SRA OSS, INC.

PostgreSQL 12
新機能解説

2019-08-02
OSC 2019 Kyoto
SRA OSS, Inc. 日本支社
近藤 雄太

講演者 と 本講演のテーマ

【講演者】

名前

近藤 雄太

所属

SRA OSS, Inc. 日本支社

略歴

PostgreSQLのセミナ講師、サポート、コンサルティング等に従事。

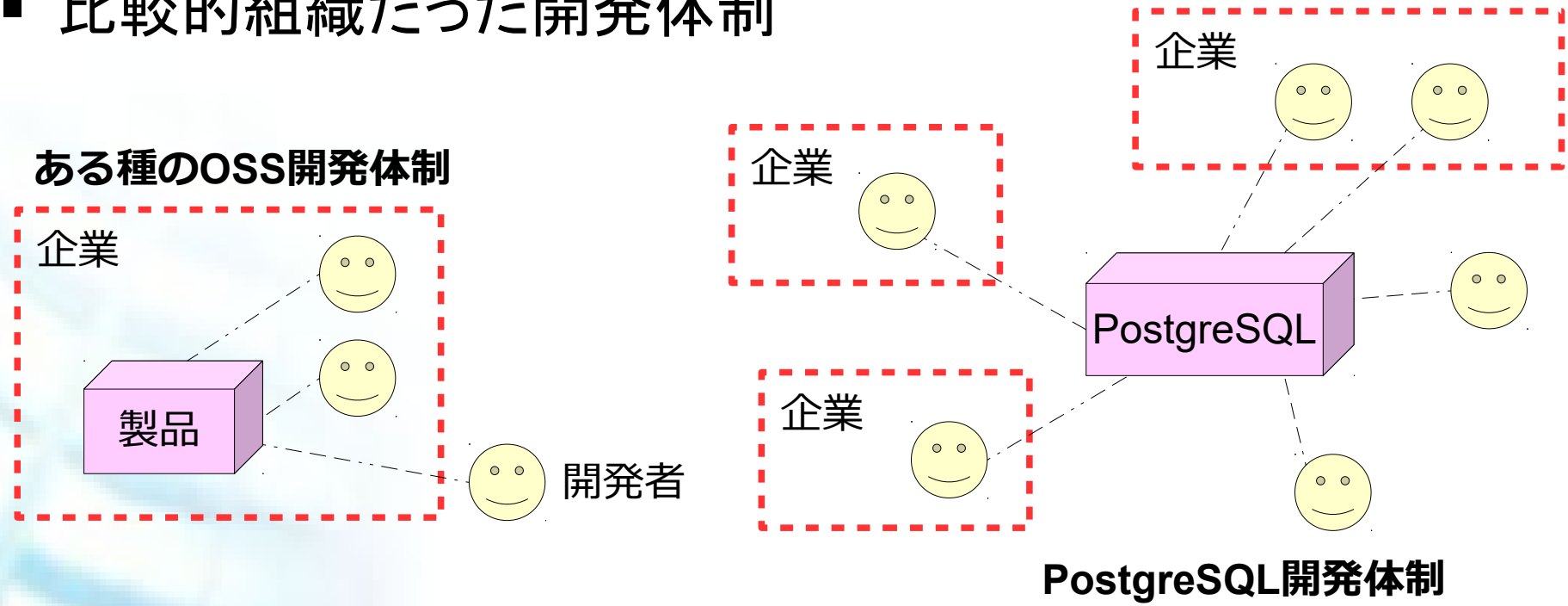
最近はPostgreSQLをベースにしたデータベース製品PowerGresの開発リーダーを務める

【本日の講演内容】

- PostgreSQL の概況
- バージョン12 の新機能解説

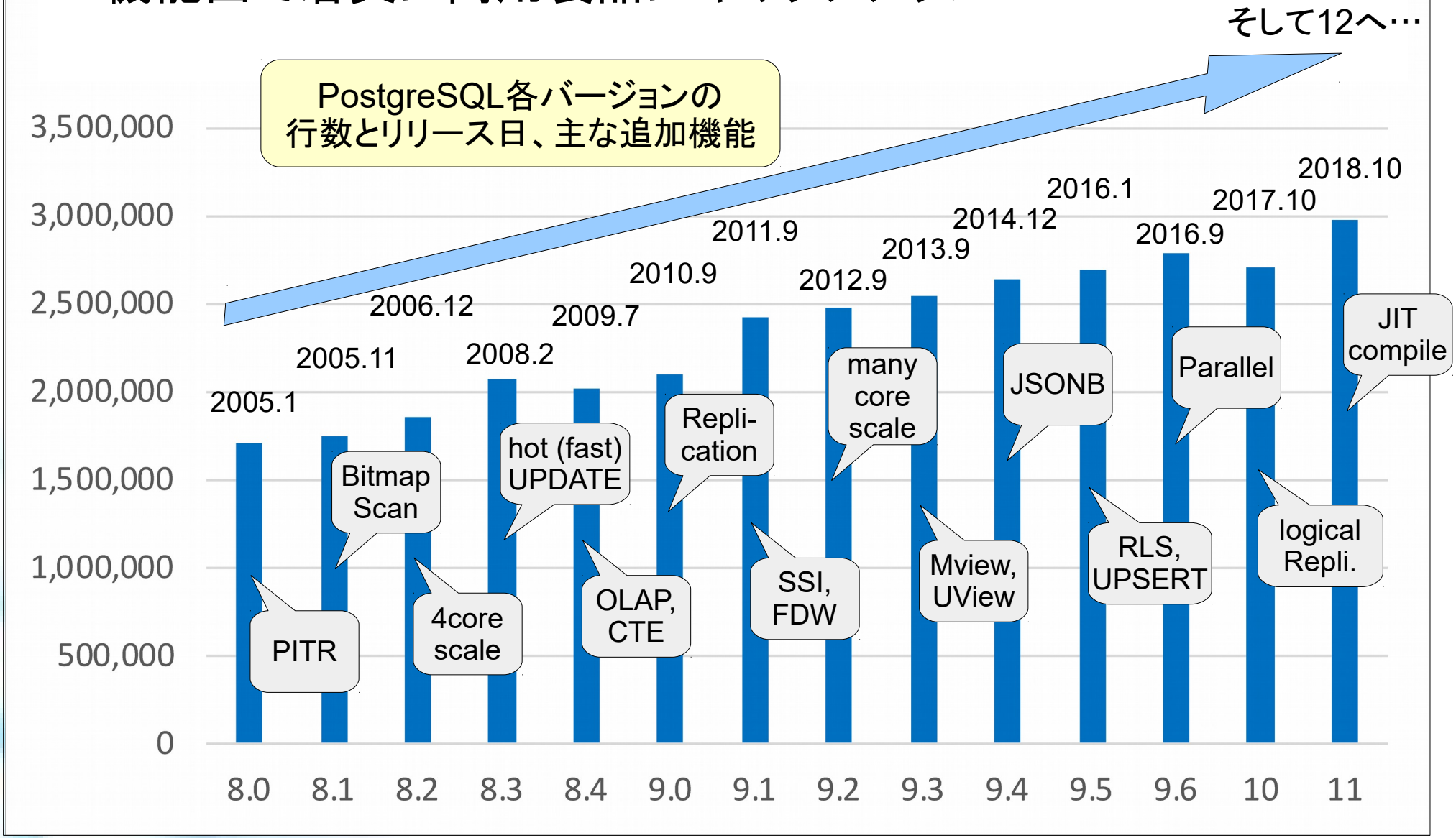
PostgreSQLの概況 (1)

- 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（RDBMS）
- INGRES('70)、POSTGRES('80) 由来の歴史
- 利用、複製、修正、再配布可能なライセンス
- 特定オーナー企業が無い
- 比較的組織だった開発体制



PostgreSQLの概況（2）

- 安定的で管理されたリリース
- 機能面で着実に商用製品にキャッチアップ



PostgreSQLの概況（3）

性能面

- ○ 多数のCPUコアによるOLTP読み書き性能スケール（単純ベンチマーク条件）
- ○ パラレル問い合わせ／JITに対応 （OLAPむけに）
- × 更新でWALボトルネック有

運用機能

- ○ PITRバックアップ
- ○ 監査ログ
- ○ クラウドによるマネージドDB
- × フラッシュバック／時系列

機能面

- ○ SQL SQL:2011 機能のほとんどをサポート
- ○ JSON対応での優越
- ○ PostGIS による GIS対応
- × マテビュー差分更新など

クラスタ構成

- ○ 論理・物理のレプリケーション
- ○ アクティブ＝スタンバイ構成
- ○ 参照負荷分散／シャーディング
- ○ 両方向論理レプリケーション
- × 共有ストレージを用いたアクティブ＝アクティブ構成（いわゆるRAC型）

PostgreSQL12の新機能

- JSON path
- 生成列（式を使って計算される列）
- 大文字小文字を区別しない比較

} アプリケーションのための SQL機能追加。

- インデックスの機能追加・性能改善
- パーティショニングの改善
- プランナのWITH句の最適化
- MCV拡張統計情報

} プラン作成や実行処理の改善を通した、実行時の性能アップ。

- ページチェックサム改善
- 認証処理の拡充

} 運用時の利便性向上。構成設計上の選択肢が広がる。

- テーブルAMインタフェース追加
- いくつかの仕様変更

— 将来のPostgreSQL拡張の布石

JSON Path (1)

- JSON Path … JSON から要素を抽出する記述方式の一つ
  - SQL:2016 に含まれる ／ PostgreSQL実装は少し異なる
  - jsonb型のみ対応 （json型は非対応）

```
db1=# SELECT family FROM t_family WHERE id = 1;
                    family
-----------------------------------------------
 [{"age": 40, "att": "head",   "name": "Tarou"},
  {"age": 35, "att": "spouse", "name": "Hanako"},
  {"age": 10, "att": "child",  "name": "Shou"}]
(1 row)

db1=# SELECT family_name,
             jsonb_path_query(family,
             '$[*]?(@.att == "head").name')
        FROM t_family WHERE id = 1;
 family_name | jsonb_path_query
-------------+------------------
 Satou       | "Tarou"
(1 row)
```

属性（att）が筆頭者（head）である人の名前（name）

JSON Path (2)

- JSON Path式は jsonpathデータ型の値
- JSON Path 検索で GINインデックスが利用可
  - より性能に優れた専用演算子クラス jsonb_path_ops
  - 従来からある jsonb のデフォルト演算子クラス jsonb_ops も動作

```
db1=# CREATE INDEX t_family_family_idx ON t_family
        USING gin (family jsonb_path_ops);

db1=# explain SELECT id FROM t_family
                WHERE family @@ '$[*].age < 15';
                        QUERY PLAN
-----------------------------------------------------------------
 Bitmap Heap Scan on t_family  (cost=56.00..60.01 rows=1 width=4)
   Recheck Cond: (family @@ '($[*]."age" < 15)'::jsonpath)
   ->  Bitmap Index Scan on t_family_family_idx  (cost=0.00..56.0
         Index Cond: (family @@ '($[*]."age" < 15)'::jsonpath)
(4 rows)
```

15才未満の家族を持つ人

生成列(1)

- 式を使って計算される列
  - 生成式の列についても実体が格納される

```
=# CREATE TABLE t_people (id int primary key,
    height_cm float4,
    height_in float4 GENERATED ALWAYS AS (height_cm / 2.54) STORED);

=# INSERT INTO t_people (id, height_cm)
    VALUES (1, 156), (2, 168), (3, 172), (4, 185), (5, 191);

=# SELECT * FROM t_people ;
 id | height_cm | height_in
----+-----------+-----------
  1 |       156 | 61.417324
  2 |       168 |  66.14173
  3 |       172 |  67.71654
  4 |       185 |  72.83465
  5 |       191 |  75.19685
(5 rows)

=# UPDATE t_people SET height_in = 61.4 WHERE id = 1;
ERROR:  column "height_in" can only be updated to DEFAULT
DETAIL:  Column "height_in" is a generated column.
```

height_in には自動計算された値が格納される

直接変更はエラーになる

生成列(2)

- 従来からあるビューでも似たことができる

```
CREATE TABLE t_people_base (
  id int primary key, height_cm float4);
CREATE VIEW v_people AS
  SELECT id, height_cm, (height_cm/2.54)::float4 AS height_in
  FROM t_people_base;
```

二つの方法の違い

	生成列を持つ テーブル	更新可能ビュー
列を計算する時点:	データ挿入・更新時	データ参照時
計算する列を直接操作しない データ更新処理:	INSERT、UPDATE、 DELETE が可能	INSERT、UPDATE、 DELETE が可能
計算する列の格納データ:	実体を持つ	実体を持たない
計算する列に対する インデックスの作成:	可能	元となるテーブルに 式インデックスを作る

大文字小文字やアクセントを区別しない文字列比較

- COLLATE（文字照合）機能の拡張
  - ICUライブラリ を使って（--with-icu）ビルドした場合
  - ICUライブラリが対応している同一視比較ができる
  - 新しい ICUライブラリバージョンを要する
  - 大文字/小文字、アクセント記号の有無、平仮名/片仮名/半角

```
db=# CREATE COLLATION ja_case_insensitive (provider = icu,
       locale = 'ja-u-ks-level2', deterministic = false);

db=# SELECT * FROM t_case WHERE txt = 'ああ'
       COLLATE ja_case_insensitive;
 id | txt
----+-----
  1 | ああ
  2 | アア
  3 | ｱｱ
(3 rows)
```

新たなオプション。
false に指定して、同一視させる。
true だと従来通りの厳格な同一比較。

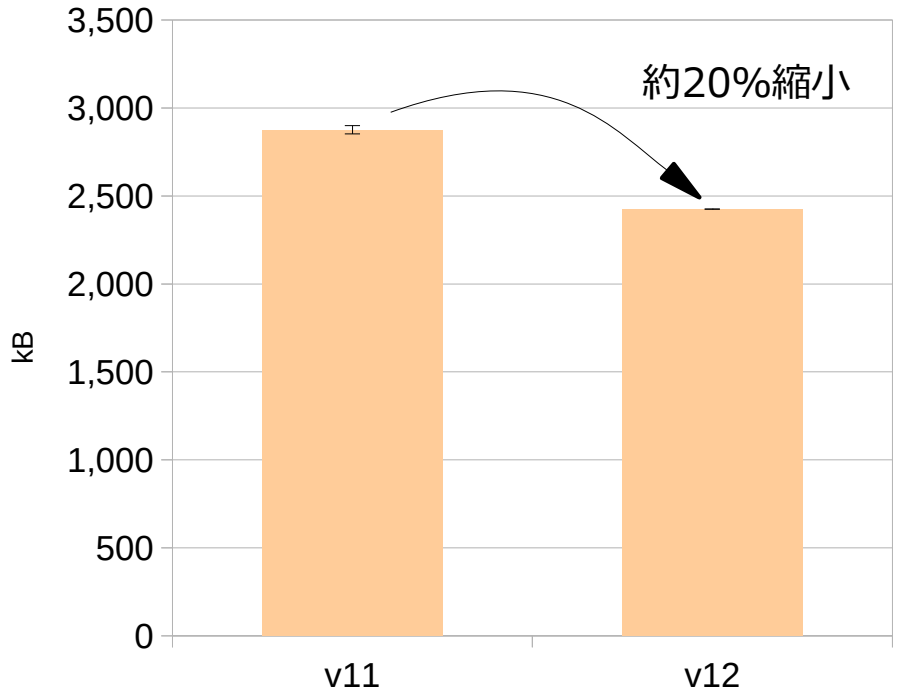
インデックスの機能追加・性能改善（1）

改善箇所は多岐にわたる：

- Btreeインデックス性能改善
  - キー重複が多くある場合に高速化／サイズを小さく
  - 複合インデックスのサイズを小さく
  - 内部ロック軽減で多コア同時処理を効率化
- REINDEX が CONCURRENTLY に対応
- GiST、SP-GiST、GIN の WAL出力量を削減
- GiST が INCLUDE に対応
- SP-GiST で kNN近傍探索に対応
- GiST に対する VACUUM 改善
- 他にもいくつか

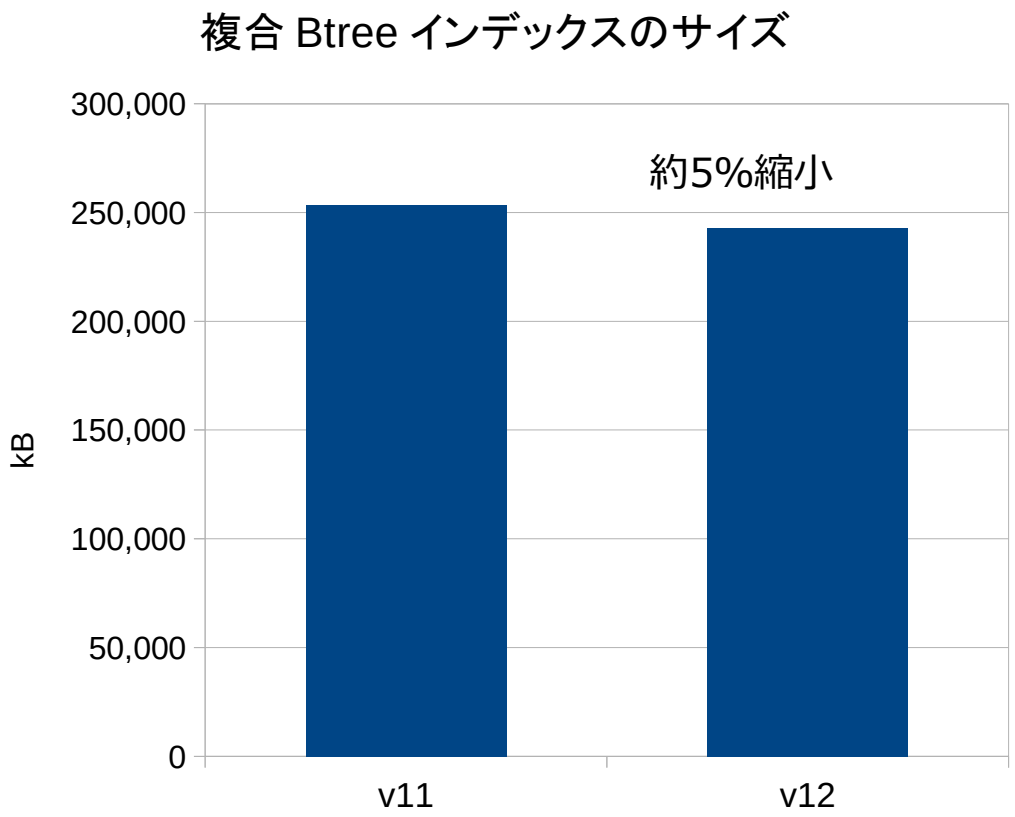
インデックスの機能追加・性能改善（2）

重複値を多く持つ Btree インデックスのサイズ




```
CREATE TABLE t_idxtest1
 (id int primary key, code bigint);
CREATE INDEX t_idxtest1_code_idx
  ON t_idxtest1 (code);
INSERT INTO t_idxtest1 SELECT g,
  (random() * 100)::bigint
  FROM generate_series(1, 100000) as g;
```

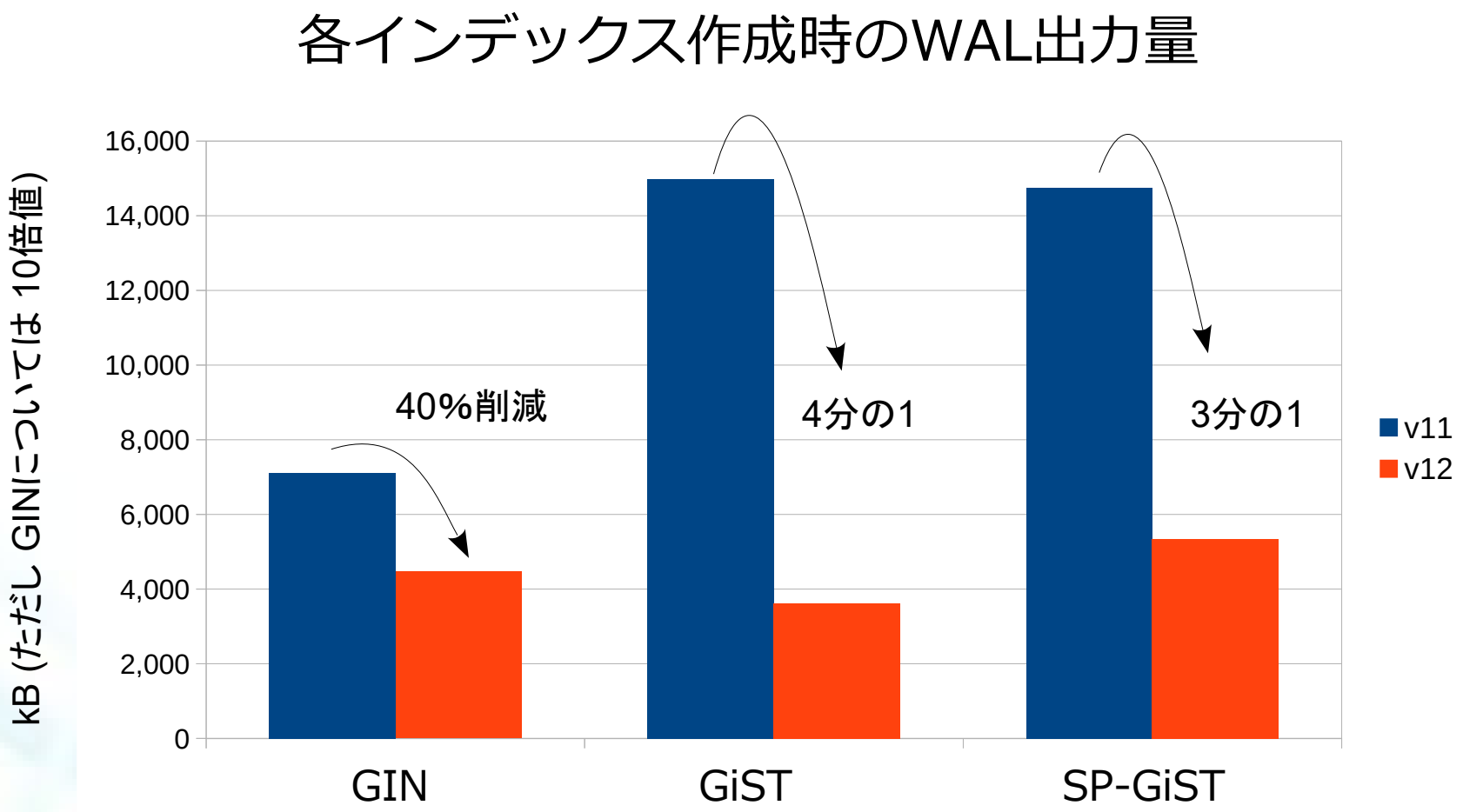
インデックスの機能追加・性能改善（3）




```
CREATE TABLE t_idxtest2
 (k1 text, k2 text, k3 text, k4 text);
CREATE INDEX ON t_idxtest2
 (k1, k2, k3, k4);

INSERT INTO t_idxtest2 SELECT
    : {適当なダミーデータ}
  FROM generate_series(1, 500000) as g;
 (× 2回)
```

インデックスの機能追加・性能改善(4)



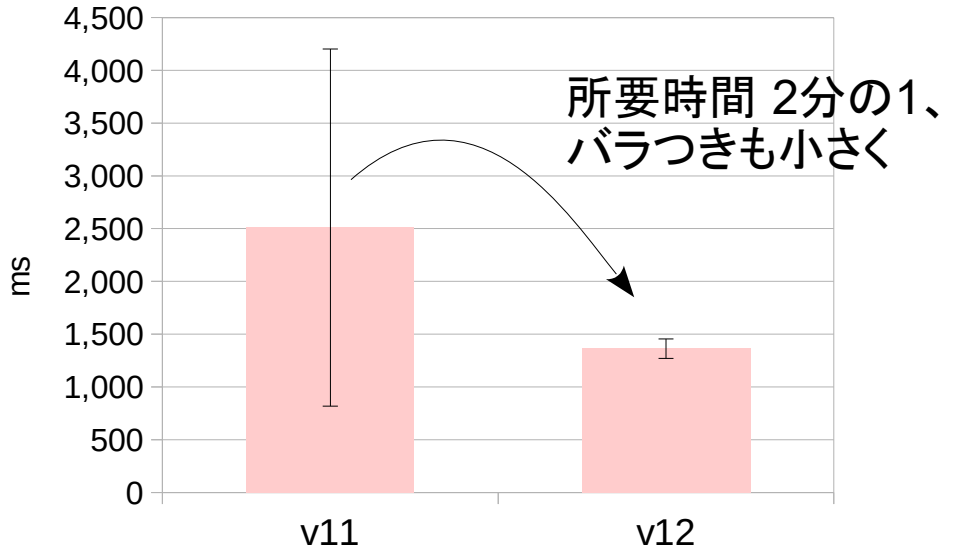
※100万件のダミーデータに各種インデックスを作成して、
WAL の LSN進行量を計測

パーティショニングの改善（1）

- 性能面
  - 多パーティションに対する検索の高速化
  - COPY TO によるデータ投入の高速化
  - SQL実行で無駄なソートを省略する
- 機能面
  - 外部キー制約で被参照側でもパーティションテーブル対応
  - パーティションを調べるビュー、関数、psqlコマンドの拡充
  - パーティション定義構文で定数でなく式を使用可能に
  - パーティションテーブルでテーブルスペース指定
- 他にもいくつか

パーティショニングの改善(2)

多パーティションに COPY データ投入

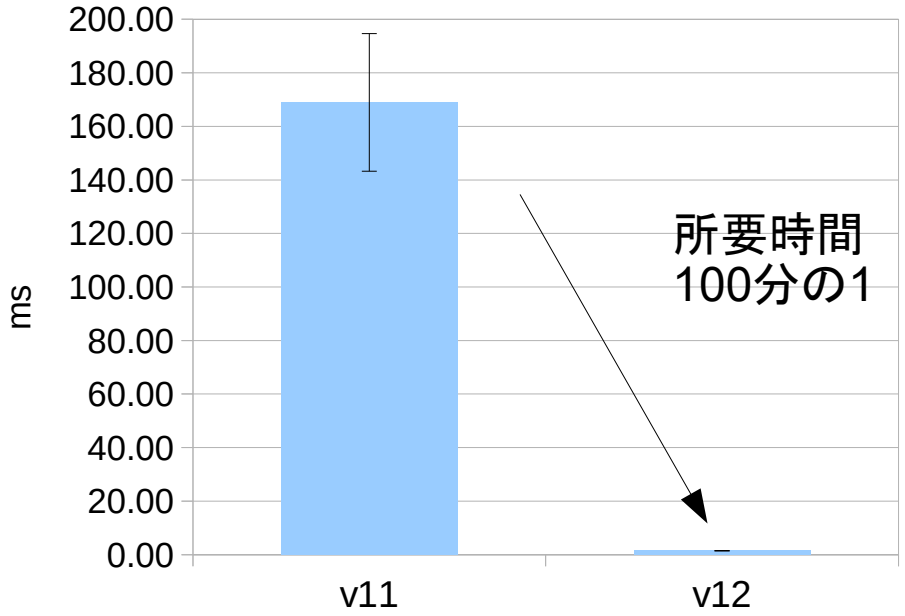


- 3,000パーティションを持つパーティションテーブル に各10件ずつ 30,000件のデータを投入

```
CREATE TABLE oya (id int primary key, v text) PARTITION BY RANGE (id);
CREATE TABLE ko1 PARTITION OF oya FOR VALUES FROM (0) TO (10);
CREATE TABLE ko2 PARTITION OF oya FOR VALUES FROM (10) TO (20);
..
```

パーティショニングの改善(3)

一部パーティションのみ参照するSQL



- 同テーブルで以下SQLを実行。
  3000パーティション中、3パーティションのみ該当。

```
SELECT * FROM oya
  WHERE id IN (100, 1000, 10000);

  id   |                 v
-------+----------------------------------
   100 | f899139df5e1059396431415e770c6dd
  1000 | a9b7ba70783b617e9998dc4dd82eb3c5
 10000 | b7a782741f667201b54880c925faec4b
(3 rows)
```

プランナのWITH句最適化（1）

- WITH句の節（CTE、共通テーブル式）も最適化される

従来のプラン： WITH句は書いた通りに独立して処理される

```
EXPLAIN
 WITH cte AS (SELECT * FROM t_customer)
 SELECT * FROM t_sales JOIN cte USING (customer_id)
  WHERE cte.info ~ '^abcde';
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=42470.00..53695.48 rows=5123 width=73)
   Hash Cond: (cte.customer_id = t_sales.customer_id)
   CTE cte
     ->  Seq Scan on t_customer  (cost=0.00..1834.00 rows=100000 width=37)
   ->  CTE Scan on cte  (cost=0.00..2250.00 rows=500 width=36)
         Filter: (info ~ '^abcde'::text)
   ->  Hash  (cost=19346.00..19346.00 rows=1000000 width=41)
         ->  Seq Scan on t_sales  (cost=0.00..19346.00 rows=1000000 width=41)
(8 rows)
```

info列にインデックスが在っても、全件取得してから絞り込む、賢くないプラン

最適化のために以下のようにサブクエリに書き直す必要があった

```
SELECT * FROM t_sales JOIN
 (SELECT * FROM t_customer) c USING (customer_id)
  WHERE c.info ~ '^abcde';
```

プランナのWITH句最適化(2)

PostgreSQL12 のプラン： WITH句も最適化の対象となる

```
EXPLAIN
 WITH cte AS (SELECT * FROM t_customer)
 SELECT * FROM t_sales JOIN cte USING (customer_id) WHERE cte.info ~ '^abcde';
                                  QUERY PLAN
-------------------------------------------------------------------------------

 Gather  (cost=1008.57..15625.03 rows=100 width=74)
   Workers Planned: 2
   ->  Hash Join  (cost=8.56..14615.03 rows=42 width=74)
         Hash Cond: (t_sales.customer_id = t_customer.customer_id)
         ->  Parallel Seq Scan on t_sales
             (cost=0.00..13512.67 rows=416667 width=41)
         ->  Hash  (cost=8.44..8.44 rows=10 width=37)
               ->  Index Scan using t_customer_info_idx on t_customer
                   (cost=0.42..8.44 rows=10 width=37)
                     Index Cond: ((info >= 'abcde'::text) AND (info < 'abcdf'::t
                     Filter: (info ~ '^abcde'::text)
(9 rows)
```

WITH cte_name MATERIALIZED AS (…)	従来動作 ／ 必ず独立プラン
WITH cte_name AS (…)	最適化を許す
WITH cte_name NOT MATERIALIZED AS (…)	複数回評価する最適化も許す

MCV拡張統計情報

- 拡張統計情報とは:
  - CREATE STATISTICS コマンドでプランナ統計情報項目を追加
  - 作成後は ANALYZE処理で統計情報が収集される
  - v11 までのサポート項目

    dependencies:　列間の依存関係
    ndistinct:　複数列の組み合わせに対する値の種類数
- MCV
  - 複数列の組み合わせに対する Most Common Values（最頻値）

列 location の最頻値:

最頻値	東京	大阪	京都	…
出現率	0.10	0.09	0.08	…

列 age の最頻値:

最頻値	20代	30代	40代	…
出現率	0.20	0.15	0.10	…

列 location、age の組み合わせの最頻値:

最頻値	京都, 40代	東京, 20代	東京, 30代	大阪, 20代	大阪, 30代	…
出現率	0.05	0.02	0.01	0.01	0.01	…

個別の統計情報では見えてこない偏りを把握

ページチェックサム改善

- ページチェックサム機能とは:
  - initdb 時に (-k) –data-checksums オプションでチェックサム付加
  - ページアクセス時に異常を検出できる
    （チェックサム無しの場合、黙ってデータが変わったり、
    構造的に不正な場合だけエラーが出たりする）
  - コマンドによる明示的な検証も可能

12での拡張:

- 既存データベースクラスタを後からチェックサム有効化

```
pg_checksums --enable {データベースクラスタディレクトリ}
```

  - 検証用コマンド pg_verify_checksums が pg_checksums に
    統合された

認証処理の拡充

- GSSAPI認証での暗号化に対応
  - pg_hba.conf の第一フィールド（host や hostssl 等が書かれる所）に hostgssenc、hostnogssenc 指定が追加

- LDAP認証で DNS から LDAPサーバを探す動作に対応
  - SRVレコードを参照する

テーブルAMインタフェース追加

- プラガブルなテーブル格納インタフェース（＝Access Method）
  - 拡張モジュールでテーブル用のストレージエンジンを追加できる
  - 従来から インデックスAM は任意に追加可能、これに加えて、テーブルAM を追加できるようになった
  - テーブル毎にテーブルAM指定できる
- 今のところインタフェースのみ
- 開発されている実装
  - zheap:　UNDOログを使って VACUUMレスを実現
  - zedstore:　列指向格納で OLAP最適化を実現

非互換部分

- recovery.conf 廃止
  - リカバリ設定も postgresql.conf に記述する
  - リカバリ起動を指示するには、recovery.signal ファイルを置く
  - スタンバイ起動を指示するには、standby.signal ファイルを置く
- テーブルの行ごとの OID 廃止
  - CREATE TABLE で WITH OIDS 指定のサポートが除去
- システムテーブルの oid 列が隠されなくなった
  - SELECT * FROM pg_class で oid 列も出力される。
- その他、様々な細かな非互換修正
  - 設定パラメータ名変更（trigger_file → promote_trigger_file）
  - 設定値のデフォルト変更（jit = on...）
  - システムテーブル／ビューの定義変更（pgclass.relhasoids列の削除...）
  - etc.

12 は今までのメジャーバージョンに比べて？

- 既存機能の性能向上が主
- 12自体で「今すぐ使える」大きな機能追加はない
  - 9.6: パラレルクエリ
  - 10: 宣言的パーティショニング
  - 11: JITコンパイル
- しかし、将来の布石として・・・
  - テーブルアクセスメソッドインターフェースの実装
- PostgreSQL は力をためている・・・

オープンソースとともに

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