Hewlett Packard
Enterprise

May 25, 2018

PostgreSQL 11 New Features

With Examples (Beta 1)

Hewlett-Packard Enterprise Japan Co, Ltd.

Noriyoshi Shinoda

Index

1. About This Document

1.1 Purpose

The purpose of this document is to provide information of the major new features of PostgreSQL 11 Beta 1.

1.2 Audience

This document is written for engineers who already have knowledge of PostgreSQL, such as installation, basic management, etc.

1.3 Scope

This document describes the major difference between PostgreSQL 10 (10.4) and PostgreSQL 11 Beta 1 (11.0). As a general rule, this document examines the features of behavior change. This document does not describe and verify all new features. In particular, the following new features are not included.

- Bug fix
- Performance improvement by changing internal behavior
- Improvement of regression test
- Operability improvement by psql command tab input
- Improvement of pgbench command
- Improvement of documentation, modify typo in the source code
- Refactoring without a change in behavior

1.4 Software Version

The contents of this document have been verified for the following versions and platforms.

Table 1 Version

Software	Version
PostgreSQL	PostgreSQL 10.4 (for comparison) PostgreSQL 11 (11.0) Beta 1 (May 21, 2018 21:14:46)
Operating System	Red Hat Enterprise Linux 7 Update 4 (x86-64)
Configure option	--with-llvm --with-python --with-perl

1.5 Question, comment and Responsibility

The contents of this document are not an official opinion of Hewlett-Packard Enterprise Japan Co, Ltd. The author and affiliation company do not take any responsibility for the problem caused by the mistake of contents. If you have any comments for this document, please contact to Noriyoshi Shinoda (noriyoshi.shinoda@hpe.com) Hewlett-Packard Enterprise Japan Co, Ltd.

1.6 Notation

This document contains examples of the execution of the command or SQL statement. Execution examples are described according to the following rules:

Table 2 Examples notation

Notation	Description
#	Shell prompt for Linux root user
$	Shell prompt for Linux general user
Bold	User input string
postgres=#	psql command prompt for PostgreSQL administrator
postgres=>	psql command prompt for PostgreSQL general user
<u>Underline</u>	Important output items

The syntax is described in the following rules:

Table 3 Syntax rules

Notation	Description
Italic	Replaced by the name of the object which users use, or the other syntax
[ ]	Indicate that it can be omitted
{ A | B }	Indicate that it is possible to select A or B
…	General syntax, it is the same as the previous version

2. New Features Summary

2.1 Overview

More than 160 new features have been added to PostgreSQL 11. Here are some typical new features and benefits. This version focuses to enhance of various new features added in PostgreSQL 10.

2.1 Improve analytic query performance

PostgreSQL 11 has been enhanced to improve the performance of long-running analytical queries. The scope to which parallel query is applied has been greatly expanded. This allows the more efficient use of multiple processors in analytical queries and better throughput. Parallel queries can be executed with the following conditions.

- With Hash Join operator
- With Append operator
- On SELECT INTO statement

At the same time, CREATE TABLE AS SELECT, CREATE INDEX, and CREATE MATERIALIZED VIEW statements that are executed to maintain large tables can also be executed in parallel. For the parallel query, see Section "2.2 Enhancements of Parallel query".

Also, JIT compilation feature by LLVM is now available for SQL statements that occupy CPU for a long time. SQL statements with high expected cost are compiled and executed by LLVM. For details of JIT compilation, refer to "2.5.3 LLVM Integration".

2.2 Improve maintenance task

The following features that can improve operability have been added.

□ Enhancement of partitioned table

A declarative partitioning feature that can manage a table storing a large amount of data by dividing it appeared in PostgreSQL 10. However, the partitioning feature of PostgreSQL 10 had many restrictions. In PostgreSQL 11, many restrictions have been resolved, and the following enhancements were implemented.

- Provided HASH partition as a partitioning method
- Create PRIMARY KEY / UNIQUE KEY for the partitioned table
- Automatic creation of indexes for each partition
- Create a FOREIGN KEY for the partitioned table
- DEFAULT partition that stores value not included in partitions

- Partition-Wise Join, Partition-Wise Aggregation for joining and aggregating partition
- Provided parameter to control partition elimination

□ Improvement of Logical Replication feature

TRUNCATE statement is now propagated in the Logical Decoding environment and the Logical Replication environment. This feature makes synchronization of replication data easier. In PostgreSQL 10, TRUNCATE statement was not transferred to the remote instance.

□ Improved pg_prewarm Contrib module

Pg_prewarm module saves automatically the information of the page cached in the shared buffer and automatically caches the page when restarting the instance.

2.3 Improve reliability

In PostgreSQL 11, integrity checking tools have been enriched to improve reliability.

□ Provides block consistency check tool

Pg_verify_checksums command to check the integrity of the block is provided. This command must be executed after stopping the instance.

□ Confirm block checksum at backup

Pg_basebackup command now checks the checksum of the backup blocks.

□ amcheck module

The Contrib module amcheck is provided. This module can check the consistency of the B-Tree index.

2.4 For application development

PostgreSQL 11 also implements many new features related to application development.

□ PROCEDURE object

A new object PROCEDURE has been added. It is an object similar to FUNCTION without a return value. PROCEDURE can control transactions within a program.

□ Enhancement of PL/pgSQL Language

A CONSTANT clause that defines constant value and a NOT NULL clause that can detect variable initialization misses have been added to PL/pgSQL language.

2.5 Incompatibility

PostgreSQL 11 has changed the following specifications from PostgreSQL 10.

□ WITH clause of CREATE FUNCTION statement

WITH clause of CREATE FUNCTION has been made obsolete.

Example 1 CREATE FUNCTION statement with WITH clause

```
postgres=> CREATE FUNCTION func1() RETURNS INTEGER AS '
  …
' LANGUAGE plpgsql
WITH (isStrict) ;
ERROR:  syntax error at or near "WITH"
LINE 12: WITH (isStrict) ;
```

□ Contrib module

The chkpass module has been deleted.

□ Removed columns from some system catalogs

Some columns have been removed from the pg_class catalog and pg_proc catalog. Please refer to "2.5.1 Changing system catalog".

□ Changed default value

The default value of ssl_compression in libpq connection string has been changed.

□ Incompatibility of TO_NUMBER function

The TO_NUMBER function has been changed to ignore the separator in the template.

Example 2 TO_NUMBER function

```
PostgreSQL 10
postgres=> SELECT to_number('1234', '9,999') ;
to_number
-----------
       134
(1 row)

PostgreSQL 11
postgres=> SELECT to_number('1234', '9,999') ;
to_number
-----------
      1234
(1 row)
```

□ Incompatibility of TO_DATE / TO_NUMBER / TO_TIMESTAMP functions

These functions now skip multibyte characters in templates by character.

Example 3 TO_NUMBER function

```
PostgreSQL 10
postgres=> SELECT to_number('1234', 'あ999') ;
to_number
-----------
         4
(1 row)

PostgreSQL 11
postgres=> SELECT to_number('1234', 'あ999') ;
to_number
-----------
       234
(1 row)
```

3. New Features Detail

3.1 Enhancements of parallel query

In PostgreSQL 11, parallel queries can now be used in many situations.

3.1.1 Parallel Hash

Hash joins and hashing can now be processed in parallel. Parallel Hash or Parallel Hash Join is displayed in the execution plan as follows.

Example 4 Parallel Hash

```
postgres=> EXPLAIN SELECT COUNT(*) FROM hash1 INNER JOIN hash2 ON
hash1.c1 = hash2.c1 ;
                                QUERY PLAN
-----------------------------------------------------------------------
Finalize Aggregate  (cost=368663.94..368663.95 rows=1 width=8)
   ->  Gather  (cost=368663.73..368663.94 rows=2 width=8)
        Workers Planned: 2
        ->     Partial   Aggregate    (cost=367663.73..367663.74   rows=1
width=8)
              ->  Parallel Hash Join  (cost=164082.00..357247.06
rows=4166667 width=0)
                   Hash Cond: (hash2.c1 = hash1.c1)
                   ->  Parallel Seq Scan on hash2  (cost=0.00..95722.40
rows=4166740 width=6)
                   ->  Parallel Hash  (cost=95721.67..95721.67
rows=4166667 width=6)
                         ->  Parallel Seq Scan on hash1
(cost=0.00..95721.67 rows=4166667 width=6)
(9 rows)
```

3.1.2 Parallel Append

Append processing can now be executed in parallel. Parallel Append is displayed in the execution plan.

Example 5 Parallel Append

```
postgres=> EXPLAIN SELECT COUNT(*) FROM data1 UNION ALL SELECT COUNT(*) FROM
data2 ;
                              QUERY PLAN
------------------------------------------------------------------------------
Gather  (cost=180053.25..180054.25 rows=2 width=8)
   Workers Planned: 2
   ->  Parallel Append  (cost=179053.25..179054.05 rows=1 width=8)
         ->  Aggregate  (cost=179054.02..179054.04 rows=1 width=8)
               ->  Seq Scan on data1  (cost=0.00..154054.22 rows=9999922 width=0)
         ->  Aggregate  (cost=179053.25..179053.26 rows=1 width=8)
               ->  Seq Scan on data2  (cost=0.00..154053.60 rows=9999860 width=0)
(7 rows)
```

3.1.3 CREATE TABLE AS SELECT statement

The SELECT processing in CREATE TABLE AS SELECT statement can now be executed as a parallel query. It was serial processing on PostgreSQL 10.

Example 6 Parallelization of CREATE TABLE AS SELECT statement

```
postgres=> EXPLAIN CREATE TABLE para1 AS SELECT COUNT(*) FROM data1 ;
                                  QUERY PLAN
-------------------------------------------------------------------------
 Finalize Aggregate  (cost=11614.55..11614.56 rows=1 width=8)
   ->  Gather  (cost=11614.33..11614.54 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=10614.33..10614.34 rows=1 width=8)
               ->  Parallel Seq Scan on data1  (cost=0.00..9572.67
rows=416667 width=0)
(5 rows)
```

3.1.4 CREATE MATERIALIZED VIEW statement

The SELECT processing of CREATE MATERIALIZED VIEW statement now works as a parallel query. It was serial running on PostgreSQL 10.

Example 7 Parallelization of CREATE MATERIALIZED VIEW statement

```
postgres=> EXPLAIN CREATE MATERIALIZED VIEW mv1 AS SELECT COUNT(*) FROM data1 ;
                                   QUERY PLAN
-----------------------------------------------------------------------------
 Finalize Aggregate  (cost=11614.55..11614.56 rows=1 width=8)
   ->  Gather  (cost=11614.33..11614.54 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=10614.33..10614.34 rows=1 width=8)
               ->  Parallel Seq Scan on data1  (cost=0.00..9572.67 rows=416667
width=0)
(5 rows)
```

3.1.5 SELECT INTO statement

SELECT INTO statement now supports parallel queries.

Example 8 Parallelization of SELECT INTO statement

```
postgres=> EXPLAIN SELECT COUNT(*) INTO val FROM data1 ;
                                   QUERY PLAN
-----------------------------------------------------------------------------
 Finalize Aggregate  (cost=11614.55..11614.56 rows=1 width=8)
   ->  Gather  (cost=11614.33..11614.54 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=10614.33..10614.34 rows=1 width=8)
               ->  Parallel Seq Scan on data1  (cost=0.00..9572.67
rows=416667 width=0)
(5 rows)
```

3.1.6 CREATE INDEX statement

BTree index creation processing can now be executed in parallel.

3.2 Enhancements of partitioned table

In PostgreSQL 11 the following features have been added to the partitioned table.

3.2.1 Hash Partition

HASH was added to partitioning method of the partitioned table. This feature calculates the hash value of column value and performs partitioning by the hash value. Specify PARTITION BY HASH clause in CREATE TABLE statement that creates the partitioned table.

Example 9 Create HASH Partitioned table

```
postgres=> CREATE TABLE hash1 (c1 NUMERIC, c2 VARCHAR(10)) PARTITION BY
HASH(c1) ;
```

For each partition, use FOR VALUES WITH clause to specify the division number in MODULUS clause and the calculation result of the hash value in the REMAINDER clause.

Example 10 Create partition

```
postgres=> CREATE TABLE hash1a PARTITION OF hash1 FOR VALUES
       WITH (MODULUS 4, REMAINDER 0) ;
CREATE TABLE
postgres=> CREATE TABLE hash1b PARTITION OF hash1 FOR VALUES
       WITH (MODULUS 4, REMAINDER 1) ;
CREATE TABLE
postgres=> CREATE TABLE hash1c PARTITION OF hash1 FOR VALUES
       WITH (MODULUS 4, REMAINDER 2) ;
CREATE TABLE
postgres=> CREATE TABLE hash1d PARTITION OF hash1 FOR VALUES
       WITH (MODULUS 4, REMAINDER 3) ;
CREATE TABLE
```

In the REMAINDER clause, specify a value smaller than MODULUS clause. If the number of partitions is smaller than the value specified in MODULES clause, there is no table storing the calculated hash value, so INSERT statement may cause an error.

Example 11 Confirmation of table structure

```
postgres=> \d hash1
                    Table "public.hash1"
 Column |         Type          | Collation | Nullable | Default
--------+-----------------------+-----------+----------+---------
 c1     | numeric               |           |          |
 c2     | character varying(10) |           |          |
Partition key: HASH (c1)
Number of partitions: 4 (Use \d+ to list them.)

postgres=> \d hash1a
                    Table "public.hash1a"
 Column |         Type          | Collation | Nullable | Default
--------+-----------------------+-----------+----------+---------
 c1     | numeric               |           |          |
 c2     | character varying(10) |           |          |
Partition of: hash1 FOR VALUES WITH (modulus 4, remainder 0)
```

Example 12 Error when there are not enough partitions

```
postgres=> INSERT INTO hash1 VALUES (102, 'data1') ;
ERROR:  no partition of relation "hash1" found for row
DETAIL:  Partition key of the failing row contains (c1) = (102).
```

When storing tuples directly on a partition, an INSERT statement that does not match the hash value will fail.

Example 13 Error in the INSERT statement

```
postgres=> INSERT INTO hash1a VALUES (100, 'data1') ;
ERROR:  new row for relation "hash1a" violates partition constraint
DETAIL:  Failing row contains (100, data1).
```

Partition elimination is executed only for matching search.

Example 14 Partition elimination

```
postgres=> EXPLAIN SELECT * FROM hash1 WHERE c1 = 1000 ;
                        QUERY PLAN
---------------------------------------------------------------
Append  (cost=0.00..20.39 rows=4 width=70)
   ->  Seq Scan on hash1c  (cost=0.00..20.38 rows=4 width=70)
         Filter: (c1 = '1000'::numeric)
 (3 rows)
```

The "relpartbound" column in the pg_class catalog and the "partstrat" column in the pg_partitioned_table catalog now store the values corresponding to the hash partitions (h).

Example 15 Catalogs for Hash Partition table

```
postgres=> SELECT pg_get_expr(relpartbound, oid) FROM pg_class WHERE
       relname='hash1a' ;
              pg_get_expr
------------------------------------------
 FOR VALUES WITH (modulus 4, remainder 0)
(1 row)

postgres=> SELECT partstrat FROM pg_partitioned_table WHERE
       partrelid='hash1'::regclass ;
 partstrat
-----------
 h
(1 row)
```

3.2.2 Default partition

The partition table automatically selects the partition where tuples are stored by column values. In PostgreSQL 11 DEFAULT partition feature that stores tuples with column values that are not stored in existing other partitions have been added. To create DEFAULT partition, specify DEFAULT clause instead of the FOR VALUES clause of CREATE TABLE statement. It is a common syntax for RANGE partitioned table and LIST partitioned table.

Example 16 Creating DEFAULT partition (LIST partition)

```
postgres=> CREATE TABLE plist1 (c1 NUMERIC, c2 VARCHAR(10)) PARTITION BY
LIST (c1) ;
CREATE TABLE
postgres=> CREATE TABLE plist11 PARTITION OF plist1 FOR VALUES IN (100) ;
CREATE TABLE
postgres=> CREATE TABLE plist12 PARTITION OF plist1 FOR VALUES IN (200) ;
CREATE TABLE
postgres=> CREATE TABLE plist1d PARTITION OF plist1 DEFAULT ;
CREATE TABLE
```

Even when attaching an existing table to a partitioned table, specify DEFAULT clause instead of FOR VALUES clause of ALTER TABLE statement.

Example 17 Attach DEFAULT partition (LIST partition)

```
postgres=> CREATE TABLE plist2d (c1 NUMERIC, c2 VARCHAR(10)) ;
CREATE TABLE
postgres=> ALTER TABLE plist2 ATTACH PARTITION plist2d DEFAULT ;
ALTER TABLE
```

DEFAULT partition has the following restrictions:

- Multiple DEFAULT partitions cannot be specified in the partitioned table
- Partitions with the same partition-key value as the tuples in DEFAULT partition cannot be added
- When attaching an existing table as DEFAULT partition, all tuples in the table to be attached are checked, and an error occurs if the same value is stored in the existing partition
- DEFAULT partition cannot be specified for HASH partitioned table

In the example below, while trying to add a partition whose c1 column value is 200, a tuple whose value in the c1 column is 200 is already stored in the default partition, it fails.

Example 18 DEFAULT partition restriction

```
postgres=> CREATE TABLE plist1(c1 NUMERIC, c2 VARCHAR(10)) PARTITION BY
LIST(c1) ;
CREATE TABLE
postgres=> CREATE TABLE plist11 PARTITION OF plist1 FOR VALUES IN (100) ;
CREATE TABLE
postgres=> CREATE TABLE plist1d PARTITION OF plist1 DEFAULT ;
CREATE TABLE
postgres=> INSERT INTO plist1 VALUES (100, 'v1'),(200, 'v2') ;
INSERT 0 2
postgres=> CREATE TABLE plist12 PARTITION OF plist1 FOR VALUES IN (200) ;
ERROR:  updated partition constraint for default partition "plist1d"
would be violated by some row
```

3.2.3 Update partition key

It is now possible to execute an UPDATE statement which moves a tuple to another partition. In PostgreSQL 10, UPDATE statement that will change the value of partition-key column failed, but in PostgreSQL 11 it will move the tuple to another partition.

Example 19 UPDATE statement to move the partition (preparation of data)

```
postgres=> CREATE TABLE part1(c1 INT, c2 VARCHAR(10)) PARTITION BY
LIST(c1) ;
CREATE TABLE
postgres=> CREATE TABLE part1v1 PARTITION OF part1 FOR VALUES IN (100) ;
CREATE TABLE
postgres=> CREATE TABLE part1v2 PARTITION OF part1 FOR VALUES IN (200) ;
CREATE TABLE
postgres=> INSERT INTO part1 VALUES (100, 'data100');
INSERT 0 1
postgres=> INSERT INTO part1 VALUES (200, 'data200');
INSERT 0 1
```

Example 20 UPDATE statement to move partitions (update and confirm data)

```
postgres=> UPDATE part1 SET c1=100 WHERE c2='data200' ;
UPDATE 1
postgres=> SELECT * FROM part1v1 ;
 c1  |   c2
-----+---------
 100 | data100
 100 | data200
(2 rows)

postgres=> SELECT * FROM part1v2 ;
 c1 | c2
----+----
(0 rows)
```

□ Trigger

If an UPDATE statement is executed to move a tuple to another partition, the behavior of the trigger is complicated. For the trigger information to be executed, refer to "2.3.8 FOR EACH ROW Trigger".

3.2.4 Automatic index creation

When creating an index on a partitioned table, an index with the same configuration is automatically created for each partition.

Example 21 Create index

```
postgres=> CREATE TABLE part1(c1 NUMERIC, c2 VARCHAR(10)) PARTITION BY
LIST(c1) ;
CREATE TABLE
postgres=> CREATE TABLE part1v1 PARTITION OF part1 FOR VALUES IN (100) ;
CREATE TABLE
postgres=> CREATE TABLE part1v2 PARTITION OF part1 FOR VALUES IN (200) ;
CREATE TABLE
postgres=> CREATE INDEX idx1_part1 ON part1(c2) ;
CREATE INDEX
postgres=> \d part1
                    Table "public.part1"
 Column |         Type          | Collation | Nullable | Default
--------+-----------------------+-----------+----------+---------
 c1     | numeric               |           |          |
 c2     | character varying(10) |           |          |
Partition key: LIST (c1)
Indexes:
    "idx1_part1" btree (c2)
Number of partitions: 2 (Use \d+ to list them.)
```

The name of the automatically generated index is "{partition name}_{column name}_idx". For indexes consisting of multiple columns, column names are concatenated by underscores (_). If an index with the same name already exists, a number will be appended to the end of the index name. It will be abbreviated if the automatically generated index name is too long.

Example 22 Check auto-created index

```
postgres=> \d part1v1
                     Table "public.part1v1"
 Column |         Type          | Collation | Nullable | Default
--------+-----------------------+-----------+----------+---------
 c1     | numeric               |           |          |
 c2     | character varying(10) |           |          |
Partition of: part1 FOR VALUES IN ('100')
Indexes:
    "part1v1_c2_idx" btree (c2)  <- Automatically generated index

postgres=> \d part1v2
                     Table "public.part1v2"
 Column |         Type          | Collation | Nullable | Default
--------+-----------------------+-----------+----------+---------
 c1     | numeric               |           |          |
 c2     | character varying(10) |           |          |
Partition of: part1 FOR VALUES IN ('200')
Indexes:
    "part1v2_c2_idx" btree (c2)   <- Automatically generated index
```

An index is automatically created even if a table is attached as a partition.

Example 23 Attaching a table

```
postgres=> CREATE TABLE part1v3 (LIKE part1) ;
CREATE TABLE
postgres=> ALTER TABLE part1 ATTACH PARTITION part1v3 FOR VALUES IN
(300) ;
ALTER TABLE
postgres=> \d part1v3
                   Table "public.part1v3"
 Column |          Type          | Collation | Nullable | Default
--------+-----------------------+-----------+----------+---------
 c1     | numeric               |           |          |
 c2     | character varying(10) |           |          |
Partition of: part1 FOR VALUES IN ('300')
Indexes:
    "part1v3_c2_idx" btree (c2)   <- Automatically generated index
```

Automatically generated indexes cannot be deleted individually.

Example 24 Individual deletion of automatically created index

```
postgres=> DROP INDEX part1v1_c2_idx ;
ERROR:  cannot drop index part1v1_c2_idx because index idx1_part1
requires it
HINT:  You can drop index idx1_part1 instead.
```

3.2.5 Create unique constraint

Unique constraints (PRIMARY KEY and UNIQUE KEY) can now be specified for the partitioned table.

Example 25 Create the primary key in the partitioned table

```
postgres=> CREATE TABLE part1(c1 NUMERIC, c2 VARCHAR(10)) PARTITION BY
RANGE(c1) ;
CREATE TABLE
postgres=> ALTER TABLE part1 ADD CONSTRAINT pk_part1 PRIMARY KEY (c1) ;
ALTER TABLE
postgres=> \d part1
                    Table "public.part1"
 Column |         Type          | Collation | Nullable | Default
--------+-----------------------+-----------+----------+---------
 c1     | numeric               |           | not null |
 c2     | character varying(10) |           |          |
Partition key: RANGE (c1)
Indexes:
    "pk_part1" PRIMARY KEY, btree (c1)
Number of partitions: 0
```

The primary key setting is automatically added to each partition.

Example 26 Attached table and primary key

```
postgres=> CREATE TABLE part1v1 (LIKE part1) ;
CREATE TABLE
postgres=> ALTER TABLE part1 ATTACH PARTITION part1v1 FOR VALUES FROM
(100) TO (200) ;
ALTER TABLE
postgres=> \d part1v1
                   Table "public.part1v1"
 Column |         Type          | Collation | Nullable | Default
--------+-----------------------+-----------+----------+---------
 c1     | numeric               |           | not null |
 c2     | character varying(10) |           |          |
Partition of: part1 FOR VALUES FROM ('100') TO ('200')
Indexes:
    "part1v1_pkey" PRIMARY KEY, btree (c1)
```

A primary key constraint must include a partition key column. Attempting to create a primary key constraint that does not contain partitioned columns results in an error.

Example 27 Create primary key without partition key column

```
postgres=> CREATE TABLE part2(c1 NUMERIC, c2 NUMERIC, c3 VARCHAR(10))
PARTITION BY RANGE(c1) ;
CREATE TABLE
postgres=> ALTER TABLE part2 ADD CONSTRAINT pk_part2 PRIMARY KEY (c2) ;
ERROR:  insufficient columns in PRIMARY KEY constraint definition
DETAIL:  PRIMARY KEY constraint on table "part2" lacks column "c1" which
is part of the partition key.
```

Attempting to attach a table whose primary key constraint is specified in a column different from the partition table results in an error.

Example 28 Attach a table with a different primary key

```
postgres=> CREATE TABLE part3(c1 NUMERIC PRIMARY KEY, c2 VARCHAR(10))
       PARTITION BY RANGE(c1) ;
CREATE TABLE
postgres=> CREATE TABLE  part3v1 (LIKE part3) ;
CREATE TABLE
postgres=> ALTER TABLE part3v1 ADD CONSTRAINT part3v1_pkey PRIMARY KEY
(c1, c2);
ALTER TABLE
postgres=> ALTER TABLE part3 ATTACH PARTITION part3v1 FOR VALUES FROM
(100) TO (200) ;
ERROR:  multiple primary keys for table "part3v1" are not allowed
```

If the partition is a FOREIGN TABLE, creating the partition fails because it cannot create an index on the partition.

Example 29 Partitions and Unique Constraints with FOREIGN TABLE

```
postgres=> CREATE TABLE part1(c1 INT PRIMARY KEY, c2 VARCHAR(10))
       PARTITION BY RANGE(c1) ;
CREATE TABLE
postgres=> CREATE FOREIGN TABLE part1v1 PARTITION OF part1 FOR VALUES
       FROM (0) TO (1000000) SERVER remhost1 ;
ERROR:  cannot create index on foreign table "part1v1"
```

3.2.6 INSERT ON CONFLICT statement

INSERT ON CONFLICT statement to the partitioned table can now be executed. Both DO NOTHING syntax and DO UPDATE syntax are executable.

Example 30 Partitioned table and INSERT ON CONFLICT statement

```
postgres=>  CREATE  TABLE  part1(c1  INT  PRIMARY  KEY,  c2  VARCHAR(10))
PARTITION BY RANGE(c1) ;
CREATE TABLE
postgres=> CREATE TABLE part1v1 PARTITION OF part1 FOR VALUES FROM (0) TO
(1000) ;
CREATE TABLE
postgres=> CREATE TABLE part1v2 PARTITION OF part1 FOR VALUES FROM (1000)
TO (2000) ;
CREATE TABLE
postgres=> INSERT INTO part1 VALUES (100, 'data1') ON CONFLICT DO NOTHING ;
INSERT 0 1
postgres=>  INSERT  INTO  part1  VALUES  (100,  'update')  ON  CONFLICT  ON
CONSTRAINT part1_pkey DO UPDATE SET c2='update' ;
INSERT 0 1
```

However, updating across partitions is not accepted.

Example 31 INSERT ON CONFLICT statement across partitions

```
postgres=> INSERT INTO part1 VALUES (100, 'update') ON CONFLICT
       ON CONSTRAINT part1_pkey DO UPDATE SET c1=1500 ;
ERROR:  invalid ON UPDATE specification
DETAIL:  The result tuple would appear in a different partition than the
original tuple.
```

3.2.7 Partition-Wise Join / Partition-Wise Aggregate

Partition-Wise Join which joins partition when joining tables and Partition-Wise Aggregate which performs aggregation is supported. These features are off by default but can be enabled by setting the following parameter to "on".

Table 4 Related parameter name

Features	Parameter name	Default value
Partition-Wise Join	enable_partitionwise_join	off
Partition-Wise Aggregate	enable_partitionwise_aggregate	off

The following is an execution plan of the SQL statement that performs the join between the partitioned table with the c1 column as the key. In the default state, each partition is integrated by Append and then joined using Parallel Hash Join.

Example 32 Execution plan in the default state

```
postgres=> EXPLAIN SELECT COUNT(*) FROM pjoin1 p1 INNER JOIN pjoin2 p2 ON p1.c1 =
p2.c1 ;
                                  QUERY PLAN
------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=79745.46..79745.47 rows=1 width=8)
   ->  Gather  (cost=79745.25..79745.46 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=78745.25..78745.26 rows=1 width=8)
               ->  Parallel Hash Join  (cost=36984.68..76661.91 rows=833333 width=0)
                     Hash Cond: (p1.c1 = p2.c1)
                     ->  Parallel Append  (cost=0.00..23312.00 rows=833334 width=6)
                           ->  Parallel Seq Scan on pjoin1v1 p1  (cost=0.00..9572.67
rows=416667 width=6)
                           ->  Parallel Seq Scan on pjoin1v2 p1_1  (cost=0.00..9572.67
rows=416667 width=6)
                     ->  Parallel Hash  (cost=23312.00..23312.00 rows=833334 width=6)
                           ->  Parallel Append  (cost=0.00..23312.00 rows=833334
width=6)
                                 ->  Parallel Seq Scan on pjoin2v1 p2
(cost=0.00..9572.67 rows=416667 width=6)
                                 ->  Parallel Seq Scan on pjoin2v2 p2_1
(cost=0.00..9572.67 rows=416667 width=7)
(13 rows)
```

When Partition-Wise Join feature is enabled, it can be confirmed that coupling is performed first between partitions of two tables.

Example 33 Execution plan with Partition-Wise Join enabled

```
postgres=> SET enable_partitionwise_join = on ;
SET
postgres=> EXPLAIN SELECT COUNT(*) FROM pjoin1 p1 INNER JOIN pjoin2 p2 ON p1.c1 =
p2.c1 ;
                              QUERY PLAN
-----------------------------------------------------------------------------------
 Finalize Aggregate  (cost=75578.78..75578.79 rows=1 width=8)
   ->  Gather  (cost=75578.57..75578.78 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=74578.57..74578.58 rows=1 width=8)
               ->  Parallel Append  (cost=16409.00..72495.23 rows=833334 width=0)
                     ->  Parallel Hash Join  (cost=16409.00..34164.28 rows=416667
width=0)
                           Hash Cond: (p1.c1 = p2.c1)
                           ->  Parallel Seq Scan on pjoin1v1 p1  (cost=0.00..9572.67
rows=416667 width=6)
                           ->  Parallel Hash  (cost=9572.67..9572.67 rows=416667
width=6)
                                 ->  Parallel Seq Scan on pjoin2v1 p2
(cost=0.00..9572.67 rows=416667 width=6)
                     ->  Parallel Hash Join  (cost=16409.00..34164.28 rows=416667
width=0)
                           Hash Cond: (p1_1.c1 = p2_1.c1)
                           ->  Parallel Seq Scan on pjoin1v2 p1_1  (cost=0.00..9572.67
rows=416667 width=6)
                           ->  Parallel Hash  (cost=9572.67..9572.67 rows=416667
width=7)
                                 ->  Parallel Seq Scan on pjoin2v2 p2_1
(cost=0.00..9572.67 rows=416667 width=7)
(15 rows)
```

Furthermore, if the Partition-Wise Aggregate feature is enabled, an execution plan is created for each summary process for each partition.

Example 34 Execution plan with Partition-Wise Aggregate enabled

```
postgres=> SET enable_partitionwise_aggregate = on ;
SET
postgres=> EXPLAIN SELECT COUNT(*) FROM pjoin1 p1 INNER JOIN pjoin2 p2 ON p1.c1 =
p2.c1 ;
                                                 QUERY PLAN
------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=71412.34..71412.35 rows=1 width=8)
   ->  Gather  (cost=36205.95..71412.33 rows=4 width=8)
         Workers Planned: 2
         ->  Parallel Append  (cost=35205.95..70411.93 rows=2 width=8)
               ->  Partial Aggregate  (cost=35205.95..35205.96 rows=1 width=8)
                     ->    Parallel  Hash  Join   (cost=16409.00..34164.28  rows=416667
width=0)
                           Hash Cond: (p1.c1 = p2.c1)
                           ->   Parallel  Seq  Scan  on  pjoin1v1  p1   (cost=0.00..9572.67
rows=416667 width=6)
                           ->     Parallel   Hash     (cost=9572.67..9572.67   rows=416667
width=6)
                                 ->        Parallel    Seq     Scan     on     pjoin2v1     p2
(cost=0.00..9572.67 rows=416667 width=6)
               ->  Partial Aggregate  (cost=35205.95..35205.96 rows=1 width=8)
                     ->    Parallel  Hash  Join   (cost=16409.00..34164.28  rows=416667
width=0)
                           Hash Cond: (p1_1.c1 = p2_1.c1)
                           ->  Parallel Seq Scan on pjoin1v2 p1_1  (cost=0.00..9572.67
rows=416667 width=6)
                           ->     Parallel   Hash     (cost=9572.67..9572.67   rows=416667
width=7)
                                 ->        Parallel    Seq    Scan    on    pjoin2v2    p2_1
(cost=0.00..9572.67 rows=416667 width=7)
(16 rows)
```

Hewlett Packard
Enterprise

3.2.8 FOR EACH ROW trigger

FOR EACH ROW trigger for the partitioned table can now be created. However, BEFORE trigger cannot be created, only AFTER trigger can be created. Also, WHEN clause cannot be specified. When the FOR EACH ROW trigger is executed, TG_TABLE_NAME variable is changed to the partition name where the tuple is actually stored, not the partitioned table name.

As a result of the verification, each trigger is executed in the following order.

Table 5 Simple INSERT statement

Order	Target table	TG_WHEN	TG_OP	TG_LEVEL	TG_TABLE_NAME
1	Partitioned table	BEFORE	INSERT	STATEMENT	Partitioned table
2	Partition	BEFORE	INSERT	ROW	Partition
3	Partitioned table	AFTER	INSERT	ROW	Partition
4	Partition	AFTER	INSERT	ROW	Partition
5	Partitioned table	AFTER	INSERT	STATEMENT	Partitioned table

Table 6 Simple UPDATE statement (No tuple movement between partitions)

Order	Target table	TG_WHEN	TG_OP	TG_LEVEL	TG_TABLE_NAME
1	Partitioned table	BEFORE	UPDATE	STATEMENT	Partitioned table
2	Partition	BEFORE	UPDATE	ROW	Partition
3	Partitioned table	AFTER	UPDATE	ROW	Partition
4	Partition	AFTER	UPDATE	ROW	Partition
5	Partitioned table	AFTER	UPDATE	STATEMENT	Partitioned table

Table 7 Simple DELETE statement

Order	Target table	TG_WHEN	TG_OP	TG_LEVEL	TG_TABLE_NAME
1	Partitioned table	BEFORE	DELETE	STATEMENT	Partitioned table
2	Partition	BEFORE	DELETE	ROW	Partition
3	Partitioned table	AFTER	DELETE	ROW	Partition
4	Partition	AFTER	DELETE	ROW	Partition
5	Partitioned table	AFTER	DELETE	STATEMENT	Partitioned table

Table 8 TRUNCATE statement

Order	Target table	TG_WHEN	TG_OP	TG_LEVEL	TG_TABLE_NAME
1	Partitioned table	BEFORE	TRUNCATE	STATEMENT	Partitioned table
2	Partition	BEFORE	TRUNCATE	STATEMENT	Partition
3	Partitioned table	AFTER	TRUNCATE	STATEMENT	Partitioned table
4	Partition	BEFORE	TRUNCATE	STATEMENT	Partitioned table

Table 9 When a tuple is moved between partitions by executing an UPDATE statement

Order	Target table	TG_WHEN	TG_OP	TG_LEVEL	TG_TABLE_NAME
1	Partitioned table	BEFORE	UPDATE	STATEMENT	Partitioned table
2	Source partition	BEFORE	UPDATE	ROW	Source partition
3	Source partition	BEFORE	DELETE	ROW	Source partition
4	Target partition	BEFORE	INSERT	ROW	Target partition
5	Partitioned table	AFTER	DELETE	ROW	Source partition
6	Source partition	AFTER	DELETE	ROW	Source partition
7	Partitioned table	AFTER	INSERT	ROW	Target partition
8	Target partition	AFTER	INSERT	ROW	Target partition
9	Partitioned table	AFTER	UPDATE	STATEMENT	Partitioned table

Table 10 INSERT ON CONFLICT DO NOTHING (With conflict)

Order	Target table	TG_WHEN	TG_OP	TG_LEVEL	TG_TABLE_NAME
1	Partitioned table	BEFORE	INSERT	STATEMENT	Partitioned table
2	Partition	BEFORE	INSERT	ROW	Partition
3	Partitioned table	AFTER	INSERT	STATEMENT	Partitioned table

3.2.9 FOREIGN KEY support

A Foreign key could not be created in PostgreSQL 10 partitioned table. In PostgreSQL 11 this restriction has been resolved.

Example 35 Partitioned table and Foreign key

```
postgres=> CREATE TABLE cities (city VARCHAR(80) PRIMARY KEY, location
point) ;
CREATE TABLE
postgres=> CREATE TABLE weather (
  city VARCHAR(80) REFERENCES cities(city),
  temp_lo INT,
  temp_hi INT,
  prcp REAL,
  date DATE) PARTITION BY LIST (city) ;
CREATE TABLE
```

3.2.10 Dynamic Partition Elimination

Partition elimination is now executed even when the partition key is specified as a parameter. However, dynamic partition elimination is available only when "generic plan" by PREPARE and EXECUTE statements are used.

Example 36 Executed SQL statement for test

```
CREATE TABLE part4 (c1 INT NOT NULL, c2 INT NOT NULL) PARTITION BY LIST
(c1) ;
CREATE TABLE part4v1 PARTITION OF part4 FOR VALUES IN (1) ;
CREATE TABLE part4v2 PARTITION OF part4 FOR VALUES IN (2) ;
CREATE TABLE part4v3 PARTITION OF part4 FOR VALUES IN (3) ;
PREPARE part4_pl (INT, INT) AS
SELECT c1 FROM part4 WHERE c1 BETWEEN $1 and $2 and c2 < 3 ;
EXECUTE part4_pl (1, 8) ;
EXECUTE part4_pl (1, 8) ;
EXECUTE part4_pl (1, 8) ;
EXECUTE part4_pl (1, 8) ;
EXECUTE part4_pl (1, 8) ;
EXPLAIN (ANALYZE, COSTS OFF, SUMMARY OFF, TIMING OFF) EXECUTE part4_pl
(2, 2) ;
```

Example 37 Execution plan on PostgreSQL 10

```
postgres=> EXPLAIN (ANALYZE, COSTS OFF, SUMMARY OFF, TIMING OFF) EXECUTE
part4_pl (2, 2) ;
                      QUERY PLAN
----------------------------------------------------------
 Append (actual rows=0 loops=1)
   ->  Seq Scan on part4v1 (actual rows=0 loops=1)
         Filter: ((c1 >= $1) AND (c1 <= $2) AND (c2 < 3))
   ->  Seq Scan on part4v2 (actual rows=0 loops=1)
         Filter: ((c1 >= $1) AND (c1 <= $2) AND (c2 < 3))
   ->  Seq Scan on part4v3 (actual rows=0 loops=1)
         Filter: ((c1 >= $1) AND (c1 <= $2) AND (c2 < 3))
(7 rows)
```

Example 38 Execution plan on PostgreSQL 11

```
postgres=> EXPLAIN (ANALYZE, COSTS OFF, SUMMARY OFF, TIMING OFF) EXECUTE
part4_pl (2, 2) ;
                      QUERY PLAN
----------------------------------------------------------
 Append (actual rows=0 loops=1)
   Subplans Removed: 2
   ->  Seq Scan on part4v2 (actual rows=0 loops=1)
         Filter: ((c1 >= $1) AND (c1 <= $2) AND (c2 < 3))
(4 rows)
```

3.2.11 Control Partition Pruning

The partition elimination feature can be controlled by setting the parameter enable_partition_pruning. Setting this parameter to "off" will disable partition pruning in the search process.

Example 39 Control Partition Pruning

```
postgres=> SHOW enable_partition_pruning ;
 enable_partition_pruning
--------------------------
 on
(1 row)

postgres=> EXPLAIN SELECT * FROM part1 WHERE c1=1000 ;
                                 QUERY PLAN
----------------------------------------------------------------------------------
 Append  (cost=0.42..8.45 rows=1 width=10)
   ->  Index Scan using part1v1_pkey on part1v1  (cost=0.42..8.44 rows=1 width=10)
         Index Cond: (c1 = 1000)
(3 rows)

postgres=> SET enable_partition_pruning = off ;
SET
postgres=> EXPLAIN SELECT * FROM part1 WHERE c1=1000 ;
                                 QUERY PLAN
----------------------------------------------------------------------------------
 Append  (cost=0.42..16.90 rows=2 width=10)
   ->  Index Scan using part1v1_pkey on part1v1  (cost=0.42..8.44 rows=1 width=10)
         Index Cond: (c1 = 1000)
   ->  Index Scan using part1v2_pkey on part1v2  (cost=0.42..8.44 rows=1 width=10)
         Index Cond: (c1 = 1000)
(5 rows)
```

3.3 Enhancement of Logical Replication

The following features have been added to PostgreSQL 11 logical replication.

3.3.1 TRUNCATE statement support

The TRUNCATE statement executed on the PUBLICATION node in the Logical Replication environment now propagates to the SUBSCRIPTION node. Along with this, the specification to propagate TRUNCATE statement has been added to WITH clause of CREATE PUBLICATION and ALTER PUBLICATION statements.

Example 40 Specify TRUNCATE clause in CREATE PUBLICATION statement

```
postgres=> CREATE PUBLICATION pub1 FOR TABLE data1
                WITH (publish='INSERT, DELETE, UPDATE, TRUNCATE') ;
CREATE PUBLICATION
```

DDLs other than TRUNCATE statement is not propagated.

3.3.2 pg_replication_slot_advance function

When a conflict occurs in the logical replication environment in PostgreSQL 10, the start LSN of logical replication needs to be specified by executing the pg_replication_origin_advance function on the standby instance to resolve a conflict. In PostgreSQL 11, similar processing can now be executed in the PUBLICATION instance. Execute pg_replication_slot_advance function. This function specifies the replication slot name and LSN.

Example 41 Set replication start LSN to current LSN

```
postgres=# SELECT
        pg_replication_slot_advance('sub1', pg_current_wal_lsn()) ;
pg_replication_slot_advance
-----------------------------
 (sub1,0/5B63E18)
(1 row)
```

3.4 Architecture

3.4.1 Modified catalogs

The following catalogs have been changed.

Table 11 System catalogs with columns added

Catalog name	Added column	Data Type	Description
pg_aggregate	aggfinalmodify	char	Whether the aggfinalfn function changes the value
	aggmfinalmodify	char	Whether the aggmfinalfn function changes the value
pg_attribute	atthasmissing	bool	Have a default value that has not updated the page
	attmissingval	anyarray	Default value not updating page
pg_class	relrewrite	oid	OID when the new relation is created during DDL execution
pg_constraint	conparentid	oid	Parent partition constraint OID
	conincluding	smallint[]	Non-constrained column number list
pg_index	indnkeyatts	smallint	Number of key columns
pg_partitioned_table	partdefid	oid	OID of the default partition
pg_proc	prokind	char	Object kind f: function p: procedure a: aggregate function w: window function
pg_publication	pubtruncate	boolean	TRUNCATE propagation
pg_stat_wal_receiver	sender_host	text	Connection destination hostname
	sender_port	integer	Connection destination port number
information_schema.table_constraints	enforced	yes_or_no	Reserved for future use

Table 12 Catalogs with columns deleted

Catalog name	Deleted column	Description
pg_class	relhaspkey	Have primary key
pg_proc	proisagg	Is aggregate function
	proiswindow	Is Window function

Table 13 Value is now stored information_schema catalog of the schema

Catalog name	Column name	Description
triggers	action_order	Trigger execution order
	action_reference_new_table	"NEW" transition table name
	action_reference_old_table	"OLD" transition table name
tables	table_type	FOREIGN stored for the foreign table (previously FOREIGN TABLE stored)

□ pg_stat_activity catalog

Backend_type column and process name are now synchronized. The value of the backend_type column of the replication launcher process was "background worker" in PostgreSQL 10, but in PostgreSQL 11 it is "logical replication launcher".

Example 42 Refer pg_stat_activity catalog

```
postgres=# SELECT pid,wait_event, backend_type FROM pg_stat_activity ;
  pid  |     wait_event      |         backend_type
-------+---------------------+-----------------------------
 17099 | LogicalLauncherMain | logical replication launcher
 17097 | AutoVacuumMain      | autovacuum launcher
 17101 |                     | client backend
 17095 | BgWriterHibernate   | background writer
 17094 | CheckpointerMain    | checkpointer
 17096 | WalWriterMain       | walwriter
(6 rows)
```

□ pg_attribute catalog

In PostgreSQL 11, columns with the DEFAULT value and NOT NULL constraint can now be added without updating the actual tuples. Along with this, the columns for this feature have been added to the pg_attribute catalog.

Example 43 Refer pg_attribute catalog

```
postgres=> ALTER TABLE cols1 ADD COLUMN c3 INT NOT NULL DEFAULT 10 ;
ALTER TABLE
postgres=> SELECT atthasmissing, attmissingval FROM pg_attribute
       WHERE attname='c3' ;
 atthasmissing | attmissingval
---------------+---------------
 t             | {10}
(1 row)
```

3.4.2 Added ROLE

The following roles were added. These roles are mainly used to allow general users to execute COPY statements and execute the file_fdw Contrib module.

Table 14 Added ROLE

Role name	Description
pg_execute_server_program	Execute programs on the server
pg_read_server_files	Read files on the server
pg_write_server_files	Write files on the server

Example 44 pg_read_server_files role

```
postgres=# GRANT pg_read_server_files TO user1 ;
GRANT ROLE

postgres(user1)=> COPY copy1 FROM '/tmp/copy1.csv' CSV ;
COPY 2000
```

3.4.3 LLVM integration

PostgreSQL 11 supports JIT compilation using LLVM (https://llvm.org/) to speed up long-running SQL statements caused by processor bottlenecks. SQL statements estimated to exceed more than a certain amount of cost are compiled beforehand and then executed.

□ Installation

In order to use LLVM, it is necessary to specify the option --with-llvm of "configure" command at the time of installation. When executing "configure" command, "llvm-config" command and "clang" command must be included in command execution path (or specified in environment variable LLVM_CONFIG and environment variable CLANG).

□ The behavior of JIT compilation

For SQL statements whose total execution cost exceeds the parameter jit_above_cost (default value 100000), the JIT compilation feature by LLVM runs. If this parameter is specified as "-1" or the parameter jit is set to "off", the JIT feature will be disabled. The behavior of JIT compilation processing is changed by inlining (parameter jit_inline_above_cost) and optimization (parameter jit_optimize_above_cost).

□ Execution plan

By executing EXPLAIN statements for SQL statement using the JIT compilation feature, information starting from "JIT:" is displayed.

Example 45 Execution plan of SQL using JIT compilation features

```
postgres=> EXPLAIN ANALYZE SELECT COUNT(*) FROM jit1 ;
                                QUERY PLAN
-----------------------------------------------------------------------
Aggregate  (cost=179053.25..179053.26 rows=1 width=8)
       (actual time=2680.558..2680.559 rows=1 loops=1)
   ->   Seq  Scan  on  jit1   (cost=0.00..154053.60  rows=9999860  width=0)
(actual time=0.022..1424.095 rows=10000000 loops=1)
 Planning Time: 0.024 ms
 JIT:
   Functions: 2
   Generation Time: 1.505 ms
   Inlining: false
   Inlining Time: 0.000 ms
   Optimization: false
   Optimization Time: 0.594 ms
   Emission Time: 8.688 ms
 Execution Time: 2682.166 ms
(12 rows)
```

3.4.4 Predicate locking for GIN / GiST / HASH index

Predicate locks are now available for GIN index, GiST index, HASH index. This feature reduces the lock range, improving the concurrency of SQL statements with multiple sessions.

The following example is the result of verification using the HASH index. In PostgreSQL 10, the lock range is the relation, but in PostgreSQL 11 it is found to be page.

Example 46 Test case

```
postgres=> CREATE TABLE lock1(c1 INT, c2 VARCHAR(10)) ;
CREATE TABLE
postgres=> CREATE INDEX idx1_lock1 ON lock1 USING hash(c1) ;
CREATE INDEX
postgres=> INSERT INTO lock1 VALUES (generate_series(1, 100000),
'data1') ;
INSERT 0 100000
postgres=> BEGIN ISOLATION LEVEL SERIALIZABLE ;
BEGIN
postgres=> SELECT * FROM lock1 WHERE c1=10000 FOR UPDATE ;
  c1   |  c2
-------+-------
 10000 | data1
(1 row)
```

Example 47 Lock status on PostgreSQL 10

```
postgres=> SELECT locktype, relation::regclass, mode FROM pg_locks ;
   locktype    |  relation  |      mode
---------------+------------+-----------------
 relation      | pg_locks   | AccessShareLock
 relation      | idx1_lock1 | AccessShareLock
 relation      | lock1      | RowShareLock
 virtualxid    |            | ExclusiveLock
 transactionid |            | ExclusiveLock
 tuple         | lock1      | SIReadLock
 relation      | idx1_lock1 | SIReadLock
(7 rows)
```

Example 48 Lock status on PostgreSQL 11

```
postgres=> SELECT locktype, relation::regclass, mode FROM pg_locks ;
   locktype    |  relation  |      mode
---------------+------------+-----------------
 relation      | pg_locks   | AccessShareLock
 relation      | idx1_lock1 | AccessShareLock
 relation      | lock1      | RowShareLock
 virtualxid    |            | ExclusiveLock
 transactionid |            | ExclusiveLock
 page          | idx1_lock1 | SIReadLock
 tuple         | lock1      | SIReadLock
(7 rows)
```

3.4.5 Enhanced LDAP authentication

"ldapsearchfilter" attribute has been added to the LDAP authentication parameter described in the pg_hba.conf file. LDAP server can be searched more flexibe than "ldapsearchattribute" attribute.

3.4.6 Extended Query timeout

The timeout of conventional Extended Queries was determined by the time until SYNC messages were sent after sending multiple SQL statements. PostgreSQL 11 now takes into account the execution time of individual SQL statements. The example below is the definition file of pgproto (https://github.com/tatsuo-ishii/pgproto). In PostgreSQL 10 timeout occurred even if "SET statement_timeout = 4s" was set.

Example 49 pgproto definition file

```
# Test case for statement timeout patch.
'Q'     "SET statement_timeout = '4s'"
'Y'

# Execute statement which takes 3 seconds.
'P'     "S1"    "SELECT pg_sleep(3)"  0
'B'     ""      "S1"    0       0       0          PostgreSQL 11
'E'     ""      0                                  Timeout check
'C'     'S'     "S1"

# Execute statement which takes 2 seconds.
'P'     "S2"    "SELECT pg_sleep(2)"  0
'B'     ""      "S2"    0       0       0      PostgreSQL 10
'E'     ""      0                              Timeout check   PostgreSQL 11
'C'     'S'     "S2"                                           Timeout check

# Issue Sync message and terminate
'S'
'Y'
'X'
```

3.4.7 Change of backup label file

A timeline ID is added to the backup_label file created when online backup is executed.

Example 50 backup_label file

```
postgres=# SELECT pg_start_backup(now()::text) ;
pg_start_backup
-----------------
 0/2000060
(1 row)

postgres-# \! cat data/backup_label
START WAL LOCATION: 0/6A000028 (file 00000001000000000000006A)
CHECKPOINT LOCATION: 0/6A000098
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2018-05-25 08:59:10 JST
LABEL: 2018-05-25 08:59:10.132746+09
START TIMELINE: 1                    <- Added line
```

3.4.8 Using Huge Pages under Windows environment

The "Lock Pages In Memory" setting is now available in the Microsoft Windows environment. When parameter huge_pages is set to "try" or "on", shared memory will be reserved in a continuous area. Internally, PAGE_READWRITE, SEC_LARGE_PAGES, and SEC_COMMIT are specified in Windows API CreateFileMapping. Previously only PAGE_READWRITE was specified.

3.4.9 Remove secondary checkpoint information

PostgreSQL 10 saved the last two checkpoint information, but now only the latest checkpoint information is saved.

3.4.10 Error code list

The {INSTALL_DIR}/share/errcodes.txt file has been added. This file contains error codes, levels, macro names etc. of PostgreSQL, PL/pgSQL, and PL/Tcl.

3.5 SQL statement

This section explains new features related to SQL statements.

3.5.1 LOCK TABLE statement

In PostgreSQL 11, it is now possible to specify a view in the LOCK TABLE statement. When locking a view, the same mode lock is applied to the table included in the view definition.

Example 51 Execution of LOCK TABLE statement for the view

```
postgres=> CREATE TABLE data1(c1 INT, c2 VARCHAR(10)) ;
CREATE TABLE
postgres=> CREATE VIEW view1 AS SELECT * FROM data1 ;
CREATE VIEW
postgres=> BEGIN ;
BEGIN
postgres=> LOCK TABLE view1 IN ACCESS EXCLUSIVE MODE ;
LOCK TABLE
postgres=> SELECT relation::regclass, mode FROM pg_locks ;
 relation |        mode
----------+---------------------
 pg_locks | AccessShareLock
          | ExclusiveLock
          | ExclusiveLock
 view1    | AccessExclusiveLock
 data1    | AccessExclusiveLock
          | ExclusiveLock
(6 rows)
```

If the view is nested, the further lower table will be locked. However, the materialized view contained in the view is not locked.

Example 52 Executing LOCK TABLE statement to MATERIALIZED VIEW

```
postgres=> CREATE MATERIALIZED VIEW mview1 AS SELECT * FROM data1 ;
SELECT 0
postgres=> BEGIN ;
BEGIN
postgres=> LOCK TABLE mview1 IN ACCESS EXCLUSIVE MODE ;
ERROR:  "mview1" is not a table or a view
postgres=> ROLLBACK ;
ROLLBACK
postgres=> CREATE VIEW view2 AS SELECT * FROM mview1 ;
CREATE VIEW
postgres=> BEGIN ;
BEGIN
postgres=> LOCK TABLE view2 IN ACCESS EXCLUSIVE MODE ;
LOCK TABLE
postgres=> SELECT relation::regclass, mode FROM pg_locks ;
relation |        mode
----------+---------------------
 pg_locks | AccessShareLock
          | ExclusiveLock
          | ExclusiveLock
 view2    | AccessExclusiveLock
(4 rows)
```

3.5.2 STATISTICS of function index

It is now possible to specify a STATISTICS value for a function index column.

Example 53 STATISTICS value for function index column

```
postgres=> CREATE INDEX idx1_stat1 ON stat1 ((c1 + c2)) ;
CREATE INDEX
postgres=> ALTER INDEX idx1_stat1 ALTER COLUMN 1 SET STATISTICS 1000 ;
ALTER INDEX
postgres=> \d+ idx1_stat1
              Index "public.idx1_stat1"
 Column |  Type   | Definition | Storage | Stats target
--------+---------+------------+---------+--------------
 expr   | numeric | (c1 + c2)  | main    | 1000
btree, for table "public.stat1"
```

3.5.3 VACUUM statement and ANALYZE statement

□ Multiple tables

The VACUUM and ANALYZE statements can now specify multiple tables at the same time.

Example 54 Execution of VACUUM, ANALYZE statement for multiple tables

```
postgres=> VACUUM data1, data2 ;
VACUUM
postgres=> ANALYZE data1, data2 ;
ANALYZE
```

□ Output aggressively

When executing the VACUUM (VERBOSE, FREEZE) statement, "aggressively" is added to the output.

Example 55 Output "aggressively"

```
postgres=> VACUUM (VERBOSE, FREEZE) data1 ;
INFO:  aggressively vacuuming "public.data1"
INFO:  "data1": found 0 removable, 0 nonremovable row versions in 0 out
of 0 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 575
There were 0 unused item pointers.
…
```

3.5.4 Push down the LIMIT clause

The content of the LIMIT clause is now pushed to the subquery when the LIMIT clause is set for the sorted subquery.

Example 56 Execution plan of PostgreSQL 10

```
postgres=> EXPLAIN ANALYZE SELECT * FROM (SELECT * FROM sort1 ORDER BY 1) AS a LIMIT 5 ;
                                        QUERY PLAN
-----------------------------------------------------------------------------------------
 Limit  (cost=56588.00..56588.54 rows=5 width=10) (actual time=1204.947..1204.958 rows=5
loops=1)
   ->  Gather  Merge  (cost=56588.00..153817.09  rows=833334  width=10)  (actual
time=1204.945..1204.951 rows=5 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Sort  (cost=55587.98..56629.65  rows=416667  width=10)  (actual
time=1182.284..1182.712 rows=912 loops=3)
              Sort Key: sort1.c1
              Sort Method: external sort  Disk: 6624kB
              ->  Parallel Seq Scan on sort1  (cost=0.00..9572.67 rows=416667 w
idth=10) (actual time=0.020..481.233 rows=333333 loops=3)
 Planning time: 0.041 ms
 Execution time: 1207.299 ms
(10 rows)
```

Example 57 Execution plan of PostgreSQL 11

```
postgres=> EXPLAIN ANALYZE SELECT * FROM (SELECT * FROM sort1 ORDER BY 1) AS a LIMIT 5 ;
                                QUERY PLAN
------------------------------------------------------------------------------------------
 Limit  (cost=56588.00..56588.54 rows=5 width=10) (actual time=276.900..276.910rows=5
loops=1)
   ->   Gather  Merge   (cost=56588.00..153817.09  rows=833334  width=10)  (actual
time=276.899..276.907 rows=5 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->     Sort     (cost=55587.98..56629.65   rows=416667   width=10)   (actual
time=257.124..257.125 rows=5 loops=3)
               Sort Key: sort1.c1
               Sort Method: top-N heapsort  Memory: 25kB
               Worker 0:  Sort Method: top-N heapsort  Memory: 25kB
               Worker 1:  Sort Method: top-N heapsort  Memory: 25kB
               ->  Parallel Seq Scan on sort1  (cost=0.00..9572.67 rows=416667 width=10)
(actual time=0.020..126.640 rows=333333 loops=3)
 Planning Time: 0.051 ms
 Execution Time: 276.983 ms
(12 rows)
```

3.5.5 CREATE INDEX statement

The following enhancements have been added to the CREATE INDEX statement.

□ INCLUDE clause

An INCLUDE clause to add columns to the index can be specified. This is effective when adding columns which are unrelated to unique constraint to a unique index etc. In the example below the UNIQUE CONSTRAINT is created in columns c1 and c2, but the index includes c3 column.

Example 58 INCLUDE clause with CREATE INDEX statement

```
postgres=> CREATE UNIQUE INDEX idx1_data1 ON data1 (c1, c2) INCLUDE (c3) ;
CREATE INDEX
postgres=> \d+ idx1_data1
              Index "public.idx1_data1"
 Column |  Type   | Definition | Storage | Stats target
--------+---------+------------+---------+--------------
 c1     | integer | c1         | plain   |
 c2     | integer | c2         | plain   |
 c3     | integer |            | plain   |
unique, btree, for table "public.data1"
```

The INCLUDE clause can also be used for the constraint specification part of the CREATE TABLE statement as the CREATE INDEX statement.

Example 59 INCLUDE clause with CREATE TABLE statement

```
postgres=> CREATE TABLE data2 (c1 INT, c2 INT, c3 INT, c4 VARCHAR(10),
CONSTRAINT data2_pkey PRIMARY KEY (c1, c2) INCLUDE (c3)) ;
CREATE TABLE
postgres=> \d data2
                     Table "public.data2"
 Column |         Type          | Collation | Nullable | Default
--------+-----------------------+-----------+----------+---------
 c1     | integer               |           | not null |
 c2     | integer               |           | not null |
 c3     | integer               |           |          |
 c4     | character varying(10) |           |          |
Indexes:
    "data2_pkey" PRIMARY KEY, btree (c1, c2) INCLUDE (c3)
```

□ Surjective indexes

The recheck_on_update option can now be specified in the WITH clause of the CREATE INDEX statement. The default value is "on". This parameter specifies whether to update the function index by HOT.

Example 60 The option of CREATE INDEX statement

```
postgres=> CREATE INDEX idx1_data1 ON data1(upper(c2))
        WITH (recheck_on_update = on) ;
CREATE INDEX
postgres=> \d+ idx1_data1
                 Index "public.idx1_data1"
 Column | Type |   Definition    | Storage  | Stats target
--------+------+-----------------+----------+--------------
 upper  | text | upper(c2::text) | extended |
btree, for table "public.data1"
Options: recheck_on_update=on
```

3.5.6 CREATE TABLE statement

It is now possible to specify the storage parameter toast_tuple_target indicating the threshold value for TOAST conversion. The default value is the same as before.

Example 61 Storage option for CREATE TABLE statement

```
postgres=> CREATE TABLE toast1(c1 INT, c2 VARCHAR(10))
        WITH (toast_tuple_target = 1024) ;
CREATE TABLE
postgres=> \d+ toast1
                                      Table "public.toast1"
 Column |         Type          | Collation | Nullable | Default | Storage  |
--------+-----------------------+-----------+----------+---------+----------+
 c1     | integer               |           |          |         | plain    |
 c2     | character varying(10) |           |          |         | extended |
Options: toast_tuple_target=1024
```

3.5.7 WINDOW function

The GROUPS clause and EXCLUDE clause in the window frame specification can now be specified in the WINDOWS function. Also, float 4 type, float 8 type, and numeric type can be used for RANGE clause.

Syntax

```
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end
[ frame_exclusion ]
```

The following syntax can be specified in the frame_exclusion clause.

Syntax (Part of the frame_exclusion)

```
EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

3.5.8 EXPLAIN statement

Information on sorting is now displayed for each worker when executing a parallel query.

Example 62 Parallel query executed in EXPLAIN statement

```
postgres=> EXPLAIN ANALYZE VERBOSE SELECT * FROM part1 ORDER BY 1 ;
                                 QUERY PLAN
------------------------------------------------------------------------------
Gather Merge  (cost=120509.21..314967.15 rows=1666666 width=10) (actual …)
   Output: part1v2.c1, part1v2.c2
   Workers Planned: 2
   Workers Launched: 2
   ->  Sort  (cost=119509.18..121592.52 rows=833333 width=10) (actual time …)
         Output: part1v2.c1, part1v2.c2
         Sort Key: part1v2.c1
         Sort Method: external merge  Disk: 13736kB
         Worker 0:  Sort Method: external merge  Disk: 12656kB
         Worker 1:  Sort Method: external merge  Disk: 12816kB
         Worker 0: actual time=267.130..357.999 rows=645465 loops=1
         Worker 1: actual time=268.723..350.636 rows=653680 loops=1
         ->  Parallel Append  (cost=0.00..23311.99 rows=833333 width=10) (actual …)
…
```

Hewlett Packard
Enterprise

3.5.9 Functions

The following functions have been added/extended.

□ Added hash functions

Hash functions using SHA-224 / SHA-256 / SHA-384 / SHA-512 has been provided. All usages are the same.

Table 15 Added hash functions

Function name	Description
sha224(bytea)	Calculate the SHA-224 hash value
sha256(bytea)	Calculate the SHA-256 hash value
sha384(bytea)	Calculate the SHA-384 hash value
sha512(bytea)	Calculate the SHA-512 hash value

Example 63 SHA512 hash function

```
postgres=> SELECT sha512('ABC') ;
                                 sha512
------------------------------------------------------------------------
\x397118fdac8d83ad98813c50759c85b8c47565d8268bf10da483153b747a74743a58a
90e85aa9f705ce6984ffc128db567489817e4092d050d8a1cc596ddc119
(1 row)
```

□ json(b)_to_tsvector function

The json(b)_to_tsvector function to convert from JSON type (or JSONB type) to tsvector type has been added.

Example 64 json_to_tsvector function

```
postgres=> SELECT jsonb_to_tsvector('english', '{"a": "The Fat Rats",
"b": 123}'::json, '["string", "numeric"]') ;
    json_to_tsvector
-------------------------
 '123':5 'fat':2 'rat':3
(1 row)
```

□ websearch_to_tsquery function

The websearch_to_tsquery function to convert Web search format string to tsquery type has been added.

Example 65 websearch_to_tsquery function

```
postgres=> SELECT websearch_to_tsquery('english', '"fat rat" or rat') ;
  websearch_to_tsquery
------------------------
 'fat' <-> 'rat' | 'rat'
(1 row)
```

3.5.10 Operator

The following operator has been added.

□ String forward match search

An operator "^ @" to search forward matches for strings has been added. It can be used for the same purpose as "LIKE 'character_string%'" in the WHERE clause.

Syntax

```
search_target ^@ search_text
```

Example 66 ^@ operator

```
postgres=> SELECT usename FROM pg_user WHERE usename ^@ 'po' ;
 usename
----------
 postgres
(1 row)
```

Unlike the LIKE clause, this operator does not use the B-Tree index. Although not verified, the SP-GiST index is available.

3.5.11 Others

□ Cast from JSONB type

Casting from JSONB type to bool type and numerical type became possible.

Example 67 Casting from JSONB type

```
postgres=> SELECT 'true'::jsonb::bool ;
 bool
------
 t
(1 row)
postgres=> SELECT '1.0'::jsonb::float ;
 float8
--------
      1
(1 row)
postgres=> SELECT '12345'::jsonb::int4 ;
 int4
-------
 12345
(1 row)
postgres=> SELECT '12345'::jsonb::numeric ;
 numeric
---------
   12345
(1 row)
```

□ Enhancement of SP-GiST index

The poly_ops operator class that can be created for polygon type has been provided. Also, user-defined methods to compress can be defined.

3.6 PL/pgSQL language

This article explains the enhancement of the PL/pgSQL language and new objects using PL/pgSQL.

3.6.1 PROCEDURE object

In PostgreSQL 11, PROCEDURE was added as a new schema object. PROCEDURE is almost the same object as FUNCTION without a return value. PROCEDURE is created with the CREATE PROCEDURE statement. Unlike the CREATE FUNCTION statement, there are no RETURNS clause, ROWS clause, PARALLEL clause, CALLED ON clause, etc.

Example 68 PROCEDURE creation

```
postgres=> CREATE PROCEDURE proc1(INOUT p1 TEXT) AS
    $$
    BEGIN
        RAISE NOTICE 'Parameter: %', p1 ;
    END ;
    $$
    LANGUAGE plpgsql ;
CREATE PROCEDURE
```

Information on the created procedure can be referred to from the pg_proc catalog as FUNCTION.

To execute PROCEDURE, use the CALL statement instead of the SELECT statement. Parentheses (()) are necessary regardless of the presence or absence of parameters. INOUT can be specified for the PROCEDURE parameter.

Example 69 Execute PROCEDURE (1)

```
postgres=> CALL proc1('test') ;
NOTICE:  Parameter: test
CALL
  p1
------
 test
(1 row)
```

A parameter name can be specified in the CALL statement.

Example 70 Execute PROCEDURE (2)

```
postgres=> CALL proc1(p1=>'test') ;
NOTICE:  Parameter: test
CALL
  p1
------
 test
(1 row)
```

To display a definition of created PROCEDURE from the psql command, use the \df command same as FUNCTION. PROCEDURE shows the Type column as "proc". And the value of type column for FUNCTION was changed to "func". For PROCEDURE, the "prorettype" column of the pg_proc catalog becomes 0.

Example 71 List of PROCEDURE

```
postgres=> \df
                      List of functions
 Schema | Name  | Result data type | Argument data types | Type
--------+-------+------------------+---------------------+------
 public | func1 | text             | text                | func
 public | proc1 |                  | INOUT p1 text       | proc
(2 rows)
```

Even when referring to or changing the PROCEDURE definition from the psql command, it can be executed in the same way as FUNCTION.

Example 72 Display PROCEDURE definition

```
postgres=> \sf proc1
CREATE OR REPLACE PROCEDURE public.proc1(INOUT p1 text)
 LANGUAGE plpgsql
AS $procedure$
    BEGIN
        RAISE NOTICE 'Parameter: %', $1 ;
    END ;
    $procedure$
```

□ Transaction control

Within procedures, it is possible to control transactions. In PL/pgSQL procedure, COMMIT statement and ROLLBACK statement can be used. It is now possible to control transactions within cursor loops.

Example 73 Transaction control in PROCEDURE

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plpgsql
AS $$
BEGIN
    FOR i IN 0..9 LOOP
        INSERT INTO test1 (a) VALUES (i) ;
        IF i % 2 = 0 THEN
            COMMIT ;
        ELSE
            ROLLBACK ;
        END IF ;
    END LOOP ;
END ;
$$ ;
```

The syntax for transaction control is also provided in languages other than PL/pgSQL.

PL/Python

plpy.commit() / plpy.rollback()

PL/Perl

spi_commit() / spi_rollback()

PL/Tcl

commit / rollback

3.6.2 Enhancement of variable definition

The following enhancements have been added to PL/pgSQL.

□ CONSTANT variables

Constants can now be declared by adding the CONSTANT clause to the variable. If an initial value is not specified, no error occurs and the value of the variable is set as NULL.

Syntax

```
DECLARE variable_name CONSTANT data_type [ := initial_value ] ;
```

Example 74 CONSTANT clause and changing value error

```
postgres=> do $$
    DECLARE
       cons1 CONSTANT NUMERIC := 1 ;
    BEGIN
       cons1 := 2 ;
    END ;
    $$ ;
ERROR:  variable "cons1" is declared CONSTANT
LINE 5:   cons1 := 2 ;
          ^
```

□ NOT NULL variables

It is now possible to specify a NOT NULL constraint on a variable. An initial value is required at the time of declaration. An error occurs if an initial value is not specified. Also, assignment of NULL value will result in an error.

Syntax

```
DECLARE variable_name data_type NOT NULL := initial_value ;
```

Example 75 NOT NULL clause and initial value

```
postgres=> do $$
    DECLARE
       nn1 NUMERIC NOT NULL ;
    BEGIN
       RAISE NOTICE 'nn1 = %', nn1 ;
    END ;
    $$ ;
ERROR:  variable "nn1" must have a default value, since it's declared
NOT NULL
LINE 3:   nn1 NUMERIC NOT NULL ;
```

□ SET TRANSACTION statement

In PostgreSQL 11, It is possible to execute the SET TRANSACTION statement in the PL/pgSQL block.

Example 76 SET TRANSACTION statement

```
postgres=> DO LANGUAGE plpgsql $$
  BEGIN
      PERFORM 1 ;
      COMMIT ;
      SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ;
      PERFORM 1 ;
      RAISE INFO '%', current_setting('transaction_isolation') ;
      COMMIT ;
  END ;
  $$ ;
INFO:  repeatable read
DO
```

3.7 Configuration parameters

In PostgreSQL 11 the following parameters have been changed.

3.7.1 Added parameters

The following parameters have been added.

Table 16 Added parameters

Parameter name	Description (context)	Default
data_directory_mode	Database cluster protection mode (internal)	0700
enable_parallel_append	Enabled Parallel Append feature (user)	on
enable_parallel_hash	Enabled Parallel Hash feature (user)	on
enable_partition_pruning	Enabled partition pruning (user)	on
enable_partitionwise_aggregate	Enabled Partition-Wise Join feature (user)	off
enable_partitionwise_join	Enabled Partition-Wise Aggregate feature (user)	off
jit	Enabled JIT compilation feature (user)	on
jit_above_cost	The Cost to enable JIT compile function (user)	100000
jit_debugging_support	Enable debugger at JIT compilation (superuser-backend)	off
jit_dump_bitcode	Add LLVM bit code (superuser)	off
jit_expressions	Enable JIT expression (user)	on
jit_inline_above_cost	The cost to decide whether to make JIT inlining (user)	500000
jit_optimize_above_cost	The cost to decide whether to perform JIT optimization (user)	500000
jit_profiling_support	Enabled perf profiler (superuser-backend)	off
jit_provider	JIT provider name (postmaster)	llvmjit
jit_tuple_deforming	Enabled JIT tuple deforming (user)	on
max_parallel_maintenance_workers	Maximum number of parallel workers used in maintenance processing (user)	2
parallel_leader_participation	Change the behavior of the leader process (user)	on
ssl_passphrase_command	Command to get SSL connection passphrases (sighup)	''

Parameter name	Description (context)	Default
ssl_passphrase_command_supports_reload	Whether to use ssl_passphrase_command when reloading (sighup)	off
vacuum_cleanup_index_scale_factor	The ratio of the number of INSERTs that perform index cleanup (user)	0.1

□ parallel_leader_participation parameter

When a parallel query is executed, the leader process which coordinates the entire query and the worker processes cooperate with each other. In PostgreSQL 10, the leader process also executed the plan on the Gather and Gather Merge nodes. If this parameter is set to "off", the leader process will not perform the same process as the worker process.

The following example searches for table data of 10 million tuples. In both examples, two worker processes are running. In the first example, the portion of Parallel Seq Scan is executed three times with 3,333,333 tuples. In the second example where the parameter parallel_leader_participation is changed to "off", it is executed in 2 loops with 5,000,000 tuples.

Example 77 Parameter parallel_leader_participation = on

```
postgres=> EXPLAIN ANALYZE SELECT COUNT(*) FROM data1 ;
                       QUERY PLAN
-----------------------------------------------------------------------
Finalize Aggregate  (cost=107139.46..107139.47 rows=1 width=8) (actual
time=8296.629..8296.630 rows=1 loops=1)
   ->  Gather  (cost=107139.25..107139.46 rows=2 width=8) (actual
time=8295.776..8296.622 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate  (cost=106139.25..106139.26 rows=1
width=8) (actual time=8276.140..8276.141 rows=1 loops=3)
             ->  Parallel Seq Scan on data1  (cost=0.00..95722.40
rows=4166740 width=0) (actual time=0.096..4168.115 rows=3333333
loops=3)  Planning time: 0.026 ms  Execution time: 8296.693 ms
(8 rows)
```

Example 78 Parameter parallel_leader_participation = off

```
postgres=> SET parallel_leader_participation = off ;
SET
postgres=> EXPLAIN ANALYZE SELECT COUNT(*) FROM data1 ;
                    QUERY PLAN
-----------------------------------------------------------------------
Finalize Aggregate  (cost=117556.31..117556.32 rows=1 width=8) (actual
time=8448.965..8448.965 rows=1 loops=1)
   ->  Gather  (cost=117556.10..117556.30 rows=2 width=8) (actual
time=8448.
278..8448.959 rows=2 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate  (cost=116556.10..116556.10 rows=1
width=8) (actual time=8441.034..8441.035 rows=1 loops=2)
              ->  Parallel Seq Scan on data1  (cost=0.00..104055.88
rows=5000088 width=0) (actual time=0.214..4423.363 rows=5000000
loops=2)  Planning time: 0.026 ms  Execution time: 8449.088 ms
(8 rows)
```

3.7.2 Changed parameters

The setting range and options were changed for the following configuration parameters.

Table 17 Changed configuration parameters

Parameter name	Changes
log_parser_stats	Memory information will be added to the log when the setting value is changed to "on"
log_statement_stats	
log_planner_stats	
log_executor_stats	
wal_segment_size	UNIT of pg_settings catalog changed from 8kB to B

□ log_parser_stats parameter

When the value is set to "on", the information output is increased. Additional output information varies depending on the operating system. The example below is for Linux.

Example 79 Parameter log_parser_stats (Linux)

```
postgres=# SHOW log_parser_stats ;
 log_parser_stats
------------------
 on
(1 row)

$ tail -11 data/log/postgresql-2018-05-25_092853.log
STATEMENT:  SHOW log_parser_stats ;
LOG:  REWRITER STATISTICS
DETAIL:  ! system usage stats:
        !       0.000002 s user, 0.000001 s system, 0.000001 s elapsed
        !       [0.370726 s user, 0.064474 s system total]
        !       12840 kB max resident size
        !       0/0 [984/0] filesystem blocks in/out
        !       0/0 [1/3208] page faults/reclaims, 0 [0] swaps
        !       0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
        !       0/0 [28/4] voluntary/involuntary context switches
```

3.7.3 Parameters with default values changed

The default values of the following configuration parameters have been changed.

Table 18 Parameters with default values changed

Parameter name	PostgreSQL 10	PostgreSQL 11
server_version	10.4	11beta1
server_version_num	100004	110000

3.7.4 Obsoleted parameter

The following parameter is obsolete.

Table 19 Obsoleted parameter

Parameter name	Alternative
replacement_sort_tuples	None

3.7.5 Authentication parameter change

□ Instance connect string

The following parameters have been added to the instance connection string of the Libpq library.

Table 20 Added parameters

Parameter name	Default	Description
scram_channel_binding	tls-unique	Channel / binding type of SCRAM authentication.
replication	false	Specify whether to use replication protocol true: Physical replication database: logical replication

The default value of ssl_compression parameter has been changed to "off".

□ LDAP Authentication parameter

The ldapschema parameter has been added for LDAP authentication. Specifying "ldaps" for the parameter uses LDAP over SSL.

3.8 Utilities

This section explains the major enhancements of utility commands.

3.8.1 psql command

The following features have been added to the psql command.

□ \gdesc command

A \gdesc command has been added to display the column name and data type of the most recently executed query.

Example 80 \gdesc command

```
postgres=> SELECT COUNT(*) AS num_settings FROM pg_settings ;
 num_settings
--------------
          272
(1 row)

postgres=> \gdesc
    Column    |  Type
--------------+--------
 num_settings | bigint
(1 row)
```

□ Pager specification

As the environment variable to specify the pager used by the psql command, PSQL_PAGER is the priority than PAGER.

□ Query status

Variables concerning the execution status of SQL statements are now available. Combined with the \if command, SQL statement execution results can now be handled in SQL scripts. The LAST_ERROR_MESSAGE and LAST_ERROR_SQLSTATE variables maintain their contents even if another SQL statement succeeds after an error.

Table 21 The Variable indicating execution result of SQL statement

Variable name	Description
ERROR	True if the most recently executed SQL failed
LAST_ERROR_MESSAGE	The error message of the last error
LAST_ERROR_SQLSTATE	The error code of the last error
ROW_COUNT	Number of tuples processed by the most recently executed SQL statement
SQLSTATE	Execution status code of the most recently executed SQL statement

Example 81 SQL execution result variable

```
postgres=> SELECT * FROM not_exists ;
ERROR:  relation "not_exists" does not exist
LINE 1: SELECT * FROM not_exists ;
                      ^
postgres=> \echo :ERROR
true
postgres=> \echo :SQLSTATE
42P01
postgres=> UPDATE data1 SET c2='update' ;
UPDATE 10
postgres=> \echo :ROW_COUNT
10
postgres=> \echo :LAST_ERROR_MESSAGE
relation "not_exists" does not exist
postgres=> \echo :LAST_ERROR_SQLSTATE
42P01
```

□ exit and quit command

The "exit" and "quit" command can now be used to terminate the psql command. Previously, the only \q was available.

□ Parameter definition check

The syntax ":{?variable_name}" to check whether variables are defined is now available.

Example 82 String output declaration

```
postgres=> \set TESTVAL 1
postgres=> \echo :{?TESTVAL}
TRUE
```

3.8.2 ECPG command

The following enhancements were implemented in ECPG.

□ Oracle mode

The option "-C ORACLE" to change the behavior when retrieving string type data has been added. Here, explains the comparison between using the default value and specifying -C ORACLE. First, declare an area for character string output and an indicator as shown below.

Example 83 String output declaration

```
EXEC SQL BEGIN DECLARE SECTION ;
  char shortstr[5] ;
  short shstr_ind = 0 ;
EXEC SQL END DECLARE SECTION ;
```

Output data of VARCHAR(10) type column to variable "char shortstr[5]".

Example 84 Fetch the column data

```
EXEC SQL FETCH C into :shortstr :shstr_ind ;
```

Depending on the length of the original data, check the contents output to the variable "shortstr".

Table 22 Default setting

Table data	shortstr value	shortstr_ind value	Note
''	''	0	shortstr[0] = NULL
'AB'	'AB'	0	shortstr[4] = not updated
'ABCD'	'ABCD'	0	shortstr[4] = NULL
'ABCDE'	'ABCDE'	0	shortstr[4] = 'E'
'ABCDEF'	'ABCDE'	6	shortstr[4] = 'E'
'ABCDEFGHIJ'	'ABCDE'	10	shortstr[4] = 'E'

Table 23 -C ORACLE setting

Table data	shortstr value	shortstr_ind value	Note
''	' '	-1	shortstr variable is filled with spaces
'AB'	'AB '	0	shortstr variable is filled with spaces
'ABCD'	'ABCD'	0	shortstr[4] = NULL
'ABCDE'	'ABCD'	5	shortstr[4] = NULL
'ABCDEF'	'ABCD'	6	shortstr[4] = NULL
'ABCDEFGHIJ'	'ABCD'	10	shortstr[4] = NULL

As in the above example, if "-C ORACLE" is specified, space is added if the variable area is surplus, and NULL is appended to the last of the variable.

□ DO CONTINUE statement

The DO CONTINUE clause can now be specified in the WHENEVER statement. This causes the flow of control to return to the beginning of loop when the result of statement matches the condition specified in WHENEVER.

Example 85 Specify DO CONTINUE clause

```
main() {
   …
   EXEC SQL WHENEVER SQLERROR DO CONTINUE ;
   while (1) {
        EXEC SQL FETCH c INTO :val ;
   }
   …
}
```

Hewlett Packard
Enterprise

3.8.3 initdb command

The following enhancements were implemented to the initdb command.

□ --wal-segsize option

A "--wal-segsize" option to specify the size of the WAL file has been added. This option can be specified to a multiple of 2, and from 1 to 1024 in megabytes. The default value is 16 MB as before.
Along with this, the "--with-wal-segsize" option of the "configure" command executed at the time of installation is obsolete.

Example 86 initdb --wal-segsize option

```
$ initdb --wal-segsize=128 data
The files belonging to this database system will be owned by user
"postgres".
This user must also own the server process.

The database cluster will be initialized with locale "en_US.UTF-8".
The default database encoding has accordingly been set to "UTF8".
…
```

□ --allow-group-access option

Added --allow-group-access option (or -g option) to allow group access to database cluster access mode. If this option is specified, the group's read / execute permission is added to the directory protection mode of the database cluster, and the database cluster parameter data_directory_mode is changed to 0750.

Example 87 initdb –allow-group-access option

```
$ initdb --allow-group-access data
The files belonging to this database system will be owned by user
"postgres".
This user must also own the server process.
…
    pg_ctl -D data -l logfile start

$ ls -ld data
drwxr-x---. 19 postgres postgres 4096 May 25 13:32 data
```

3.8.4 pg_dump / pg_dumpall command

The following options have been added to the pg_dump and pg_dumpall commands.

□ --load-via-partition-root option

The --load-via-partition-root option has been added to the pg_dump and pg_dumpall commands. When this option is specified, data is loaded via the root table of the partition table, not the individual partitions when loading the data.

□ --encoding option

The --encoding option (or -E parameter) has been added to the pg_dumpall command. As with the pg_dump command, this option specifies the character encoding of the output data.

Example 88 pg_dumpall command with specifying the character encoding

```
$ pg_dumpall -E utf8 > dump.sql
```

□ --no-comments option

The --no-comments option has been added to the pg_dump command. If this option is specified, COMMENT will not be loaded.

3.8.5 pg_receivewal command

The following options have been added to the pg_receivewal command.

□ --endpos option

Added --endpos option (or -E option) to specify the LSN where the command will stop has been added. This is the same feature as the --endpos option added to the pg_recvlogical command in PostgreSQL 10.

□ --no-sync option

The --no-sync option was added. When this option is specified, the sync system call is not executed when writing data.

Hewlett Packard
Enterprise

3.8.6 pg_ctl command

The feature to send SIGKILL signal from pg_ctl command was implemented.

Example 89 pg_ctl command

```
$ pg_ctl --help
pg_ctl is a utility to initialize, start, stop, or control a PostgreSQL
server.

Usage:
  pg_ctl init[db] [-D DATADIR] [-s] [-o OPTIONS]
  …
  pg_ctl promote  [-D DATADIR] [-W] [-t SECS] [-s]
  pg_ctl kill     SIGNALNAME PID

Common options:
 …
Allowed signal names for kill:
  ABRT HUP INT KILL QUIT TERM USR1 USR2
```

3.8.7 pg_basebackup command

The following enhancements were implemented for the pg_basebackup command.

□ --no-verify-checksum option

The pg_basebackup command now checks the checksum of the backed-up block. The --no-verify-checksum option skips the checksum verification process. If an error occurs in the checksum, the pg_basebackup command exits with a nonzero value.

□ --create-slot option

The pg_basebackup command now has a --create-slot option (or -C option) to create a replication slot. This option is used with the --slot option. The created replication slots are maintained even after the pg_basebackup command completes. If a replication slot with the same name already exists, the pg_basebackup command displays an error message and exits.

Example 90 pg_basebackup command

```
$ pg_basebackup --create-slot --slot=test1 -v -D back
pg_basebackup: initiating base backup, waiting for checkpoint to complete
pg_basebackup: checkpoint completed
pg_basebackup: write-ahead log start point: 0/2000028 on timeline 1
pg_basebackup: starting background WAL receiver
pg_basebackup: created replication slot "test1"
pg_basebackup: write-ahead log end point: 0/2000130
pg_basebackup: waiting for background process to finish streaming ...
pg_basebackup: base backup completed
$
```

□ Operation change in batch mode

When specifying the --progress option to the pg_basebackup command and executing it from batch mode (redirect from a shell script to a file), a line feed code "\n" is output instead of the "\r".

Example 91 Batch mode operation

```
$ pg_basebackup --progres -D back &> back.out
$ cat back.out
waiting for checkpoint
    0/74635 kB (0%), 0/1 tablespace
74644/74644 kB (100%), 0/1 tablespace
74644/74644 kB (100%), 0/1 tablespace
74644/74644 kB (100%), 1/1 tablespace
```

□ Exclude pg_internal.init file

The pg_internal.init file is excluded from the backup target of the pg_basebackup command.

□ Exclude UNLOGGED table

The UNLOGGED and TEMPORARY tables are excluded from the transfer data.

3.8.8 pg_resetwal / pg_controldata command

The --wal-segsize option to specify the size of the WAL file has been added to the pg_resetwal command. Also, the pg_resetwal and pg_controldata commands have long name options corresponding to existing short options.

Table 24 Added options for pg_resetwal command

Short option	Added option
-c	--commit-timestamp-ids
-D	--pgdata
-e	--epoch
-f	--force
-l	--next-wal-file
-m	--multixact-ids
-n	--dry-run
-o	--next-oid
-O	--multixact-offset
-x	--next-transaction-id
None	--wal-segsize

Table 25 Option for pg_controldata command

Short option	Added option
-D	--pgdata

3.8.9 configure command

The following options have been changed in the configure command.

□ --with-wal-segsize option

The --with-wal-segsize option has been obsoleted. The size of the WAL file can be specified by --wal-segsize option of initdb command.

□ --with-llvm option

The --with-llvm option to provide JIT compilation feature with LLVM has been added. When specifying this option, llvm-config command and clang command must be included in the command search path (or set the environment variable LLVM_CONFIG and the environment variable CLANG).

3.8.10 pg_verify_checksums command

Pg_verify_checksums has been added as a command for checksum consistency check from outside the database. This command cannot be executed when the instance is running. In the example below, checksums are confirmed for only certain files, indicating that some blocks do not match checksums.

Example 92 Usage of pg_verify_checksums command

```
$ pg_verify_checksums --help
pg_verify_checksums verifies page level checksums in offline PostgreSQL database
cluster.

Usage:
  pg_verify_checksums [OPTION] [DATADIR]

Options:
 [-D] DATADIR    data directory
  -r relfilenode check only relation with specified relfilenode
  -d             debug output, listing all checked blocks
  -V, --version  output version information, then exit
  -?, --help     show this help, then exit

If no data directory (DATADIR) is specified, the environment variable PGDATA
is used.

Report bugs to <pgsql-bugs@postgresql.org>.
```

Example 93 Execute pg_verify_checksums command

```
$ pg_verify_checksums -D data -r 16410
pg_verify_checksums:     checksum     verification     failed     in     file
"data/base/16385/16410", block 0: calculated checksum 42D6 but expected 84E0
Checksum scan completed
Data checksum version: 1
Files scanned:  1
Blocks scanned: 1
Bad checksums:  1
$
```

3.9 Contrib modules

Describes the new features of the Contrib modules.

3.9.1 adminpack

Previously, functions included in this module required SUPERUSER privilege. But now, general users are also executable if allowed by the GRANT statement.

3.9.2 amcheck

A new Contrib module amcheck has been added. The only B-Tree index can be checked. In the following example, the index idx1_data1 whose integrity has been destroyed is checked. Also, check the HASH index and an error has occurred.

Example 94 Use amcheck module

```
postgres=# CREATE EXTENSION amcheck ;
CREATE EXTENSION
postgres=# SELECT bt_index_check('idx1_data1') ;
ERROR:  invalid page in block 0 of relation base/16385/16479

postgres=# CREATE INDEX idxh1_data1 ON data1 USING hash (c1) ;
CREATE INDEX
postgres=# SELECT bt_index_check('idxh1_data1') ;
ERROR:  only B-Tree indexes are supported as targets for verification
DETAIL:  Relation "idxh1_data1" is not a B-Tree index.
```

Table 26 Provided functions

Function name	Description
bt_index_check	Consistency check for B-Tree index
bt_index_parent_check	Consistency check of B-Tree index including parentage relationship

3.9.3 btree_gin

B-Tree GIN indexes can now be created for bool, bpchar, and uuid columns.

Example 95 Create B-Tree Gin index

```
postgres=# CREATE EXTENSION btree_gin ;
CREATE EXTENSION

postgres=> CREATE TABLE gintbl1(c1 bool, c2 bpchar(10), c3 uuid) ;
CREATE TABLE
postgres=> CREATE INDEX idx1_gintbl1 ON gintbl1 USING gin(c1) ;
CREATE INDEX
postgres=> CREATE INDEX idx2_gintbl1 ON gintbl1 USING gin(c2) ;
CREATE INDEX
postgres=> CREATE INDEX idx3_gintbl1 ON gintbl1 USING gin(c3) ;
CREATE INDEX
```

3.9.4 citext

Operator class citext_pattern_ops for the index has been added. Operators ~<~, ~<=~, ~>~, ~>=~ can be used for the citext type similarly to text type. Conventionally, it was converted to text type and executed.

Example 96 Execution plan on PostgreSQL 10

```
postgres=> EXPLAIN ANALYZE SELECT * FROM citext1 WHERE c2 ~<~ '111' ;
                            QUERY PLAN
----------------------------------------------------------------------
Seq Scan on citext1  (cost=0.00..17906.00 rows=11574 width=12) (actual
time=0.011..76.417 rows=12225 loops=1)
   Filter: ((c2)::text ~<~ '111'::text)
   Rows Removed by Filter: 987775
 Planning time: 0.046 ms
 Execution time: 76.881 ms
(5 rows)
```

Example 97 Execution plan on PostgreSQL 11

```
postgres=> CREATE INDEX idx1_citext1 ON
               citext1(c2 citext_pattern_ops) ;
CREATE INDEX
postgres=> EXPLAIN ANALYZE SELECT * FROM citext1 WHERE c2 ~<~ '111' ;
                           QUERY PLAN
-----------------------------------------------------------------
Bitmap Heap Scan on citext1  (cost=44.03..603.77 rows=1499 width=11)
(actual time=0.146..0.243 rows=1225 loops=1)
   Recheck Cond: (c2 ~<~ '111'::citext)
   Heap Blocks: exact=10
   ->  Bitmap Index Scan on idx1_citext1  (cost=0.00..43.66 rows=1499
width=0) (actual time=0.142..0.142 rows=1225 loops=1)
         Index Cond: (c2 ~<~ '111'::citext)
 Planning time: 0.107 ms
 Execution time: 0.309 ms
(7 rows)
```

3.9.5 cube / seg

Index Only Scan is now executable with the GiST index.

3.9.6 jsonb_plpython

A new Contrib module jsonb_plpython has been added. The --with-python option must be specified in the configure command for installation. The module name specified in the CREATE EXTENSION statement is "jsonb_plpythonu", "jsonb_plpython2u", or "jsonb_plpython3u". Jsonb can be specified in the TRANSFORM clause of the CREATE FUNCTION statement.

Example 98 jsonb_plpython module

```
postgres=# CREATE EXTENSION jsonb_plpythonu CASCADE ;
NOTICE:  installing required extension "plpythonu"
CREATE EXTENSION
postgres=# CREATE OR REPLACE FUNCTION fpython(val jsonb)
        RETURNS jsonb
        TRANSFORM FOR TYPE jsonb
        LANGUAGE plpythonu
        AS $$
          return (val) ;
        $$ ;
CREATE FUNCTION
postgres=# SELECT fpython('{"1":1,"example": null}'::jsonb) ;
         fpython
---------------------------
 {"1": 1, "example": null}
(1 row)
```

3.9.7 jsonb_plperl

A new Contrib module jsonb_plperl has been added. The --with-perl option must be specified in the "configure" command for installation. To use it, specify jsonb in the TRANSFORM clause of the CREATE FUNCTION statement.

Example 99 jsonb_plperl module

```
postgres=# CREATE EXTENSION jsonb_plperl CASCADE ;
NOTICE:  installing required extension "plperl"
CREATE EXTENSION

postgres=> CREATE OR REPLACE FUNCTION fperl(val jsonb)
       RETURNS jsonb
       TRANSFORM FOR TYPE jsonb
       LANGUAGE plperl
       AS $$
          return $_[0] ;
       $$ ;
CREATE FUNCTION
postgres=> SELECT fperl('{"1":1,"example": null}'::jsonb) ;
          fperl
---------------------------
 {"1": 1, "example": null}
(1 row)
```

3.9.8 pageinspect

The last_cleanup_num_tuples column was added to the output of bt_metap function.

Example 100 bt_metap function

```
postgres=# SELECT * FROM bt_metap('idx1_data1') ;
-[ RECORD 1 ]-----------+-------
magic                   | 340322
version                 | 3
root                    | 3
level                   | 1
fastroot                | 3
fastlevel               | 1
oldest_xact             | 0
last_cleanup_num_tuples | -1  <- Added column
```

3.9.9 pg_prewarm

The feature which automatically loads pages that existed in shared memory when the instance was running into shared memory at instance startup has been added. When "pg_prewarm" is specified for the parameter shared_preload_libraries and the instance is started, the background worker process "postgres: autoprewarm master" is started automatically. The autoprewarm master process loads the blocks previously stored in the shared memory into the shared memory at instance startup. After that, this process saves the block information on the shared memory in a file at regular intervals. If the parameter max_worker_processes is 0, the background worker process will not start.

□ File

The file to which the autoprewarm process saves the block information loaded on shared memory is {PGDATA}/autoprewarm.blocks. This file is in text format and saves information such as a database, tablespace, filenode, block etc... While updating the file, it writes to the {PGDATA}/autoprewarm.blocks.tmp file and the file name is changed.

□ Parameter pg_prewarm.autoprewarm

Specifying the default value "on" for this parameter enables the automatic prewarm feature.

□ Parameter pg_prewarm.autoprewarm_interval

Specify the minimum interval, in seconds, at which block information on shared memory is saved periodically. The default value is 300 seconds (5 minutes). When this parameter is set to 0, periodic shared memory saving processing is not performed. This process will be performed only when the instance is stopped.

3.9.10 pg_trgm

The function strict_word_similarity has been added. This function is similar to the word_similarity function, but it matches the extent boundary to the word boundary.

Example 101 strict_word_similarity function

```
postgres=> SELECT strict_word_similarity('word', 'two words'),
               similarity('word', 'words') ;
 strict_word_similarity | similarity
------------------------+------------
               0.571429 |   0.571429
(1 row)
```

3.9.11 postgres_fdw

The following enhancements have been added to the postgres_fdw module.

□ Update remote partition

Even if the partition is FOREIGN TABLE, it is now possible to update tuples via partition table. It is also possible to insert tuples by COPY statement.

Example 102 Partition with FOREIGN TABLE

```
postgres=> CREATE TABLE part1(c1 INT, c2 VARCHAR(10)) PARTITION BY RANGE(c1) ;
CREATE TABLE
postgres=> CREATE FOREIGN TABLE part1v1 PARTITION OF part1 FOR VALUES FROM (0)
TO (1000000) SERVER remhost1 ;
CREATE FOREIGN TABLE
postgres=> CREATE FOREIGN TABLE part1v2 PARTITION OF part1 FOR VALUES FROM
(1000000) TO (2000000) SERVER remhost1 ;
```

Example 103 The behavior of PostgreSQL 10

```
postgres=> INSERT INTO part1 VALUES (100, 'data1') ;
ERROR:  cannot route inserted tuples to a foreign table
```

Example 104 The behavior of PostgreSQL 11

```
postgres=> INSERT INTO part1 VALUES (100, 'data1') ;
INSERT 0 1
```

□ Change superuser's checking method

The connection check to the remote instance is now done by the USER MAPPING user, not the session user.

□ Behavior when joining by UPDATE / DELETE statement

DELETE or UPDATE statements which join FOREIGN TABLEs on the same FOREIGN SERVER can now be pushed down to remote instances.

Example 105 Execution plan of PostgreSQL 10

```
postgres=> EXPLAIN VERBOSE DELETE FROM fdata2 USING fdata1
        WHERE fdata1.c1 = fdata2.c2 AND fdata1.c1 % 10 = 2 ;
                                QUERY PLAN
------------------------------------------------------------------------
 Delete on public.fdata2  (cost=100.00..292.72 rows=84 width=38)
   Remote SQL: DELETE FROM public.fdata2 WHERE ctid = $1
   ->  Foreign Scan  (cost=100.00..292.72 rows=84 width=38)
         Output: fdata2.ctid, fdata1.*
         Relations: (public.fdata2) INNER JOIN (public.fdata1)
         Remote SQL: SELECT r1.ctid, CASE WHEN (r2.*)::text IS NOT NULL
THEN ROW(r2.c1, r2.c2) END FROM (public.fdata2 r1 INNER JOIN pub
lic.fdata1 r2 ON (((r1.c2 = r2.c1)) AND (((r2.c1 % 10) = 2)))) FOR UPDATE
OF r1
         ->  Hash Join  (cost=230.70..322.85 rows=84 width=38)
               Output: fdata2.ctid, fdata1.*
               Hash Cond: (fdata2.c2 = fdata1.c1)
               ->  Foreign Scan on public.fdata2  (cost=100.00..182.27
rows=2409 width=10)
                     Output: fdata2.ctid, fdata2.c2
                     Remote SQL: SELECT c2, ctid FROM public.fdata2 FOR
UPDATE
               ->  Hash  (cost=130.61..130.61 rows=7 width=36)
                     Output: fdata1.*, fdata1.c1
                     ->  Foreign Scan on public.fdata1  (cost=100.00..130.61
rows=7 width=36)
                           Output: fdata1.*, fdata1.c1
…
```

Example 106 Execution plan of PostgreSQL 11

```
postgres=> EXPLAIN VERBOSE DELETE FROM fdata2 USING fdata1
        WHERE fdata1.c1 = fdata2.c2 AND fdata1.c1 % 10 = 2 ;
                                QUERY PLAN
--------------------------------------------------------------------------
 Delete on public.fdata2  (cost=100.00..292.72 rows=84 width=38)
   ->  Foreign Delete  (cost=100.00..292.72 rows=84 width=38)
         Remote SQL: DELETE FROM public.fdata2 r1 USING public.fdata1 r2
WHERE ((r1.c2 = r2.c1)) AND (((r2.c1 % 10) = 2))
(3 rows)
```

URL list

The following websites are the references to create this material.

- Release Notes
  https://www.postgresql.org/docs/devel/static/release-11.html
- Commitfests
  https://commitfest.postgresql.org/
- PostgreSQL 11 Beta Manual
  https://www.postgresql.org/docs/11/static/index.html
- GitHub
  https://github.com/postgres/postgres
- Open source developer based in Japan (Michael Paquier)
  http://paquier.xyz/
- Qiita (Nuko@Yokohama)
  http://qiita.com/nuko_yokohama
- PostgreSQL Deep Dive
  http://pgsqldeepdive.blogspot.jp/ (Satoshi Nagayasu)
- pgsql-hackers Mailing list
  https://www.postgresql.org/list/pgsql-hackers/
- Announce of PostgreSQL 11 Beta 1
  https://www.postgresql.org/about/news/1855/
- PostgreSQL 11 Roadmap
  https://wiki.postgresql.org/wiki/PostgreSQL11_Roadmap
- Slack - postgresql-jp
  https://postgresql-jp.slack.com/

Change history

Change history

Version	Date	Author	Description
0.1	Apr 19, 2018	Noriyoshi Shinoda	Create internal review version Reviewer: Tomoo Takahashi Takahiro Kitayama (Hewlett Packard Enterprise Japan)
0.2	May 24, 2018	Noriyoshi Shinoda	Recheck completed to respond to for PostgreSQL 11 Beta 1 Reviewer: Satoshi Nagayasu (Uptime Technologies, LCC.)
1.0	May 25, 2018	Noriyoshi Shinoda	Create a published version