Table of Contents

Orafce Documentation

Orafce - Oracle’s compatibility functions and packages

This documentation describes the environment settings and functionality offered for features that are compatible with Oracle databases.

Chapter 1 Overview

Features compatible with Oracle databases are provided. These features enable you to easily migrate to PostgreSQL and reduce the costs of reconfiguring applications.

The table below lists features compatible with Oracle databases.

1.1 Features compatible with Oracle databases

Data type

Item	Overview
VARCHAR2	Variable-length character data type
NVARCHAR2	Variable-length national character data type
DATE	Data type that stores date and time

SQL Queries

Item	Overview
DUAL table	Table provided by the system

SQL Functions

- Mathematical functions

Item	Overview
BITAND	Performs a bitwise AND operation
COSH	Calculates the hyperbolic cosine of a number
SINH	Calculates the hyperbolic sine of a number
TANH	Calculates the hyperbolic tangent of a number

- String functions

Item	Overview

Item	Overview
INSTR	Returns the position of a substring in a string
LENGTH	Returns the length of a string in number of characters
LENGTHB	Returns the length of a string in number of bytes
LPAD	Left-pads a string to a specified length with a sequence of characters
LTRIM	Removes the specified characters from the beginning of a string
NLSSORT	Returns a byte string used to sort strings in linguistic sort sequence based on locale
REGEXP_COUNT	searches a string for a regular expression, and returns a count of the matches
REGEXP_INSTR	returns the beginning or ending position within the string where the match for a pattern was located
REGEXP_LIKE	condition in the WHERE clause of a query, causing the query to return rows that match the given pattern
REGEXP_SUBSTR	returns the string that matches the pattern specified in the call to the function
REGEXP_REPLACE	replace substring(s) matching a POSIX regular expression
RPAD	Right-pads a string to a specified length with a sequence of characters
RTRIM	Removes the specified characters from the end of a string
SUBSTR	Extracts part of a string using characters to specify position and length
SUBSTRB	Extracts part of a string using bytes to specify position and length

- Date/time functions

Item	Overview
ADD_MONTHS	Adds months to a date
DBTIMEZONE	Returns the value of the database time zone
LAST_DAY	Returns the last day of the month in which the specified date falls
MONTHS_BETWEEN	Returns the number of months between two dates
NEXT_DAY	Returns the date of the first instance of a particular day of the week that follows the specified date
ROUND	Rounds a date
SESSIONTIMEZONE	Returns the time zone of the session
SYSDATE	Returns the system date
TRUNC	Truncates a date

- Data type formatting functions

Item	Overview
TO_CHAR	Converts a value to a string
TO_DATE	Converts a string to a date in accordance with the specified format
TO_MULTI_BYTE	Converts a single-byte string to a multibyte string
TO_NUMBER	Converts a value to a number in accordance with the specified format
TO_SINGLE_BYTE	Converts a multibyte string to a single-byte string

- Conditional expressions

Item	Overview
DECODE	Compares values, and if they match, returns a corresponding value
GREATEST	Returns the greatest of the list of one or more expressions
LEAST	Returns the least of the list of one or more expressions
LNNVL	Evaluates if a value is false or unknown
NANVL	Returns a substitute value when a value is not a number (NaN)
NVL	Returns a substitute value when a value is NULL
NVL2	Returns a substitute value based on whether a value is NULL or not NULL

- Aggregate functions

Item	Overview
LISTAGG	Returns a concatenated, delimited list of string values
MEDIAN	Calculates the median of a set of values

- Functions that return internal information

Item	Overview
DUMP	Returns internal information of a value

SQL Operators

Item	Overview
Datetime operator	Datetime operator for the DATE type

Packages

Item	Overview
DBMS_ALERT	Sends alerts to multiple sessions
DBMS_ASSERT	Validates the properties of an input value
DBMS_OUTPUT	Sends messages to clients
DBMS_PIPE	Creates a pipe for inter-session communication
DBMS_RANDOM	Generates random numbers
DBMS_UTILITY	Provides various utilities
UTL_FILE	Enables text file operations

Chapter 2 Notes on Using orafce

Orafce is defined as user-defined functions in the “public” schema created by default when database clusters are created, so they can be available for all users without the need for special settings. For this reason, ensure that “public” (without the double quotation marks) is included in the list of schema search paths specified in the search_path parameter.

The following features provided by orafce are implemented in PostgreSQL and orafce using different external specifications. In the default configuration of PostgreSQL, the standard features of PostgreSQL take precedence.

Features implemented in PostgreSQL and orafce using different external specifications

- Data type

Item	Standard feature of PostgreSQL	Compatibility feature added by orafce
DATE	Stores date only.	Stores date and time.

- Function

Item	Standard feature of PostgreSQL	Compatibility feature added by orafce
LENGTH	If the string is CHAR type, trailing spaces are not included in the length.	If the string is CHAR type, trailing spaces are included in the length.
SUBSTR	If 0 or a negative value is specified for the start position, simply subtracting 1 from the start position, the position will be shifted to the left, from where extraction will start.	- If 0 is specified for the start position, extraction will start from the beginning of the string. - If a negative value is specified for the start position, extraction will start from the position counted from the end of the string.

Item	Standard feature of PostgreSQL	Compatibility feature added by orafce
LPAD RPAD	- If the string is CHAR type, trailing spaces are removed and then the value is padded. - The result length is handled as a number of characters.	- If the string is CHAR type, the value is padded without removing trailing spaces. - The result length is based on the width of the displayed string. Therefore, fullwidth characters are handled using a width of 2, and halfwidth characters are handled using a width of 1.
LTRIM RTRIM BTRIM (*1)	If the string is CHAR type, trailing spaces are removed and then the value is removed.	If the string is CHAR type, the value is removed without removing trailing spaces.
TO_DATE	The data type of the return value is DATE.	The data type of the return value is TIMESTAMP.

*1: BTRIM does not exist for Oracle databases, however, an external specification different to PostgreSQL is implemented in orafce to align with the behavior of the TRIM functions.

Also, the following features cannot be used in the default configuration of PostgreSQL.

Features that cannot be used in the default configuration of PostgreSQL

- Function

Feature
SYSDATE
DBTIMEZONE
SESSIONTIMEZONE
TO_CHAR (date/time value)

- Operator

Feature
Datetime operator
To use these features, set “oracle” and “pg_catalog” in the “search_path” parameter of postgresql.conf. You must specify “oracle” before “pg_catalog” when doing this.

```
search_path = '"$user", public, oracle, pg_catalog'
```

Information

- The search_path parameter specifies the order in which schemas are searched. Each feature compatible with Oracle databases is defined in the oracle schema.
- It is recommended to set search_path in postgresql.conf. In this case, it will be effective for each instance.
- The configuration of search_path can be done at the user level or at the database level. Setting examples are shown below.
- If the standard features of PostgreSQL take precedence, and features that cannot be used with the default configuration of PostgreSQL are not required, it is not necessary to change the settings of search_path.

- Example of setting at the user level
  - This can be set by executing an SQL command. In this example, user1 is used as the username.

    ```
    ALTER USER user1 SET search_path = "$user",public,oracle,pg_catalog;
    ```

- Example of setting at the database level
  - This can be set by executing an SQL command. In this example, db1 is used as the database name.
    You must specify “oracle” before “pg_catalog”.

    ```
    ALTER DATABASE db1 SET search_path = "$user",public,oracle,pg_catalog;
    ```

See

- Refer to “Server Administration” > “Client Connection Defaults” > “Statement Behavior” in the PostgreSQL Documentation for information on search_path.
- Refer to “Reference” > “SQL Commands” in the PostgreSQL Documentation for information on ALTER USER and ALTER DATABASE.

Chapter 3 Data Types

The following data types are supported:

- VARCHAR2
- NVARCHAR2
- DATE

3.1 VARCHAR2

Syntax

Specify the VARCHAR2 type as follows.

Data type syntax	Explanation
VARCHAR2(*len*)	String with a variable length up to *len* characters For *len*, specify an integer greater than 0. If *len* is omitted, the string can be any length.

General rules

- VARCHAR2 is a character data type. Specify the number of bytes for the length.
- Strings are of variable length. The specified value will be stored as is. The upper limit for this data type is approximately one gigabyte.

Note

The VARCHAR2 type does not support collating sequences. Therefore, the following error occurs when a collating sequence like that of an ORDER BY clause is required. At this time, the following HINT will prompt to use a COLLATE clause, however, because collating sequences are not supported, it is not possible to use this clause.

```
ERROR:  could not determine which collation to use for string comparison
HINT:  Use the COLLATE clause to set the collation explicitly.
```

If the error shown above is displayed, explicitly cast the column to VARCHAR or TEXT type.

3.2 NVARCHAR2

Syntax

Specify the NVARCHAR2 type as follows.

Data type syntax	Explanation
NVARCHAR2(*len*)	National character string with a variable length up to *len* characters. For *len*, specify an integer greater than 0. If *len* is omitted, the string can be any length.

General rules

- NVARCHAR2 is a national character data type. Specify the number of characters for the length.
- Strings are of variable length. The specified value will be stored as is. The upper limit for this data type is approximately one gigabyte.

Note

The NVARCHAR2 type does not support collating sequences. Therefore, the following error occurs when a collating sequence like that of an ORDER BY clause is required. At this time, the following HINT will prompt to use a COLLATE clause, however, because collating sequences are not supported, it is not possible to use this clause.

```
ERROR:  could not determine which collation to use for string comparison
HINT:  Use the COLLATE clause to set the collation explicitly.
```

If the error shown above is displayed, explicitly cast the column to NCHAR VARYING or TEXT type.

3.3 DATE

Syntax

Specify the DATE type as follows.

Data type syntax	Explanation
DATE	Stores date and time

General rules

- DATE is a date/time data type.
- Date and time are stored in DATE. The time zone is not stored.

Note

If the DATE type of orafce is used in DDL statements such as table definitions, always set search_path before executing a DDL statement. Even if search_path is changed after definition, the data type will be the DATE type of PostgreSQL.

Information

The DATE type of orafce is equivalent to the TIMESTAMP type of PostgreSQL. Therefore, of the existing functions of PostgreSQL, functions for which the data type of the argument is TIMESTAMP can be used.

Chapter 4 Queries

The following queries are supported:

- DUAL Table

4.1 DUAL Table

DUAL table is a virtual table provided by the system. Use when executing SQL where access to a base table is not required, such as when performing tests to get result expressions such as functions and operators.

Example

In the following example, the current system date is returned.

```
SELECT  CURRENT_DATE  "date" FROM DUAL;
   date
------------
 2013-05-14
(1 row)
```

Chapter 5 SQL Function Reference

5.1 Mathematical Functions

The following mathematical functions are supported:

- BITAND
- COSH
- SINH
- TANH

5.1.1 BITAND

Description

Performs a bitwise AND operation.

Syntax

General rules

- BITAND performs an AND operation on each bit of two integers, and returns the result.
- Specify integer type values.
- The data type of the return value is BIGINT.

Example

In the following example, the result of the AND operation on numeric literals 5 and 3 is returned.

```
SELECT BITAND(5,3) FROM DUAL;
 bitand
-------
      1
(1 row)
```

5.1.2 COSH

Description

Calculates the hyperbolic cosine of a number.

Syntax

General rules

- COSH returns the hyperbolic cosine of the specified number.
- The number must be a numeric data type.
- The data type of the return value is DOUBLE PRECISION.

Example

In the following example, the hyperbolic cosine of the numeric literal 2.236 is returned.

```
SELECT COSH(2.236) FROM DUAL;
    cosh
-----------------
4.7313591000247
(1 row)
```

5.1.3 SINH

Description

Calculates the hyperbolic sine of a number.

Syntax

General rules

- SINH returns the hyperbolic sine of the specified number.
- The number must be a numeric data type.
- The data type of the return value is DOUBLE PRECISION.

Example

In the following example, the hyperbolic sine of the numeric literal 1.414 is returned.

```
SELECT SINH(1.414) FROM DUAL;
    sinh
-----------------
1.93460168824956
(1 row)
```

5.1.4 TANH

Description

Calculates the hyperbolic tangent of a number.

Syntax

General rules

- TANH returns the hyperbolic tangent of the specified number.
- The number must be a numeric data type.
- The data type of the return value is DOUBLE PRECISION.

Example

In the following example, the hyperbolic tangent of the numeric literal 3 is returned.

```
SELECT TANH(3) FROM DUAL;
    tanh
-----------------
0.995054753686731
(1 row)
```

5.2 String Functions

The following string functions are supported:

- BTRIM
- INSTR
- LENGTH
- LENGTHB
- LPAD
- LTRIM
- NLSSORT
- REGEXP_COUNT
- REGEXP_INSTR
- REGEXP_LIKE
- REGEXP_SUBSTR
- RPAD
- RTRIM
- SUBSTR
- SUBSTRB

5.2.1 BTRIM

Description

Removes the specified characters from the beginning and end of a string.

Syntax

General rules

- BTRIM returns a string with *trimChars* removed from the beginning and end of string *str*.
- If multiple trim characters are specified, all characters matching the trim characters are removed. If *trimChars* is omitted, all leading and trailing halfwidth spaces are removed.
- The data type of the return value is TEXT.

Note

- BTRIM does not exist for Oracle databases.
- The CHAR type specification for BTRIM uses orafce for its behavior, which is different to that of BTRIM of PostgreSQL. The search_path parameter must be modified for it to behave the same as the specification described above.

Information

The general rule for BTRIM of PostgreSQL is as follows:

- If the string is CHAR type, trailing spaces are removed and then the trim characters are removed.

See

- Refer to “Notes on Using orafce” for information on how to edit search_path.
- Refer to “The SQL Language” > “Functions and Operators” > “String Functions and Operators” in the PostgreSQL Documentation for information on BTRIM.

Example

In the following example, a string that has had “a” removed from both ends of “aabcaba” is returned.

```
SELECT BTRIM('aabcaba','a') FROM DUAL;
 btrim
-------
 bcab
(1 row)
```

5.2.2 INSTR

Description

Returns the position of a substring in a string.

Syntax

General rules

- INSTR searches for substring *str2* in string *str1* and returns the position (in characters) in *str1* of the first character of the occurrence.
- The search starts from the specified start position *startPos* in *str1*.
- When *startPos* is 0 or negative, the start position will be the specified number of characters from the left of the end of *str1*, and INSTR will search backward from that point.
- If the start position is not specified, the search will be performed from the beginning of *str1*.
- If *occurrences* is specified, the position in *str1* of the nth occurrence of *str2* is returned. Only positive numbers can be specified.
- If *occurrences* is not specified, the start position of the first occurrence that is found is returned.
- If *str2* is not found in *str1*, 0 is returned.
- For *startPos* and *occurrences*, specify a SMALLINT or INTEGER type.
- The data type of the return value is INTEGER.

Example

In the following example, characters “BC” are found in string “ABCACBCAAC”, and the position of those characters is returned.

```
SELECT INSTR('ABCACBCAAC','BC') FROM DUAL;
 instr
-------
     2
(1 row)

SELECT INSTR('ABCACBCAAC','BC',-1,2) FROM DUAL;
 instr
-------
     2
(1 row)
```

5.2.3 LENGTH

Description

Returns the length of a string in number of characters.

Syntax

General rules

- LENGTH returns the number of characters in string *str*.
- If the string is CHAR type, trailing spaces are included in the length.
- The data type of the return value is INTEGER.

Note

The LENGTH specification above uses orafce for its behavior, which is different to that of LENGTH of PostgreSQL. The search_path parameter must be modified for it to behave according to the orafce specification.

Information

The general rule for LENGTH of PostgreSQL is as follows:

- If the string is CHAR type, trailing spaces are not included in the length.

See

- Refer to “Notes on Using orafce” for information on how to edit search_path.
- Refer to “The SQL Language” > “Functions and Operators” > “String Functions and Operators” in the PostgreSQL Documentation for information on LENGTH.

Example

In the following example, the number of characters in column col2 (defined using CHAR(10)) in table t1 is returned.

```
SELECT col2,LENGTH(col2) FROM t1 WHERE col1 = '1001';
   col2    | length
-----------+--------
 AAAAA     |     10
(1 row)
```

5.2.4 LENGTHB

Description

Returns the length of a string in number of bytes.

Syntax

General rules

- LENGTHB returns the number of bytes in string *str*.
- If the string is CHAR type, trailing spaces are included in the length.
- The data type of the return value is INTEGER.

Example

In the following example, the number of bytes in column col2 (defined using CHAR(10)) in table t1 is returned. Note that, in the second SELECT statement, each character in “*” has a length of 3 bytes, for a total of 9 bytes, and 7 bytes are added for the 7

trailing spaces. This gives a result of 16 bytes.

```
SELECT col2,LENGTHB(col2) FROM t1 WHERE col1 = '1001';
    col2     | lengthb
-------------+---------
 AAAAA       |      10
(1 row)

SELECT col2,LENGTHB(col2) FROM t1 WHERE col1 = '1004';
    col2     | lengthb
-------------+---------
 ***         |      16
(1 row)
```

5.2.5 LPAD

Description

Left-pads a string to a specified length with a sequence of characters.

Syntax

General rules

- LPAD returns the result after repeatedly padding the beginning of string*str* with padding characters *paddingStr* until the string reaches length *len*.
- If the string is CHAR type, the padding characters are added to the string without removing trailing spaces.
- In the resultant string, fullwidth characters are recognized as having a length of 2, and halfwidth characters having a length of 1. If a fullwidth character cannot be included in the resultant string because there is only space available for one halfwidth character, the string is padded with a single-byte space.
- The data type of the return value is TEXT.

Note

The LPAD specification above uses orafce for its behavior, which is different to that of LPAD of PostgreSQL. The search_path parameter must be modified for it to behave according to the orafce specification.

Information

The general rules for LPAD of PostgreSQL are as follows:

- If the string is CHAR type, trailing spaces are removed and then the padding characters are added to the string.
- The result length is the number of characters.

See

- Refer to “Notes on Using orafce” for information on how to edit search_path.
- Refer to “The SQL Language” > “Functions and Operators” > “String Functions and Operators” in the PostgreSQL Documentation for information on LPAD.

Example

In the following example, a 10-character string that has been formed by left-padding the string “abc” with “a” is returned.

```
SELECT LPAD('abc',10,'a') FROM DUAL;
   lpad
------------
 aaaaaaaabc
(1 row)
```

5.2.6 LTRIM

Description

Removes the specified characters from the beginning of a string.

Syntax

General rules

- LTRIM returns a string with *trimChars* removed from the beginning of string *str*.
- If multiple trim characters are specified, all characters matching the trim characters are removed. If *trimChars* is omitted, all leading halfwidth spaces are removed.
- The data type of the return value is TEXT.

Note

The LTRIM specification above uses orafce for its behavior, which is different to that of LTRIM of PostgreSQL. The search_path parameter must be modified for it to behave according to the orafce specification.

Information

The general rule for LTRIM of PostgreSQL is as follows:

- If the string is CHAR type, trailing spaces are removed and then the trim characters are removed.

See

- Refer to “Notes on Using orafce” for information on how to edit search_path.
- Refer to “The SQL Language” > “Functions and Operators” > “String Functions and Operators” in the PostgreSQL Documentation for information on LTRIM.

Example

In the following example, a string that has had “ab” removed from the beginning of “aabcab” is returned.

```
SELECT LTRIM('aabcab','ab') FROM DUAL;
 ltrim
-------
 cab
(1 row)
```

5.2.7 NLSSORT

Description

Returns a byte string that denotes the lexical order of the locale (COLLATE).

Syntax

General rules

- NLSSORT is used for comparing and sorting in the collating sequence of a locale (COLLATE) that differs from the default locale.
- Values that can be specified for the locale differ according to the operating system of the database server.
- If the locale is omitted, it is necessary to use set_nls_sort to set the locale in advance. To set the locale using set_nls_sort, execute a SELECT statement.

Example of setting set_nls_sort using a SELECT statement

```
SELECT set_nls_sort('en_US.UTF8');
```

- The data type of the return value is BYTEA.

Note

If specifying locale encoding, ensure it matches the database encoding.

See

Refer to “Server Administration” > “Localization” > “Locale Support” in the PostgreSQL Documentation for information on the locales that can be specified.

Example

[Composition of table (t3)]

col1	col2
1001	aabcababc
2001	abcdef
3001	aacbaab

In the following example, the result of sorting column col2 in table t3 by “da_DK.UTF8” is returned.

```
SELECT col1,col2 FROM t3 ORDER BY NLSSORT(col2,'da_DK.UTF8');
 col1 |    col2
------+------------
 2001 | abcdef
 1001 | aabcababc
 3001 | aacbaab
(3 row)
```

5.2.8 REGEXP_COUNT

Description

Searches a string for a regular expression, and returns a count of the matches.

General rules

- REGEXP_COUNT returns the number of times *pattern* occurs in a source *string*. It returns an integer indicating the number of occurrences of *pattern*. If no match is found, then the function returns 0.
- The search starts from the specified start position *startPos* in *string*, default starts from the beginning of *string*.
- *startPos* is a positive integer, negative values to search from the end of *string* are not allowed.

- *flags* is a character expression that lets you change the default matching behavior of the function. The value of *flags* can include one or more of the following characters:
  - 'i': case-insensitive matching.
  - 'c': case-sensitive and accent-sensitive matching.
  - 'n': the period (.) match the newline character. By default the period does not match the newline character.
  - 'm': treats the source string as multiple lines.
  - 'x': ignores whitespace characters. By default, whitespace characters match themselves. If you omit *flags*, then:
  - The default is case and accent sensitivity.
  - A period (.) does not match the newline character.
  - The source string is treated as a single line.

Example

```
SELECT REGEXP_COUNT('a'||CHR(10)||'d', 'a.d') FROM DUAL;
 regexp_count
--------------
            0
(1 row)

SELECT REGEXP_COUNT('a'||CHR(10)||'d', 'a.d', 1, 'm') FROM DUAL;
 regexp_count
--------------
            0
(1 row)

SELECT REGEXP_COUNT('a'||CHR(10)||'d', 'a.d', 1, 'n') FROM DUAL;
 regexp_count
--------------
            1
(1 row)

SELECT REGEXP_COUNT('a'||CHR(10)||'d', '^d$', 1, 'm') FROM DUAL;
 regexp_count
--------------
            1
(1 row)
```

5.2.9 REGEXP_INSTR

Description

Returns the beginning or ending position within the string where the match for a pattern was located.

General rules

- REGEXP_INSTR returns an integer indicating the beginning or ending position of the matched substring, depending on the value of the *return_opt* argument. If no match is found, then the function returns 0.
- The search starts from the specified start position *startPos* in *string*, default starts from the beginning of *string*.
- *startPos* is a positive integer, negative values to search from the end of *string* are not allowed.
- *occurrence* is a positive integer indicating which occurrence of *pattern* in *string* should be search for. The default is 1, meaning the first occurrence of *pattern* in *string*.
- *return_opt* lets you specify what should be returned in relation to the occurrence:
  - 0, the position of the first character of the occurrence is returned. This is the default.
  - 1, the position of the character following the occurrence is returned.
- *flags* is a character expression that lets you change the default matching behavior of the function. See REGEXP_COUNT for detailed information.
- For a *pattern* with capture group, *group* is a positive integer indicating which capture group in *pattern* shall be returned by the function. Capture groups can be nested, they are numbered in order in which their left parentheses appear in *pattern*.

If *group* is zero, then the position of the entire substring that matches the pattern is returned. If *group* value exceed the number of capture groups in *pattern*, the function returns zero. A null *group* value returns *NULL*. The default value for *group* is zero.

Example

```
SELECT REGEXP_INSTR('1234567890', '(123)(4(56)(78))') FROM DUAL;
 regexp_instr
--------------
            1
(1 row)

SELECT REGEXP_INSTR('1234567890', '(4(56)(78))', 3) FROM DUAL;
 regexp_instr
--------------
            4
(1 row)

SELECT REGEXP_INSTR('123 123456 1234567, 1234567 1234567 12', '[^ ]+', 1, 6) FROM DUAL;
 regexp_instr
--------------
           37

(1 row)

SELECT REGEXP_INSTR('199 Oretax Prayers, Riffles Stream, CA', '[S|R|P][[:alpha:]]{6}', 3, 2, 1) FROM DUAL;
 regexp_instr
--------------
           28
(1 row)
```

5.2.10 REGEXP_LIKE

Description

Condition in the WHERE clause of a query, causing the query to return rows that match the given pattern.

General rules

- REGEXP_LIKE is similar to the LIKE condition, except it performs regular expression matching instead of the simple pattern matching performed by LIKE.
- Returns a boolean, *true* when *pattern* match in *string*, *false* otherwise.
- *flags* is a character expression that lets you change the default matching behavior of the function. See REGEXP_COUNT for detailed information.

Example

```
SELECT REGEXP_LIKE('a'||CHR(10)||'d', 'a.d', 'm') FROM DUAL;
 regexp_like
-------------
 f
(1 row)

SELECT REGEXP_LIKE('a'||CHR(10)||'d', 'a.d', 'n') FROM DUAL;
 regexp_like
-------------
 t
(1 row)
```

5.2.11 REGEXP_SUBSTR

Description

Returns the string that matches the pattern specified in the call to the function.

General rules

- REGEXP_SUBSTR returns the matched substring resulting from matching a POSIX regular expression pattern to a string. If no match is found, then the function returns *NULL*.
- The search starts from the specified start position *startPos* in *string*, default starts from the beginning of *string*.
- *startPos* is a positive integer, negative values to search from the end of *string* are not allowed.
- *occurrence* is a positive integer indicating which occurrence of *pattern* in *string* should be search for. The default is 1, meaning the first occurrence of *pattern* in *string*.
- *flags* is a character expression that lets you change the default matching behavior of the function. See REGEXP_COUNT for detailed information.
- For a *pattern* with capture group, *group* is a positive integer indicating which capture group in *pattern* shall be returned by the function. Capture groups can be nested, they are numbered in order in which their left parentheses appear in *pattern*. If *group* is zero, then the position of the entire substring that matches the pattern is returned. If *group* value exceed the number of capture groups in *pattern*, the function returns *NULL*. A null *group* value returns *NULL*. The default value for *group* is zero.

Example

```
SELECT REGEXP_SUBSTR('number of your street, zipcode town, FR', ',[^,]+') FROM DUAL;
 regexp_substr
----------------
 , zipcode town
(1 row)

SELECT regexp_substr('number of your street, zipcode town, FR', ',[^,]+', 24) FROM DUAL;
 regexp_substr
---------------
 , FR
(1 row)

SELECT regexp_substr('number of your street, zipcode town, FR', ',[^,]+', 1, 2) FROM DUAL;
 regexp_substr
---------------
 , FR
(1 row)

SELECT regexp_substr('1234567890 1234567890', '(123)(4(56)(78))', 1, 1, 'i', 0) FROM DUAL;
 regexp_substr
---------------
 12345678
(1 row)
```

5.2.12 REGEXP_REPLACE

Description

Returns the string that matches the pattern specified in the call to the function.

General rules

- REGEXP_REPLACE returns a modified version of the source string where occurrences of a POSIX regular expression pattern found in the source string are replaced with the specified replacement string. If no match is found or the occurrence queried exceed the number of match, then the source string untouched is returned.
- The search and replacement starts from the specified start position *startPos* in *string*, default starts from the beginning of *string*.

- *startPos* is a positive integer, negative values to search from the end of*string* are not allowed.
- *occurrence* is a positive integer indicating which occurrence of*pattern* in *string* should be search for and replaced. The default is 0, meaning all occurrences of *pattern* in *string*.
- *flags* is a character expression that lets you change the default matching behavior of the function. See REGEXP_COUNT for detailed information.

Example

```
SELECT regexp_replace('512.123.4567 612.123.4567', '([[:digit:]]{3})\.([[:digit:]]{3})\.([[:digit:]]{4})', '(\1) \2-\3') FROM DUAL;
       regexp_replace
-------------------------------
 (512) 123-4567 (612) 123-4567
(1 row)

SELECT oracle.REGEXP_REPLACE('number   your     street,    zipcode  town, FR', '( ){2,}', ' ', 9);
            regexp_replace
---------------------------------------
 number   your street, zipcode town, FR
(1 row)

SELECT oracle.REGEXP_REPLACE('number   your     street,    zipcode  town, FR', '( ){2,}', ' ', 9, 2);
              regexp_replace
-----------------------------------------------
 number   your     street, zipcode  town, FR
(1 row)
```

5.2.13 RPAD

Description

Right-pads a string to a specified length with a sequence of characters.

Syntax

```
```

General rules

- RPAD returns the result after repeatedly padding the end of string*str* with padding characters *paddingStr* until the string reaches length *len*.
- If the string is CHAR type, the padding characters are added to the string without removing trailing spaces.
- In the resultant string, fullwidth characters are recognized as having a length of 2, and halfwidth characters having a length of 1. If a fullwidth character cannot be included in the resultant string because there is only space available for one halfwidth character, the string is padded with a single-byte space.
- The data type of the return value is TEXT.

Note

The RPAD specification above uses orafce for its behavior, which is different to that of RPAD of PostgreSQL. The search_path parameter must be modified for it to behave according to the orafce specification.

Information

The general rules for RPAD of PostgreSQL are as follows:

- If the string is CHAR type, trailing spaces are removed and then the padding characters are added to the string.
- The result length is the number of characters.

See

- Refer to “Notes on Using orafce” for information on how to edit search_path.
- Refer to “The SQL Language” > “Functions and Operators” > “String Functions and Operators” in the PostgreSQL Documentation for information on RPAD.

Example

In the following example, a 10-character string that has been formed by right-padding the string “abc” with “a” is returned.

```
SELECT RPAD('abc',10,'a') FROM DUAL;
   rpad
------------
 abcaaaaaaa
(1 row)
```

5.2.14 RTRIM

Description

Removes the specified characters from the end of a string.

Syntax

General rules

- RTRIM returns a string with *trimChars* removed from the end of string *str*.
- If multiple trim characters are specified, all characters matching the trim characters are removed. If *trimChars* is omitted, all trailing halfwidth spaces are removed.
- The data type of the return value is TEXT.

Note

The RTRIM specification above uses orafce for its behavior, which is different to that of RTRIM of PostgreSQL. The search_path parameter must be modified for it to behave the same as the orafce specification.

Information

The general rule for RTRIM of PostgreSQL is as follows:

- If the string is CHAR type, trailing spaces are removed and then the trim characters are removed.

See

- Refer to “Notes on Using orafce” for information on how to edit search_path.
- Refer to “The SQL Language” > “Functions and Operators” > “String Functions and Operators” in the PostgreSQL Documentation for information on RTRIM.

Example

In the following example, a string that has had “ab” removed from the end of “aabcab” is returned.

```
SELECT RTRIM('aabcab','ab') FROM DUAL;
 rtrim
-------
 aabc
(1 row)
```

5.2.15 SUBSTR

Description

Extracts part of a string using characters to specify position and length.

Syntax

General rules

- SUBSTR extracts and returns a substring of string *str*, beginning at position *startPos*, for number of characters *len*.
- When *startPos* is positive, it will be the number of characters from the beginning of the string.
- When *startPos* is 0, it will be treated as 1.
- When *startPos* is negative, it will be the number of characters from the end of the string.
- When *len* is not specified, all characters to the end of the string are returned. NULL is returned when *len* is less than 1.
- For *startPos* and *len*, specify an integer or NUMERIC type. If numbers including decimal places are specified, they are truncated to integers.
- The data type of the return value is TEXT.

Note

- There are two types of SUBSTR. One that behaves as described above and one that behaves the same as SUBSTRING. The search_path parameter must be modified for it to behave the same as the specification described above.
- If the change has not been implemented, SUBSTR is the same as SUBSTRING.

Information

The general rules for SUBSTRING are as follows:

- The start position will be from the beginning of the string, whether the start position is positive, 0, or negative.
- When *len* is not specified, all characters to the end of the string are returned.
- An empty string is returned if no string is extracted or *len* is less than 1.

See

Refer to “The SQL Language” > “Functions and Operators” > “String Functions and Operators” in the PostgreSQL Documentation for information on SUBSTRING.

Example

In the following example, part of the string “ABCDEFG” is extracted.

```
SELECT SUBSTR('ABCDEFG',3,4) "Substring" FROM DUAL;

 Substring
-----------
 CDEF
(1 row)

SELECT SUBSTR('ABCDEFG',-5,4) "Substring" FROM DUAL;

 Substring
-----------
 CDEF
(1 row)
```

5.2.16 SUBSTRB

Description

Extracts part of a string using bytes to specify position and length.

Syntax

General rules

- SUBSTRB extracts and returns a substring of string *str*, beginning at byte position *startPos*, for number of bytes *len*.
- When *startPos* is 0 or negative, extraction starts at the position found by subtracting 1 from the start position and shifting by that number of positions to the left.
- When *len* is not specified, all bytes to the end of the string are returned.
- An empty string is returned if no string is extracted or *len* is less than 1.
- For *startPos* and *len*, specify a SMALLINT or INTEGER type.
- The data type of the return value is VARCHAR2.

Note

The external specification of SUBSTRB is different to that of SUBSTR added by orafce, conforming with SUBSTRING of PostgreSQL.

Example

In the following example, part of the string “aaabbbccc” is extracted.

```
SELECT SUBSTRB('aaabbbccc',4,3) FROM DUAL;
 substrb
-----------
 bbb
(1 row)

SELECT SUBSTRB('aaabbbccc',-2,6) FROM DUAL;
 substrb
-----------
 aaa
(1 row)
```

5.3 Date/time Functions

The following date/time functions are supported:

- ADD_MONTHS
- DBTIMEZONE
- LAST_DAY
- MONTHS_BETWEEN
- NEXT_DAY
- ROUND
- SESSIONTIMEZONE
- SYSDATE
- TRUNC

Note

If the DATE type only is shown in the date/time functions, these functions can be used in both orafce and PostgreSQL.

5.3.1 ADD_MONTHS

Description

Adds months to a date.

Syntax

General rules

- ADD_MONTHS returns *date* plus *months*.
- For *date*, specify a DATE type.
- For *months*, specify a SMALLINT or INTEGER type.
- If a negative value is specified for *months*, the number of months is subtracted from the date.
- The data type of the return value is DATE.

Note

If using the DATE type of orafce, it is necessary to specify “oracle” for search_path in advance.

See

Refer to “Notes on Using orafce” for information on how to edit search_path.

Example

The example below shows the result of adding 3 months to the date May 1, 2016.

```
SELECT ADD_MONTHS(DATE'2016/05/01',3) FROM DUAL;
    add_months
---------------------
 2016-08-01 00:00:00
(1 row)
```

5.3.2 DBTIMEZONE

Description

Returns the value of the database time zone.

Syntax

General rules

- DBTIMEZONE returns the time zone value of the database.
- The data type of the return value is TEXT.

Note

- If using DBTIMEZONE, it is necessary to specify “oracle” for search_path in advance.
- The time zone of the database is set to “GMT” by default. To change the time zone, change the “orafce.timezone” parameter. An example using the SET statement is shown below.

Setting example of orafce.timezone using a SET statement

```
SET orafce.timezone = 'Japan';
```

- The orafce.timezone settings can be set using any of the methods for setting server parameters.
- If the SQL statement is executed with orafce.timezone set, the following message may be displayed, however, the parameter settings are enabled, so you can ignore this.

  ```
  WARNING:  unrecognized configuration parameter "orafce.timezone"
  ```

- The time zones that can be set in “orafce.timezone” are the same as for the “TimeZone” server parameter.

See

- Refer to “Notes on Using orafce” for information on how to edit search_path.
- Refer to “The SQL Language” > “Data Types” > “Date/Time Types” in the PostgreSQL Documentation for information on the time zone.

Example

In the following example, the DBTIMEZONE result is returned.

```
SELECT DBTIMEZONE() FROM DUAL;
 dbtimezone
------------
 GMT
(1 row)
```

5.3.3 LAST_DAY

Description

Returns the last day of the month in which the specified date falls.

Syntax

General rules

- LAST_DAY returns the last day of the month in which the specified date falls.
- For *date*, specify a DATE type.
- The data type of the return value is DATE.

Note

If using the DATE type of orafce, it is necessary to specify “oracle” for search_path in advance.

See

Refer to “Notes on Using orafce” for information on how to edit search_path.

Example

In the example below, the last date of “February 01, 2016” is returned.

```
SELECT LAST_DAY(DATE'2016/02/01') FROM DUAL;
    last_day
---------------------
 2016-02-29 00:00:00
(1 row)
```

5.3.4 MONTHS_BETWEEN

Description

Returns the number of months between two dates.

Syntax

General rules

- MONTHS_BETWEEN returns the difference in the number of months between*date1* and *date2*.
- For *date1* and *date2*, specify a DATE type.
- If *date2* is earlier than *date1*, the return value will be negative.
- If two dates fall on the same day, or each of the two dates are the last day of the month to which they belong, an integer is returned. If the days are different, one month is considered to be 31 days, and a value with the difference in the number of days divided by 31 added is returned.
- The data type of the return value is NUMERIC.

Note

If using the DATE type of orafce, it is necessary to specify “oracle” for search_path in advance.

See

Refer to “Notes on Using orafce” for information on how to edit search_path.

Example

In the following example, the difference between the months of March 15, 2016 and November 15, 2015 is returned.

```
SELECT MONTHS_BETWEEN(DATE'2016/03/15', DATE'2015/11/15') FROM DUAL;
 months_between
----------------
              4
(1 row)
```

5.3.5 NEXT_DAY

Description

Returns the date of the first instance of a particular day of the week that follows the specified date.

Syntax

General rules

- NEXT_DAY returns the date matching the first instance of*dayOfWk* that follows *date*.
- For *date*, specify a DATE type.
- Specify a numeric value or string indicating the day of the week.

Values that can be specified for the day

Setting example	Overview
1	1 (Sunday) to 7 (Saturday) can be specified
‘Sun’, or ‘Sunday’	English display of the day
‘*’	Japanese display of the day

- The data type of the return value is DATE.

Note

- If using the DATE type of orafce, it is necessary to specify “oracle” for search_path in advance.
- The ability to use Japanese for entering days is provided by the orafce proprietary specification. Japanese cannot be used for entering days when using date/time functions other than NEXT_DAY (such as TO_DATE).

See

Refer to “Notes on Using orafce” for information on how to edit search_path.

Example

In the example below, the date of the first Friday on or after “May 1, 2016” is returned.

```
SELECT NEXT_DAY(DATE'2016/05/01', 'Friday') FROM DUAL;
     next_day
---------------------
 2016-05-06 00:00:00
(1 row)
```

5.3.6 ROUND

Description

Rounds a date.

Syntax

General rules

- ROUND returns a date rounded to the unit specified by format model *fmt*.
- For *date*, specify a DATE or TIMESTAMP type.
- Specify the format model as a string.

Values that can be specified for the format model

Format model	Rounding unit
Y,YY,YYY,YYYY, SYYYY,YEAR,SYEAR	Year
I,IY,IYY,IYYY	Year (values including calendar weeks, in compliance with the ISO standard)
Q	Quarter
WW	Week (first day of the year)
IW	Week (Monday of that week)
W	Week (first weekday on which the first day of the month falls)
DAY,DY,D	Week (Sunday of that week)
MONTH,MON,MM,RM	Month
CC,SCC	Century
DDD,DD,J	Day
HH,HH12,HH24	Hour
MI	Minute

- If decimal places are rounded: for year, the boundary for rounding is July 1; for month, the day is 16; and for week, the weekday is Thursday.
- If *fmt* is omitted, the date is rounded by day.
- If the DATE type of PostgreSQL is specified for the date, that DATE type will be the data type of the return value. If the TIMESTAMP type is specified for the date, the data type will be TIMESTAMP WITH TIME ZONE, irrespective of whether a time zone is used.

Example

In the example below, the result of “June 20, 2016 18:00:00” rounded by Sunday of the week is returned.

```
SELECT ROUND(TIMESTAMP'2016/06/20 18:00:00','DAY') FROM DUAL;
        round
------------------------
 2016-06-19 00:00:00+09
(1 row)
```

5.3.7 SESSIONTIMEZONE

Description

Returns the time zone of the session.

Syntax

General rules

- SESSIONTIMEZONE returns the time zone value between sessions.

- The data type of the return value is TEXT.

Note

- If using SESSIONTIMEZONE, it is necessary to specify “oracle” for search_path in advance.
- The value returned by SESSIONTIMEZONE becomes the value set in the “TimeZone” server parameter.

See

Refer to “Notes on Using orafce” for information on how to edit search_path.

Example

In the following example, the time zone of the session is returned.

```
SELECT SESSIONTIMEZONE() FROM DUAL;
 sessiontimezone
-----------------
 Japan
(1 row)
```

5.3.8 SYSDATE

Description

Returns the system date.

Syntax

General rules

- SYSDATE returns the system date.
- The data type of the return value is the DATE type of orafce.

Note

- If using SYSDATE, it is necessary to specify “oracle” for search_path in advance.
- The date returned by SYSDATE depends on the time zone value of the orafce database.

See

- Refer to “Notes on Using orafce” for information on how to edit search_path.
- Refer to “DBTIMEZONE” for information on the time zone values of the database.
- Refer to “The SQL Language” > “Data Types” > “Date/Time Types” in the PostgreSQL Documentation for information on the time zone.

Example

In the following example, the system date is returned.

```
SELECT SYSDATE() FROM DUAL;
      sysdate
---------------------
 2016-06-22 08:06:51
(1 row)
```

5.3.9 TRUNC

Description

Truncates a date.

Syntax

General rules

- TRUNC returns a date truncated to the unit specified by format model*fmt*.
- For *date*, specify a DATE or TIMESTAMP type.
- Specify the format model as a string. The values that can be specified are the same as for ROUND.
- If *fmt* is omitted, the date is truncated by day.
- If the DATE type of PostgreSQL is specified for the date, that DATE type will be the data type of the return value. If the TIMESTAMP type is specified for the date, the data type will be TIMESTAMP WITH TIME ZONE, irrespective of whether a time zone is used.

See

Refer to “ROUND” for information on the values that can be specified for the format model.

Example

In the example below, the result of “August 10, 2016 15:30:00” truncated by the day is returned.

```
SELECT TRUNC(TIMESTAMP'2016/08/10 15:30:00','DDD') FROM DUAL;
        trunc
------------------------
 2016-08-10 00:00:00+09
(1 row)
```

5.4 Data Type Formatting Functions

The following data type formatting functions are supported:

- TO_CHAR
- TO_DATE
- TO_MULTI_BYTE
- TO_NUMBER
- TO_SINGLE_BYTE

5.4.1 TO_CHAR

Description

Converts a value to a string.

Syntax

General rules

- TO_CHAR converts the specified number or date/time value to a string.
- For *num*, specify a numeric data type.
- For *date*, specify a DATE or TIMESTAMP type. Also, you must set a date/time format for the orafce.nls_date_format variable in advance. A setting example using the SET statement is shown below. Setting example of orafce.nls_date_format using a SET statement

  ```
  SET orafce.nls_date_format = 'YYYY/MM/DD HH24:MI:SS';
  ```

- The data type of the return value is TEXT.

Note

- If using TO_CHAR for specifying date/time values, it is necessary to specify “oracle” for search_path in advance.
- The orafce.nls_date_format settings can be set using any of the methods for setting server parameters.
- If orafce.nls_date_format is set, the following message may be displayed when an SQL statement is executed, however, the parameter settings are enabled, so you can ignore this.

  ```
  WARNING:  unrecognized configuration parameter "orafce.nls_date_format"
  ```

See

- Refer to “Notes on Using orafce” for information on how to edit search_path.
- Refer to “Server Administration” > “Server Configuration” > “Setting Parameters” in the PostgreSQL Documentation for information on how to set the server parameters.

Example

In the following example, the numeric value “123.45” is returned as a string.

```
SELECT TO_CHAR(123.45) FROM DUAL;
 to_char
---------
 123.45
(1 row)
```

5.4.2 TO_DATE

Description

Converts a string to a date in accordance with the specified format.

Syntax

General rules

- TO_DATE converts string *str* to a date in accordance with the specified format*fmt*.
- Specify a string indicating the date/time.
- Specify the required date/time format. If omitted, the format specified in the oracle.nls_date_format variable is used. If the oracle.nls_date_format variable has not been set, the existing date/time input interpretation is used. A setting example using the SET statement is shown below.

Setting example of orafce.nls_date_format using a SET statement

```
SET orafce.nls_date_format = 'YYYY/MM/DD HH24:MI:SS';
```

- The data type of the return value is TIMESTAMP.

Note

- The above TO_DATE specification uses orafce for its behavior, which is different to that of TO_DATE of PostgreSQL. The search_path parameter must be modified for it to behave according to the orafce specification.
- The orafce.nls_date_format settings can be set using any of the methods for setting server parameters.
- If orafce.nls_date_format is set, the following message may be displayed when an SQL statement is executed, however, the parameter settings are enabled, so you can ignore this.

  ```
  WARNING:  unrecognized configuration parameter "orafce.nls_date_format"
  ```

Information

The general rule for TO_DATE for specifying the data type format of PostgreSQL is as follows:

- The data type of the return value is the DATE type of PostgreSQL.

See

- Refer to “Notes on Using orafce” for information on how to edit search_path.
- Refer to “The SQL Language” > “Functions and Operators” > “Data Type Formatting Functions” in the PostgreSQL Documentation for information on TO_DATE of PostgreSQL.
- Refer to “Server Administration” > “Server Configuration” > “Setting Parameters” in the PostgreSQL Documentation for information on how to set the server parameters.
- Refer to “Date/Time Support” > “Date/Time Input Interpretation” in the PostgreSQL Documentation for information on the interpretation of existing date/time input.

Example

In the following example, the string “2016/12/31” is converted to a date and returned.

```
SELECT TO_DATE('2016/12/31','YYYY/MM/DD') FROM DUAL;
      to_date
---------------------
 2016-12-31 00:00:00
(1 row)
```

5.4.3 TO_MULTI_BYTE

Description

Converts a single-byte string to a multibyte string.

Syntax

```
```

General rules

- TO_MULTI_BYTE converts halfwidth characters in string *str* to fullwidth characters, and returns the converted string.
- Only halfwidth alphanumeric characters, spaces and symbols can be converted.
- The data type of the return value is TEXT.

Example

In the following example, “abc123” is converted to fullwidth characters and returned.

```
SELECT TO_MULTI_BYTE('abc123') FROM DUAL;
 to_multi_byte
---------------
   ******
(1 row)
```

“******” is multibyte “abc123”.

5.4.4 TO_NUMBER

Description

Converts a value to a number in accordance with the specified format.

Syntax

General rules

- TO_NUMBER converts the specified value to a numeric value in accordance with the specified format *fmt*.
- For *num*, specify a numeric data type.
- For *str*, specify a string indicating the numeric value. Numeric values must comprise only of convertible characters.
- Specify the required numeric data format. The specified numeric value is handled as is as a data type expression.
- The data type of the return value is NUMERIC.

See

Refer to “The SQL Language” > “Functions and Operators” > “Data Type Formatting Functions” in the PostgreSQL Documentation for information on numeric value formats.

Example

In the following example, the numeric literal “-130.5” is converted to a numeric value and returned.

```
SELECT TO_NUMBER(-130.5) FROM DUAL;
 to_number
-----------
    -130.5
(1 row)
```

5.4.5 TO_SINGLE_BYTE

Description

Converts a multibyte string to a single-byte string.

Syntax

General rules

- TO_SINGLE_BYTE converts fullwidth characters in string *str* to halfwidth characters, and returns the converted string.
- Only fullwidth alphanumeric characters, spaces and symbols that can be displayed in halfwidth can be converted.

- The data type of the return value is TEXT.

Example

In the following example, “******” is converted to halfwidth characters and returned. “******” is multibyte “xyz999”.

```
SELECT TO_SINGLE_BYTE('******') FROM DUAL;
 to_single_byte
----------------
 xyz999
(1 row)
```

5.5 Conditional Expressions

The following functions for making comparisons are supported:

- DECODE
- GREATEST
- LEAST
- LNNVL
- NANVL
- NVL
- NVL2

5.5.1 DECODE

Description

Compares values and if they match, returns a corresponding value.

Syntax

General rules

- DECODE compares values of the value expression to be converted and the search values one by one. If the values match, a corresponding result value is returned. If no values match, the default value is returned if it has been specified. A NULL value is returned if a default value has not been specified.
- If the same search value is specified more than once, then the result value returned is the one listed for the first occurrence of the search value.
- The following data types can be used in result values and in the default value:
  - CHAR
  - VARCHAR
  - VARCHAR2
  - NCHAR
  - NCHAR VARYING
  - NVARCHAR2
  - TEXT
  - INTEGER
  - BIGINT
  - NUMERIC
  - DATE
  - TIME WITHOUT TIME ZONE
  - TIMESTAMP WITHOUT TIME ZONE
  - TIMESTAMP WITH TIME ZONE
- The same data type must be specified for the values to be converted and the search values. However, note that different data types may also be specified if a literal is specified in the search value, and the value expression to be converted

contains data types that can be converted.

- If the result values and default value are all literals, the data types for these values will be as shown below:
  - If all values are string literals, all will become character types.
  - If there is one or more numeric literal, all will become numeric types.
  - If there is one or more literal cast to the datetime/time types, all will become datetime/time types.
- If the result values and default value contain a mixture of literals and non-literals, the literals will be converted to the data types of the non-literals.
- The same data type must be specified for all result values and for the default value. However, different data types can be specified if the data type of any of the result values or default value can be converted - these data types are listed below:

Data type combinations that can be converted by DECODE (summary)

		Other result values or default value		
		Numeric type	Character type	Date/time type
Result value (any)	Numeric type	Y	N	N
	Character type	N	Y	N
	Date/time type	N	N	S(*1)

Y: Can be converted

S: Some data types can be converted

N: Cannot be converted

*1: The data types that can be converted for date/time types are listed below:

Result value and default value date/time data types that can be converted by DECODE

		Other result values or default value			
		DATE	TIME WITHOUT TIME ZONE	TIMESTAMP WITHOUT TIME ZONE	TIMESTAMP WITH TIME ZONE
Result value (any)	DATE	Y	N	Y	Y
	TIME WITHOUT TIME ZONE	N	Y	N	N
	TIMESTAMP WITHOUT TIME ZONE	Y	N	Y	Y
	TIMESTAMP WITH TIME ZONE	Y	N	Y	Y

Y: Can be converted

N: Cannot be converted

- The data type of the return value will be the data type within the result or default value that is longest and has the highest precision.

Example

In the following example, the value of col3 in table t1 is compared and converted to a different value. If the col3 value matches search value 1, the result value returned is “one”. If the col3 value does not match any of search values 1, 2, or 3, the default value “other number” is returned.

```
SELECT col1,
      DECODE(col3, 1, 'one',
                   2, 'two',
                   3, 'three',
                   'other number') "num-word"
      FROM t1;
col1  | num-word
------+----------
 1001 | one
 1002 | two
 1003 | three
(3 rows)
```

5.5.2 GREATEST and LEAST

Description

The GREATEST and LEAST functions select the largest or smallest value from a list of any number of expressions. The expressions must all be convertible to a common data type, which will be the type of the result

Syntax

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

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

General rules

- These two function are the same behavior than the POstgreSQL one except that instead of retunring NULL only when all parameters are NULL ,they return NULL when one of the parameters is NULL like in Oracle.

Example

In the following example, col1 and col3 of table t1 are returned when col3 has a value of 2000 or less, or null values.

```
SELECT GREATEST ('C', 'F', 'E')
 greatest
----------
 F
(1 row)

\pset null ###
SELECT LEAST ('C', NULL, 'E')
 greatest
----------
 ###
(1 row)
```

5.5.3 LNNVL

Description

Determines if a value is TRUE or FALSE for the specified condition.

Syntax

General rules

- LNNVL determines if a value is TRUE or FALSE for the specified condition. If the result of the condition is FALSE or NULL, TRUE is returned. If the result of the condition is TRUE, FALSE is returned.

- The expression for returning TRUE or FALSE is specified in the condition.
- The data type of the return value is BOOLEAN.

Example

In the following example, col1 and col3 of table t1 are returned when col3 has a value of 2000 or less, or null values.

```
SELECT col1,col3 FROM t1 WHERE LNNVL( col3 > 2000 );
 col1 | col3
------+------
 1001 | 1000
 1002 | 2000
 2002 |
(3 row)
```

5.5.4 NANVL

Description

Returns a substitute value when a value is not a number (NaN).

Syntax

General rules

- NANVL returns a substitute value when the specified value is not a number (NaN). The substitute value can be either a number or a string that can be converted to a number.
- For *expr* and *substituteNum*, specify a numeric data type. If *expr* and *substituteNum* have different data types, they will be converted to the data type with greater length or precision, and that is the data type that will be returned.
- For *substituteNum*, you can also specify a string indicating the numeric value.
- The data type used for the return value if a string is specified for the substitute value will be the same as the data type of *expr*.

Example

In the following example, “0” is returned if the value of col1 in table t1 is a NaN value.

```
SELECT col1, NANVL(col3,0) FROM t1;
 col1 | nanvl
------+-------
 2001 |     0
(1 row)
```

5.5.5 NVL

Description

Returns a substitute value when a value is NULL.

Syntax

General rules

- NVL returns a substitute value when the specified value is NULL. When*expr1* is NULL, *expr2* is returned. When *expr1* is not NULL, *expr1* is returned.

- Specify the same data types for *expr1* and *expr2*. However, if a constant is specified in *expr2*, and the data type can also be converted by *expr1*, different data types can be specified. When this happens, the conversion by *expr2* is done to suit the data type in *expr1*, so the value of *expr2* returned when *expr1* is a NULL value will be the value converted in the data type of *expr1*. This is not necessary for types (numeric, int) and (bigint, int).

Example

In the following example, “IS NULL” is returned if the value of col1 in table t1 is a NULL value.

```
SELECT col2, NVL(col1,'IS NULL') "nvl" FROM t1;
 col2 |   nvl
------+---------
 aaa  | IS NULL
(1 row)
```

5.5.6 NVL2

Description

Returns a substitute value based on whether a value is NULL or not NULL.

Syntax

General rules

- NVL2 returns a substitute value based on whether the specified value is NULL or not NULL. When *expr* is NULL, *substitute2* is returned. When it is not NULL, *substitute1* is returned.
- Specify the same data types for *expr*, *substitute1*, and *substitute2*. However, if a literal is specified in *substitute1* or *substitute2*, and the data type can also be converted by *expr*, different data types can be specified. When this happens, *substitute1* or *substitute2* is converted to suit the data type in *expr*, so the value of *substitute2* returned when *expr* is a NULL value will be the value converted to the data type of *expr*.

Example

In the following example, if a value in column col1 in table t1 is NULL, “IS NULL” is returned, and if not NULL, “IS NOT NULL” is returned.

```
SELECT col2, NVL2(col1,'IS NOT NULL','IS NULL') FROM t1;
 col2 |   nvl2
------+---------
 aaa  | IS NULL
 bbb  | IS NOT NULL
(2 row)
```

5.6 Aggregate Functions

The following aggregation functions are supported:

- LISTAGG
- MEDIAN

5.6.1 LISTAGG

Description

Returns a concatenated, delimited list of string values.

Syntax

General rules

- LISTAGG concatenates and delimits a set of string values and returns the result.
- For *delimiter*, specify a string. If the delimiter is omitted, a list of strings without a delimiter is returned.
- The data type of the return value is TEXT.

Example

In the following example, the result with values of column col2 in table t1 delimited by ':' is returned.

```
SELECT LISTAGG(col2,':') FROM t1;
     listagg
--------------------
 AAAAA:BBBBB:CCCCC
(1 row)
```

5.6.2 MEDIAN

Description

Calculates the median of a set of numbers.

Syntax

General rules

- MEDIAN returns the median of a set of numbers.
- The numbers must be numeric data type.
- The data type of the return value will be REAL if the numbers are REAL type, or DOUBLE PRECISION if any other type is specified.

Example

In the following example, the median of column col3 in table t1 is returned.

```
SELECT MEDIAN(col3) FROM t1;
 median
--------
   2000
(1 row)
```

5.7 Functions That Return Internal Information

The following functions that return internal information are supported:

- DUMP

5.7.1 DUMP

Description

Returns internal information of a value.

Syntax

General rules

- DUMP returns the internal information of the values specified in expressions in a display format that is in accordance with the output format.
- The internal code (Typ) of the data type, the data length (Len) and the internal expression of the data are output as internal information.
- Any data type can be specified for the expressions.
- The display format (base *n*) of the internal expression of the data is specified for the output format. The base numbers that can be specified are 8, 10, and 16. If omitted, 10 is used as the default.
- The data type of the return value is VARCHAR.

Note

The information output by DUMP will be the complete internal information. Therefore, the values may change due to product updates, and so on.

Example

In the following example, the internal information of column col1 in table t1 is returned.

```
SELECT col1, DUMP(col1) FROM t1;
 col1 |               dump
------+-----------------------------------
 1001 | Typ=25 Len=8: 32,0,0,0,49,48,48,49
 1002 | Typ=25 Len=8: 32,0,0,0,49,48,48,50
 1003 | Typ=25 Len=8: 32,0,0,0,49,48,48,51
(3 row)
```

5.8 Datetime Operator

The following datetime operators are supported for the DATE type of orafce.

Datetime operator

Operation	Example	Result
+	DATE'2016/01/01' + 10	2016-01-11 00:00:00
-	DATE'2016/03/20' - 35	2016-02-14 00:00:00
-	DATE'2016/09/01' - DATE'2015/12/31'	245

Note

If using datetime operators for the DATE type of orafce, it is necessary to specify "oracle" for search_path in advance.

See

Refer to "Notes on Using orafce" for information on how to edit search_path.

Chapter 6 Package Reference

A "package" is a group of features, brought together by schemas, that have a single functionality, and are used by calling from

PL/pgSQL.

The following packages are supported:

- DBMS_ALERT
- DBMS_ASSERT
- DBMS_OUTPUT
- DBMS_PIPE
- DBMS_RANDOM
- DBMS_UTILITY
- UTL_FILE

To call the different functionalities from PL/pgSQL, use the PERFORM statement or SELECT statement, using the package name to qualify the name of the functionality. Refer to the explanations for each of the package functionalities for information on the format for calling.

6.1 DBMS_ALERT

Overview

The DBMS_ALERT package sends alerts from a PL/pgSQL session to multiple other PL/pgSQL sessions.

This package can be used when processing 1:N, such as when notifying alerts from a given PL/pgSQL session to another PL/pgSQL session at the same time.

Features

Feature	Description
REGISTER	Registers the specified alert.
REMOVE	Removes the specified alert.
REMOVEALL	Removes all alerts from a session.
SIGNAL	Notifies alerts.
WAITANY	Waits for notification of any alerts for which a session is registered.
WAITONE	Waits for notification of a specific alert for which a session is registered.

Syntax

6.1.1 Description of Features

This section explains each feature of DBMS_ALERT.

REGISTER

- REGISTER registers the specified alert to a session. By registering alerts to a session, SIGNAL notifications can be received.
- Specify the name of the alert.
- Alerts are case-sensitive.
- Multiple alerts can be registered within a single session. If registering multiple alerts, call REGISTER for each alert.

Example

```
PERFORM DBMS_ALERT.REGISTER('sample_alert');
```

REMOVE

- REMOVE removes the specified alert from a session.
- Specify the name of the alert.

- Alerts are case-sensitive.
- The message left by the alert will be removed.

Example

```
PERFORM DBMS_ALERT.REMOVE('sample_alert');
```

REMOVEALL

- REMOVEALL removes all alerts registered within a session.
- All messages left by the alerts will be removed.

Example

```
PERFORM DBMS_ALERT.REMOVEALL();
```

SIGNAL

- SIGNAL sends a message notification for the specified alert.
- Specify the name of the alert for which message notifications are sent.
- Alerts are case-sensitive.
- In the message, specify the alert message for notifications.
- Message notifications are not complete at the stage when SIGNAL is executed. Message notifications are sent upon committing the transaction. Message notifications are discarded if a rollback is performed after SIGNAL is executed.
- If message notifications are sent for the same alert from multiple sessions, the messages will be accumulated without being removed.

Example

```
PERFORM DBMS_ALERT.SIGNAL('ALERT001','message001');
```

Note

If SIGNAL is issued continuously and the accumulated messages exceed a certain amount, an insufficient memory error may be output. If the memory becomes insufficient, call AITANY or WAITONE to receive an alert, and reduce the accumulated messages.

WAITANY

- WAITANY waits for notification of any alerts registered for a session.
- Specify the maximum wait time*timeout* in seconds to wait for an alert.
- Use a SELECT statement to obtain the notified information, which is stored in the name, message and status columns.
- The name column stores the alert names. The data type of name is TEXT.
- The message column stores the messages of notified alerts. The data type of message is TEXT.
- The status column stores the status code returned by the operation: 0-an alert occurred; 1-a timeout occurred. The data type of status is INTEGER.

Example

```
DECLARE
   alert_name        TEXT := 'sample_alert';
   alert_message     TEXT;
   alert_status      INTEGER;
BEGIN
   SELECT name,message,status INTO alert_name,alert_message,alert_status FROM DBMS_ALERT.WAITANY(60);
```

WAITONE

- WAITONE waits for notification of the specified alert.
- Specify the name of the alert to wait for.
- Alerts are case-sensitive.
- Specify the maximum wait time*timeout* in seconds to wait for the alert.
- Use a SELECT statement to obtain the notified information, which is stored in the message and status columns.
- The message column stores the messages of notified alerts. The data type of message is TEXT.
- The status column stores the status code returned by the operation: 0-an alert occurred; 1-a timeout occurred. The data type of status is INTEGER.

Example

```
DECLARE
   alert_message   TEXT;
   alert_status    INTEGER;
BEGIN
   SELECT message,status INTO alert_message,alert_status FROM DBMS_ALERT.WAITONE('sample_alert', 60);
```

6.1.2 Usage Example

Below is a usage example of the processing flow of DBMS_ALERT.

DBMS_ALERT flow

Note

- The target of message notifications by SIGNAL is sessions for which REGISTER is executed at the time of executing SIGNAL.
- On the receiving side, always ensure that REMOVE or REMOVEALL is used to remove alerts as soon as the alerts are no longer needed. If a session is closed without removing the alerts, it may no longer be possible to receive a SIGNAL for alerts of the same name in another session.
- DBMS_ALERT and DBMS_PIPE use the same memory environment. Therefore, when insufficient memory is detected for DBMS_PIPE, it is possible that insufficient memory will also be detected for DBMS_ALERT.

Usage example

- Sending side

  ```
  CREATE FUNCTION send_dbms_alert_exe() RETURNS VOID AS $$
  BEGIN
   PERFORM DBMS_ALERT.SIGNAL('sample_alert','SIGNAL ALERT');
  END;
  $$ LANGUAGE plpgsql;
  SELECT send_dbms_alert_exe();
  DROP FUNCTION send_dbms_alert_exe();
  ```

- Receiving side

```
CREATE FUNCTION receive_dbms_alert_exe() RETURNS VOID AS $$
DECLARE
 alert_name    TEXT := 'sample_alert';
 alert_message TEXT;
 alert_status  INTEGER;
BEGIN
 PERFORM DBMS_ALERT.REGISTER(alert_name);
 SELECT message,status INTO alert_message,alert_status FROM DBMS_ALERT.WAITONE(alert_name,300);
 RAISE NOTICE 'Message : %', alert_message;
 RAISE NOTICE 'Status  : %', alert_status;
 PERFORM DBMS_ALERT.REMOVE(alert_name);
END;
$$ LANGUAGE plpgsql;
SELECT receive_dbms_alert_exe();
DROP FUNCTION receive_dbms_alert_exe();
```

6.2 DBMS_ASSERT

Overview

Performs verification of the properties of input values in PL/pgSQL.

Features

Feature	Description
ENQUOTE_LITERAL	Returns the specified string enclosed in single quotation marks.
ENQUOTE_NAME	Returns the specified string enclosed in double quotation marks.
NOOP	Returns the specified string as is.
OBJECT_NAME	Verifies if the specified string is a defined identifier.
QUALIFIED_SQL_NAME	Verifies if the specified string is in the appropriate format as an identifier.
SCHEMA_NAME	Verifies if the specified string is a defined schema.
SIMPLE_SQL_NAME	Verifies if the specified string is in the appropriate format as a single identifier.

Syntax

6.2.1 Description of Features

This section explains each feature of DBMS_ASSERT.

ENQUOTE_LITERAL

- ENQUOTE_LITERAL returns the specified string enclosed in single quotation marks.
- Specify a string enclosed in single quotation marks.
- The data type of the return value is VARCHAR.

Example

```
DECLARE
   q_literal    VARCHAR(256);
BEGIN
   q_literal := DBMS_ASSERT.ENQUOTE_LITERAL('literal_word');
```

ENQUOTE_NAME

- ENQUOTE_NAME returns the specified string enclosed in double quotation marks.
- Specify a string enclosed in double quotation marks.
- For lowercase conversion, specify TRUE or FALSE. Specify TRUE to convert uppercase characters in the string to

lowercase. If FALSE is specified, conversion to lowercase will not take place. The default is TRUE.

- If all the characters in the string are lowercase, they will not be enclosed in double quotation marks.
- The data type of the return value is VARCHAR.

See

Refer to “The SQL Language” > “Data Types” > “Boolean Type” in the PostgreSQL Documentation for information on boolean type (TRUE/FALSE) values.

Example

```
DECLARE
   dq_literal    VARCHAR(256);
BEGIN
   dq_literal := DBMS_ASSERT.ENQUOTE_NAME('TBL001');
```

NOOP

- NOOP returns the specified string as is.
- Specify a string.
- The data type of the return value is VARCHAR.

Example

```
DECLARE
   literal    VARCHAR(256);
BEGIN
   literal := DBMS_ASSERT.NOOP('NOOP_WORD');
```

OBJECT_NAME

- OBJECT_NAME verifies if the specified string is a defined identifier.
- Specify the identifier for verification. If the identifier has been defined, the specified identifier will be returned. Otherwise, the following error will occur.

  ```
  ERROR:  invalid object name
  ```

- The data type of the return value is VARCHAR.

Example

```
DECLARE
   object_name    VARCHAR(256);
BEGIN
   object_name := DBMS_ASSERT.OBJECT_NAME('SCM001.TBL001');
```

QUALIFIED_SQL_NAME

- QUALIFIED_SQL_NAME verifies if the specified string is in the appropriate format as an identifier.
- Specify the identifier for verification. If the string can be used as an identifier, the specified identifier will be returned. Otherwise, the following error will occur.

  ```
  ERROR:  string is not qualified SQL name
  ```

- The data type of the return value is VARCHAR.

See

Refer to “The SQL Language” > “Lexical Structure” > “Identifiers and Key Words” in the PostgreSQL Documentation for information on the formats that can be used as identifiers.

Example

```
DECLARE
   object_name    VARCHAR(256);
BEGIN
   object_name := DBMS_ASSERT.QUALIFIED_SQL_NAME('SCM002.TBL001');
```

SCHEMA_NAME

- SCHEMA_NAME verifies if the specified string is a defined schema.
- Specify a schema name for verification. If the schema has been defined, the specified schema name will be returned. Otherwise, the following error will occur.

  ERROR: invalid schema name

- The data type of the return value is VARCHAR.

Example

```
DECLARE
   schema_name    VARCHAR(256);
BEGIN
   schema_name := DBMS_ASSERT.SCHEMA_NAME('SCM001');
```

SIMPLE_SQL_NAME

- SIMPLE_SQL_NAME verifies if the specified string is in the appropriate format as a single identifier.
- Specify an identifier for verification. If the specified string can be used as an identifier, the specified identifier will be returned. Otherwise, the following error will occur.

  ERROR: string is not qualified SQL name

- The data type of the return value is VARCHAR.

See

Refer to “The SQL Language” > “Lexical Structure” > “Identifiers and Key Words” in the PostgreSQL Documentation for information on the formats that can be used as identifiers. Note that an error will occur if an identifier using fullwidth characters is specified. If fullwidth characters are included, specify a quoted identifier.

Example

```
DECLARE
   simple_name    VARCHAR(256);
BEGIN
   simple_name := DBMS_ASSERT.SIMPLE_SQL_NAME('COL01');
```

6.2.2 Usage Example

A usage example of DBMS_ASSERT is shown below.

```
CREATE FUNCTION dbms_assert_exe() RETURNS VOID AS $$
DECLARE
 w_schema VARCHAR(20) := 'public';
 w_table  VARCHAR(20) := 'T1';
 w_object VARCHAR(40);
BEGIN
 PERFORM DBMS_ASSERT.NOOP(w_schema);
 PERFORM DBMS_ASSERT.SIMPLE_SQL_NAME(w_table);
 PERFORM DBMS_ASSERT.SCHEMA_NAME(w_schema);
 w_object := w_schema || '.' || w_table;
 PERFORM DBMS_ASSERT.QUALIFIED_SQL_NAME(w_object);
 PERFORM DBMS_ASSERT.OBJECT_NAME(w_object);
 RAISE NOTICE 'OBJECT     : %', DBMS_ASSERT.ENQUOTE_LITERAL(w_object);
 RAISE NOTICE 'TABLE_NAME : %', DBMS_ASSERT.ENQUOTE_NAME(w_table);
END;
$$
LANGUAGE plpgsql;
SELECT dbms_assert_exe();
DROP FUNCTION dbms_assert_exe();
```

6.3 DBMS_OUTPUT

Overview

Sends messages to clients such as psql from PL/pgSQL.

Features

Feature	Description
ENABLE	Enables features of this package.
DISABLE	Disables features of this package.
SERVEROUTPUT	Controls whether messages are sent.
PUT	Sends messages.
PUT_LINE	Sends messages with a newline character appended.
NEW_LINE	Sends a newline character.
GET_LINE	Retrieves a line from the message buffer.
GET_LINES	Retrieves multiple lines from the message buffer.

Syntax

6.3.1 Description

This section explains each feature of DBMS_OUTPUT.

ENABLE

- ENABLE enables the use of PUT, PUT_LINE, NEW_LINE, GET_LINE, and GET_LINES.
- With multiple executions of ENABLE, the value specified last is the buffer size (in bytes). Specify a buffer size from 2000 to 1000000.
- The default value of the buffer size is 20000. If NULL is specified as the buffer size, 1000000 will be used.
- If ENABLE has not been executed, PUT, PUT_LINE, NEW_LINE, GET_LINE, and GET_LINES are ignored even if they are executed.

Example

```
PERFORM DBMS_OUTPUT.ENABLE(20000);
```

DISABLE

- DISABLE disables the use of PUT, PUT_LINE, NEW_LINE, GET_LINE, and GET_LINES.
- Remaining buffer information is discarded.

Example

```
PERFORM DBMS_OUTPUT.DISABLE();
```

SERVEROUTPUT

- SERVEROUTPUT controls whether messages are sent.
- Specify TRUE or FALSE for *sendMsgs*.
- If TRUE is specified, when PUT, PUT_LINE, or NEW_LINE is executed, the message is sent to a client such as psql and not stored in the buffer.
- If FALSE is specified, when PUT, PUT_LINE, or NEW_LINE is executed, the message is stored in the buffer and not sent to a client such as psql.

See

Refer to “The SQL Language” > “Data Types” > “Boolean Type” in the PostgreSQL Documentation for information on boolean type (TRUE/FALSE) values.

Example

```
PERFORM DBMS_OUTPUT.SERVEROUTPUT(TRUE);
```

PUT

- PUT sets the message to be sent.
- The string is the message to be sent.
- When TRUE is specified for SERVEROUTPUT, the messages are sent to clients such as psql.
- When FALSE is specified for SERVEROUTPUT, the messages are retained in the buffer.
- PUT does not append a newline character. To append a newline character, execute NEW_LINE.
- If a string longer than the buffer size specified in ENABLE is sent, an error occurs.

Example

```
PERFORM DBMS_OUTPUT.PUT('abc');
```

PUT_LINE

- PUT_LINE sets the message to be sent appended with a newline character.
- The string is the message to be sent.
- When TRUE is specified for SERVEROUTPUT, the messages are sent to clients such as psql.
- When FALSE is specified for SERVEROUTPUT, the messages are retained in the buffer.
- If a string longer than the buffer size specified in ENABLE is sent, an error occurs.

Example

```
PERFORM DBMS_OUTPUT.PUT_LINE('abc');
```

NEW_LINE

- NEW_LINE appends a newline character to the message created with PUT.
- When TRUE is specified for SERVEROUTPUT, the messages are sent to clients such as psql.
- When FALSE is specified for SERVEROUTPUT, the messages are retained in the buffer.

Example

```
PERFORM DBMS_OUTPUT.NEW_LINE();
```

GET_LINE

- GET_LINE retrieves a line from the message buffer.
- Use a SELECT statement to obtain the retrieved line and status code returned by the operation, which are stored in the line and status columns.
- The line column stores the line retrieved from the buffer. The data type of line is TEXT.
- The status column stores the status code returned by the operation: 0-completed successfully; 1-failed because there are no more lines in the buffer. The data type of status is INTEGER.
- If GET_LINE or GET_LINES is executed and then PUT, PUT_LINE or PUT_LINES is executed while messages that have not been retrieved from the buffer still exist, the messages not retrieved from the buffer will be discarded.

Example

```
DECLARE
   buff1   VARCHAR(20);
   stts1   INTEGER;
BEGIN
   SELECT line,status INTO buff1,stts1 FROM DBMS_OUTPUT.GET_LINE();
```

GET_LINES

- GET_LINES retrieves multiple lines from the message buffer.
- Specify the number of lines to retrieve from the buffer.
- Use a SELECT statement to obtain the retrieved lines and the number of lines retrieved, which are stored in the lines and numlines columns.
- The lines column stores the lines retrieved from the buffer. The data type of lines is TEXT.
- The numlines column stores the number of lines retrieved from the buffer. If this number is less than the number of lines requested, then there are no more lines in the buffer. The data type of numlines is INTEGER.
- If GET_LINE or GET_LINES is executed and then PUT, PUT_LINE, or NEW_LINE is executed while messages that have not been retrieved from the buffer still exist, the messages not retrieved from the buffer will be discarded.

Example

```
DECLARE
   buff    VARCHAR(20)[10];
   stts    INTEGER := 10;
BEGIN
   SELECT lines, numlines INTO buff,stts FROM DBMS_OUTPUT.GET_LINES(stts);
```

6.3.2 Usage Example

A usage example of DBMS_OUTPUT is shown below.

```
CREATE FUNCTION dbms_output_exe() RETURNS VOID AS $$
DECLARE
 buff1 VARCHAR(20);
 buff2 VARCHAR(20);
 stts1 INTEGER;
 stts2 INTEGER;
BEGIN
 PERFORM DBMS_OUTPUT.DISABLE();
 PERFORM DBMS_OUTPUT.ENABLE();
 PERFORM DBMS_OUTPUT.SERVEROUTPUT(FALSE);
 PERFORM DBMS_OUTPUT.PUT('DBMS_OUTPUT TEST 1');
 PERFORM DBMS_OUTPUT.NEW_LINE();
 PERFORM DBMS_OUTPUT.PUT_LINE('DBMS_OUTPUT TEST 2');
 SELECT line,status INTO buff1,stts1 FROM DBMS_OUTPUT.GET_LINE();
 SELECT line,status INTO buff2,stts2 FROM DBMS_OUTPUT.GET_LINE();
 PERFORM DBMS_OUTPUT.SERVEROUTPUT(TRUE);
 PERFORM DBMS_OUTPUT.PUT_LINE(buff1);
 PERFORM DBMS_OUTPUT.PUT_LINE(buff2);
END;
$$ LANGUAGE plpgsql;
SELECT dbms_output_exe();
DROP FUNCTION dbms_output_exe();
```

6.4 DBMS_PIPE

Overview

Performs communication between sessions that execute PL/pgSQL.

This package can be used for 1:1 communication, such as when data is being exchanged between sessions executing PL/pgSQL.

For pipes, there are explicit pipes and implicit pipes, and furthermore, for explicit pipes, you can select public pipes and private pipes. The characteristics of each type are as follows:

Types of pipes

Type	Characteristics
Explicit pipe	- CREATE_PIPE is used to create a pipe explicitly. - While creating a pipe, you can select between a public pipe and private pipe. - It is necessary to use REMOVE_PIPE to explicitly remove a pipe.
Implicit pipe	- Created automatically when SEND_MESSAGE and RECEIVE_MESSAGE are used. - The pipe that is created becomes a public pipe. - When messages are received using RECEIVE_MESSAGE, if there are no additional messages remaining in the pipe, the pipe will be removed automatically.
Public pipe	- Can be created as an explicit pipe or implicit pipe. - Can also be used by users other than the creator.
Private pipe	- Can only be created as an explicit pipe. - Can only be used by its creator.

Note

- Up to 50 pipes can be used concurrently by a single instance.
- In cases where pipes are frequently created and removed repetitively, use public pipes. If you create a private pipe, internal information (the creator of the private pipe) will remain even after the pipe is removed. Thus, repeatedly creating and removing pipes may ultimately cause memory to run out.

- If a timeout occurs without receiving a message when an implicit pipe is created by RECEIVE_MESSAGE, the pipe will not be removed.

Features

Feature	Description
CREATE_PIPE	Creates a public or private pipe.
NEXT_ITEM_TYPE	Determines the data type of the next item in the local buffer, and returns that type.
PACK_MESSAGE	Sets a message in the local buffer.
PURGE	Empties the contents of the specified pipe.
RECEIVE_MESSAGE	Sets a received message in the local buffer.
REMOVE_PIPE	Removes the specified pipe.
RESET_BUFFER	Resets the set position of the local buffer.
SEND_MESSAGE	Sends the contents of the local buffer.
UNIQUE_SESSION_NAME	Returns a unique session name.
UNPACK_MESSAGE_BYTEA	Receives a message in the local buffer in BYTEA type.
UNPACK_MESSAGE_DATE	Receives a message in the local buffer in DATE type.
UNPACK_MESSAGE_NUMBER	Receives a message in the local buffer in NUMERIC type.
UNPACK_MESSAGE_RECORD	Receives a message in the local buffer in RECORD type.
UNPACK_MESSAGE_TEXT	Receives a message in the local buffer in TEXT type.
UNPACK_MESSAGE_TIMESTAMP	Receives a message in the local buffer in TIMESTAMP type.

Syntax

6.4.1 Description of Features

This section explains each feature of DBMS_PIPE.

CREATE_PIPE

- CREATE_PIPE explicitly creates a pipe environment for data communication.
- Specify the name of the pipe to be created.
- Pipe names are case-sensitive.
- Specify the maximum number of messages that can be sent or received. If omitted, 0 (cannot send messages) will be used. Specify from 1 to 32767.
- Specify TRUE or FALSE for *private*. If TRUE is specified, a private pipe will be created. If FALSE is specified, a public pipe will be created. The default is FALSE.
- An error will occur if a pipe of the same name has already been created.

See

Refer to “The SQL Language” > “Data Types” > “Boolean Type” in the PostgreSQL Documentation for information on boolean type (TRUE/FALSE) values.

Example

```
PERFORM DBMS_PIPE.CREATE_PIPE('P01', 100, FALSE);
```

NEXT_ITEM_TYPE

- NEXT_ITEM_TYPE returns the next data type in the local buffer.
- The data type of the return value is INTEGER. One of the following values is returned:

Values returned by NEXT_ITEM_TYPE

Return value	Data type
9	NUMERIC type
11	TEXT type
12	DATE type
13	TIMESTAMP type
23	BYTEA type
24	RECORD type
0	No data in the buffer

Example

```
DECLARE
  i_iType   INTEGER;
BEGIN
  i_iType := DBMS_PIPE.NEXT_ITEM_TYPE();
```

PACK_MESSAGE

- PACK_MESSAGE sets the specified message in the local buffer.
- Specify the data to be set in the local buffer. The following data types can be used:
  - Character type (*1)
  - Integer type (*2)
  - NUMERIC type
  - DATE type
  - TIMESTAMP type (*3)
  - BYTEA type
  - RECORD type

*1: The character type is converted internally to TEXT type.

*2: The integer type is converted internally to NUMERIC type.

*3: The TIMESTAMP type is converted internally to TIMESTAMP WITH TIME ZONE type.

- Each time PACK_MESSAGE is called, a new message is added to the local buffer.
- The size of the local buffer is approximately 8 KB. However, each message has overhead, so the total size that can be stored is actually less than 8 KB. To clear the local buffer, send a message (SEND_MESSAGE), or reset the buffer (RESET_BUFFER) to its initial state.

Example

```
PERFORM DBMS_PIPE.PACK_MESSAGE('Message Test001');
```

PURGE

- PURGE removes the messages in the pipe.
- Specify the name of the pipe for which the messages are to be removed.
- Pipe names are case-sensitive.

Example

```
PERFORM DBMS_PIPE.PURGE('P01');
```

Note

When PURGE is executed, the local buffer is used to remove the messages in the pipe. Therefore, if there are any messages remaining in the pipe, the local buffer will be overwritten by PURGE.

RECEIVE_MESSAGE

- RECEIVE_MESSAGE receives messages that exist in the specified pipe, and sets those messages in the local buffer.
- Messages are received in the units in which they are sent to the pipe by SEND_MESSAGE. Received messages are removed from the pipe after being set in the local buffer.
- Specify the name of the pipe for which the messages are to be received.
- Pipe names are case-sensitive.
- Specify the maximum wait time *timeout* in seconds to wait for a message. If omitted, the default is 31536000 seconds (1 year).
- The data type of the return value is INTEGER. If a message is received successfully, 0 is returned. If a timeout occurs, 1 is returned.

Example

```
DECLARE
   i_Ret    INTEGER;
BEGIN
   i_Ret := DBMS_PIPE.RECEIVE_MESSAGE('P01', 60);
```

REMOVE_PIPE

- REMOVE_PIPE removes the specified pipe.
- Specify the name of the pipe to be removed.
- Pipe names are case-sensitive.

Example

```
PERFORM DBMS_PIPE.REMOVE_PIPE('P01');
```

RESET_BUFFER

- RESET_BUFFER resets the set position of the local buffer. Any unnecessary data remaining in the local buffer can be discarded using this operation.

Example

```
PERFORM DBMS_PIPE.RESET_BUFFER();
```

SEND_MESSAGE

- SEND_MESSAGE sends data stored in the local buffer to the specified pipe.
- Specify the name of the pipe that the data is to be sent to.
- Pipe names are case-sensitive.
- Specify the maximum wait time *timeout* in seconds for sending data stored in the local buffer. If omitted, the default is 31536000 seconds (1 year).
- Specify the maximum number of messages that can be sent or received. If omitted, the maximum number of messages set in CREATE_PIPE is used. If omitted in the implicit pipe, the number of messages will be unlimited. Specify from 1 to 32767.
- If the maximum number of messages is specified in both SEND_MESSAGE and CREATE_PIPE, the larger of the values

will be used.
- The data type of the return value is INTEGER. If a message is received successfully, 0 is returned. If a timeout occurs, 1 is returned.

Example

```
DECLARE
  i_Ret    INTEGER;
BEGIN
  i_Ret := DBMS_PIPE.SEND_MESSAGE('P01', 10, 20);
```

Note

A timeout will occur during sending if the maximum number of messages is reached, or if the message being sent is too large. If a timeout occurs, use RECEIVE_MESSAGE to receive any messages that are in the pipe.

UNIQUE_SESSION_NAME

- UNIQUE_SESSION_NAME returns a name that is unique among all the sessions. This name can be used as the pipe name.
- Multiple calls from the same session always return the same name.
- The data type of the return value is VARCHAR. Returns a string of up to 30 characters.

Example

```
DECLARE
  p_Name   VARCHAR(30);
BEGIN
  p_Name := DBMS_PIPE.UNIQUE_SESSION_NAME();
```

UNPACK_MESSAGE_BYTEA

- NPACK_MESSAGE_BYTEA receives BTYEA type messages in the local buffer.
- Messages are received in the unit set in the local buffer by PACK_MESSAGE. Received messages are removed from the local buffer.
- The data type of the return value is BYTEA.
- If no messages exist in the local buffer, a NULL value is returned.
- For the data type, it is necessary to align with the data type set by PACK_MESSAGE. If the data type is different, the following error will occur.

  ```
  ERROR:  datatype mismatch
  DETAIL:  unpack unexpected type: xx
  ```

Example

```
DECLARE
  g_Bytea   BYTEA;
BEGIN
  g_Bytea := DBMS_PIPE.UNPACK_MESSAGE_BYTEA();
```

UNPACK_MESSAGE_DATE

- UNPACK_MESSAGE_DATE receives DATE type messages in the local buffer.
- Messages are received in the unit set in the local buffer by PACK_MESSAGE. Received messages are removed from the local buffer.

- The data type of the return value is DATE.
- If no messages exist in the local buffer, a NULL value is returned.
- For the data type, it is necessary to align with the data type set by PACK_MESSAGE. If the data type is different, the following error will occur.

  ```
  ERROR:  datatype mismatch
  DETAIL:  unpack unexpected type: xx
  ```

Example

```
DECLARE
   g_Date   DATE;
BEGIN
   g_Date := DBMS_PIPE.UNPACK_MESSAGE_DATE();
```

Note

If the “oracle” schema is set in search_path, the DATE type of orafce will be used, so for receiving data, use UNPACK_MESSAGE_TIMESTAMP. UNPACK_MESSAGE_DATE is the interface for the DATE type of PostgreSQL.

UNPACK_MESSAGE_NUMBER

- UNPACK_MESSAGE_NUMBER receives NUMERIC type messages in the local buffer.
- Messages are received in the unit set in the local buffer by PACK_MESSAGE. Received messages are removed from the local buffer.
- The data type of the return value is NUMERIC.
- If no messages exist in the local buffer, a NULL value is returned.
- For the data type, it is necessary to align with the data type set by PACK_MESSAGE. If the data type is different, the following error will occur.

  ```
  ERROR:  datatype mismatch
  DETAIL:  unpack unexpected type: xx
  ```

Example

```
DECLARE
   g_Number   NUMERIC;
BEGIN
   g_Number := DBMS_PIPE.UNPACK_MESSAGE_NUMBER();
```

UNPACK_MESSAGE_RECORD

- UNPACK_MESSAGE_RECORD receives RECORD type messages in the local buffer.
- Messages are received in the unit set in the local buffer by PACK_MESSAGE. Received messages are removed from the local buffer.
- The data type of the return value is RECORD.
- If no messages exist in the local buffer, a NULL value is returned.
- For the data type, it is necessary to align with the data type set by PACK_MESSAGE. If the data type is different, the following error will occur.

  ```
  ERROR:  datatype mismatch
  DETAIL:  unpack unexpected type: xx
  ```

Example

```
DECLARE
   msg1     TEXT;
   status   NUMERIC;
BEGIN
   SELECT col1, col2 INTO msg1, status FROM DBMS_PIPE.UNPACK_MESSAGE_RECORD();
```

UNPACK_MESSAGE_TEXT

- UNPACK_MESSAGE_TEXT receives TEXT type messages in the local buffer.
- Messages are received in the unit set in the local buffer by PACK_MESSAGE. Received messages are removed from the local buffer.
- The data type of the return value is TEXT.
- If no messages exist in the local buffer, a NULL value is returned.
- For the data type, it is necessary to align with the data type set by PACK_MESSAGE. If the data type is different, the following error will occur.

  ```
  ERROR:  datatype mismatch
  DETAIL:  unpack unexpected type: xx
  ```

Example

```
DECLARE
   g_Text   TEXT;
BEGIN
   g_Text := DBMS_PIPE.UNPACK_MESSAGE_TEXT();
```

UNPACK_MESSAGE_TIMESTAMP

- UNPACK_MESSAGE_TIMESTAMP receives TIMESTAMP WITH TIME ZONE type messages in the local buffer.
- Messages are received in the unit set in the local buffer by PACK_MESSAGE. Received messages are removed from the local buffer.
- The data type of the return value is TIMESTAMP WITH TIME ZONE.
- If no messages exist in the local buffer, a NULL value is returned.
- For the data type, it is necessary to align with the data type set by PACK_MESSAGE. If the data type is different, the following error will occur.

  ```
  ERROR:  datatype mismatch
  DETAIL:  unpack unexpected type: xx
  ```

Example

```
DECLARE
   g_Timestamptz   TIMESTAMP WITH TIME ZONE;
BEGIN
   g_Timestamptz := DBMS_PIPE.UNPACK_MESSAGE_TIMESTAMP();
```

6.4.2 Usage Example

Below is a usage example of the processing flow of DBMS_PIPE.

Flow of DBMS_PIPE

Note

- When CREATE_PIPE is used to explicitly create a pipe, ensure to use REMOVE_PIPE to remove the pipe. If a pipe is not removed explicitly, once created, it will remain until the instance is stopped.

- In the flow diagram, CREATE_PIPE and REMOVE_PIPE are described on the receiving side, however, these can be executed on the sending side. In order to maintain consistency, it is recommended to create and remove pipes on one side.
- An error will occur for CREATE_PIPE if a pipe of the same name already exists. Implicitly created pipes are also the target of SEND_MESSAGE and RECEIVE_MESSAGE, so when executing CREATE_PIPE, ensure that SEND_MESSAGE and RECEIVE_MESSAGE are not called beforehand.
- DBMS_ALERT and DBMS_PIPE use the same memory environment. Therefore, when insufficient memory is detected for DBMS_ALERT, it is possible that insufficient memory will also be detected for DBMS_PIPE.

Information

The information of pipes that are in use can be viewed in the DBMS_PIPE.DB_PIPES view.

```
SELECT * from dbms_pipe.db_pipes;
 name | items | size | limit | private | owner
------+-------+------+-------+---------+-------
 P01  |     1 |   18 |   100 | f       |
(1 row)
```

Usage example

- Sending side

```
CREATE FUNCTION send_dbms_pipe_exe(IN pipe_mess text) RETURNS void AS $$
DECLARE
 pipe_name text := 'sample_pipe';
 pipe_time timestamp := current_timestamp;
 pipe_stat int;
BEGIN
 PERFORM DBMS_PIPE.RESET_BUFFER();
 PERFORM DBMS_PIPE.PACK_MESSAGE(pipe_mess);
 PERFORM DBMS_PIPE.PACK_MESSAGE(pipe_time);
 pipe_stat := DBMS_PIPE.SEND_MESSAGE(pipe_name);
 RAISE NOTICE 'PIPE_NAME: % SEND Return Value =%', pipe_name, pipe_stat;
END;
$$ LANGUAGE plpgsql;

SELECT send_dbms_pipe_exe('Sample Message.');
DROP FUNCTION send_dbms_pipe_exe(text);
```

- Receiving side

```
CREATE FUNCTION receive_dbms_pipe_exe() RETURNS void AS $$
DECLARE
 pipe_name text := 'sample_pipe';
 pipe_text text;
 pipe_nume numeric;
 pipe_date date;
 pipe_time timestamp with time zone;
 pipe_byte bytea;
 pipe_reco record;
 pipe_item int;
 pipe_stat int;
BEGIN
 pipe_stat := DBMS_PIPE.RECEIVE_MESSAGE(pipe_name,300);
 RAISE NOTICE 'Return Value = %', pipe_stat;
 LOOP
  pipe_item := DBMS_PIPE.NEXT_ITEM_TYPE();
  RAISE NOTICE 'Next Item : %', pipe_item;
  IF (pipe_item = 9) THEN
   pipe_nume := DBMS_PIPE.UNPACK_MESSAGE_NUMBER();
   RAISE NOTICE 'Get Message : %' ,pipe_nume;
  ELSIF (pipe_item =11) THEN
   pipe_text := DBMS_PIPE.UNPACK_MESSAGE_TEXT();
   RAISE NOTICE 'Get Message : %' ,pipe_text;
  ELSIF (pipe_item = 12) THEN
   pipe_date := DBMS_PIPE.UNPACK_MESSAGE_DATE();
   RAISE NOTICE 'Get Message : %' ,pipe_date;
  ELSIF (pipe_item = 13) THEN
   pipe_time := DBMS_PIPE.UNPACK_MESSAGE_TIMESTAMP();
   RAISE NOTICE 'Get Message : %' ,pipe_time;
  ELSIF (pipe_item = 23) THEN
   pipe_byte := DBMS_PIPE.UNPACK_MESSAGE_BYTEA();
   RAISE NOTICE 'Get Message : %' ,pipe_byte;
  ELSIF (pipe_item = 24) THEN
   pipe_reco := DBMS_PIPE.UNPACK_MESSAGE_RECORD();
   RAISE NOTICE 'Get Message : %' ,pipe_reco;
  ELSE
   EXIT;
  END IF;
 END LOOP;
 PERFORM DBMS_PIPE.REMOVE_PIPE(pipe_name);
END;
$$ LANGUAGE plpgsql;

SELECT receive_dbms_pipe_exe();
DROP FUNCTION receive_dbms_pipe_exe();
```

6.5 DBMS_RANDOM

Overview

Generates random numbers in PL/pgSQL.

Features

Feature	Description
INITIALIZE	Initializes the generation of random numbers.
NORMAL	Returns a normally distributed random number.
RANDOM	Generates a random number.
SEED	Resets the seed value.
STRING	Generates a random string.
TERMINATE	Terminates generation of random numbers.

Feature	Description
VALUE	Generates a random decimal number between 0 and 1, or between specified values.

Syntax

6.5.1 Description of Features

This section explains each feature of DBMS_RANDOM.

INITIALIZE

- INITIALIZE initializes the generation of random numbers using the specified seed value.
- For *seedVal*, specify a SMALLINT or INTEGER type.

Example

```
PERFORM DBMS_RANDOM.INITIALIZE(999);
```

NORMAL

- NORMAL generates and returns a normally distributed random number.
- The return value type is DOUBLE PRECISION.

Example

```
DECLARE
  d_RunNum   DOUBLE PRECISION;
BEGIN
  d_RunNum := DBMS_RANDOM.NORMAL();
```

RANDOM

- RANDOM generates and returns a random number.
- The data type of the return value is INTEGER.

Example

```
DECLARE
  d_RunInt   INTEGER;
BEGIN
  d_RunInt := DBMS_RANDOM.RANDOM();
```

SEED

- SEED initializes the generation of a random number using the specified seed value or seed string.
- For *seedVal*, specify a SMALLINT or INTEGER type.
- Any string can be specified for the seed string.

Example

```
PERFORM DBMS_RANDOM.SEED('123');
```

STRING

- STRING generates and returns a random string in accordance with the specified display format and string length.

- For the display format *fmt*, specify any of the following values. An error will occur if any other value is specified.

Values that can be specified for the display format

Setting value	Generated string
'u', 'U'	Uppercase letters only
'l', 'L'	Lowercase letters only
'a', 'A'	Mixture of uppercase and lowercase letters
'x', 'X'	Uppercase letters and numbers
'p', 'P'	Any displayable character

- Specify the length of the string to be generated. Specify a SMALLINT or INTEGER type.
- The data type of the return value is TEXT.

Example

```
DECLARE
  d_RunStr   TEXT;
BEGIN
  d_RunStr := DBMS_RANDOM.STRING('a', 20);
```

TERMINATE

- Call TERMINATE to terminate generation of random numbers.

Information

TERMINATE does not do anything, but has been included for compatibility with Oracle databases.

Example

```
PERFORM DBMS_RANDOM.TERMINATE();
```

VALUE

- VALUE generates and returns a random number within the specified range.
- For *min* and *max*, specify a numeric data type. A random number between and inclusive of the minimum value and maximum value is generated.
- If the minimum value and maximum value are omitted, a random decimal number between 0 and 1 will be generated.
- The data type of the return value is DOUBLE PRECISION.

Example

```
DECLARE
  d_RunDbl   DOUBLE PRECISION;
BEGIN
  d_RunDbl := DBMS_RANDOM.VALUE();
```

6.5.2 Usage Example

A usage example of DBMS_RANDOM is shown below.

```
CREATE FUNCTION dbms_random_exe() RETURNS VOID AS $$
DECLARE
 w_rkey VARCHAR(10) := 'rnd111';
 i_rkey INTEGER := 97310;
BEGIN
 PERFORM DBMS_RANDOM.INITIALIZE(i_rkey);
 RAISE NOTICE 'RANDOM -> NORMAL : %', DBMS_RANDOM.NORMAL();
 RAISE NOTICE 'RANDOM -> RANDOM : %', DBMS_RANDOM.RANDOM();
 RAISE NOTICE 'RANDOM -> STRING : %', DBMS_RANDOM.STRING('a',10);
 RAISE NOTICE 'RANDOM -> VALUE  : %', DBMS_RANDOM.VALUE();
 PERFORM DBMS_RANDOM.SEED(w_rkey);
 RAISE NOTICE 'RANDOM -> NORMAL : %', DBMS_RANDOM.NORMAL();
 RAISE NOTICE 'RANDOM -> RANDOM : %', DBMS_RANDOM.RANDOM();
 RAISE NOTICE 'RANDOM -> STRING : %', DBMS_RANDOM.STRING('p',10);
 RAISE NOTICE 'RANDOM -> VALUE  : %', DBMS_RANDOM.VALUE(1,100);
 PERFORM DBMS_RANDOM.TERMINATE();
END;
$$ LANGUAGE plpgsql;
SELECT dbms_random_exe();
DROP FUNCTION dbms_random_exe();
```

6.6 DBMS_UTILITY

Overview

Provides utilities of PL/pgSQL.

Features

Feature	Description
FORMAT_CALL_STACK	Returns the current call stack.
GET_TIME	Returns the number of hundredths of seconds that have elapsed since a point in time in the past.

Syntax

6.6.1 Description of Features

This section explains each feature of DBMS_UTILITY.

FORMAT_CALL_STACK

- FORMAT_CALL_STACK returns the current call stack of PL/pgSQL.
- For the display format fmt, specify any of the following values. An error will occur if any other value is specified.

Values that can be specified for the display format

Setting value	Displayed content
‘o’	Standard-format call stack display (with header)
’s’	Standard-format call stack display (without header)
‘p’	Comma-delimited call stack display (without header)

- If the display format is omitted, display format ‘o’ will be used.
- The data type of the return value is TEXT.

Example

```
DECLARE
   s_StackTrace   TEXT
BEGIN
   s_StackTrace := DBMS_UTILITY.FORMAT_CALL_STACK();
```

Note

If a locale other than English is specified for the message locale, the call stack result may not be retrieved correctly. To correctly retrieve the call stack result, specify English as the message locale.

6.6.2 Usage Example

A usage example of DBMS_UTILITY is shown below.

```
CREATE FUNCTION dbms_utility1_exe() RETURNS VOID AS $$
DECLARE
 s_StackTrace TEXT;
BEGIN
 s_StackTrace := DBMS_UTILITY.FORMAT_CALL_STACK();
 RAISE NOTICE '%', s_StackTrace;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION dbms_utility2_exe() RETURNS VOID AS $$
BEGIN
 PERFORM dbms_utility1_exe();
END;
$$ LANGUAGE plpgsql;

SELECT dbms_utility2_exe();
DROP FUNCTION dbms_utility2_exe();
DROP FUNCTION dbms_utility1_exe();
```

GET_TIME

- GET_TIME returns the current time in 100th’s of a second from a point in time in the past. This function is used for determining elapsed time.

Example

```
DO $$
DECLARE
   start_time integer;
   end_time integer;
BEGIN
   start_time := DBMS_UTILITY.GET_TIME;
   PERFORM pg_sleep(10);
   end_time := DBMS_UTILITY.GET_TIME;
   RAISE NOTICE 'Execution time: % seconds', (end_time - start_time)/100;
END
$$;
```

Note

The function is called twice, the first time at the beginning of some procedural code and the second time at end. Then the first (earlier) number is subtracted from the second (later) number to determine the time elapsed. Must be divided by 100 to report the number of seconds elapsed.

6.7 UTL_FILE

Overview

Text files can be written and read using PL/pgSQL.

To perform these file operations, the directory for the operation target must be registered in the UTL_FILE.UTL_FILE_DIR table beforehand. Use the INSERT statement as the database administrator or a user who has INSERT privileges to register the directory. Also, if the directory is no longer necessary, delete it from the same table. Refer to “Registering and Deleting Directories” for information on the how to register and delete the directory.

Declare the file handler explained hereafter as follows in PL/pgSQL:

```
DECLARE
f UTL_FILE.FILE_TYPE;
```

Features

Feature	Description
FCLOSE	Closes a file.
FCLOSE_ALL	Closes all files open in a session.
FCOPY	Copies a whole file or a contiguous portion thereof.
FFLUSH	Flushes the buffer.
FGETATTR	Retrieves the attributes of a file.
FOPEN	Opens a file.
FREMOVE	Deletes a file.
FRENAME	Renames a file.
GET_LINE	Reads a line from a text file.
IS_OPEN	Checks if a file is open.
NEW_LINE	Writes newline characters.
PUT	Writes a string.
PUT_LINE	Appends a newline character to a string and writes the string.
PUTF	Writes a formatted string.

Syntax

6.7.1 Registering and Deleting Directories

Registering the directory

1 . Check if the directory is already registered (if it is, then step 2 is not necessary).

```
SELECT * FROM UTL_FILE.UTL_FILE_DIR WHERE dir='/home/pgsql';
```

2 . Register the directory.

```
INSERT INTO UTL_FILE.UTL_FILE_DIR VALUES('/home/pgsql');
```

Deleting the directory

```
DELETE FROM UTL_FILE.UTL_FILE_DIR WHERE dir='/home/pgsql';
```

6.7.2 Description

This section explains each feature of UTL_FILE.

FCLOSE

- FCLOSE closes a file that is open.

- Specify an open file handle.
- The value returned is a NULL value.

Example

```
f := UTL_FILE.FCLOSE(f);
```

FCLOSE_ALL

- FCLOSE_ALL closes all files open in a session.
- Files closed with FCLOSE_ALL can no longer be read or written.

Example

```
PERFORM UTL_FILE.FCLOSE_ALL();
```

FCOPY

- FCOPY copies a whole file or a contiguous portion thereof. The whole file is copied if *startLine* and *endLine* are not specified.
- Specify the directory location of the source file.
- Specify the source file.
- Specify the directory where the destination file will be created.
- Specify the name of the destination file.
- Specify the line number at which to begin copying. Specify a value greater than 0. If not specified, 1 is used.
- Specify the line number at which to stop copying. If not specified, the last line number of the file is used.

Example

```
PERFORM UTL_FILE.FCOPY('/home/pgsql', 'regress_pgsql.txt', '/home/pgsql', 'regress_pgsql2.txt');
```

FFLUSH

- FFLUSH forcibly writes the buffer data to a file.
- Specify an open file handle.

Example

```
PERFORM UTL_FILE.FFLUSH(f);
```

FGETATTR

- FGETATTR retrieves file attributes: file existence, file size, and information about the block size of the file.
- Specify the directory where the file exists.
- Specify the relevant file name.
- Use a SELECT statement to obtain the file attributes, which are stored in the fexists, file_length, and blocksize columns.
- The fexists column stores a boolean (TRUE/FALSE) value. If the file exists, fexists is set to TRUE. If the file does not exist, fexists is set to FALSE. The data type of fexists is BOOLEAN.
- The file_length column stores the length of the file in bytes. If the file does not exist, file_length is NULL. The data type of file_length is INTEGER.
- The blocksize column stores the block size of the file in bytes. If the file does not exist, blocksize is NULL. The data type of blocksize is INTEGER.

Example

```
SELECT fexists, file_length, blocksize INTO file_flag, file_len, size
FROM UTL_FILE.FGETATTR('/home/pgsql', 'regress_pgsql.txt');
```

FOPEN

- FOPEN opens a file.
- Specify the directory where the file exists.
- Specify the file name.
- Specify the mode for opening the file:

r: Read

w: Write

a: Add

- Specify the maximum string length (in bytes) that can be processed with one operation. If omitted, the default is 1024. Specify a value from 1 to 32767.
- Up to 50 files per session can be open at the same time.

Example

```
f := UTL_FILE.FOPEN('/home/pgsql','regress_pgsql.txt','r',1024);
```

FREMOVE

- FREMOVE deletes a file.
- Specify the directory where the file exists.
- Specify the file name.

Example

```
PERFORM UTL_FILE.FREMOVE('/home/pgsql', 'regress_pgsql.txt');
```

FRENAME

- FRENAME renames a file.
- Specify the directory location of the source file.
- Specify the source file to be renamed.
- Specify the directory where the renamed file will be created.
- Specify the new name of the file.
- Specify whether to overwrite a file if one exists with the same name and in the same location as the renamed file. If TRUE is specified, the existing file will be overwritten. If FALSE is specified, an error occurs. If omitted, FALSE is set.

See

Refer to “The SQL Language” > “Data Types” > “Boolean Type” in the PostgreSQL Documentation for information on boolean type (TRUE/FALSE) values.

Example

```
PERFORM UTL_FILE.FRENAME('/home/pgsql', 'regress_pgsql.txt', '/home/pgsql',
 'regress_pgsql2.txt', TRUE);
```

GET_LINE

- GET_LINE reads a line from a file.
- Specify the file handle returned by FOPEN using r (read) mode.
- Specify the number of bytes to read from the file. If not specified, the maximum string length specified at FOPEN will be used.
- The return value is the buffer that receives the line read from the file.
- Newline characters are not loaded to the buffer.
- An empty string is returned if a blank line is loaded.
- Specify the maximum length (in bytes) of the data to be read. Specify a value from 1 to 32767. If not specified, the maximum string length specified at FOPEN is set. If no maximum string length is specified at FOPEN, 1024 is set.
- If the line length is greater than the specified number of bytes to read, the remainder of the line is read on the next call.
- A NO_DATA_FOUND exception will occur when trying to read past the last line.

Example

```
buff := UTL_FILE.GET_LINE(f);
```

IS_OPEN

- IS_OPEN checks if a file is open.
- Specify the file handle.
- The return value is a BOOLEAN type. TRUE represents an open state and FALSE represents a closed state.

See

Refer to “The SQL Language” > “Data Types” > “Boolean Type” in the PostgreSQL Documentation for information on boolean type (TRUE/FALSE) values.

Example

```
IF UTL_FILE.IS_OPEN(f) THEN
  PERFORM UTL_FILE.FCLOSE(f);
END IF;
```

NEW_LINE

- NEW_LINE writes one or more newline characters.
- Specify an open file handle.
- Specify the number of newline characters to be written to the file. If omitted, “1” is used.

Example

```
PERFORM UTL_FILE.NEW_LINE(f, 2);
```

PUT

- PUT writes a string to a file.
- Specify the file handle that was opened with FOPEN using w (write) or a (append).
- Specify the string to be written to the file.
- The maximum length (in bytes) of the string to be written is the maximum string length specified at FOPEN.
- PUT does not append a newline character. To append a newline character, execute NEW_LINE.

Example

```
PERFORM UTL_FILE.PUT(f, 'ABC');
```

PUT_LINE

- PUT_LINE appends a newline character to a string and writes the string.
- Specify the file handle that was opened with FOPEN w (write) or a (append).
- Specify whether to forcibly write to the file. If TRUE is specified, file writing is forced. If FALSE is specified, file writing is asynchronous. If omitted, FALSE will be set.
- The maximum length of the string (in bytes) is the maximum string length specified at FOPEN.

Example

```
PERFORM UTL_FILE.PUT_LINE(f, 'ABC', TRUE);
```

PUTF

- PUTF writes a formatted string.
- Specify the file handle that was opened with FOPEN w (write) or a (append).
- Specify the format, which is a string that includes the formatting characters \n and %s.
- The \n in the format is code for a newline character.
- Specify the same number of input values as there are %s in the format. Up to a maximum of five input values can be specified. The %s in the format are replaced with the corresponding input characters. If an input value corresponding to %s is not specified, it is replaced with an empty string.

Example

```
PERFORM UTL_FILE.PUTF(f, '[1=%s, 2=%s, 3=%s, 4=%s, 5=%s]\n', '1', '2', '3', '4', '5');
```

6.7.3 Usage Example

The procedure when using UTL_FILE, and a usage example, are shown below.

1 . Preparation

Before starting a new job that uses UTL_FILE, register the directory in the UTL_FILE.UTL_FILE_DIR table.

Refer to “Registering and Deleting Directories” for information on how to register the directory.

2 . Performing a job

Perform a job that uses UTL_FILE. The example is shown below.

```
CREATE OR REPLACE FUNCTION gen_file(mydir TEXT, infile TEXT, outfile TEXT, copyfile TEXT) RETURNS void AS $$
DECLARE
  v1 VARCHAR(32767);
  inf UTL_FILE.FILE_TYPE;
  otf UTL_FILE.FILE_TYPE;
BEGIN
  inf := UTL_FILE.FOPEN(mydir, infile,'r',256);
  otf := UTL_FILE.FOPEN(mydir, outfile,'w');
  v1 := UTL_FILE.GET_LINE(inf,256);
  PERFORM UTL_FILE.PUT_LINE(otf,v1,TRUE);
  v1 := UTL_FILE.GET_LINE(inf,256);
  PERFORM UTL_FILE.PUTF(otf,'%s\n',v1);
  v1 := UTL_FILE.GET_LINE(inf, 256);
  PERFORM UTL_FILE.PUT(otf,v1);
  PERFORM UTL_FILE.NEW_LINE(otf);
  PERFORM UTL_FILE.FFLUSH(otf);

  inf := UTL_FILE.FCLOSE(inf);
  otf := UTL_FILE.FCLOSE(otf);

  PERFORM UTL_FILE.FCOPY(mydir, outfile, mydir, copyfile, 2, 3);
  PERFORM UTL_FILE.FRENAME(mydir, outfile, mydir, 'rename.txt');

END;
$$ LANGUAGE plpgsql;

SELECT gen_file('/home/pgsql', 'input.txt', 'output.txt', 'copyfile.txt');
```

3 . Post-processing

If you remove a job that uses UTL_FILE, delete the directory information from the UTL_FILE.UTL_FILE_DIR table. Ensure that the directory information is not being used by another job before deleting it.

Refer to “Registering and Deleting Directories” for information on how to delete the directory.

Chapter 7 Transaction behavior

Most of the transaction behavior are exactly same, however the below stuff is not.

7.1 Handled Statement Failure.

```
create table t (a int primary key, b int);
begin;
insert into t values(1,1);
insert into t values(1, 1);
commit;
```

Oracle : commit can succeed. t has 1 row after that.

PostgreSQL: commit failed due to the 2nd insert failed. so t has 0 row.

7.2 DML with Subquery

Case 1:

```
create table dml(a int, b int);
insert into dml values(1, 1), (2,2);

-- session 1:
begin;
delete from dml where a in (select min(a) from dml);

--session 2:
delete from dml where a in (select min(a) from dml);

-- session 1:
commit;
```

In Oracle: 1 row deleted in sess 2. so 0 rows in the dml at last.

In PG : 0 rows are deleted in sess 2, so 1 rows in the dml at last.

Oracle probably detects the min(a) is changed and rollback/rerun the statement.

The same reason can cause the below difference as well.

```
create table su (a int, b int);
insert into su values(1, 1);

- session 1:
begin;
update su set b = 2 where b = 1;

- sess 2:
select * from su where a in (select a from su where b = 1) for update;

- sess 1:
commit;
```

In oracle, 0 row is selected. In PostgreSQL, 1 row (1, 2) is selected.

A best practice would be never use subquery in DML & SLEECT … FOR UPDATE. Even in Oracle, the behavior is inconsistent as well. Oracle between 11.2.0.1 and 11.2.0.3 probably behavior same as Postgres, but other versions not.