Oracle Direct Seminar

ORACLE®

実践！！PL/SQLチューニング

日本オラクル株式会社

Oracle Direct

Oracle Direct Seminar

アジェンダ

- はじめに
- PL/SQLプログラムの計測
- PL/SQLコードのチューニングの検討
- パフォーマンスを意識したコーディング

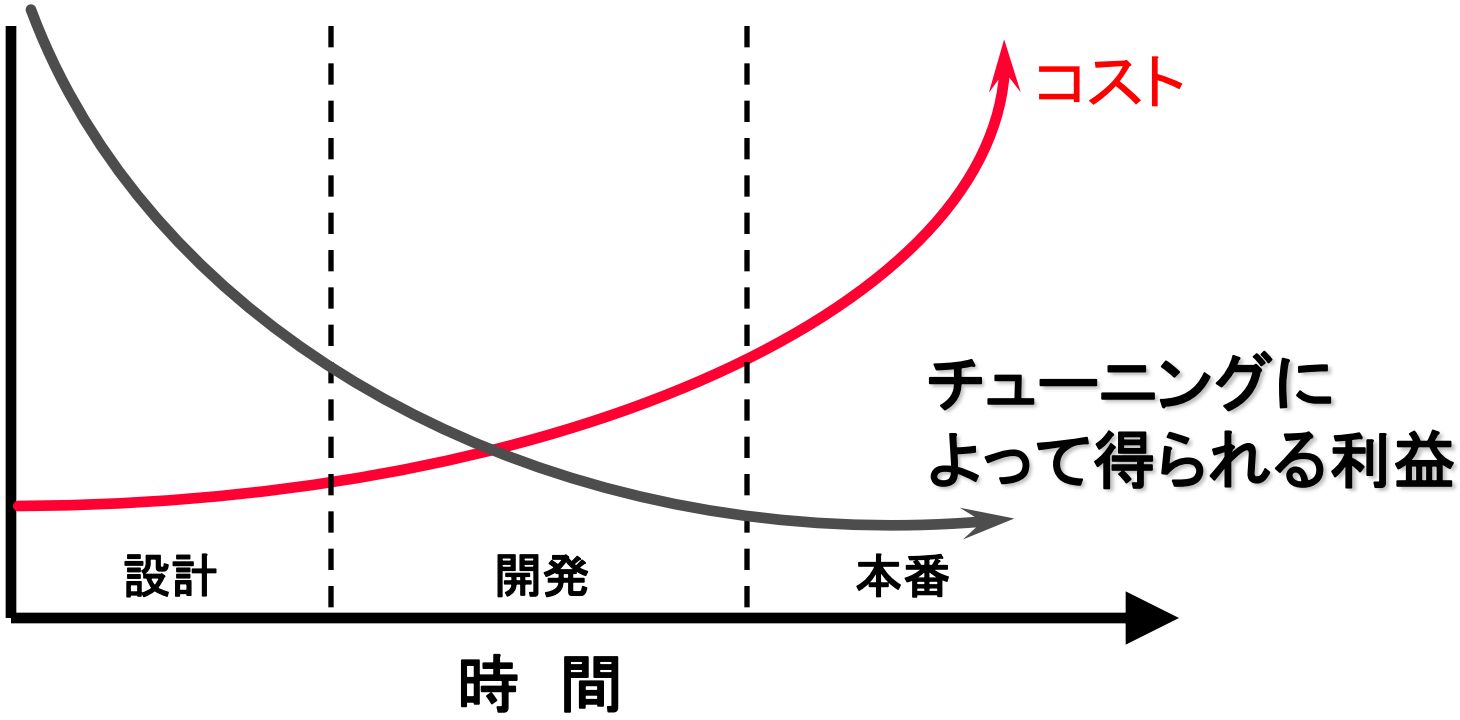
はじめに

- 前提
  - PL/SQLの基本的な知識がある方を対象としています
  - 原則として動作確認はOracle Database 11g Enterprise Edition 11.2.0.1にて実施しています
  - ※記述がなくても古いバージョンでは使えない可能性があります

- 動作検証環境
  - マシン：仮想マシン環境
  - OS: Oracle Enterprise Linux R5 update5（64bit）
  - インスタンスはDBCAで全てデフォルトで作成、追加チューニングー切なし

チューニングにかかるコストと利益

- アプリケーションの設計から本番稼動までの間の
  チューニングにかかるコストと利益



PL/SQLのチューニングとは？

- PL/SQLコードの内部で実行されるSQLの最適化
- アプリケーションとしての「無駄」を省く
  - 「まとめて」実行する機能を利用する
    - 複数の処理を1回の処理で
    - 繰り返し処理を1回の処理で
  - SQLで可能なものはSQLで処理

- PL/SQL高速化機能の採用検討
  - PL/SQLオプティマイザ（10g～）

アジェンダ

- はじめに
- PL/SQLプログラムの計測
  - DBMS_UTILITIY.GET_TIME（Oracle 7～）
  - DBMS_PROFILER（8.1.5 ～）
  - PL/SQL階層型プロファイラ（11.1.0.6～）
- PL/SQLコードのチューニングの検討
- パフォーマンスを意識したコーディング

PL/SQLプログラムの計測

- プログラム単体の処理時間計測
  - DBMS_UTILITY.GET_TIME ファンクション
- プログラムのボトルネックの識別（プロファイラの利用）
  - DBMS_PROFILER （8.1.5～）
  - PL/SQL階層型プロファイラ（11.1.0.6～）
    - SQL Developerなどの開発ツールで利用

PL/SQLプログラムの計測

DBMS_UTILITY.GET_TIMEファンクション

- 経過時間を確認する為に利用します（単位：1/100秒）

```
DECLARE
  t_begin NUMBER;
  t_end   NUMBER;
  t_diff  NUMBER;
BEGIN
  t_begin := DBMS_UTILITY.GET_TIME;
  DBMS_LOCK.SLEEP(3); -- 計測したい処理
  t_end := DBMS_UTILITY.GET_TIME;
  t_diff := t_end - t_begin;
  DBMS_OUTPUT.PUT_LINE('execute time(1/100 sec):'||
    TO_CHAR((t_diff), '9999999.99'));
  DBMS_OUTPUT.PUT_LINE('execute time: '||
    TO_CHAR(TO_DATE(TRUNC(t_diff/100,0),'SSSSS'),'HH24:MI:SS.')||
      TO_CHAR(MOD(t_diff,100),'FM00'));
END;
```

PL/SQLプログラムの計測

DBMS_PROFILER（8.1.5～）（その1）

- 事前準備
  - PL/SQLプログラムを実行するユーザにてDBMS_PROFILERが利用するテーブルを作成

```
SQL> show user
ユーザーは"HR"です。
SQL> @?/rdbms/admin/proftab.sql
```

- 作成されるオブジェクト
  - 表：plsql_profiler_data　プロファイリング・データの格納
  - 表：plsql_profiler_units　プロファイリング対象ユニットに関する情報を格納
  - 表：plsql_profiler_runs　プロファイリングの実行に関する情報を格納
  - 順序：plsql_profiler_runnumber

PL/SQLプログラムの計測

DBMS_PROFILER（8.1.5～）（その2）

- 利用方法
  1. 調査対象のPL/SQLプログラムの前後にて、プロファイリングの開始と終了のプロシージャを実行

```
SQL> execute dbms_profiler.start_prifiler('test 2');
SQL> execute emp_max_salary_slow; -- プロファイリング対象
SQL > execute dbms_profiler.stop_profiler;
```

コメントを付加

  2. プロファイル情報を表示する為のキー、runid を調べる

```
SQL> SELECT runid,run_date,run_comment FROM plsql_profiler_runs;
RUNID    RUN_DATE       RUN_COMMENT
------   ----------------   ------------------------------
1        10-11-02           test 1
2        10-11-02           test 2
```

PL/SQLプログラムの計測

DBMS_PROFILER（8.1.5～）（その3）

- 利用方法 (続き)
  3. 調査対象のPL/SQLプログラムのプロファイル情報の表示

```
SELECT p.unit_name, p.occured, p.tot_time, p.line# line,
          substr(s.text, 1,75) text
 FROM (SELECT u.unit_name, d.TOTAL_OCCUR occured,
                  (d.TOTAL_TIME/1000000000) tot_time, d.line#
            FROM plsql_profiler_units u,
                  plsql_profiler_data d
          WHERE d.RUNID =u.runid
              AND d.unit_number = u.unit_number
              AND d.TOTAL_OCCUR >0
              AND u.runid = 2
          ) p, user_source s
WHERE p.unit_name = s.name(+)  AND  p.line# = s.line (+)
ORDER BY p.unit_name, p.line# ;
```

ナノ秒単位を
秒単位に変換

前頁2で
調べたrunid

PL/SQLプログラムの計測

DBMS_PROFILER（8.1.5～）（その4）

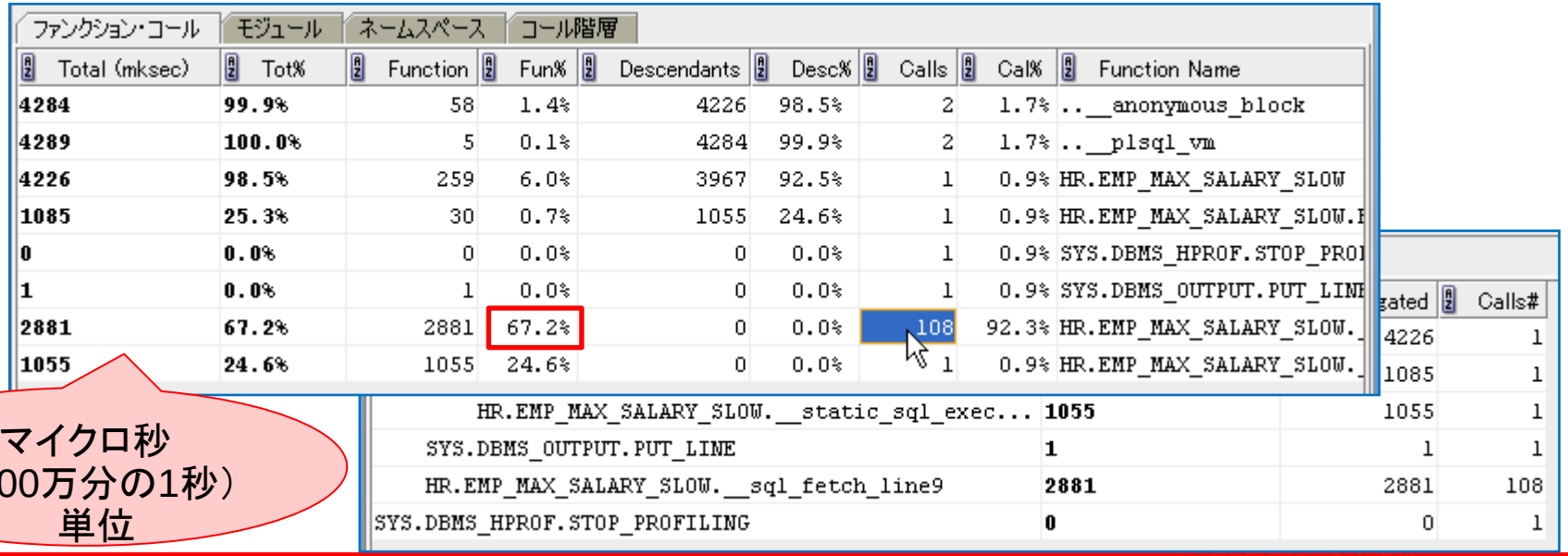
プロファイル表示例

```
UNIT_NAME              OCCURED  TOT_TIME     LINE  TEXT
---------------------- -------- ------------ ----- ------------------
<anonymous>            1        0            1
<anonymous>            3        0.000466839  1
<anonymous>            2        0.000040985  1
EMP_MAX_SALARY_SLOW    1        0.000002998  1     PROCEDURE emp_max_salary_slow
EMP_MAX_SALARY_SLOW    1        0            3       sal     NUMBER := 0;
EMP_MAX_SALARY_SLOW    1        0            4       max_sal NUMBER := 0;
EMP_MAX_SALARY_SLOW    2        0.000854705  5       CURSOR c1 IS SELECT salary FROM employees;
EMP_MAX_SALARY_SLOW    1        0            6     BEGIN
EMP_MAX_SALARY_SLOW    1        0            7       OPEN c1;
EMP_MAX_SALARY_SLOW    108      0            8       LOOP
EMP_MAX_SALARY_SLOW    108      0.001196587  9         FETCH c1 INTO sal;
EMP_MAX_SALARY_SLOW    108      0.000044984  10        EXIT WHEN c1%NOTFOUND;
EMP_MAX_SALARY_SLOW    107      0.000023991  12        IF sal > max_sal THEN
EMP_MAX_SALARY_SLOW    4        0            13          max_sal := sal;
EMP_MAX_SALARY_SLOW    4        0.000017993  14        END IF;
EMP_MAX_SALARY_SLOW    1        0            16      END LOOP;
EMP_MAX_SALARY_SLOW    1        0.000091968  17      CLOSE c1;
EMP_MAX_SALARY_SLOW    1        0.000052981  18      DBMS_OUTPUT.PUT_LINE(TO_CHAR(max_sal));
EMP_MAX_SALARY_SLOW    1        0.000000999  26    END;
```

PL/SQLプログラムの計測

PL/SQL階層型プロファイラ（11.1.0.6～）

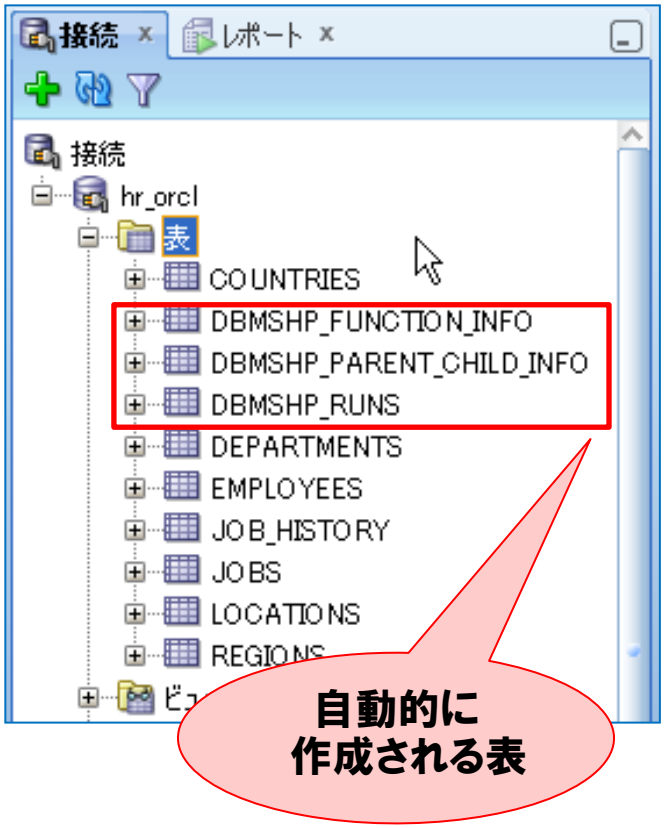
- サブプログラム・レベルの実行サマリー情報の提供
  - サブプログラムに対するコールの数
  - サブプログラム自体で費やされた時間（関数時間または自己時間）
  - サブプログラム自体およびその子サブプログラムで費やされた時間
- SQL Developerにより簡単に利用可能

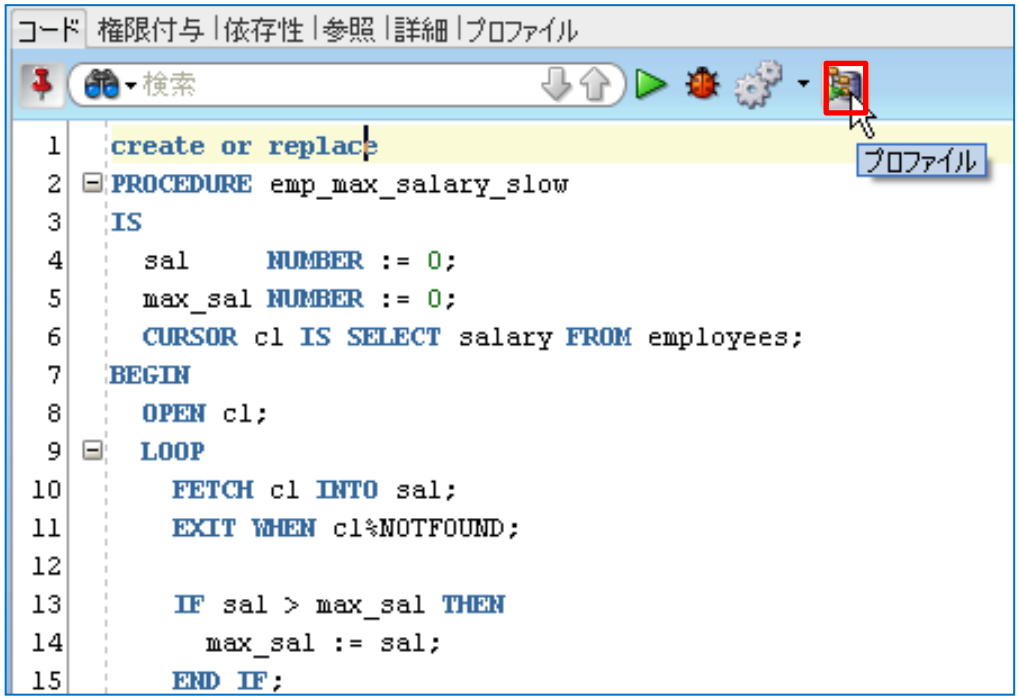


PL/SQLプログラムの計測

PL/SQL階層型プロファイラ（11.1.0.6～）

- ボタンをクリックするだけの操作
  - 階層型プロライラが必要なオブジェクトは自動で作成されます





アジェンダ

- はじめに
- PL/SQLプログラムの計測
- PL/SQLコードのチューニングの検討
  - PL/SQLコード中のSQLが遅い
  - PL/SQLで作り込んでいるファンクションが遅い
  - 内部でSELECTをおこなうファンクションの多用
  - PL/SQLで記述したロジックのSQLへの移管
- パフォーマンスを意識したコーディング

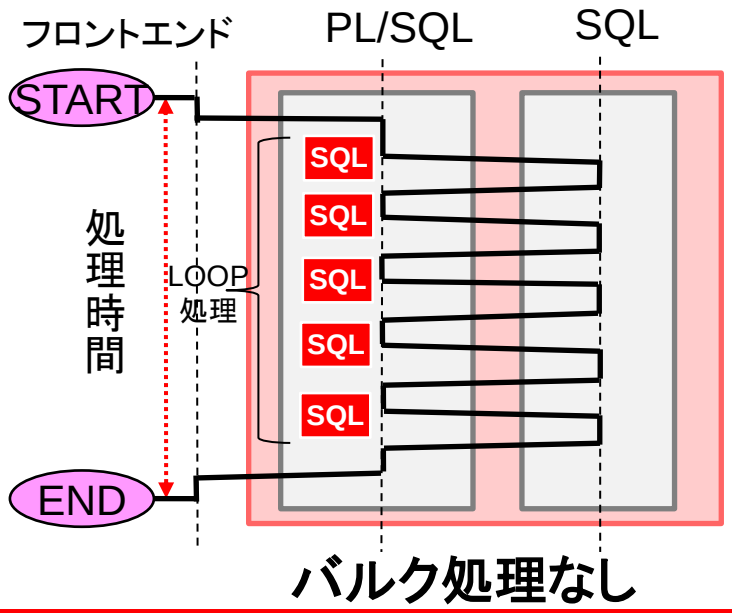
PL/SQLコード中のSQLが遅い

- SQLそのものが遅い → SQLチューニング実施
  - 適切な索引を利用していることを確認
  - オプティマイザ統計情報が取得されていることの確認
    - DBMS_STATS
  - SQLチューニング・アドバイザの利用
    - パーティショニング、マテリアライズド・ビューの適用検討
- SQLの繰り返しで時間がかかる
  - PL/SQLループの中でSQLを実行する場合はバルク処理を検討
  - DML処理の後にSELECTを実施している場合はRETURNINGを検討

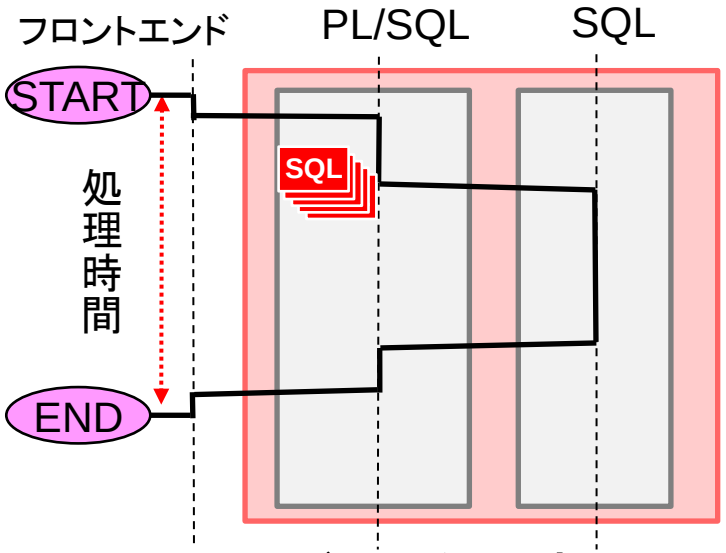
PL/SQLコード中のSQLが遅い

バルク処理（8i～）

- データ（主にレコード）をまとめて処理する機能
- PL/SQLエンジンとSQLエンジンの切り替えを減らすことでパフォーマンスを向上させる
  - LOOP内でSQLを実行している場合はバルク処理を検討する



バルク処理なし



バルク処理あり

PL/SQLコード中のSQLが遅い

バルク処理（構文）

- FORALL（INSERT/UPDATE/DELETE で利用）

```
FOR i IN depts.FIRST .. depts.LAST LOOP
  DELETE FROM emp
    WHERE dptno = depts(i);
END LOOP;
```

バルク処理なし

```
FORALL i IN depts.FIRST .. depts.LAST
  DELETE FROM emp
    WHERE dptno = depts(i);
```

バルク処理あり

- BULK COLLECT（SELECT/FETCH で利用）

```
OPEN c1;
LOOP
  FETCH c1 INTO emp_rec ;
  EXIT WHEN c1%NOTFOUND;
END LOOP;
CLOSE c1;
```

バルク処理なし

```
OPEN c1;
LOOP
  FETCH c1 BULK COLLECT INTO
    emp_rec_tbl LIMIT 200;
  EXIT WHEN c1%NOTFOUND;
END LOOP;
CLOSE c1;
```

バルク処理あり

PL/SQLコード中のSQLが遅い

RETURNING句

- SQLの発行回数を減らすことができる機能
- 更新系DML文の操作対象行のうち、指定されたカラムの内容を返します

```
INSERT INTO … VALUES (…) RETURNING  COL1  INTO  :COL1;
UPDATE … SET … RETURNING  COL1  INTO  :COL1;
DELETE … RETURNING  COL1  INTO  :COL1;
```

- 返すデータは複数カラム指定可能です

```
…… RETURNING  COL1,  COL2  INTO  :COL1,  :COL2;
```

- 返すデータとしてコレクションも指定できます

```
…… RETURNING  COL1  INTO  :COL1_ARRAY;
```

PL/SQLで作り込んでいるファンクションが遅い

- 文字列操作や演算などの処理をPL/SQLで作り込んでいる

```
SELECT FUNC1(COL1) FROM TBL1 WHERE COL2 = FUNC1(COL1);
```

対処案1）ファンクションのチューニング

- Oracle Databaseの提供している文字列関数などを極力利用
  - 低レベル(マシン語に近い）コードを利用しているため高速
  - REGEXP_SUBSTRなど、正規表現用ファンクションも提供（10g～）

対処案2）整数演算、浮動小数点演算に適したデータ型の利用

- PLS_INTEGER、SIMPLE_INTEGER
- BINARY_FLOAT、BINARY_DOUBLE、SIMPLE_FLOAT、SIMPLE_DOUBLE

対処案3）**ネイティブ・コンパイル**

対処案4）ファンクション索引の利用

- INSERTやUPDATE時に索引メンテナンスによる性能劣化の可能性も要検討

PL/SQLで作り込んでいるファンクションが遅い

ネイティブ・コンパイル

- PL/SQLのサブプログラムを、プロセッサ固有のネイティブコードにコンパイル、実行することができます

～10gR2

- Cコンパイラを利用してコンパイル（要：Cコンパイラ設定）
  共有ライブラリの形で利用（WindowsはDLL、UNIX/Linuxは *.so）
- 指定したOS上のディレクトリに共有ライブラリを格納（要：ディレクトリ指定）

11g～

- コンパイラ不要
- SYSTEM表領域上に配置
  RAC環境でもバイナリを共有

- PL/SQLサブプログラムを作成する際の、初期化パラメータ PLSQL_CODE_TYPE の設定により、ネイティブコンパイルするかどうかが決まります

```
SQL> alter session set plsql_code_type = 'NATIVE';
SQL> create or replace procedure …;
```

内部でSELECTをおこなうファンクションの多用

- ファンクション内部でSELECTを実施しており、データ量によってはパフォーマンスが悪くなることがあります

```
SELECT … FROM  tbl1  WHERE  col1 > get_price(item_id, sysdate-60) ;
```

```
CREATE FUNCTION get_price  (item_id_in IN NUMBER, dt_in IN DATE)   RETURN NUMBER
IS
  ret_num NUMBER;
BEGIN
  SELECT price  INTO ret_num FROM price_list
   WHERE item_id = item_id_in  AND from_dt <= TRUNC(dt_in)  AND to_dt  >   TRUNC(dt_in) ;
    RETURN ret_num;
END get_price;
```

対処案1）ファンクションの利用をやめ、結合処理に作り直す

対処案2）**PL/SQLファンクションの結果キャッシュ**を利用する

内部でSELECTをおこなうファンクションの多用

PL/SQLファンクションの結果キャッシュ（11g R1～、Enterprise Edition）

- PL/SQLファンクションの結果をSGAにキャッシュし、複数のセッションで利用できます
  - ファンクションおよびパラメータの値を組にして結果をキャッシュ
  - システムで必要なメモリが足りなくなると古いものから破棄
  - ファンクション内で参照している表が変更されるとキャッシュは破棄

```
CREATE OR REPLACE FUNCTION get_price(item_id_in IN NUMBER, dt_in IN DATE)
RETURN NUMBER  RESULT_CACHE   RELIES_ON ( price_list )
IS
  ret_num NUMBER;
BEGIN
  SELECT price  INTO ret_num FROM price_list
   WHERE item_id   = item_id_in
        AND from_dt <= TRUNC(dt_in)
        AND to_dt      >   TRUNC(dt_in) ;
    RETURN ret_num;
END get_price;
```

11g R2**からは**
RELIES_ON
記述は不要

PL/SQLで記述したロジックのSQLへの移管

- 取得データを元にした変換処理
  - SQLにてDECODE、CASEにて対応できないか検討する

```
SELECT  col1  INTO  xxx  FROM  TBL1  WHERE …;
IF xxx = 'A' THEN
  yyy := 'FOO';
ELSE
  yyy := 'BAR';
END IF;
```

```
SELECT  DECODE(col1,'A','FOO','BAR')  INTO yyy  FROM  TBL1  WHERE …;
```

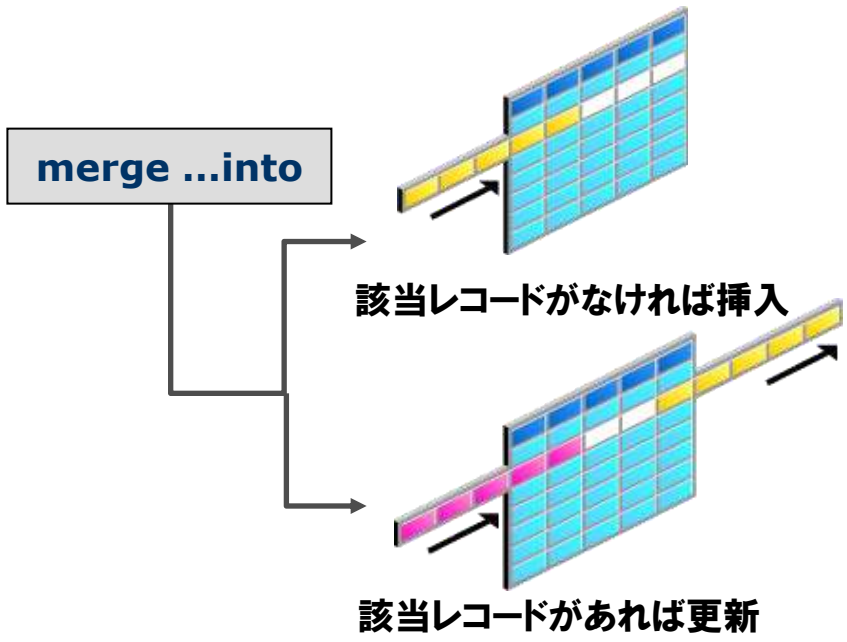
- PL/SQLロジックそのものをSQLで実現する
  - **MERGE文**
  - **DMLエラーロギングとの組み合わせ**

PL/SQLで記述したロジックのSQLへの移管

MERGE文（9i R1～）

- MERGE文を使用すると、1つ以上のソースから行を選択し、表またはビューに対して更新および挿入できます

```
MERGE INTO bonuses D
  USING (
    SELECT employee_id, salary, department_id
      FROM employees
    WHERE department_id = 80
  ) S
  ON (D.employee_id = S.employee_id)
  WHEN MATCHED THEN
    UPDATE SET D.bonus = D.bonus + S.salary*.01
    DELETE WHERE (S.salary > 8000)
  WHEN NOT MATCHED THEN
    INSERT (D.employee_id, D.bonus)
   VALUES (S.employee_id, S.salary*.01)
    WHERE (S.salary <= 8000);
```



PL/SQLで記述したロジックのSQLへの移管

DMLエラー・ロギング（10g R2～）

- INSERT、UPDATE、MERGE、DELETE文で利用可能
  - これまでは大量の行を対象とした単一のDMLにエラーが発生すると処理のすべてがロールバックされていました
  - 上記DMLに「ERROR LOGS」句をつけることで利用します

```
INSERT INTO employees (empno, ename, dptno, sal)
 ( SELECT empno, ename, dptno, sal FROM employees_wk1 )
 LOG ERRORS  INTO  ERR$_EMPLOYEES ('WEEKLY_BATCH') REJECT  LIMIT  50;
```

  - エラー・ロギング表は DBMS_ERRLOG パッケージで作成します
  - 実行DMLとエラーロギング表への書き込みトランザクションは分離
  - DML操作に失敗したデータを記録する為のものであり、すべてのエラーを書き込むものではありません
    - 例）ORA-01653(領域不足)、ORA-01555

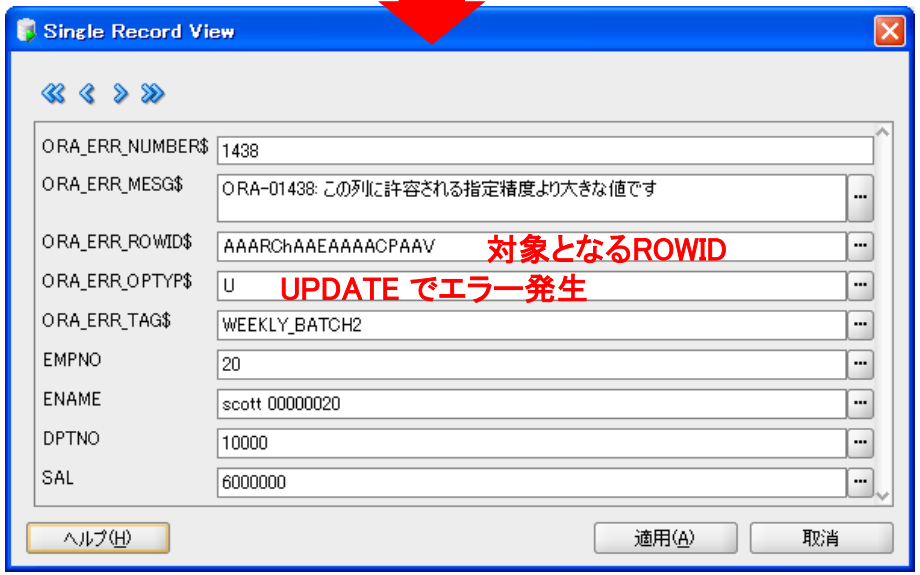
ご参考）エラーロギング表の例

```
INSERT INTO employees (empno, ename, dptno, sal)
 (  SELECT empno, ename, dptno*1000, sal
      FROM  employees_old
     WHERE  empno <= 1)
LOG ERRORS INTO ERR$_EMPLOYEES
 ('WEEKLY_BATCH') REJECT LIMIT 50;
```

故意に桁あふれ

```
UPDATE employees
      SET dptno = dptno * 1000
 WHERE empno = 20
LOG ERRORS INTO ERR$_EMPLOYEES
('WEEKLY_BATCH2') REJECT LIMIT 50;
```

故意に桁あふれ

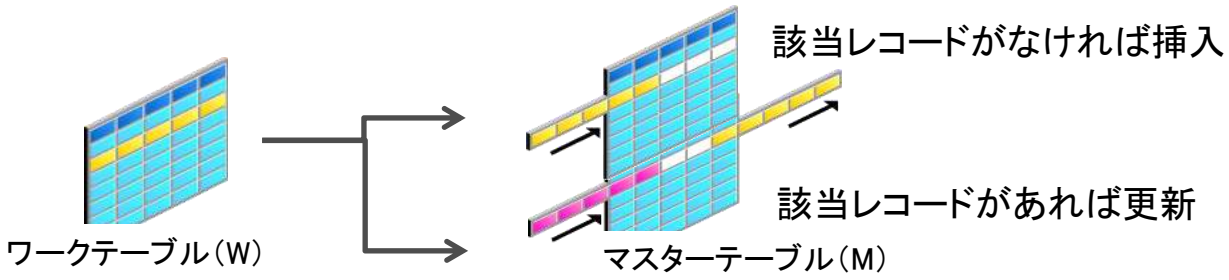


アジェンダ

- はじめに
- PL/SQLプログラムの計測
- PL/SQLコードのチューニングの検討
- パフォーマンスを意識したコーディング

アプリケーション例での考察（仕様）

- ワークテーブル（W）の内容をマスターテーブル（M）に反映させる例：
  - キー項目は同じ
  - MにWの内容が存在しないデータ（行）はINSERT処理
  - MにWの内容が存在するデータ（行）はUPDATE処理
  - エラーが発生した場合に、少なくとも以下の情報を取得
    - Oracleのエラー番号、キー項目
  - エラーが発生しなかったものはすべてMに反映（COMMIT）
    - 大量のエラー（データエラー、システムエラー）が発生した場合、処理を中断（ROLLBACK）



検証アプリケーションによる比較（前提）

- 検証データ
  - M（employees）テーブル：　　1,000,000件（平均行長：26byte）
  - W（employees_wk1）テーブル：200,000件（平均行長：26byte）
  - INSERT対象：100,000件（正常系データのみ）
  - UPDATE対象：100,000件（正常系データのみ）
- 比較時の留意事項
  - M（employees）テーブルはTRUNCATE、INSERTにて測定毎にデータ再作成
  - M（employees）テーブル再作成後、統計情報取得
  - 処理中のREDOログスイッチを避ける為に、事前にログスイッチ
  - バッファキャッシュにデータ、パッケージがのっていない状態にて実施

```
alter system flush shared_pool;
alter system flush buffer_cache;
```

アプリケーション例での考察（実装案）

ケース1：	Wの内容をCURSORで全件取得。LOOPにてMへの反映処理（INSERTを実施し、キー重複エラーが発生した場合にUPDATE実施）
ケース2：	1）キー項目の内容がMおよびWに存在するWのデータをCURSORにて取得。LOOPにてMへ反映（UPDATE）処理 2）キー項目の内容がMに存在しないWのデータをCURSORにて取得LOOPにてMへ反映（INSERT）処理
ケース3：	ケース2のバルク処理 ※ レコードを使用した挿入・更新機能を利用
ケース4：	1）キー項目の内容がMおよびWに存在するWのデータをそのままMへ反映（UPDATE）処理 2）キー項目の内容がMに存在しないWのデータをそのままMへ反映（INSERT）処理 ※ CURSORを利用しない DMLエラー・ロギング機能を利用
ケース5：	Wの内容をMへ反映（MERGE）処理 ※ DMLエラー・ロギング機能を利用

一般的な実装

コーディング時のポイント

- バルク処理（BULK COLLECT INTO..）ではLIMIT指定
  - 利用する結合配列の要素数を100～200程度にすることで
    メモリを無駄に使わないようにします

ケース3

- バルク処理（FORALL）ではSAVE EXCEPTIONS指定
  - エラーが発生した場合でもバルク処理を完了させ、その後
    エラー処理をまとめておこなうようにします

ケース3

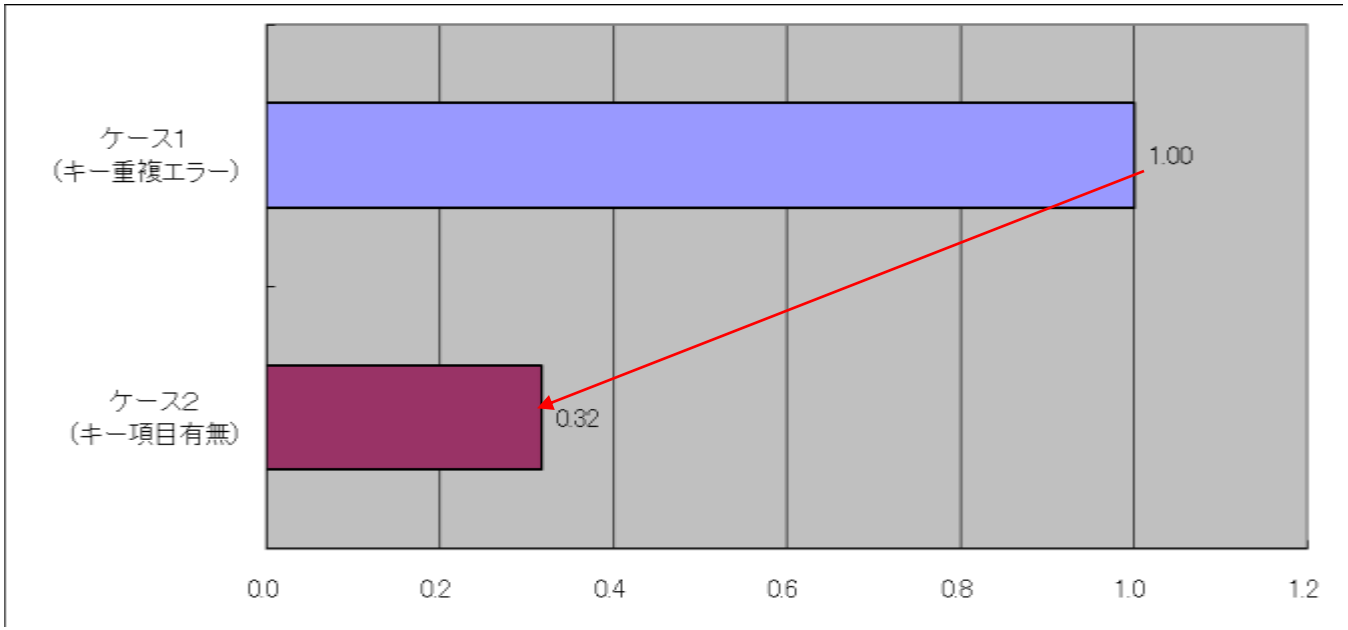
- 「DMLエラー・ロギング」の機能を使うことでINSERT、UPDATE、MERGE処理をPL/SQLのバッチ処理に組み込みやすくなりました
  - これまではDML操作に失敗したデータを明確にする為に、
    CURSOR＋LOOP処理が必須でした

ケース4

ケース5

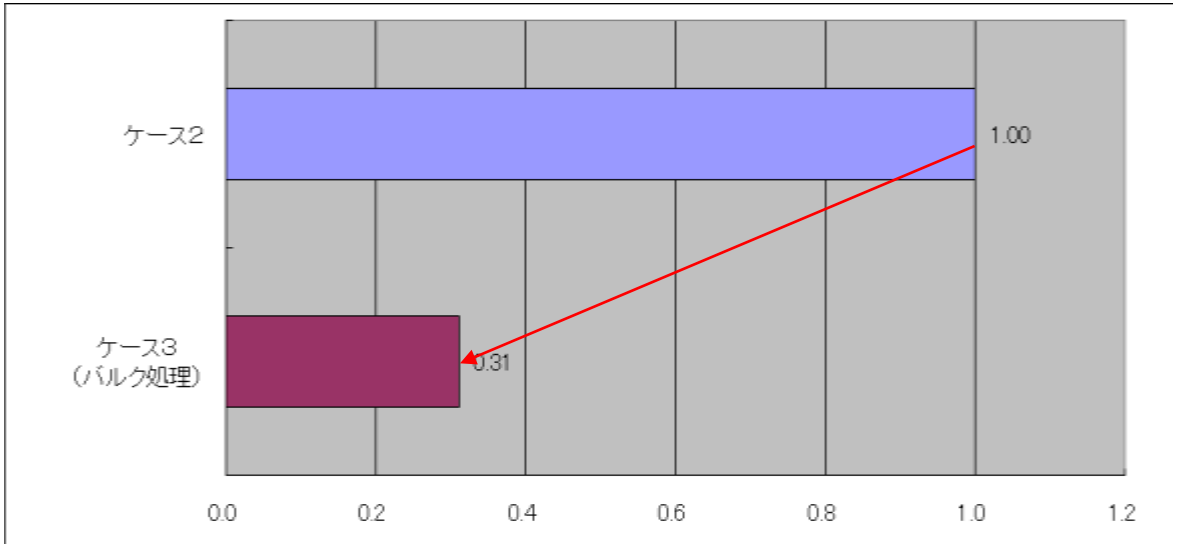
ケース1とケース2の性能比較

- ケース1は例外処理を多発させており非常に効率が悪い



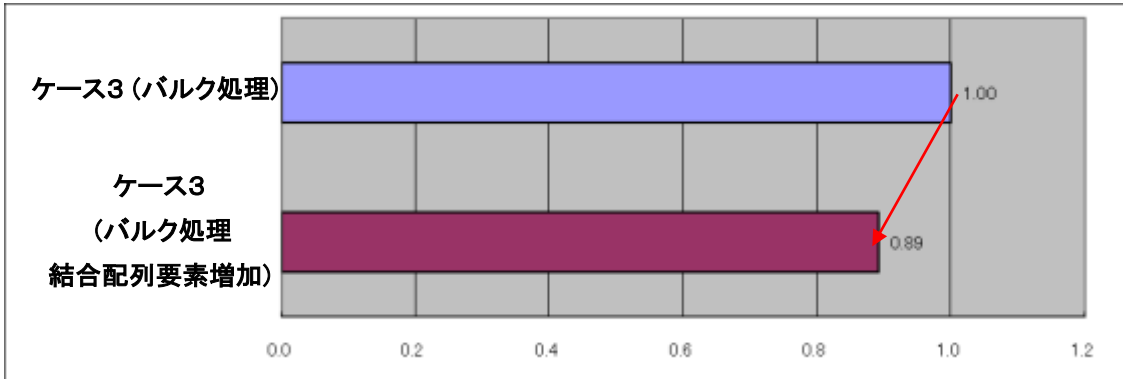
ケース2とケース3の性能比較

- ケース3はケース2のバルク処理対応
  - ケース2ではLOOP処理がUPDATEで100,000回、INSERTで100,000回実行されている
  - ケース3では一度に200件のバルク処理を実施。LOOP処理がUPDATEで500回、INSERTで500回に削減している



ケース3のバルク処理に関する考察

- バルク処理にて結合配列にて利用する要素数を変更し、性能差を比較
- ケース3では結合配列で利用する要素数を200にしたものと100,000（全件分）にしたものにて実行



- 実行直後のPGAメモリ量（session pga memory）を比較

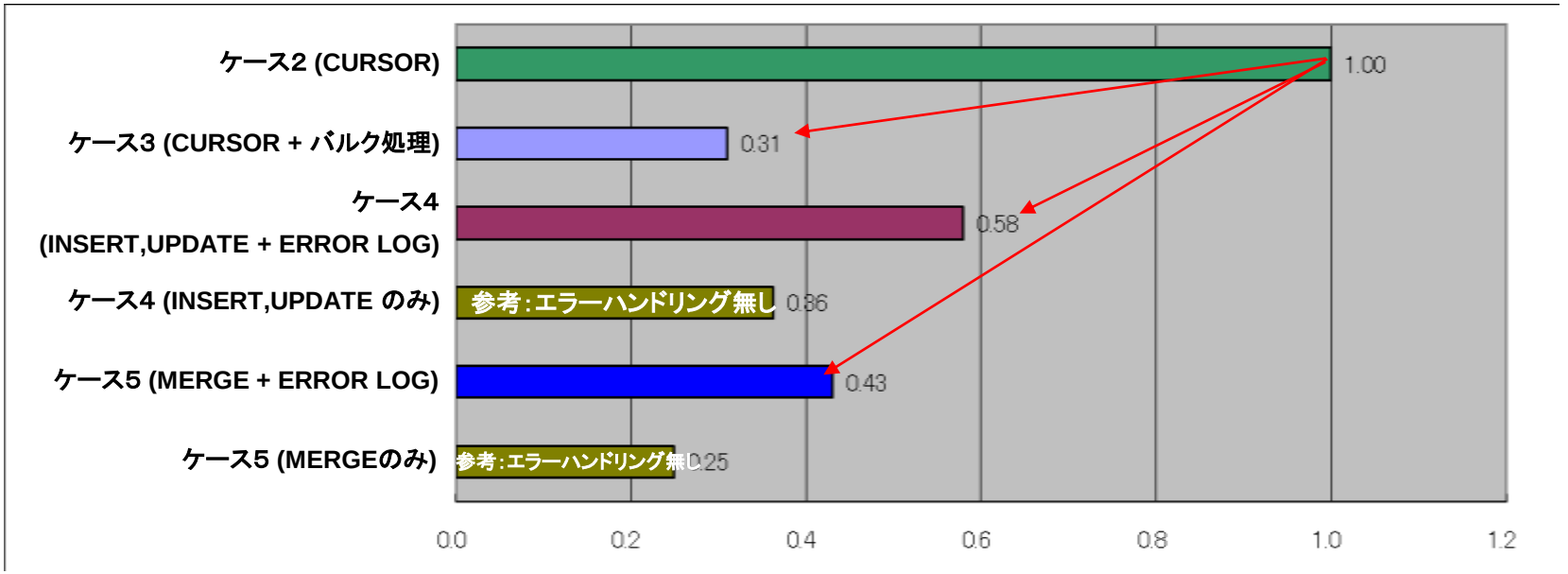
結合配列要素数（　　200）:	22,233,640(byte)
結合配列要素数（100,000）:	56,295,304(byte)

むやみに結合配列の要素数を増やしても、性能が極端に向上するわけではない

※ 上記のケースでは、メモリを2.5倍浪費しているにも関わらず、処理時間はさほど変わらない

ケース3、ケース4、ケース5の性能比較

- こちらのいずれかのケースにてコーディングすることになる
- ソースコードが簡潔になるのは、ケース4もしくはケース5
- DMLエラー・ロギング機能が動作する際の負荷も存在する



バルク処理(FORALL)

11g R1～

```
DECLARE
  TYPE tbl_emp_rec IS TABLE OF employees%ROWTYPE INDEX BY PLS_INTEGER;
  emp_rec_array tbl_emp_rec;
  CURSOR emp_rec_upd_cur IS SELECT * FROM employees_wk1;
BEGIN
  OPEN emp_rec_upd_cur;
  LOOP
    emp_rec_array.DELETE; /* 結合配列の初期化 */
    empno_array.DELETE;
    FETCH emp_rec_upd_cur BULK COLLECT INTO emp_rec_array  LIMIT 200;
      IF emp_rec_array.COUNT = 0 THEN
        EXIT;
      END IF;
      FORALL i IN emp_rec_array.FIRST .. emp_rec_array.LAST
          UPDATE employees
             SET ROW   = emp_rec_array(i)
           WHERE empno = emp_rec_array(1).empno;
    EXIT WHEN emp_rec_upd_cur%NOTFOUND;
  END LOOP;
  CLOSE emp_rec_upd_cur;
END;
```

一度に200件
取得

FORALL処理中に結合配列の要素を
参照可能になりました（11g～）

バルク処理(FORALL)

～10g R2

```
DECLARE
  TYPE tbl_emp_rec IS TABLE OF employees%ROWTYPE    INDEX BY PLS_INTEGER;
  TYPE tbl_empno   IS TABLE OF employees.EMPNO%TYPE INDEX BY PLS_INTEGER;
  emp_rec_array tbl_emp_rec;
  empno_array   tbl_empno;
  CURSOR emp_rec_upd_cur IS SELECT * FROM employees_wk1;
BEGIN
  OPEN emp_rec_upd_cur;
  LOOP
    emp_rec_array.DELETE; /* 結合配列の初期化 */
    empno_array.DELETE;
    FETCH emp_rec_upd_cur BULK COLLECT INTO emp_rec_array LIMIT 200;
      IF emp_rec_array.COUNT = 0 THEN
        EXIT;
      END IF;
      FOR i IN emp_rec_array.FIRST .. emp_rec_array.LAST LOOP
        empno_array(i) := emp_rec_array(i).empno;
      END LOOP;
      FORALL i IN emp_rec_array.FIRST .. emp_rec_array.LAST
          UPDATE employees
            SET ROW   = emp_rec_array(i)
          WHERE empno = empno_array(i);
    EXIT WHEN emp_rec_upd_cur%NOTFOUND;
  END LOOP;
  CLOSE emp_rec_upd_cur;
END;
```

一度に200件
取得

FORALL処理中に結合配列の要素を
参照できない為の対応

バルク処理(FORALL)のSAVE EXCEPTIONS

```
DECLARE
  TYPE tbl_exception_index   IS TABLE OF VARCHAR2(80) INDEX BY PLS_INTEGER;
  TYPE tbl_exception_errcode IS TABLE OF PLS_INTEGER  INDEX BY PLS_INTEGER;
  err_index_array tbl_exception_index;
  err_code_array tbl_exception_errcode;
  ins_errors PLS_INTEGER := 0;
  err_count  PLS_INTEGER := 0;
BEGIN
  OPEN ... ;
  LOOP
    FETCH ...;
    BEGIN
      FORALL i IN emp_rec_array.FIRST .. emp_rec_array.LAST SAVE EXCEPTIONS
          INSERT INTO employees VALUES emp_rec_array(i);
    EXCEPTION
      WHEN OTHERS THEN
        ins_errors := ins_errors + SQL%BULK_EXCEPTIONS.COUNT; -- error件数
        FOR i IN 1 .. SQL%BULK_EXCEPTIONS.COUNT LOOP
          err_count := err_count + 1; -- COUNT UP
          err_index_array(err_count) := SUBSTRB(SQL%BULK_EXCEPTIONS(i).ERROR_INDEX,1,80);
          err_code_array(err_count)  := SQL%BULK_EXCEPTIONS(i).ERROR_CODE;
        END LOOP;
    END;
    EXIT WHEN ... %NOTFOUND;
  END LOOP;
  CLOSE ...;
END;
```

例外が発生しても
まとめて処理

DMLエラー・ロギングの利用（MERGEの例）

```
DECLARE
  /* 事前に当該スキーマで実行： execute dbms_errlog.create_error_log('EMPLOYEES'); */
  ins_errors PLS_INTEGER := 0;
  upd_errors PLS_INTEGER := 0;
BEGIN
  MERGE INTO employees e
  USING employees_wk3 w
    ON (e.empno = w.empno)
   WHEN MATCHED THEN
     UPDATE
        SET e.ename = w.ename,
            e.dptno = w.dptno,
            e.sal   = w.sal
   WHEN NOT MATCHED THEN
     INSERT (e.empno, e.ename, e.dptno, e.sal)
     VALUES (w.empno, w.ename, w.dptno, w.sal)
   LOG ERRORS INTO ERR$_EMPLOYEES ('WEEKLY_BATCH') REJECT LIMIT 50;
  COMMIT;

  SELECT count(*) INTO ins_errors FROM err$_employees
   WHERE ora_err_tag$   = 'WEEKLY_BATCH' AND ora_err_optyp$ = 'I'; /* INSERT */
  SELECT count(*) INTO upd_errors FROM err$_employees
   WHERE ora_err_tag$   = 'WEEKLY_BATCH' AND ora_err_optyp$ = 'U'; /* UPDATE */
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END;
```

50件以上エラーがあると全体をROLLBACK。
エラーの情報はERR$_EMPLOYEES表に
登録される。

アプリ内部でエラー
状況を把握できる

まとめ

- PL/SQLコードのパフォーマンス問題を可視化するための仕組みを提供しています
  - SQL Developer等のツールより簡単に利用できるようになりました
- 問題別の対処案ではPL/SQLで提供している機能を活用できます
  - バルク処理、RETURINIG句
  - PL/SQLネイティブコンパイル
  - PL/SQLファンクションの結果キャッシュ
- PL/SQLのみではなくSQLを含めたアプリケーション全体の見直しが効果をあげることが多い
  - バルク処理、DMLエラーロギング機能を利用した処理を比較してみました

OTNセミナー　オンデマンド　コンテンツ

ダイセミで実施された技術コンテンツを動画で配信中!!

ダイセミのライブ感はそのままに、お好きな時間で受講頂けます。

twitter

最新情報つぶやき中

oracletechnetjp

- 人気コンテンツは？
- お勧め情報
- 公開予告　など

OTN　オンデマンド　検索

※掲載のコンテンツ内容は予告なく変更になる可能性があります。
期間限定での配信コンテンツも含まれております。お早めにダウンロード頂くことをお勧めいたします。

Oracle エンジニアのための技術情報サイト

オラクルエンジニア通信

http://blogs.oracle.com/oracle4engineer/

twitter

最新情報つぶやき中

oracletechnetjp

- 技術資料
  - ダイセミの過去資料や製品ホワイトペーパー、スキルアップ資料などを多様な方法で検索できます
  - キーワード検索、レベル別、カテゴリ別、製品・機能別
- コラム
  - オラクル製品に関する技術コラムを毎週お届けします
  - 決してニッチではなく、誰もが明日から使える技術の「あ、そうだったんだ！」をお届けします

こんな資料が人気です

- ✓ 6ヶ月連続で「RAC/ASMインストール資料」が第一位。
  根強い人気のチュートリアル系コンテンツですが、
  レプリケーション解説資料が上位に挙がってきました。
- ✓ .NetやWindowsサーバーならではの機能　を集めた特集ページも好評です。

オラクルエンジニア通信

ITプロジェクト全般に渡る無償支援サービス

Oracle Direct Conciergeサービス

■パフォーマンス診断サービス
- Webシステム　ボトルネック診断サービス NEW
- データベースパフォーマンス 診断サービス

■移行支援サービス
- SQL Serverからの移行支援サービス
- DB2からの移行支援サービス
- Sybaseからの移行支援サービス
- MySQLからの移行支援サービス
- Postgre SQLからの移行支援サービス
- Accessからの移行支援サービス
- Oracle Application ServerからWeblogicへ移行支援サービス NEW

■システム構成診断サービス
- Oracle Database構成相談サービス
- サーバー統合支援サービス
- 仮想化アセスメントサービス
- メインフレーム資産活用相談サービス
- BI EEアセスメントサービス
- 簡易業務診断サービス

■バージョンアップ支援サービス
- Oracle Databaseバージョンアップ支援サービス
- Weblogic Serverバージョンアップ支援サービス NEW
- Oracle Developer/2000(Froms/Reports) Webアップグレード相談サービス

オラクル社のエンジニアが 直接ご支援します
お気軽にご活用ください!

オラクル　無償支援　検索

ORACLE

あなたにいちばん近いオラクル

Oracle Direct

まずはお問合せください

Oracle Direct　検索

システムの検討・構築から運用まで、ITプロジェクト全般の相談窓口としてご支援いたします。

システム構成やライセンス/購入方法などお気軽にお問い合わせ下さい。

Web問い合わせフォーム

専用お問い合わせフォームにてご相談内容を承ります。
http://www.oracle.co.jp/inq_pl/INQUIRY/quest?rid=28

※フォームの入力には、Oracle Direct Seminar申込時と同じ
　ログインが必要となります。
※こちらから詳細確認のお電話を差し上げる場合がありますので、ご登録されている連絡先が最新のものになっているか、ご確認下さい。

フリーダイヤル

0120－155－096

※月曜~金曜　9:00~12:00、13:00~18:00
（祝日および年末年始除く）

ORACLE