大規模ストアドプロシージャの開発・移行・検証

PGConf.ASIA 2018
DAY1 12/11 13:50 – 14:30 @ Track B

高塚 遥 TAKATSUKA Haruka　SRA OSS, Inc. Japan

- About this speaker
  - PostgreSQL についての ヘルプデスク、コンサルティング、構築導入、トレーナー等に15年以上従事

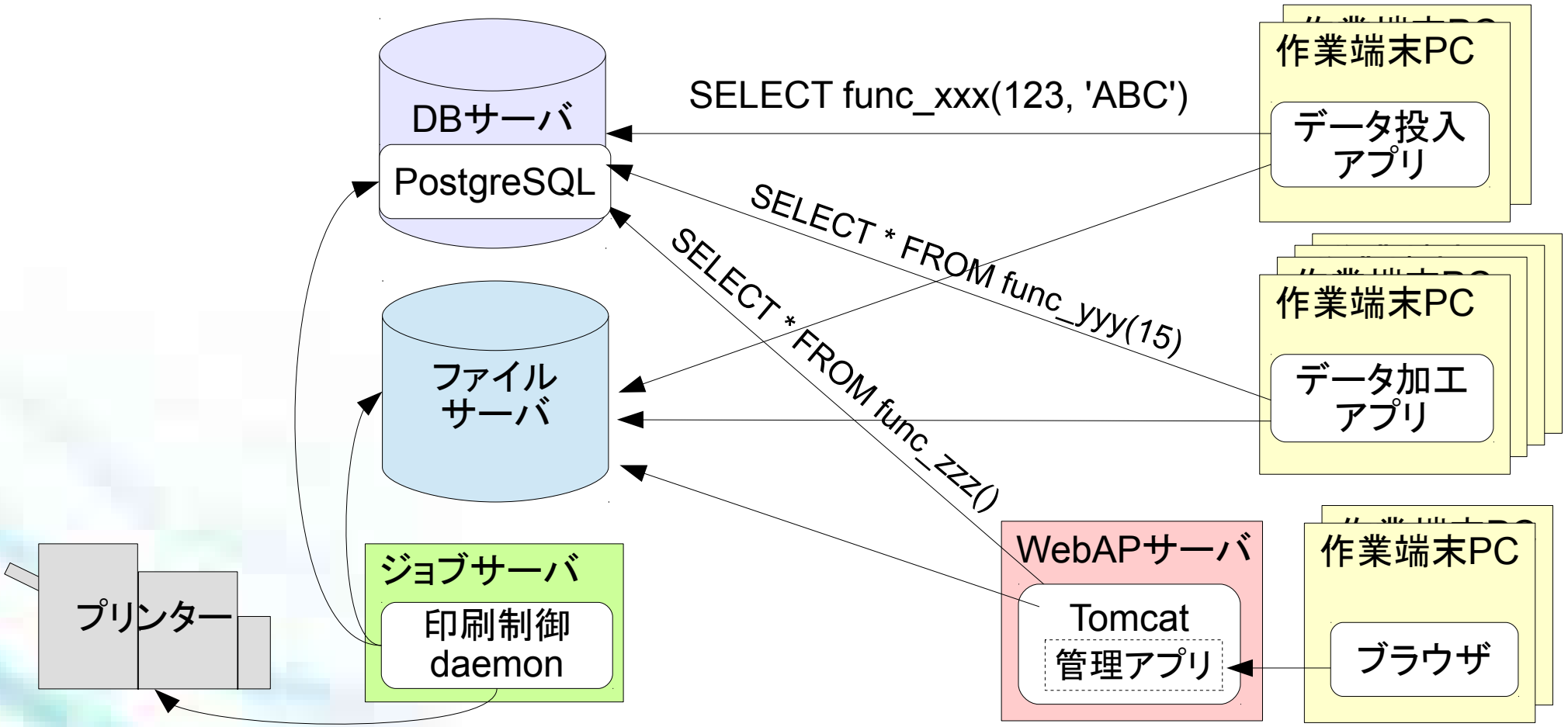
- About this talk
  - ストアドプロシージャ指向設計について
  - PostgreSQLのストアドプロシージャ指向の大規模開発
    - 有用なツール
    - 注意点
  - そのようなシステムを商用DBMSからの移行

ストアドプロシージャ指向(1)

- ストアドプロシージャ指向とは？
  - ストアドプロシージャ/ユーザ定義関数を多用する開発方針
    ・・・ 造語です
  - DBクライアントは必ずストアドプロシージャ/関数を使う
  - 先にインタフェースとなるプロシージャ仕様を決める
  - 主要ビジネスロジックはストアドプロシージャで書く
  - 「アプリケーション開発者がSQL嫌い」の対処手段の一つ
    - ORマッパー ・・・ DBは単なるデータストアに徹します
    - ストアドプロシージャ指向 ・・・ 大事な部分はDBでやります

ストアドプロシージャ指向(2)

- とあるシステム
  - 流れ作業での印刷物デザインをする業務システム



ストアドプロシージャ指向(3)

- メリット
  - DB / AP の疎結合を実現できる
  - ORマッパー由来の非効率ロジックを排除
    - 無駄なループ処理等
  - ロジックのオンライン差し替えが容易
- デメリット
  - 複数DBサーバ構成には対応しにくい
  - DB製品間のPL互換性はSQL文より低い
  - 分散しにくいDBサーバ上のリソースを使う
    - もったいない
  - ストアドプロシージャ記述言語の洗練度
    - 現代的な言語と比べると

大規模ストアドプロシージャ指向システムの開発

大規模ストアドプロシージャ開発

- ストアドプロシージャを大量に使用する大規模開発のために何が必要か？

  - 開発支援ツールを揃える
    - Java等で作るのと同じように！
    - ツールの充実という観点では手続き言語は PL/pgSQLが有利
  - ドキュメンティング
    - Java等で作るのと同じように！
  - テーブル定義と連動したバージョン管理
    - 注意　データベースGUIクライアントソフトでの開発
    - 注意　共用開発データベース
  - いくつかのストアドプロシージャ固有の注意点

ストアドプロシージャ開発のためのツール(1)

- plpgsql_check
  - PL/pgSQL のコード検査ツール
    - PL/pgSQL は 単純ミスがあっても CREATE時にはエラーが出ない
    - 登録済み関数群を一括チェックできる

```
CREATE FUNCTION f_activity_score(p_uid IN int, p_term interval)
RETURNS float4 LANGUAGE plpgsql AS $func$
DECLARE
  v_f0 int := 1;
  v_f1 int := 1;
  v_f2 int;
  v_score float4 := 0.0;
  r_score int;
BEGIN
  FOR r_score IN SELECT score FROM t_act JOIN m_act USING (actid)
    WHERE m_act.ts > CURRENT_TIMESTAMP AND t_act.uid = p_uid ORDER BY ts DESC
  LOOP
    v_score := v_score + 1.0 * r_score / v_f0;
    v_f2 := v_f0 + v_f1;
    v_f0 := v_f1;
    v_f1 := v_f2;
  END LOOP;
  RETURN v_score;
END;
$func$;
```

この部分は、
t_act.ts > CURRENT_TIMESTAMP - p_term
が、正しい

ストアドプロシージャ開発のためのツール(2)

- plpgsql_check 実行結果

```
-[ RECORD 1 ]-----------------------------------------------------------------
functionid | f_activity_score(integer,interval)
lineno     | 9
statement  | FOR over SELECT rows
sqlstate   | 42703
message    | column m_act.ts does not exist
detail     |
hint       | Perhaps you meant to reference the column "t_act.ts".
level      | error
position   | 60
query      | SELECT score FROM t_act JOIN m_act USING (actid)                +
           |     WHERE m_act.ts > CURRENT_TIMESTAMP AND uid = p_uid ORDER BY ts DESC
context    |
```

```
-[ RECORD 1 ]---------------------------------
functionid | f_activity_score(integer,interval)
lineno     |
statement  |
sqlstate   | 00000
message    | parameter "$2" is never read
detail     |
hint       |
level      | warning extra
position   |
query      |
context    |
```

ある関数について重大なエラーを見つけると、それ以降は調べてくれない。

直す → 再実行で別エラー報告 → 直すのサイクルが必要。

ストアドプロシージャ開発のためのツール(3)

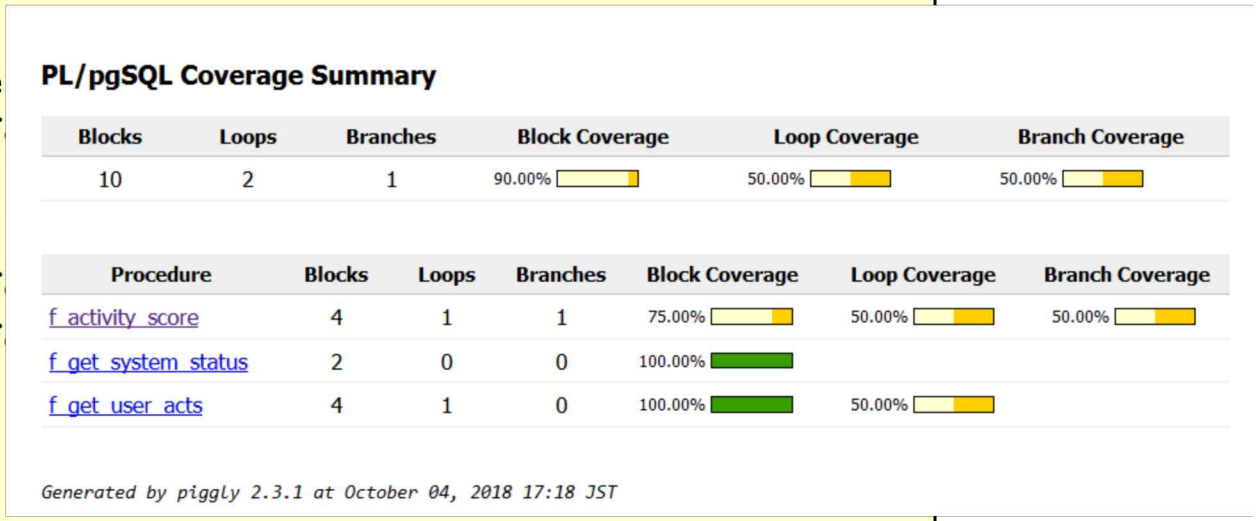
- piggly
  - PL/pgSQLカバレッジ検査ツール

トレース開始（調査用コード埋め込み）

```
$ piggly trace -d config/database.yml
```

psqlで SQL実行するテストを行うと、pigglyのWARNING出力が得られる

```
$ sh mytest/test1.sh
WARNING:  PIGGLY e0b64ad66a7e
WARNING:  PIGGLY f99d62107e6f
WARNING:  PIGGLY 3bbf263073a7
WARNING:  PIGGLY 1545828a1690
WARNING:  PIGGLY f99d62107e6f
WARNING:  PIGGLY f99d62107e6f
  :
```



PL/pgSQL Coverage Summary

Blocks	Loops	Branches	Block Coverage	Loop Coverage	Branch Coverage
10	2	1	90.00%	50.00%	50.00%

Procedure	Blocks	Loops	Branches	Block Coverage	Loop Coverage	Branch Coverage
f_activity_score	4	1	1	75.00%	50.00%	50.00%
f_get_system_status	2	0	0	100.00%		
f_get_user_acts	4	1	0	100.00%	50.00%	

Generated by piggly 2.3.1 at October 04, 2018 17:18 JST

これを元にHTMLレポートを生成

```
$ sh mytest/test1.sh &> test1.out
$ piggly report -o piggly/report -c piggly/cache -f test1.out
```

トレース終了（調査用埋め込みコードを除去）

```
$ piggly untrace -d config/database.yml
```

ストアドプロシージャ開発のためのツール(4)

- 出力例
  - 関数ソースコードに対応した詳細レポートが出力される

public.f_activity_score

Blocks	Loops	Branches	Block Coverage	Loop Coverage	Branch Coverage
4	1	1	75.00%	50.00%	50.00%

```
CREATE FUNCTION public.f_activity_score
        ( IN    p_uid   integer,
          IN    p_term  interval )
RETURNS real
  VOLATILE
```

```
1  DECLARE
2    v_f0 int := 1;
3    v_f1 int := 1;
4    v_f2 int;
5    v_score float4 := 0.0;
6    r_score int;
7  BEGIN
8    FOR r_score IN SELECT score FROM t_act JOIN m_act USING (actid)
9      WHERE t_act.ts > CURRENT_TIMESTAMP - p_term AND t_act.uid = p_uid ORDER BY t
10   LOOP
11     v_score := v_score + 1.0 * r_score / v_f0;
12     v_f2 := v_f0 + v_f1;
13     IF v_f2 > 100000 THEN
14       v_f2 := 100000;
15     END IF;
16     v_f0 := v_f1;
17     v_f1 := v_f2;
18   END LOOP;
19   RETURN v_score;
20 END;
```

Notes

1. loop always iterates more than once
2. never evaluates true
3. never evaluated

ループ 0回の実行パターンが無い、と指摘

条件が真になる実行パターンが無い、と指摘

この箇所が実行されていない、と指摘

ストアドプロシージャ開発のためのツール(5)

- 注意点
  - Ruby で書かれている、環境によってはビルド・インストール手間取るかも
  - 条件分岐の組み合わせ検査まではしてくれない

public.f_switch

Blocks	Loops	Branches	Block Coverage	Loop Coverage	Branch Coverage
6	0	2	100.00%		100.00%

Notes

```
CREATE FUNCTION public.f_switch ( IN p_i integer )
RETURNS int
  VOLATILE
1  DECLARE
2    r int;
3  BEGIN
4    IF p_i % 2
5      r := 0;
6    ELSE
7      r := 1;
8    END IF;
9    IF p_i % 3
10     r := 10
11   ELSE
12     r := 10
13   END IF;
14   RETURN r;
15 END;
```

```
3  BEGIN
4    IF p_i % 2 = 0 THEN
5      r := 0;
6    ELSE
7      r := 1;
8    END IF;
9    IF p_i % 3 = 0 THEN
10     r := 10 / r;
11   ELSE
12     r := 10 * r;
13   END IF;
14   RETURN r;
```

テスト内容:

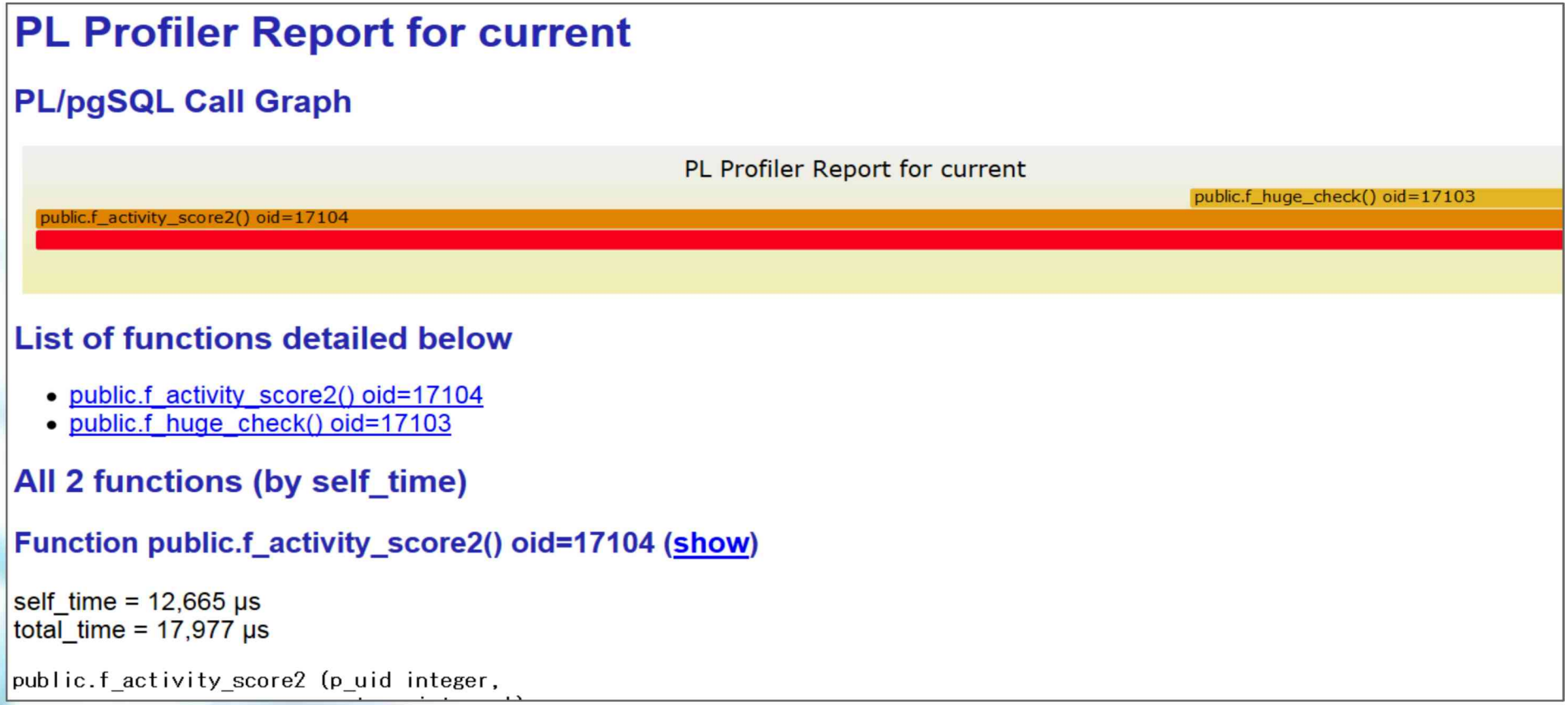
```
SELECT f_switch(1);
SELECT f_switch(2);
SELECT f_switch(3);
```

2でも 3でも割り切れる場合がテストされていない。
しかし、**Branch Coverrage** は **100%** になっている。

```
db=# SELECT f_switch(6);
ERROR:  division by zero
```

ストアドプロシージャ開発のためのツール(6)

- PLprofiler
  - PL/pgSQLむけのプロファイリングツール
    - 個別のSQL実行における計測分析、
      一定時間の稼動状態での計測分析 が可能
    - HTMLレポートを出力
    - C言語 ＋ Python による Hook を使った洗練された低負荷実装



ストアドプロシージャ開発のためのツール(7)

SRA OSS, INC.

- 関数内の部分ごとに所要時間が報告される

Function public.f_activity_score2() oid=17104 (hide)

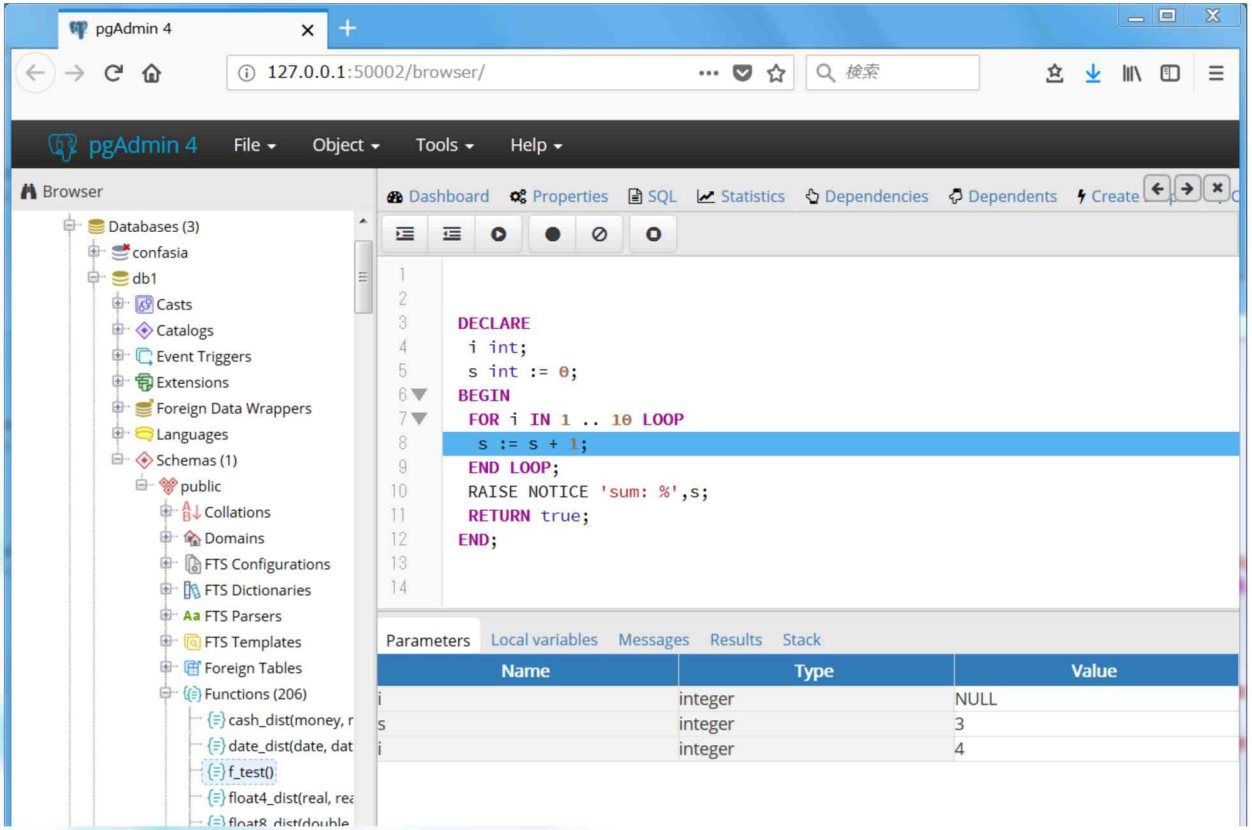
self_time = 12,665 µs
total_time = 17,977 µs

```
public.f_activity_score2 (p_uid integer,
                          p_term interval)
    RETURNS real
```

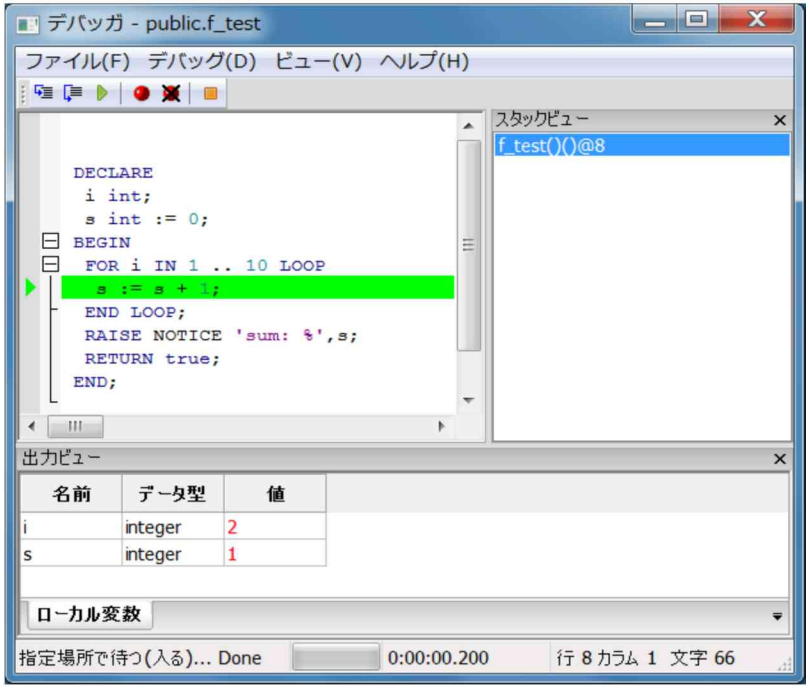
Line	exec_count	total_time	longest_time	Source Code
0	1	17,977 µs (100.00%)	17,977 µs	-- Function Totals
1	0	0 µs (0.00%)	0 µs	
2	0	0 µs (0.00%)	0 µs	DECLARE
3	0	0 µs (0.00%)	0 µs	v_f0 int := 1;
4	0	0 µs (0.00%)	0 µs	v_f1 int := 1;
5	0	0 µs (0.00%)	0 µs	v_f2 int;
6	0	0 µs (0.00%)	0 µs	v_score float4 := 0.0;
7	0	0 µs (0.00%)	0 µs	r_score int;
8	0	0 µs (0.00%)	0 µs	BEGIN
9	1	16,701 µs (92.90%)	16,701 µs	FOR r_score IN SELECT score FROM t_act JOIN m_act USING (actid)
10	0	0 µs (0.00%)	0 µs	WHERE t_act.ts > CURRENT_TIMESTAMP - p_term AND t_act.uid = p_uid ORDER BY ts DES
11	0	0 µs (0.00%)	0 µs	LOOP
12	114	2,917 µs (16.23%)	2,398 µs	v_score := v_score + 1.0 * r_score / v_f0;
13	114	190 µs (1.06%)	155 µs	v_f2 := v_f0 + v_f1;
14	114	7,629 µs (42.44%)	2,946 µs	IF f_huge_check(v_f2) THEN
15	91	28 µs (0.16%)	24 µs	v_f2 := 100000;
16	0	0 µs (0.00%)	0 µs	END IF;
17	114	65 µs (0.36%)	50 µs	v_f0 := v_f1;
18	114	175 µs (0.97%)	169 µs	v_f1 := v_f2;
19	0	0 µs (0.00%)	0 µs	END LOOP;
20	1	0 µs (0.00%)	0 µs	RETURN v_score;
21	0	0 µs (0.00%)	0 µs	END;
22	0	0 µs (0.00%)	0 µs	

ストアドプロシージャ開発のためのツール(8)

- pldebugger + pgAdmin
  - PL/pgSQLのデバッガ
    - PostgreSQL側にpldebugger拡張モジュールを導入し、pgAdmin4 (pgAdmin3) でステップ実行、変数ウォッチ等のデバッガ操作



最小限の



ストアドプロシージャ開発のためのツール(9)

- SQL のユニットテストフレームワーク
  - pg_regress ・・・ src/test/regress
  - pgUnit ・・・ PL/pgSQLベース
  - pgTAP ・・・ perlベース
  - testgres ・・・ pythonベース

  - PostgreSQL固有のツールである必要は無い
    - プロジェクトで既存のテストフレームワークで十分
    - 初期化 ⇒ SQL実行 ⇒「期待される結果」と比較 ⇒ 実行結果集計という点はどれも同じ

ストアドプロシージャ開発のためのツール(10)

- 開発支援ツール全般のポイント
  - Hook を使った拡張モジュールには相性問題がありえる
    - 使うときに導入して、使い終わったら無効にする
    - 本番サーバでは外しておきたい
    - shared_preload_library に書く順番で問題回避できる場合がある
  - 商用エディションのPostgreSQL
    - EnterpriseDB社製品など
    - この種の機能が様々同梱されている

ストアドプロシージャ固有の開発の注意点(1)

《各種の注意点》

- 識別子の長さ制限
  - 63 byte は窮屈
    - 多数のプロシージャを区別できる意味のあるプロシージャ名を付けるには
    - もっと上限が短いDBMS製品もありました
    - 「関数種別をあらわす接頭辞_機能モジュール名_機能コード」など、番号や機能コードを使ったものにせざるを得ない

ストアドプロシージャ固有の開発の注意点(2)

- エラー設計
  - エラー処理の方針決め： プロシージャ内 ⇔ 呼び出し元

    A) 読み出し元に意図せぬ全ての例外を外に出す
    B) システムレベル例外は拾う／代わりに固有の例外を外に出す
    C) システムレベル例外は拾う／基本的に例外を外に出さない

  - SQLSTATE 5文字は狭い
    - メッセージの接頭文字列やDETAILで固有の区分コードを

ストアドプロシージャ固有の開発の注意点(3)

- ログ設計
  - PostgreSQLログに出す
    - PostgreSQLログを後で仕分けできるように接頭文字列を加える
    - 拾ってしまった例外はログに出ない:
      GET STACKED DIAGNOSTICS で取得して記録
  - 独自ログ
    - 独自にログファイル書き出しするのは色々問題あり
    - plpythonu、plperlu なら syslog 出力できる
    - 「INSERT INTO log ...」は、ロールバック問題

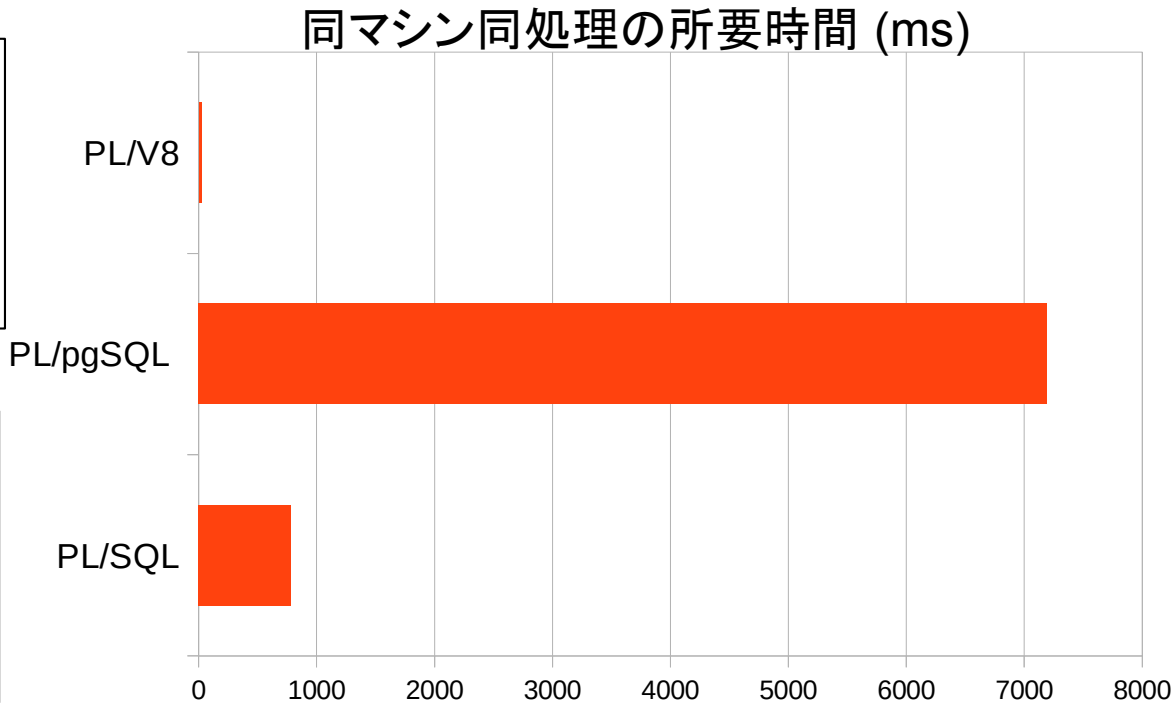
ストアドプロシージャ固有の開発の注意点(4)

- PL/pgSQLは遅い
  - 最速の PL/v8 と比べてロジック実行が 100倍遅い
    - 全てのデータ型、演算子が使えるというリッチな仕様の代償
    - 部分的に別のPL言語、あるいはC言語実装の関数を使う
    - できるだけループ処理を書かず、SQL一括処理を目指して性能改善

処理内容: 100万回の関数コール
- func1: 1000回ループで func2 コール
- func2: 1000回ループで func3 をコール
- func3: 引き数値に 1 を加えて返す
- 各関数で演算と変数代入

速い ---------- 遅い

C関数　PL/Python
PL/V8　PL/Perl
PL/SQL　PL/pgSQL



大規模ストアドプロシージャ指向システム
の
商用DBMS からの移植

ストアドプロシージャ指向システムの移植(1)

- 商用RDMBS からのマイグレーション
  - 移行元がストアドプロシージャ指向のシステムの場合
    - コード修正コストが余計にかかる
    - SQL文を移植するほうが楽
  - 基本手順:

    方針策定 ⇒ 機械変換 ⇒ 手動補正 ⇒ テスト

    - 手動補正の過程で変換方針に追加すべきことが加わっていく
    - 一次手動補正（意図の理解なし）→ 二次手動補正（意図の理解あり）とするか？
  - SQL移植と共通の課題 + プロシージャ固有の課題
    - 共通：NULLと空文字列、組み込み関数の非互換、データ型非互換、etc
    - プロシージャ固有：
      基本構文非互換、パッケージ変数、トランザクション処理対応、etc

ストアドプロシージャ指向システムの移植(2)

- プロシージャに変換ツールは使えるか？
  - ora2pg
    - ・・・ 柔軟な設定調整 / 対応できない箇所も多い
  - Ispirer
    - ・・・ マルチDB対応の商用製品 / 変換対応 は ora2pg と同程度くらいか
  - SQLines
    - ・・・ マルチDB対応の OSS / 上2つに比べるとやや劣る
  - EDB Postgres Migration Toolkit
    - ・・・最も優れているが ネイティブ PostgreSQL に移植するわけではない
  - AWS Schema Conversion Tool
    - ・・・ オンプレ同士目的にも使える ／ 変換不能レポートが良い
- パッケージは鬼門 ⇔ 大規模では当然パッケージ使用

ストアドプロシージャ指向システムの移植(3)

- パッケージ変数
  - セッション単位 + トランザクション制御外 の変数

どの方法でも
配列型や行型、
テーブル型を
(テキスト表現で)
格納可能。

実現方法	説明
関数 PKG1.CONSTVAL1() など	参照専用の定数ならこれで良い。
一時テーブル PKG2.set('VAL2', '100'::int) PKG2.get('VAL2')::int などラッパ関数を用意する。 初回に CREATE TEMP TABLE する。	• セッションが寿命となる点は一致 • 実行ステップ数が多くて低速 • 値の変更がロールバックしてしまう
GUCパラメータ current_setting('PKG3.VAL3', true) set_config('PKG3.VAL3', 'A', false)	• セッションが寿命となる点は一致 • 大文字小文字同一視も実現 • 動作するが本来的に大量データ用途でない • 値の変更がロールバックしてしまう
pg_variables 拡張モジュールを導入して、 pgv_set('pkg4', 'val4', 123) pgv_get('pkg4', 'val4', NULL::int)	• セッションが寿命となる点は一致 • パッケージ変数移植用途が意識されている • 値の変更がトランザクションと独立 • クラウドPostgreSQLサービスでは使えない

ストアドプロシージャ指向システムの移植(4)

- COMMIT/ROLLBACK
  - PostgreSQLは関数内で COMMIT/ROLLBACKできない
  - PG11 の PROCEDURE で対応したが制限も多い
- 自律型トランザクション
  - PostgreSQLに対応機能は無い
  - DBlink拡張で対応する
- 例外定義
  - PL/pgSQLではできない
  - エラーメッセージやエラー属性で識別させる

ストアドプロシージャ指向システムの移植(5)

- カーソル
  - PostgreSQLではカーソルの名前空間がグローバル
    - 呼び出し先関数と呼び出し元に同名カーソル名があると衝突する
  - カーソルでの行ロックの仕様差異
    - FOR UPDATE付きのカーソルが行をロックするタイミングは？
    - カーソルループ中に対象行更新をすることは安全ではない
- 配列／複合型
  - PL/pgSQL の表現力は高く、複雑なデータ型もOK
  - 配列利用時の仕様違いに注意

ストアドプロシージャ指向システムの移植(6)

《別の方針》

- プロシージャをアプリケーションコードに移植
  - プロシージャ to プロシージャ　をあきらめる
  - MVC の Modelオブジェクト内のメソッドにする

  - プロシージャに多段コール／依存関係があると面倒
  - 大量データ取得に注意
  - トリガはプロシージャ移植するしかない

まとめ

- ストアドプロシージャ指向にメリットあり
  - 単一のデータベースで完結するなら今でも有益
- 大規模開発にはそれなりの手順／道具立てを
- DBマイグレーションではプロシージャは鬼門だが、立ち向かえないほどではない

  《注意を要する箇所》
  - パッケージ
  - トランザクション
  - カーソル
  - ロギング／例外処理

オープンソースとともに

SRA OSS, INC.

URL：http://www.sraoss.co.jp/