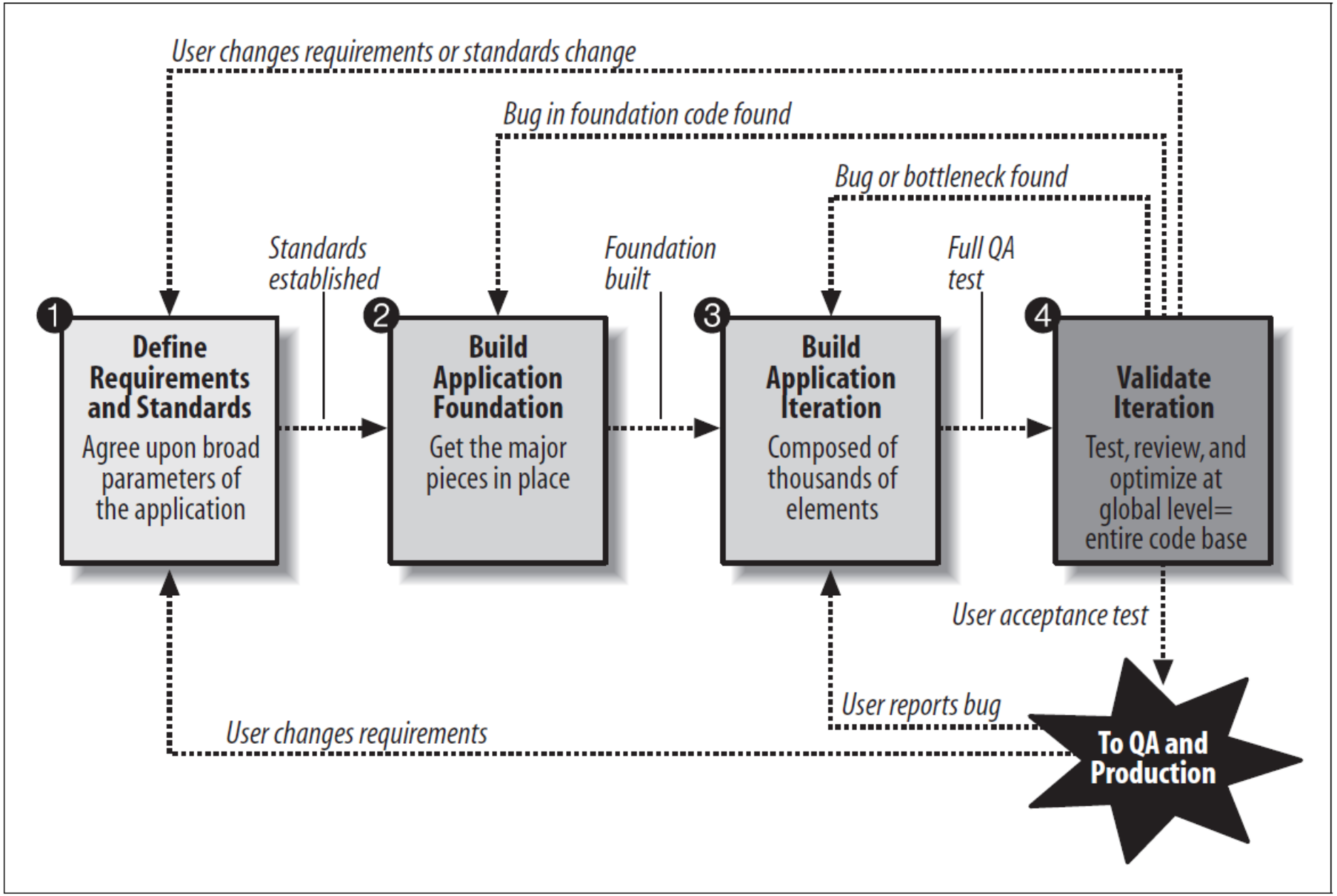
EJADA EAI

Database Center of Excellence

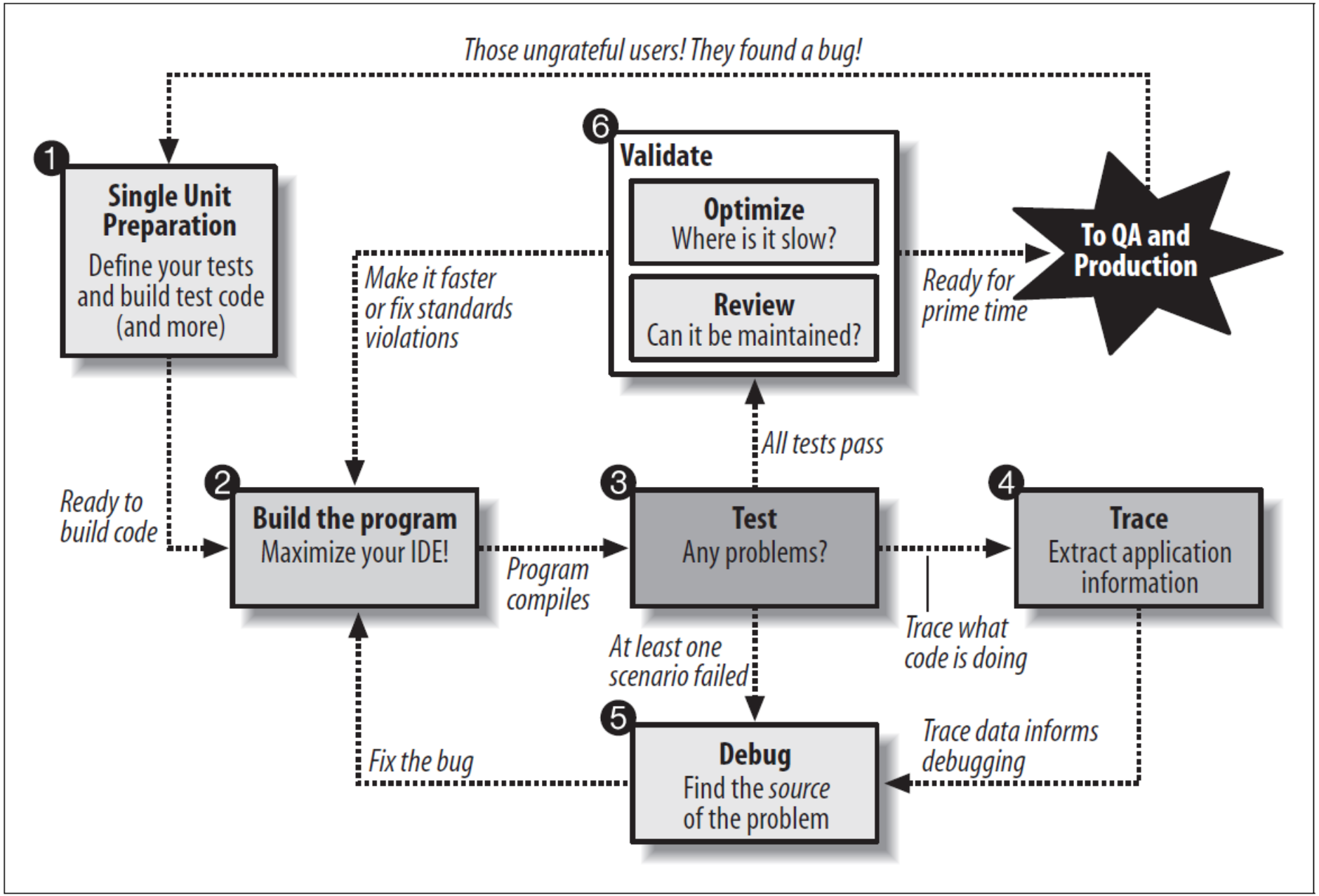
Agenda

- Database Design and Development Process
  - DB Application Lifecycle
  - Single Program Lifecycle
  - Don't Jump to Code (Program Preparation)
  - Exception handling
  - Testing, Tracing, and Debugging
  - Code Generation
- DB Team Standards and Best Practices
  - Naming Conventions (standards)
  - PL/SQL good news
  - PL/SQL Best Practices (selected topics)
- Continuous Improvement

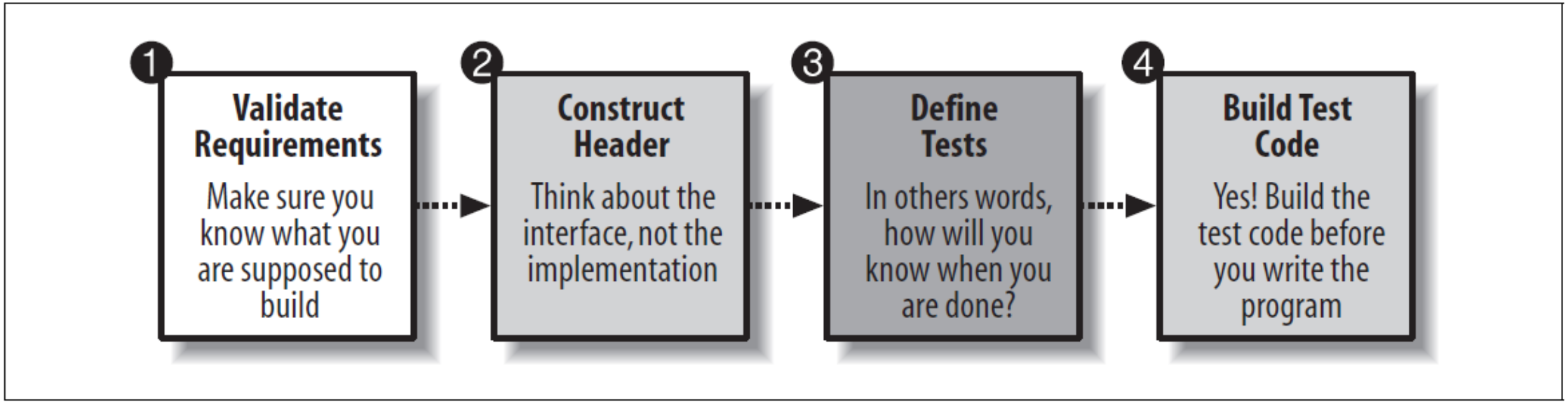
Recommended workflow for the entire application



Workflow for each program you build



Four Steps for preparing an application



DB Program Contract

- Functional Requirements
  - Description
  - Parameters
    - IN/OUT/INOUT
    - Names
    - Data Types*
    - Allowable Values and their meaning
    - By reference or by value
  - Return Value
    - Type
    - Allowable values and their meaning
  - Exceptions and Error Codes

- Non-Functional Requirements
  - Expected Call Rate
  - Expected Data Volume
  - Synch/Asynch
  - Interactive (online) / Batch
  - Required response time
  - Parallel Execution and Concurrency

* For data types use <TABLE_NAME>.<COLUMN_NAME>%TYPE or SUBTYPES rather than hard-coded lengths

Database Design Considerations

- Involve the DBA from design phase (physical DB Design)
  - Indexes (B+ tree, bitmap, …)
  - Partitioning
  - Clustering and Index-organized tables
  - Column-organized tables
  - …

Version Control

- Liquibase
  - Open-source extensible Tool for DB Version Control
  - Migration-based (not state-based)
  - Initially for Schema and Lookup Data (REF Data)
  - Can automate stored procedure VC by customization (future)
- TFS
  - GIT repo in TFS for DB objects
  - Includes stored procedure logic and schema/REF data updates
  - Updates in SP, Schema, REF Data must be linked to tasks in TFS

Testing

DB Program Test Cases

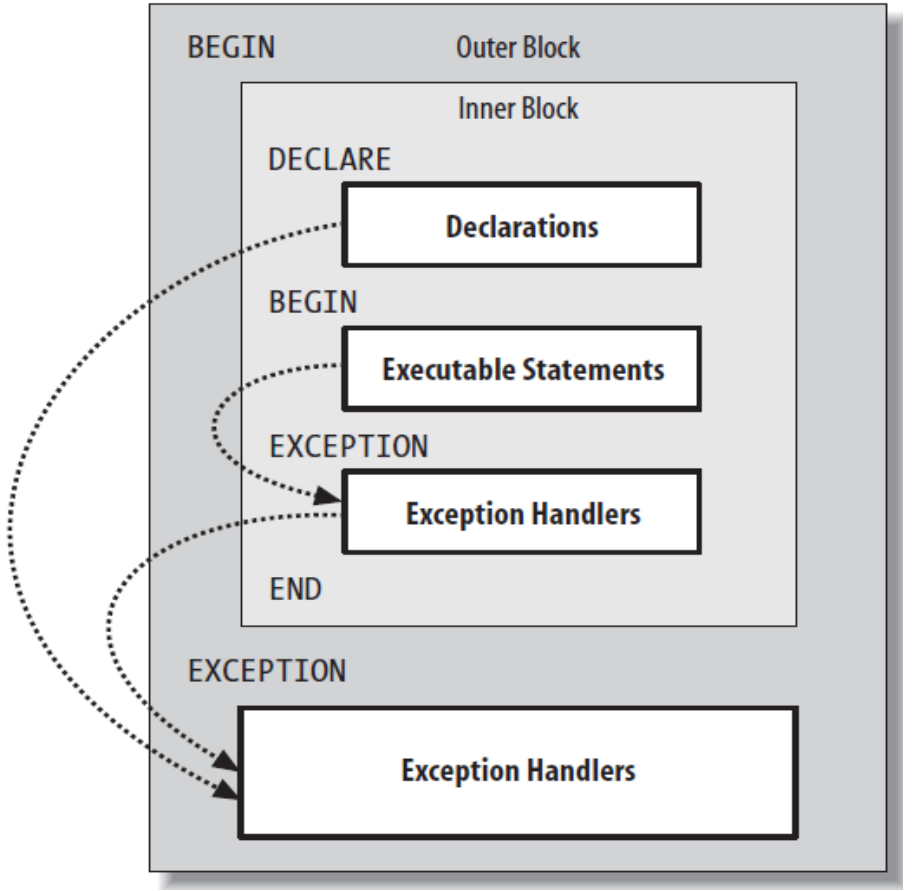
- Test for success (happy scenarios)
- Test boundary conditions (maximum and minimum expected values)
- Test for Exceptions (expected and unexpected exceptions)
- Test for performance
  - Throughput
  - Response Time

Unit Testing (utPLSQL)

- Auto Testing using utPLSQL
- Best Open-source unit testing package for PL/SQL
- Includes many assertion features out of the box
- Automatic rollback of all test data (sandbox)
- Can be run from SQL Developer, command Line, Maven …
- Can be integrated with DevOps pipelines

Exception Handling Guidelines

- Study how error raising, handling, and logging work in PL/SQL
- Distinguish between deliberate, unfortunate, and unexpected errors.
- Decide on the standard ways that you will write code to respond to these different errors.
- Avoid application logic in the exception section.
- Transform the exception to a status indicator that can be interpreted by the user of that code.
- Handle those unexpected, “hard” errors and then re-raise the exception.
- How do I log my error?
  Develop and use a single component (which will usually consist of a central package or two, plus one or more supporting tables) that does most of the work
- Use the EXCEPTION_INIT pragma to name exceptions and make your code more accessible.



All exceptions are not created equal

- *Deliberate*
  - The code architecture itself deliberately relies upon an exception in the way it works. This means you must (well, *should*) anticipate and code for this exception. An example is UTL_FILE.GET_LINE.
- *Unfortunate*
  - This is an error, but one that is to be expected and one that may not even indicate that a problem has occurred. An example is a SELECT INTO statement that raises NO_DATA_FOUND.
- *Unexpected*
  - This is a "hard" error indicating a problem in the application. An example is a SELECT INTO statement that is supposed to return a row for a given primary key, but instead raises TOO_MANY ROWS.

Life After Compilation (Test, Trace, Debug)

- Testing
  - to verify the functionality of the program
  - the more you can automate the testing process the better
  - You then need to figure out which lines of code are causing that problem
- Tracing
  - Used to narrow the scope of the problem (before going to Debugging)
  - Avoid direct calls to DBMS_OUTPUT.PUT_LINE
  - There are much better options to choose from (open-source packages)
- Debugging
  - Better to utilize a full-fledged IDE
  - Don’t forget to add test case for the found bug!

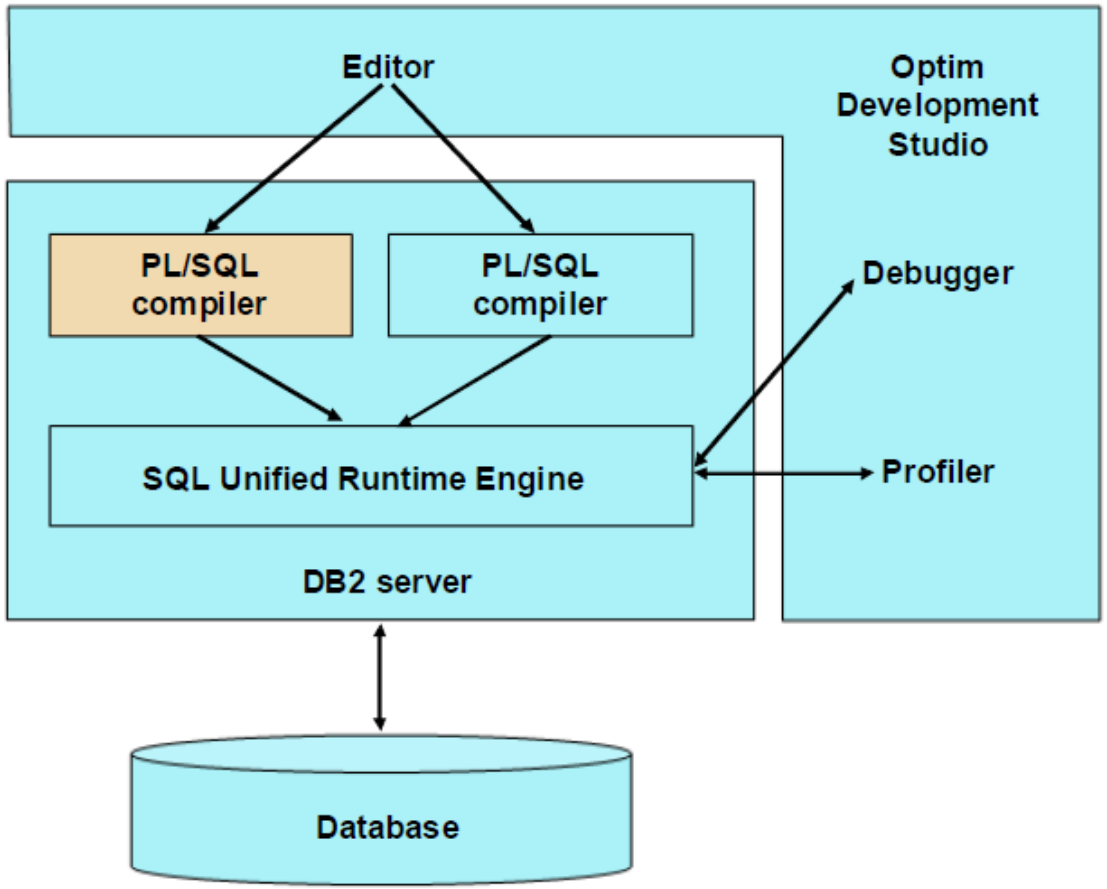
Code Generation

- Quick SQL
  - Included in Oracle APEX and Live SQL
  - Hundreds of standardized error-free lines of code (SQL & PL/SQL) from a handful lines of code
- **oddgen.org**
  - Dictionary-driven PL/SQL code Generation
  - Extensible
  - Has a SQL Developer Plugin
- PL/SQL TAPI (Table API's) Generators
  - We will study the best available options and use them as a starting point to our generator.
  - *Our generator will embed our standards (uTesting, Tracing, Comments, Exception Handling, Naming conventions, …)*
  - *Our generator Can generate PL/SQL, TSQL, PGPlsql, …*

Naming Conventions (Real Developers follow standards)

What I am declaring	Naming convention
IN parameter	<*root*>_in
OUT parameter	<*root*>_out
IN OUT parameter	<*root*>_inout or <*root*>_io
Constant	c_<*root*>
Global package variable	g_<*root*>
Local variable	l_<*root*>
Cursor	<*root*>_cur
Associative array type	<*root*>_aat
Nested table type	<*root*>_nt
VARRAY type	<*root*>_vat
Record type	<*root*>_rt
Object type	<*root*>_ot
Procedure and function name	None. I do not subscribe to a prefix of "p_" for procedure and "f_" for function. It seems to me that the *usage* of these subprograms clearly indicates what they are.

PL/SQL in DB2 ☺



- DB2 introduced native PL/SQL support
- DB2 engine now includes a PL/SQL compiler with the SQL PL compiler
- Both compilers produce virtual machine code for DB2 SQL Unified Runtime Engine
- Database Tools (Debugger, Profiler ...) are hooked into DB2 at the runtime engine level

The integration of PL/SQL into DB2 as a **first class** procedural language has several implications:

- There is **no translation**. The source code remains as it is in the schema catalog
- Developers can continue working in the language with which they are familiar. There is no need to translate logic to DB2 dialect even if new logic is written in SQL PL. Routines using different dialects can call each other
- Packaged application vendors can use one source code against both Oracle and DB2
- Both PL/SQL and SQL PL produce the same virtual machine code for the DB2 SQL Unified Runtime Engine. Therefore, by design, both PL/SQL and SQL PL perform at the same speed.
- Because the debugger infrastructure hooks directly into the SQL Unified Runtime Engine, PL/SQL is naturally supported by IBM Data Studio

DB2 PL/SQL Support (reference)

Suggested Approach (for using PL/SQL in DB2)

- Use PL/SQL for all our standard features (common core like EAIR, NE, SDR …).
- Compile, test, and optimize on both DB2 and Oracle (preferably DB2 first)
- If a needed feature is implemented differently (or poorly performing) in DB2 than Oracle, we should have a separate procedure for it with 2 versions (one in PL/SQL and one in SQL PL)
- For project-specific business logic, we will prefer PL/SQL unless it is forced by customer to use SQL PL.

Analyze the following PL/SQL Examples

Avoid Row-by-Row Processing

- This type of loop-based processing construct is highly discouraged as it leads to ***non-scalable code***
- If SELECT takes 0.1 sec, INSERT takes 0.1 sec → ***5.5 h for 100,000 rows***
- ***Context switches*** increase elapsed time of your programs and introduce unnecessary CPU overhead
- Tom Kyte (AskTom) termed this type of processing as ***slow-by-slow processing***

```
DECLARE
  CURSOR c1 IS
    SELECT prod_id, cust_id, time_id, amount_sold
    FROM sales
    WHERE amount_sold > 100;
  c1_rec c1%rowtype;
  l_cust_first_name customers.cust_first_name%TYPE;
  l_cust_lasT_name customers.cust_last_name%TYPE;
BEGIN
  FOR c1_rec IN c1
  LOOP
    -- Query customer details
    SELECT cust_first_name, cust_last_name
    INTO l_cust_first_name, l_cust_last_name
    FROM customers
    WHERE cust_id=c1_rec.cust_id;
    --
    -- Insert in to target table
    --
    INSERT INTO top_sales_customers (
      prod_id, cust_id, time_id, cust_first_name, cust_last_name,amount_sold
      )
      VALUES
      (
        c1_rec.prod_id,
        c1_rec.cust_id,
        c1_rec.time_id,
        l_cust_first_name,
        l_cust_last_name,
        c1_rec.amount_sold
      );
  END LOOP;
  COMMIT;
END;
/
PL/SQL procedure successfully completed.

Elapsed: 00:00:10.93
```

```
INSERT
INTO top_sales_customers
  (
    prod_id,
    cust_id,
    time_id,
    cust_first_name,
    cust_last_name,
    amount_sold
  )
SELECT s.prod_id,
  s.cust_id,
  s.time_id,
  c.cust_first_name,
  c.cust_last_name,
  s.amount_sold
FROM sales s,
     customers c
WHERE s.cust_id    = c.cust_id and
s.amount_sold> 100;

135669 rows created.

Elapsed: 00:00:00.26
```

Avoid Nested Row-by-Row Processing

```
DECLARE
  CURSOR c1 AS
    SELECT n1 FROM t1;
  CURSOR c2 (p_n1) AS
    SELECT n1, n2 FROM t2 WHERE n1=p_n1;
  CURSOR c3 (p_n1, p_n2) AS
    SELECT text FROM t3 WHERE n1=p_n1 AND n2=p_n2;
BEGIN
  FOR c1_rec IN c1
  LOOP
    FOR c2_rec IN c2 (c1_rec.n1)
    LOOP
      FOR c3_rec IN c3(c2_rec.n1, c2_rec.n2)
      LOOP
        -- execute some sql here;
        UPDATE … SET ..where n1=c3_rec.n1 AND n2=c3_rec.n2;
      EXCEPTION
      WHEN no_data_found THEN
         INSERT into… END;
      END LOOP;
    END LOOP;
  END LOOP;
 COMMIT;
END;
/
```

- This type of nested loop-based processing construct is highly discouraged as it leads to ***non-scalable code***
- if the UPDATE statement is optimized to execute in 0.01 seconds, performance of the program suffers due to the deeply nested cursor. Say that cursors c1, c2, and c3 retrieve 20, 50, and 100 rows, respectively. The code then loops through 100,000 rows, and the total elapsed time of the program exceeds 1,000 seconds.
- ***Context switches*** increase elapsed time of your programs and introduce unnecessary CPU overhead

```
MERGE INTO fact1 USING
(SELECT DISTINCT c3.n1,c3.n2
 FROM t1, t2, t3
 WHERE t1.n1      = t2.n1
 AND t2.n1        = t3.n1
 AND t2.n2        = t3.n2
) t
ON (fact1.n1=t.n1 AND fact1.n2=t.n2)
WHEN matched THEN
  UPDATE SET .. WHEN NOT matched THEN
  INSERT .. ;
  COMMIT;
```

The MERGE statement was introduced in DB2 V8.1 FP1

What if the logic is too complex to replace with SQL?

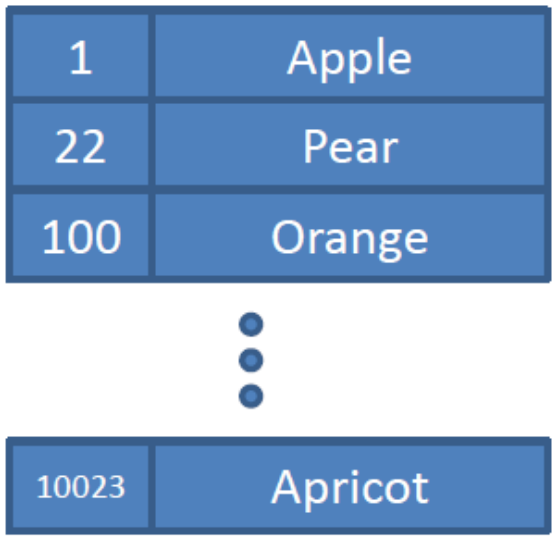
→ Use Collections as a scratchpad then bulk DML

PL/SQL Collections

- Collections are *single-dimensioned* lists of information, similar to 3GL arrays.
- They are an invaluable data structure.
  - *All* PL/SQL developers should be very comfortable with collections and use them often.
- Collections take some getting used to.
  - They are not the most straightforward implementation of array-like structures.
  - Advanced features like string indexes and multi-level collections can be a challenge.

What is a collection?

- A collection is an "ordered group of elements, all of the same type." (PL/SQL User Guide)
  - In short, a list of "stuff"
- Collections are similar to single-dimensional arrays in other programming languages.
  - With lots of subtle differences, as well.



Why use collections?

- Generally, to manipulate lists of information.
  - Of course, you can use relational tables, too.
  - Collection manipulation is generally much faster than using SQL to modify the contents of tables.
- Collections *enable* other key features of PL/SQL
  - BULK COLLECT and FORALL use them to boost multi-row SQL performance
  - Serve up complex datasets of information to non-PL/SQL host environments using *table functions*.

What makes collections so fast?

- You can read and write data in a collection much, *much* more quickly than in a relational table. Why is that?
- There are two general types of memory for Oracle data:
  - System or instance-level memory (SGA)
  - Process or session-specific memory (PGA)

Memory Management and Collections

- Memory for collections (and almost all PL/SQL data structures) is allocated from the PGA (Process Global Area).
- Accessing PGA memory is quicker than accessing SGA memory.
  - Sometimes much, much faster.
- Collections represent a very clear tradeoff: use more memory *(per session)* to improve performance.
  - But you definitely need to keep an eye on your PGA memory consumption.

Different Types of Collections

- Three types of collections
  - Associative array
  - Nested table
  - Varray (varying arrays)
- Associative array is a PL/SQL-only datatype.
- Nested tables and varrays can be used within PL/SQL blocks and also from within the SQL layer.

Bulk Processing of SQL in PL/SQL

- The central purpose of PL/SQL is to provide a portable, fast, easy way to write and execute SQL against an Oracle database.
- Unfortunately, this means that most developers take SQL for granted when writing SQL...and just *assume* Oracle has fully (automagically) optimized how SQL will run from within PL/SQL.

The Problem with SQL in PL/SQL

- Many PL/SQL blocks execute the same SQL statement repeatedly with different bind values.
  - Retrieve data one row at a time.
  - Performs same DML operation for each row retrieved.

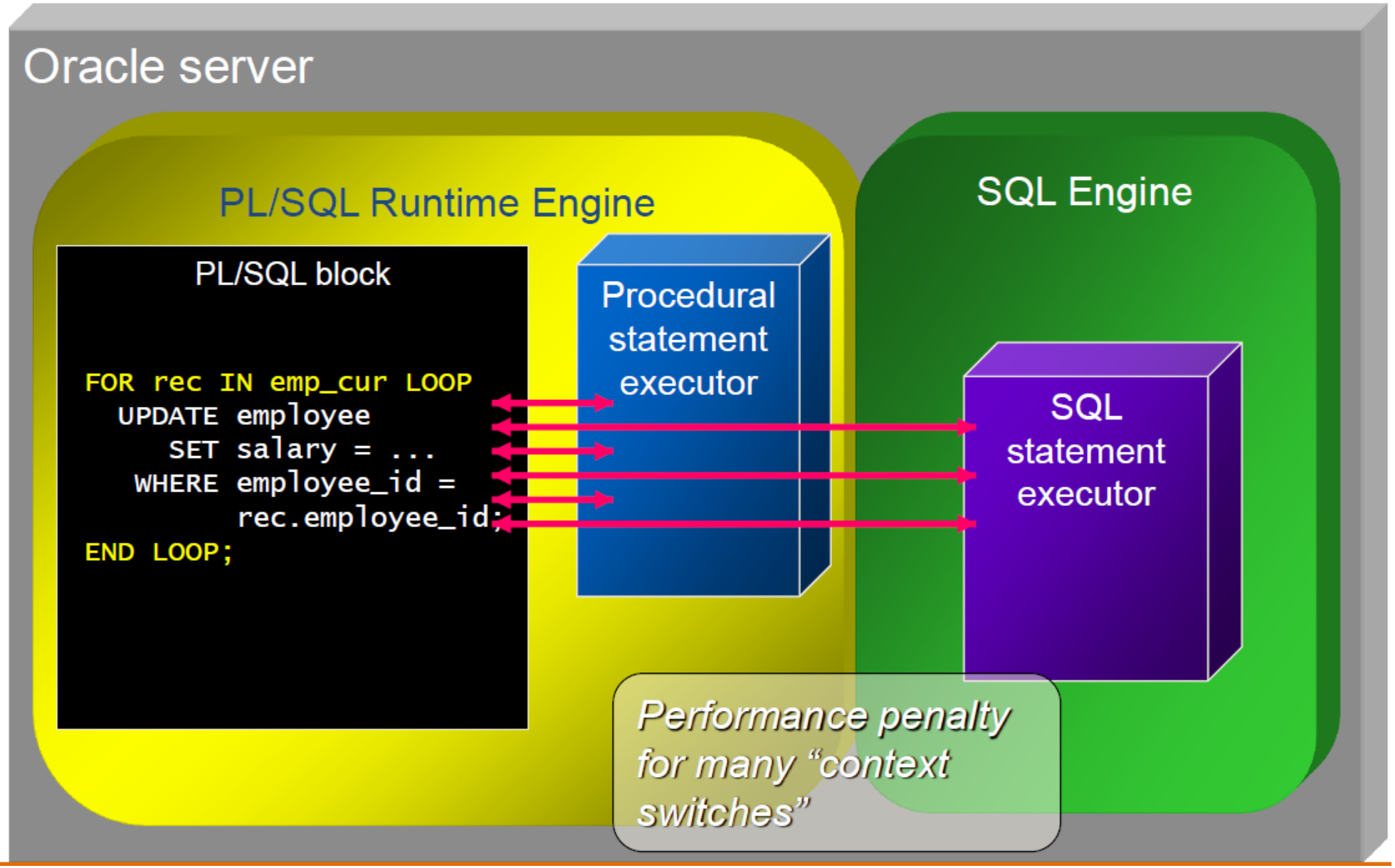
```
CREATE OR REPLACE PROCEDURE upd_for_dept (
   dept_in IN employee.department_id%TYPE
  ,newsal_in IN employee.salary%TYPE)
IS
   CURSOR emp_cur IS
      SELECT employee_id,salary,hire_date
        FROM employee WHERE department_id = dept_in;
BEGIN
   FOR rec IN emp_cur LOOP

      adjust_compensation (rec, newsal_in);

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

The result? Simple and elegant but *inefficient*... Why is this?

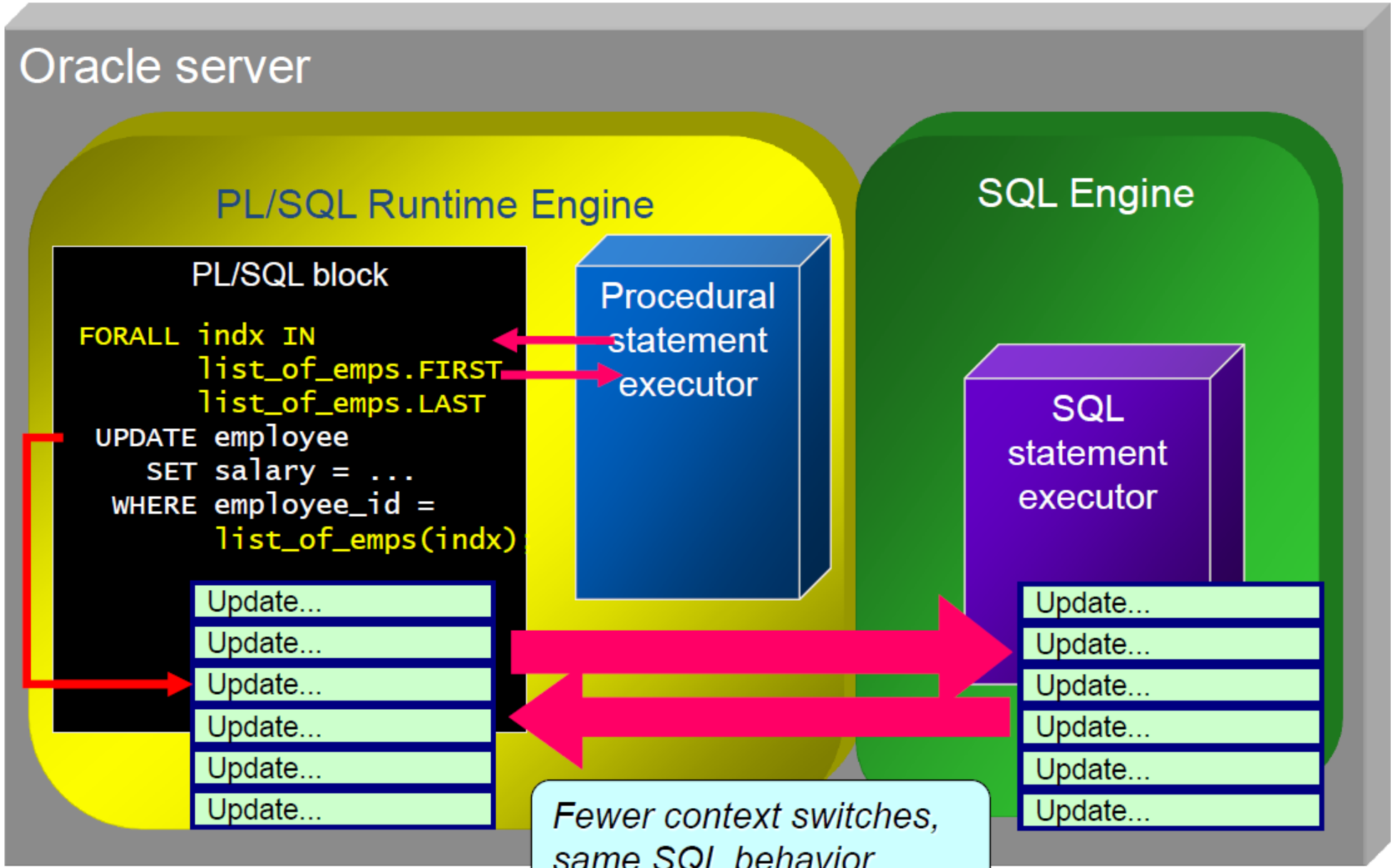
Repetitive statement processing from PL/SQL



Bulk Processing in PL/SQL

- The goal is straightforward: reduce the number of context switches and you improver 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 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.

A phased approach with bulk processing

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

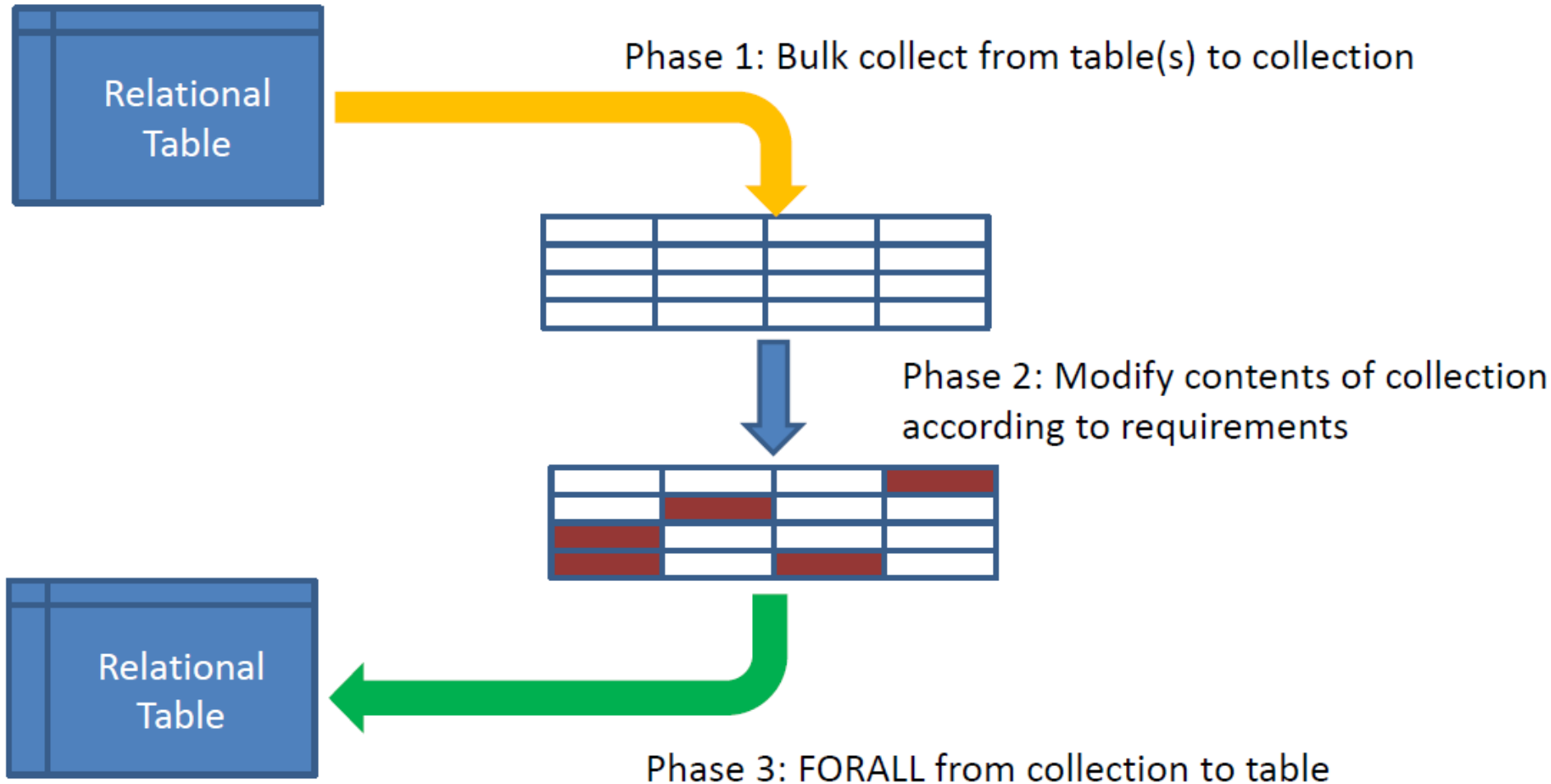


Table Functions

- A table function is a function that you can call in the FROM clause of a query, and have it be treated as if it were a *relational table.*
- Table functions allow you to perform arbitrarily complex transformations of data and then make that data available through a query: "just" rows and columns!
  - After all, not everything can be done in SQL.
- Table functions can also help improve performance in several ways.

When should you use a table function?

- To pass datasets back to a non-PL/SQL host environment, such as Java and .Net.
  - They just deal with rows and columns of data.
  - To do this, you will need to take advantage of cursor variables.
- Improve query performance with pipelined table functions.
  - For parallel query environments (data warehouse)
  - And to reduce user *perceptions* of elapsed time

The NOCOPY hint

- By default, Oracle passes all IN OUT and OUT arguments by *value*, not reference.
  - This means that OUT and IN OUT arguments always involve some *copying* of data.
- With NOCOPY, you turn off the copy process.
  - But it comes with a risk: Oracle will not automatically "rollback" or reverse changes made to your variables if the NOCOPY-ed program raises an exception.

Dynamic SQL in PL/SQL Programs

- Dynamic SQL actually refers, in the world of PL/SQL, to two things:
  - SQL statements, such as a DELETE or DROP TABLE, that are constructed and executed at run-time.
  - Anonymous PL/SQL blocks that are constructed, compiled and executed at run-time.

```
'DROP ' ||
   l_type || ' ' || l_name
```

```
'BEGIN ' ||
   l_proc_name || ' (' ||
   l_parameters || '); END;'
```

Use Dynamic SQL To...

- Build ad-hoc query and update applications.
  - The user decides what to do and see.
- Execute DDL statements from within PL/SQL.
  - Not otherwise allowed in a PL/SQL block.
- Soft-code your application logic, placing business rules in tables and executing them dynamically.
  - Usually implemented through dynamic PL/SQL

Two Mechanisms Available

- DBMS_SQL
  - A large and complex built-in package that made dynamic SQL possible in Oracle7 and Oracle8.
- Native Dynamic SQL
  - A new (with Oracle8i), native implementation of dynamic SQL that does *almost* all of what DBMS_SQL can do, but much more easily and usually more efficiently.
  - EXECUTE IMMEDIATE
  - OPEN cv FOR 'SELECT ... '

Quiz!

- What's wrong with this code?
- How would you fix it?

```
PROCEDURE process_lineitem (
   line_in IN PLS_INTEGER)
IS
BEGIN
   IF line_in = 1
   THEN
      process_line1;
   END IF;

   IF line_in = 2
   THEN
      process_line2;
   END IF;
   ...
   IF line_in = 22045
   THEN
      process_line22045;
   END IF;
END;
```

From 22,000 lines of code to 1!

```
PROCEDURE process_lineitem (
   line_in IN INTEGER)
IS
BEGIN
   IF line_in = 1
   THEN
      process_line1;
   END IF;

   IF line_in = 2
   THEN
      process_line2;
   END IF;
   ...
   IF line_in = 22045
   THEN
      process_line22045;
   END IF;
END;
```

```
PROCEDURE process_lineitem (
  line_in IN INTEGER)
IS
BEGIN
   EXECUTE IMMEDIATE
      'BEGIN process_line'||
         line_in ||'; END;';
END;
```

- Identify the pattern and resolve it either with reusable modules or dynamic abstractions.

Must Know Error Management Features

- DBMS_UTILITY functions
  - FORMAT_CALL_STACK
  - FORMAT_ERROR_STACK
  - FORMAT_ERROR_BACKTRACE
- DBMS_ERRLOG and LOG ERRORS
  - Suppress errors at row level in SQL layer

Oracle Built-ins For Handling Exceptions

- In addition to the application-specific information you may want to log, Oracle built-ins provide you with answers to the following questions:
  - How did I get here?
  - What is the error code?
  - What is the error message and/or stack?
  - On what line was the error raised?

DBMS_UTILITY error functions

- Answer the question "How did I get here?" with DBMS_UTILITY.FORMAT_CALL_STACK.
- Get a more complete error message with DBMS_UTILITY.FORMAT_ERROR_STACK.
- Find line number on which error was raised with DBMS_UTILITY.FORMAT_ERROR_BACKTRACE.

DBMS_ERRLOG and LOG ERRORS- agenda

- Impact of errors on DML execution
- Introduction to LOG ERRORS feature
- Creating an error log table
- Adding LOG ERRORS to your DML statement
- "Gotchas" in the LOG ERRORS feature
- The DBMS_ERRLOG helper package

Impact of errors on DML execution

- A single DML *statement* can result in changes to multiple *rows*.
- When an error occurs on a change to a row....
  - All previous changes from that statement are rolled back.
  - No other rows are processed.
  - An error is passed out to the calling block (turns into a PL/SQL exception).
  - No rollback on completed DML in that session.
- *Usually* acceptable, but what if you want to:
  - Avoid losing all prior changes?
  - Avoid the performance penalty of exception management in PL/SQL?

Row-level Error Suppression in DML with LOG ERRORS

- Once the error propagates out to the PL/SQL layer, it is too late; all changes to rows have been rolled back.
- The only way to preserve changes to rows is to add the LOG ERRORS clause in your DML statement.
  - Errors are suppressed at row level *within* the SQL Layer.
- But you will first need to created an error log table with DBMS_ERRLOG.

The Oracle 11g Function Result Cache

- Oracle offers a far superior caching solution than PGA caching in 11g: the Function Result Cache.
- This cache is...
  - stored in the SGA
  - shared across sessions
  - purged of dirty data automatically
- You can use and should use it to retrieve data from any table that is queried more frequently than updated.

How the Function Result Cache Works

- Add the RESULT_CACHE clause to your function's header.
- 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.

Hard-Coding Avoidance: Principles and Concepts

- Single point of definition (no repetition)
  - You should always aim for a single point of definition or *SPOD* for everything in your application.
- Information hiding – the *name* is the thing
  - Avoid exposing the implementation details of formulas, rules, algorithms, data access.
  - The more you hide, the more flexibility you have.
- "Never" and "Always" in software
  - It's *never* going to stay the same.
  - It's *always* going to change.
- Pay attention to that "voice in your head."

Potential Hard-Codings in PL/SQL Code

- Literal values
  - Especially language-specific literals
- Constrained declarations
  - Especially VARCHAR2(n)
- Fetch into a list of variables
- Rules and formulas - *especially* the "trivial" ones
- SQL statements – ah, very scary!
- Algorithmic details
  - Example: error logging mechanisms
- Transaction boundaries and dates

Magical Values (Literals)

"The maximum salary allowed for employees is 1,000,000."

- The most commonly recognized form of hard-coding.
- The only place a literal should appear in your code is in its *SPOD*.
- Hide literals behind constants or functions.
- Consider soft coding values in tables.

Hide error codes with EXCEPTION_INIT

- Oracle doesn't provide a name for every error code, but *you* can do this.
- Best place to put exception declarations is a package, so they can be shared across the application.

```
WHEN OTHERS
THEN
   IF SQLCODE = -24381
   THEN
      ...
   ELSIF SQLCODE = -1855
   THEN
      ...
   ELSE
      RAISE;
END;
```

```
   e_forall_failure EXCEPTION;
   PRAGMA EXCEPTION_INIT (
      e_forall_failure, -24381);
BEGIN
   ....
EXCEPTION
   WHEN e_forall_failure
   THEN      ...
END;
```

Avoiding Hard-Coded Declarations

- What's the problem?
- Always fetch into records
- Use %TYPE and %ROWTYPE whenever possible
- Use SUBTYPE to define application-specific types

What's the problem?

- Hard-coded declarations are declarations that reference PL/SQL base types and/or constrain values in some way.
- Very often our variables hold data that is stored elsewhere (columns of tables).
- If the underlying column definition changes, we can get VALUE_ERROR (ORA-06502) and other errors.
- Generally, our variable types can become "out of synch" with the data.

Fetch into list of variables

- If your FETCH statement contains a list of individual variables, you are hard-coding the number of elements in the SELECT list.
  - When the cursor changes, you must change the FETCH as well.
- Solution: always fetch into a record, defined with %ROWTYPE against the cursor.

Do not expose constrained declarations

- Every declaration requires a datatype.
- If you are not careful, the way you specify that datatype could be a hard-coding.
- Generally, any declaration that relies on a *constrained datatype* is a hard-coding.
  - BOOLEAN and DATE are unconstrained.
- Another way to remember this is:

Consider every VARCHAR2(N) declaration to be a *bug* – unless it's a SPOD.

"SPODification" for Datatypes

- Two problems with hard-coding the datatype:
  - Constraints can lead to errors in future.
  - The datatype does not explain the *application significance* of the element declared.
- Whenever possible, *anchor* the datatype of your declaration to an already-existing type.
  - That way, if the existing type or SPOD ever changes, then your code will be marked INVALID and automatically recompiled to pick up the latest version of the anchoring type.

%TYPE and %ROWTYPE

- Use %TYPE for declarations based on columns in tables.
- Use %ROWTYPE for records based on tables, views or cursors.
- The lookup of the datatype from these attributes occurs at compile-time.
  - There is no run-time overhead.

SUBTYPEs

- You can't always use %TYPE or %ROWTYPE in your declaration.
- You can, however, *always* define a "subtype" or subset of an existing type with the SUBTYPE statement. SUBTYPE benefits:
  - Avoid exposing and repeating constraints.
  - Give application-specific names to types. Critical when working with complex structures like collections of records, and nested collections.
  - *Apply* constraints, such as numeric ranges, to the variable declared with the subtype.

SUBTYPE Details and Examples

```
SUBTYPE type_name IS data_type [ constraint ] [ NOT NULL ]
```

- Define a subtype based on any pre-defined type or other, already-defined subtype.
- If the base type can be constrained, then you can constrain the subtype.
  - (precision,scale) or RANGE
- You can also, always specify NOT NULL.
  - Even if the base type could be NULL.

Applying SUBTYPEs

- Two key scenarios:
  - Whenever you are about to write a VARCHAR2(N) or other constrained declaration, define a subtype instead, preferably in a package specification.
  - Instead of writing a comment explaining a declaration, put the explanation *into* a subtype.

Instead of this:

```
DECLARE
   l_full_name VARCHAR2(100);
   l_big_string VARCHAR2(32767);
```

Write this:

```
DECLARE
   l_full_name employees_rp.full_name_t;
   l_big_string plsql_limits.maxvarchar2;
```

Stop Writing So Much SQL

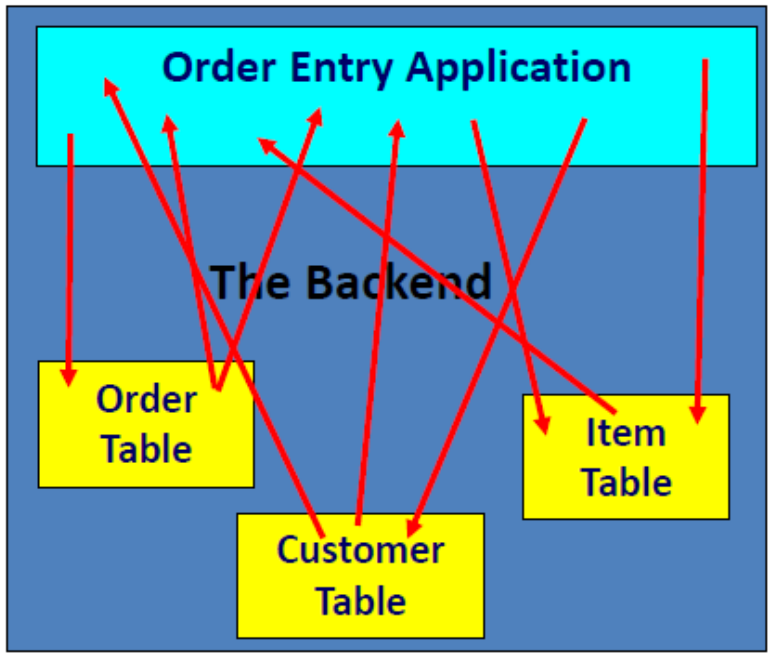
- What's the problem with writing lots of SQL?
- Why talk about SQL in a lesson on hard-coding?
- Concepts behind and benefits of data encapsulation
- Data encapsulation recommendations

Writing SQL in PL/SQL

- The most critical aspect of our programs.
- SQL statements directly reflect our business models.
  - And those models are always changing.
- SQL statements cause most of the performance problems in our applications.
  - Tuning SQL and the way that SQL is called in PL/SQL overwhelms all other considerations.
- Many runtime errors in applications result from integrity and check constraints on tables.

The fundamental problem with SQL in PL/SQL

- We take it entirely for granted.
  - Why not? It's so easy to write SQL in PL/SQL!
- We don't set rules on how, when and where SQL should be written in PL/SQL.



The result? Slow, buggy code that is difficult to optimize and maintain.

So set some SQL standards!

- At a minimum, before starting your next application, ask yourselves explicitly:
- 1. Are we taking full advantage of SQL, particularly new features in our version?
  - You should do as much as possible in "pure" SQL.
- 2. Do we want standards or should we just do whatever we want, whenever we want?
  - That way, you are making a conscious decision.

Fully leverage SQL in your PL/SQL code

- Oracle continually adds significant new functionality to the SQL language.
- If you don't keep up with SQL capabilities, you will write slower, more complicated PL/SQL code than is necessary.
  - I am actually a good example of what you *don't* want to do or how to be.
- So take the time to refresh your understanding of Oracle SQL in 10g and 11g.

Some exciting recently added SQL features

- Courtesy of Lucas Jellema of AMIS Consulting
- Analytical Functions
  - Especially LAG and LEAD; these allow you to look to previous and following rows to calculate differences.
- WITH clause (subquery factoring)
  - Allows the definition of 'views' inside a query that can be used and reused; they allow procedural top-down logic inside a query
- Flashback query
  - No more need for journal tables, history tables, etc.
- ANSI JOIN syntax
  - Replaces the (+) operator and introduces FULL OUTER JOIN
- SYS_CONNECT_BY_PATH and CONNECT_BY_ROOT for hierarchical queries
- Scalar subquery
  - Adds a subquery to a query like a function call.

```
select d.deptno
     , (select count(*)
        from emp e where
e.deptno = d.deptno)
number_staff from dept
```

SQL statements as hard-codings

- I suggest that every SQL statement you will ever write is a hard-coding. Consider....
- I need to write a complex query to return HR data for a report.

```
SELECT . . .
  FROM employees, departments, locations
 WHERE . . . (a page full of complex conditions)
```

- And Joe needs to use that same query in his business rule procedure. And so on...
- And then the three way join turns into a four way join – and we have to find all occurrences of this query. **A very tough thing to do!**

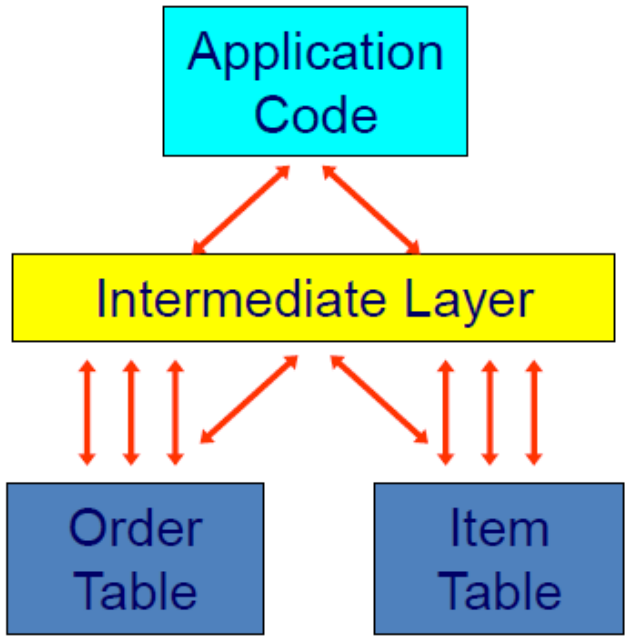
What to do about SQL hard coding

- Of course, you have to (and should) write SQL statements in your PL/SQL code.
  - PL/SQL is, in fact, the ***best*** place for SQL.
- But we should be *very* careful about where, when and how we write these statements.
  - Follow the principles; they are your guide.
  - ***Don't repeat anything!***
- The best approach: hide SQL statements inside a data access layer.

SQL as a Service

- Think of SQL as a *service* that is provided to you, not something you *write*.
  - Or if you write it, you put it somewhere so that it can be easily found, reused, and maintained.

- This service consists of views and programs defined in the data access layer.
  - Views hide complex query construction
  - Packaged APIs – for tables, transactions and business entiries

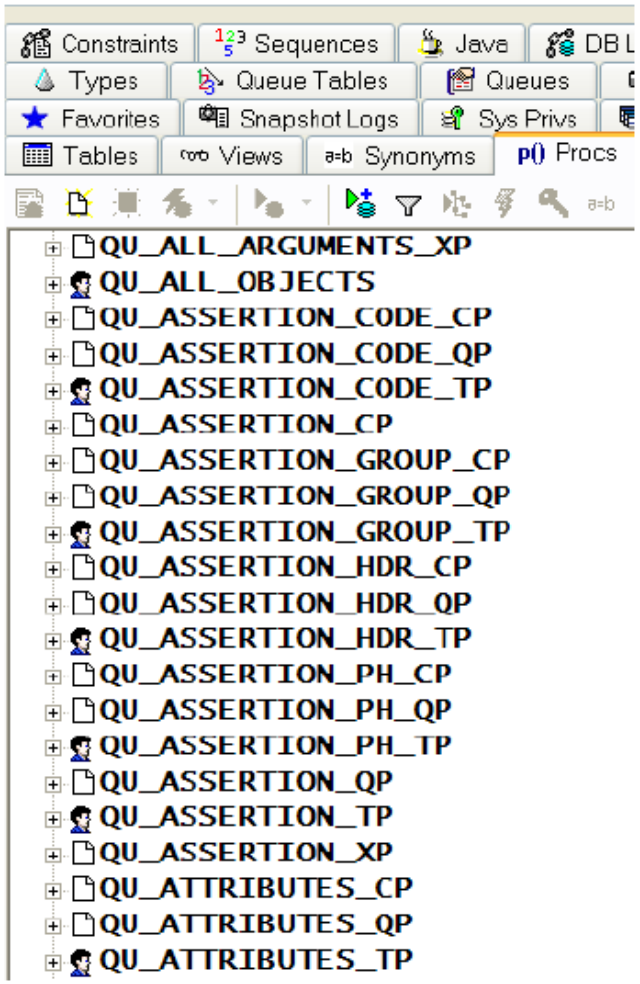


With a data access layer, I can...

- Change/improve my implementation with minimal impact on my application code.
  - The underlying data model is constantly changing.
  - We can depend on Oracle to add new features.
  - We learn new ways to take advantage of PL/SQL.
- Vastly improve my SQL-related error handling.
  - Do you handle dup_val_on_index for INSERTs, too_many_rows for SELECT INTOs, etc?
- Greatly increase my productivity
  - I want to spend as much time as possible implementing business requirements.

Example: Quest Code Tester backend

- For each table, we have three generated packages:
  - <table>_CP for DML
  - <table>_QP for queries
  - <table>_TP for types
- And usually an "extra stuff" package with custom SQL logic and related code:
  - <table>_XP



How to implement data encapsulation

- It must be very consistent, well-designed and efficient - or it will not be used.
- Best solution: generate as much of the code as possible.
  - This includes products like APEX and Hibernate that generate lots of their own SQL for you.
- Any custom SQL statements should be written once and placed in a shareable *container* (usually a package).

Hiding the Mechanics

- When you hard-code a "magic value," you write that value directly into your code as a literal.
  - Over and over again; the value is "exposed".
- When you "expose" the way you *get things done*, you are hard-coding a particular implementation.
  - The more you repeat it, the more difficult it is to *change* that implementation in the future.

Everything Changes, Hide the Details

- Even if the users don't change their minds, we (developers) and Oracle technology change.
  - And let's face it: sometimes the way that Oracle implements things is not ideal.
- So *assume* that whatever you are working on will change – and hide it behind an API.
  - Logging errors and tracing execution
  - Calls to Oracle built-ins, like UTL_FILE.
  - Soft-code transaction boundaries
  - Watch out for SYSDATE

Logging Errors

- We usually, but not always, want to write error information out to a log table. How's this?

```
WHEN NO_DATA_FOUND
THEN
   l_code := SQLCODE;
   INSERT INTO errlog
     VALUES ( l_code
            , 'No company for id ' || TO_CHAR ( v_id )
            , 'fixdebt', SYSDATE, USER );
WHEN OTHERS
THEN
   l_code := SQLCODE; l_errm := SQLERRM;
   INSERT INTO errlog
     VALUES (l_code, l_errm, 'fixdebt', SYSDATE, USER );
   RAISE;
END;
```

- It's easy to "read" but only because it exposes the logging mechanism.

Hide how and what you log

- Don't call RAISE_APPLICATION_ERROR.
- Don't explicitly insert into log table or write to file.
- Don't call all the useful built-in functions in each handler.
- *Do* use a generic and shared error management utility.
  - Check out Quest Error Manager at PL/SQL Obsession for an example.

```
WHEN NO_DATA_FOUND
THEN
   q$error_manager.register_error (
       text_in => 'No company for id ' || TO_CHAR ( v_id ));
WHEN OTHERS
THEN
   q$error_manager.raise_unanticipated (
      name1_in => 'COMPANY_ID', value1_in => v_id);
END;
```

Execution Tracing (Instrumentation)

- We often need to retrieve additional, application-specific information from our code while running.
  - Especially in production.
- DBMS_OUTPUT.PUT_LINE is the "default" tracing mechanism – and should *never* appear in your application code.
  - You are exposing the trace/display mechanism.
  - Too many drawbacks, too little flexibility.

Alternative Tracing Mechanisms

- The Quest Error Manager available at www.ToadWorld.com/SF offers built-in tracing.
- The demo.zip watch.pkg offers lots of flexibility.
- OTN samplecode logger utility, designed primarily for use with APEX
- The log4plsql open source utility
- DBMS_APPLICATION_INFO
  - Writes to V$ views

Say goodbye to hard coding!

- It's not all that difficult to do, once you recognize all the different ways that hard coding can manifest itself in your code.
- *Repeat nothing*: become allergic to redundant repetition.
- Aim for a "*single point of definition*" in everything you write.
- Hide, hide, hide: values, implementations, workarounds

Your Reward

- Elegant, functional code that you and others can maintain easily
- The respect of your peers
- A deep sense of satisfaction with a job well done
- The opportunity to continue making a very fine living, mostly from just thinking about abstractions.

Extreme Modularization

- Spaghetti code is the bane of a programmer's existence.
- It is impossible to understand and therefore debug or maintain code that has long, twisted executable sections.
- Fortunately, it is really easy to make spaghetti code a thing of the past.

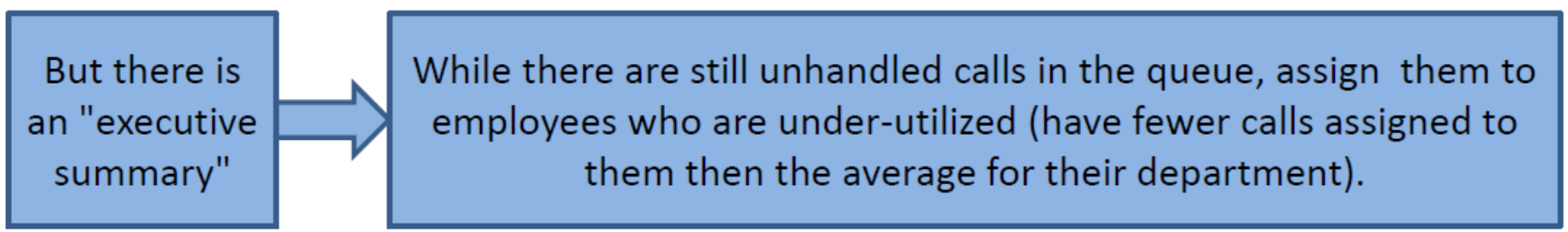
Organize your code so that the executable section has no more than fifty lines of code.

Fifty lines of code? That's ridiculous!

- Of course you write lots more than 50 lines of code in your applications.
- The question is: how will you organize all that code?
- Turns out, it is actually quite straightforward to organize your code so that it is transparent in meaning, with a minimal need for comments.
- Key technique: local or nested subprograms.

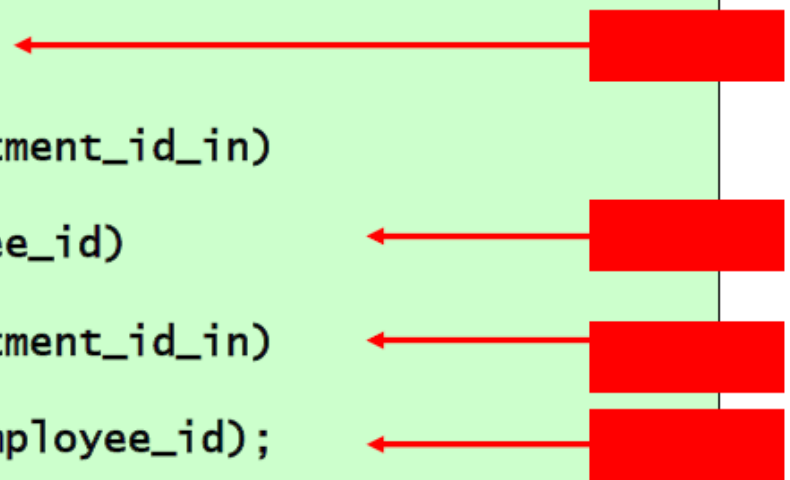
Let's write some code!

- My team is building a support application. Customers call with problems, and we put their call in a queue if it cannot be handled immediately.
  - I must now write a program that distributes unhandled calls out to members of the support team.
- Fifty pages of doc, complicated program!



First: Translate the summary into code.

```
PROCEDURE distribute_calls (
   department_id_in IN departments.department_id%TYPE)
IS
BEGIN
   WHILE ( calls_are_unhandled ( ) )
   LOOP
      FOR emp_rec IN emps_in_dept_cur (department_id_in)
      LOOP
         IF current_caseload (emp_rec.employee_id)
                <
                avg_caseload_for_dept (department_id_in)
         THEN
            assign_next_open_call (emp_rec.employee_id);
         END IF;
      END LOOP;
   END LOOP;
END distribute_calls;
```



- A more or less direct translation. No need for comments, the subprogram names "tell the story" – but those subprograms don't yet exist!

Next: Implement stubs for subprograms

```
PROCEDURE call_manager.distribute_calls (
   department_id_in IN departments.department_id%TYPE)
IS
   FUNCTION calls_are_handled RETURN BOOLEAN
   IS BEGIN ... END calls_are_handled;

   FUNCTION current_caseload (
      employee_id_in IN employees.employee_id%TYPE) RETURN PLS_INTEGER
   IS BEGIN ... END current_caseload;

   FUNCTION avg_caseload_for_dept (
      employee_id_in IN employees.employee_id%TYPE) RETURN PLS_INTEGER
   IS BEGIN ... END current_caseload;

   PROCEDURE assign_next_open_call (
      employee_id_in IN employees.employee_id%TYPE)
   IS BEGIN ... END assign_next_open_call;
BEGIN
```

- These are all defined *locally* in the procedure.

- Write *tiny* chunks of code.
  - Your programs will be transparent and readable, to you and everyone else.
  - They will contain many fewer bugs.
- The quality of your code will be instantly transformed!
- "All" it takes is discipline.
  - No special tools required (but a top-notch IDE to will make your job much easier).

Summary: Do Not Use (PL/SQL Pitfalls)

- Row-by-Row Processing