Hewlett Packard
Enterprise

2018 年 5 月 25 日

PostgreSQL 11 新機能検証結果（Beta 1）

日本ヒューレット・パッカード株式会社
篠田典良

Hewlett Packard
Enterprise

目次

Hewlett Packard
Enterprise

1. 本文書について

1.1 本文書の概要

本文書はオープンソース RDBMS である PostgreSQL 11 Beta 1 の主な新機能について検証した結果をまとめた文書です。

1.2 本文書の対象読者

本文書は、既にある程度 PostgreSQL に関する知識を持っているエンジニア向けに記述しています。インストール、基本的な管理等は実施できることを前提としています。

1.3 本文書の範囲

本文書は PostgreSQL 10 (10.4) と PostgreSQL 11 Beta 1 (11.0) の主な差分を記載しています。原則として利用者が見て変化がわかる機能について調査しています。すべての新機能について記載および検証しているわけではありません。特に以下の新機能は含みません。

- バグ解消
- 内部動作の変更によるパフォーマンス向上
- レグレッション・テストの改善
- psql コマンドのタブ入力による操作性改善
- pgbench コマンドの改善
- ドキュメントの改善、ソース内の Typo 修正
- 動作に変更がないリファクタリング

1.4 本文書の対応バージョン

本文書は以下のバージョンとプラットフォームを対象として検証を行っています。

表 1　対象バージョン

種別	バージョン
データベース製品	PostgreSQL 10.4 (比較対象) PostgreSQL 11 (11.0) Beta 1 (2018/5/21 21:14:46)
オペレーティング・システム	Red Hat Enterprise Linux 7 Update 4 (x86-64)
Configure オプション	--with-llvm --with-python --with-perl

Hewlett Packard
Enterprise

1.5 本文書に対する質問・意見および責任

本文書の内容は日本ヒューレット・パッカード株式会社の公式見解ではありません。また内容の間違いにより生じた問題について作成者および所属企業は責任を負いません。本文書に対するご意見等ありましたら作成者 篠田典良（Mail: noriyoshi.shinoda@hpe.com）までお知らせください。

1.6 表記

本文書内にはコマンドや SQL 文の実行例および構文の説明が含まれます。実行例は以下のルールで記載しています。

表 2 例の表記ルール

表記	説明
#	Linux root ユーザーのプロンプト
$	Linux 一般ユーザーのプロンプト
太字	ユーザーが入力する文字列
postgres=#	PostgreSQL 管理者が利用する psql コマンド・プロンプト
postgres=>	PostgreSQL 一般ユーザーが利用する psql コマンド・プロンプト
下線部	特に注目すべき項目
<<以下省略>>	より多くの情報が出力されるが文書内では省略していることを示す
<<途中省略>>	より多くの情報が出力されるが文書内では省略していることを示す

構文は以下のルールで記載しています。

表 3 構文の表記ルール

表記	説明
斜体	ユーザーが利用するオブジェクトの名前やその他の構文に置換
[ ]	省略できる構文であることを示す
{ A | B }	A または B を選択できることを示す
…	旧バージョンと同一である一般的な構文

Hewlett Packard
Enterprise

2. 新機能概要

PostgreSQL 11 には 160 以上の新機能が追加されました。ここでは代表的な新機能と利点について説明します。このバージョンでは PostgreSQL 10 で追加された様々な機能に対して適用範囲を広めることに重点が置かれています。

2.1 分析系クエリーの性能向上

PostgreSQL 11 には長時間実行される分析系クエリーの性能を向上させる機能が拡充されました。PostgreSQL 11 では、大規模なテーブルの検索を並列化するパラレル・クエリーが適用される範囲が大幅に拡張されました。このため分析系クエリーで複数プロセッサがより効率的に使用され、スループットが向上します。以下のような処理でパラレル・クエリーが実行される可能性があります。

- ハッシュ結合
- アペンド処理
- SELECT INTO 文の実行

同時に、大規模なテーブルのメンテナンスを行うために実行する CREATE TABLE AS SELECT 文、CREATE INDEX 文、CREATE MATERIALIZED VIEW 文も並列に実行される可能性があります。パラレル・クエリーについては「2.2 パラレル・クエリーの拡張」を参照してください。

また、長時間 CPU を占有する SQL 文については、LLVM による JIT コンパイル機能が動作するようになりました。予想コストが大きい SQL 文は、LLVM によりコンパイルされて実行されます。JIT コンパイルについては「2.5.3 LLVM の統合」を参照してください。

2.2 運用性を向上させる新機能

運用性を向上できる以下の機能が追加されました。

□ パーティショニング機能の拡張

大量のデータを格納するテーブルを分割して管理できるパーティショニング機能は PostgreSQL 10 で実装されました。しかし PostgreSQL 10 のパーティショニング機能には多くの制約がありました。PostgreSQL 11 では多くの制約が解消し、以下のような拡張が実装されました。

- パーティション方法として HASH パーティションの提供
- パーティション・テーブルに対する主キー／一意キーの設定
- 各パーティションに対するインデックスの自動作成

- 外部キーの設定
- パーティションに含まれない値を格納する DEFAULT パーティション
- パーティション同士の結合や集計を行う Partition-Wise Join、Partition-Wise Aggregation
- パーティション・プルーニングを制御するパラメーターの提供

□ 論理レプリケーション機能の拡張

レプリケーション環境では論理デコーディング環境、論理レプリケーション環境に TRUNCATE 文を伝播する機能が追加されました。このためレプリケーション・データの同期がより簡単になります。PostgreSQL 10 では TRUNCATE 文はリモート・インスタンスに転送されませんでした。

□ pg_prewarm Contrib モジュールの拡張

共有バッファ上にキャッシュされたページの情報を自動的に保存し、インスタンス再起動時に該当ページを自動的にキャッシュすることができるようになりました。

2.3 信頼性を向上させる新機能

PostgreSQL 11 では信頼性を向上させるために整合性のチェック・ツールが充実しました。

□ ブロック整合性チェック・ツールの提供

ブロックの整合性をチェックする pg_verify_checksums コマンドが提供されます。このコマンドはインスタンスを停止した後で実行する必要があります。

□ バックアップ時のブロック・チェックサム確認

pg_basebackup コマンドはブロックのチェックサムを確認するようになりました。

□ amcheck モジュール

B-Tree インデックスの整合性をチェックできる Contrib モジュール amcheck が提供されました。

2.4 アプリケーション開発に関する新機能

PostgreSQL 11 にはアプリケーション開発に関する新機能も数多く実装されました。

□ PROCEDURE オブジェクト

新しいオブジェクト PROCEDURE が追加されました。戻り値の無い FUNCTION と同様のオブジェクトです。PROCEDURE は、プログラム内でトランザクションの制御を行うことができます。

□ PL/pgSQL の拡張

PL/pgSQL 内で定数を定義する CONSTANT 句や、変数の初期化漏れを検知できる NOT NULL 句が追加されました。

2.5 非互換

PostgreSQL 11 は PostgreSQL 10 から以下の仕様が変更されました。

□ CREATE FUNCTION 文の WITH 句

CREATE FUNCTION の WITH 句はサポートされなくなりました。

例 1 WITH 句付きの CREATE FUNCTION 文

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

□ Contrib モジュール

chkpass モジュールが削除されました。

□ 一部のシステム・カタログから列が削除されました

pg_class カタログと pg_proc カタログからは一部の列が削除されました。「2.5.1 システム・カタログの変更」を参照してください。

□ デフォルト値の変更

libpq 接続文字列の ssl_compression はデフォルト値が変更されました。

□ TO_NUMBER 関数の仕様変更

TO_NUMBER 関数ではテンプレート内のセパレータを無視するように変更されました。

例 2 TO_NUMBER 関数

```
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)
```

□ TO_DATE / TO_NUMBER / TO_TIMESTAMP 関数の仕様変更

これらの関数ではテンプレート内のマルチバイト文字を文字単位でスキップするようになりました。

例 3 TO_NUMBER 関数

```
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. 新機能解説

3.1 パラレル・クエリーの拡張

PostgreSQL 11 ではパラレル・クエリーを利用できる範囲が拡大しました。

3.1.1 Parallel Hash

ハッシュ結合やハッシュ処理を並列に行うことができるようになりました。実行計画上は Parallel Hash または Parallel Hash Join と表示されます。

例 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 処理を並列に行うことができるようになりました。実行計画には Parallel Append と表示されます。

例 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 文

CREATE TABLE AS SELECT 文の検索部分がパラレル・クエリーとして動作できるようになりました。PostgreSQL 10 ではシリアル実行されていました。

例 6 CREATE TABLE AS SELECT 文のパラレル化

```
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 文

CREATE MATERIALIZED VIEW 文の検索部分がパラレル・クエリーとして動作できるようになりました。PostgreSQL 10 ではシリアル実行されていました。

例 7 CREATE MATERIALIZED VIEW 文のパラレル化

```
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 文

SELECT INTO 文がパラレル・クエリーに対応しました。

例 8 SELECT INTO 文のパラレル化

```
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 文

BTree インデックスの作成処理がパラレルに実行できるようになりました。

3.2 パーティション機能の拡張

PostgreSQL 11 ではパーティション・テーブルに以下の機能が追加されました。

3.2.1 ハッシュ・パーティション

PostgreSQL 10 から追加されたパーティション・テーブルのパーティション化メソッドにハッシュ（HASH）が追加されました。列値のハッシュ値によりパーティショニングを行う機能です。

パーティション・テーブルを作成する CREATE TABLE 文に PARTITION BY HASH 句を指定します。

例 9 HASH パーティション・テーブルの作成

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

各パーティションには FOR VALUES WITH 句を使って MODULUS 句に分割数と、ハッシュ値の計算結果を REMAINDER 句に指定します。

例 10 パーティションの作成

```
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
```

REMAINDER 句には MODULUS 句よりも小さい値を指定します。パーティションの個数が MODULES 句で指定した値より小さい場合、計算したハッシュ値を格納するテーブルが存在しないことになるため、INSERT 文がエラーになる可能性があります。

例 11 テーブル構造の確認

```
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)
```

例 12 パーティションが足りない場合のエラー

```
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).
```

パーティションに直接データを格納する場合、ハッシュ値に合致しない INSERT 文は失敗します。

例 13 INSERT 文のエラー

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

パーティション・プルーニングは一致検索の場合に限り行われます。

例 14 パーティション・プルーニング

```
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)
```

pg_class カタログの relpartbound 列と pg_partitioned_table カタログの partstrat 列はハッシュ・パーティションに対応する値（h）が格納されるようになりました。

例 15 ハッシュ・パーティションに対応するカタログ

```
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 デフォルト・パーティション

PostgreSQL 10 の新機能であるパーティション・テーブルは、列値によってタプルを格納されるパーティションを自動的に選択します。PostgreSQL 11 では既存のパーティションに含まれない列値を持つタプルを格納するデフォルト・パーティションの機能が追加されました。デフォルト・パーティションを作成する方法は、CREATE TABLE 文の FOR VALUES 句の代わりに DEFAULT 句を指定するだけです。RANGE パーティションと LIST パーティションで共通の構文です。

例 16 DEFAULT パーティションの作成（LIST パーティション）

```
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
```

既存のテーブルをパーティションに追加する場合も、ALTER TABLE 文の FOR VALUES 句の代わりに DEFAULT 句を指定します。

例 17 DEFAULT パーティションのアタッチ（LIST パーティション）

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

デフォルト・パーティションには以下の制限があります。

- デフォルト・パーティションはパーティション・テーブルに複数指定できません。
- デフォルト・パーティション内のタプルと同じパーティション・キー値を含むパーティションは追加できません。
- 既存のテーブルをデフォルト・パーティションとしてアタッチする場合、アタッチするテーブル内の全タプルがチェックされ、既存のパーティションに同じ値が格納されているとエラーになります。
- HASH パーティションにはデフォルト・パーティションを指定できません。

下記の例では、c1 列の値が 200 であるタプルが既にデフォルト・パーティションに格納されている状態で、c1 列の値が 200 であるパーティションを追加しようとして失敗しています。

例 18 DEFAULT パーティションの制約

```
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 パーティション・キーの更新

タプルがパーティションを移動する UPDATE 文を実行することができるようになりました。従来はパーティション・キーの条件に合致しない列値への UPDATE 文は失敗していましたが、PostgreSQL 11 では他のパーティションへの移動が行われるようになります。

例 19 パーティションを移動する UPDATE 文（データの準備）

```
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
```

例 20 パーティションを移動する UPDATE 文（データの更新と確認）

```
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)
```

□ トリガー

パーティションをまたがる UPDATE 文が実行された場合、トリガーの動作が複雑になります。実行されるトリガーの情報は「2.3.8 FOR EACH ROW トリガー」を参照してください。

3.2.4 インデックスの自動作成

パーティション・テーブルにインデックスを作成すると、各パーティションに同一構成のインデックスが自動的に作成されるようになりました。

例 21 インデックスの作成

```
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.)
```

自動生成されるインデックスの名前は「{パーティション名}_{列名}_idx」です。複数列から構成されるインデックスの場合は列名がアンダースコア（_）により連結されます。同一名称のインデックスが既に存在した場合、インデックス名の末尾に数字が付けられます。自動生成されるインデックス名が長すぎる場合には短縮されます。

例 22 自動作成されたインデックスの確認

```
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)  ←自動生成されたインデックス

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)  ←自動生成されたインデックス
```

パーティションとしてテーブルをアタッチした場合でも自動的にインデックスが作成されます。

例 23 テーブルのアタッチ

```
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)  ←自動生成されたインデックス
```

自動生成されたインデックスは個別に削除できません。

例 24 自動作成されたインデックスの個別削除

```
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 一意制約の作成

パーティション・テーブルに対して一意制約（PRIMARY KEY および UNIQUE KEY）が指定できるようになりました。

例 25 パーティション・テーブルに主キー作成

```
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
```

各パーティションにも自動的に主キーの設定が追加されます。

例 26 アタッチされたテーブルと主キー

```
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)
```

主キー制約にはパーティション対象列を含める必要があります。パーティション対象列を含まない主キー制約を作成しようとするとエラーになります。

例 27 パーティション対象列を含まない主キー作成

```
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.
```

パーティション・テーブルとは異なる列に主キー制約が指定されたテーブルをアタッチしようとするとエラーになります。

例 28 異なる主キーを持つテーブルをアタッチ

```
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
```

パーティションが外部テーブルの場合、パーティションに対してインデックスが作成できないため、パーティションの作成に失敗します。

例 29 外部テーブルを使ったパーティションと一意制約

```
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 文

パーティション・テーブルに対する INSERT ON CONFLICT 文が実行できるようになりました。DO NOTHING 構文と DO UPDATE 構文のどちらも実行できます。

例 30 パーティション・テーブルと INSERT ON CONFLICT 文

```
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
```

ただし、パーティションをまたがる更新は受け付けられません。

例 31 パーティションをまたがる INSERT ON CONFLICT 文

```
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

テーブルの結合時にパーティション単位で結合を行う Partition-Wise Join と集計を行う Partition-Wise Aggregate がサポートされます。これらの機能はデフォルトではオフになっていますが、下記のパラメーターを on に設定することで有効にすることができます。

表 4　関係するパラメーター名

機能	パラメーター	デフォルト値
Partition-Wise Join	enable_partitionwise_join	off
Partition-Wise Aggregate	enable_partitionwise_aggregate	off

下記は c1 列でパーティション化されたテーブル間の結合を行う SQL 文の実行計画です。
デフォルト状態では、各パーティションを Append により統合してから Parallel Hash Join を使って結合しています。

例 32 デフォルト状態の実行計画

```
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)
```

Hewlett Packard Enterprise

Partition-Wise Join 機能を有効にすると、２つのテーブルのパーティション間で結合が先に行われることがわかります。

例 33 PARTITION-WISE JOIN を有効にした実行計画

```
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)
```

更に Partition-Wise Aggregate 機能を有効にすると集計処理もパーティション単位で行う実行計画が作成されます。

例 34 PARTITION-WISE AGGREGATGE を有効にした実行計画

```
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)
```

3.2.8 FOR EACH ROW トリガー

パーティション・テーブルに対する FOR EACH ROW トリガーが設定できるようになりました。ただし BEFORE トリガーは設定できず、AFTER トリガーのみになります。また WHEN 句を指定することができません。FOR EACH ROW トリガー実行時には、TG_TABLE_NAME 変数がパーティション・テーブル名ではなく、実際にデータが格納されるパーティション名に変更されます。

検証の結果、各トリガーは以下の順番で実行されます。

表 5　単純な INSERT 文

順番	トリガー対象テーブル	TG_WHEN	TG_OP	TG_LEVEL	TG_TABLE_NAME
1	パーティションテーブル	BEFORE	INSERT	STATEMENT	パーティションテーブル
2	パーティション	BEFORE	INSERT	ROW	パーティション
3	パーティション	AFTER	INSERT	ROW	パーティション
4	パーティション	AFTER	INSERT	ROW	パーティション
5	パーティションテーブル	AFTER	INSERT	STATEMENT	パーティションテーブル

表 6　単純な UPDATE 文（タプルのパーティション間移動なし）

順番	トリガー対象テーブル	TG_WHEN	TG_OP	TG_LEVEL	TG_TABLE_NAME
1	パーティションテーブル	BEFORE	UPDATE	STATEMENT	パーティションテーブル
2	パーティション	BEFORE	UPDATE	ROW	パーティション
3	パーティション	AFTER	UPDATE	ROW	パーティション
4	パーティション	AFTER	UPDATE	ROW	パーティション
5	パーティションテーブル	AFTER	UPDATE	STATEMENT	パーティションテーブル

表 7　単純な DELETE 文

順番	トリガー対象テーブル	TG_WHEN	TG_OP	TG_LEVEL	TG_TABLE_NAME
1	パーティションテーブル	BEFORE	DELETE	STATEMENT	パーティションテーブル
2	パーティション	BEFORE	DELETE	ROW	パーティション
3	パーティション	AFTER	DELETE	ROW	パーティション
4	パーティション	AFTER	DELETE	ROW	パーティション
5	パーティションテーブル	AFTER	DELETE	STATEMENT	パーティションテーブル

表 8　TRUNCATE 文

順番	トリガー対象テーブル	TG_WHEN	TG_OP	TG_LEVEL	TG_TABLE_NAME
1	パーティションテーブル	BEFORE	TRUNCATE	STATEMENT	パーティションテーブル
2	パーティション	BEFORE	TRUNCATE	STATEMENT	パーティション
3	パーティションテーブル	AFTER	TRUNCATE	STATEMENT	パーティションテーブル
4	パーティション	BEFORE	TRUNCATE	STATEMENT	パーティション

表 9　UPDATE 文によるタプルのパーティション間移動発生時

順番	トリガー対象テーブル	TG_WHEN	TG_OP	TG_LEVEL	TG_TABLE_NAME
1	パーティションテーブル	BEFORE	UPDATE	STATEMENT	パーティションテーブル
2	移動元パーティション	BEFORE	UPDATE	ROW	移動元パーティション
3	移動元パーティション	BEFORE	DELETE	ROW	移動元パーティション
4	移動先パーティション	BEFORE	INSERT	ROW	移動先パーティション
5	パーティションテーブル	AFTER	DELETE	ROW	移動元パーティション
6	移動元パーティション	AFTER	DELETE	ROW	移動元パーティション
7	パーティションテーブル	AFTER	INSERT	ROW	移動先パーティション
8	移動先パーティション	AFTER	INSERT	ROW	移動先パーティション
9	パーティションテーブル	AFTER	UPDATE	STATEMENT	パーティションテーブル

表 10　INSERT ON CONFLICT DO NOTHING（CONFLICT 有）

順番	トリガー対象テーブル	TG_WHEN	TG_OP	TG_LEVEL	TG_TABLE_NAME
1	パーティションテーブル	BEFORE	INSERT	STATEMENT	パーティションテーブル
2	パーティション	BEFORE	INSERT	ROW	パーティション
3	パーティションテーブル	AFTER	INSERT	STATEMENT	パーティションテーブル

3.2.9 FOREGN KEY のサポート

PostgreSQL 10 のパーティション・テーブルでは外部キーを作成できませんでした。PostgreSQL 11 ではこの制約が解消されました。

例 35 パーティション・テーブルと外部キー

```
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 動的パーティション・プルーニング

パーティション・キーがパラメーター指定された場合でもパーティション・プルーニングが実行されるようになりました。ただし PREPARE 文と EXECUTE 文による「一般的な実行計画」が利用される場合に限ります。

例 36 テスト用に実行する SQL

```
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) ;
```

例 37 PostgreSQL 10 の実行計画

```
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)
```

例 38 PostgreSQL 11 の実行計画

```
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 パーティション・プルーニングの制御

パーティション・プルーニング機能は、パラメーターenable_partition_pruning の設定で制御できます。このパラメーターを off に設定すると検索処理でパーティション・プルーニングが無効になります。

例 39 パーティション・プルーニングの制御

```
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 論理レプリケーションの拡張

PostgreSQL 11 では論理レプリケーションに以下の機能が追加されました。

3.3.1 TRUNCATE 文の伝播

論理レプリケーション環境において PUBLICATION 側で発行された TRUNCATE 文が SUBSCRIPTION 側に伝播するようになりました。これに伴い、CREATE PUBLICATION 文および ALTER PUBLICATION 文の WITH 句に TRUNCATE を伝播する指定が追加されました。

例 40 CREATE PUBLICATION 文に TRUNCATE を指定

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

PostgreSQL 10 と同様、TRUNCATE 文以外の DDL は伝播されません。

3.3.2 pg_replication_slot_advance 関数

論理レプリケーションでコンフリクトが発生した場合、PostgreSQL 10 では SUBSCRIPTION インスタンスで pg_replication_origin_advance 関数を実行して、論理レプリケーションの開始 LSN の指定を行いました。PostgreSQL 11 では同様の処理を PUBLICTION インスタンス上で実行できるようになりました。pg_replication_slot_advance 関数を実行します。関数にはレプリケーション・スロット名と LSN を指定します。

例 41 レプリケーション開始 LSN を現在の LSN に設定

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

3.4 アーキテクチャの変更

3.4.1 システム・カタログの変更

以下のシステム・カタログが変更されました。

表 11　列が追加されたシステム・カタログ

カタログ名	追加列名	データ型	説明
pg_aggregate	aggfinalmodify	char	aggfinalfn 関数が値を変更するか
	aggmfinalmodify	char	aggmfinalfn 関数が値を変更するか
pg_attribute	atthasmissing	bool	ページを更新していないデフォルト値を持つ
	attmissingval	anyarray	ページを更新していないデフォルト値
pg_class	relrewrite	oid	DDL 実行中に新規リレーションが作成される場合の OID
pg_constraint	conparentid	oid	親パーティションの制約 OID
	conincluding	smallint[]	制約以外の列番号リスト
pg_index	indnkeyatts	smallint	キー列の数
pg_partitioned_table	partdefid	oid	デフォルト・パーティションの OID
pg_proc	prokind	char	種類を示す f: function p: procedure a: aggregate function w: window function
pg_publication	pubtruncate	boolean	TRUNCATE 伝播可能
pg_stat_wal_receiver	sender_host	text	接続先ホスト名
	sender_port	integer	接続先ポート番号
information_schema.table_constraints	enforced	information_schema.yes_or_no	将来利用のため予約

表 12　列が削除されたシステムカタログ

カタログ名	削除列名	説明
pg_class	relhaspkey	主キーを持つ
pg_proc	proisagg	集約関数である
	proiswindow	Window 関数である

表 13　値が格納されるようになった information_schema スキーマのカタログ

カタログ名	列名	説明
triggers	action_order	トリガー実行順
	action_reference_new_table	NEW 代理テーブル名
	action_reference_old_table	OLD 代理テーブル名
tables	table_type	外部テーブルは FOREIGN が格納（従来は FOREIGN TABLE）

□　pg_stat_activity カタログ

backend_type 列とプロセス名が同期されるようになりました。replication launcher プロセスの backend_type 列は PostgreSQL 10 では background worker でしたが、PostgreSQL 11 では logical replication launcher と出力されます。

例 42 pg_stat_activity カタログの検索

```
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 カタログ

DEFAULT 値と NOT NULL 制約を指定する列を追加する際に、実データを更新せずに列を追加できるようになりました。pg_attribute カタログにはこの機能に対する列が追加されています。

例 43 pg_attribute カタログの検索

```
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 ロールの追加

以下のロールが追加されました。これらのロールは主に COPY 文の実行や file_fdw Contrib モジュールの実行を一般ユーザーに許可するために使用されます。

表 14　追加ロール

ロール	用途
pg_execute_server_program	サーバー上のプログラムを実行可能
pg_read_server_files	サーバー上のファイルを読み込み可能
pg_write_server_files	サーバー上のファイルに書き込み可能

例 44 pg_read_server_files ロール

```
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 の統合

PostgreSQL 11 はプロセッサ・ボトルネックとなる長時間実行 SQL 文の高速化を目指して、LLVM（https://llvm.org/）を使った JIT コンパイルをサポートします。一定以上のコストが発生すると予想された SQL 文は事前にコンパイルされてから実行されます。

□ インストール

LLVM を利用するためには、インストール時に configure コマンドのオプション--with-llvm を指定する必要があります。configure コマンド実行時には llvm-config コマンドと clang コマンドがコマンド実行パスに含まれる必要があります（または環境変数 LLVM_CONFIG と環境変数 CLANG に指定）。

□ JIT コンパイルの動作

実行総コストがパラメーターjit_above_cost（デフォルト値 100000）を超える SQL 文は LLVM による JIT コンパイル機能が動作します。このパラメーターを「-1」に指定するか、パラメーターjit を「off」に設定すると、JIT 機能は無効になります。

JIT コンパイル処理はインライン化（パラメーターjit_inline_above_cost）や、最適化（パラメーターjit_optimize_above_cost）によって動作が変更されます。

□ 実行計画

EXPLAIN 文を使って JIT コンパイル機能を使用する SQL 文の実行計画を確認すると「JIT:」から始まる情報が追加されることがわかります。

例 45 JIT コンパイル機能を使う SQL の実行計画

```
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 GIN / GiST / HASH インデックスの述語ロック

GIN インデックス、GiST インデックス、HASH インデックスに対して述語ロック（predicate locks）が利用できるようになりました。ロック範囲が小さくなるため、複数セッションによる SQL 文の同時実行性が向上します。

下記の例は HASH インデックスを使った検証結果です。PostgreSQL 10 ではロック範囲がリレーション全体（relation）になっていますが、PostgreSQL 11 ではページ（page）になっていることがわかります。

例 46 検証方法

```
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)
```

例 47 PostgreSQL 10 の結果

```
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)
```

例 48 PostgreSQL 11 の結果

```
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 LDAP 認証の強化

pg_hba.conf ファイルに記述する LDAP 認証パラメーターに ldapsearchfilter 属性が追加されました。ldapsearchattribute 属性よりも LDAP サーバーを柔軟に検索することができます。

3.4.6 拡張クエリーのタイムアウト

従来の拡張クエリーは複数の SQL 文が送信された後、SYNC メッセージが送信されるまでの時間でタイムアウトを決定していました。PostgreSQL 11 では個別の SQL 文の実行時間が考慮されるようになりました。下記の例は pgproto（https://github.com/tatsuo-ishii/pgproto）の定義ファイルです。PostgreSQL 10 では「SET statement_timeout = 4s」を設定してもタイムアウトが発生していました。

例 49 pgproto 定義ファイル

```
# 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
'E'     ""      0
'C'     'S'     "S1"

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

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

PostgreSQL 11 Timeout check

PostgreSQL 10 Timeout check

PostgreSQL 11 Timeout check

3.4.7 バックアップ・ラベルの変更

オンライン・バックアップを実行した時に作成される backup_label ファイルに、タイムライン ID が追加されるようになりました。

例 50 backup_label ファイル

```
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 000000010000000000000006A)
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                          ←追加
```

3.4.8 Windows 環境における Huge Pages の利用

Microsoft Windows環境でLock Pages In Memory設定が利用できるようになりました。パラメーターhuge_pages を try または on に設定すると、共有メモリーが連続領域を確保するようになります。内部的には Windows API CreateFileMapping に PAGE_READWRITE、SEC_LARGE_PAGES , SEC_COMMIT が指定されるようになりました。従来は PAGE_READWRITE のみでした。

3.4.9 古いチェックポイント情報の削除

PostgreSQL 10 までは過去２回のチェックポイント情報を保存していましたが、最新のチェックポイントの情報のみ保存するようになりました。

3.4.10 エラー・コードの一覧

{INSTALL_DIR}/share/errcodes.txt ファイルが追加されました。このファイルにはPostgreSQL、PL/pgSQL、PL/Tcl のエラーコード、レベル、マクロ名等が含まれます。

3.5 SQL 文の拡張

ここでは SQL 文に関係する新機能を説明しています。

3.5.1 LOCK TABLE 文の拡張

LOCK TABLE 文にビューを指定することができるようになりました。ビューをロックするとビュー定義に含まれるテーブルに対しても同じモードのロックがかかります。

例 51 ビューに対する LOCK TABLE 文の実行

```
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)
```

ビューがネストしている場合は、更に下位のテーブルまでロックされます。ただし、ビューに含まれるマテリアライズド・ビューはロックされません。

例 52 マテリアライズド・ビューに対する LOCK TABLE 文の実行

```
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 関数インデックスの STATISTICS

関数インデックスの列に対して STATISTICS 値を指定できるようになりました。

例 53 関数インデックスの STATISTICS

```
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 文／ANALYZE 文の拡張

□ 複数テーブルの指定

VACUUM 文と ANALYZE 文は複数のテーブルを同時に指定できるようになりました。

例 54 複数テーブルに対する VACUUM, ANALYZE 文の実行

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

□ 積極的な VACUUM の出力

VACUUM (VERBOSE, FREEZE) 文を実行した場合、出力に aggressively が追加されます。

例 55 VACUUM (VERBOSE, FREEZE) 文の出力変更

```
demodb=> 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.
<<以下省略>>
```

3.5.4 LIMIT 句のプッシュダウン

ソート済のサブ・クエリーに LIMIT 句が設定されていた場合に、LIMIT 句の内容がサブ・クエリーにプッシュされるようになりました。

例 56 PostgreSQL 10 の実行計画

```
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 width=10) (actual time=0.020..481.233 rows=333333 loops=3)
 Planning time: 0.041 ms
 Execution time: 1207.299 ms
(10 rows)
```

例 57 PostgreSQL 11 の実行計画

```
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 文の拡張

CREATE INDEX 文には以下の拡張が追加されました。

□ INCLUDE 句

インデックスに列を追加する INCLUDE 句が指定できます。これは一意制約等に対して制約に含まない列を追加したい場合等に有効です。

下記の例は一意インデックスを c1 列と c2 列で作成していますが、インデックスとしては c3 列を含んでいます。

例 58 CREATE INDEX 文の INCLUDE 句

```
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"
```

CREATE INDEX 文と同様に CREATE TABLE 文の制約指定部分にも INCLUDE 句は使用できます。

例 59 CREATE TABLE 文の INCLUDE 句

```
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

CREATE INDEX 文の WITH 句に recheck_on_update オプションを指定できるようになりました。デフォルト値は'on'です。このパラメーターは関数インデックスに対して HOT による更新を使うかを指定します。

例 60 CREATE INDEX 文のオプション

```
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 文の拡張

TOAST 化を行う敷居値を示すストレージ・パラメーターtoast_tuple_target を指定できるようになりました。デフォルト値は従来と変わりません。

例 61 CREATE TABLE 文のオプション

```
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 関数の拡張

WINDOW 関数に GROUPS 句とウィンドウフレームに EXCLUDE 句を指定できるようになりました。また RANGE 句には float4 型、float8 型、numeric 型が利用できるようになりました。

構文

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

frame_exclusion 句に指定できるのは以下の構文です。

構文（frame_exclusion 部分）

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

3.5.8 EXPLAIN 文の拡張

パラレル・クエリー実行時にワーカー毎にソートに関する情報が表示されるようになりました。

例 62 EXPLAIN 文で実行されるパラレル・クエリー

```
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 time=0.033..116.654 rows=666666 loops=3)
<< 以下省略 >>
```

3.5.9 関数

以下の関数が追加／拡張されました。

□ ハッシュ関数の追加

SHA-224 / SHA-256 / SHA-384 / SHA-512 を利用するハッシュ関数が提供されました。使い方はどれも同じです。

表 15 ハッシュ関数

関数名	説明	備考
sha224(bytea)	SHA-224 ハッシュ	
sha256(bytea)	SHA-256 ハッシュ	
sha384(bytea)	SHA-384 ハッシュ	
sha512(bytea)	SHA-512 ハッシュ	

例 63 ハッシュ関数 SHA512

```
postgres=> SELECT sha512('ABC') ;
                                                  sha512
----------------------------------------------------------------------------------------------------
¥x397118fdac8d83ad98813c50759c85b8c47565d8268bf10da483153b747a74743a58a90e85aa9f705ce6984ffc128db567489817e4092d050d8a1cc596ddc119
(1 row)
```

□ json(b)_to_tsvector 関数

JSON 型（または JSONB 型）から tsvector 型への変換を行う json(b)_to_tsvector 関数が利用できます。

例 64 json_to_tsvector 関数

```
postgres=> SELECT json_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 関数

Web Search 形式の文字列から tsquery 型への変換を行う関数 websearch_to_tsquery 関数が利用できます。

例 65 websearch_to_tsquery 関数

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

3.5.10 演算子

以下の演算子が追加されました。

□ 文字列前方一致検索

文字列の前方一致を検索する演算子「^@」が追加されました。WHERE 句の「LIKE'文字列%'」と同じ用途で使うことができます。

構文

```
検索対象 ^@ 検索文字列
```

例 66 ^@演算子

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

子の演算子は LIKE 句と異なり、B-Tree インデックスは利用されません。未検証ですが、SP-GiST インデックスは利用できます。

3.5.11 その他

□ JSONB 型からのキャスト

JSONB 型から bool 型、数値型へのキャストが可能になりました。

例 67 JSONB 型のキャスト

```
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)
```

□ SP-GiST インデックスの拡張

polygon 型に作成できる poly_ops 演算子クラスが提供されました。また圧縮を行うユーザー定義メソッドが定義できるようになりました。

3.6 PL/pgSQL の拡張

ここでは PL/pgSQL 言語の拡張と、PL/pgSQL を使用する新しいオブジェクトについて説明しています。

3.6.1 PROCEDURE オブジェクト

新しいスキーマ・オブジェクトとして PROCEDURE が追加されました。PROCEDURE は戻り値の無い FUNCTION とほぼ同じオブジェクトです。PROCEDURE は CREATE PROCEDURE 文で作成します。CREATE FUNCTION 文とは異なり、RETURNS 句、ROWS 句、PARALLEL 句、CALLED ON 句などがありません。

例 68 PROCEDURE の作成

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

作成したプロシージャの情報は FUNCTION と同様、pg_proc カタログから参照することができます。

PROCEDURE の実行には SELECT 文ではなく CALL 文を使用します。パラメーターの有無にかかわらず、括弧（()）が必要です。PROCEDURE のパラメーターには INOUT を指定することができます。

例 69 PROCEDURE の実行（１）

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

CALL 文にはパラメーター名を指定することもできます。

例 70 PROCEDURE の実行（2）

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

psql コマンドから、作成した PROCEDURE の一覧を表示するには FUNCTION と同様に¥df コマンドを使います。PROCEDURE は Type 列が proc として表示されます。同時に FUNCTION の表示は func に変更されました。PROCEDURE は pg_proc カタログの prorettype 列が 0 になります。

例 71 PROCEDURE の一覧

```
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)
```

psql コマンドから PROCEDURE の定義参照や変更を行う場合も FUNCTION と同じ方法で実行できます。

例 72 PROCEDURE の定義表示

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

□ トランザクション制御

プロシージャ内ではトランザクションの制御を行うことができます。PL/pgSQL では COMMIT 文と ROLLBACK 文を記述できます。カーソル・ループ内でもトランザクションの制御を行うことができるようになりました。

例 73 PROCEDURE 内のトランザクション

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

PL/pgSQL 以外の言語でもトランザクション制御用の構文が提供されています。

PL/Python

plpy.commit() / plpy.rollback()

PL/Perl

spi_commit() / spi_rollback()

PL/Tcl

commit / rollback

3.6.2 変数定義の拡張

PL/pgSQL には以下の拡張が追加されました。

□ CONSTANT 変数

変数に CONSTANT 指定を追加することで定数の宣言ができるようになりました。初期値を指定しない場合もエラーにはならず、NULL 値になります。値の変更操作はエラーになります。

構文

```
DECLARE 変数名 CONSTANT データ型 [ := 初期値 ] ;
```

例 74 CONSTANT オプションと値の変更

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

□ NOT NULL 変数

変数に NOT NULL 制約を指定することができるようになりました。宣言時には初期値が必要です。初期値を指定しない場合エラーになります。また NULL 値の代入でもエラーになります。

構文

```
DECLARE 変数名 データ型 NOT NULL := 初期値 ;
```

例 75 NOT NULL オプションと初期値

```
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 文

PL/pgSQL ブロック内で SET TRANSACTION 文を実行できます。

例 76 SET TRANSACTION 文

```
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 パラメーターの変更

PostgreSQL 11 では以下のパラメーターが変更されました。

3.7.1 追加されたパラメーター

以下のパラメーターが追加されました。

表 16　追加されたパラメーター

パラメーター	説明（context）	デフォルト値
data_directory_mode	データベース・クラスタの保護モード（internal）	0700
enable_parallel_append	Parallel Append 機能を有効（user）	on
enable_parallel_hash	Parallel Hash 機能を有効（user）	on
enable_partition_pruning	パーティション・プルーニングを使用する（user）	on
enable_partitionwise_aggregate	Partition-Wise Join を有効（user）	off
enable_partitionwise_join	Partition-Wise Aggregate を有効（user）	off
jit	JIT コンパイル機能を利用する（user）	on
jit_above_cost	JIT コンパイル機能を有効にするコスト（user）	100000
jit_debugging_support	JIT コンパイル時にデバッガを有効にする（superuser-backend）	off
jit_dump_bitcode	LLVM ビットコードを追加する（superuser）	off
jit_expressions	JIT expression を有効にする（user）	on
jit_inline_above_cost	JIT インライン化を行うかを決めるコスト（user）	500000
jit_optimize_above_cost	JIT 最適化を行うかを決めるコスト（user）	500000
jit_profiling_support	perf プロファイラーを有効にする（superuser-backend）	off
jit_provider	JIT プロバイダ名（postmaster）	llvmjit
jit_tuple_deforming	タプル修正に JIT 機能を使う（user）	on
max_parallel_maintenance_workers	メンテナンス処理で使われるパラレル・ワーカー最大数（user）	2

パラメーター	説明（context）	デフォルト値
parallel_leader_participation	リーダー・プロセスの動作を変更（user）	on
ssl_passphrase_command	SSL 接続の passphrases を取得するコマンド（sighup）	''
ssl_passphrase_command_supports_reload	リロード時に ssl_passphrase_command を使うか（sighup）	off
vacuum_cleanup_index_scale_factor	インデックスのクリーンアップを行う INSERT 数の割合（user）	0.1

□ parallel_leader_participation パラメーター

パラレル・クエリーが実行される場合、全体のとりまとめを行うリーダー・プロセスと複数のワーカー・プロセスが協調して処理を行います。PostgreSQL 10 ではリーダー・プロセスも Gather と Gather Merge ノードでプランを実行していました。このパラメーターを off に指定すると、リーダー・プロセスはワーカー・プロセスと同様の処理を行わなくなります。

下記の例は 1,000 万レコードのテーブル data1 に対して検索を行っています。どちらの例でもワーカー・プロセスは 2 個起動しています。最初の例では Parallel Seq Scan の部分がレコード数 3,333,333 でループ 3 回、パラメーターparallel_leader_participation を off に変更した次の例ではレコード数 5,000,000 でループ 2 回になっていることがわかります。

例 77　パラメーター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)
```

例 78　パラメーター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 変更されたパラメーター

以下のパラメーターは設定範囲や選択肢が変更されました。

表 17　変更されたパラメーター

パラメーター	変更内容
log_parser_stats	設定値を on に変更した場合のログにメモリー情報が追加されます
log_statement_stats	
log_planner_stats	
log_executor_stats	
wal_segment_size	pg_settings カタログの UNIT が 8kB から B に変更されました

□ log_parser_stats パラメーター

設定値を on に指定した場合に、出力される情報が増えました。追加出力される情報はオペレーティング・システムによって異なります。下記の例は Linux の場合です。

例 79　パラメーター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 デフォルト値が変更されたパラメーター

以下のパラメーターはデフォルト値が変更されました。

表 18　デフォルト値が変更されたパラメーター

パラメーター	PostgreSQL 10	PostgreSQL 11	備考
server_version	10.4	11beta1	
server_version_num	100004	110000	

3.7.4 廃止されたパラメーター

以下のパラメーターは廃止されました。

表 19　廃止されたパラメーター

パラメーター	代替値	備考
replacement_sort_tuples	なし	

3.7.5 認証パラメーターの変更

□ インスタンス接続文字列

Libpq ライブラリのインスタンス接続文字列に以下のパラメーターが追加されました。

表 20　追加されたパラメーター

パラメーター	デフォルト値	説明
scram_channel_binding	tls-unique	SCRAM 認証のチャネル・バインディング種別
replication	false	レプリケーション・プロトコルを使うかを指定 true: 物理レプリケーション database: 論理レプリケーション

ssl_compression パラメーターはデフォルト値が off に変更されました。

□ LDAP 認証パラメーター

LDAP 認証に ldapschema パラメーターが追加されました。パラメーターに ldaps を指定すると LDAP over SSL を使用します。

3.8 ユーティリティの変更

ユーティリティ・コマンドの主な機能強化点を説明します。

3.8.1 psql コマンド

psql コマンドには以下の機能が追加されました。

□ ¥gdesc コマンド

直前に実行されたクエリーの列名とデータ型を表示する¥gdesc コマンドが追加されました。

例 80 ¥gdesc コマンド

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

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

□ ページャーの指定

psql コマンドが使用するページャーを示す環境変数は PAGER よりも PSQL_PAGER が優先されます。環境変数 PSQL_PAGER が設定されていない場合は引き続き環境変数 PAGER が利用できます。

□ クエリーのステータス

SQL 文の実行ステータスに関する変数が利用できるようになりました。¥if コマンドと組み合わせることで SQL 文の実行結果を SQL スクリプト内でハンドリングできるようになりました。LAST_ERROR_MESSAGE 変数と LAST_ERROR_SQLSTATE 変数は、エラー発生後に、別の SQL 文が成功しても内容を維持します。

表 21　SQL 文の実行結果を示す変数

変数名	説明
ERROR	直前に実行した SQL がエラーの場合は true
LAST_ERROR_MESSAGE	最後に発生したエラーのメッセージ
LAST_ERROR_SQLSTATE	最後に発生したエラーのコード
ROW_COUNT	直前に実行した SQL 文が処理したタプル数
SQLSTATE	直前に実行した SQL 文の実行ステータス・コード

例 81 SQL 実行結果変数

```
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 / quit コマンド

psql コマンドの終了に exit / quit コマンドが使用できるようになりました。これまでは ¥q を使っていました。

□　パラメータの定義チェック

変数が定義されるかをチェックする構文「:{?変数名}」が利用できるようになりました。

例 82 変数定義の確認

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

3.8.2 ECPG コマンド

ECPG には以下の拡張が行われました。

□ Oracle mode

文字列型データ取得時の動作を変更するオプション「-C ORACLE」オプションが追加されました。ここではデフォルト値と-C ORACLE を指定した場合の比較を行います。まず文字列出力用の領域と、指示子を宣言します。

例 83 文字列出力宣言

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

VARCHAR(10) 型の列のデータを変数 char shortstr[5] に出力します。

例 84 列データの出力

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

元データの長さによって、変数 shortstr に出力される内容を確認します。

表 22　モードによる違い（デフォルト設定）

元データ	shortstr	shortstr_ind	備考
''	''	0	shortstr[0] = NULL
'AB'	'AB'	0	shortstr[4] = 変更なし
'ABCD'	'ABCD'	0	shortstr[4] = NULL
'ABCDE'	'ABCDE'	0	shortstr[4] = 'E'
'ABCDEF'	'ABCDE'	6	shortstr[4] = 'E'
'ABCDEFGHIJ'	'ABCDE'	10	shortstr[4] = 'E'

表 23　モードによる違い（-C ORACLE）

元データ	shortstr	shortstr_ind	備考
''	'　　'	-1	スペース埋め、shortstr[4] = NULL
'AB'	'AB　'	0	スペース埋め、shortstr[4] = NULL
'ABCD'	'ABCD'	0	shortstr[4] = NULL
'ABCDE'	'ABCD'	5	shortstr[4] = NULL
'ABCDEF'	'ABCD'	6	shortstr[4] = NULL
'ABCDEFGHIJ'	'ABCD'	10	shortstr[4] = NULL

上記例のように、-C ORACLE オプションを指定すると領域が余っている場合にはスペースを付加され、最終部分に NULL が付加されます。

□ DO CONTINUE 文

WHENEVER 文に DO CONTINUE 句が指定できるようになりました。これは WHENEVER に指定された条件に合致した場合にループの最初に戻る動作になります。

例 85 DO CONTINUE 句の指定

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

3.8.3 initdb コマンド

initdb コマンドには以下の拡張が行われました。

□ --wal-segsize オプション

WAL ファイルのサイズを指定する「--wal-segsize」オプションが追加されました。このオプションは 2 の乗数で、1 から 1024 の値をメガバイト単位で指定できます。デフォルト値は従来と同様に 16MB です。

これに伴い、インストール時に実行する configure コマンドの「--with-wal-segsize」オプションは廃止されました。

例 86 initdb --wal-segsize

```
$ 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".
<<以下省略>>
```

□ --allow-group-access オプション

データベース・クラスタのアクセス・モードにグループ・アクセスを許可する--allow-group-access オプション（または-g オプション）が追加されました。このオプションを指定すると、データベース・クラスタのディレクトリの保護モードに、グループの読み込み／実行権限が追加され、データベース・クラスタのパラメーターdata_directory_mode が 0750 に変更されます。

例 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.
<<途中省略>>
    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 コマンド

pg_dump コマンドおよび pg_dumpall コマンドに以下のオプションが追加されました。

□ --load-via-partition-root オプション

pg_dump コマンドおよび pg_dumpall コマンドに--load-via-partition-root オプションが追加されました。このオプションを指定すると、データのロード時に、個別のパーティションではなく、パーティション・テーブルのルート・テーブルを経由してデータをロードします。

□ --encoding オプション

pg_dumpall コマンドに--encoding オプション（または-E オプション）が追加されました。このオプションは、pg_dump コマンドと同様に出力データの文字エンコーディングを指定します。

例 88 pg_dumpall コマンドの文字エンコード指定

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

□ --no-comments オプション

pg_dump コマンドに--no-comments オプションが追加されました。このオプションを指定すると、COMMENT がロードされません。

3.8.5 pg_receivewal コマンド

pg_receivewal コマンドには以下のオプションが追加されました。

□ --endpos オプション

コマンドを終了する LSN を指定する--endpos オプション（または-E オプション）が追加されました。PostgreSQL 10 で pg_recvlogical コマンドに追加された--endpos オプションと同じ機能です。

□ --no-sync オプション

--no-sync オプションが追加されました。このオプションを指定すると、データ書き込み時に sync システムコールを実行しません。

3.8.6 pg_ctl コマンド

pg_ctl コマンドから SIGKILL シグナルを送信できるようになりました。

例 89 pg_ctl コマンド

```
$ 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]
<<途中省略>>
  pg_ctl promote  [-D DATADIR] [-W] [-t SECS] [-s]
  pg_ctl kill     SIGNALNAME PID

Common options:
<<途中省略>>
Allowed signal names for kill:
  ABRT HUP INT KILL QUIT TERM USR1 USR2
```

3.8.7 pg_basebackup コマンド

pg_basebackup コマンドには以下の拡張が行われました。

□ --no-verify-checksum オプション

pg_basebackup コマンドはバックアップしたブロックのチェックサムを確認するようになりました。--no-verify-checksum オプションはチェックサムの確認処理をスキップします。チェックサムにエラーが発生した場合、pg_basebackup コマンドは 0 以外の値で終了します。

□ --create-slot オプション

pg_basebackup コマンドには、レプリケーション・スロットを作成する--create-slot オプション（または-C オプション）が追加されました。このオプションは--slot オプションと同時に使用します。作成したレプリケーション・スロットは pg_basebackup コマンドの終了後も維持されます。

すでに同じ名前のレプリケーション・スロットが存在する場合、pg_basebackup コマンドはエラー・メッセージを表示して終了します。

例 90 pg_basebackup コマンド

```
$ 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
$
```

□ バッチ・モードの動作変更

pg_basebackup コマンドに--progress オプションを指定し、バッチ・モード（シェルスクリプトからファイルにリダイレクト）から実行する場合には、改行コード「¥r」の代わりに改行コード「¥n」が出力されるようになりました。

例 91 バッチ・モードの動作

```
$ 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
```

□ pg_internal.init ファイルの除外

pg_basebackup コマンドのバックアップ対象から pg_internal.init ファイルが除外されます。

□ UNLOGGED テーブルの除外

転送データから UNLOGGED テーブル、TEMPORARY テーブルが除外されます。

3.8.8 pg_resetwal / pg_controldata コマンド

pg_resetwal コマンドには WAL ファイルのサイズを指定する--wal-segsize オプションが追加されました。また、pg_resetwal コマンドと pg_controldata コマンドには、既存の短いオプションに対応する長い名前のオプションが追加されました。

表 24　pg_resetwal コマンドの追加オプション

短いオプション	追加されたオプション	備考
-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	
なし	--wal-segsize	

表 25　pg_controldata コマンドのオプション

短いオプション	追加されたオプション	備考
-D	--pgdata	

3.8.9 configure コマンド

configure コマンドには以下の変更がありました。

□ --with-wal-segsize オプション

--with-wal-segsize オプションは廃止されました。initdb コマンドの--wal-segsize オプションで設定します。

□ --with-llvm オプション

LLVM を組み込むオプション--with-llvm が追加されました。このオプションを指定する場合にはコマンド検索パスに llvm-config コマンドと、clang コマンドが含まれる必要があります。または環境変数 LLVM_CONFIG と環境変数 CLANG の指定が必要です。

3.8.10 pg_verify_checksums コマンド

データベース外部からチェックサムの整合性をチェックするコマンドとして pg_verify_checksums が追加されました。このコマンドはインスタンス起動状態では実行できません。下記は特定のファイルのみチェックサムを確認しており、一部のブロックがチェックサムと合わないことを示しています。

例 92 pg_verify_checksums コマンドの使い方

```
$ 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>.
```

例 93 pg_verify_checksums コマンドの実行

```
$ 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 モジュール

Contrib モジュールに関する新機能を説明しています。

3.9.1 adminpack

モジュールに含まれる関数は、従来は SUPERUSER 権限が必要でしたが、GRANT 文の実行により一般ユーザーにも実行が許可できるようになりました。

3.9.2 amcheck

新しい Contrib モジュール amcheck が追加されました。B-Tree インデックスの整合性をチェックすることができます。チェックできるインデックスの種類は B-Tree インデックスのみです。下記の例では整合性が破壊されたインデックス idx1_data1 のチェックを行っています。また HASH インデックスのチェックを行いエラーが発生しています。

例 94 amcheck モジュール

```
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.
```

表 26　提供される関数一覧

関数名	説明
bt_index_check	B-Tree インデックスの整合性チェック
bt_index_parent_check	親子関係も含む B-Tree インデックスの整合性チェック

3.9.3 btree_gin

bool 型、bpchar 型、uuid 型の列に対して B-Tree GIN インデックスが作成できるようになりました。

例 95 B-Tree Gin インデックスの作成

```
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

インデックスの演算子クラス citext_pattern_ops が追加されました。text 型と同様の演算子~<~, ~<=~, ~>~, ~>=~を citext 型のまま利用できます。従来は text 型に変換されて実行されていました。

例 96 PostgreSQL 10 の実行計画

```
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)
```

例 97 PostgreSQL 11 の実行計画

```
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

GiST インデックスで Index Only Scan が実行可能になりました。

3.9.6 jsonb_plpython

新しい Contrib モジュール jsonb_plpython が追加されました。インストールするためには configure コマンドに--with-python オプションの指定が必要です。CREATE EXTENTION 文に指定するモジュール名は「jsonb_plpythonu」、「jsonb_plpython2u」、「jsonb_plpython3u」のいずれかです。CREATE FUNCTION 文の TRANSFORM 句に jsonb を指定できます。

例 98 jsonb_plpython モジュール

```
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

新しい Contrib モジュール jsonb_plperl が追加されました。インストールするためには configure コマンドに--with-perl オプションを指定してインストールが必要です。CREATE FUNCTION 文の TRANSFORM 句に jsonb を指定できます。

例 99 jsonb_plperl モジュール

```
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

bt_metap 関数の出力に、last_cleanup_num_tuples 列が追加されました。

例 100 bt_metap 関数

```
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    ←追加
```

3.9.9 pg_prewarm

インスタンス起動中に共有メモリー上に存在していたブロックを、インスタンス起動時に自動的に共有メモリーにロードする機能が追加されました。パラメーターshared_preload_libraries に pg_prewarm を指定してインスタンスを起動すると、バックグラウンド・ワーカー・プロセス「postgres: autoprewarm master」が自動的に起動します。autoprewarm master プロセスは、インスタンス起動時に以前共有メモリー上に保存されていたブロックを共有メモリーにロードします。その後、一定間隔で共有メモリー上のブロック情報をファイルに保存します。パラメーターmax_worker_processes が 0 になっている場合、バックグラウンド・ワーカー・プロセスは起動しません。

□ ファイル

autoprewarm プロセスが共有メモリー上にロードされたブロック情報を保存するファイルは{PGDATA}/autoprewarm.blocks です。ファイルはテキスト形式で、データベース、表スペース、filenode、ブロック等の情報を保存します。ファイルの更新中は{PGDATA}/autoprewarm.blocks.tmp ファイルに書き込みを行い、ファイル名が変更されます。

□ パラメーター pg_prewarm.autoprewarm

このパラメーターにデフォルト値の'on'を指定すると、自動プリワーム機能を有効にします。

□ パラメーター pg_prewarm.autoprewarm_interval

共有メモリー上のブロック情報を定期的に保存する最小間隔を秒単位で指定します。デフォルト値は 300 秒（5 分）です。このパラメーターを 0 に指定すると、定期的な共有メモリーの保存処理は行われなくなります。インスタンス停止時にのみこの処理が行われるようになります。

3.9.10 pg_trgm

関数 strict_word_similarity が追加されました。この関数は word_similarity 関数と似ていますが、エクステント境界をワード境界に一致させます。

例 101 strict_word_similarity 関数

```
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

postgres_fdw モジュールには以下の拡張機能が加えられました。

□ リモート・パーティションの更新

パーティションが FOREIGN TABLE の場合でも、パーティション・テーブル経由でタプルの更新が行えるようになりました。COPY 文によるタプルの挿入も実行できます。

例 102 パーティションと 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 ;
```

例 103 PostgreSQL 10 の動作

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

例 104 PostgreSQL 11 の動作

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

□ スーパーユーザーのチェック方法変更

リモート・インスタンスへの接続チェックがセッション・ユーザーではなく、USER MAPPING ユーザーにより行われるようになりました。

□ UPDATE / DELETE 文による結合時の動作

同一 FOREIGN SERVER を使う FOREIGN TABLE 同士を結合する DELETE 文や UPDATE 文がリモート・インスタンスにプッシュダウンされるようになりました。

例 105 PostgreSQL 10 の実行計画

```
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
                           Remote SQL: SELECT c1, c2 FROM public.fdata1 WHERE
(((c1 % 10) = 2))
(17 rows)
```

例 106 PostgreSQL 11 の実行計画

```
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)
```

Hewlett Packard
Enterprise

参考にした URL

本資料の作成には、以下の URL を参考にしました。

- 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 さん）
  http://paquier.xyz/
- Qiita（ぬこ@横浜さん）
  http://qiita.com/nuko_yokohama
- PostgreSQL Deep Dive
  http://pgsqldeepdive.blogspot.jp/ （Satoshi Nagayasu さん）
- pgsql-hackers Mailing list
  https://www.postgresql.org/list/pgsql-hackers/
- PostgreSQL 11 Beta 1 のアナウンス
  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/

Hewlett Packard
Enterprise

変更履歴

変更履歴

版	日付	作成者	説明
0.1	2018/04/19	篠田典良	内部レビュー版作成 レビュー担当（敬称略）: 高橋智雄（日本ヒューレット・パッカード） 北山貴広（日本ヒューレット・パッカード）
0.2	2018/05/24	篠田典良	PostgreSQL 11 Beta 1 公開版に合わせて修正 レビュー担当（敬称略）: 永安悟史（アップタイム・テクノロジーズ合同会社）
1.0	2018/05/25	篠田典良	公開版作成

以上

Hewlett Packard
Enterprise