Oracle 10g Advanced PL/SQL Programming

Student Workbook

itcourseware
Training Materials For IT Professionals

ORACLE 10G ADVANCED PL/SQL PROGRAMMING

Contributing Authors: Robert Freeman, Mary Gable, Roger Jones, Brian Peasland, and Rob Roselius.

Published by ITCourseware, LLC., 7245 South Havana Street, Suite 100, Centennial, CO 80112

Editor: Danielle Hopkins and Jan Waleri.

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook, and the many students who have offered comments, suggestions, criticisms, and insights.

Contents

CHAPTER 1 - COURSE INTRODUCTION

Course Objectives

- Use detailed understanding of the PL/SQL execution environment in your application design and tuning.
- Develop programs that make sophisticated and effective use of cursors.
- Use all kinds of dynamic SQL in your PL/SQL code.
- Design and write solutions using Oracle's object types.
- Use Oracle's tools and supplied packages to trace, profile, and tune your PL/SQL programs.
- Use a variety of techniques and tools for debugging PL/SQL code.
- Write programs that interface between PL/SQL, and external procedures and programs.
- Use package state to solve application problems.
- Use autonomous transactions in stored subprograms and triggers.
- Choose which user's application context and rights will apply when a stored subprogram runs.
- Write high-performance code using **NOCOPY** and pipelined table functions.
- Create functions to implement fine-grained access control.
- Use **DBMS_PIPE** to set up inter-session communication between PL/SQL programs.

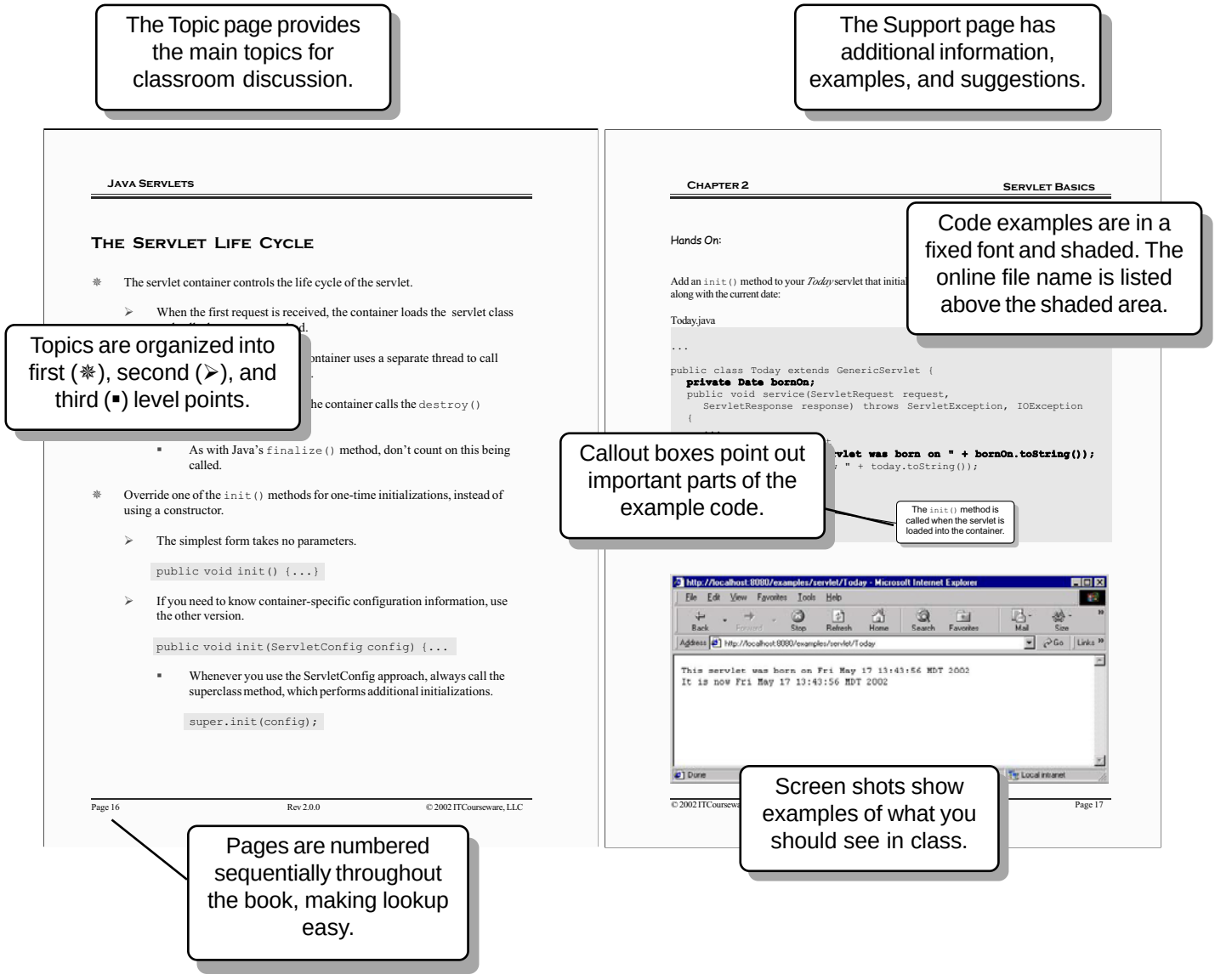
COURSE OVERVIEW

- **Audience**: Oracle application developers and database administrators.
- **Prerequisites**: *Introduction to Oracle10g PL/SQL Programming.*
- **Classroom Environment:**
  - A workstation per student.

USING THE WORKBOOK

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. *Hands On* sections provide opportunities for practical application of key concepts. Try It and Investigate sections help direct individual discovery.

In addition, there is an index for quick lookup. Printed lab solutions are in the back of the book as well as online if you need a little help.



Suggested References

Allen, Christopher. 2004. ***Oracle Database 10g PL/SQL 101***. McGraw-Hill Osborne, Emeryville, CA. ISBN 0072255404

Boardman, Susan, Melanie Caffrey, Solomon Morse, and Benjamin Rosenzweig. 2002. ***Oracle Web Application Programming for PL/SQL Developers***. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 0130477311

Date, C.J and Hugh Darwen. 1996. ***A Guide to SQL Standard***. Addison-Wesley, Boston, MA. ISBN 0201964260

Date, C.J. 2003. ***An Introduction to Database Systems***. Addison-Wesley, Boston, MA. ISBN 0321197844

Feuerstein, Steven, Charles Dye, and John Beresniewicz. 1998. ***Oracle Built-in Packages***. O'Reilly and Associates, Sebastopol, CA. ISBN 1565923758

Feuerstein, Steven. 2001. ***Oracle PL/SQL Best Practices***. O'Reilly and Associates, Sebastopol, CA. ISBN 0596001215

Feuerstein, Steven. 2000. ***Oracle PL/SQL Developer's Workbook***. O'Reilly and Associates, Sebastopol, CA. ISBN 1565926749

Feuerstein, Steven. 2002. ***Oracle PL/SQL Programming***. O'Reilly and Associates, Sebastopol, CA. ISBN 0596003811

Loney, Kevin. 2004. ***Oracle Database 10g: The Complete Reference***. McGraw-Hill Osborne, Emeryville, CA. ISBN 0072253517

McDonald, Connor, Chaim Katz, Christopher Beck, Joel R. Kallman, and David C. Knox. 2004. ***Mastering Oracle PL/SQL: Practical Solutions***. Apress, Berkeley, CA. ISBN 1590592174

Pribyl, Bill. 2001. ***Learning Oracle PL/SQL***. O'Reilly and Associates, Sebastopol, CA. ISBN 0596001800

Price, Jason. 2004. ***Oracle Database 10g SQL***. McGraw-Hill Osborne, Emeryville, CA. ISBN 0072229810

Rosenzweig, Benjamin and Elena Silvestrova. 2003. ***Oracle PL/SQL by Example***. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 0131172611

Trezzo, Joseph C., Bradley D. Brown, and Richard J. Niemiec. 1999. ***Oracle PL/SQL Tips and Tricks***. McGraw-Hill Osborne, Emeryville, CA. ISBN 0078824389

Urman, Scott and Michael McLaughlin. 2004. ***Oracle Database 10g PL/SQL Programming***. McGraw-Hill Osborne, Emeryville, CA. ISBN 0072230665

Urman, Scott and Tim Smith. 1996. ***Oracle PL/SQL Programming***. Oracle Press, Emeryville, CA. ISBN 0078821762

tahiti.oracle.com
www.oracle.com/technology/tech/pl_sql

CHAPTER 2 - THE PL/SQL EXECUTION ENVIRONMENT

OBJECTIVES

- Describe the basic architecture of the Oracle Database.
- Explain the purpose and components of the Program Global Area.
- Describe all stages in the execution of a PL/SQL program.

The Server Process

- All SQL and (usually all) PL/SQL for your session is performed by a *server process*.
  - Normally, each session has a server process dedicated to it.
  - The DBA can configure the database so that the SQL and PL/SQL work for all users is handled by a pool of shared server processes.
- The server process:
  - Receives and processes SQL and PL/SQL from your application.
  - Reads blocks of data from data files into the Buffer Cache for use.
  - Returns results to your application.
- The server process contains the heart of Oracle's database technology:
  - The SQL Cost-Based Optimizer
  - The SQL Executor
  - The PL/SQL compiler
  - The PL/SQL Virtual Machine
- Each server process creates its own memory region called the Program Global Area (PGA).
- All Oracle processes, including the server processes and the background processes comprising the instance, access a region of shared memory called the System Global Area (SGA).
  - This allows Oracle processes to share common data and to coordinate their activities.

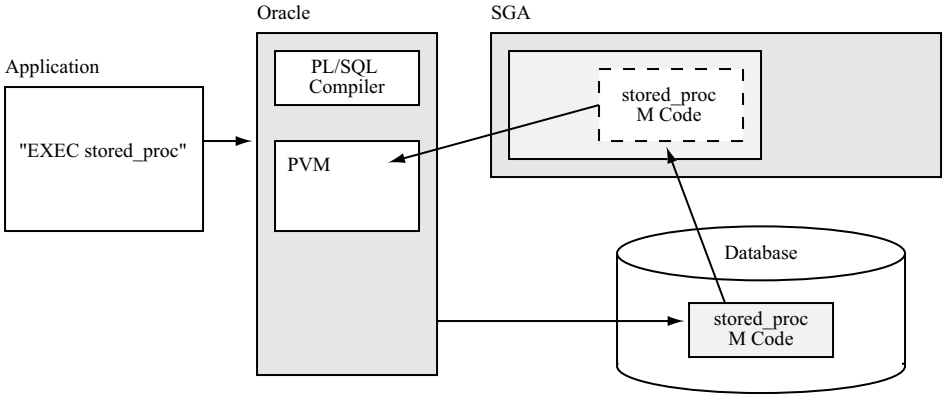
The program that executes when a server process starts is the Oracle Kernel — *ORACLE.EXE* on Windows, *oracle* or *oracle.bin* on UNIX and Linux platforms. The Oracle Kernel is a single, large (for Oracle10*g*, over 90 megabytes), binary executable, compiled for the database host platform. In addition to the SQL and PL/SQL subsystems, this program includes all functionality needed to run the database instance — the background processes which run when the Oracle Database is started are also copies of the Oracle Kernel. Thus, all processes accessing the database do so with the same technology and follow the same mechanisms.

A PL/SQL program runs embedded in the multi-user Oracle server environment. PL/SQL programs typically include SQL statements. To effectively design new PL/SQL applications, and to debug and tune existing ones, requires insight into how both PL/SQL and SQL execute at runtime.

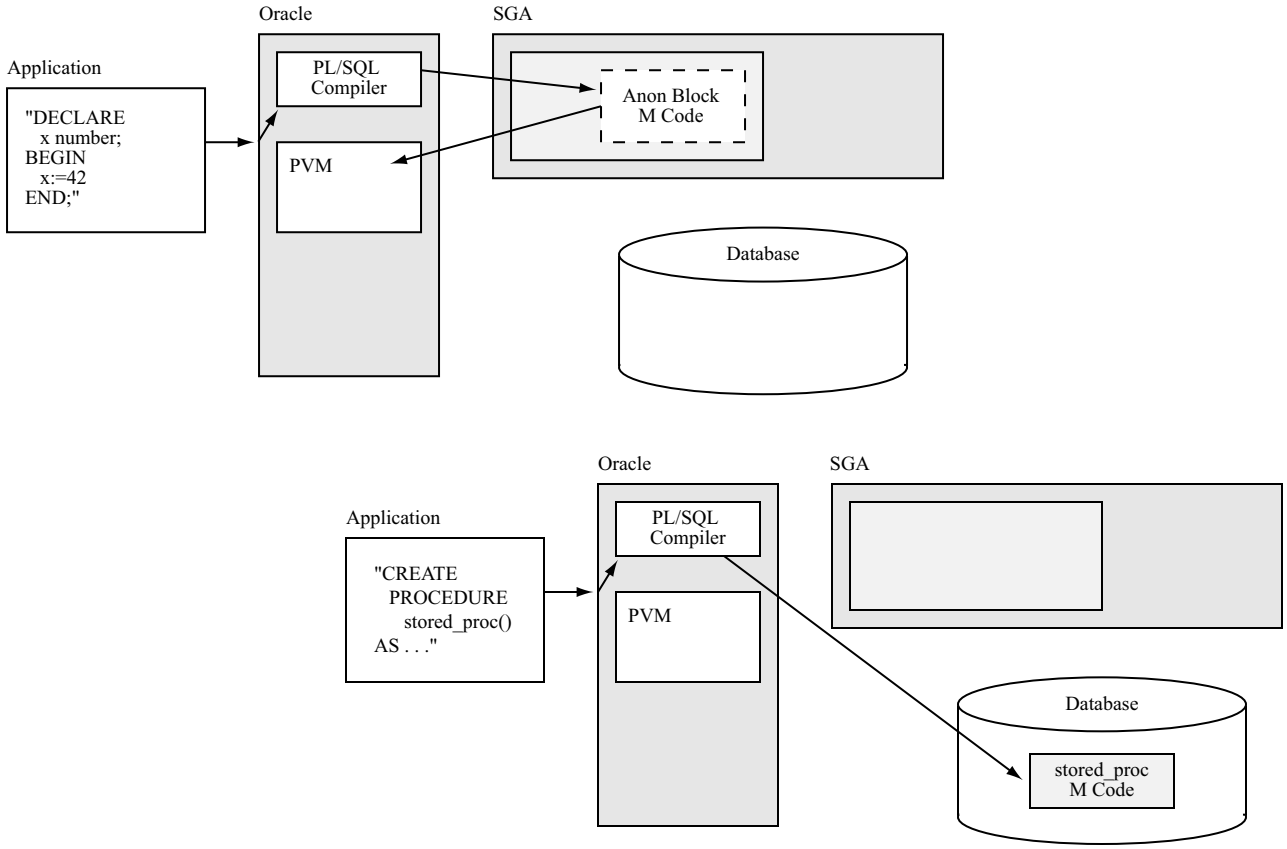
PL/SQL Execution

- PL/SQL programs are executed by Oracle's PL/SQL Virtual Machine (PVM).
  - The PVM is sometimes referred to as the *PL/SQL Executor* or the *PL/SQL Engine*.
- Your PL/SQL program's source code is first processed by the PL/SQL Compiler.
  - The compiler parses your program, and produces an equivalent set of PVM instructions (sometimes referred to as *MCode*).
  - The compiled MCode can be run immediately by the PVM (for example, when you run an anonymous PL/SQL block in SQL*Plus).
  - The compiled MCode can be stored in the database for later execution (such as when you run a **CREATE PROCEDURE**, **CREATE PACKAGE**, **CREATE TRIGGER**, or **CREATE FUNCTION** statement.)
- The PVM behaves like a computer chip, executing the sequence of numeric instructions (MCode) derived from your PL/SQL source code.
  - This is similar to the way Java programs work: Java source code is compiled into Java byte-code, which is run by the Java Virtual Machine JVM).
  - The PVM is, of course, highly optimized for programs that access the Oracle Database.
- The PVM is built into the Oracle Server.
  - It's also built into certain client-side applications, such as Oracle Forms.

Most commonly, PL/SQL is executed on the Oracle Server, as the result of a call to a stored procedure, stored function or, a package procedure or function; or when a trigger is fired as the side effect of some event in the database. In these cases, the pre-compiled PL/SQL has been stored in the database (in **SYS**-owned tables in the Data Dictionary, in the **SYSTEM** tablespace). Once read from the database and executed, the code remains in server memory in case it is called again, by the same user or any other user.



An anonymous PL/SQL block, such as one entered into SQL*Plus, or contained in a client application, is also executed on the Oracle Server. Such anonymous PL/SQL blocks must be parsed and compiled, and the resulting MCode is not stored in the database. However, the compiled MCode remains in server memory, in case the exact same block is executed again.



The PL/SQL Compiler

❋ To compile your program, the PL/SQL compiler:

1. Parses your source code:

   - Checks for syntactic validity (source code keywords and tokens used correctly.)

   - Checks for semantic validity (database objects referenced in the source code exist and are used correctly.)

2. If successful, parsing results in a temporary, internal representation of your PL/SQL program's semantics.

   - The parser (or compiler *front end*) generates this semantic description of your program, in the language DIANA (Descriptive Intermediate Attributed Notation for Ada), which will be used be the compiler *back end* in the next step.

3. From its internal representation, the compiler generates a binary, executable form of your program, consisting of numeric machine instructions.

   - Normally, this is MCode: instructions for the PL/SQL Virtual Machine.

   - Alternatively, you can have the compiler generate native machine code for the actual CPU hardware of the platform on which the program will run.

4. When the source code is part of a **CREATE** statement, the resulting executable code is stored in the Data Dictionary until it needs to be executed.

   - If the source code is a dynamically-created anonymous block (such as one executed in SQL*Plus), it is executed immediately and not stored.

When Oracle developed the PL/SQL language (the first release was in 1991, as part of Oracle Version 6), they modeled many of the language features after the existing programming language Ada, including:

- Basic program structure and syntax
- Exception propagation and handling
- **RECORD** types
- Package structure
- Procedures and functions
- Parameter modes

Compiler Optimization

- During the code-generation phase, the compiler performs significant optimization of your PL/SQL program.
  - In doing so, the compiler may reorganize your code in several ways.
  - These optimizations preserve the behavior of your program.
- The compiler has several areas available for optimization.
  - Within expressions (when operators have equal precedence), operator evaluation can be arranged in any order.
  - For many operators, the operands can be evaluated in any order.
  - In a function or procedure call, the arguments can be evaluated in any order.
  - In a complex expression with many possible evaluation orders, the evaluations of various pieces can be intermixed.
  - When a package member is referenced, and the package hasn't been initialized for the calling session, the compiler can decide whether or not the reference requires package initialization.
  - If a statement can raise any of several exceptions, Oracle can choose which of the exceptions to raise.
  - If one way of performing an operation may raise an exception, while a different way of performing it wouldn't, Oracle is not required to raise the exception.

As of Oracle10*g*, the PL/SQL compiler automatically performs optimization during code generation, potentially rearranging the statements expressed in your source code. Some are the kind of optimizations other language compilers (such as those for C and C++) do; others are very specific to Oracle. Exactly which reoganizations the compiler performs will depend on the actual program. Consider this contrived example:

```
DECLARE
  x NUMBER := 17;
  y NUMBER := 42
  z NUMBER = 1;
BEGIN
  FOR num IN 1..1000 LOOP
    z := (x / y) + (x / y);
    INSERT INTO sometable (somecolumn)
           VALUES (z);
  END LOOP;
END;
/
```

The evaluation of **z := (x / y) + (x / y)** is performed in each loop iteration, even though the results are the same each time. The statement could be moved above the loop, and be evaluated only once. Furthermore, the value of **(x / y)** could be determined once, and the result reused (as though there were a temporary variable **w**), used like this: **w := (x / y); z := w + w;**. The compiler will perform this sort of optimization automatically. While such poor coding practices are easy to see in a tiny sample program, they (and more subtle variations) can be easily missed in a larger PL/SQL application.

Prior to Oracle10*g*, making such changes yourself in your source code could yield better-performing programs. Now the compiler does these and many other optimizations for you. In the very unlikely event you wish to reduce the amount of optimization performed, you can set the parameter **pl_sql_optimize_level;** the values are:

2: (the default) Perform all optimizations, considering your program as a whole and how it interacts with other PL/SQL programs it calls.
1 Perform only local optimizations within individual branches of the program code.
0: Do compilation as it was done in Oracle9*i*, with (for example) expressions and procedure arguments evaluated (more or less) left-to-right.

```
ALTER SESSION SET pl_sql_optimize_level = 0
```

You would likely only do this while debugging code being ported from an earlier release to 10*g*.

According to Oracle, PL/SQL programs run twice as fast under Oracle10*g* as they did under Oracle9*i* (which itself was generally about 1.5 times faster than Oracle8*i*). Part of this is due to code optimization and part due to improvements in the PVM.

SQL – Parse

- SQL statements are always executed on the Oracle Server, which is, of course, the primary function of Oracle.
- The SQL Statement Executor processes a SQL statement in several phases, Parse (Soft or Hard), Execute, and Fetch.
- Soft Parse:

  1. Allocate a Private SQL Area for the statement.
  2. From the exact text of the SQL statement, generate a unique hash code which identifies that statement.
  3. Parse the SQL statement to determine if it's syntactically correct.
  4. Parse the statement for semantic correctness (tables and columns found, current user has appropriate privileges, etc.)
  5. Check to see if exactly the same (syntactically, semantically, and environmentally) statement, identified by that hash code, has already been saved in server memory; if not, Oracle will need to perform a Hard Parse.

- Hard Parse:

  1. Allocate a Shared SQL Area in server memory for this statement.
  2. Generate an optimized execution plan for the statement.
  3. Generate an internal, executable representation of the statement based on the plan.
  4. Store the plan and the statement in the Shared SQL Area, identifying it with the hash code.

The Private SQL Area for a statement holds information used for the current execution of the statement. This includes runtime information during the actual execution of the statement (temporary memory for sorting, data structures used in joins, etc.), as well as persistent information about the statement (bind information, etc.) needed by the current session. The runtime information isn't needed once execution is finished (a DML statement completes, a query's last row is fetched). The persistent information may remain available longer — until you close the **CURSOR** for a query, for example.

In fact, a PL/SQL **CURSOR** is your program's reference to a specific Private SQL Area. Cursor attributes (**%ISOPEN**, **%FOUND**, **%ROWCOUNT**, etc.) allow your PL/SQL code access to specific information from the Private SQL Area for a statement. When your PL/SQL executes DML without creating an explicit **CURSOR**, you can use the predefined identifier **SQL** to refer to the SQL Area for the most recent statement.

The Shared SQL Area holds the parse tree (internal, executable representation) and the execution plan of the statement. These can be re-used by anyone who executes that same statement, saving the time needed for hard parsing. Hard parsing includes SQL statement optimization by Oracle's Cost-Based Optimizer (CBO). The CBO determines how best to access the rows affected by the query — which table to access first in a join, how to perform the join, which indexes to use in which order, etc. It makes these choices based on *metadata* (presence and types of indexes, types of tables, etc.) and on *statistics* about the data itself (number of rows, average row length, etc.) stored in the Data Dictionary. For effective optimization, statistics must be collected periodically. Normally, the DBA manages statistics collection, though you can manage the statistics on your own tables if you wish. Changes to the database or statistics which affect a statement's tables will require the statement to be hard parsed again.

Properties and statistics of Oracle's memory structures and other important internal information are visible in real time by querying the dynamic performance views (the **v$** views). By default, these are visible only to the DBA account. For tuning and diagnostic purposes, the DBA can grant access to these views to non-DBA, developer accounts:

```
GRANT select_catalog_role TO some_user;
```

The Shared Pool is one of the main components of Oracle's SGA. It holds many data structures used internally by Oracle's server processes and background processes, only a few of which are important to us at this point. You can see them in the view **V$SGASTAT**.

SQL – Execute and Fetch

- Execute:
  - For a query, generate the result set.
    - That is, identify the exact rows which satisfy the query.
  - For DML, perform the data manipulation.
    - This may require data to be retrieved from data files into server memory.
- Fetch (if the statement is a query):
  - If the program executed a **SELECT INTO...** statement, copy the column values of the result set row into the given program variables.
    - If there is no row in the result set, throw **NO_DATA_FOUND**.
    - If the result set contains more than one row, throw **TOO_MANY_ROWS**.
  - If the program opened a cursor, as it performs each **FETCH ... INTO ...** (or the equivalent), copy column values of the current result set row into the given program variable(s), and advance the cursor's row pointer to the next row.
- Close — Release the private SQL area.

The SQL statement optimizer's goal is to return all rows needed for your statement, using minimal throughput (I/O and other resource consumption). You can ask the optimizer instead to focus on returning the first few rows, or the first 1, 10, 100, or 1000 rows, as quickly as possible, even if that will increase the overall throughput needed to complete the entire statement.

```
ALTER SESSION SET optimizer_mode = first_rows;
                                              /* first "few" rows fast */
ALTER SESSION SET optimizer_mode = first_rows_1;     /* first row fast */
ALTER SESSION SET optimizer_mode = all_rows;
                                 /* default: minimum overall throughput */
```

Oracle reads and writes data from data files one block at a time. A *block* is a unit of storage whose size is fixed for any given tablespace. A block may be 2, 4, 8, 16, or 32 kilobytes, and typically holds many rows. To fetch a row, the block containing that row must first be read into server memory. Once read, it remains in memory so that other rows in that block are available for future SQL operations. Reading a block from its data file into memory is a *physical get* or *physical read*.

When fetching rows for a query, Oracle must read the corresponding blocks that are in memory. Since all rows for a query must reflect the state of the blocks as they were at the beginning of the query, Oracle must get the correct version of each block (multiple versions of a block may exist, with each transaction potentially seeing different versions). This includes blocks containing index entries necessary to process the query. Each read of a block in a set of blocks of the same version is called a *consistent get*.

When performing updates or deletes, Oracle must get the current version of the relevant blocks — that is, the blocks as they appear as of the most recently-committed DML affecting those blocks. These are called *current reads* (also referred to as *db block gets*).

Consistent gets and db block gets are both forms of *logical read*. A physical read is more costly (that is, slower), by orders of magnitude, than a logical read. The greater the ratio of logical reads to physical reads necessary to process a statement, the faster the statement is likely to execute. This ratio, called the *buffer cache hit ratio*, can be calculated as:

1 - ((physical reads) / (consistent gets + db block gets))

SERVER MEMORY

- When an Oracle instance starts up, several background processes run.
  - These manage critical tasks and services of the Oracle database, such as storing data properly in the data files, recovering from crashes, etc.
- At startup, the instance allocates a very large region of shared memory Oracle calls the System Global Area (SGA).
  - The SGA is shared by all users of the database.
- Programmers need to familiarize themselves with some critical components of the SGA:
  - The Buffer Cache stores blocks of data containing the rows needed by a statement, which must first be read from their data files.
    - Any subsequent statement that accesses the same rows, or other rows in the same blocks, will find them already in memory.
  - The Shared Pool of the SGA contains the Dictionary Cache, which contains rows of metadata that have been read from the Data Dictionary.
    - This is read, for example, when parsing the semantics of a SQL statement.
  - The Shared Pool also contains the Library Cache, which itself contains:
    - Shared SQL Areas for recent statements.
    - Compiled PL/SQL procedures, functions, triggers, packages, anonymous blocks, and Java classes.
  - The Shared Pool also contains latches, locks, and other data structures.

The SGA contains many regions and data structures used for the vast variety of Oracle operations. You can find the current sizes of all of these in the view **V$SGASTAT**. For PL/SQL developers, the most important SGA components include the Buffer Cache and the Shared Pool.

The Buffer Cache holds working copies of data file blocks, containing table rows, index entries, and other database data.

The Shared Pool contains many structures. Of interest to developers are:

- Dictionary Cache

  To process a SQL statement, Oracle must look up information about the tables and other objects referenced in the statement in the Data Dictionary. Oracle uses additional SQL statements, called *recursive SQL*, to retrieve this information. Recursive SQL is also used in processing DDL statements and when performing space allocation. Once retrieved, this information is cached in the Dictionary Cache. The Dictionary Cache stores the individual rows of this information, and for this reason is also called the *Row Cache* (do not confuse this with rows of actual database data, residing in blocks in the Buffer Cache). Recursive SQL is not needed when the required information is already in the Dictionary Cache.

- Library Cache

  Compiled PL/SQL programs, SQL statements, and Java classes are cached in the Library Cache. Recently-run SQL statements and PL/SQL blocks are in the SQL Area of the Library Cache. Each is identified by its SQL ID, a hash code derived from the exact text of the statement. Along with the text of the statement, the Shared SQL Area of the statement includes the statement's compiled form and execution plan (along with other data and statistics for the statement). Since two sessions may execute the exact same statement under different conditions (such as optimizer mode) which result in different execution plans, there may be multiple execution plans for a single statement in the SQL Area.

  You can see statements in the Library Cache by querying **V$SQL**.

sqlarea.sql

```
SELECT  sql_id,  parsing_schema_name,  sql_text
  FROM v$sql
/
```

Latches

- A *latch* is a simple data structure in memory whose value indicates whether or not its associated resource is currently in use.
- Before using a shared memory resource, each Oracle process first obtains the latch for that resource, then accesses the resource.
  - A latch can be held by only one process at a time.
- If the latch is already taken, your Oracle process may:
  - Spin (*active wait*) — Just keep trying, as fast as it can, until the latch is free.
  - Sleep — Wait a few fractions of a second, then try again.
  - Not wait — Abandon the operation.
- Latches are very efficient, and are typically held only briefly.
  - However, the processing of a statement may require acquisition of many latches.

There are hundreds of different latches, for different shared resources in the SGA. In most cases, each latch protects a single resource. In some cases a latch may have associated child latches, protecting different parts of a data structure.

Latches allow for orderly, serialized (one-at-a-time) access to shared resources by any number of different Oracle processes. They are compact and very efficient, using a single atomic machine instruction to check or acquire the latch. However, it's possible for an application that works fine under single-user access, or small-scale testing, to encounter *latch contention* — processes spinning or waiting an inordinate amount of time — under heavy loads.

Oracle uses latches internally to protect hundreds of different resources. Statistics on their use is available in the dynamic performance view **V$LATCH**:

latches.sql

```
SELECT name, gets, misses, sleeps, immediate_gets, spin_gets
  FROM v$latch
 ORDER BY gets
/
```

LOCKS

- A *lock* is a complex data structure in shared memory that allows a session or transaction to wait for a resource to become free.
  - A lock uses a complex data structure, allowing sessions to add themselves to a linked list, or *queue*, to wait on the lock.
  - Multiple sessions may interact with a lock, so each must first obtain the latch protecting the lock before making its change to the lock.
    - The session frees the latch immediately after modifying the lock.
  - Another name for a lock is an *enqueue*.
- When you update database rows, your session:
  - Creates a lock in shared memory for your transaction — a Transaction (TX) lock.
  - Flags the rows as locked, placing information identifying your transaction in the block containing the rows.
  - Creates one lock in shared memory for each object referenced by the SQL statement — DML (TM) locks.
- When another transaction tries to update any of the rows you have locked, it:
  - Identifies your transaction using information in the rows' block.
  - Creates its own lock structure.
  - Adds its lock structure to your lock's "waiters" queue.

Oracle uses hundreds of types of lock. Most of these are used internally to coordinate activity of Oracle background processes during various normal operations. The lock types relevent to programmers are Transaction (TX) locks, and DML (TM) locks. When you start a transaction, your session creates a single TX lock, which allows other sessions to wait on yours. If another session tries to update one of the same rows, it enqueues on the TX lock. During your transaction, your session also obtained a TM lock for each table you updated or other object you used. Because of this, no other session will alter those objects until those locks are free. For example, if another session attempts an **ALTER TABLE** on one of the tables you're updating, it will enqueue on the TM lock.

When a transaction updates rows in a block, it adds itself to the block's Interested Transaction List (ITL), a small data structure in the block header. It then updates the row directory, another small data structure in the block header, with a 'locked' flag for each row being updated. The flag points to the transaction's ITL entry, which points (indirectly) to the transaction itself. Another transaction that wishes to update those same rows will find them marked as locked in the row directory. The lock flag will point to the ITL and thus the original transaction, which allows the second transaction to locate the TX lock of the first transaction and enqueue as a waiter. You can see currently-held locks in the view **V$LOCK**:

```
SELECT * FROM v$lock WHERE type IN ('TX','TM');
```

A different presentation of the same information is available from **DBA_LOCKS**:

```
SELECT * FROM dba_locks WHERE lock_type IN ('Transaction','DML');
```

For a TM lock, the **ID1** field is the numeric object ID of the locked table. Using this you can find the locked table in **DBA_OBJECTS** or **ALL_OBJECTS**. The mode of the TM lock determines what operations other transactions can perform on the entire table:

Row Share (SS or RS) lock: Other transactions can **SELECT**, **INSERT**, **UPDATE**, **DELETE**, and lock other rows in the table. However, another transaction cannot get an exclusive lock on the table, for example, to **ALTER** or **DROP** the table, while the first transaction has the RS lock.

Row Exclusive (SX or RX) lock: Other transactions can **SELECT**, **INSERT**, **UPDATE**, **DELETE**, and lock other rows in the table. However, another transaction cannot get an exclusive or a share row exclusive table lock (another kind of lock, more restrictive than a row share table lock) on the table.

The mode of the TX lock determines the lock mode for the rows. An **INSERT**, **UPDATE**, or **DELETE** establishes an exclusive (RX) lock on the rows. A **SELECT ... FOR UPDATE** establishes a share (RS) lock on the rows. If, after a **SELECT ... FOR UPDATE**, you later actually update the rows within the same transaction, Oracle automatically converts the share (RS) lock to an exclusive (RX) lock.

See the Oracle Concepts guide, "Data Concurrency and Consistency," for more information on lock types and lock modes.

LABS

Labs in this chapter assume that a student Oracle account has been set up and granted **SELECT ANY TABLE**, **UPDATE ANY TABLE**, and **SELECT_CATALOG_ROLE**.

❶ Query **V$SGASTAT**. Find the current sizes of the Library Cache, the Row Cache, the SQL Area, and the PL/SQL portion of the SQL Area.
(Solution: *sgastats.sql*)

❷ Querying **V$SQL**, find the statements that were originally issued under your current Oracle user name.
(Solution: *usersql.sql*)

❸ Query both **V$LOCK** and **DBA_LOCKS**, looking only for TX and TM locks. Are there any transactions currently performing DML?
(Solution: *locks1.sql*)

❹ Run an **UPDATE** statement to update the **OE.CUSTOMERS** table, setting **ACCOUNT_MGR_ID** to **148** — do not commit (you will rollback at the end of this exercise!). Query **V$LOCK** and **DBA_LOCKS**, looking only for TX and TM locks. What locks were created? Do not **COMMIT**, **ROLLBACK**, or exit yet.
(Solution: *locks2.sql*)

❺ From the output of the **DBA_LOCKS** query with your **UPDATE** transaction still open, identify the object IDs of all locked objects. Using this information, look the locked table or tables up in **DBA_OBJECTS**. What tables are locked? What mode is each locked in? Why? Do not **COMMIT**, **ROLLBACK**, or exit yet.
(Solution: *locked_objects.sql*)

❻ In a second session, run a new update statement to update the **ACCOUNT_MGR_ID** to **148** where the **NLS_TERRITORY** is equal to '**JAPAN**'. What happens (or doesn't happen...)? In your first session, query **V$LOCK** and **DBA_LOCKS**. What TX and TM locks are there now? Which are held, which are requested, which are blocking? Now **ROLLBACK** your first session, and examine **V$LOCK** and **DBA_LOCKS** again. What locks exist? Now **ROLLBACK** your second session, and examine **V$LOCK** and **DBA_LOCKS** again. Are there any TX or TM locks left from your transactions?
(Solutions: *wait_session1.sql*, *wait_session2.sql*).

CHAPTER 5 - OBJECT-ORIENTED ORACLE

OBJECTIVES

- Describe the different object types in Oracle.
- Understand the advantages in using object types.
- Create PL/SQL code that contains different object types.

INTRODUCING OBJECT-ORIENTED ORACLE

- Oracle's Object Relational Model is similar to Object-Oriented Programming (OOP).
  - Your can store an object in a table, query the object, and enjoy all the other database features standard relational tables benefit from.
- An *object type* in Oracle is a user-defined datatype composed of standard datatypes and/or other user-defined datatypes.
  - Object types let you extend the standard datatypes with more complex representations of real-world items.
- An *object table* in Oracle is a table where each row represents one object.
  - An object table only has one column — the object type.
- Oracle's object-relational features lend themselves nicely to working with OO programming languages.
  - No mapping layer is required between objects in the database and objects in the application code.
  - Reusability of objects makes application development more efficient.
- It is easy to model complex, real-world business entities and logic.
- Objects encapsulate operations along with the data.
- As a disadvantage, it can be more cumbersome to work with user-defined datatypes, especially in non-OO programming languages.

As an example, a car's attributes can include **make**, **model**, **year**, and **engine**. For the **car** object example, the **engine** attribute could be an object itself, with attributes of **ignition** and **num_cylinders**.

The following table shows the owner of the car in one column, and the car object type in the second column.

Owner	Car
Bob	Ford, Mustang, 2006, (Fuel Injected, 8)
Sue	Chevrolet, Camaro, 1966, (Carburated, 8)
Sally	Nissan, 300ZX, 1988, (Fuel Injected, 6)

The following object table contains objects representing classes at a university. Each row holds an object representing a university class.

Classes
Math 101, Algebra I, 3.0
Engl 203, Advanced Composition, 3.0
Phys 102, Physics II, 4.0

Note:
Though not technically accurate, some resources may describe an object table as any table in which any column is declared with a user-defined datatype.

Comparison of Object-Oriented vs. Relational design

	Object-Oriented	Relational
Definition of attributes	Can reuse objects for other definitions.	Must explicitly declare each new attribute.
Storage of attributes	Each column is an object, with potentially multiple attributes.	Each column is one and only one attribute.
Functions	Functions can be directly associated with the data.	Functions stand alone, not tied to any table.
Join operations	Not able to join one object to another.	Can join attributes in two tables.
Interaction with application objects	Interact directly with objects in OOP languages.	Requires mapping between attributes in the table and attributes in the object.
Interaction with SQL	Requires special techniques to use SQL on objects.	Native interactions with SQL statements.

DEFINING OBJECT TYPES AND TABLES IN SQL

- The **CREATE TYPE...AS OBJECT** statement is used to create a user-defined object datatype, specifying its attributes.

```
CREATE TYPE engine_typ AS OBJECT (
  cylinders NUMBER,
  manufacturer VARCHAR2(30)) NOT FINAL;
```

  - You must have been granted the **CREATE TYPE** or **CREATE ANY TYPE** system privilege.
  - Specify **NOT FINAL** to allow inheritance from this type, as **FINAL** is the default.

- Once the user-defined datatype has been created, create a table using the datatype name just like you would use **CHAR**, **NUMBER**, **DATE**, etc.

- The type definitions are stored in the Data Dictionary, accessible through the **DBA_TYPES** view.
  - The **DESCRIBE** command in SQL*Plus can be used to easily show the type definition.

- Once the type is created, it can be used in a **CREATE TABLE** statement just like any standard datatype.

```
CREATE TABLE vehicles (
  vehicle_id NUMBER,
  date_of_service DATE,
  characteristics car_typ) NOT FINAL;
```

- To create an object table, with the object type as its only column, you can use the **OF** clause.

```
CREATE TABLE car_characteristics OF car_typ;
```


We can create a user-defined datatype called **car_typ**, which is comprised of simple attributes, as well as the **engine_typ** type that we create first.

engine_car.sql

```
CREATE TYPE engine_typ AS OBJECT (
   cylinders NUMBER,
   manufacturer VARCHAR2(30)) NOT FINAL;

CREATE TYPE car_typ AS OBJECT (
   make VARCHAR2(30),
   model VARCHAR2(30),
   year CHAR(4),
   engine engine_typ) NOT FINAL;
```

The **DESCRIBE** command in SQL*Plus will show you the attributes of the user-defined datatype. The **SET DESCRIBE DEPTH ALL** command will let you see all levels of the object types.

```
SQL> set describe depth all
SQL> desc car_typ
car_typ is NOT FINAL
 Name                 Null?    Type
 -------------------  --------  ------------
 MAKE                           VARCHAR2(30)
 MODEL                          VARCHAR2(30)
 YEAR                           CHAR(4)
 ENGINE                         ENGINE_TYP
   CYLINDERS                    NUMBER
   MANUFACTURER                 VARCHAR2(30)
```

Once the type is created, you can create a tables with the user-defined datatype.

vehicle.sql

```
CREATE TABLE vehicles (
   vehicle_id        NUMBER,
   date_of_service   DATE,
   characteristics   car_typ);
```

car_char.sql

```
CREATE TABLE car_characteristics OF car_typ;
```

Try It:

Perform a **DESCRIBE** on the **car_characteristics** object table to show all of the attributes.

QUERYING AND MODIFYING OBJECT DATA

- Query the individual attributes of the user-defined datatype with the ***table.column.attribute*** notation.

```
SELECT v.characteristics.make
  FROM vehicles v;
```

  - The table or its alias must appear in the **SELECT** clause, otherwise the **ORA-00904: invalid column name** error will appear.

  - The object values will be wrapped in the object type name.

- When inserting a row of data, wrap the object's attributes in the object name.

```
INSERT INTO vehicle
  VALUES (1001, '08-MAR-2005',
  car_typ('FORD', 'F-150', '2005', engine_typ(8, 'Triton'))
);
```

- You can update the attribute of an object type directly by using the ***table.column.attribute*** notation.

```
UPDATE vehicle v
   SET v.characteristics.model = 'IMPALA'
 WHERE vehicle_id = 1002;
```

- When you delete a row, the object is automatically destroyed.

vehicle_data.sql

```
INSERT INTO vehicle VALUES (
  1001, '08-MAR-2005',
  car_typ('FORD', 'F-150', '2005', engine_typ(8, 'Triton'))
);

INSERT INTO vehicle
  VALUES (1002, '17-FEB-2006',
  car_typ('CHEVROLET', 'MALIBU', '2003', engine_typ(6, 'Chevrolet'))
);

UPDATE vehicle v
   SET v.characteristics.model = 'IMPALA'
 WHERE vehicle_id = 1002;

SELECT * FROM vehicle;

SELECT vehicle_id, v.characteristics.make, v.characteristics.model
  FROM vehicle v;
```

OBJECT METHODS

- Functions associated with an object type are called *member methods*.
  - Declare a method when creating the user-defined object datatype.
  - Provide the method code by writing a **BODY** for the datatype.
    - This is similar to writing a package specification and body.
- Member methods interact with the object's attributes.
  - Member methods have a built-in reference called **SELF**, which points to the current instance of the object.
- *Constructor* methods instantiate the object by giving values to the attributes.
  - An *attribute value constructor* is defined by Oracle and can accept literals for each of the attributes.
  - You may provide additional constructors by writing methods that have the same name as the datatype.
- You can define a **MAP** method to be used in equality comparisons.
- You can define an **ORDER** method to be used in sorting operations.
  - The method needs to return a negative, positive, or zero value for an object that is less than, greater than, or equal to the current object.
- Methods marked **STATIC** operate on the object type, not the individual object.
- Additional methods can be written to manipulate the individual object.
  - These are called *instance methods* and can reference the object's attributes.

The following version of our object type defines a method that can be used for sorting the objects.

car2.sql

```
CREATE OR REPLACE TYPE car_typ2 AS OBJECT (
  make VARCHAR2(30),
  model VARCHAR2(30),
  year CHAR(4),
  engine engine_typ,
  ORDER MEMBER FUNCTION ordering (c car_typ2) RETURN INTEGER
) NOT FINAL;
/

CREATE OR REPLACE TYPE BODY car_typ2 AS
  ORDER MEMBER FUNCTION ordering (c car_typ2) RETURN INTEGER IS
  BEGIN
    IF (year < c.year) THEN
      RETURN -1;
    ELSIF (year > c.year) THEN
      RETURN 1;
    ELSE  -- years are equal, sort by the makes
      IF (make < c.make) THEN
        RETURN -1;
      ELSIF (make > c.make) THEN
        RETURN 1;
      ELSE  -- same year and make, they sort equal
        RETURN 0;
      END IF;
    END IF;
  END;
END;
/
```

Now, if we create a **vehicle2** table with a **car_typ2** field, we can perform a sort on the object field. The Oracle engine will automatically call our method marked with **ORDER** to perform the **ORDER BY**.

vehicle2.sql

```
...
  SELECT v.characteristics.year year, v.characteristics.make make
    FROM vehicle2 v
ORDER BY v.characteristics;
```

Notice that the data was sorted by year and then make, as indicated by the ordering function.

```
YEAR MAKE
---- ------------------------------
2003 CHEVROLET
2005 FORD
```

INHERITANCE

- *Type inheritance* means that any user-defined datatype inherits, or takes over, features from any parent types.
  - Type inheritance provides a higher level of abstraction for modeling complex business entities.
  - The parent type is called the *supertype*.
  - The child type is called the *subtype*.
    - Due to type inheritance, subtypes automatically acquire any changes made to its supertypes.
- To inherit from an existing user-defined type, specify the **UNDER** keyword along with the supertype.

```
CREATE TYPE convertible_typ UNDER car_typ (
  shell VARCHAR2(4),
  retraction VARCHAR2(6));
```

- You cannot inherit from a **FINAL** type.
- *Attribute chaining* is the succession of attributes from a parent object type to its child object type.
  - The **UNDER** keyword causes all of the attributes of the supertype to exist in the subtype as well.

A subtype of a car might be a convertible, that has either a hard or soft shell top, and manual or power retraction.

convertible.sql

```
CREATE TYPE convertible_typ UNDER car_typ (
   shell  VARCHAR2(4),
   retraction  VARCHAR2(6));
```

conv_tab.sql

```
CREATE TABLE conv OF convertible_typ;
```

A **DESCRIBE** of the **conv** table shows that it has all of the attributes of the supertype, **car_typ**, as well as the attributes specified in the subtype, **convertible_typ**.

```
Name                                      Null?    Type
----------------------------------------- -------- --------------------
MAKE                                               VARCHAR2(30)
MODEL                                              VARCHAR2(30)
YEAR                                               CHAR(4)
ENGINE                                             ENGINE_TYP
SHELL                                              VARCHAR2(4)
RETRACTION                                         VARCHAR2(6)
```

TYPE EVOLUTION

- *Type evolution* is the process of changing the object type with the **ALTER TYPE** command.
- Several operations can be performed to change an existing user-defined datatype.
  - Add or remove attributes.
    - Any dependent tables receiving a new attribute will store **NULL** in that column.
    - Any dependent tables that have an attribute removed will have their columns removed as well.
  - Add, remove or modify methods.
  - Change a numeric attribute to increase its length, precision, or scale.
  - Change a variable character attribute to increase its length.
  - Change the finality of the type (**FINAL** or **NOT FINAL**).
- Due to inheritance and attribute chaining, any objects that reference this type will be affected by the change.
  - Any dependent views, operators, index types, and stored PL/SQL blocks will be marked **INVALID**.

The following example shows the **car_typ2** type receiving a new attribute to store the color of a car.

car2_evolve.sql

```
ALTER TYPE car_typ2
  ADD ATTRIBUTE (color VARCHAR2(15)) CASCADE;
```

```
SQL> DESC car_typ2
 car_typ2 is NOT FINAL
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------
--------
 MAKE                                               VARCHAR2(30)
 MODEL                                              VARCHAR2(30)
 YEAR                                               CHAR(4)
 ENGINE                                             ENGINE_TYP
 COLOR                                              VARCHAR2(15)

METHOD
------
 ORDER MEMBER FUNCTION ORDERING RETURNS NUMBER
 Argument Name                  Type                    In/Out Default?
 ------------------------------ ----------------------- ------ --------
 C                              CAR_TYP2                IN
```

OBJECT VIEWS

- If you have an existing relational database schema, without any fields or tables stored as objects, you can still treat this data as objects from an OO application.
  - An object view is created as a wrapper around the existing relational data.
    - The OO application can then utilize the view without changing the relational database model.
- The object view is created in two steps:

1. Create an object datatype to be used by the OO application.

```
CREATE TYPE region_typ AS OBJECT (
  region_id NUMBER,
  region_name VARCHAR2(25));
```

2. Create an object view based on this datatype, which queries the underlying relational tables.

```
CREATE VIEW region_obj_view OF region_typ
  WITH OBJECT IDENTIFIER (region_id) AS
  SELECT region_id, region_name FROM regions
```

- Each row in the object view is considered an object.

Before an object view can be created, we must first create a user-defined type. Our example will create an object view on the **hr.regions** table, so we must first create an object type which resembles this table. We can then create a view based on the object type, and the underlying table.

region.sql

```
CREATE TYPE region_typ AS OBJECT (
  region_id NUMBER,
  region_name VARCHAR2(25))
/

CREATE VIEW region_obj_view OF region_typ
    WITH OBJECT IDENTIFIER (region_id) AS
  SELECT region_id, region_name FROM hr.regions
/
```

This view can now be used in the same way that a true object table, such as **car_characteristics**, can be used. An object-oriented application can query an object view, based on relational tables, the same way that it can query an object table.

```
SQL> SELECT * FROM region_obj_view;

 REGION_ID REGION_NAME
---------- -------------------------
         1 Europe
         2 Americas
         3 Asia
         4 Middle East and Africa
```

OBJECT TYPES IN PL/SQL

- Object types in PL/SQL can be used wherever standard datatypes can be used.
  - Variables in PL/SQL can be defined with object types.
  - Parameters in PL/SQL functions and procedures can use object types.
  - Functions can return object types as well.
- Scope and instantiation rules hold for object types.
  - Local objects are instantiated when you enter the PL/SQL block and destroyed when you leave the block.
  - Objects local to the block can only be seen inside that block.
- An object in PL/SQL is automatically **NULL** until you call its constructor to initialize the object.
  - The object itself is **NULL**, not just its attributes.
- Attributes of the object can be referred to individually with the ***variable.attribute*** notation.

In the following example, we use the **car_typ** datatype and its constructor method in an anonymous block.

anon_block.sql

```
SET SERVEROUTPUT ON
DECLARE
  -- Create a variable (initally empty) of CAR_TYP
  car car_typ;
BEGIN
  -- Call the constructor method to initialize car
   car := car_typ('Nissan','Maxima','2004', engine_typ(6,230));
  -- Add the car to the vehicles table
  INSERT INTO vehicle VALUES (1003, SYSDATE, car);
  DBMS_OUTPUT.PUT_LINE(car.make || ' : ' || car.model);
END;
/
```

Note that the **DBMS_OUTPUT** line refers to the individual attributes of the object.

The **car_typ** datatype can be used as an input parameter, as shown in this stored procedure.

add_vehicle.sql

```
CREATE OR REPLACE PROCEDURE add_vehicle (car_id NUMBER, car car_typ)
AS
BEGIN
  INSERT INTO vehicle VALUES (car_id, SYSDATE, car);
END;
/
```

REF POINTERS

- A **REF** is a logical pointer, or reference, to an object.
  - A **REF** can be used to examine or update the object to which it refers.
  - A common use for a **REF** is to implement a foreign key to another object.
- The **DEREF** function will follow the **REF** pointer and return the object being pointed to.
- An object that is deleted, may have **REF**s still pointing to it.
  - Any **REF** that is no longer pointing to a valid object is a *dangling **REF***.
  - The **IS DANGLING** operator can be used to determine which entries in a table have a dangling **REF**.

In the following example, we create a table called **fleet** to represent a fleet of vehicles for our company. This table will contain pointers to the **car_characteristics** table. The **car_ref** column is a **REF** pointing to a **car_typ** object, limited to the **car_characteristics** table. We will then insert two rows of information using a **REF** to a specific row in the **car_characteristics** table.

fleet.sql

```
CREATE TABLE fleet (
   department VARCHAR2(20),
   car_ref REF car_typ SCOPE IS car_characteristics)
/

INSERT INTO fleet SELECT 'PAYROLL',
   REF(c) FROM car_characteristics c WHERE c.make = 'CHEVROLET'
/

INSERT INTO fleet SELECT 'MOTOR POOL',
   REF(c) FROM car_characteristics c WHERE c.make = 'FORD'
/
```

Querying the **fleet** table, we can see the reference pointer, and then use **DEREF** to see the row from the **car_characteristics** table.

```
SELECT department, car_ref, DEREF(car_ref) FROM fleet;
```

```
DEPARTMENT
--------------------
CAR_REF
--------------------------------------------------------------------------------
DEREF(CAR_REF)(MAKE, MODEL, YEAR, ENGINE(CYLINDERS, MANUFACTURER))
--------------------------------------------------------------------------------
PAYROLL
0000220208247F7B4CD1E0A58FE04011AC2C0A7021247F7B4CD1DEA58FE04011AC2C0A7021
CAR_TYP('CHEVROLET', 'MALIBU', '2003', ENGINE_TYP(6, 'Chevrolet'))

MOTOR POOL
0000220208247F7B4CD1DFA58FE04011AC2C0A7021247F7B4CD1DEA58FE04011AC2C0A7021
CAR_TYP('FORD', 'F-150', '2005', ENGINE_TYP(8, 'Triton'))
```

OBJECT FUNCTIONS AND OPERATORS

- The **IS OF** operator tests whether the given object belongs to the given type.
- The **VALUE** function takes the table alias as a parameter and returns the object instance from the appropriate rows.
  - **VALUE** returns the data as an object.
- The **SYS_TYPEID** function returns the typeid of the most specific type of the given parameter.
  - This function is useful in determining which type in the type hierarchy the object belongs to.
- The **TREAT** function checks that an expression can be operated on as if it were a different datatype.
  - The most common use is to determine if a supertype can function as the subtype.
- Oracle also includes the **UTL_REF** supplied package to perform reference-based operations.
  - **UTL_REF** procedures let you write generic methods without knowing the object name.

In the following example, we return the object as data from the **car_characteristics** table where that object is of the type **car_typ**. The second **SELECT** raises an error, since the value is not an **engine_typ**.

value.sql

```
SELECT VALUE(c) FROM car_characteristics c
 WHERE VALUE(c) IS OF (car_typ);

SELECT VALUE(c) FROM car_characteristics c
 WHERE VALUE(c) IS OF (engine_typ);
```

```
VALUE(C)(MAKE, MODEL, YEAR, ENGINE(CYLINDERS, MANUFACTURER))
--------------------------------------------------------------------------------
CAR_TYP('FORD', 'F-150', '2005', ENGINE_TYP(8, 'Triton'))
CAR_TYP('CHEVROLET', 'MALIBU', '2003', ENGINE_TYP(6, 'Chevrolet'))

 WHERE VALUE(c) IS OF (engine_typ)
                       *
ERROR at line 2:
ORA-00932: inconsistent datatypes: expected UDT got HR.CAR_TYP
```

The **UTL_REF** supplied package contains the following subprograms:

Procedure	Usage
DELETE_OBJECT	Removes an object from an object table pointed to by the given reference.
LOCK_OBJECT	Locks an object in the object table pointed to by the given reference.
SELECT_OBJECT	Selects an object pointed to by the given reference and stores the object in the PL/SQL variable.
UPDATE_OBJECT	Updates the object pointed to by the given reference.

LABS

❶ Create an object type called **current_weather_typ** to model current weather observations. This type should have attributes for city, state, current temperature, and current status (such as snowy, sunny, etc.)
(Solution: *weather.sql*)

❷ Create an object table called **current_weather** with the only column defined as the datatype you just created.
(Solution: *weather_tab.sql*)

❸ Perform a **DESCRIBE** on both the type and the table you created to verify that all has been created properly.
(Solution: *weather_desc.sql*)

❹ Insert the following values into the table you created. Don't forget to commit!

City	State	Temp	Status
New York	NY	32	Sunny
Boston	MA	27	Cloudy
Chicago	IL	15	Blizzard

(Solution: *weather_ins.sql*)

❺ Query the table to ensure the values have been entered correctly.
(Solution: *weather_sel.sql*)

❻ Modify the type to add a new attribute for the wind speed. Query the table to see the results of your change and the values for your new attribute.
(Solution: *wind_speed.sql*)

❼ Create a procedure called **new_weather** that will insert a new row of data into the table, accepting the type attributes as input.
(Solution: *new_weather.sql*)

❽ Use the new procedure to add a new weather report into the table. Query the table to ensure the data was added correctly.
(Solution: *add_weather.sql*)

CHAPTER 8 - ADVANCED PROGRAMMING TOPICS

OBJECTIVES

- Use autonomous transactions in stored subprograms and triggers.
- Choose which user's application context and rights will apply when a stored subprogram runs.
- Create functions to implement fine-grained access control.
- Use **DBMS_PIPE** to set up inter-session communication between PL/SQL programs.
- Dynamically generate data that can be accessed from a **SELECT** statement with table functions.
- Write high-performance code using pipelined table functions.

AUTONOMOUS TRANSACTIONS

- From an Oracle session, all of the changes made to data are part of a single transaction, regardless of the number of DML statements, or procedure and function calls made.
  - Most of the time, this behavior is desired, but can be overridden by declaring a PL/SQL unit to be an autonomous transaction.
- An *autonomous transaction* is an independent transaction started within another transaction (the main transaction).
  - Autonomous transactions allow you to temporarily suspend the main transaction, perform additional SQL operations, commit or rollback those operations separately, then resume the main transaction.
- The autonomous transaction does not share any resources, locks, or commit dependencies with the main transaction, allowing you to perform operations such as log events and increment counters even when the main transaction is rolled back.
- Autonomous transactions can be used in anonymous PL/SQL blocks, local and stand-alone PL/SQL units, packaged PL/SQL units, database triggers, and object type methods.
  - Autonomous transactions are specified individually for each subprogram in a package.
- To define an autonomous transaction, you use a **PRAGMA** statement (compiler directive) in the declaration section of the PL/SQL block.

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

  - You must commit or rollback within the specific program unit that is declared as autonomous.

auton_trig.sql

```
CREATE TABLE emp_log
  ( empno NUMBER(6),
    userid VARCHAR2(30),
    tran_date DATE )
/

CREATE OR REPLACE TRIGGER insert_log
  BEFORE INSERT OR UPDATE ON hr.employees FOR EACH ROW
  DECLARE
     PRAGMA AUTONOMOUS_TRANSACTION;
  BEGIN
    INSERT INTO emp_log
      VALUES (:new.employee_id, USER, SYSDATE);
    COMMIT;
  END;
/
INSERT INTO hr.employees (employee_id, last_name, email,
    hire_date, job_id)
  VALUES (999, 'Ellison', 'larry@oracle.com', SYSDATE, 'SA_REP');
ROLLBACK;
SELECT * FROM hr.employees WHERE employee_id = 999;
SELECT * FROM emp_log;
```

The output:

```
INSERT INTO hr.employees (employee_id, last_name, email,
    hire_date, job_id)
  VALUES (999, 'Ellison', 'larry@oracle.com', SYSDATE, 'SA_REP');

1 row inserted.

ROLLBACK;

Rollback complete.

SELECT * FROM hr.employees WHERE employee_id = 999;

no rows selected

SELECT * FROM emp_log;

     EMPNO USERID                         TRAN_DATE
---------- ------------------------------ ---------
       999 HR                             14-AUG-06
```

INVOKER'S RIGHTS

- By default, stored PL/SQL program units execute with the privileges and schema context of the creator of the unit, rather than the user who is executing the code.
- Use the **AUTHID** clause to override this default behavior.

```
AUTHID CURRENT_USER
```

  - This signals that the program will execute using the rights of the user currently executing the stored program.

- When the **AUTHID** clause is used for a package, it is defined in the package specification and applies to all routines in the package.
- Priviliges are checked when executing a stored subprogram.

  - Invoker's rights will require that the invoker (current user) has all of the necessary privileges for the tables and other subprograms being manipulated.

- Using the invoker's schema context means that object resolution will be based on the invoker's schema, not the schema of the user that defined the subprogram.

  - Each user may have different synonyms, tables, and views defined.

    - Make sure that tables and views are fully-qualified with their schema name to avoid confusion.

- You can explicitly specify the default behavior of definer's rights with the **AUTHID** clause.

```
AUTHID DEFINER
```


Consider a procedure which updates an unqualified reference to **emp**.

With invoker's rights, a user would need to have the **DELETE** privilege on the **emp** table. And, the resolution of **emp** would depend on the current user's schema definition.

delete_emp1.sql

```
CREATE OR REPLACE PROCEDURE delete_emp1 (in_empid NUMBER)
  AUTHID CURRENT_USER IS
BEGIN
  DELETE FROM emp
   WHERE employee_id = in_empid;
END;
/
```

The same procedure using definer's rights would not require the invoker to have the **DELETE** privilege on **emp**, only the **EXECUTE** privilege on the procedure. Additionally, the resolution of **emp** would be based on the schema of the procedure's creator, not the user running it.

```
delete_emp2.sql
CREATE OR REPLACE PROCEDURE delete_emp2 (in_empid NUMBER)
  AUTHID DEFINER IS
BEGIN
  DELETE FROM emp
   WHERE employee_id = in_empid;
END;
/
```

Try It:

Log into the database as **hr**, and create a synonym of **emp**:

```
CREATE SYNONYM emp FOR employees;
```

Then, create each of the procedures above:

```
@delete_emp1
@delete_emp2
```

Grant **EXECUTE** privilege on these procedures to the **oe** user:

```
GRANT EXECUTE ON delete_emp1 TO oe;
GRANT EXECUTE ON delete_emp2 TO oe;
```

Now log in as the **oe** user and try executing each procedure:

```
EXEC hr.delete_emp1(133);   -- this call fails
EXEC hr.delete_emp2(133);   -- this call succeeds
```

FINE-GRAINED ACCESS CONTROL WITH DBMS_RLS

- Fine-grained access control, also referred to as row-level security, is used to create a Virtual Private Database (VPD).
  - Security rules are applied to a statement at parse time based on the database object(s) (a table, view or synonym) that it references.
  - You can associate your own security policy with an object using a PL/SQL function, called a *policy function*.

```
DBMS_RLS.ADD_POLICY('hr', 'jobs', 'job_policy',
   'hr', 'job_policy_func', 'select');
```

- Whenever the database object is referenced in the specified type of statement, a transient view is created and its result set substituted for the original object.
  - Oracle produces this transient view by calling the policy function.

```
SELECT * FROM hr.jobs WHERE predicate
```

    - The *predicate* is the return value from this function, which is used to limit the rows that can be accessed by the end user.
    - The policy function can reference session environment variables to help determine its resulting predicate.

- You can achieve column-level VPD by specifying columns for **INDEX**, **SELECT**, **INSERT**, **UPDATE**, or **DELETE** statements.
  - The policy will only be applied when a specified column is accessed.

- Oracle invokes the function with definer's rights; therefore, no privileges to the function, or any objects that it references, need to be granted to any other users.

Your policy function must have the following interface:

```
FUNCTION policy_function(object_schema IN VARCHAR2,
  object_name VARCHAR2) RETURN VARCHAR2
```

DBMS_RLS subprograms:

Subprogram	Description
ADD_GROUPED_POLICY	Adds a policy associated with a policy group
ADD_POLICY	Adds a fine-grained access control policy to a table, view, or synonym
ADD_POLICY_CONTEXT	Adds the context for the active application
CREATE_POLICY_GROUP	Creates a policy group
DELETE_POLICY_GROUP	Deletes a policy group
DISABLE_GROUPED_POLICY	Disables a row-level group security policy
DROP_GROUPED_POLICY	Drops a policy associated with a policy group
DROP_POLICY	Drops a fine-grained access control policy from a table, view, or synonym
DROP_POLICY_CONTEXT	Drops a driving context from the object so that it will have one less driving context
ENABLE_GROUPED_POLICY	Enables or disables a row-level group security policy
ENABLE_POLICY	Enables or disables a fine-grained access control policy
REFRESH_GROUPED_POLICY	Reparses the SQL statements associated with a refreshed policy
REFRESH_POLICY	Causes all the cached statements associated with the policy to be reparsed

Note:
DMBS_RLS is only available in the Enterprise Edition.

Creating Pipes with DBMS_PIPE

- You can communicate between multiple Oracle sessions using pipes, which utilize shared memory to temporarily store data.
  - Data in a pipe is lost when the Oracle instance is shutdown.
- The **DBMS_PIPE** package allows you to create public or private pipes.
  - Any user can access a public pipe, if they know its name.
  - A private pipe can only be used between sessions belonging to the same user, procedures running with that user's rights, or a user connected with **SYSDBA** privilege.
- Use the **CREATE_PIPE** function to explicitly create a pipe.
  - A flag is used to specify whether the pipe is private (the default) or public.
  - You can also create a public pipe implicitly by simply referencing it by name.
    - An implicit pipe will disappear when it no longer contains data.
  - Since public pipes can be created implicitly, **CREATE_PIPE** is typically used to create private pipes.
- An attempt to explicitly create a pipe with an existing name, whether public or private, generates an error.

Note:
EXECUTE privilege must be granted on **DBMS_PIPE** to users working with pipes.

WRITING TO AND READING FROM A PIPE

- Write data items into a message buffer with one or more calls to **PACK_MESSAGE**, which is overloaded for various datatypes.

```
DBMS_PIPE.PACK_MESSAGE('First item in message');
DBMS_PIPE.PACK_MESSAGE('Second item in message');
DBMS_PIPE.PACK_MESSAGE(numeric_item_3);
```

  - Currently, the maximum buffer size is 4096 bytes, which includes 4 bytes per item for header and terminating information.

- Place the message in a pipe by calling **SEND_MESSAGE** with the pipe name.

```
status := DBMS_PIPE.SEND_MESSAGE('Pipe1');
```

- Read a message from a pipe by calling **RECEIVE_MESSAGE**.

```
status := DBMS_PIPE.RECEIVE_MESSAGE('Pipe1', 432,000);
```

  - A timeout value in seconds, which defaults to **DBMS_PIPE**.**MAXWAIT** or 1,000 days, may be passed.

  - If **RECEIVE_MESSAGE** does not error or timeout, then the next message is removed from the pipe.

- Call **UNPACK_MESSAGE** one or more times to retrieve the individual items.

```
DBMS_PIPE.UNPACK_MESSAGE(item1);
```

- If you call **SEND_MESSAGE** or **RECEIVE_MESSAGE** with a pipe name that was not explicitly created, then a public pipe of that name will be created implicitly.

pipe_send.sql

```
SET SERVEROUTPUT ON
DECLARE
  pipe_status INTEGER;
BEGIN
  DBMS_PIPE.PACK_MESSAGE('Error information can be sent ' ||
    'from a trigger to another session that does logging ' ||
    'in an autonomous transaction!');
  pipe_status := DBMS_PIPE.SEND_MESSAGE('Pipe1');

  IF pipe_status = 0 THEN
     DBMS_OUTPUT.PUT_LINE('Message sent successfully');
  ELSE
     DBMS_OUTPUT.PUT_LINE('Message failed');
  END IF;
END;
/
```

pipe_receive.sql

```
SET SERVEROUTPUT ON
DECLARE
  pipe_status INTEGER;
  message VARCHAR2(256);
BEGIN
  pipe_status := DBMS_PIPE.RECEIVE_MESSAGE('Pipe1', 120);

  IF pipe_status = 0 THEN
     DBMS_PIPE.UNPACK_MESSAGE(message);
     DBMS_OUTPUT.PUT_LINE(message);
  ELSE
     DBMS_OUTPUT.PUT_LINE('Message failed or timed out');
  END IF;
END;
/
```

Try It:

In one session, run *pipe_receive.sql*. It will wait for 120 seconds for a message to appear in the pipe. In another session, run *pipe_send.sql*. The first session will immediately return and output the message.

TABLE FUNCTIONS

- A *table function* produces a collection of rows (a PL/SQL table or varray) that can be accessed in the **FROM** clause of a **SELECT** statement, just like a physical database table.

```
SELECT * FROM TABLE (horse_family(sire, dam));
```

  - The **TABLE** keyword is placed before the function call.

  - You can also call a table function from the **SELECT** list of a query, or use it in the definition of a **VIEW**.

```
CREATE OR REPLACE VIEW horse_view AS
  SELECT * FROM TABLE (horse_family
    (horse_t('King', 'PALOMINO', '30-JUN-2004'),
     horse_t('Goldie', 'PALOMINO', '01-JAN-2005')));
```

    - A **SELECT** from the view makes the table function as easy to use as a database table.

```
SELECT * FROM horse_view WHERE dob < SYSDATE;
```

- Table functions are typically used to produce a result set from information outside of the database, that needs to be generated on the fly.

- Define a table function with an autonomous transaction to produce fanout.

  - With *fanout* you save data for later use that is otherwise being excluded from the result set.

    - DML statements can be used to **INSERT** and **COMMIT** the data into a secondary table within the function.

- You may also pass collections or ref cursors into a table function which allows calls to be daisy-chained together.

horse_family.sql

```
...
CREATE OR REPLACE FUNCTION horse_family (sire_in IN horse_t,
    dam_in IN horse_t)
  RETURN horse_tab
IS
  pony_count PLS_INTEGER;
  horse_list horse_tab := horse_tab();
BEGIN
...
  FOR i IN 1 .. pony_count
  LOOP
    horse_list.EXTEND;
    horse_list(horse_list.LAST) :=
      horse_t('Baby' || i, dam_in.breed, ADD_MONTHS(SYSDATE, i * 12));
  END LOOP;

  RETURN horse_list;
END horse_family;
/
```

show_horse_family.sql

```
...
BEGIN
  sire := horse_t('King', 'PALOMINO', '30-JUN-2004');
  dam := horse_t('Goldie', 'PALOMINO', '01-JAN-2005');

  FOR rec IN (SELECT * FROM TABLE(horse_family(sire, dam)))
  LOOP
    DBMS_OUTPUT.PUT_LINE(rec.name || ' ' || rec.breed ||
      ' ' || rec.dob);
  END LOOP;
END;
/
```

horse_view.sql

```
CREATE OR REPLACE VIEW horse_view AS
  SELECT * FROM TABLE
     (horse_family(horse_t('King', 'PALOMINO', '30-JUN-2004'),
                   horse_t('Goldie', 'PALOMINO', '01-JAN-2005')));
```

show_view_family.sql

```
SELECT * FROM horse_view WHERE dob < SYSDATE;
```

PIPELINED TABLE FUNCTIONS

- *Pipelined table functions* allow you to return result sets one row at a time.
  - The caller may use the data as soon as it is produced, without it being staged or cached.
  - Pipelining helps eliminate the use of intermediate staging tables during transformations for datawarehouses and reduces memory usage.
- Define the table function as **PIPELINED**, and **RETURN** a collection.

```
CREATE FUNCTION ticker_f(stocklist refcur_pkg.refcur_t)
   RETURN tickertypeset PIPELINED
IS ...
```

  - The collection returned must be made up of SQL datatype elements, such as **NUMBER** or **VARCHAR2**.
- To return the individual records, call **PIPE ROW** with an element that matches the datatype declared for the **RETURN** collection.

```
PIPE ROW (ticker_item);
```

  - **PIPE ROW** may be used only in the body of pipelined table functions.
  - The function's **RETURN** statement is only used to transfer control out of the function, since the data is returned as it is generated with **PIPE ROW**.
- Autonomous transactions require a **COMMIT** or **ROLLBACK** before each **PIPE ROW** statement, since they are swapping control with the calling routine.
- Pipelined table functions can dramatically speed up first rows queries, since the remaining rows never need to be generated.

```
... WHERE ROWNUM < 25;
```


stock_func.sql

```
...
CREATE OR REPLACE FUNCTION ticker_f(stocklist refcur_pkg.refcur_t)
  RETURN tickertypeset PIPELINED
IS
  the_row_as_object tickertype := tickertype(NULL, NULL, NULL, NULL);

  TYPE stocklist_tt IS TABLE OF stocklist%ROWTYPE
    INDEX BY PLS_INTEGER;

  l_stocklist stocklist_tt;
  retval tickertypeset := tickertypeset();
  the_row pls_integer;
BEGIN
  FETCH stocklist
    BULK COLLECT INTO l_stocklist;
  CLOSE stocklist;

  the_row := l_stocklist.FIRST;

  WHILE (the_row IS NOT NULL)
  LOOP
    -- row one
     the_row_as_object.ticker := l_stocklist(the_row).ticker;
     the_row_as_object.pricetype := 'O';
     the_row_as_object.price := l_stocklist(the_row).open_price;
     the_row_as_object.pricedate := l_stocklist(the_row).trade_date;

    PIPE ROW (the_row_as_object);

    -- row two
     the_row_as_object.pricetype := 'C';
     the_row_as_object.price := l_stocklist(the_row).close_price;
     the_row_as_object.pricedate := l_stocklist(the_row).trade_date;

    PIPE ROW (the_row_as_object);

    the_row := l_stocklist.NEXT (the_row);
  END LOOP;
  RETURN;
END;
/
```

first_stocks.sql

```
SELECT *
  FROM TABLE (ticker_f (CURSOR (SELECT * FROM stocks)))
 WHERE ROWNUM < 25;
```

ENABLING PARALLEL EXECUTION

- Use the **PARALLEL_ENABLE** hint to take advantage of multiple processors.
  - The Oracle runtime system will then be able to execute the function in parallel, when called from a parallel DML statement.
- When a table function's parameter list consists only of a single ref cursor, you must specify what type of partitioning should be used to distribute the data among the slave sessions.
  - Use **PARTITION ... BY ANY** to allow the runtime to arbitrarily distribute the data.

```
CREATE FUNCTION parallel_exmpl(
     input_rows REF CURSOR)
   RETURN My_Types.output_tab PIPELINED
   PARALLEL_ENABLE(PARTITION input_rows BY ANY) ...
```

    - Both weak and strongly-typed ref cursors can use **ANY** for the partitioning.
  - If the input **REF CURSOR** is strongly-typed, you can use **HASH** or **RANGE** with a column list to group the data given to each slave.

```
CREATE FUNCTION parallel_exmpl(
     input_rows My_Types.cur_t)
   RETURN My_Types.output_tab PIPELINED
   PARALLEL_ENABLE(
     PARTITION input_rows BY HASH(input_column)) ...
```

- Do not use session state information, such as package variables, from a parallel function, as those variables should not be shared among the slaves.

Parallel table functions are used mainly in data warehouses where queries must be run in parallel to achieve results in a reasonable amount of time. In this scenario, parallel table functions can be used as the input to other parallel table functions, allowing complex results, without storing all of the intermediate data in temporary tables.

stock_parallel.sql

```
...
CREATE OR REPLACE FUNCTION ticker_f(stocklist refcur_pkg.refcur_t)
   RETURN tickertypeset
   PIPELINED
   PARALLEL_ENABLE (PARTITION stocklist BY ANY)
IS
   ...
BEGIN
   ...
END;
/
```

show_stock_parallel.sql

```
SELECT * /* PARALLEL (my_tab, 4) */
  FROM TABLE (ticker_f (CURSOR (SELECT * FROM stocks))) my_tab
 WHERE price BETWEEN 500 AND 2000;
```

DETERMINISTIC FUNCTIONS

- A *deterministic function* produces the same result value for any combination of argument values passed into it.

- Oracle automatically attempts to use previously-calculated results for a deterministic function, when the same input values are used in subsequent invocations.

- To mark a function as deterministic, place the **DETERMINISTIC** keyword after the return type in a declaration of the function.

```
CREATE FUNCTION times_two (val NUMBER)
   RETURN NUMBER DETERMINISTIC IS
BEGIN
  RETURN val * 2; -- same result for every val passed in
END;
```

  - A function is non-deterministic if it utilizes package or database data, since that information could change from one invocation to the next.

  - Since Oracle has no way of detecting whether a function is deterministic, unpredictable results can occur if the function is not truly deterministic.

- Only functions that are **DETERMINISTIC** are allowed in function-based indexes and in certain snapshots and materialized views.

The **DETERMINISTIC** keyword may be used:

- On a function defined in a **CREATE FUNCTION** statement.
- In a function declaration in a **CREATE PACKAGE** statement.
- On a method declaration in a **CREATE TYPE** statement.

The following features require that any function used with them be declared **DETERMINISTIC**:

- Any user-defined function used in a function-based index.
- Any function used in a materialized view, if that view is to qualify for Fast Refresh or is marked **ENABLE QUERY REWRITE**.

Functions that fall in the following categories should typically be **DETERMINISTIC**:

- Functions used in a **WHERE**, **ORDER BY**, or **GROUP BY** clause.
- Functions that in any other way help determine whether or where a row should appear in a result set.

Consider the following when you create **DETERMINISTIC** functions:

- The database cannot recognize whether the behavior of the function is indeed deterministic. If the **DETERMINISTIC** keyword is applied to a function whose behavior is not truly deterministic, then the result of queries involving that function is unpredictable.

- If you change the semantics of a **DETERMINISTIC** function and recompile it, then existing function-based indexes and materialized views must be rebuilt.

LABS

❶ As the **hr** user, create a package called **logging**. Include a stored procedure called **log_mesg** that takes a string message and writes it into the logging table, along with the current date and time. Test your procedure to make sure it works. (Hint: you will first need to create a **log** table that has **TIMESTAMP** and **VARCHAR2** columns to hold the log messages.)
(Solutions: *log_tab.sql*, *logging_pack.sql*)

❷ Add a trigger to the **hr.employees** table that will call **log_mesg** when a row is inserted or updated. Test the trigger, including the case when the modification is rolled back. Does the log entry persist across a **ROLLBACK**? Make the stored procedure in the **logging** package an autonomous transaction and test it again.
(Solutions: *emplog_trig.sql*, *logging_test2.sql*, *logging_pack2.sql*)

❸ a. As the **hr** user, grant **EXECUTE** on the logging package to the **oe** user.
(Solution: *logging_grant.sql*)

b. As the **oe** user, note that you cannot examine the **log** table.
(Solution: *query_log.sql*)

c. Add a trigger to the **oe.customers** table that calls **hr.logging.log_mesg** when a row is inserted or updated. Test the trigger.
(Solution: *custlog_trig.sql*, *logging_test3.sql*)

d. Examine the log table as the **hr** user. Note that the **oe** user is able to make these changes.
(Solution: *query_log.sql*)

e. Change the logging procedure to use the current user's rights, instead of the definer's.
(Solution: *logging_pack3.sql*)

f. What happens now? Add a **log** table as the **oe** user and test as both the **hr** and **oe** user - note where the log messages are written for each user.
(Solution: *logging_test3.sql*, *log_tab.sql*, *query_log.sql*, *logging.txt*)

❹ Create two scripts that write to the same pipe. The first script should write a single message with three items, while the second should write three messages of one item each. Create a script that will read and display all of the data on the pipe, assuming that the two scripts were executed to populate the pipe.
(Solution: *sender1.sql*, *sender2.sql*, *receiver.sql*)

❺ Create a table function named **evens** that takes an integer **limit** as a parameter, and returns a table of all positive, even numbers up to **limit**. Test this function with both small and large values for **limit**. (Hint: Use a large value that is big enough to cause a delay in the returned results.)
(Solution: *evens.sql*, *show_evens.sql*, *show_evens_big.sql*)

❻ Modify the function created in ❺ to be a pipelined table function. Test it again with small and large values to see the effect of pipelining the results.
(Solution: *evens2.sql*, *show_evens.sql*, *show_evens_big.sql*)

