1210.73')	' 1210.73'
TO_NUMBER('0101.010','FM99999999.99999')	' 101.010'

2.3.7.3.1 Numeric Modifiers

The following table shows the modifier pattern for numeric formatting:

Pattern	Description	Example
FM prefix	fill mode (suppress trailing zeroes and padding blanks)	FM99.99

2.3.8 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 (+)	DATE '2001-09-28' + 7	05-OCT-01 00:00:00
plus (+)	TIMESTAMP '2001-09-28 13:30:00' + 3	01-OCT-01 13:30:00
minus (-)	DATE '2001-10-01' – 7	24-SEP-01 00:00:00
minus (-)	TIMESTAMP '2001-09-28 13:30:00' - 3	25-SEP-01 13:30:00
minus (-)	TIMESTAMP '2001-09-29 03:00:00' - TIMESTAMP '2001-09-27 12:00:00'	@ 1 day 15 hours

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
ADD_MONTHS (DATE, NUMBER)	DATE	Add months to a date	ADD_MONTHS ('28-FEB-97', ,3.8	31-MAY-97 00:00:00
CURRENT_DATE	DATE	Current date	CURRENT_DATE	04-JUL-07
CURRENT_TIMESTAMP	TIMESTAMP	Returns the current date and time	CURRENT_TIMESTAMP	04-JUL-07 15:33:23.484
EXTRACT(field FROM TIMESTAMP)	DOUBLE PRECISION	Get subfield	EXTRACT(hour FROM TIMESTAMP '2001-02-16 20:38:40')	20

continues on next page

Table 8 – continued from previous page

LAST_DAY(DATE)	DATE	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.	LAST_DAY('14-APR-98')	30-APR-98 00:00:00
LOCALTIMESTAMP [(precision)]	TIMESTAMP	Current date and time (start of current transaction	LOCALTIMESTAMP	04-JUL-07 15:33:23.484
MONTHS_BETWEEN (DATE, DATE)	NUMBER	Number of months between two dates	MONTHS_BETWEEN ('28-FEB-07', '30-NOV-06')	3
NEXT_DAY(DATE, dayofweek)	DATE	Date falling on dayofweek following specified date	NEXT_DAY('16 (-APR-07','FRI')	20-APR-07 00:00:00

continues on next page

Table 8 – continued from previous page

NEW_TIME(DATE, VARCHAR, VARCHAR)	DATE	Converts a date and time to an alternate time zone	NEW_TIME (TO_DATE ‘2005/05/29 01:45’, ‘AST’, ‘PST’)	2005/05/29 21:45:00
NUMTODSINTERVAL(NUMBER, INTERVAL)	INTERVAL	Converts a number to a specified day or second interval	SELECT numtodsinterval(100, ‘hour’);	4 days 04:00:00
NUMTOYMINTERVAL(NUMBER, INTERVAL)	INTERVAL	Converts a number to a specified year or month interval.	SELECT numtoymin terval(100, ‘month’);	8 years 4 mons
ROUND(DATE [, format])	DATE	Date rounded according to format	ROUND(TO_DATE(‘29-MAY-05’),’MON’)	01-JUN-05 00:00:00
SYS_EXTRACT_UTC(TIMESTAMP WITH TIME ZONE)	TIMESTAMP	Returns Coordinated Universal Time	SYS_EXTRACT_UTC(CAST (‘24-MAR-11 12:30:00PM -04:00’ AS TIMESTAMP WITH TIME ZONE))	24-MAR-11 16:30:00
SYSDATE	DATE	Returns current date and time	SYSDATE	01-AUG-12 11:12:34
SYS TIMESTAMP()	TIMESTAMP	Returns current date and time	SYSTIMESTAMP	01-AUG-12 11:11:23. 665229 -07:00
TRUNC(DATE [format])	DATE	Truncate according to format	TRUNC(TO_DATE (‘29-MAY-05’), ‘MON’)	01-MAY-05 00:00:00

2.3.8.1 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)
```

2.3.8.2 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
```

(continues on next page)

(continued from previous page)

```
-----------
 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.

2.3.8.3 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
```

(continues on next page)

(continued from previous page)

```
-----------
      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)
```

HOUR

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)
```

2.3.8.4 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)
```

2.3.8.5 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)
```

2.3.8.6 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
AST	UTC+4	Atlantic Standard Time
ADT	UTC+3	Atlantic Daylight Time
BST	UTC+11	Bering Standard Time
BDT	UTC+10	Bering Daylight Time
CST	UTC+6	Central Standard Time
CDT	UTC+5	Central Daylight Time
EST	UTC+5	Eastern Standard Time
EDT	UTC+4	Eastern Daylight Time
GMT	UTC	Greenwich Mean Time
HST	UTC+10	Alaska-Hawaii Standard Time
HDT	UTC+9	Alaska-Hawaii Daylight Time
MST	UTC+7	Mountain Standard Time
MDT	UTC+6	Mountain Daylight Time
NST	UTC+3:30	Newfoundland Standard Time
PST	UTC+8	Pacific Standard Time
PDT	UTC+7	Pacific Daylight Time
YST	UTC+9	Yukon Standard Time
YDT	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)
```

2.3.8.7 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`, `HOUR`, `MINUTE`, and `SECOND`.

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

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

2.3.8.8 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)
```

2.3.8.9 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, cc01 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)
SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y	Returns January 1, yyyy where yyyy is rounded to the nearest year; rounds down on June 30, rounds up on July 1
IYYY, IYY, IY, 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
Q	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
MONTH, MON, MM, RM	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
WW	Round to the nearest date that corresponds to the same day of the week as the first day of the year
IW	Round to the nearest date that corresponds to the same day of the week as the first day of the ISO year
W	Round to the nearest date that corresponds to the same day of the week as the first day of the month
DDD, DD, J	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
DAY, DY, D	Rounds to the nearest Sunday
HH, HH12, HH24	Round to the nearest hour
MI	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;
```

(continues on next page)

(continued from previous page)

```
      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
--------------------
 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;
```

(continues on next page)

(continued from previous page)

```
      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
--------------------
 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)
```

(continues on next page)

(continued from previous page)

```
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;

       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)
```

2.3.8.10 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)
```

2.3.8.11 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
CC, SCC	Returns January 1, cc 01 where cc is first 2 digits of the given year
SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y	Returns January 1, yyyy where yyyy is the given year
IYYY, IYY, IY, I	Returns the start date of the ISO year containing the given date
Q	Returns the first day of the quarter containing the given date
MONTH, MON, MM, RM	Returns the first day of the specified month
WW	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
IW	Returns the start of the ISO week containing the given date
W	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
DDD, DD, J	Returns the start of the day for the given date
DAY, DY, D	Returns the start of the week (Sunday) containing the given date
HH, HH12, HH24	Returns the start of the hour
MI	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;
```

(continues on next page)

(continued from previous page)

```
   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
--------------------
```

(continues on next page)

(continued from previous page)

```
 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
--------------------
 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
--------------------
```

(continues on next page)

(continued from previous page)

```
 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)
```

2.3.9 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.

2.3.10 Conditional Expressions

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

2.3.10.1 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
```

(continues on next page)

(continued from previous page)

```
  [ 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;
```

2.3.10.2 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.

2.3.10.3 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.

2.3.10.4 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 employees 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)
```

2.3.10.5 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)
```

2.3.10.6 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.

2.3.11 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
AVG (expression)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	NUMBER for any integer type, DOUBLE PRECISION for a floating-point argument, otherwise the same as the argument data type	The average (arithmetic mean) of all input values
COUNT(*)		BIGINT	Number of input rows
COUNT (expression)	Any	BIGINT	Number of input rows for which the value of expression is not null
MAX (expression)	Any numeric, string, date/time, or bytea type	Same as argument type	Maximum value of expression across all input values
MEDIAN (expression)	BIGINT, DOUBLE PRECISION, INTEGER, INTERVAL, NUMERIC, REAL, SMALLINT, TIMESTAMP, TIMESTAMPZ	NUMBER for any integer type, DOUBLE PRECISION for a floating-point argument, otherwise the same as the argument data type	Identifies the middle value of an expression
MIN (expression)	Any numeric, string, date/time, or bytea type	Same as argument type	Minimum value of expression across all input values
SUM (expression)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	BIGINT for SMALLINT or INTEGER arguments, NUMBER for BIGINT arguments, DOUBLE PRECISION 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
CORR(Y, X)	DOUBLE PRECISION	DOUBLE PRECISION	Correlation coefficient
COVAR_POP(Y, X)	DOUBLE PRECISION	DOUBLE PRECISION	Population covariance
COVAR_SAMP(Y, X)	DOUBLE PRECISION	DOUBLE PRECISION	Sample covariance
REGR_AVGX(Y, X)	DOUBLE PRECISION	DOUBLE PRECISION	Average of the independent variable (sum(X) / N)
REGR_AVGY(Y, X)	DOUBLE PRECISION	DOUBLE PRECISION	Average of the dependent variable (sum(Y) / N)
REGR_COUNT(Y, X)	DOUBLE PRECISION	DOUBLE PRECISION	Number of input rows in which both expressions are nonnull
REGR_INTERCEPT(Y, X)	DOUBLE PRECISION	DOUBLE PRECISION	y-intercept of the least-squares-fit linear equation determined by the (X, Y) pairs
REGR_R2(Y, X)	DOUBLE PRECISION	DOUBLE PRECISION	Square of the correlation coefficient
REGR_SLOPE(Y, X)	DOUBLE PRECISION	DOUBLE PRECISION	Slope of the least-squares-fit linear equation determined by the (X, Y) pairs
REGR_SXX(Y, X)	DOUBLE PRECISION	DOUBLE PRECISION	Sum (X^2) – sum (X) ^{2} / N ("sum of squares" of the independent variable)
REGR_SXY(Y, X)	DOUBLE PRECISION	DOUBLE PRECISION	Sum (X*Y) – sum (X) * sum (Y) / N ("sum of products" of independent times dependent variable)

continues on next page

Table 10 – continued from previous page

REGR_SYY(Y, X)	DOUBLE PRECISION	DOUBLE PRECISION	Sum (Y^2) – sum (Y) ^{2} / N ("sum of squares" of the dependent variable)
STDDEV(expression)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Historic alias for STDDEV_SAMP
STDDEV_POP(expression)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Population standard deviation of the input values
STDDEV_SAMP(expression)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Sample standard deviation of the input values
VARIANCE(expression)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Historical alias for VAR_SAMP
VAR_POP(expression)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Population variance of the input values (square of the population standard deviation)
VAR_SAMP(expression)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Sample variance of the input values (square of the sample standard deviation)

2.3.11.1 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)
```

2.3.11.2 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
BIGINT	NUMERIC
FLOAT, DOUBLE PRECISION	DOUBLE PRECISION
INTEGER	NUMERIC
INTERVAL	INTERVAL
NUMERIC	NUMERIC
REAL	REAL
SMALLINT	NUMERIC
TIMESTAMP	TIMESTAMP
TIMESTAMPTZ	TIMESTAMPTZ

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;
```

(continues on next page)

(continued from previous page)

```
 median | percentile_cont
--------+-----------------
   1250 |            1250
(1 row)
```

2.3.11.3 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
```

(continues on next page)

(continued from previous page)

```
 7654 | MARTIN| SALESMAN| 7698 | 28-SEP-81 00:00:00| 1250.00| 1400.00|   30
(5 rows)

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

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)
```

2.3.12 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.

2.3.12.1 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)
```

2.3.12.2 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.

2.3.12.3 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.

2.3.12.4 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.

2.3.12.5 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.

CHAPTER 3

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.

3.1 dual

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

Column	Type	Modifiers	Description
dummy	VARCHAR2(1)		Provided for compatibility only.

3.2 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
dirname	"name"	not null	The name of the alias.
dirowner	oid	not null	The OID of the user that owns the alias.
dirpath	text		The directory name to which access is granted.
diracl	aclitem[]		The access control list that determines which users may access the alias.

3.3 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
passhistroleid	oid	pg_authid.oid	The ID of a role.
passhistpassword	text		Role password in md5 encrypted form.
passhistpasswordsetat	timestamptz		The time the password was set.

3.4 edb_policy

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

Column	Type	Modifiers	Description
policyname	name	not null	The policy name.
policygroup	oid	not null	Currently unused.
policyobject	oid	not null	The OID of the table secured by this policy (the object_schema plus the object_name).
policykind	char	not null	The kind of object secured by this policy: 'r' for a table 'v' for a view = for a synonym Currently always 'r'.
policyproc	oid	not null	The OID of the policy function (function_schema plus policy_function).
policyinsert	boolean	not null	True if the policy is enforced by INSERT statements.
policyselect	boolean	not null	True if the policy is enforced by SELECT statements.

continues on next page

Table 1 – continued from previous page

Column	Type	Modifiers	Description
policydelete	boolean	not null	True if the policy is enforced by DELETE statements.
policyupdate	boolean	not null	True if the policy is enforced by UPDATE statements.
policyindex	boolean	not null	Currently unused.
policyenabled	boolean	not null	True if the policy is enabled.
policyupdatecheck	boolean	not null	True if rows updated by an UPDATE statement must satisfy the policy.
policystatic	boolean	not null	Currently unused.
policytype	integer	not null	Currently unused.
policyopts	integer	not null	Currently unused.
policyseccols	int2vector	not null	The column numbers for columns listed in sec_relevant_cols.

3.5 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
oid	oid		Row identifier (hidden attribute; must be explicitly selected).
prfname	name		The name of the profile.
prffailedloginattempts	integer		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.
prfpasswordlocktime	integer		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.

continues on next page

Table 2 – continued from previous page

Column	Type	References	Description
prfpasswordlifetime	integer		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.
prfpasswordgracetime	integer		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.
prfpasswordreusetime	integer		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.
prfpasswordreusemax	integer		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.
prfpasswordallowhashed	integer		The password allow hashed parameter specifies whether an encrypted password to be allowed for use or not. The possible values can be true/on/yes/1, false/off/no/0, and DEFAULT.
prfpasswordverifyfuncdb	oid	pg_database.oid	The OID of the database in which the password verify function exists.
prfpasswordverifyfunc	oid	pg_proc.oid	The OID of the password verify function associated with the profile.

3.6 edb_variable

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

Column	Type	Modifiers	Description
varname	"name"	not null	The name of the variable.
varpackage	oid	not null	The OID of the pg_namespace row that stores the package.
vartype	oid	not null	The OID of the pg_type row that defines the type of the variable.
varaccess	"char"	not null	+ 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.
varsrc	text		Contains the source of the variable declaration, including any default value expressions for the variable.
varseq	smallint	not null	The order in which the variable was declared in the package.

3.7 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
synname	“name”	not null	The name of the synonym.
synnamespace	oid	not null	Replaces synowner. Contains the OID of the pg_namespace row where the synonym is stored
synowner	oid	not null	The OID of the user that owns the synonym.
synobjschema	“name”	not null	The schema in which the referenced object is defined.
synobjname	“name”	not null	The name of the referenced object.
synlink	text		The (optional) name of the database link in which the referenced object is defined.

3.8 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
product	character varying (74)	The name of the product.
version	character varying (74)	The version number of the product.
status	character varying (74)	The status of the release.

CHAPTER 4

Conclusion

EnterpriseDB® Corporation

34 Crosby Drive, Suite 201, Bedford, MA 01730, USA

T +1 781 357 3390 F +1 978 467 1307 E

info@enterprisedb.com

www.enterprisedb.com

Index