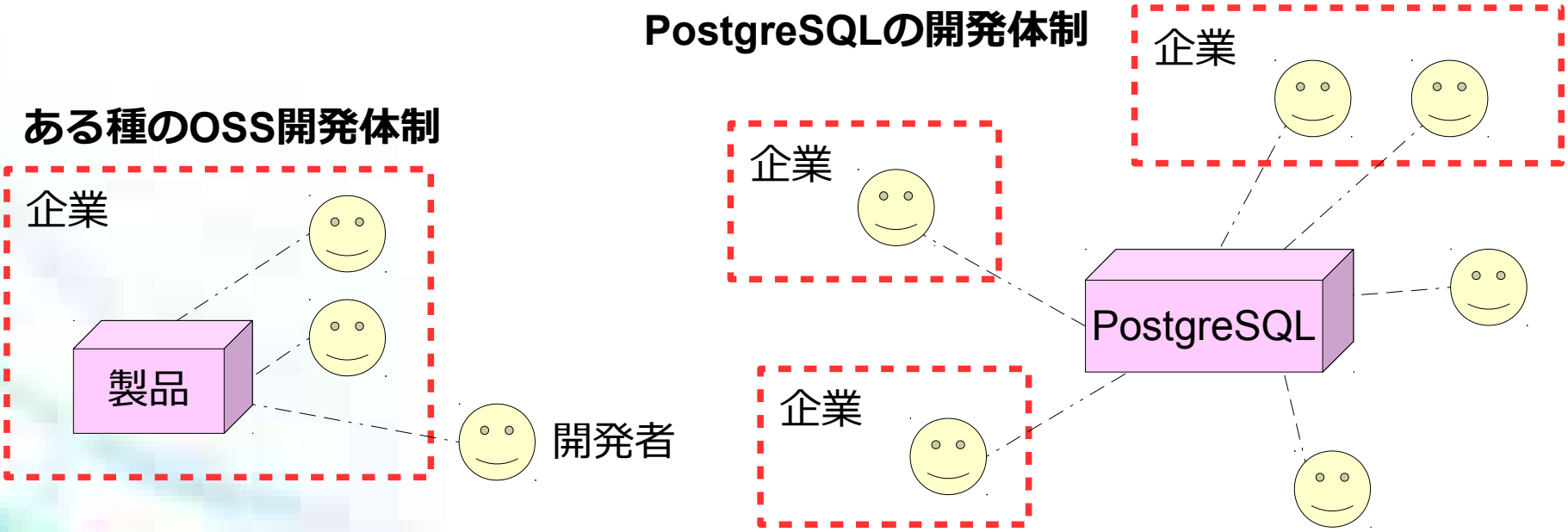
PostgreSQL 11 技術解説

2018-08-24
PostgreSQL 11 技術解説セミナー

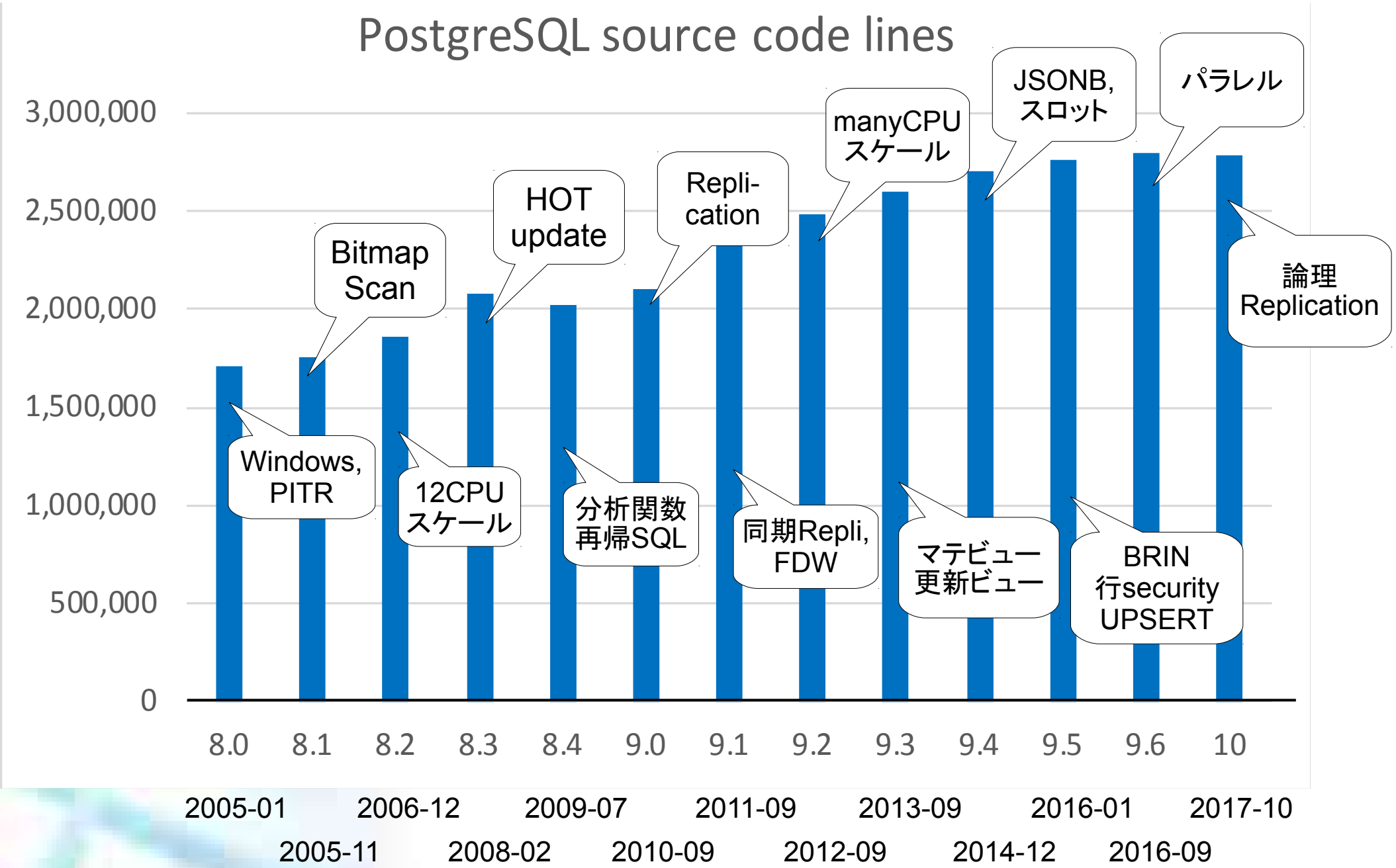
SRA OSS, Inc. 日本支社　高塚 遥

PostgreSQLとは

- 多機能、高性能、かつオープンソースの
  リレーショナルデータベース管理システム
- INGRES('70),POSTGRES('80)由来の歴史
- BSDスタイルのライセンス
- 特定オーナー企業が無い



これまでのリリース



現在の PostgreSQL（+周辺ツール/拡張）の実力

SRA OSS,INC.

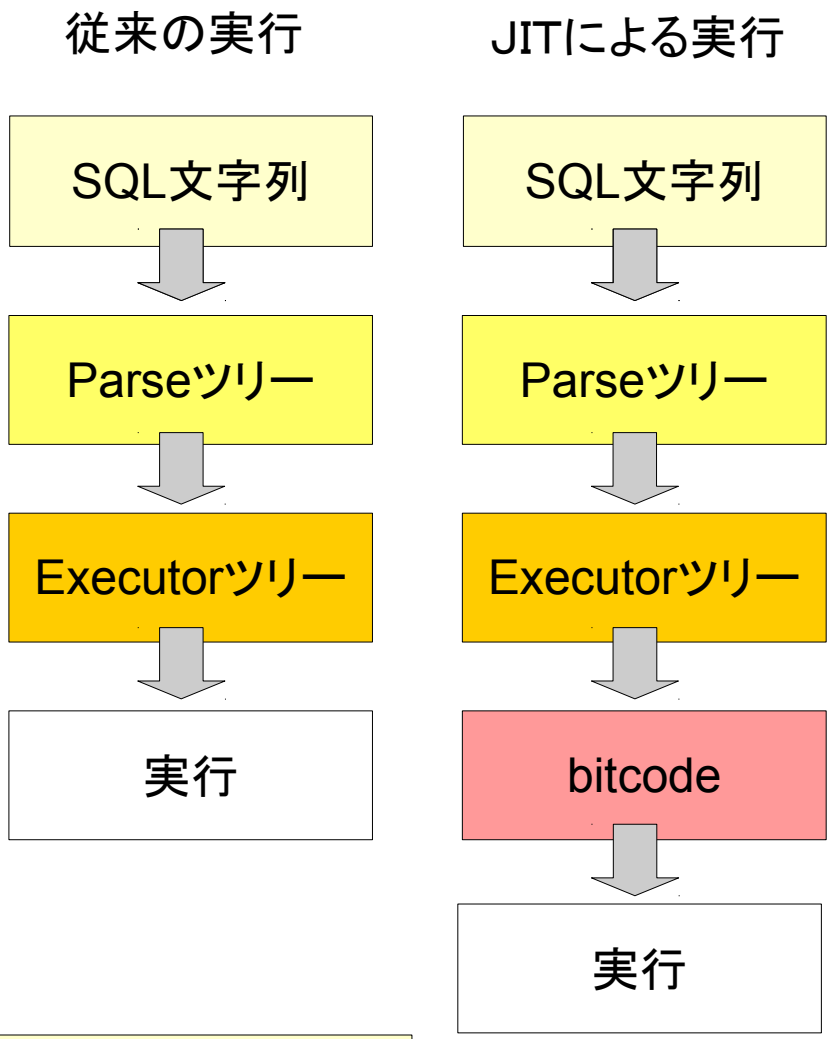
SQL機能的には： • ANSI SQL:2011コア 概ね準拠 • 各種の組み込み言語 • 地理情報システム • JSON型 • 他DB連携（FDW）	クラスタ構成： • インスタンス単位レプリケーション • テーブル単位論理レプリケーション • HAクラスタ、BDRクラスタ、 MPPクラスタ(shared nothing) • RAC型(shared disk)は不可
性能的には： • OLTPの多CPUスケール： 参照→96コアでもスケール 更新→96コアでもスケール • OLAPには： パラレルクエリ対応拡大	運用支援など： • ログ/状態の集積分析ツール pg_statsinfo / pg_badger • クライアントツール PgAdmin4 / SI Object Browser • AWS / Azure / Google Cloudで対応 AWS Aurora で対応

PostgreSQL 11 での拡張ポイント

- JITコンパイル
- 途中でCOMMIT/ROLLBACKできるストアドプロシージャ
- パーティショニング機能の拡充
- パラレル処理の拡充
- SCRAMチャンネルバインド
- その他

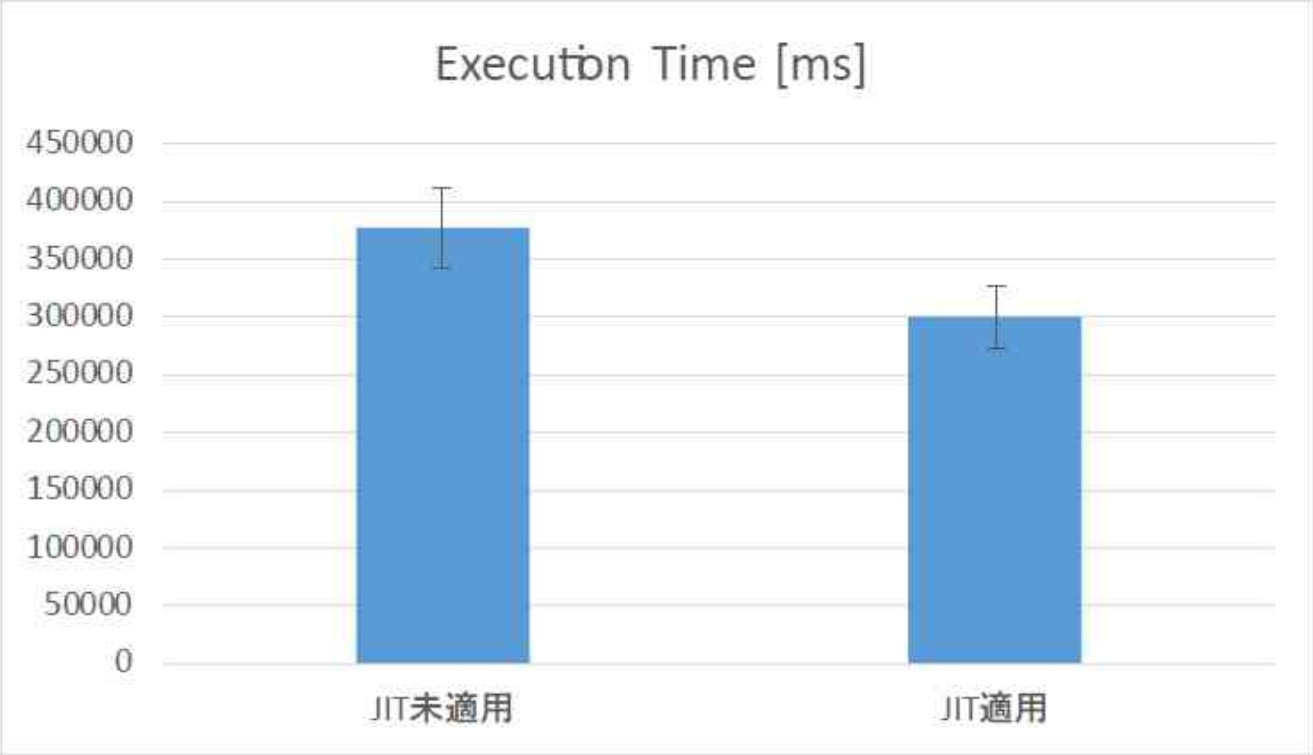
JITコンパイル（1）

- SQL実行にあたって Just In Time コンパイルを行い、ネイティブ実行を実現
- LLVMコンパイラ基盤を使用
- 繰り返し処理回数が多い場合に
- プランナコストでJIT適用を判断
- 行データ取り出し、SELECTリストの式、WHERE句の式に適用




```
SELECT id, c1, c2, c3, (c4 * random() * 10)::int
  FROM t1 WHERE typ = 101;
```

JITコンパイル（2）



簡易な性能テスト例：

11の関数・演算子と
3箇所のキャスト
×
1億件のループ
↓
このくらいから
効果がでる

```
CREATE FUNCTION f100000000() RETURNS SETOF bigint
  ROWS 100000000 LANGUAGE sql AS $$
  SELECT g FROM generate_series(1::bigint, 100000000::bigint) AS g;
$$;

SELECT g, 'X is "' || random() * pi() *
    substr((g * ln(g::float8 + g / 2))::text, 1, 5)::float8 || '"'
  FROM f100000000() AS g;
```

ストアドプロシージャ（1）

- 途中でCOMMIT/ROLLBACKできるストアドプロシージャ

- Oracle Database の PL/SQL から移植で役立つ

```
 :
FOR i IN 1..1000 LOOP
  FOR j IN 1..100 LOOP
    INSERT INTO t_sample
      SELECT * FROM f1(i * 100 + j);
  END LOOP;
  COMMIT;
END LOOP;
 :
```

100件ごとに COMMIT しつつデータ生成する。

PL/pgSQL の ユーザ定義関数では そのままには実現 できない。

- 新たなデータベースオブジェクト「PROCEDURE」
  - CALLで呼び出す
  - 中で COMMIT と ROLLBACKが可能
    - 暗黙に次トランザクションが開始される – PL/SQL と似た振る舞い

ストアドプロシージャ（2）

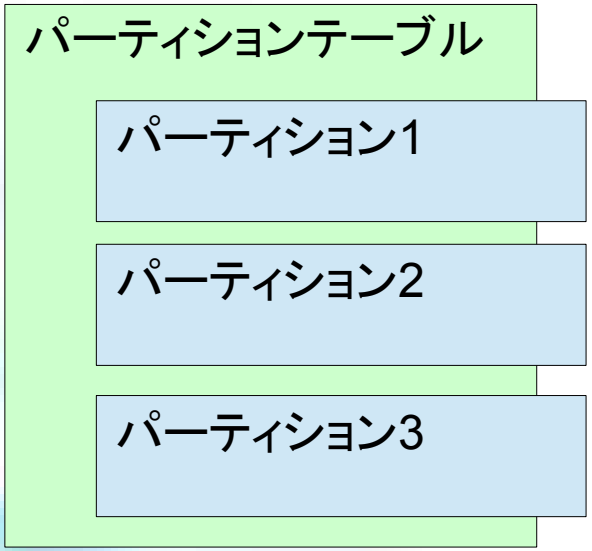
- プロシージャで COMMIT / ROLLBACK が利用可能
- いろいろ制限がある
  - 入れ子で CALL しても、サブトランザクションにはならない
  - 明示的トランザクション内では実行できない
  - 関数内からの CALLではトランザクション制御不可
- 他のPL言語でも対応

```
db1=# CREATE PROCEDURE p_tx1()
         LANGUAGE plpgsql AS $$
 BEGIN
    FOR i IN 0..9 LOOP
        INSERT INTO test1 (a) VALUES (i);
        IF i % 2 = 0 THEN COMMIT;
        ELSE ROLLBACK;
        END IF;
    END LOOP;
 END;
$$;
db1=# CALL p_tx1();
db1=# SELECT * FROM test1;
a
---
0
2
4
6
8
(5 rows)
```

さしあたりは
シンプルな用途に
使いたい

パーティショニング機能の拡充（1）

- パーティションテーブル全体にインデックスを作成できる
- パーティションテーブル全体に外部キーを設定できる
- パーティション間の行更新によるデータ移動
- ハッシュパーティショニング
- 問い合わせでのパーティション除外処理の改善
- 問い合わせでのパーティション指向の結合／集約

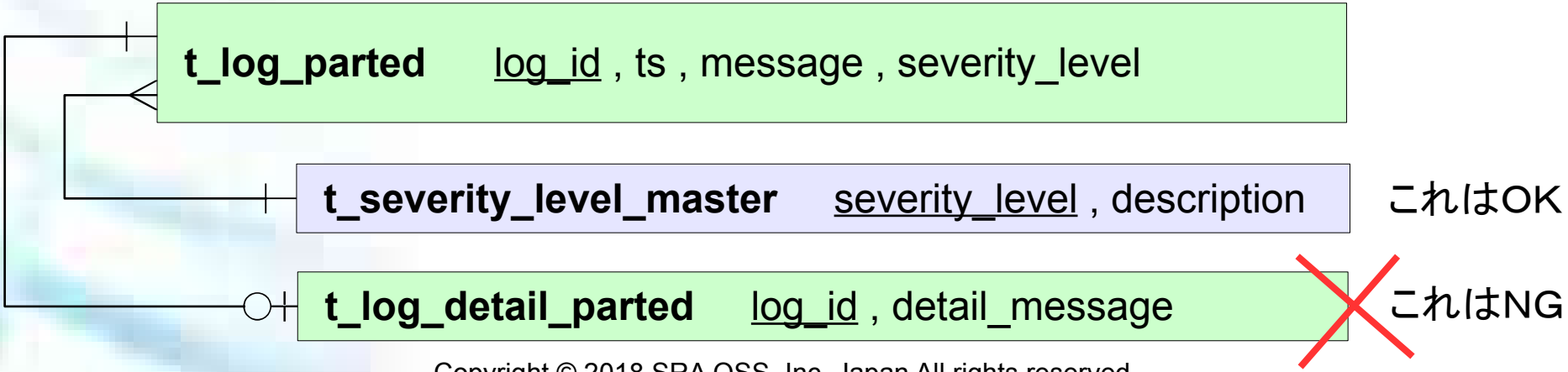


バージョン10 までで欠けていた
機能が補完された

パーティショニング機能の拡充(2)

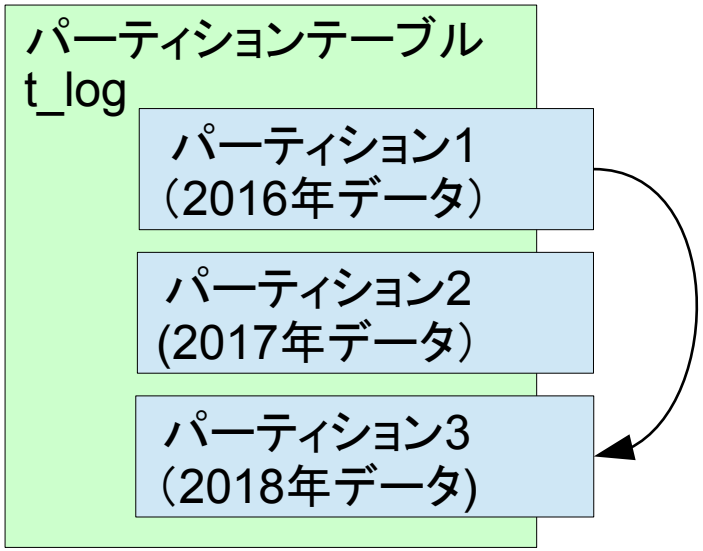
- パーティションテーブル全体にインデックスを作成できる
  - グローバルインデックスが作られるわけではない。
    各パーティションに各々インデックスが作られるだけ。
  - パーティション分割条件の列を含む必要がある。
  - パーティションテーブル全体に主キー制約が作れる

- パーティションテーブル全体に外部キーを設定できる
  - パーティションテーブルを外部キーの被参照テーブルにするのは不可



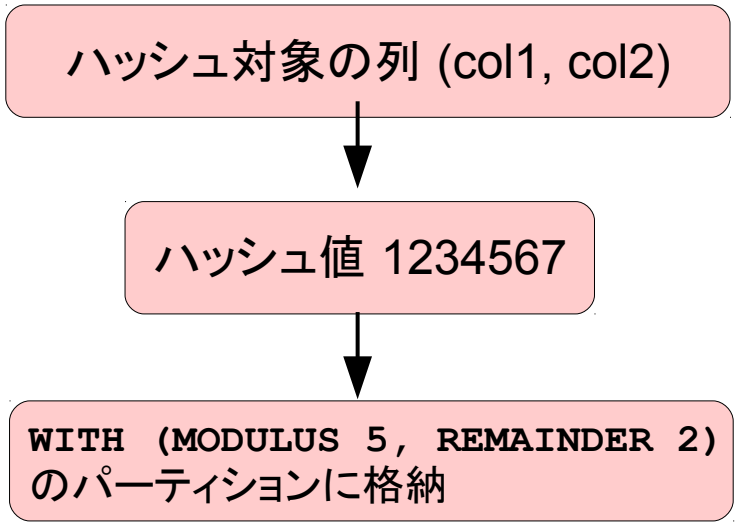
パーティショニング機能の拡充(3)

- パーティション間の行更新によるデータ移動




```
UPDATE t_log
 SET ts = '2018-08-03 12:00'
 WHERE log_id = 12345;
```

- ハッシュパーティショニング
  - キー列のハッシュ値(正整数値)の剰余でパーティション分けを行う
  - 各パーティションに行が均一に配付される
    - パラレル全件スキャンに向いている



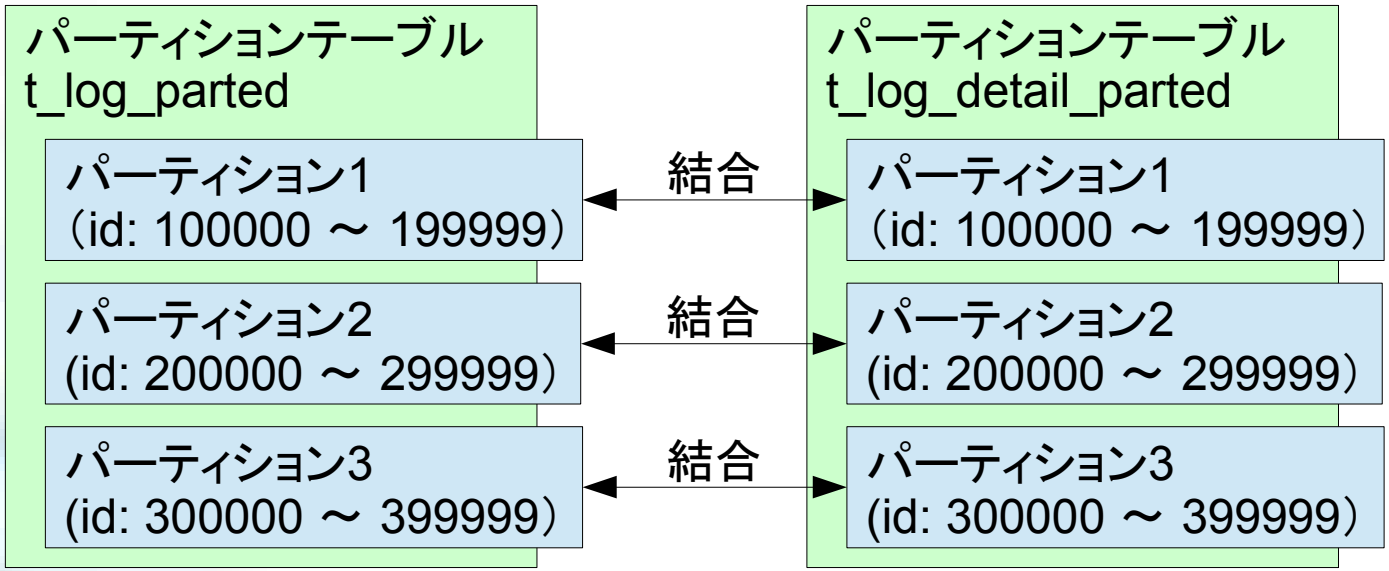
パーティショニング機能の拡充(4)

- 問い合わせでのパーティション除外処理の改善
  - 従来の constraint_exclusion では除外できなかったケースに対応
    - 「`SELECT count(*) FROM t_log WHERE ts < '2017-01-01'`」など
    - プラン作成時点では判別できないケースでも実行時に除外する
    - enable_partition_pruning = on で有効になる(デフォルト)

- 問い合わせでのパーティション指向の結合／集約

結合の例:

```
SELECT lo.mes, ld.detail FROM t_log_parted lo
  LEFT JOIN t_log_detail_parted ld ON (lo.id = ld.id);
```



パラレル処理の拡充（1）

- 並列ハッシュ結合
- 並列Append

新しいプランナ要素の追加
ほとんどの主要プランナ要素が並列化された

- 並列CREATE TABLE .. AS
  - 並列SELECT .. INTO ..
  - 並列CREATE MATERIALIZED VIEW ..

これまで、これら構文では、
並列処理が行われなかった。

- 並列インデックス作成

```
db1=# CREATE INDEX ON t_log (ts, id);
DEBUG:  building index "t_log_0_ts_id_idx" on table
"t_log_0" with request for 1 parallel worker
DEBUG:  building index "t_log_1_ts_id_idx" on table
"t_log_1" with request for 1 parallel worker
DEBUG:  building index "t_log_2_ts_id_idx" on table
"t_log_2" serially
CREATE INDEX
```

他にも多数の
細かな
パラレル対応

並列数はmax_parallel_maintenance_workers 設定の影響を受ける／Btreeのみ

パラレル処理の拡充（2）

- 並列ハッシュ結合
- 並列Append

例

```
b1=# explain SELECT max(length(l.mes || d.detail)) FROM t_log l
               LEFT JOIN t_log_detail d ON (l.id = d.id);
                            QUERY PLAN
-----------------------------------------------------------------
Finalize Aggregate (cost=15810.11..15810.12 rows=1 width=4)
  -> Gather (cost=15809.89..15810.10 rows=2 width=4)
  Workers Planned: 2
    -> Partial Aggregate (cost=14809.89..14809.90 rows=1 width=4)
    -> Parallel Hash Left Join
    (cost=6192.53..14028.64 rows=104167 width=66)
      Hash Cond: (lo.id = ld.id)
      -> Parallel Append
      (cost=0.00..4569.43 rows=104168 width=37)
        -> Parallel Seq Scan on t_log_0 lo
        (cost=0.00..1619.24 rows=58824 width=37)
        -> Parallel Seq Scan on t_log_1 lo_1
        (cost=0.00..1619.24 rows=58824 width=37)
                        :
```

SCRAMチャンネルバインド

- SCRAM認証がチャンネルバインドに対応
  - SSL接続の場合に意味がある
  - セッション固有情報を認証に使う
    ↓
    やりとり内容を別のセッションに使いまわすことができない
    ↓
    Man-in-the-middle攻撃を防ぐ
  - SSL接続 + scram-sha-256認証 であればデフォルトで使われる

方式は接続文字列で指定（他には tls-server-end-point か 空欄）:

```
psql "host=test1.example.com dbname=db1 user=test
sslmode=require scram_channel_binding=tls-unique"
```

（実際は一行）

その他（1）ウィンドウ関数の対応構文追加

- GROUPSウィンドウフレーム対応
- ウィンドウフレームRANGEモード対応
- ウィンドウフレームのEXCLUDEオプション

欠けていた
機能の補完

```
db1=# SELECT * FROM t_temperature2 ;
 id |         dt          |   t
----+---------------------+-------
   1 | 2018-08-03 00:00:00 | 18.20
   2 | 2018-08-03 00:11:18 | 20.74
   3 | 2018-08-03 00:23:49 | 22.65
   4 | 2018-08-03 00:35:41 | 21.51
   5 | 2018-08-03 00:38:29 | 20.13
     :                     :
     :                     :
 142 | 2018-08-03 22:40:37 | 27.74
 143 | 2018-08-03 22:55:29 | 26.32
 144 | 2018-08-03 23:14:59 | 24.40
 145 | 2018-08-03 23:27:11 | 23.61
 146 | 2018-08-03 23:47:06 | 27.40
(146 rows)
```

例えば、一定でない間隔で採取された
8月24日のある所の気温データがあるとして･･･

その他(2) ウィンドウ関数の対応構文追加

```
SELECT dt, t,
   round(avg(t) OVER (PARTITION BY to_char(dt,'YYYYMMDD-HH24')), 2) AS t_avg,
   round(avg(t) OVER (
     ORDER BY to_char(dt,'YYYYMMDD-HH24')
     GROUPS BETWEEN 1 PRECEDING AND 1 FOLLOWING), 2) AS t_avg3
 FROM t_temperature2;
```

```
         dt          |   t   | t_avg | t_avg3
---------------------+-------+-------+--------
                     :       :       :
 2018-08-24 08:30:18 | 29.28 | 29.06 |  28.48
 2018-08-24 08:39:33 | 30.59 | 29.06 |  28.48
 2018-08-24 08:56:40 | 29.82 | 29.06 |  28.48
 2018-08-24 09:15:05 | 27.54 | 28.14 |  29.00
 2018-08-24 09:32:42 | 29.11 | 28.14 |  29.00
 2018-08-24 09:40:13 | 27.75 | 28.14 |  29.00
 2018-08-24 09:45:58 | 28.14 | 28.14 |  29.00
 2018-08-24 10:00:06 | 29.11 | 29.62 |  29.91
 2018-08-24 10:12:01 | 29.15 | 29.62 |  29.91
 2018-08-24 10:19:12 | 28.72 | 29.62 |  29.91
 2018-08-24 10:39:05 | 29.65 | 29.62 |  29.91
 2018-08-24 10:52:36 | 31.47 | 29.62 |  29.91
 2018-08-24 11:12:09 | 32.01 | 31.12 |  31.47
 2018-08-24 11:13:22 | 30.55 | 31.12 |  31.47
 2018-08-24 11:20:27 | 31.89 | 31.12 |  31.47
                     :       :       :
```

t_avg は含まれる1時間単位の平均。
「年月日時」でパーティション区切り。
これは以前から使えた機能。

t_avg3 は含まれる前後3時間の平均。
「年月日時」順で並べて同じ値を
持つ前後1件をフレームに含めている。

その他(3) ウィンドウ関数の対応構文追加

```
SELECT dt, t,
  round(avg(t) OVER (ORDER BY dt RANGE BETWEEN
   '1 hour'::interval PRECEDING AND '1 hour'::interval FOLLOWING),2) AS t_avg_1h,

  round(avg(t) OVER (ORDER BY dt RANGE BETWEEN
   '1 hour'::interval PRECEDING AND '1 hour'::interval FOLLOWING
   EXCLUDE CURRENT ROW ), 2) AS t_avg_1hx     FROM t_temperature2;
```

dt	t	t_avg_1h	t_avg_1hx
:	:	:	
2018-08-24 08:30:18	29.28	28.74	28.68
2018-08-24 08:39:33	30.59	28.77	28.57
2018-08-24 08:56:40	29.82	28.72	28.61
2018-08-24 09:15:05	27.54	28.94	29.09
2018-08-24 09:32:42	29.11	28.88	28.85
2018-08-24 09:40:13	27.75	28.78	28.91
2018-08-24 09:45:58	28.14	28.78	28.86
2018-08-24 10:00:06	29.11	28.96	28.94
2018-08-24 10:12:01	29.15	28.96	28.94
2018-08-24 10:19:12	28.72	29.57	29.66
2018-08-24 10:39:05	29.65	30.04	30.08
2018-08-24 10:52:36	31.47	30.53	30.43
2018-08-24 11:12:09	32.01	31.06	30.97
2018-08-24 11:13:22	30.55	31.06	31.11
2018-08-24 11:20:27	31.89	31.30	31.23
:	:	:	

t_avg_1h は前後1時間の平均。
当該行の値との距離で
フレーム範囲を決めている。

t_avg_1hx は前後1時間から
当該行を除いた平均。
EXCLUDEで除外している。

その他(4) ALTER TABLE .. ADD COLUMN 改善

- デフォルト値を伴った ADD COLUMN が高速化

```
db1=# CREATE TABLE t_alt (id int primary key, c1 int);
db1=# INSERT INTO t_alt
        SELECT g, g FROM generate_series(1, 100000) AS g;
db1=# \timing
Timing is on.
db1=# ALTER TABLE t_alt ADD c2 int;
ALTER TABLE
Time: 16.918 ms
db1=# ALTER TABLE t_alt ADD c3 int DEFAULT 100;
ALTER TABLE
Time: 25.824 ms  ← 同マシンPostgreSQL 10では 600～700ms
```

デフォルト値を行データに格納しない。

- ALTER TYPE は従来と変わらず遅い

```
db1=# ALTER TABLE t_alt ALTER c3 TYPE bigint;
ALTER TABLE
Time: 640.206 ms
db1=# ALTER TABLE t_alt ALTER c3 TYPE int;
ALTER TABLE
Time: 602.989 ms
```

その他(5)

- CREATE INDEX が INCLUDE句に対応
  - index-only-scan用

```
CREATE INDEX idx1234 ON t1 (c1, c2) INCLUDE (c3, c4);
```

- initdb時にWALファイルサイズ設定
  - 従来は --with-wal-segsize=NN として ビルド時に指定
  - blocksize、wal-blocksize については変わらずビルド時に指定
- ロジカルレプリケーションでTRUNCATE対応
- ビューに対するテーブルロック
- 各種 psql 、pgbench の機能追加
- 各種のロック軽減とオプティマイザ改良

まとめ

- 強力な開発体制を持つ PostgreSQL の適用範囲は
  十分に広い
- PostgreSQL 11 は未実現機能を埋めていくバージョンアップ
- 累積データに対する分析問い合わせ むけの拡充
  - JIT ／ パラレル問い合わせ ／ パーティショニング ／ ウィンドウ関数
- Oracle からの移植でのニーズによる拡充
  - プロシージャでの COMMIT / ROLLBACK
- クラウド利用をにらんだ SSL接続の拡充
  - SCRAMチャンネルバインド

オープンソースとともに

SRA OSS,INC.

URL：http://www.sraoss.co.jp/
E-mail：sales@sraoss.co.jp
Tel：03-5979-2701