High Performance PL/SQL

Bulk Processing, Function Result Cache and More

Steven Feuerstein
Oracle Developer Advocate for PL/SQL
Oracle Corporation

Email: steven.feuerstein@oracle.com
Twitter: @sfonplsql
Blog: stevenfeuersteinonplsql.blogspot.com
YouTube: Practically Perfect PL/SQL

ORACLE

Deepen Your PL/SQL and SQL Expertise

- Take advantage of our community websites.
- Oracle AskTOM – https://asktom.oracle.com
  - Q&A site, Office Hours with database experts, and much more
- Oracle Dev Gym – https://devgym.oracle.com
  - Quizzes, workouts and classes for an active learning experience
- Oracle LiveSQL – https://livesql.oracle.com
  - 24x7 access to the latest release of Oracle Database, plus a script library and tutorials

Key Performance Features

- Bulk processing with FORALL and BULK COLLECT
- Function Result Cache
- Improved performance of PL/SQL functions from SQL
- NOCOPY
- Automatic optimization

What's the problem with this code?

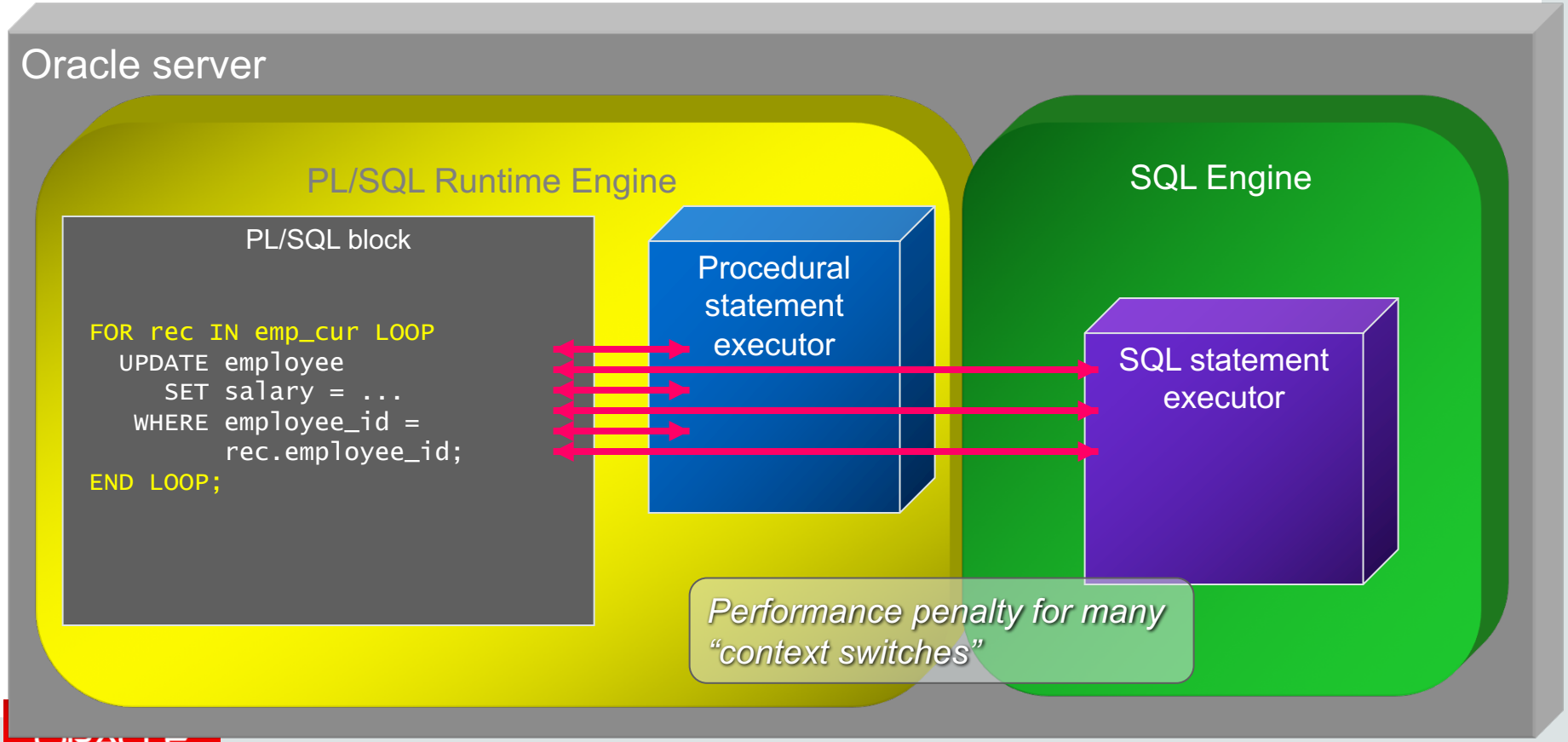
- We have, on average, 10,000 employees per department.

```
CREATE OR REPLACE PROCEDURE upd_for_dept (
   dept_in IN employees.department_id%TYPE
  ,newsal_in IN employees.salary%TYPE)
IS
   CURSOR emp_cur IS
      SELECT employee_id,salary,hire_date
        FROM employees WHERE department_id = dept_in;
BEGIN
   FOR rec IN emp_cur
   LOOP
      adjust_compensation (rec, newsal_in);

      UPDATE employee SET salary = rec.salary
       WHERE employee_id = rec.employee_id;
   END LOOP;
END upd_for_dept;
```

Row-by-row = Slow-by-slow?

- Many PL/SQL blocks execute the same SQL statement repeatedly, with different bind values.
  - Retrieves data one row at a time.
  - Performs same DML operation for each row retrieved.
- The SQL engine does a lot to optimize performance, but row-by-row processing is inherently slow.
  - But, but...aren't SQL and PL/SQL supposed to be very tightly integrated? Let's take a look "under the covers.

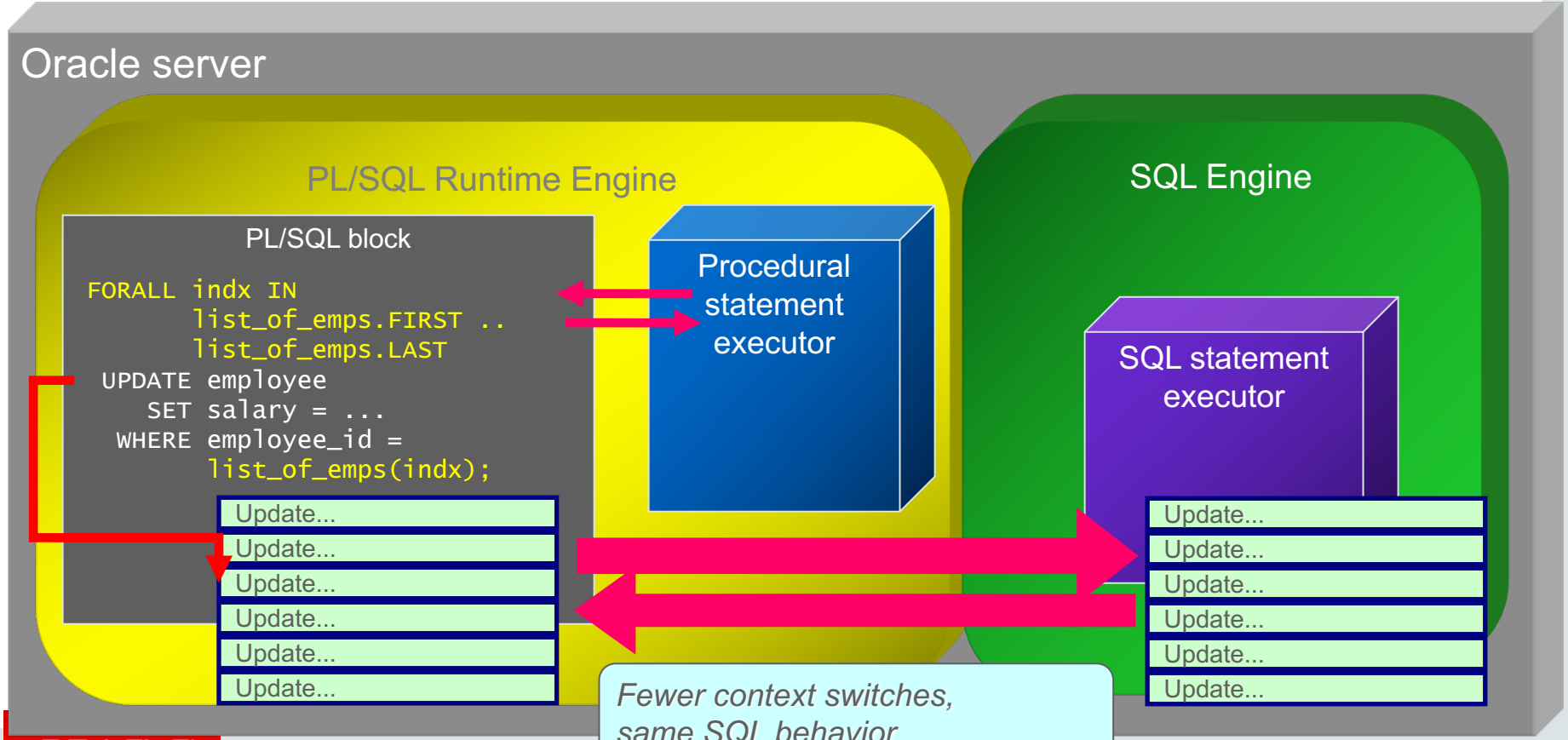
Why Row-by-Row Goes Slow (Relatively)



Bulk Processing in PL/SQL

- The goal is straightforward: reduce the number of context switches and you improve performance.
- To do this, Oracle "bundles up" the requests for data (or to change data) and then passes them with a single context switch.
- FORALL speeds up non-query DML.
  - Use with inserts, updates, deletes and merges.
  - Move data from collections to tables.
- BULK COLLECT speeds up queries.
  - Can be used with all kinds of queries: implicit, explicit, static and dynamic.
  - Move data from tables into collections.

Bulk processing with FORALL



Impact of Bulk Processing in SQL layer

- The bulk processing features of PL/SQL change the way the PL/SQL engine communicates with the SQL layer.
- For both FORALL and BULK COLLECT, the processing in the SQL engine is almost completely unchanged.
  - Same transaction and rollback segment management
  - Same number of individual SQL statements will be executed.
- Only one difference: BEFORE and AFTER statement-level triggers only fire *once* per FORALL INSERT statements.
  - Not for each INSERT statement passed to the SQL engine from the FORALL statement.

statement_trigger_and_forall.sql

BULK COLLECT for multi-row querying

```
SELECT * BULK COLLECT INTO collection(s) FROM table;

FETCH cur BULK COLLECT INTO collection(s) [ LIMIT number_of_rows ];

EXECUTE IMMEDIATE query BULK COLLECT INTO collection(s);
```

- Retrieve multiple rows into a collection with a single fetch (context switch to the SQL engine).
  - Deposit the multiple rows of data into one or more *collections*.
- NO_DATA_FOUND is *not* raised when no rows are fetched; instead, the collection is empty.
- The "INTO" collections are filled sequentially from index value 1.
  - There are no "gaps" between 1 and the index value returned by the COUNT method.
- Only integer-indexed collections may be used.
- No need to initialize or extend nested tables and varrays. Done automatically by Oracle.

BULK COLLECT with Implicit Cursor

- An "unlimited" bulk collect operation that populates the collection with all rows found. Which can lead to PGA memory errors!

Declare a nested table of records to hold the queried data.

Fetch all rows into collection sequentially, starting with 1.

Iterate through the collection contents with a loop.

```
DECLARE
   TYPE employees_aat IS TABLE OF
            employees%ROWTYPE;

   l_employees employees_aat;
BEGIN
   SELECT *
   BULK COLLECT INTO l_employees
     FROM employees;

   FOR indx IN 1 .. l_employees.COUNT
   LOOP
      process_employee (l_employees(indx));
   END LOOP;
END;
```

bulkcoll.sql
bulkcollect.tst

Limit rows returned by BULK COLLECT

```
CREATE OR REPLACE PROCEDURE bulk_with_limit
   (deptno_in IN dept.deptno%TYPE)
IS
   CURSOR emps_in_dept_cur IS
      SELECT * FROM emp
       WHERE deptno = deptno_in;

   TYPE emp_tt IS TABLE OF emps_in_dept_cur%ROWTYPE;
   emps emp_tt;
BEGIN
   OPEN emps_in_dept_cur;
   LOOP
      FETCH emps_in_dept_cur
         BULK COLLECT INTO emps LIMIT 1000;

      EXIT WHEN emps.COUNT = 0;

      process_emps (emps);
   END LOOP;
   CLOSE emps_in_dept_cur;
END bulk_with_limit;
```

Use the LIMIT clause with the INTO to manage the amount of memory used with the BULK COLLECT operation.

Definitely the preferred approach in production applications with large or varying datasets.

bulklimit.sql

Details on that LIMIT clause

- The limit value can be a literal or a variable.
  - I suggest passing the limit as a *parameter* to give you maximum flexibility.
- A limit of 100 seems like a good default value.
  - Setting it to 500 or 1000 doesn't seem to make much difference in performance.
- With very large volumes of data and small numbers of batch processes, however, a larger LIMIT could help.

When to convert to BULK COLLECT

- Prior to Oracle10g, you should convert *all* multiple row fetch code to BULK COLLECTs.
- On 10.1 and higher, the optimizer will automatically optimize cursor FOR loops to run at performance levels similar to BULK COLLECT.
- So leave your cursor for loops in place if they...
  - contain *no* DML operations.
  - seem to be running fast enough.
- Explicit BULK COLLECTs may run *a little* faster than cursor for loops optimized to return 100 rows with each fetch.

10g_optimize_cfl.sql

BULK COLLECT Conclusions

- BULK COLLECT improves performance of queries that retrieve more than one row.
- Use the LIMIT clause to avoid excessive PGA memory consumption.
- Leave it to the optimizer to speed up "read only" cursor FOR loops.

FORALL

- Introduction to FORALL
- Using the SQL%BULK_ROWCOUNT
- Referencing fields of collections of records
- Using FORALL with sparsely-filled collections
- Handling errors raised during execution of FORALL

Use FORALL for repeated DML operations

```
PROCEDURE upd_for_dept (...)  IS
BEGIN
   FORALL indx IN low_value .. high_value
      UPDATE employee
         SET salary = newsal_in
       WHERE employee_id = list_of_emps (indx);
END;
```

- Convert loops that contain inserts, updates, deletes or merges to FORALL statements.
- Header (usually) looks identical to a numeric FOR loop.
  - Implicitly declared integer iterator
  - At least one bind array that uses the iterator as its index value.
  - Use INDICES OF and VALUES OF when bind arrays are sparse

forall_timing.sql
forall_examples.sql

More on FORALL

- Use any type of collection with FORALL.
- Only *one DML statement* is allowed per FORALL.
  - Each FORALL is its *own "extended"* DML statement.
- The collection must be indexed by integer.
- The bind array must be sequentially filled.
  - Unless you use the INDICES OF or VALUES OF clause.
- Indexes cannot be expressions.

forall_restrictions.sql

How many rows were modified?

- SQL%ROWCOUNT returns total number of rows modified by entire FORALL.
  - Not to be relied on when used with LOG ERRORS.
- Use the SQL%BULK_ROWCOUNT cursor attribute to determine how many rows are modified by each statement.
  - A "pseudo-collection" of integers; no methods are defined for this element.

bulk_rowcount.sql

Using FORALL with Sparse Collections

- Prior to 10.2, the binding arrays in a FORALL statement *must* be sequentially filled.
  - Using the IN low .. high syntax.
- Now, however, you can bind sparse collections by using INDICES OF and VALUES OF in the FORALL header.

```
PROCEDURE upd_for_dept (...)  IS
BEGIN
   FORALL indx IN INDICES OF list_of_emps
      UPDATE employee
         SET salary = newsal_in
       WHERE employee_id = list_of_emps (indx);
```

10g_indices_of*.sql
10g_values_of*.sql

FORALL and DML Errors

- FORALLs typically execute a large number of DML statements.
- When an exception occurs in one of those DML statement, the default behavior is:
  - *That* statement is rolled back and the FORALL stops.
  - All (previous) successful statements are *not* rolled back.
- What if you want the FORALL processing to continue, even if an error occurs in one of the statements?
- Just add the SAVE EXCEPTIONS clause!

Example: FORALL with SAVE EXCEPTIONS

- Add SAVE EXCEPTIONS to enable FORALL to suppress errors at the *statement* level.

```
CREATE OR REPLACE PROCEDURE load_books (books_in IN book_obj_list_t)
IS
  bulk_errors EXCEPTION;
  PRAGMA EXCEPTION_INIT ( bulk_errors, -24381 );
BEGIN
  FORALL indx IN books_in.FIRST .. books_in.LAST
    SAVE EXCEPTIONS
    INSERT INTO book values (books_in(indx));
EXCEPTION
   WHEN bulk_errors THEN
      FOR indx in 1..SQL%BULK_EXCEPTIONS.COUNT
      LOOP
         log_error (SQL%BULK_EXCEPTIONS(indx).ERROR_INDEX
                  , SQL%BULK_EXCEPTIONS(indx).ERROR_CODE);
      END LOOP;
END;
```

If any exception is encountered, Oracle raises -24381 when done.

Allows processing of all statements, even after an error occurs.

Iterate through "pseudo-collection" of errors.

bulkexc.sql

SAVE EXCEPTIONS and FORALL

```
PROCEDURE upd_for_dept (newsal_in IN NUMBER,
   list_of_emps_in IN DBMS_SQL.NUMBER_TABLE) IS
BEGIN
   FORALL indx IN list_of_emps_in.FIRST .. list_of_emps_in.LAST
      SAVE EXCEPTIONS
      UPDATE employees
         SET salary = newsal_in
       WHERE employee_id = list_of_emps_in (indx);
END;
```

- The SAVE EXCEPTIONS clause tells Oracle to *save* exception information and continue processing all of the DML statements.
- When the FORALL statement completes, if at least one exception occurred, Oracle then raises ORA-24381.
- You then check the contents of SQL%BULK_EXCEPTIONS.

SAVE EXCEPTIONS in Detail

- For each exception raised, Oracle populates the SQL%BULK_EXCEPTIONS *pseudo-collection* of records.
  - The record has two fields : ERROR_INDEX and ERROR_CODE.
  - ERROR_INDEX: the index in the bind array for which the error occurred.
  - ERROR_CODE: the number (positive) for the error that was raised
- It's a *pseudo-collection* and only supports a single method: COUNT.
- So you iterate from 1 to SQL%BULK_EXCEPTIONS.COUNT to get information about each error.
- Unfortunately, it does not store the error *message.*

Converting to Bulk Processing

- Let's take a look at the process by which you go from "old-fashioned" code to a bulk processing-based solution.
- From integrated row-by-row to *phased* processing
- Challenges include:
  - With multiple DML statements in loop, how do you "communicate" from one to the other?
  - Avoid excessive PGA consumption

The "Old Fashioned" Approach

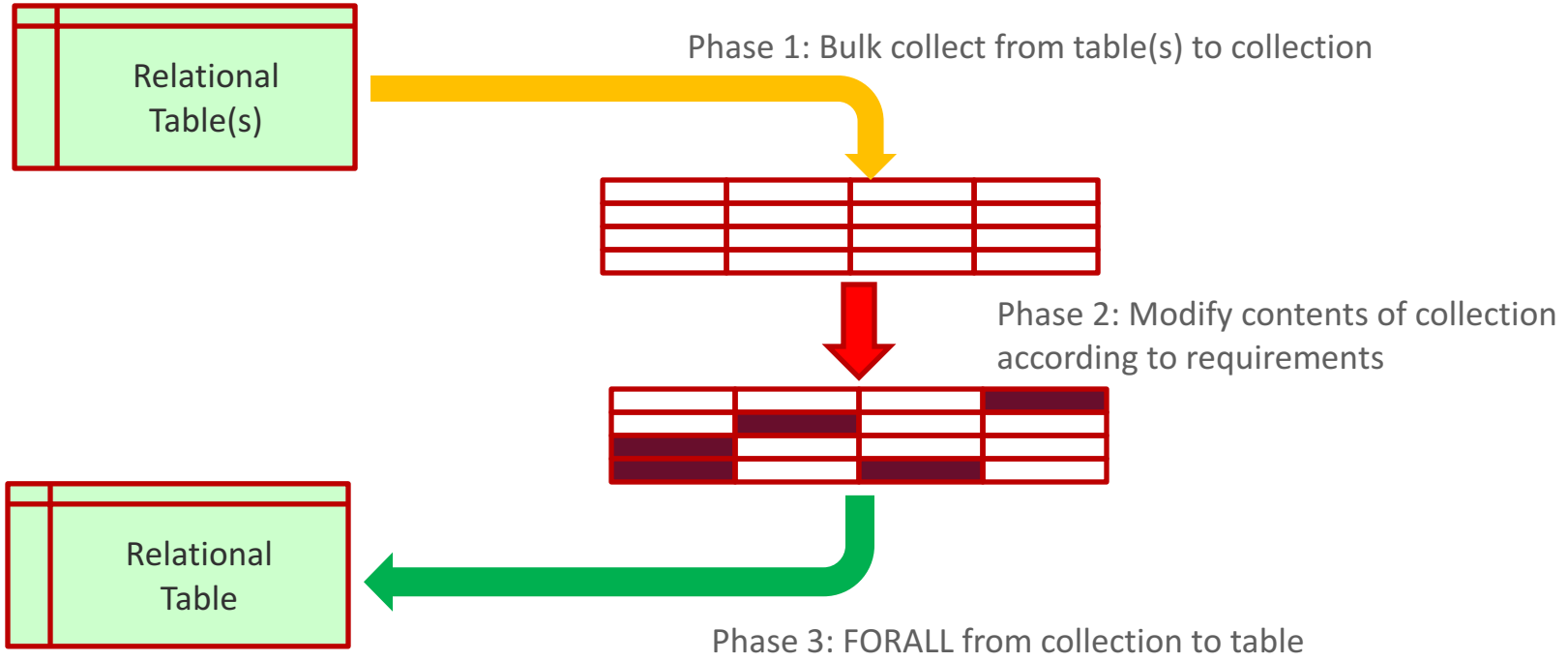
- Cursor FOR loop with two DML statements, trap exception, and keep on going.

```
CREATE OR REPLACE PROCEDURE upd_for_dept (
   dept_in     IN   employees.department_id%TYPE
 , newsal_in   IN   employees.salary%TYPE)
IS
   CURSOR emp_cur ...;
BEGIN
   FOR rec IN emp_cur
   LOOP
      BEGIN
         INSERT INTO employee_history ...

         adjust_compensation (rec.employee_id, rec.salary);

         UPDATE employees SET salary = rec.salary ...
      EXCEPTION
         WHEN OTHERS THEN log_error;
      END;
   END LOOP;
END upd_for_dept;
```

cfl_to_bulk_0.sql

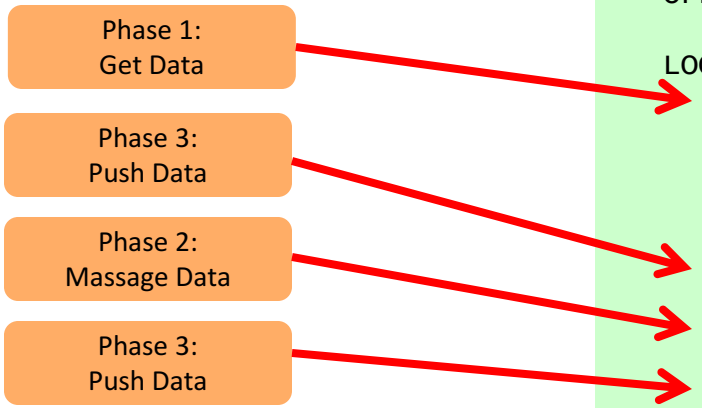
A phased approach with bulk processing

- Change from integrated, row-by-row approach to a *phased* approach.



Translating phases into code

- The cfl_to_bulk_5.sql file contains the converted program, following the phased approach.



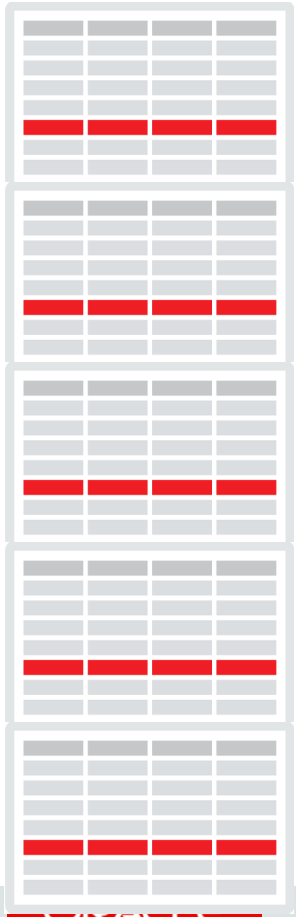

```
BEGIN
   OPEN employees_cur;

   LOOP
      fetch_next_set_of_rows (
           bulk_limit_in, employee_ids, salaries, hire_dates);

      EXIT WHEN employee_ids.COUNT = 0;

      insert_history;

      adj_comp_for_arrays (employee_ids, salaries);

      update_employee;
   END LOOP;
END;
```

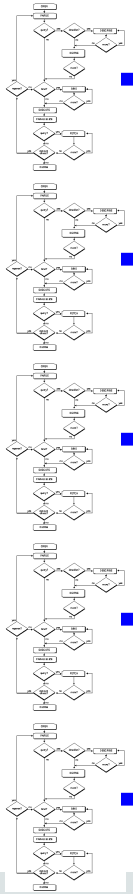
cfl_to_bulk_0.sql
cfl_to_bulk_5_11g.sql

Conclusions – Bulk Processing

- FORALL is the most important performance tuning feature in PL/SQL.
  - Almost always the fastest way to execute repeated SQL operations in PL/SQL.
  - Look that for prime anti-pattern: Loops containing non-query DML.
  - **Then ask yourself: can I do it entirely within SQL?** If not...bulk it up!
- You trade off increased complexity of code for dramatically faster execution.
  - But remember that Oracle will automatically optimize cursor FOR loops to BULK COLLECT efficiency.
  - No need to convert unless the loop contains DML or you want to maximally optimize your code.
- Watch out for the impact on PGA memory!

What's wrong with this picture?





```
SELECT * FROM materialized_view
WHERE pky = 12345;
```

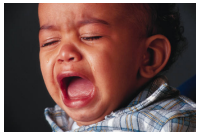
```
SELECT * FROM materialized_view
WHERE pky = 12345;
```

```
SELECT * FROM materialized_view
WHERE pky = 12345;
```

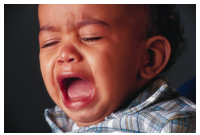
```
SELECT * FROM materialized_view
WHERE pky = 12345;
```

```
SELECT * FROM materialized_view
WHERE pky = 12345;
```

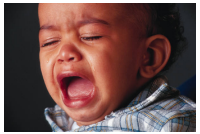
What's Taking So Long?



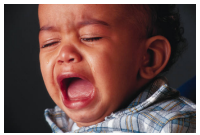
What's Taking So Long?



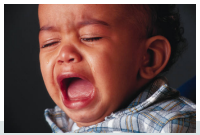
What's Taking So Long?



What's Taking So Long?



What's Taking So Long?



Here's the question to ask oneself:

If the data hasn't changed, why should my users have to query it again and again and again?

- But they need the data! And it needs to be the right data! What's a developer supposed to do?

The Wonderful Function Result Cache

- The Function Result Cache should be the caching mechanism of choice for PL/SQL developers.
  - Others: DETERMINISTIC, PGA (session-specific) caching
- This cache is stored in the SGA ; shared across sessions; purged of dirty data automatically
- You can and should use it to retrieve data from any table that is *queried more frequently than updated*.
  - Static datasets like materialized views (the "extreme case")
  - Same rows fetched multiple times (parameter values serve as unique index into cache)
- Enterprise Edition only.

How the Function Result Cache Works

- Add the RESULT_CACHE clause to your function's header.
  - Yes, that's the only change you have to make to your code!
- When a call is made to function, Oracle compares IN argument values to the cache.
- If no match, the function is executed and the inputs and return data are cached.
- If a match is found, the function is not executed; cached data is returned.
- If changes to a "relies on" table are committed, the cache is marked invalid and will be re-built.

11g_frc_demo.sql

Performance Impact of Result Cache

- The result cache is stored in the SGA.
- So we should expect it be slower than a PGA-based cache.
- But accessing result cache data does not require going through the SQL engine.
- So it should be much faster than executing a query.
  - Even if the statement is parsed and the data blocks are already in the SGA.
- Let's find out!

11g_emplu*.*

Result Cache – Things to Keep in Mind - 1

- If you have uncommitted changes in your session, dependent caches are ignored.
  - The cache will *not* override your own changed data.
- Caching is not performed for complex types: records with CLOBs, collections, etc.
  - But Oracle is optimistic!
- The cache is *not* related to SQL statements in your function.
  - It only keeps track of the input values and the RETURN clause data.

11g_frc_demo.sql

Result Cache – Things to Keep in Mind - 2

- You cannot use the result cache with invoker rights program units until 12.1.
  - Bypass execution of function body, Oracle cannot resolve references to objects - the *whole point* of IR.
- Functions with session-specific dependencies must be "result-cached" with great care.
  - Virtual private database configurations
  - References to SYSDATE, reliance on NLS_DATE_FORMAT, time zone changes
  - Application contexts (calls to SYS_CONTEXT)
- Solution: move all dependencies into parameter list.

11g_frc_vpd.sql
11g_frc_vpd2.sql

Managing the Result Cache

- Oracle offers a number of ways to manage the result cache and tune it to your specific application needs:
- RESULT_CACHE_MAX_SIZE initialization parameter
  - If the cache is too small, then the LRU algorithm negates the point of the cache.
- DBMS_RESULT_CACHE management package
- v$RESULT_CACHE_* performance views
- Warning: use of the function result cache can result in latch contention that can cause database performance *degradation.*

show_frc_dependencies.sp

Fine Grained Dependencies in 11.2

- Oracle keeps track of table dependencies on a per-result level.
  - Each result cached could have a different set of dependencies.
- A change to a table could invalidate just a subset of the results in the cache.
  - It's *not* all or nothing - when your function's different logic paths could "hit" different tables.

11g_frc_dependencies.sql
11g_frc_dependencies2.sql

Make It Easy on Yourself

- I hope you will agree that the result cache is a great feature.
- But how easy will it be for you to apply it?
- If you write/duplicate queries throughout your code, upgrading will be expensive and slow.
- If you hide your queries behind functions, you have a single point of definition, and you can upgrade "instantly."

11g_frc_encapsulation.sql

The NOCOPY Parameter Hint

- By default, Oracle passes all OUT and IN OUT arguments by *value*, not reference.
  - This means that OUT and IN OUT arguments always involve some *copying* of data.
  - All IN arguments are always passed by reference (no copying).
- With NOCOPY, you turn off the copy process.
  - For "large" formal parameters, such as collections, you may see a performance benefit.
- Compile-time warnings will notify you of opportunities to use this hint.

nocopy*.*

Optimizing Function Execution in SQL

- That seems like an awfully good idea!
- Two methods:
  - WITH clause that *defines* a function
  - UDF pragma
- WITH FUNCTION: define a function directly within your SQL statement.
  - Most helpful for avoiding redundancy in long SQL statements
- UDF pragma declaratively tells the compiler to "prepare" the function for execution from SQL.
  - Reduces the cost of the context switch

12c_with_function*.sql
12c_udf*.sql

Automatic Optimization – Set to Level 3

- The default optimization level is 2.
  - It's been 2 since 10.1, and it's a fine optimization level.
- But 11.1 introduced automatic subprogram inlining – level 3
  - Works with nested subprograms only!
- Recommendation from PL/SQL dev team: always move optimization level up to 3.
  - Only downside you are likely to see is increased compiled code size. That should not be an issue.
  - You can also use INLINE pragma for selective inlining.

11g_inline*.sql

PL/SQL Optimization

- Make sure you integrate the "big ticket" performance features as you develop.
  - Native SQL, bulk processing, FORALL, NOCOPY, UDF, opt level 3...
- Focus primarily on correctness and maintainability.
- Then identify bottlenecks.
  - DBMS_HPROF, DBMS_PROFILER
- Then perform more granular optimization within identified subprograms.


ORACLE®