

Oracle for Developers PLSQL

Lesson 00:

People matter, results count.



©2016 Capgemini. All rights reserved.
The information contained in this document is proprietary and
confidential. For Capgemini only.

Document History

Date	Course Version No.	Software Version No.	Developer / SME	Change Record Remarks
13-Nov-2008	1.0	Oracle9i	Rajita Dhumal	Content Creation.
28-Nov-2008	1.1	Oracle9i	CLS team	Review.
14-Jan-2010	1.2	Oracle9i	Anu Mitra,	Review
14-Jan-2010	1.2	Oracle9i	Rajita Dhumal, CLS Team	Incorporating Review Comments
Jun-2011	2.0	Oracle 9i	Anu Mitra	Integration Refinements



Copyright © Capgemini 2015. All Rights Reserved 2

Course Goals and Non Goals

■ Course Goals

- To understand RDBMS Methodology
- To code PL/SQL Blocks for Implementing business rules
- To create Stored Subprograms using Packages, Procedures and Functions.
- To implement Complex Business Rule using Triggers, Constraints

Course Goals and Non Goals

- Course Non Goals

- Object Oriented programming concepts (ORDBMS) are not covered as a part of this course.

Pre-requisites

- Required a fair proficiency level in Relational Database Concepts.
- Required good proficiency in DBMS SQL

Intended Audience

- Software Programmers
- Software Analysts



Day Wise Schedule

- Day 1
 - Lesson 1: Introduction to Oracle Architecture
- Day 2
 - Lesson 2: Introduction to Data Dictionary
 - Lesson 3: PL/SQL Basics
 - Lesson 4: Introduction to Cursors (Till SELECT..FOR UPDATE)

Day Wise Schedule

■ Day 3

- Lesson 4: Introduction to Cursors contd..
- Lesson 5: Exception Handling and Dynamic SQL
- Lesson 6: Procedures, Functions, and Packages (Till Functions)

■ Day 4

- Lesson 6: Procedures, Functions, and Packages contd..
- Lesson 7: Database Triggers
- Lesson 8: Locks

Day Wise Schedule

- Day 5
 - Lesson 9: Built-in packages in Oracle
 - Lesson 10: SQL*Plus Reports
 - Lesson 11:SQL * Loader
 - Lesson 12: Oracle Tools

Table of Contents

- Lesson 1: Introduction to Oracle Architecture
 - 1.1: Overview of Primary Components
 - 1.2: Oracle Server
 - 1.3: Oracle Instance
 - 1.4: Creating a Session
 - 1.5: Oracle Database and its components
- Lesson 2: Introduction to Data Dictionary
 - 2.1: Data Dictionary
 - 2.2: Contents of Data Dictionary

Table of Contents

- Lesson 2: Introduction to Data Dictionary contd..
 - 2.3: Structure of Oracle Data Dictionary
 - 2.4: Examples of Data Dictionary
 - 2.5: Queries for Data Retrieval from Data Dictionary
- Lesson 3: PL/SQL Basics
 - 3.1: Introduction to PL/SQL
 - 3.2: PL/SQL Block Structure
 - 3.3: Handling Variables in PL/SQL

Table of Contents

- Lesson 3: PL/SQL Basics contd..
 - 3.4: Declaring a PL/SQL table
 - 3.5: Scope and Visibility of Variables
 - 3.6: SQL in PL/SQL
 - 3.7: Programmatic Constructs
- Lesson 4: Introduction to Cursors
 - 4.1: Introduction to Cursors

Table of Contents

- Lesson 4: Introduction to Cursors contd..
 - 4.2: Implicit Cursors
 - 4.3: Explicit Cursors
 - 4.4: Cursor Attributes
 - 4.5: Cursor with Parameters
 - 4.6: Usage of Cursor Variables
- Lesson 5: Exception Handling and Dynamic SQL
 - 5.1: Error Handling (Exception Handling)

Table of Contents

- Lesson 5: Exception Handling and Dynamic SQL contd..
 - 5.2: Predefined Exception
 - 5.3: Numbered Exceptions
 - 5.4: User Defined Exceptions
 - 5.5: OTHERS Exception Handler
 - 5.6: Dynamic SQL
- Lesson 6: Procedures, Functions, and Packages
 - 6.1: Subprograms in PL/SQL
 - 6.2: Anonymous Blocks versus Stored Subprograms
 - 6.3: Procedures

Table of Contents

- Lesson 6: Procedures, Functions, and Packages (contd.)
 - 6.4: Functions
 - 6.5: Packages
 - 6.6: Autonomous Transactions
- Lesson 7: Database Triggers
 - 7.1: What is a Trigger?
 - 7.2: Types of Triggers
 - 7.3: Where are Triggers Used?

Table of Contents

- Lesson 8: Locks
 - 8.1: Data Concurrency and Consistency
 - 8.2: Locking in Oracle
 - 8.3: Types of Locks
 - 8.4: DDL and DML Locks
- Lesson 9: Built-in Packages in Oracle
 - 9.1: Overview
 - 9.2: DBMS_OUTPUT: Displaying Output
 - 9.3: UTL_FILE
 - 9.4: DBMS_LOB

Table of Contents

- Lesson 10: SQL * Plus Reports
 - 10.1: SQL * Plus Reporting
 - 10.2: SQL * Plus Commands
- Lesson 11: SQL * Loader
 - 11.1: What is SQL * Loader?
 - 11.2: SQL * Loader as a Utility
 - 11.3: SQL * Loader Environment
 - 11.4: The Bad File and Discard File

Table of Contents

- Lesson 11: SQL * Loader contd..
 - 11.5: Invoking SQL * Loader
 - 11.6: SQL * Loader Examples
- Lesson 12: Oracle Tools
 - 12.1: Oracle Provided Management Tools
 - 12.2: Oracle Enterprise Manager

References

- Oracle PL/SQL Programming, Third Edition; by Steven Feuerstein
- Oracle 9i PL/SQL: A Developer's Guide
- Oracle9i PL/SQL Programming; by Scott Urman



Next Step Courses

- Data Warehousing Concepts
- Reporting / ETL tools
- Database Administration / Database Performance Tuning



Other Parallel Technology Areas

- Microsoft SQL Server
- IBM DB2

Oracle (PL/SQL)

Lesson 1: Introduction to Oracle Architecture

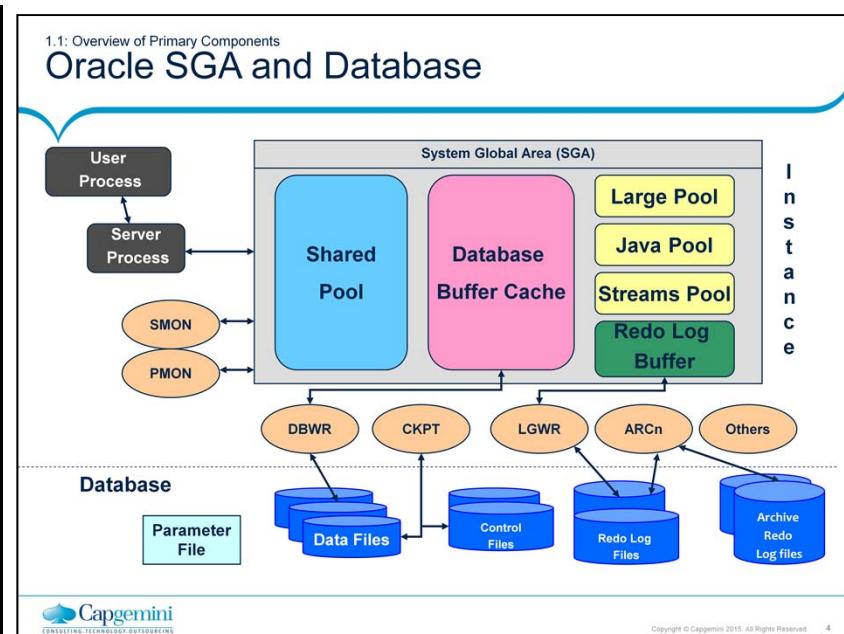
Lesson Objectives

- To understand the following topics:
 - Outline of Oracle architecture and it's main components
 - List of structures involved in connecting a user to an Oracle instance



Introduction to Oracle Database:

- Oracle is an Object Relational Database Management System (ORDBMS).
- Oracle uses:
 - Relational Data Model as well as Object Relational Data Model to store its database, and
 - SQL (commonly abbreviated as Structured Query Language) to process the stored data.
- Oracle database is designed with improvements in the areas such as:
 - database performance
 - ease of management
 - scalability
 - security
 - availability, etc.
- The following features makes Oracle very powerful:
 - usage of XMLType, which is a new data type that lets you store native XML documents directly in the database
 - support of multimedia and large objects
 - support for Oracle Streams, which are a generic mechanism for sharing data that can be used as the basis of many processes including messaging, replication, and warehouse ETL processes



Overview of Primary Components in Oracle Architecture:

The Oracle Server consists of an “Oracle instance” and an “Oracle database”.

- 1. Oracle instance:** An “Oracle instance” is the combination of the “background processes” and “memory structures”.
 - The instance must be started to access the data in the database. Every time an instance is started, a “System Global Area (SGA)” is allocated, and “Oracle background processes” are started.
 - Background processes perform functions on behalf of the invoking process. The background processes perform input / output (I/O), and monitor other Oracle processes to provide increased parallelism for “better performance” and “reliability”.
- 2. Oracle database:** An Oracle database consists of “Operating System files”, also known as “database files” that provide the actual physical storage for database information. The database files are used to ensure that the data is kept consistent and can be recovered in the event of a failure of the instance.
- 3. Other key files:** Non-database files are used to configure the instance, authenticate privileged users, and recover the database in the event of a disk failure. Eg: Parameter File, Archive files
- 4. User and server processes:** The “user processes” and “server processes” are the primary processes involved when a SQL statement is executed. However, other processes may help the server complete the processing of the SQL statement.

1.1: Overview of Primary Components

Oracle Server Defined

- An Oracle server:

- is a “database management system (DBMS)” that provides an open, comprehensive, integrated approach to information management.
- consists of an “Oracle instance” and an “Oracle database”.



Copyright © Capgemini 2015. All Rights Reserved 5

Oracle Server:

- The Oracle Server can run on a number of different computers in one of the following environments:
 - Client-Application Server-Server
 - Client-Server
 - Host-Based
- **Client-Application Server-Server:** (Three-tier) Users access the database from their personal computers (clients) through an application server, which is used for the application's processing requirements.
- **Client-Server:** (Two-tier) Users access the database from their personal computer (client) over a network, and the database sits on a separate computer (server).
- **Host-Based:** Users are connected directly to the same computer which houses the database.

1.1: Overview of Primary Components

Oracle Instance

- An Oracle instance:
 - is a means to access an Oracle database.
 - always opens one and only one database.
 - consists of memory and process structures.

The diagram illustrates the architecture of an Oracle Instance. At the top, the System Global Area (SGA) is shown as a large grey box containing two main components: the Shared Pool (blue) and the Database Buffer Cache (pink). To the right of the SGA, four additional memory structures are listed vertically: Large Pool, Java Pool, Streams Pool, and Redo Log Buffer. Below the SGA, seven background processes are represented by orange ovals: SMON, PMON, DBWR, CKPT, LGWR, ARCN, and Others. The Capgemini logo is at the bottom left, and a copyright notice is at the bottom right.

Oracle Instance:

- When a database is started on a database server (regardless of the type of computer), the Oracle instance starts up. An Oracle instance consists of two components, one is memory structures and two is background processes. The memory structures allocated are termed as the System Global Area (SGA). Along with the allocation of memory the background processes also start.
- This combination of the SGA and the Oracle processes is called an “Oracle instance”.
- The memory and processes of an Oracle instance:
 - efficiently manage the data of the associated database, and
 - serve one or multiple users of the database
- Oracle instance is a means to access an Oracle database.
- Oracle instance always opens “one and only one database”.

We will understand the SGA first in the subsequent slides and then move on to the database

1.1: Overview of Primary Components

Connection and Creating a Session

- Connecting to an Oracle instance consists of:
 - establishing a User Connection, and
 - creating a Session

The diagram shows a 'Database user' icon (a person in front of three blue cylinders) connected by a red arrow labeled 'Connection established' to a 'User process' icon (a person in front of a computer monitor). This 'User process' is then connected by a red arrow labeled 'Session created' to a 'Server process' icon (a person in front of a server rack). The 'Server process' is shown above an 'Oracle Server' icon (a server tower and monitor).

Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 7

Connecting to an Oracle Instance:

- Before users can submit SQL statements to an Oracle database, they must connect to an instance.

➤ Establishing a User Connection

- The user starts a tool such as SQL*Plus or runs an application developed by using a tool such as Oracle Forms. This application or tool is executed as a "user process".
- Establishing the connection creates a communication pathway between a user process and an Oracle Server. As depicted in the figure on the slide, the user process communicates with a server process. The user process executes on the client machine and server process executes on the server machine and actually executes SQL statements submitted by the system user.
- On the slide you can see one –to-one correspondence between the user and server process. This kind of connection is called as a "Dedicated Server" connection. Alternatively you can also use "Shared Server" connection. In a Dedicated Server process one user process connects to one server process. In a Shared Server connection the server process is shared amongst more than one user process.

➤Creating a Session

- A session is a “specific connection” of a user to an Oracle server.
 - The session starts when the user is validated by the Oracle server.
 - The session ends when the user logs out(disconnect) or when there is an abnormal termination (network failure or client machine failure).
- For a given database user, many “concurrent sessions” are possible if the user logs on from many tools, applications, or terminals at the same time. The DBA can limit the number of concurrent sessions.

If the Oracle Server is not running due to whatever reason, the user trying to connect will get “Oracle not available” error.

In one of the earlier lessons, we also talked about using HostString or connect string to connect to Oracle database. Let us understand about this.

On the client exists a tnsnames.ora file. This file is used by Oracle client to connect to oracle server/database. To connect to the database you provide hoststring or connect string thorough SQL*Plus

For example : scott/tiger@trgdb

This connect identifier i.e trgdb as in the above example is resolved by Oracle client by looking into the tnsnames.ora file which is placed on the client machine. The tnsnames.ora provides the network resolution for connect identifier used by Oracle clients and applications.

Below is a sample entry of tnsnames.ora file. Address_List and Address specifies the address of the server and port to which the client needs to connect The connect_data specifies the service name i.e sid of the database.

```
TRGDB =
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = TCP)(HOST = 192.168.224.26)(PORT =
      1521))
    )
    (CONNECT_DATA =
      (SERVICE_NAME = trgdb)
    )
  )
```

Note: SID is a unique name for the Oracle database instance which is running on the server

1.2: Oracle Database Memory Structure

- Memory Structure:
 - The Memory Structure of Oracle consists of two memory areas known as:
 - **System Global Area (SGA):** Allocated at instance startup, and is a fundamental component of an Oracle Instance.
 - **Program Global Area (PGA):** Allocated when the server process is started.



Copyright © Capgemini 2015. All Rights Reserved 9

Memory Structure:

- Memory structure of Oracle consists of two memory areas known as:
 - **System Global Area (SGA):** It is allocated at instance startup, and is a fundamental component of an Oracle Instance.
 - **Program Global Area (PGA):** It is allocated when the server process is started.

1.2: Oracle Database

System Global Area (SGA)

- The SGA consists of following memory structures:

- Mandatory
 - Shared pool
 - Database buffer cache
 - Redo log buffer
 - Streams Pool
- Optional
 - Large pool
 - Java Pool

- SGA can be sized by the SGA_MAX_SIZE parameter



Copyright © Capgemini 2015. All Rights Reserved 10

System Global Area (SGA):

- System Global Area (SGA) is a group of “shared memory structures” that contain data and control information for one Oracle database instance. When “multiple users” are connected to the same instance, the data in the SGA is “shared by all users”. Hence, it is appropriately called “Shared Global Area”.
- Oracle allocates memory for the SGA, when the database instance is started and returns the memory when the instance is shut down. The maximum size of the SGA is determined by SGA_MAX_SIZE initialization parameter in the initInstanceName.ora file or server parameter (SPFILE) file.
- The following statement can be used to view SGA memory allocations:
SHOW SGA;

➤ Total System Global Area	36437964 bytes
➤ Fixed Size	6543794 bytes
➤ Variable Size	19521536 bytes
➤ Database Buffers	16777216 bytes
➤ Redo Buffers	73728 bytes
- Oracle 9i onwards Oracle Server uses a Dynamic SGA. Memory structures for the SGA can be made without shutting down the database

System Global Area (SGA) (contd.):

- There are various initialization parameters to affect the amount of memory allocated to SGA.
 - SGA_MAX_SIZE: As mentioned earlier it is used to set the limit on the amount of memory allocated to SGA. A typical value could be 1 GB. If the value for this parameter set in the parameter file is less than the sum of memory allocated for all components within SGA either explicitly in the parameter file or default, then the database ignores the setting for SGA_MAX_SIZE. For optimal performance, the entire SGA should fit in real memory to avoid paging to/from disk by the operating system.
 - Other parameters which influence the SGA size are DB_CACHE_SIZE, LOG_BUFFER, SHARED_POOL etc... As we go along we will cover the parameters in the respective memory structures.
- Memory allocated to SGA is contiguous virtual memory unit termed as granules. Granule size depends on the size of SGA. If the size of SGA is less than 1 GB in total, each granule is 4 MB and it is greater the 1 GB then each granule is 16 MB. These granules are assigned to each memory structure. The actual number of granules assigned to each of these memory components can be determined by querying v\$BUFFER_POOL.
- Automatic Shared Memory Management: Prior to Oracle 10g, the DBA had to manually specify SGA component sizes through the initialization parameters, like SHARED_POOL_SIZE, DB_CACHE_SIZE and so on. With Automatic Shared Memory Management, the DBA can specify the total SGA memory available through SGA_TARGET initialization parameter. The Oracle database automatically distributes this memory among various subcomponents to ensure most optimum memory utilization. With automatic memory management, the different SGA components are flexible sized to adapt to SGA available. Once DBA has set this parameter, DBA can forget about the sizes of individual components. No out of memory errors are generated unless the system has actually run out of memory. Manual tuning effort is reduced.
- The SGA_TARGET parameter reflects the total size of the SGA and include the following components:-
 - Fixed SGA and other internal allocations needed by Oracle
 - Log Buffer
 - Shared Pool
 - Database Buffer Cache (Keep and Recycle buffer caches, if specified)
 - Java Pool
 - Streams Pool
- If the value of SGA_TARGET is greater than SGA_MAX_SIZE at startup, then the SGA_MAX_SIZE value is bumped up to accommodate SGA_TARGET. The shared pool, Java pool, large pool and buffer cache are automatically sized in Oracle 10g if SGA_TARGET is set. The other SGA components like Keep/Recycle buffer caches, any additional buffer caches and streams pool is not automatically sized and the DBA must specify their sizes explicitly

1.3: System Global Area (SGA) Shared Pool

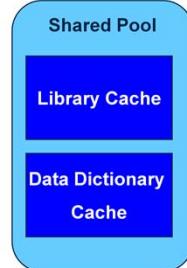
- Shared Pool:

The shared pool is used to store the most recently executed SQL statements, and the most recently used Data Definitions.

- It consists of two key memory structures:

- Library cache
 - Data dictionary cache

- It is sized by the parameter SHARED_POOL_SIZE.



Copyright © Capgemini 2015. All Rights Reserved 12

Shared Pool:

The Shared Pool is a memory structure shared by all users. It consists of both fixed and variable structures. The variable component grows and shrinks depending on the demands placed on memory size by users and application programs. The Shared Pool includes the Library Cache and Data Dictionary Cache.

Memory can be allocated to the Shared Pool by the parameter SHARED_POOL_SIZE in the parameter file. The default value is 8 MB on 32 bit platforms and 64MB on 64-bit platforms. Increasing the value of this parameter increases the amount of memory reserved for the shared pool. You can alter the size of the shared pool dynamically with the

ALTER SYSTEM SET SHARED_POOL_SIZE=20M

You cannot exceed the maximum size of the SGA. The shared pool stores the most recently executed SQL statement and used data definitions. This will help in reuse of the same SQL statements executed by the user. This helps in performance improvement.

1.3. System Global Area (SGA) Library Cache

- Library Cache can be described as follows:
 - The Library Cache stores information about the most recently used SQL and PL/SQL statements. The Library Cache enables sharing of commonly used statements.
 - It is managed by a least recently used (LRU) algorithm.
 - It consists of two structures:
 - Shared SQL area
 - Shared PL/SQL area
 - It has its size determined by the Shared Pool Sizing.



Copyright © Capgemini 2015. All Rights Reserved 13

Library Cache:

- Library Cache contains statement text, parsed code, and execution plan. It contains fully parsed or compiled representations of PL/SQL blocks (procedures, triggers, etc) and SQL statements. The library cache size is based on the sizing defined for the shared pool. Memory is allocated when a statement is parsed or a program unit is called. If the Shared Pool is too small the size of the library cache is also affected. As a result the statement definitions are continually purged in order to have space to load new SQL and PL/SQL statements into the library cache, which affects performance. The library cache is managed by a least recently used (LRU) algorithm. As the cache is filled, less recently used execution paths and parse trees are removed from the library cache to make room for new entries. If the SQL and PL/SQL statements that are oldest and not reused, eventually age out.
- The library cache consists of two memory structures:
 - Shared SQL: This structure stores and shares the execution plan and parse tree for SQL statements run against the database. If the user executes an identical SQL statement then it is able to take benefit of the parse information available in the shared SQL area to accelerate the execution. To ensure that statements use Shared SQL area whenever possible, the text, schema, and bind variables should be exactly same.
 - Shared PL/SQL: This structure stores and shares the most recently used parsed and compiled PL/SQL statements like functions, procedures, packages & triggers.

1.3: System Global Area (SGA)

Data Dictionary Cache

- Data Dictionary Cache can be described as follows:
 - The Data Dictionary Cache is a collection of the most recently used data dictionary information from the database.
 - It includes database files, tables and their descriptions, Indexes, columns, users, privileges, and other database objects.
 - During the "parse phase", the "server process" looks at the Data Dictionary for information to resolve "object names", and validate the access.
 - The query response time is improved by caching the data dictionary information
 - The size is determined by the Shared Pool Sizing.



Copyright © Capgemini 2015. All Rights Reserved 14

Data Dictionary Cache:

- The data dictionary cache is sometimes also referred to as row cache. The overall response time of the query improves by caching information from the data dictionary. Whenever this information is needed for execution of a SQL statement it is read from the database and stored in the cache.
- The size of the data dictionary is dependent on the size of shared pool. If the data dictionary is too small, then the dictionary tables residing on the disk will have to repeatedly queried for information needed which will slow down the response time for queries.

1.3. System Global Area (SGA)

Database Buffer Cache

- The Database Buffer Cache stores copies of data blocks that have been retrieved from the data files.
- Enables performance improvement when you obtain and update data.
- Managed through a LRU algorithm.
- DB_BLOCK_SIZE determines the primary block size.

The diagram illustrates the Database Buffer Cache structure. At the top is a purple rounded rectangle labeled "Database Buffer Cache". Below it are three columns of smaller rectangles representing different pools: "Default", "Keep", and "Recycle". Each pool has three sub-sections labeled "2K", "4K", and "16K" at the bottom. Each section contains a grid of small squares, some of which are shaded dark grey to represent data blocks.

Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 15

Database Buffer Cache:

- The Database Buffer cache (DB cache) is a large memory structure which stores copies of actual data blocks retrieved from datafiles for queries and DML commands. Whenever a query is issued the server process first looks in the DB cache to determine if the requested information happens to be already located in memory, if not available in memory the server process retrieves the information from disk and stores it in the cache.
- The data read from the disk is read one block at a time and not a row at a time. Blocks in the DB cache is stored according to the LRU algorithm and are aged out of memory if a buffer cache block is not used in order to provide space for the insertion of newly needed database blocks. Before reading a block from the database the process must find a free buffer. As you can see in the figure on the slide, the DB cache is made of many different pools
- The block size for a database is set when a database is created and is specified by DB_BLOCK_SIZE parameter. The size of each buffer in the default, keep, recycle pool is influenced by this parameter.

1.3: System Global Area (SGA)

Database Buffer Cache (Contd...)

- Database Buffer Cache consists of sub caches
 - Default
 - Keep
 - Recycle
 - Non-standard block sizes
- The size of sub caches can be controlled by parameters:
 - DB_CACHE_SIZE
 - DB_KEEP_CACHE_SIZE
 - DB_RECYCLE_CACHE_SIZE
 - DB_nK_CACHE_SIZE



Copyright © Capgemini 2015. All Rights Reserved 16

Database Buffer Cache:

- The DB cache also consists of independent sub-caches for non-standard block sizes. The size of the DB cache can be controlled by the following parameters:

DB_CACHE_SIZE : Sizes the default buffer pool, it cannot be set to 0. It allows to dynamically change the memory allocated to cache

DB_KEEP_CACHE_SIZE: Sizes the KEEP buffer pool. This pool holds on to blocks in memory that are more likely to be reused. For example, table data containing username and passwords.

DB_RECYCLE_CACHE_SIZE: Sizes the RECYCLE buffer pool. This pool stores data that have little chance of being reused. Thus the data blocks are quickly removed from memory when not needed.

DB_nK_CACHE_SIZE: an Oracle database can also be created with non-standard block sizes apart from the standard block sizes. You can create upto 4 non-standard block sizes from 2KB to 32KB. To size the non-standard buffer pools within the DB cache this parameter can be used.

To change the size dynamically you can use

```
ALTER SYSTEM SET DB_CACHE_SIZE=100M
```

1.3: System Global Area (SGA) Redo Log Buffer

- The Redo Log Buffer Cache records all changes made to the data blocks.
- Its main purpose is recovery.
- Recorded changes are called redo entries
- Redo entries contain information to reconstruct or redo changes.
It is sized by the parameter LOG_BUFFER.



Copyright © Capgemini 2015. All Rights Reserved 17

Redo Log Buffer :

- The Redo Log buffer memory stores images of all changes made to database data blocks. It is a circular buffer that is used over and over. As the buffer fills the redo entries are flushed out of the memory.
- A data block typically stores several rows and change in any value of the row will require the redo entry to be created. The redo entries contain information necessary to recreate the data prior to the change which was done by INSERT,DELETE,UPDATE,CREATE, ALTER, or DROP
- To size the Redo Log Buffer you can use
`ALTER SYSTEM SET LOG_BUFFER=95M`

1.3: System Global Area (SGA)

Large Pool

- The Large Pool is an optional memory area in the SGA and is configured only in shared server environment.
- The burden on Shared Pool is reduced.
- This memory area is typically used for Session Memory (UGA), I/O slaves, and backup and restore operations.
- The Large Pool does not use an LRU list.
- It is sized by the parameter LARGE_POOL_SIZE.



Copyright © Capgemini 2015. All Rights Reserved 18

Large Pool:

- The Large Pool is an optional memory area that reduces the burden on the Shared Pool. Whenever users connect through a Shared Server process, Oracle will need to allocate space in the Shared Pool for storing information about the user processes and server processes they are connected to. Hence memory for Large Pool can be allocated in this case so that Shared Pool will not have to give memory for storing this additional information.
- If the Large Pool is allocated then it is used for:
 - Allocating space for session memory requirements from User Global Area (UGA) where the Shared Server environment is configured. (In dedicated server environment the UGA is part of PGA which is covered later in the lesson)
 - Backup and Restore operations by the Recovery Manager
- The LARGE_POOL_SIZE is not a dynamic parameter and it does not use LRU algorithm to manage memory.

1.3. System Global Area (SGA)

Java Pool

- The Java Pool services the parsing requirements for Java commands.
- Required in case of installation and use of Java.
- Stored in the same way as PL/SQL in database tables.
- Sized by the JAVA_POOL_SIZE parameter.



Copyright © Capgemini 2015. All Rights Reserved 19

Java Pool:

- The Java Pool is an optional setting and is required if the database has
 - Oracle Java installed and is using Oracle Java Virtual Machine (JVM). The Java Pool even if not defined is set to default size of 24 MB. The Java Pool is used to parse Java commands and store data associated with Java commands. This is similar to storing SQL and PL/SQL code in Shared pool.

1.3. System Global Area (SGA)
Streams Pool

- The Streams Pool stores data and control structures to support the Oracle streams feature of Oracle Enterprise Edition.
- It is new cache introduced in Oracle 10g
- Sized with parameter STREAMS_POOL_SIZE



Copyright © Capgemini 2015. All Rights Reserved 20

Streams Pool:

Oracle Streams manage sharing of data and events in a distributed environment. If the streams pool is set to 0 then the memory for streams operation is allocated from Shared Pool memory up to 10%

1.2: Oracle Database

Program Global Area

- Program Global Area (PGA) is the memory reserved for each user process that connects to an Oracle database.

Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 21

Program Global Area (PGA):

- The Program Global Area is part of memory allocated outside the SGA. It is sometimes also termed as Process Global Area.
- The PGA stores data and control information related to Server Process.
- Unlike the SGA, the PGA is an area that is stored only by one process.
- The PGA contains the following
 - Stack Space: Contains Session variables
 - Cursor State: Indicates the stage in the processing of the SQL statements that are currently used by the session
 - Session Information: Includes privileges for user, session variables, performance statistics for the session
 - Sort Area: Used for storing sorting information required to process the SQL statements
- In a shared server configuration some of these structures are stored in the SGA. As mentioned earlier in a shared server environment multiple user processes share one server process. If a large pool exists within SGA then Large Pool is used for storage else it is stored in Shared Pool.

1.2: Oracle Database

Program Global Area (Contd...)

- It is allocated when a process is created and deallocated when the process is terminated
- Can be sized with parameter PGA_AGGREGATE_TARGET

Process Structure

- An Oracle process is a program, which can request information, execute a series of steps, or perform a specific task, depending on its type.
- Oracle has the following types of processes:
 - **User process:** Starts at the time a database user requests connection to the Oracle server.
 - **Server process:** Connects to the Oracle Instance and starts when a user establishes a session.
 - **Background process:** Available when an Oracle instance is started.



Copyright © Capgemini 2015. All Rights Reserved 23

Process structure:

Oracle uses the following types of processes:

- **User Process**

A user process is a program that requests interaction with the Oracle server.

- It must first establish a connection.
- It does not interact directly with the Oracle server.

- **Server Process**

A server process is a program that directly interacts with the Oracle server.

- It fulfills calls generated and returns results.
- It can be dedicated or shared server.

- **Background Processes**

The relationship between the “physical structures” and “memory structures” is maintained and enforced by background processes in Oracle.

1.4: Process structure

Background Process

- The physical structure and memory structures are related to each other by Oracle background processes
 - Mandatory background processes:
 - SMON
 - PMON
 - DBWR
 - LGWR
 - CKPT
 - Optional background process:
 - Arcn

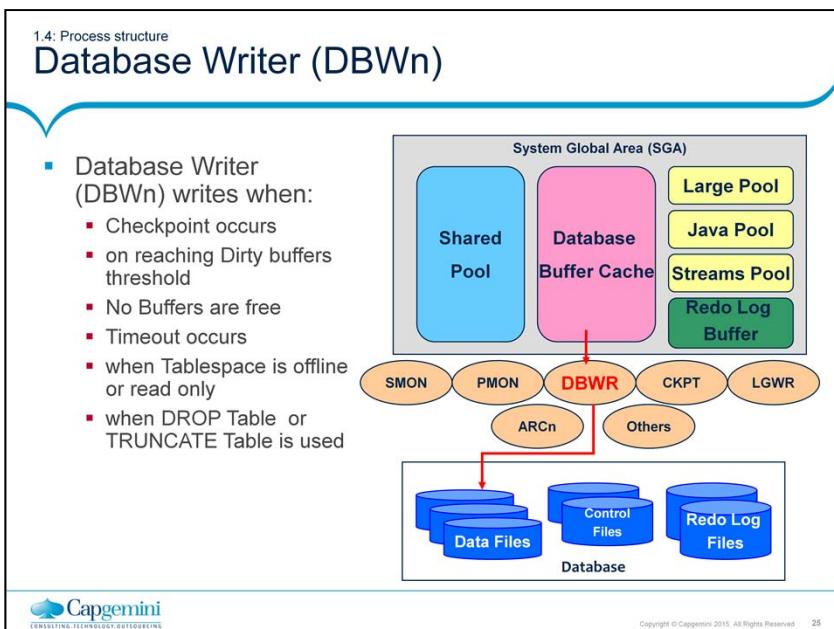


Copyright © Capgemini 2015. All Rights Reserved 24

Background Process:

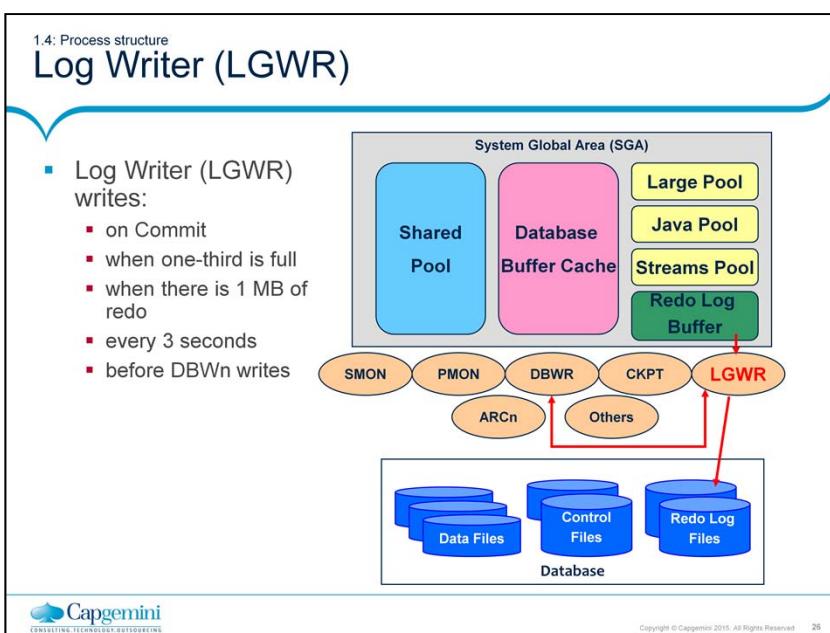
The Oracle architecture has five mandatory background process as mentioned on the slide.

Apart from that Oracle also has optional process which are available only if the option is being used in the Database. ARCh is the most common optional background process



Database Writer:

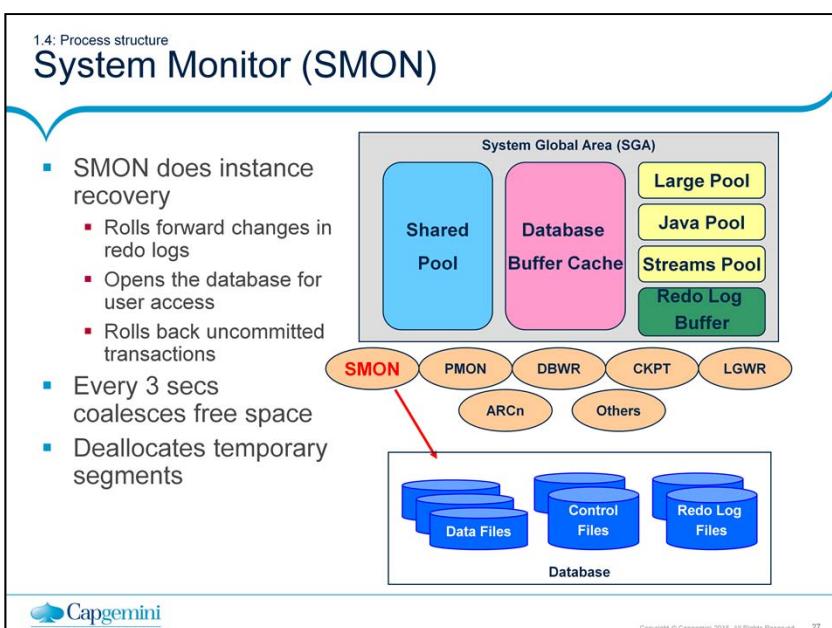
- The server process records changes to rollback and data blocks in the Buffer Cache. Database Writer (DBWn) writes the dirty buffers from the Database Buffer Cache to the Data files. To ensure that sufficient number of free buffers are available in DB cache is also taken care by DBWn. Free buffers are buffers that can be overwritten when server processes need to read in blocks from the data files.
- Database performance is improved because server processes make changes only in the Buffer Cache.
- DBWn delays writing to the data files until one of the following events occur:
 - System reaching incremental or normal Checkpoint
 - The number of dirty buffers reaching a threshold value
 - A process scanning a specified number of blocks, during the scan for free buffers, and cannot find any.
 - System placing a normal or temporary tablespace offline.
 - System placing a tablespace in read-only mode.
 - System is dropping or truncating a table.



LOG Writer

- The Log Writer (LGWR) performs sequential writes from the Redo Log Buffer Cache to the Redo Log File under the following situations:
 - when a transaction commits
 - when the Redo Log Buffer Cache is one-third full
 - when there is more than one megabyte of change records in the Redo Log Buffer Cache
 - every 3 seconds
 - before DBWn writes modified blocks in the DB Cache to the data files
- Since the redo is needed for recovery, LGWR confirms the commit only after the redo is written to disk.
- LGWR can also call on DBWn to write to the data files.

Note: DBWn does not write to the online redo logs.



System Monitor (SMON):

- When the Oracle instance fails due to whatever reason, any information in the SGA which has not been written to disk is lost. SMON automatically performs instance recovery when the database is reopened. Whenever the instance recovery happens the following tasks are carried out:
 - Rolling forward to recover data that has not been recorded in the data files but has been recorded in the online redo logs. Due to SGA failure data is not written to disk. During this process SMON reads the redo files and applies changes from the redo logs to data blocks. Since all the committed transactions are have been written to redo logs this process completely recovers transactions.
 - Opens the database so that users can log on and any data that is not locked by unrecovered transactions is available immediately
 - Rolling back uncommitted transactions. They are rolled back by SMON or by individual server processes as they access locked data.
- Apart from instance recover which is the main function of SMON, it also performs some maintenance tasks
 - It combines adjacent free space in the data files
 - It deallocates temporary segments to return them as free space data files.

1.4: Process structure

Process Monitor (PMON)

- Process Monitor (PMON)
 - PMON cleans up after failed processes, by:
 - rolling back the transaction
 - releasing locks
 - releasing other resources

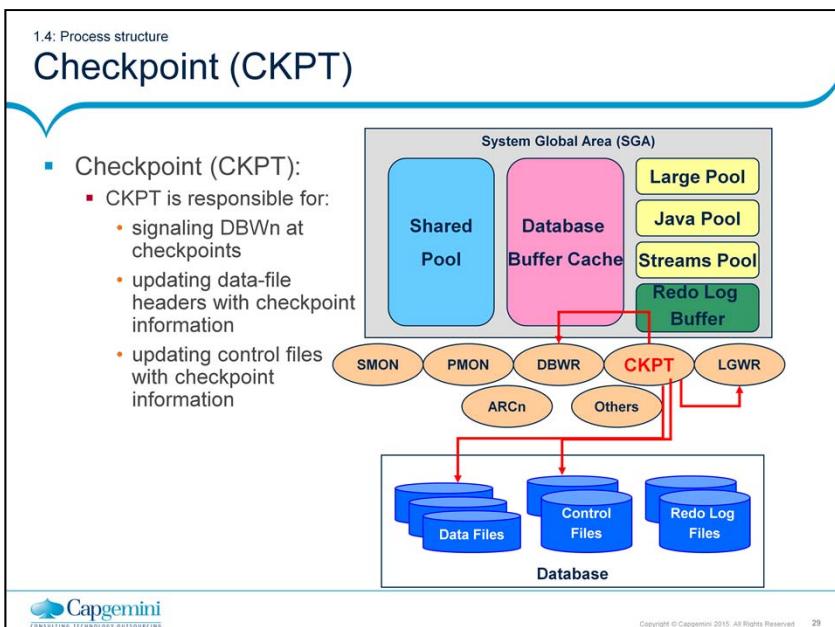
The diagram illustrates the Oracle process structure. At the top, the System Global Area (SGA) is shown as a large grey box containing two main components: the Shared Pool (blue) and the Database Buffer Cache (pink). To the right of the SGA are five smaller boxes representing different pools: Large Pool (yellow), Java Pool (light blue), Streams Pool (orange), Redo Log Buffer (green), and LGWR (dark green). Below the SGA, several background processes are represented by orange ovals: SMON, PMON (highlighted in red), DBWR, CKPT, and LGWR. ARCo is also shown. A red arrow points from PMON to the SGA. Another red arrow points from PMON to a box labeled 'PGA' at the bottom.

Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 28

Process Monitor:

- The PMON background process cleans up after the failed processes, by:
 - rolling back the current transaction of the user
 - releasing all currently held table or row locks
 - freeing other resources that are currently reserved by the user



Checkpoint (CKPT):

- An event called as a Checkpoint occurs when the Oracle background process DBWn writes all the modified Database Buffers in the SGA, including both committed and uncommitted data, to the data files.
- Checkpoints are implemented for the following reasons:
 - Checkpoints ensure that memory data blocks, which frequently change, are regularly written to data files.
 - Because of the Least Recently Used (LRU) algorithm of DBWn, a data block that changes frequently might never qualify as the least recently used block, and thus might never be written to disk if checkpoints do not occur.
 - In case, instance recovery is required, the redo log entries before the Checkpoint no longer need to be applied to the data files, since all database changes up to the Checkpoint have been recorded in the data files. Hence, checkpoints are useful because they can expedite instance recovery.
- **Note:** CKPT does not write data blocks to disk, or redo blocks to the online redo logs.

1.4: Process structure

Archiver (ARCn)

- Archiver (ARCn):**
 - It is an optional background process, responsible for:
 - Automatically archiving online redo logs when ARCHIVELOG mode is set
 - Preserving the record of all changes made to the database

System Global Area (SGA)

Shared Pool

Database Buffer Cache

Large Pool

Java Pool

Streams Pool

Redo Log Buffer

SMON

PMON

DBWR

CKPT

LGWR

ARCn

Others

Redo Log Files

Archive Redo Log files

Copyright © Capgemini 2015. All Rights Reserved 30

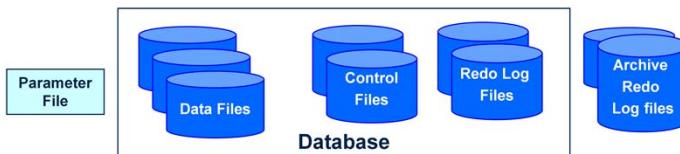
Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Archiver (ARCn)

- Archiver (ARCn) is an optional background process. It is responsible for:
 - Automatically archiving online redo logs when ARCHIVELOG mode is set
 - Preserving the record of all changes made to the database
- ARCn is crucial for recovering a database after the loss of a disk. As online redo log files are filled, the Oracle server begins writing to the next online redo log file. The process of switching from one redo log to another is called a "log switch".
 - The ARCn process initiates backing up or archiving, of the filled log group, at every log switch.
 - It automatically archives the online redo log before the log can be reused, such that all the changes made to the database are preserved.
 - This enables the DBA to recover the database to the point of failure, even if a disk drive is damaged.
- One of the important decisions that a DBA has to make is whether to configure the database to operate in ARCHIVELOG or in NOARCHIVELOG mode.
 - In NOARCHIVELOG mode, the online redo log files are overwritten each time a log switch occurs. LGWR does not overwrite a redo log group until the checkpoint for that group is complete. This ensures that committed data is recovered in case there is an instance crash.

1.2: Oracle Database Physical Structure

- An Oracle database:
 - is a collection of data that is treated as a unit.
 - consists of mainly three file types.



Oracle Database:

- An Oracle database has a “logical structure” and a “physical structure”.

1.2: Oracle Database

Physical Structure (Contd...)

- Physical structure:

- The physical structure of an Oracle database is determined by the Operating System files that provide the actual physical storage for database information, namely:
 - Control files
 - Data files
 - Redo log files



Copyright © Capgemini 2015. All Rights Reserved 32

- **Physical structure:**

- The physical structure of the database is the set of “operating system files” in the database.
- An Oracle database consists of three file types.
 - **Control files** that contain information necessary to maintain and verify database integrity.
 - **Data files** that contain the actual data in the database.
 - **Redo logs** that contain a record of changes made to the database to enable recovery of the data in case of failures.
- However, the physical structure of an Oracle database includes only three types of files: control files, data files, and redo log files.

Other Key File Structures

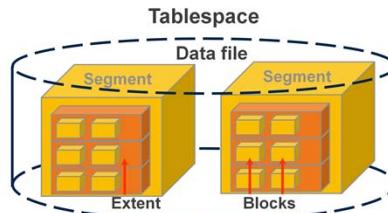
- The Oracle server uses other files, as well, that are not part of the database.
 - **Parameter file** defines the characteristics of an Oracle instance. For example: The parameter file contains “parameters” that size some of the memory structures in the SGA.

To start an instance, the Oracle server must read the “initialization parameter file”.
There are two types of “initialization parameter files”:

 - **Static parameter file**, PFILE, commonly referred to as “initSID.ora”. The PFILE is a text file that can be maintained by using a standard operating system editor. The parameter file is read-only during “instance startup”. If the file is modified, the instance must be shut down, and restarted in order to make the new parameter values effective.
 - **Persistent parameter file**, SPFILE, commonly referred to as “spfileSID.ora”. SPFILE, new to Oracle9i, is a binary file. The file is not meant to be manually modified and must always reside on the “server side”. By default, the file is located in \$ORACLE_HOME/dbs, and has a default name in the format of “spfileSID.ora”. Once the file is created it is maintained by the Oracle server. The SPFILE provides the ability to make changes to the database that are persistent across shutdown and startup.
 - **Password file** authenticates the users, who are privileged to start up and shut down an Oracle instance.
 - **Archived redo log files** are offline copies of the redo log files that may be necessary to recover from media failures.

1.2: Oracle Database Logical Structure

- The “logical structure” of the Oracle architecture dictates how the “physical space” of a database should be used.
- A hierarchy exists in this structure that consists of tablespaces, segments, extents, and blocks.



Logical Structure (Contd...)

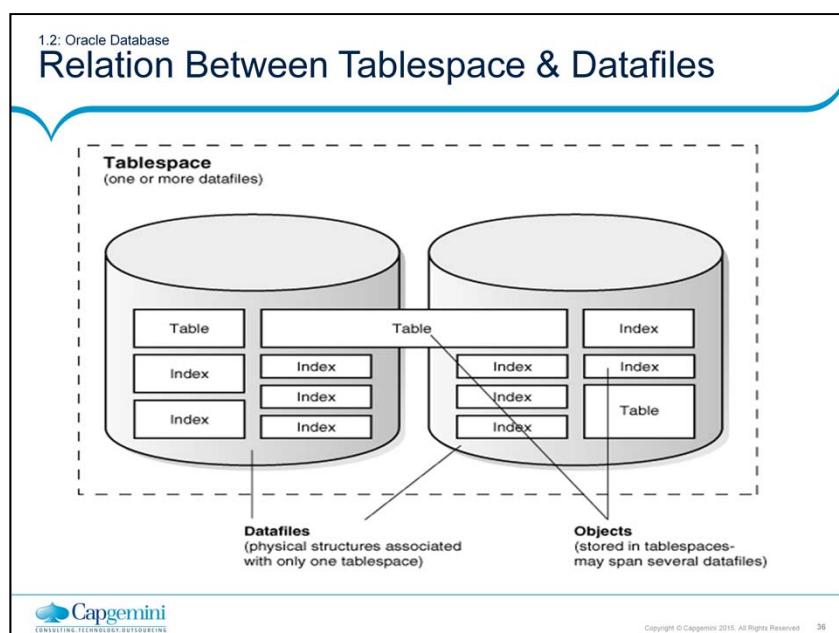
- Oracle stores data - logically in “ tablespaces ”, and physically in “ datafiles ” associated with the corresponding tablespace.
 - An Oracle database consists of one or more logical storage units called tablespaces, which collectively store all the data in the database.
 - Tablespaces are further divided into logical units of storage called “ Segments ”.
 - “ Segments ” are further divided into “ Extents ”.
 - “ Extents ” are a collection of “ contiguous blocks ”.

Logical Structure:

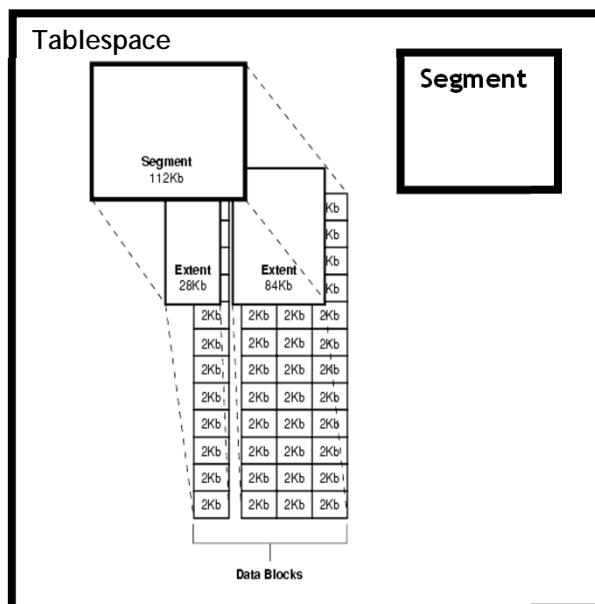
- Oracle stores data - logically in “ tablespaces ”, and physically in “ datafiles ” associated with the corresponding tablespace.
- An Oracle database consists of one or more logical storage units called as “ tablespaces ”, which collectively store all the data of the database.
- Each “ tablespace ” in an Oracle database consists of one or more files called “ datafiles ”, which are physical structures that conform to the operating system in which Oracle is running.
 - A database's data is collectively stored in the datafiles that constitute each tablespace of the database.

For example: The simplest Oracle database will have one tablespace and one datafile.

- Tablespaces are divided into logical units of storage called “ segments ”.
- Segments are further divided into “ Extents ”.
- Extents are a collection of “ contiguous blocks ”.



Logical components of an Oracle Database:



1.2: Oracle Database Schemas

- An Oracle database contains many schemas.
- A “schema” is a logical structure that contains objects like segments, views, procedures, functions, packages, triggers, user-defined objects, collection types, sequences, synonyms, and database links.
 - A “segment” is a data structure that can be a table, index, or temporary or undo segment.
 - The schema name is the user that controls the schema.
 - **Examples of schemas:** System, Sys, Scott, and SH schemas

1.2: Oracle Database

Schemas and Tablespaces

- Relationship between Schemas and Tablespace:

The diagram illustrates the hierarchical structure of Oracle database components:

- Schema 1** contains **Segment 1**, which is defined as a "Table or partition or cluster or index or temporary or undo segment".
- Schema 2** contains multiple segments: **Segment n**, **Segment 4**, **Segment 3**, and **Segment 2**.
- Segment 2** is further divided into **Extent 1**, **Extent 2**, **Extent 3**, and **Extent 4**.
- Extent 1** and **Extent 2** are composed of **Blocks**.
- Datafile 1** and **Datafile 2** belong to **Tablespace 1**.
- Arrows indicate the mapping from **Segment 1** to **Datafile 1** and **Datafile 2**, and from **Segment 2** to **Datafile 1** and **Datafile 2**.

Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 38

Relationship between Schemas and Tablespace:

- As shown in the slide, a “schema” can have many “segments” and many “segment types”.
 - Each segment is a single instance of a table, partition, cluster, index, or temporary or undo segment.
- For example:** A table with two indexes is implemented as three segments in the schema.
- A “Segment” is further broken down into “Extents”, which are a collection of contiguous “data blocks”.
 - As data is added to Oracle, it will first fill the blocks in the allocated Extents. Once these extents are full, new extents can be added to the segment as long as it is allowed by the available space.

1.2: Oracle Database

Schemas and Tablespaces (Contd...)

- However, each partition is itself a “segment”, and each segment can only reside in one “tablespace”.
- “Clustered tables” are another special case where two tables with a close link between them can have their data stored together in a “single block” to improve join operations.
- “Indexes” are optionally built on tables to help improve performance and to help implement “integrity constraints” such as primary keys and uniqueness.

1.2: Oracle Database

Schemas and Tablespaces (Contd...)

- “Temporary segments” are used as a temporary storage area by Oracle to run an SQL statement.
 - For example: Temporary segments may be used for sorting data, and then discarded once a query or transaction is complete.
- “Undo Segments” or “Rollback Segments” are used:
 - to manage the before image of changes to allow data to roll back, if required, and
 - to help provide data consistency for users querying data that is being changed.

1.2: Oracle Database

RowID

- RowID:

- Oracle has its own way of storing the data.
- To retrieve the data quickly, Oracle assigns each row with a unique ROWID.
 - The ROWID does not change throughout the life of the row.
 - Oracle always retrieves the row using the ROWID.
 - When you create an Index, Oracle stores the key column and the ROWID for that row in the index.
 - ROWID cannot be used for computation and it is HEX value

1.2: Oracle Database

Characteristics

- Characteristics of ROWID are:
 - ROWID provides the fastest access to a row.
 - It stores the “disk block address” where the row is stored.
 - The ROWID of a row does not change ever for a row as long as it exists
 - All Oracle applications such as Forms, Reports, PL/SQL use ROWID to access, lock, and update rows.
 - ROWID of a row does not change unless you export and import the table.
 - As a programmer you will always try to use the primary key to access the row. It is not recommended that the programmer explicitly uses the ROWID.

1.2: Oracle Database

Types of RowID

- Types of ROWIDs are:

- Oracle has two different representations of ROWIDS – namely Restricted ROWID and Extended ROWID.

- **Restricted ROWID:**

- It uses a binary representation, to store the ROWID (discontinued in Oracle 8i)
- When ROWID is used in SQLPLUS, it is converted into a varchar2 format and displayed.
- The format of restricted ROWID is:
 - Block.row.file
 - For example: 00000DD5.00000.0001

1.2: Oracle Database

Types of RowID (Contd...)

▪ Extended ROWID:

- Oracle 8i and above, always uses Extended ROWID for storing rows.
 - Extended ROWID uses a 64 bit representation of every row. It has the following format:
 - OOOOOOFFBBBBBBBRRR
- where:
- OOOOOO: Data object number
 - FFF: Tablespace relative datafile number
 - BBBBBB: Data block number within that file
 - RRR: Row in that file
- For example: AAAAaoATAAAABrXAAA

1.2: Oracle Database

Types of RowID (Contd...)

▪ Note:

- The ROWID of a record is the fastest method of record retrieval.
- The performance can be improved by selecting a record before updating or deleting it and including ROWID in the initial selection list.

Summary

- In this lesson you have learnt about:

- Concept of Database file
 - Data files
 - Control files
 - Online redo logs

- Concept of SGA memory structure:
 - Database Buffer cache
 - Shared SQL Pool
 - Redo Log Buffer



Summary

Review Question

- Question 1: The “Oracle server” consists of an ___ and an ___.
- Question 2: ___ provide the actual physical storage for database information.
- Question 3: A single Oracle9i instance can open multiple databases.
 - True / False



Review Question

- Question 4: The Background process communicates with the Oracle instance on behalf of the “user process”, which runs on the client.
 - True / False

- Question 5: Redo logs contain a record of changes made to the database.
 - True / False



Review Question: Match the Following

1. Java Pool

2. Large Pool

3. SGA

4. PGA

a. is allocated when the database instance is started

b. is sized by the JAVA_POOL_SIZE parameter

c. is allocated when a user process is created

d. is an optional area of memory in the shared global area



Oracle (PL/SQL)

Lesson 2: Introduction to Data Dictionary

Lesson Objectives

- To understand the following topics:
 - Identifying key data dictionary components
 - Identifying the contents and uses of Data Dictionary
 - Querying the Data Dictionary



2.1: Data Dictionary

Introduction to Data Dictionary

- Oracle uses the term “Data Dictionary” for its system catalogs.
- Each Oracle database has its own set of “system tables” and “views”, which store information about both the physical and logical database structure.
- The Data Dictionary objects are read-only.
 - That is to say, no database user ever manually modifies these objects.
 - However, Oracle RDBMS itself automatically updates data in these objects in response to specific actions.



Copyright © Capgemini 2015. All Rights Reserved 3

Introduction to Data Dictionary:

- **For example:** Suppose an user USER1 creates a new object (table, view, stored procedure, etc.), adds a column or a constraint to a table, and so forth. Then the appropriate Data Dictionary tables are updated at once behind the scenes, and the corresponding changes are visible through the system views.

2.1: Data Dictionary

Contents of Data Dictionary

- The Data Dictionary contains:
 - Definitions of all schema objects in the database
 - Information about the amount of space that is allocated for, and is currently used by, the schema objects
 - Default values for columns
 - Information about Integrity Constraints
 - Names of Oracle users
 - Privileges and roles that have been granted to each user
 - Auditing information
 - Other database information in general

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 4

Contents of Data Dictionary:

- It contains definitions of all schema objects in the database (tables, views, indexes, clusters, synonyms, sequences, procedures, functions, packages, triggers, and so on), auditing information, such as who has accessed or updated various other general database information, etc.
- The Data Dictionary provides information about:
 - Logical and physical database structure
 - Definitions and space allocations of objects
 - Integrity constraints
 - Users
 - Roles
 - Privileges
 - Auditing

2.1: Data Dictionary

Structure of Data Dictionary

- During database creation, the Oracle server creates the following additional object structures within the data files:
 - Data Dictionary Tables
 - Dynamic Performance Tables

```
graph TD; subgraph OracleDatabase [Oracle Database]; direction LR; A[Data Files] --- OracleDatabase; B[Control Files] --- OracleDatabase; C[Redo log Files] --- OracleDatabase; end; D[Data Dictionary & Dynamic Performance Tables] --- OracleDatabase;
```

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 5.

Structure of Oracle Data Dictionary:

- The Data Dictionary is structured in Tables and Views, just like other database data.
- All the Data Dictionary tables and views, for a given database, are stored in that "SYSTEM tablespace" of the database.
- During database creation, the Oracle server creates additional object structures within the data files.
 - Data Dictionary Tables
 - Dynamic Performance Tables

2.2: Structure of Oracle Data Dictionary

Data Dictionary Tables

- The Data Dictionary is a set of read-only “tables” and “views”, which record, verify, and provide information about its associated database.
- Data Dictionary describes the database and its objects.
- Data Dictionary includes two types of objects:
 - Base tables
 - Store description of database
 - Created with CREATE DATABASE
 - Data Dictionary views
 - Summarize base table information
 - Created using catalog.sql script

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 6

Data Dictionary Tables:

- Data Dictionary Tables include two types of objects:

➤ **Base tables:**

Base Tables are underlying tables, which store information about the database. The base tables are the first objects created in any Oracle database. They are automatically created when the Oracle server runs the “sql.bsq script” at the time the database is created. Only the Oracle server should write to these tables. Users rarely access them directly, because most of the data is stored in a cryptic format that is difficult to understand.

➤ **Data Dictionary views:**

These user accessible views summarize base table information and are created by using “catalog.sql script”. They display the information stored in the base tables in readable and / or simplified form by using joins, column aliases, and so on.

2.2: Structure of Oracle Data Dictionary

How is the Data Dictionary Used?

- Three primary uses of Data Dictionary:
 - Used by Oracle Server to find information about:
 - Users
 - Schema objects
 - Storage structures
 - Modified by Oracle Server when a DDL statement is executed
 - Used by users and DBAs as a read-only reference for information about the database.

2.2: Structure of Oracle Data Dictionary

Data Dictionary View Categories

- Data Dictionary View categories are:
 - The Data Dictionary consists of three main sets of “static views”, which are distinguished from each other by their scope:
 - DBA: What is in all the schemas?
 - ALL: What can the user access?
 - USER: What is in the user's schema?

- DBA_xxx**
All of the objects in the database
- ALL_xxx**
Objects accessible by the current user
- USER_xxx**
Objects owned by the current user

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 8

Data Dictionary View Categories:

- The Data Dictionary consists of three main sets of “static views”, which are distinguished from each other by their scope:

➤ **DBA: What is in all the schemas?**

It displays all relevant information in the entire database. DBA_ views are intended only for administrators. They can be accessed only by users with the SELECT_ANY_TABLE privilege. (This privilege is assigned to the DBA role when the system is initially installed.)

➤ **ALL: What can the user access?**

It displays all the information accessible to the current user, including information from the schema of the current user. It displays information from objects in other schemas, as well, if the current user has access to those objects by way of grants of privileges or roles.

➤ **USER: What is in the user's schema?**

It displays all the information from the schema of the current user. No special privileges are required to query these Views.

contd.

Data Dictionary View Categories (contd.):

- **For example:**

- The following query returns all the objects contained in the users schema:

```
SELECT owner, object_name, object_type FROM  
users_objects;
```

- The following query returns information about all the objects to which you have access:

```
SELECT owner, object_name, object_type FROM  
ALL_OBJECTS;
```

- Data Dictionary Views are “static views” that answer questions such as:
 - Was the object ever created?
 - What is the object a part of?
 - Who owns the object?
 - What privileges do users have?
- The name "static" denotes that the information in this group of Views changes only when a change is made to the Data Dictionary (for example: a column is added to a table, a new database user is created, etc).
- The “dynamic views” are constantly updated while a database is in use.
- Examples of Data Dictionary Views for the information on Schema objects:
 - USER_TABLES
 - USER_INDEXES
 - USER_TAB_COLUMNS
 - ALL_CONSTRAINTS

2.2: Structure of Oracle Data Dictionary

Dynamic Performance Tables

- Dynamic Performance Tables:
 - Dynamic Performance Views record current database activity.
 - Views are continually updated while the database is operational.
 - Information is accessed from:
 - Memory
 - Control file
 - Dynamic Views are used by DBA to monitor and tune the database
 - Dynamic Views are owned by SYS user
 - DML is not allowed



Copyright © Capgemini 2015. All Rights Reserved 10

Dynamic Performance Tables:

- Dynamic Performance Views record current database activity.
 - Views are continually updated while the database is operational
 - Information is accessed from:
 - Memory
 - Control file
 - Dynamic Views are used by DBA to monitor and tune the database
 - Dynamic Views are owned by SYS user
 - DML is not allowed
- These virtual tables exist in memory only when the database is running, to reflect real-time conditions of the database operation. They point to actual sources of information in "memory" and the "control file".
- SYS owns the Dynamic Performance tables. Their names begin with V\$.
- Views are created on these tables, and then public synonyms are created for the views. The synonym names begin with V\$.

For example:

- The V\$DATAFILE view contains information about the datafiles in the database, and
- The V\$FIXED_TABLE view contains information about all of the dynamic performance tables and views in the database.

contd.

Dynamic Performance Tables (contd.):

- The Dynamic Tables answer questions such as:
 - Is the object online and available?
 - Is the object open?
 - What locks are being held?
 - Is the session active?
- To display the backup status of all online datafiles, V\$BACKUP dynamic performance table can be queried. This view shows the datafile status, such as, backup is in progress, offline, description of error.
- To get an overview of the data dictionary, the DICTIONARY view or its synonym DICT can be queried.

For example:

- To narrow your responses, you can include the where clause :

```
SELECT * FROM dictionary;
```

- To get an overview of the columns in the Data Dictionary and Dynamic Performance views, the DICT_COLUMNS view can be queried.

```
SELECT * FROM dictionary WHERE table_name  
like 'EMP%'
```

2.2: Structure of Oracle Data Dictionary

Some Data Dictionary tables

- Given below are a few examples of Data Dictionary:
 - General Overview
 - DICTONARY, DICT_COLUMNS
 - Schema objects
 - DBA_TABLES, DBA_INDEXES, DBA_TAB_COLUMNS, DBA_CONSTRAINTS
 - Space allocation
 - DBA_SEGMENTS, DBA_EXTENTS
 - Database structure
 - DBA_TABLESPACES, DBA_DATA_FILES

2.2: Structure of Oracle Data Dictionary

Using the Data Dictionary Tables

■ Example 1:

- To list details of tables owned by current user:

```
SQL>SELECT TABLE_NAME, TABLESPACE_NAME, BLOCKS FROM  
USER_TABLES;
```

■ Example 2:

- The USER_ERRORS data dictionary view contains information about the last error that occurred in a user's schema.

```
SQL>SELECT * FROM USER_ERRORS;
```



Copyright © Capgemini 2015. All Rights Reserved 13

Summary

- In this lesson you have learnt about:
 - Use of the Data Dictionary Views to retrieve information about the database and instance
 - Obtaining information about Data Dictionary Views from DICTIONARY and DICT_COLUMNS



Review Question

- Question 1: Data Dictionary contains definitions of all schema objects in the database.
 - True / False
- Question 2: DBA_ views are intended for all users in database.
 - True / False
- Question 3: Data dictionary view category ALL_xxx displays all the information from the schema of the current user.
 - True / False



Oracle (PL/SQL)

Lesson 3: Introduction to PL/SQL

Lesson Objectives

- To understand the following topics:
 - Features of PL/SQL
 - PL/SQL Block structure
 - Handling variables in PL/SQL
 - Declaring a PL/SQL table
 - Variable scope and Visibility
 - SQL in PL/SQL
 - Programmatic Constructs



3.1: Introduction to PL/SQL

Overview

- PL/SQL is a procedural extension to SQL.
 - The “data manipulation” capabilities of “SQL” are combined with the “processing capabilities” of a “procedural language”.
 - PL/SQL provides features like conditional execution, looping and branching.
 - PL/SQL supports subroutines, as well.
 - PL/SQL program is of block type, which can be “sequential” or “nested” (one inside the other).



Copyright © Capgemini 2015. All Rights Reserved 3

Introduction to PL/SQL:

- PL/SQL stands for Procedural Language/SQL. PL/SQL extends SQL by adding constructs found in procedural languages, resulting in a structural language that is “more powerful than SQL”.
 - With PL/SQL, you can use SQL statements to manipulate Oracle data and flow-of-control statements to process the data.
 - Moreover, you can declare constants and variables, define procedures and functions, and trap runtime errors.
 - Thus PL/SQL combines the “data manipulating power” of SQL with the “data processing power” of procedural languages.
- PL/SQL is an “embedded language”. It was not designed to be used as a “standalone” language but instead to be invoked from within a “host” environment.
 - You cannot create a PL/SQL “executable” that runs all by itself.
 - It can run from within the database through SQL*Plus interface or from within an Oracle Developer Form (called client-side PL/SQL).

3.1: Introduction to PL/SQL

Features of PL/SQL

- PL/SQL provides the following features:
 - Tight Integration with SQL
 - Better performance
 - Several SQL statements can be bundled together into one PL/SQL block and sent to the server as a single unit.
 - Standard and portable language
 - Although there are a number of alternatives when it comes to writing software to run against the Oracle Database, it is easier to run highly efficient code in PL/SQL, to access the Oracle Database, than in any other language.

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 4

Features of PL/SQL

- Tight Integration with SQL:
 - This integration saves both, your learning time as well as your processing time.
 - PL/SQL supports SQL data types, reducing the need to convert data passed between your application and database.
 - PL/SQL lets you use all the SQL data manipulation, cursor control, transaction control commands, as well as SQL functions, operators, and pseudo columns.
- Better Performance:
 - Several SQL statements can be bundled together into one PL/SQL block, and sent to the server as a single unit.
 - This results in less network traffic and a faster application. Even when the client and the server are both running on the same machine, the performance is increased. This is because packaging SQL statements results in a simpler program that makes fewer calls to the database.
- Portable:
 - PL/SQL is a standard and portable language.
 - A PL/SQL function or procedure written from within the Personal Oracle database on your laptop will run without any modification on your corporate network database. It is “Write once, run everywhere” with the only restriction being “everywhere” there is an Oracle Database.
- Efficient:
 - Although there are a number of alternatives when it comes to writing software to run against the Oracle Database, it is easier to run highly efficient code in PL/SQL, to access the Oracle Database, than in any other language.

3.1: Introduction to PL/SQL

PL/SQL Block Structure

- A PL/SQL block comprises of the following structure:
 - DECLARE – Optional
 - Variables, cursors, user-defined exceptions
 - BEGIN – Mandatory
 - SQL statements
 - PL/SQL statements
 - EXCEPTION – Optional
 - Actions to perform when errors occur
 - END; – Mandatory

DECLARE
...
BEGIN
...
EXCEPTION
...
END;

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

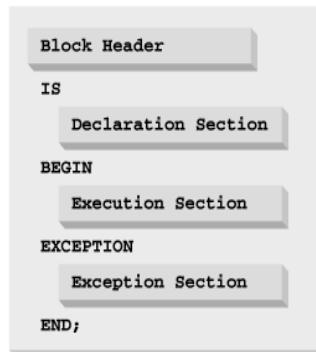
Copyright © Capgemini 2015. All Rights Reserved 5

PL/SQL Block Structure:

- PL/SQL is a block-structured language. Each basic programming unit that is written to build your application is (or should be) a “logical unit of work”. The PL/SQL block allows you to reflect that logical structure in the physical design of your programs.
- Each PL/SQL block has up to four different sections (some are optional under certain circumstances).

contd.

PL/SQL block structure (contd.):



- **Header**

It is relevant for named blocks only. The header determines the way the named block or program must be called.

- **Declaration section**

The part of the block that declares variables, cursors, and sub-blocks that are referenced in the Execution and Exception sections.

- **Execution section**

It is the part of the PL/SQL blocks containing the executable statements; the code that is executed by the PL/SQL runtime engine.

- **Exception section**

It is the section that handles exceptions for normal processing (warnings and error conditions).

3.2: PL/SQL Block Structure

Block Types

- There are three types of blocks in PL/SQL:
 - Anonymous
 - Named:
 - Procedure
 - Function

Anonymous	Procedure	Function
<pre>[DECLARE] BEGIN --statements [EXCEPTION] END;</pre>	<pre>PROCEDURE name IS BEGIN --statements [EXCEPTION] END;</pre>	<pre>FUNCTION name RETURN datatype IS BEGIN --statements RETURN value; [EXCEPTION] END;</pre>

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 7

Block Types:

- The basic units (procedures and functions, also known as subprograms, and anonymous blocks) that make up a PL/SQL program are “logical blocks”, which can contain any number of nested sub-blocks.
- Therefore one block can represent a small part of another block, which in turn can be part of the whole unit of code.

➤ **Anonymous Blocks**

Anonymous blocks are unnamed blocks. They are declared at the point in an application where they are to be executed and are passed to the PL/SQL engine for execution at runtime.

➤ **Named :**

▪ **Subprograms**

Subprograms are named PL/SQL blocks that can take parameters and can be invoked. You can declare them either as “procedures” or as “functions”.

Generally, you use a “procedure” to perform an “action” and a “function” to compute a “value”.

Representation of a PL/SQL block:

```
DECLARE          -- Declaration Section
    V_Salary  NUMBER(7,2);
/* V_Salary is a variable declared in a PL/SQL block. This variable is
used to store JONES' salary. */
Low_Sal EXCEPTION;           -- an exception
BEGIN                  -- Execution Section
    SELECT sal INTO V_Salary

    FROM emp WHERE ename = 'JONES'
        FOR UPDATE of sal ;
        IF V_Salary < 3000 THEN
            RAISE Low_Sal ;
        END IF;
EXCEPTION          -- Exception Section
    WHEN Low_Sal THEN
        UPDATE emp SET sal = sal + 500 WHERE ename = 'JONES' ;
END ;                -- End of Block
/                   -- PL/SQL block terminator

Output
SQL> /
PL/SQL procedure successfully completed.
```

- The notations used in a PL/SQL block are given below:
 1. -- is a single line comment.
 2. /* */ is a multi-line comment.
 3. Every statement must be terminated by a semicolon (;).
 4. PL/SQL block is terminated by a slash (/) on a line by itself.
- A PL/SQL block must have a “Declaration section” and an “Execution section”. It can optionally have an Exception section, as well.

3.3: Handling Variables in PL/SQL

Points to Remember

- While handling variables in PL/SQL:
 - declare and initialize variables within the declaration section
 - assign new values to variables within the executable section
 - pass values into PL/SQL blocks through parameters
 - view results through output variables

3.3: Handling Variables in PL/SQL

Guidelines

- Given below are a few guidelines for declaring variables:
 - follow the naming conventions
 - initialize the variables designated as NOT NULL
 - initialize the identifiers by using the assignment operator (:=) or by using the DEFAULT reserved word
 - declare at most one Identifier per line

3.3: Handling Variables in PL/SQL

Types of Variables

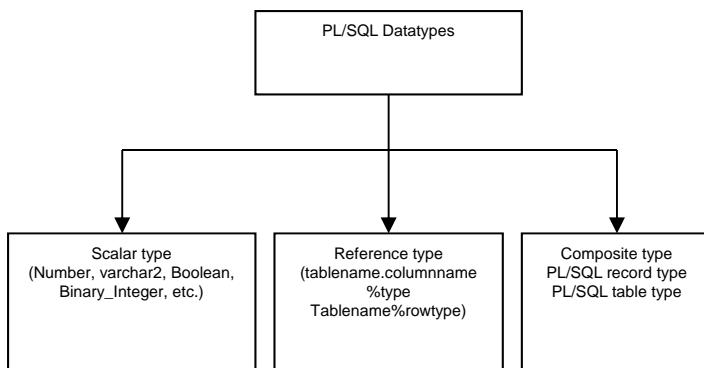
- PL/SQL variables
 - Scalar
 - Composite
 - Reference
 - LOB (large objects)
- Non-PL/SQL variables
 - Bind and host variables

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 11

Types of Variables: PL/SQL Datatype:

- All PL/SQL datatypes are classified as scalar, reference and Composite type.
- Scalar datatypes do not have any components within it, while composite datatypes have other datatypes within them.
- A reference datatype is a pointer to another datatype.



3.3: Handling Variables in PL/SQL

Declaring PL/SQL variables

▪ Syntax

```
identifier [CONSTANT] datatype [NOT NULL]
[:= | DEFAULT expr];
```

▪ Example

```
DECLARE
    v_hiredate      DATE;
    v_deptno        NUMBER(2) NOT NULL := 10;
    v_location       VARCHAR2(13) := 'Atlanta';
    c_comm CONSTANT NUMBER := 1400;
```

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 12

Declaring PL/SQL Variables:

- You need to declare all PL/SQL identifiers within the “declaration section” before referencing them within the PL/SQL block.
- You have the option to assign an initial value.
 - You do not need to assign a value to a variable in order to declare it.
 - If you refer to other variables in a declaration, you must separately declare them in a previous statement.
 - Syntax:

```
identifier [CONSTANT] datatype [NOT NULL]
[:= | DEFAULT expr];
```

- In the syntax given above:
 - **identifier** is the name of the variable.
 - **CONSTANT** constrains the variable so that its value cannot change. Constants must be initialized.
 - **datatype** is a scalar, composite, reference, or LOB datatype.
 - **NOT NULL** constrains the variable so that it must contain a value. NOT NULL variables must be initialized.
 - **expr** is any PL/SQL expression that can be a literal, another variable, or an expression involving operators and functions.

contd.

Declaring PL/SQL Variables (contd.):

For example:

```
DECLARE
    v_description  varchar2 (25);
    v_sal          number (5) not null;
    = 3000;
    v_compcode     varchar2 (20)
constant: = 'abc
                      consultants';
    v_comm         not null default  0;
```

3.3: Handling Variables in PL/SQL

Base Scalar Data Types

- Base Scalar Datatypes:
 - Given below is a list of Base Scalar Datatypes:
 - VARCHAR2 [(maximum_length)]
 - NUMBER [(precision, scale)]
 - DATE
 - CHAR [(maximum_length)]
 - LONG
 - LONG RAW
 - BOOLEAN
 - BINARY_INTEGER
 - PLS_INTEGER

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 14

Base Scalar Datatypes:

1. NUMBER

This can hold a numeric value, either integer or floating point. It is same as the number database type.

2. BINARY_INTEGER

If a numeric value is not to be stored in the database, the BINARY_INTEGER datatype can be used. It can only store integers from - 2147483647 to + 2147483647. It is mostly used for counter variables.

```
V_Counter BINARY_INTEGER DEFAULT 0;
```

3. VARCHAR2 (L)

L is necessary and is max length of the variable. This behaves like VARCHAR2 database type. The maximum length in PL/SQL is 32,767 bytes whereas VARCHAR2 database type can hold max 2000 bytes. If a VARCHAR2 PL/SQL column is more than 2000 bytes, it can only be inserted into a database column of type LONG.

4. CHAR (L)

Here L is the maximum length. Specifying length is optional. If not specified, the length defaults to 1. The maximum length of CHAR PL/SQL variable is 32,767 bytes, whereas the maximum length of the database CHAR column is 255 bytes. Therefore a CHAR variable of more than 255 bytes can be inserted in the database column of VARCHAR2 or LONG type.

contd.

3.3: Handling Variables in PL/SQL

Base Scalar Data Types - Example

- Here are a few examples of Base Scalar Datatypes:

```
v_job      VARCHAR2(9);
v_count    BINARY_INTEGER := 0;
v_total_sal NUMBER(9,2) := 0;
v_orderdate DATE := SYSDATE + 7;
c_tax_rate CONSTANT NUMBER(3,2) := 8.25;
v_valid    BOOLEAN NOT NULL := TRUE;
```



Copyright © Capgemini 2015. All Rights Reserved 15

Base Scalar Datatypes (contd.):

5. LONG

PL/SQL LONG type is just 32,767 bytes. It behaves similar to LONG DATABASE type.

6. DATE

The DATE PL/SQL type behaves the same way as the date database type. The DATE type is used to store both date and time. A DATE variable is 7 bytes in PL/SQL.

7. BOOLEAN

A Boolean type variable can only have one of the two values, i.e. either TRUE or FALSE. They are mostly used in control structures.

```
V_Does_Dept_Exist BOOLEAN := TRUE;
V_Flag BOOLEAN := 0; -- illegal
```

One more example

```
declare
    pie constant number := 7.18;
    radius number := &radius;
begin
    dbms_output.put_line('Area:
'||pie*power(radius,2));
    dbms_output.put_line('Diameter: '||
2*pie*radius);
end;
/
```

3.3: Handling Variables in PL/SQL

Declaring Datatype with %TYPE Attribute

- While using the %TYPE Attribute:
 - Declare a variable according to:
 - a database column definition
 - another previously declared variable
 - Prefix %TYPE with:
 - the database table and column
 - the previously declared variable name



Copyright © Capgemini 2015. All Rights Reserved 16

Reference types:

- A “reference type” in PL/SQL is the same as a “pointer” in C. A “reference type” variable can point to different storage locations over the life of the program.

Using %TYPE

- %TYPE is used to declare a variable with the same datatype as a column of a specific table. This datatype is particularly used when declaring variables that will hold database values.
- **Advantage:**
 - You need not know the exact datatype of a column in the table in the database.
 - If you change database definition of a column, it changes accordingly in the PL/SQL block at run time.
 - Syntax:

```
Var_Name    table_name.col_name%TYPE;  
V_Empno     emp.empno%TYPE;
```

- **Note:** Datatype of V_Empno is same as datatype of Empno column of the EMP table.

3.3: Handling Variables in PL/SQL

Declaring Datatype with %TYPE Attribute (Contd...)

- Example:

```
...
v_name          staff_master.staff_name%TYPE;
v_balance       NUMBER(7,2);
v_min_balance   v_balance%TYPE := 10;
...
```



Copyright © Capgemini 2015. All Rights Reserved 17

Using %TYPE (contd.)

- Example

```
declare
    nSalary employee.salary%type;
begin
    select salary into nsalary
    from employee
    where emp_code = 11;

    update employee set salary
    = salary + 101 where emp_code = 11;
end;
```

3.3: Handling Variables in PL/SQL

Declaring Datatype by using %ROWTYPE

- Example:

```

DECLARE
    nRecord staff_master%rowtype;
BEGIN
    SELECT * into nrecord
        FROM staff_master
        WHERE staff_code = 100001;

    UPDATE staff_master
    SET staff_sal = staff_sal + 101
    WHERE emp_code = 100001;

END;

```



Copyright © Capgemini 2015. All Rights Reserved 18

Using %ROWTYPE

- %ROWTYPE is used to declare a compound variable, whose type is same as that of a row of a table.
- Columns in a row and corresponding fields in record should have same names and same datatypes. However, fields in a %ROWTYPE record do not inherit constraints, such as the NOT NULL, CHECK constraints, or default values.
- **Syntax:**

Var_Name table_name%ROWTYPE;
V_Emprec emp%ROWTYPE;
- where V_Emprec is a variable, which contains within itself as many variables, whose names and datatypes match those of the EMP table.
- To access the Empno element of V_Emprec, use V_Emprec.empno;
- **For example:**

```

DECLARE emprec emp%rowtype;
BEGIN
    emprec.empno :=null;
    emprec.deptno :=50;
    dbms_output.put_line ('emprec.employee's
    number'||emprec.empno);
END;
/

```

3.3: Handling Variables in PL/SQL

Inserting and Updating using records

- Example:

```
DECLARE
    dept_info department_master%ROWTYPE;
BEGIN
    -- dept_code, dept_name are the table columns.
    -- The record picks up these names from the %ROWTYPE.
    dept_info.dept_code := 70;
    dept_info.dept_name := 'PERSONNEL';
    /*Using the %ROWTYPE means we can leave out the column list
    (deptno, dname) from the INSERT statement. */
    INSERT into department_master VALUES dept_info;
END;
```



Copyright © Capgemini 2015. All Rights Reserved 19

3.3: Handling Variables in PL/SQL

User-defined SUBTYPES

- User-defined SUBTYPES:

- User-defined SUBTYPES are subtypes based on an existing type.
 - They can be used to give an alternate name to a type.
 - Syntax:

```
SUBTYPE New_Type IS original_type;
```

- It can be a predefined type, subtype, or %type reference.

```
SUBTYPE T_Counter IS NUMBER;
```

```
V_Counter T_Counter;
```

```
SUBTYPE T_Emp_Record IS EMP%ROWTYPE;
```



Copyright © Capgemini 2015. All Rights Reserved 20

User-defined SUBTYPES:

- A SUBTYPE is a PL/SQL type based on an existing type. A subtype can be used to give an alternate name to a type to indicate its purpose.
- A new sub_type base type can be a predefined type, subtype, or %type reference.
- You can declare a dummy variable of the desired type with the constraint and use %TYPE in the SUBTYPE definition.

```
V_Dummy      NUMBER(4);
SUBTYPE T_Counter IS
V_Dummy%TYPE;

V_Counter    T_Counter ;
SUBTYPE T_Numeric IS NUMBER;

V_Counter IS T_Numeric(5);
```

3.3: Handling Variables in PL/SQL

User-defined SUBTYPES (Contd...)

- It is illegal to constrain a subtype.

```
SUBTYPE T_Counter IS NUMBER(4) -- Illegal
```

- Possible solutions:

```
V_Dummy NUMBER(4);
SUBTYPE T_Counter IS V_Dummy%TYPE;
V_Counter T_Counter ;
SUBTYPE T_Numeric IS NUMBER;
V_Counter IS T_Numeric(5);
```



Copyright © Capgemini 2015. All Rights Reserved 21

3.3: Handling Variables in PL/SQL

Composite Data Types

- Composite Datatypes in PL/SQL:
 - Two composite datatypes are available in PL/SQL:
 - records
 - tables
 - A composite type contains components within it. A variable of a composite type contains one or more scalar variables.



Copyright © Capgemini 2015. All Rights Reserved 22

Record Data Types

- Record Datatype:

- A record is a collection of individual fields that represents a row in the table.
- They are unique and each has its own name and datatype.
- The record as a whole does not have value.

- Defining and declaring records:

- Define a RECORD type, then declare records of that type.
- Define in the declarative part of any block, subprogram, or package.



Copyright © Capgemini 2015. All Rights Reserved 23

Record Datatype:

- A record is a collection of individual fields that represents a row in the table. They are unique and each has its own name and datatype. The record as a whole does not have value. By using records you can group the data into one structure and then manipulate this structure into one “entity” or “logical unit”. This helps to reduce coding and keeps the code easier to maintain and understand.

3.3: Handling Variables in PL/SQL

Record Data Types (Contd...)

- Syntax:

```
TYPE type_name IS RECORD (field_declaration [,field_
declaration] ...);
```



Copyright © Capgemini 2015. All Rights Reserved 24

Defining and Declaring Records

- To create records, you define a RECORD type, then declare records of that type. You can define RECORD types in the declarative part of any PL/SQL block, subprogram, or package by using the syntax.
- where field_declaration stands for:
 - field_name field_type [[NOT NULL] {:= | DEFAULT} expression]
 - type_name is a type specifier used later to declare records. You can use %TYPE and %ROWTYPE to specify field types.

3.3: Handling Variables in PL/SQL

Record Data Types - Example

- Here is an example for declaring Record datatype:

```
DECLARE  
TYPE DeptRec IS RECORD (  
Dept_id      department_master.dept_code%TYPE,  
Dept_name     varchar2(15),
```



Copyright © Capgemini 2015. All Rights Reserved 25

Record Datatype (contd.):

- **Field declarations** are like variable declarations.
- Each field has a unique name and specific datatype.
- Record members can be accessed by using “.” (Dot) notation.
- The value of a record is actually a collection of values, each of which is of some simpler type. The attribute %ROWTYPE lets you declare a record that represents a row in a database table.
- After a record is declared, you can reference the record members directly by using the “.” (Dot) notation. You can reference the fields in a record by indicating both the record and field names.

For example: To reference an individual field, you use the dot notation DeptRec.deptno;

- You can assign expressions to a record.

For example: DeptRec.deptno := 50;

- You can also pass a record type variable to a procedure as shown below:

```
get_dept(DeptRec);
```

3.3: Handling Variables in PL/SQL

Record Data Types - Example (Contd...)

- Here is an example for declaring and using Record datatype:

```
DECLARE
  TYPE recname is RECORD
    (customer_id number,
     customer_name varchar2(20));
  var_rec recname;
BEGIN
  var_rec.customer_id:=20;
  var_rec.customer_name:='Smith';
  dbms_output.put_line(var_rec.customer_id||
  '||var_rec.customer_name);
END;
```



Copyright © Capgemini 2015. All Rights Reserved 26

Table Data Type

- A PL/SQL table is:

- a one-dimensional, unbounded, sparse collection of homogeneous elements
- indexed by integers
- In technical terms, a PL/SQL table:
 - is like an array
 - is like a SQL table; yet it is not precisely the same as either of those data structures
 - is one type of collection structure
 - is PL/SQL's way of providing arrays

Table Datatype

- Like PL/SQL records, the table is another composite datatype. PL/SQL tables are objects of type TABLE, and look similar to database tables but with slight difference.
- PL/SQL tables use a primary key to give you array-like access to rows.
 - Like the size of the database table, the size of a PL/SQL table is unconstrained. That is, the number of rows in a PL/SQL table can dynamically increase. So your PL/SQL table grows as new rows are added.
 - PL/SQL table can have one column and a primary key, neither of which can be named.
 - The column can have any datatype, but the primary key must be of the type BINARY_INTEGER.
- Arrays are like temporary tables in memory. Thus they are processed very quickly.
- Like the size of the database table, the size of a PL/SQL table is unconstrained.
- The “column” can have any datatype. However, the “primary key” must be of the type BINARY_INTEGER.

1.4: Declaring a PL/SQL table

Table Data Type (Contd...)

- Declaring a PL/SQL table:

- There are two steps to declare a PL/SQL table:
 - Declare a TABLE type.
 - Declare PL/SQL tables of that type.

```
TYPE type_name is TABLE OF  
{Column_type | table.column%type} [NOT NULL]  
INDEX BY BINARY_INTEGER;
```

- If the column is defined as NOT NULL, then PL/SQL table will reject NULLs.



Copyright © Capgemini 2015. All Rights Reserved 28

Declaring a PL/SQL table

- PL/SQL tables must be declared in two steps. First you declare a TABLE type, then declare PL/SQL tables of that type. You can declare TABLE type in the declarative part of any block, subprogram or package.
- In the syntax on the above slide:
 - Type_name is type specifier used in subsequent declarations to define PL/SQL tables and column_name is any datatype.
 - You can use %TYPE attribute to specify a column datatype. If the column to which table.column refers is defined as NOT NULL, the PL/SQL table will reject NULLs.

1.4: Declaring a PL/SQL table

Table Data Type - Examples

- Example 1:

```
DECLARE
  ■ To create a PL/SQL table named as "student_table" of char column.
  ■ TYPE student_table is table of char(10)
    INDEX BY BINARY_INTEGER;
```

- Example 2:

- To create "student_table" based on the existing column of "student_name" of EMP table.

```
DECLARE
  ■ TYPE student_table is table of student_master.student_name%type
    INDEX BY BINARY_INTEGER;
```



Copyright © Capgemini 2015. All Rights Reserved 29

Declaring a PL/SQL table (contd.):

- Example 3:

- To declare a NOT NULL constraint

```
DECLARE
  ■ TYPE student_table is table of
    student_master.student_name%TYPE NOT
    NULL
    INDEX BY BINARY_INTEGER;
```

- **Note:** INDEX BY BINARY INTERGER is a mandatory feature of the PL/SQL table declaration.

1.4: Declaring a PL/SQL table

Table Data Type - Examples (Contd...)

- After defining type emp_table, define the PL/SQL tables of that type.
 - For example:

```
Student_tab      student_table;
```

- These tables are unconstrained tables.
- You cannot initialize a PL/SQL table in its declaration.
 - For example:

```
Student_tab :=('SMITH','JONES','BLAKE');      --Illegal
```



Copyright © Capgemini 2015. All Rights Reserved 30

Note:

- The PL/SQL tables are unconstrained tables, because its primary key can assume any value in the range of values defined by BINARY_INTEGER.
- You cannot initialize a PL/SQL table in its declaration.

1.4: Declaring a PL/SQL table

Referencing PL/SQL Tables

- Here is an example of referencing PL/SQL tables:

```
DECLARE
    TYPE staff_table IS TABLE OF
        staff_master.staff_name%TYPE
        INDEX BY BINARY_INTEGER;
    staff_tab staff_table;
BEGIN
    staff_tab(1) := 'Smith'; --update Smith's salary
    UPDATE staff_master
    SET staff_sal = 1.1 * staff_sal
    WHERE staff_name = staff_tab(1);
END;
```



Copyright © Capgemini 2015. All Rights Reserved 31

Referencing PL/SQL tables:

- To reference rows in a PL/SQL table, you specify the PRIMARY KEY value using the array-like syntax as shown below:

PL/SQL table_name (primary key value)

- When primary key value belongs to type BINARY_INTEGER you can reference the first row in PL/SQL table named emp_tab as shown in the slide.

1.4: Declaring a PL/SQL table

Referencing PL/SQL Tables - Examples

- To assign values to specific rows, the following syntax is used:

```
PLSQL_table_name(primary_key_value) := PLSQL expression;
```

- From ORACLE 7.3, the PL/SQL tables allow records as their columns.



Copyright © Capgemini 2015. All Rights Reserved 32

Referencing PL/SQL tables:

Examples:

```
type staff_rectype is record (
    staff_id integer,
    staff_sname varchar2(60)) ;

type staff_table is table of staff_rectype
index by binary_integer;

staff_tab      staff_table;
```

- Referencing fields of record elements in PL SQL tables:

```
staff_tab(375).staff_sname := 'SMITH';
```

Scope and Visibility of Variables

▪ Scope of Variables:

- The scope of a variable is the portion of a program in which the variable can be accessed.
- The scope of the variable is from the “variable declaration” in the block till the “end” of the block.
- When the variable goes out of scope, the PL/SQL engine will free the memory used to store the variable, as it can no longer be referenced.



Copyright © Capgemini 2015. All Rights Reserved 33

Scope and Visibility of Variables:

- References to an identifier are resolved according to its scope and visibility.
 - The scope of an identifier is that region of a program unit (block, subprogram, or package) from which you can reference the identifier.
 - An identifier is visible only in the regions from which you can reference the identifier using an unqualified name.
- Identifiers declared in a PL/SQL block are considered “local” to that “block” and “global” to all its “sub-blocks”.
 - If a global identifier is re-declared in a sub-block, both identifiers remain in scope. However, the local identifier is visible within the sub-block only because you must use a qualified name to reference the global identifier.
- Although you cannot declare an identifier twice in the same block, you can declare the same identifier in two different blocks.
 - The two items represented by the identifier are “distinct”, and any change in one does not affect the other. However, a block cannot reference identifiers declared in other blocks at the same level because those identifiers are neither local nor global to the block.

3.3: Handling Variables in PL/SQL

Scope and Visibility of Variables (Contd...)

▪ Visibility of Variables:

- The visibility of a variable is the portion of the program, where the variable can be accessed without having to qualify the reference. The visibility is always within the scope, it is not visible.

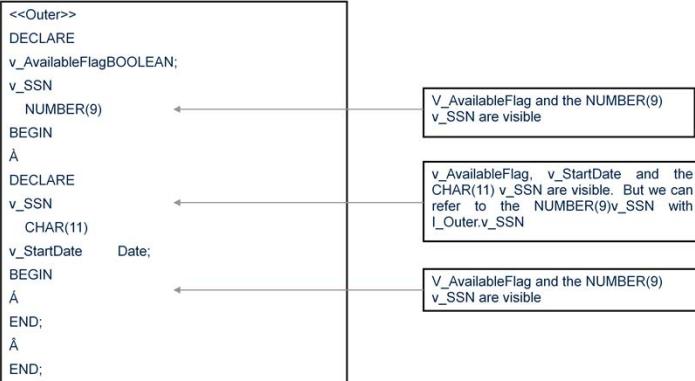


Copyright © Capgemini 2015. All Rights Reserved 34

3.3: Handling Variables in PL/SQL

Scope and Visibility of Variables (Contd...)

- Pictorial representation of visibility of a variable:



3.3: Handling Variables in PL/SQL

Scope and Visibility of Variables (Contd...)

```
<<OUTER>>
DECLARE
V_Flag BOOLEAN ;
V_Var1 CHAR(9);
BEGIN
<<INNER>>
DECLARE
V_Var1 NUMBER(9);
V_Date DATE;
BEGIN
NULL;
END;
NULL;
END;
```



Copyright © Capgemini 2015. All Rights Reserved 36

3.4: SQL in PL/SQL

Types of Statements

- Given below are some of the SQL statements that are used in PL/SQL:
 - INSERT statement
 - The syntax for the INSERT statement remains the same as in SQL-INSERT.
 - For example:

```
DECLARE
    v_dname varchar2(15) := 'Accounts';
BEGIN
    INSERT into department_master
    VALUES (50, v_dname);
END;
```



Copyright © Capgemini 2015. All Rights Reserved 37

3.4: SQL in PL/SQL

Types of Statements (Contd...)

- DELETE statement
 - For example:

```
DECLARE
    v_sal_cutoff number := 2000;
BEGIN
    DELETE FROM staff_master
    WHERE staff_sal < v_sal_cutoff;
END;
```



Copyright © Capgemini 2015. All Rights Reserved 38

3.4: SQL in PL/SQL

Types of Statements (Contd...)

- UPDATE statement
- For example:

```
DECLARE
    v_sal_incr number(5) := 1000;
BEGIN
    UPDATE staff_master
    SET staff_sal = staff_sal + v_sal_incr
    WHERE staff_name='Smith';
END;
```



Copyright © Capgemini 2015. All Rights Reserved 39

3.4: SQL in PL/SQL

Types of Statements (Contd...)

- SELECT statement
 - Syntax:

```
SELECT Column_List INTO Variable_List  
      FROM Table_List  
      [WHERE expr1]  
      CONNECT BY expr2 [START WITH expr3]]  
      GROUP BY expr4][HAVING expr5]  
      [UNION | INTERSECT | MINUS SELECT ...]  
      [ORDER BY expr | ASC | DESC]  
      [FOR UPDATE [OF Col1,...][NOWAIT]]  
      INTO Variable_List;
```



Types of Statements (Contd...)

- The column values returned by the SELECT command must be stored in variables.
- The Variable_List should match Column_List in both COUNT and DATATYPE.
- Here the variable lists are PL/SQL (Host) variables. They should be defined before use.



SELECT Statement:

Note:

- The SELECT clause is used if the selected row must be modified through a DELETE or UPDATE command.
- Since the contents of the row must not be modified between the SELECT command and the UPDATE command, it is necessary to lock the row after the SELECT command.
 - FOR UPDATE will lock the selected row.
 - OF Col1,.. lists the columns which can be modified by the UPDATE command.
 - If OF Col1,... is missing, all the columns can be modified.
- It is possible that when SELECT..UPDATE is issued, the concerned row is already locked by some other user or a previous SELECT..UPDATE command.
 - If NOWAIT is not given, then the current SELECT..UPDATE command will wait until the lock is cleared.
 - IF NOWAIT is given, then the SELECT..UPDATE will return immediately with a failure.

3.4: SQL in PL/SQL

Types of Statements (Contd...)

- Example: <>BLOCK1>>

```
DECLARE
    deptno  number(10) := 30;
    dname   varchar2(15);
BEGIN
    SELECT dept_name INTO dname FROM
    department_master WHERE dept_code = Block1. deptno;
    DELETE FROM department_master
        WHERE dept_code = Block1. deptno ;
END;
```



Copyright © Capgemini 2015. All Rights Reserved 42

SELECT statement (contd.):

SELECT statement (contd.):

More examples

```
DECLARE
dept_code          number(10) := 30;
v_dname            varchar2(15);
BEGIN
SELECT dept_name INTO v_dname FROM
department_master WHERE
dept_code=dept_code
DELETE FROM department_master
WHERE dept_code = dept_code ;
END;
```

- Here the SELECT statement will select names of all the departments and not only deptno. 30.
- The DELETE statement will delete all the employees.
 - This happens because when the PL/SQL engine sees a condition expr1 = expr2, the expr1 and expr2 are first checked to see whether they match the database columns first and then the PL/SQL variables.
 - So in the above example, where you see deptno = deptno, both are treated as database columns, and the condition will become TRUE for every row of the table.
- If a block has a label, then variables with same names as database columns can be used by using <<blockname>>. Variable_Name notation.
- It is not a good programming practice to use same names for PL/SQL variables and database columns.

3.5: Programmatic Constructs in PL/SQL

Types of Programmatic Constructs

- Programmatic Constructs are of the following types:
 - Selection structure
 - Iteration structure
 - Sequence structure

The diagram illustrates three types of programmatic constructs:

- Selection:** Represented by a diamond labeled "T" (True) and "F" (False). Two parallel horizontal boxes branch from the "T" and "F" paths, which then converge at a circular connector node before continuing downwards.
- Iteration:** Represented by a diamond labeled "T" (True) and "F" (False). A single vertical box branches from the "T" path down to another diamond labeled "T" and "F". This creates a loop where the process continues until the condition is False.
- Sequence:** Represented by a single vertical column of three rectangular boxes, indicating a linear sequence of operations.

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 44

Programming Constructs:

- The **selection structure** tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is TRUE or FALSE.
 - A condition is any variable or expression that returns a Boolean value (TRUE or FALSE).
- The **iteration structure** executes a sequence of statements repeatedly as long as a condition holds true.
- The **sequence structure** simply executes a sequence of statements in the order in which they occur.

IF Construct

- Given below is a list of Programmatic Constructs which are used in PL/SQL:

- Conditional Execution:

- This construct is used to execute a set of statements only if a particular condition is TRUE or FALSE.

- Syntax:

```
IF Condition_Expr  
THEN  
    PL/SQL_Statements  
END IF;
```



Copyright © Capgemini 2015. All Rights Reserved 45

Programmatic Constructs (contd.)

Conditional Execution:

- Conditional execution is of the following type:
 - IF-THEN-END IF
 - IF-THEN-ELSE-END IF
 - IF-THEN-ELSIF-END IF
- Conditional Execution construct is used to execute a set of statements only if a particular condition is TRUE or FALSE.

IF Construct - Example

- For example:

```
IF v_staffno = 100003  
THEN  
    UPDATE staff_master  
    SET staff_sal = staff_sal + 100  
    WHERE staff_code = 100003 ;  
END IF;
```



Programmatic Constructs (contd.)

Conditional Execution (contd.):

- As shown in the example in the slide, when the condition evaluates to TRUE, the PL/SQL statements are executed, otherwise the statement following END IF is executed.
- UPDATE statement is executed only if value of v_staffno variable equals 100003.
- PL/SQL allows many variations for the IF – END IF construct.

3.5: Programmatic Constructs in PL/SQL

IF Construct - Example (Contd...)

- To take alternate action if condition is FALSE, use the following syntax:

```
IF Condition_Expr THEN  
    PL/SQL_Statements_1 ;  
ELSE  
    PL/SQL_Statements_2 ;  
END IF;
```



Copyright © Capgemini 2015. All Rights Reserved 47

Programmatic Constructs (contd.)

Conditional Execution (contd.):

Note:

- When the condition evaluates to TRUE, the PL/SQL_Statements_1 is executed, otherwise PL/SQL_Statements_2 is executed.
- The above syntax checks **only one** condition, namely Condition_Expr.

IF Construct - Example (Contd...)

- To check for multiple conditions, use the following syntax.

```
IF Condition_Expr_1
THEN
    PL/SQL_Statements_1 ;
ELSIF Condition_Expr_2
THEN
    PL/SQL_Statements_2 ;
ELSIF Condition_Expr_3
THEN
    PL/SQL_Statements_3 ;
ELSE
    PL/SQL_Statements_n ;
END IF;
```

- Note: Conditions for NULL are checked through IS NULL and IS NOT NULL predicates.



Programmatic Constructs (contd.)

Conditional Execution (contd.):

```
DECLARE
    D VARCHAR2(3):= TO_CHAR(SYSDATE, 'DY')
BEGIN
    IF D= 'SAT' THEN
        DBMS_OUTPUT.PUT_LINE('ENJOY YOUR
WEEKEND');
    ELSIF D= 'SUN' THEN
        DBMS_OUTPUT.PUT_LINE('ENJOY YOUR
WEEKEND');
    ELSE
        DBMS_OUTPUT.PUT_LINE('HAVE A NICE
DAY');
    END IF;
END;
```

Programmatic Constructs (contd.)

Conditional Execution (contd.):

- As every condition must have at least one statement, NULL statement can be used as filler.
- NULL command does nothing.
- Sometimes NULL is used in a condition merely to indicate that such a condition has been taken into consideration, as well. So your code will resemble the code as given below:

```
IF Condition_Expr_1 THEN  
    PL/SQL_Statements_1 ;  
ELSIF Condition_Expr_2 THEN  
    PL/SQL_Statements_2 ;  
ELSIF Condition_Expr_3 THEN  
    Null;  
END IF;
```

- Conditions for NULL are checked through IS NULL and IS NOT NULL predicates.

3.5: Programmatic Constructs in PL/SQL

Simple Loop

- Looping

- A LOOP is used to execute a set of statements more than once.
- Syntax:

```
LOOP  
    PL/SQL_Statements;  
END LOOP ;
```



Copyright © Capgemini 2015. All Rights Reserved 50

Simple Loop (Contd...)

- For example:

```
DECLARE
    v_counter number := 50 ;
BEGIN
LOOP
    INSERT INTO department_master
        VALUES(v_counter,'new dept');
    v_counter := v_counter + 10 ;
END LOOP;
COMMIT ;
END ;
/
```



Programmatic Constructs (contd.)

Looping

- The example shown in the slide is an endless loop.
- When LOOP ENDLOOP is used in the above format, then an exit path must necessarily be provided. This is discussed in the following slide.

3.5: Programmatic Constructs in PL/SQL

Simple Loop – EXIT statement

- EXIT

- Exit path is provided by using EXIT or EXIT WHEN commands.
- EXIT is an unconditional exit. Control is transferred to the statement following END LOOP, when the execution flow reaches the EXIT statement.



Copyright © Capgemini 2015. All Rights Reserved 52

3.5: Programmatic Constructs in PL/SQL

Simple Loop – EXIT statement (Contd...)

- Syntax:

```
BEGIN  
    ....  
    LOOP  
        ....  
        EXIT;          -- Exits loop immediately  
        END IF;  
    END LOOP;  
    LOOP  
        ....  
        EXIT WHEN <condition>;  
    END LOOP;  
    ....  
    COMMIT;  
    END;
```



Copyright © Capgemini 2015. All Rights Reserved 53

Note:

EXIT WHEN is used for conditional exit out of the loop.

Simple Loop – EXIT statement (Contd...)

- For example:

```
DECLARE
    v_counter number := 50 ;
BEGIN
    LOOP
        INSERT INTO department_master
        VALUES(v_counter,'NEWDEPT');
        DELETE FROM emp WHERE deptno = v_counter;
        v_counter := v_counter + 10 ;
        EXIT WHEN v_counter >100 ;

    END LOOP;
    COMMIT ;
END ;
```

- Note: As long as v_counter has a value less than or equal to 100, the loop continues.



Note:

LOOP.. END LOOP can be used in conjunction with FOR and WHILE for better control on looping.

3.5: Programmatic Constructs in PL/SQL

For Loop

- FOR Loop:
 - Syntax:

```
FOR Variable IN [REVERSE] Lower_Bound..Upper_Bound
LOOP
    PL/SQL_Statements
END LOOP ;
```

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 55

Programmatic Constructs (contd.)

FOR Loop:

- FOR loop is used for executing the loop a fixed number of times. The number of times the loop will execute equals the following:
 - Upper_Bound - Lower_Bound + 1.
- Upper_Bound and Lower_Bound must be integers.
- Upper_Bound must be equal to or greater than Lower_Bound.
- Variables in FOR loop need not be explicitly declared.
 - Variables take values starting at a Lower_Bound and ending at a Upper_Bound.
 - The variable value is incremented by 1, every time the loop reaches the bottom.
 - When the variable value becomes equal to the Upper_Bound, then the loop executes and exits.
- When REVERSE is used, then the variable takes values starting at Upper_Bound and ending at Lower_Bound.
- Value of the variable is decremented each time the loop reaches the bottom.

For Loop - Example

- For Example:

```
DECLARE
v_counter number := 50 ;
BEGIN
    FOR Loop_Counter IN 2..5
    LOOP
        INSERT INTO dept
        VALUES(v_counter , 'NEW DEPT') ;
        v_counter := v_counter + 10 ;
    END LOOP;
    COMMIT ;
END ;
```



Programmatic Constructs (contd.)

- In the example in the above slide, the loop will be executed $(5 - 2 + 1) = 4$ times.
- A Loop_Counter variable can also be used inside the loop, if required.
- Lower_Bound and/or Upper_Bound can be integer expressions, as well.

While Loop

- WHILE Loop

- The WHILE loop is used as shown below.
 - Syntax:

```
WHILE Condition  
LOOP  
    PL/SQL Statements;  
END LOOP;
```

- EXIT OR EXIT WHEN can be used inside the WHILE loop to prematurely exit the loop.



Programmatic Constructs (contd.)

WHILE Loop:

Example:

```
DECLARE  
    ctr number := 1;  
BEGIN  
    WHILE ctr <= 10  
    LOOP  
        dbms_output.put_line(ctr);  
        ctr := ctr+1;  
    END LOOP;  
END;  
/
```

Labeling of Loops

- Labeling of Loops:

- The label can be used with the EXIT statement to exit out of a particular loop.

```
BEGIN
    <<Outer_Loop>>
    LOOP
        PL/SQL
        << Inner_Loop>>
        LOOP
            PL/SQL Statements ;
            EXIT Outer_Loop WHEN <Condition Met>
        END LOOP Inner_Loop
    END LOOP Outer_Loop
END ;
```



Copyright © Capgemini 2015. All Rights Reserved 58

Programmatic Constructs (contd.)

Labeling of Loops:

- Loops themselves can be labeled as in the case of blocks.
- The label can be used with the EXIT statement to exit out of a particular loop.

Summary

- In this lesson, you have learnt:
 - PL/SQL is a procedural extension to SQL.
 - PL/SQL exhibits a block structure, different block types being: Anonymous, Procedure, and Function.
 - While declaring variables in PL/SQL:
 - declare and initialize variables within the declaration section
 - assign new values to variables within the executable section
 - pass values into PL/SQL blocks through parameters
 - view results through output variables



Summary

- Different types of PL/SQL Variables are: Scalar, Composite, Reference, LOB
- Scope of a variable: It is the portion of a program in which the variable can be accessed.
- Visibility of a variable: It is the portion of the program, where the variable can be accessed without having to qualify the reference.
- Different programmatic constructs in PL/SQL are Selection structure, Iteration structure, Sequence structure



Review Question

- Question 1: User-defined SUBTYPES are subtypes based on an existing type.
 - True / False

- Question 2: A record is a collection of individual fields that represents a row in the table.
 - True/ False



Review Question

- Question 3: %ROWTYPE is used to declare a variable with the same datatype as a column of a specific table.
 - True / False

- Question 4: PL/SQL tables use a primary key to give you array-like access to rows.
 - True / False



Review Question

- Question 5: While using FOR loop, Upper_Bound, and Lower_Bound must be integers.
 - True / False



Oracle (PL/SQL)

Lesson 04 : Cursors

Lesson Objectives

- To understand the following topics:
 - Introduction to Cursors
 - Implicit and Explicit Cursors
 - Cursor attributes
 - Processing Implicit Cursors and Explicit Cursors
 - Cursor with Parameters
 - Difference between Cursors and Cursor Variables
 - Use of Cursor Variables



4.1: Cursors

Concept

- A cursor is a “handle” or “name” for a private SQL area.
- An SQL area (context area) is an area in the memory in which a parsed statement and other information for processing the statement are kept.
- PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including queries that return “only one row”.
- For queries that return “more than one row”, you must declare an explicit cursor.
- Thus the two types of cursors are:
 - implicit
 - explicit

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 3

Introduction to Cursors:

- ORACLE allocates memory on the Oracle server to process SQL statements. It is called as “**context area**”. Context area stores information like number of rows processed, the set of rows returned by a query, etc.
- A Cursor is a “handle” or “pointer” to the context area. Using this cursor the PL/SQL program can control the context area, and thereby access the information stored in it.
- There are two types of cursors - “explicit” and “implicit”.
 - In an explicit cursor, a cursor name is explicitly assigned to a SELECT statement through CURSOR IS statement.
 - An implicit cursor is used for all other SQL statements.
- Processing an explicit cursor involves four steps. In case of implicit cursors, the PL/SQL engine automatically takes care of these four steps.

4.1: Cursors

Concept

- **Implicit Cursor:**

- The PL/SQL engine takes care of automatic processing.
- PL/SQL implicitly declares cursors for all DML statements.
- They are simple SELECT statements and are written in the BEGIN block (executable section) of the PL/SQL.
- They are easy to code, and they retrieve exactly one row

4.1: Cursors

Implicit Cursors

- Processing Implicit Cursors:
 - Oracle implicitly opens a cursor to process each SQL statement that is not associated with an explicitly declared cursor.
 - This implicit cursor is known as SQL cursor.
 - Program cannot use the OPEN, FETCH, and CLOSE statements to control the SQL cursor. PL/SQL implicitly does those operations .
 - You can use cursor attributes to get information about the most recently executed SQL statement.
 - Implicit Cursor is used to process INSERT, UPDATE, DELETE, and single row SELECT INTO statements.

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 5

Processing Implicit Cursors:

- All SQL statements are executed inside a context area and have a cursor, which points to that context area. This implicit cursor is known as SQL cursor.
- Implicit SQL cursor is not opened or closed by the program. PL/SQL implicitly opens the cursor, processes the SQL statement, and closes the cursor.
- Implicit cursor is used to process INSERT, UPDATE, DELETE, and single row SELECT INTO statements.
- The cursor attributes can be applied to the SQL cursor.

4.1: Cursors

Implicit Cursors - Example

```
BEGIN  
    UPDATE dept SET dname ='Production' WHERE deptno= 50;  
    IF SQL%NOTFOUND THEN  
        INSERT into department_master VALUES ( 50, 'Production');  
    END IF;  
END;
```

```
BEGIN  
    UPDATE dept SET dname ='Production' WHERE deptno = 50;  
    IF SQL%ROWCOUNT = 0 THEN  
        INSERT into department_master VALUES ( 50, 'Production');  
    END IF;  
END;
```



Copyright © Capgemini 2015. All Rights Reserved 6

Processing Implicit Cursors:

Note:

- SQL%NOTFOUND should not be used with SELECT INTO Statement.
- This is because SELECT INTO.... Statement will raise an ORACLE error if no rows are selected, and
 - control will pass to exception * section (discussed later), and
 - SQL%NOTFOUND statement will not be executed at all
- The slide shows two code snippets using Cursor attributes SQL%NOTFOUND and SQL%ROWCOUNT respectively.

Explicit Cursors

▪ Explicit Cursor:

- The set of rows returned by a query can consist of zero, one, or multiple rows, depending on how many rows meet your search criteria.
- When a query returns multiple rows, you can explicitly declare a cursor to process the rows.
- You can declare a cursor in the declarative part of any PL/SQL block, subprogram, or package.
- Processing has to be done by the user.

Explicit Cursor:

- When you need precise control over query processing, you can explicitly declare a cursor in the declarative part of any PL/SQL block, subprogram, or package.
- This technique requires more code than other techniques such as the implicit cursor FOR loop. But it is beneficial in terms of flexibility. You can:
 - Process several queries in parallel by declaring and opening multiple cursors.
 - Process multiple rows in a single loop iteration, skip rows, or split the processing into more than one loop.

4.1: Cursors

Processing Explicit Cursors

- While processing Explicit Cursors you have to perform the following four steps:

- Declare the cursor
- Open the cursor for a query
- Fetch the results into PL/SQL variables
- Close the cursor



Copyright © Capgemini 2015. All Rights Reserved 8

4.1: Cursors

Processing Explicit Cursors

- Declaring a Cursor:

- Syntax:

```
CURSOR Cursor_Name IS Select_Statement;
```

- Any SELECT statements are legal including JOINS, UNION, and MINUS clauses.
 - SELECT statement should not have an INTO clause.
 - Cursor declaration can reference PL/SQL variables in the WHERE clause.
 - The variables (bind variables) used in the WHERE clause must be visible at the point of the cursor.

4.1: Cursors

Processing Explicit Cursors

▪ Usage of Variables Legal Use of Variable Illegal Use of Variable

```
DECLARE  
v_deptno number(3); CURSOR  
c_dept IS  
SELECT * FROM  
department_master  
WHERE deptno=v_deptno;  
BEGIN  
NULL;  
END;
```

```
DECLARE  
CURSOR c_dept IS  
SELECT * FROM  
department_master  
WHERE deptno=v_deptno;  
v_deptno number(3);  
BEGIN  
NULL;  
END;
```

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 10

Processing Explicit Cursors: Declaring a Cursor:

- The code snippets on the slide show the usage of variables in cursor declaration. You cannot use a variable before it has been declared. It will be illegal.

4.1: Cursors

Processing Explicit Cursors

- Opening a Cursor
 - Syntax:

```
OPEN Cursor_Name;
```
 - When a cursor is opened, the following events occur:
 1. The values of bind variables are examined.
 2. The active result set is determined.
 3. The active result set pointer is set to the first row.

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 11

Processing Explicit Cursors: Opening a Cursor:

- When a Cursor is opened, the following events occur:
 1. The values of “bind variables” are examined.
 2. Based on the values of bind variables , the “active result set” is determined.
 3. The active result set pointer is set to the first row.
- “Bind variables” are evaluated only once at Cursor open time.
 - Changing the value of Bind variables after the Cursor is opened will not make any changes to the active result set.
 - The query will see changes made to the database that have been committed prior to the OPEN statement.
- You can open more than one Cursor at a time.

4.1: Cursors

Processing Explicit Cursors

- Fetching from a Cursor

- Syntax:

```
FETCH Cursor_Name INTO List_Of_Variables;  
FETCH Cursor_Name INTO PL/SQL_Record;
```

- The “list of variables” in the INTO clause should match the “column names list” in the SELECT clause of the CURSOR declaration, both in terms of count as well as in datatype.
 - After each FETCH, the active set pointer is increased to point to the next row.
 - The end of the active set can be found out by using %NOTFOUND attribute of the cursor.



Copyright © Capgemini 2015. All Rights Reserved 12

Processing Explicit Cursors: Fetching from Cursor:

Processing Explicit Cursor

▪ Fetching Data

```
DECLARE  
    v_deptno  department_master.dept_code%type;  
    v_dept     department_master%rowtype;  
    CURSOR c_alldept IS SELECT * FROM  
        department_master;  
BEGIN  
    OPEN  c_alldept;  
    FETCH c_Alldept INTO V_Dept;  
  
    FETCH c_alldept INTO V_DDeptno;  
END;
```

Legal Fetch

Illegal Fetch



Processing Explicit Cursors: Fetching from Cursor:

- The code snippets on the slide shows example of fetching data from cursor. The second snippet FETCH is illegal since SELECT * selects all columns of the table, and there is only one variable in INTO list.
- For each column value returned by the query associated with the cursor, there must be a corresponding, type-compatible variable in the INTO list.
- To change the result set or the values of variables in the query, you must close and reopen the cursor with the input variables set to their new values.

4.1: Cursors

Processing Explicit Cursors

- Closing a Cursor

- Syntax

```
CLOSE Cursor_Name;
```

- Closing a Cursor frees the resources associated with the Cursor.
 - You cannot FETCH from a closed Cursor.
 - You cannot close an already closed Cursor.



Copyright © Capgemini 2015. All Rights Reserved 14

4.1: Cursors Attributes

■ Cursor Attributes:

- Explicit cursor attributes return information about the execution of a multi-row query.
- When an “Explicit cursor” or a “cursor variable” is opened, the rows that satisfy the associated query are identified and form the result set.
- Rows are fetched from the result set.
- Examples: %ISOPEN, %FOUND, %NOTFOUND, %ROWCOUNT, etc.

4.1: Cursors

Types of Cursor Attributes

- The different types of cursor attributes are described in brief, as follows:

- %ISOPEN

- %ISOPEN returns TRUE if its cursor or cursor variable is open. Otherwise it returns FALSE.

- Syntax:

```
Cur_Name%ISOPEN
```



Copyright © Capgemini 2015. All Rights Reserved 16

Cursor Attributes: %ISOPEN

- This attribute is used to check the open/close status of a Cursor.
- If the Cursor is already open, the attribute returns TRUE.
- Oracle closes the SQL cursor automatically after executing its associated SQL statement. As a result, %ISOPEN always yields FALSE for Implicit cursor.

4.1: Cursors

Types of Cursor Attributes

- Example:

```
DECLARE
    cursor c1 is
        select_statement ;
BEGIN
    IF c1%ISOPEN THEN
        pl/sql_statements ;
    END IF;
END ;
```



Copyright © Capgemini 2015. All Rights Reserved 17

Cursor Attributes: %ISOPEN (contd.)

Note:

- In the example shown in the slide, C1%ISOPEN returns FALSE as the cursor is yet to be opened.
- Hence, the PL/SQL statements within the IF...END IF are not executed.

4.1: Cursors

Types of Cursor Attributes

▪ %FOUND

- %FOUND yields NULL after a cursor or cursor variable is opened but before the first fetch.
- Thereafter, it yields:
 - TRUE if the last fetch has returned a row, or
 - FALSE if the last fetch has failed to return a row
- Syntax:

```
cur_Name%FOUND
```



Copyright © Capgemini 2015. All Rights Reserved 18

Cursor Attributes: %FOUND:

- Until a SQL data manipulation statement is executed, %FOUND yields NULL. Thereafter, %FOUND yields TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows, or a SELECT INTO statement returned one or more rows. Otherwise, %FOUND yields FALSE

4.1: Cursors

Types of Cursor Attributes

- Example:

```
DECLARE section;
  open c1 ;
  fetch c1 into var_list ;
  IF c1%FOUND THEN
    pl/sql_statements ;
  END IF ;
```



Copyright © Capgemini 2015. All Rights Reserved 19

4.1: Cursors

Types of Cursor Attributes

▪ %NOTFOUND

- %NOTFOUND is the logical opposite of %FOUND.
- %NOTFOUND yields:
 - FALSE if the last fetch has returned a row, or
 - TRUE if the last fetch has failed to return a row
- It is mostly used as an exit condition.
- Syntax:

```
cur_Name%NOTFOUND
```



Copyright © Capgemini 2015. All Rights Reserved 20

Cursor Attributes: %NOTFOUND:

- %NOTFOUND is the logical opposite of %FOUND. %NOTFOUND yields TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, %NOTFOUND yields FALSE.

4.1: Cursors

Types of Cursor Attributes

- %ROWCOUNT
 - %ROWCOUNT returns number of rows fetched from the cursor area using FETCH command.
 - %ROWCOUNT is zeroed when its cursor or cursor variable is opened.
 - Before the first fetch, %ROWCOUNT yields 0.
 - Thereafter, it yields the number of rows fetched at that point of time.
 - The number is incremented if the last FETCH has returned a row.
 - Syntax:

cur_Name%NOTFOUND



Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 21

Cursor Attributes: %ROWCOUNT

For example: To give a 10% raise to all employees earning less than Rs. 2500.

```

DECLARE
    V_Salary emp.sal%TYPE;
    V_Empno emp.empno%TYPE;
    CURSOR C_Empsal IS
        SELECT empno, sal FROM emp
        WHERE sal <2500;
BEGIN
    IF NOT C_Empsal%ISOPEN THEN
        OPEN C_Empsal ;
    END IF ;
    LOOP
        FETCH C_Empsal INTO
            V_Empno,V_Salary;
        EXIT WHEN C_Empsal %NOTFOUND ;
        --Exit out of block when no
        rows
        UPDATE emp SET sal = 1.1 * V_Salary
        WHERE empno = V_Empno;
    END LOOP ;
    CLOSE C_Empsal ;
    COMMIT ;
END ;

```

4.1: Cursors

Cursor FETCH loops

- They are examples of simple loop statements.
- The FETCH statement should be followed by the EXIT condition to avoid infinite looping.
- Condition to be checked is cursor%NOTFOUND.
- Examples: LOOP .. END LOOP, WHILE LOOP, etc



Copyright © Capgemini 2015. All Rights Reserved 22

4.1: Cursors

Cursor using LOOP ... END LOOP:

```
DECLARE
    cursor c1 is .....
BEGIN
    open cursor c1; /* open the cursor and identify the active result set.*/
LOOP
    fetch c1 into variable_list ;
    -- exit out of the loop when there are no more rows.
    /* exit is done before processing to prevent handling of null rows.*/
    EXIT WHEN C1%NOTFOUND ;
    /* Process the fetched rows using variables and PL/SQLstatements */
END LOOP;
    -- Free resources used by the cursor
    close c1;
    -- commit
    commit;
END;
```



Copyright © Capgemini 2015. All Rights Reserved 23

4.1: Cursors

Cursor using LOOP ... END LOOP:

- There should be a FETCH statement before the WHILE statement to enter into the loop.
- The EXIT condition should be checked as cursorname%FOUND.
- Syntax:

```
DECLARE
    cursor c1 is .....
BEGIN
    open cursor c1 /* open the cursor and identify the active
result set.*/
    -- retrieve the first row to set up the while loop
    FETCH C1 INTO VARIABLE_LIST;
```



Copyright © Capgemini 2015. All Rights Reserved 24

Processing Explicit Cursors: Cursor using WHILE loops:

Note:

- FETCH statement appears twice, once before the loop and once after the loop processing.
- This is necessary so that the loop condition will be evaluated for each iteration of the loop.

Cursor using WHILE loops...contd

/*Continue looping , processing & fetching till last row is retrieved.*/

```
WHILE C1%FOUND
```

```
LOOP
```

```
    fetch c1 into variable_list;
```

```
END LOOP;
```

```
CLOSE C1; -- Free resources used by the cursor.
```

```
commit; -- commit
```

```
END;
```



4.1: Cursors

FOR Cursor LOOP

- FOR Cursor Loop

```
FOR Variable in Cursor_Name
LOOP
    Process the variables
END LOOP;
```

- You can pass parameters to the cursor in a CURSOR FOR loop.

```
FOR Variable in Cursor_Name ( PARAM1 , PARAM 2 ....)
LOOP
    Process the variables
END LOOP;
```



Copyright © Capgemini 2015. All Rights Reserved 26

Processing Explicit Cursors: FOR CURSOR Loop:

- For all other loops we had to go through all the steps of OPEN, FETCH, AND CLOSE statements for a cursor.
- PL/SQL provides a shortcut via a CURSOR FOR Loop. The CURSOR FOR Loop implicitly handles the cursor processing.

```
DECLARE
CURSOR C1 IS .....
BEGIN
-- An implicit Open of the cursor C1 is done here.
-- Record variable should not be declared in
declaration section
FOR Record_Variable IN C1 LOOP
-- An implicit Fetch is done here
-- Process the fetched rows using variables and
PL/SQL statements
-- Before the loop is continued an implicit
C1%NOTFOUND test is done by PL/SQL
END LOOP;
-- An automatic close C1 is issued after termination of
the loop
-- Commit
COMMIT ;
END;
/
```

- In this snippet, the record variable is implicitly declared by PL/SQL and is of the type C1%ROWTYPE and the scope of Record_Variable is only for the cursor FOR LOOP.

Explicit Cursor - Examples

- Example 1: To add 10 marks in subject3 for all students

```
DECLARE
    v_sno      student_marks.student_code%type;
    cursor c_stud_marks is
        select student_code from student_marks;
BEGIN
    OPEN c_stud_marks;
    FETCH c_stud_marks into v_sno;
    WHILE c_stud_marks%found
    LOOP
        UPDATE student_marks SET subject3 =subject3+10
        WHERE student_code = v_sno ;
        FETCH c_stud_marks into v_emphno;
    END LOOP ;
    CLOSE c_stud_marks;
END;
```



Copyright © Capgemini 2015. All Rights Reserved 27

Explicit Cursor - Examples

- Example 2: The block calculates the total marks of each student for all the subjects. If total marks are greater than 220 then it will insert that student detail in "Performance" table

```
DECLARE
    cursor c_stud_marks is select * from student_marks;
    total_marks number(4);
BEGIN
    FOR mks in c_stud_marks
    LOOP
        total_marks:=mks.subject1+mks.subject2+mks.subject3;
        IF (total_marks >220) THEN
            INSERT into performance
            VALUES (mks.student_code,total_marks);
        END IF;
    END LOOP;
END;
```



Copyright © Capgemini 2015. All Rights Reserved 29

In the above example "Performance" is a user defined table.

4.1: Cursors

SELECT... FOR UPDATE

- **SELECT ... FOR UPDATE cursor:**
- The method of locking records which are selected for modification, consists of two parts:
 - The FOR UPDATE clause in CURSOR declaration.
 - The WHERE CURRENT OF clause in an UPDATE or DELETE statement.
- Syntax: FOR UPDATE

```
CURSOR Cursor_Name IS SELECT ..... FROM ... WHERE .. ORDER BY  
FOR UPDATE [OF column names] [ NOWAIT]
```

where column names are the names of the columns in the table against which the query is fired. The column names are optional.

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 29

Processing Explicit Cursors: Using FOR UPDATE

- The FOR UPDATE clause is part of the SELECT statement of the cursor. It is the last clause of the SELECT statement after the ORDER BY clause (if present).
- Normally, a SELECT operation does not lock the rows being accessed. This allows others to change the data selected by the SELECT statement. Besides, at OPEN time the active set consists of changes which were committed. Any changes made after OPEN even if they are committed, are not reflected in the active result set unless the cursor is reopened. This results in Data Inconsistency.
 - If the FOR UPDATE clause is present, exclusive “row locks” are taken on the rows in the active set.
 - These locks prevent other sessions / users from changing these rows unless the changes are committed.
- If another session / user already has locks on the rows in the active set, then the SELECT FOR UPDATE will wait indefinitely for these locks to be released. This statement will hang the system till the locks are released.
 - To avoid this NOWAIT clause is used.
 - With the NOWAIT clause SELECT FOR UPDATE will not wait for the locks acquired by previous sessions to be released and will return immediately with an ORACLE error.

4.1: Cursors

SELECT... FOR UPDATE

- If the cursor is declared with a FOR UPDATE clause, the WHERE CURRENT OF clause can be used in an UPDATE or DELETE statement.

— Syntax: WHERE CURRENT OF

WHERE CURRENT OF Cursor_Name

- The WHERE CURRENT OF clause evaluates up to the row that was just retrieved by the cursor.
- When querying multiple tables Rows in a table are locked only if the FOR UPDATE OF clause refers to a column in that table.

contd.



Copyright © Capgemini 2015. All Rights Reserved 30

Processing Cursors: Using WHERE CURRENT OF:

Note:

- When querying multiple tables you can use the FOR UPDATE OF column to confine row locking for a particular table.

4.1: Cursors

SELECT... FOR UPDATE

- For example: Following query locks the staff_master table but not the department_master table.

CURSOR C1 is SELECT staff_code, job, dname from emp,
dept WHERE emp.deptno=dept.deptno FOR UPDATE OF sal;

- Using primary key simulates the WHERE CURRENT OF clause but does not create any locks.



Copyright © Capgemini 2015. All Rights Reserved 31

Processing Cursors: Using WHERE CURRENT OF (contd.):

Note:

- If you are using NOWAIT clause, then OF Column_List is essential for syntax purpose.
- Any COMMIT should be done after the processing is over in a CURSOR LOOP which has FOR UPDATE clause. This is because after commit locks will be released, the cursor will become invalid, and further FETCH will result in an error.
- This problem can be solved by using primary key. Using primary key simulates the WHERE CURRENT of clause, however it does not create any locks.

4.1: Cursors

SELECT... FOR UPDATE - Examples

- To promote professors who earn more than 20000

```
DECLARE
CURSOR c_staff IS SELECT staff_code, staff_master.design_code
FROM staff_master,designation_master
WHERE design_name = 'Professor' AND staff_sal > 20000
AND staff_master.design_code = designation_master.design_code
FOR UPDATE OF design_code NOWAIT;
d_code designation_master.design_code%type;
BEGIN
  SELECT design_code INTO d_code FROM designation_master
  WHERE design_name='Director';
  FOR v_rec IN c_staff
  LOOP
    UPDATE staff_master SET design_code = d_code
    WHERE current of c_staff;
  END LOOP;
END;
```



Copyright © Capgemini 2015. All Rights Reserved 32

4.2: Cursors with Parameters

Parameterized Cursor

- You must use the OPEN statement to pass parameters to a cursor.
- Unless you want to accept default values, each “formal parameter” in the Cursor declaration must have a corresponding “actual parameter” in the OPEN statement.
- The scope of parameters is local to the cursor.
- Syntax:

```
OPEN Cursor-name(param1, param2.....)
```



Copyright © Capgemini 2015. All Rights Reserved 33

Cursor with Parameters:

- A cursor can take parameters, which can appear in the associated query wherever constants can appear. The formal parameters of a cursor must be IN parameters; they supply values in the query, but do not return any values from the query. You cannot impose the constraint NOT NULL on a cursor parameter.
- Cursor parameters can be referenced only within the query specified in the cursor declaration. The parameter values are used by the associated query when the cursor is opened.
- Example:

```
CURSOR C_Select_staff (Low_Sal NUMBER  
DEFAULT 0 , High_Sal DEFAULT 10000) IS  
SELECT * from staff_master WHERE staff_sal  
BETWEEN Low_Sal AND High_Sal);
```

- The scope of parameters is local to the cursor.
- The values of cursor parameters are used by associate queries when the cursor is OPEN.

4.2: Cursors with Parameters

Parameterized Cursor - Examples

- Parameters are passed to a parametric cursor using the syntax OPEN (param1, param2 ...) as shown in the following example:

```
OPEN C_Select_staff( 800,5000);
Query → SELECT * from staff_master
WHERE staff_sal BETWEEN 800 AND 5000;
```



Copyright © Capgemini 2015. All Rights Reserved 34

Cursor with Parameters: Passing Parameters to Cursors

- In a FOR CURSOR LOOP parameters are passed using the following syntax:

```
FOR Variable in Cursor_Name (PARAM1 ,
PARAM 2 ....);
FOR V_Get_Det in C_Select_staff( 800,5000)
LOOP
Process the variables
END LOOP;
```

- Unless you want to accept “default values”, each “formal parameter” in the cursor declaration must have a corresponding “actual parameter” in the OPEN statement.
 - Formal parameters having default values need not have corresponding actual values.
 - Each parameter must belong to a datatype which is compatible with the data type of its corresponding formal parameter.

4.3: Cursors Variables

Usage

- Like a Cursor, a Cursor Variable points to the current row in the result set of a multi-row query.
- A Cursor is static whereas a Cursor Variable is dynamic because it is not tied to a specific query.
- You can open a Cursor Variable for any type-compatible query.
 - This offers more flexibility
- You can assign new values to a Cursor Variable and pass it as a parameter to subprograms, including those in database.
 - This offers an easy way to centralize data retrieval.



Copyright © Capgemini 2015. All Rights Reserved 35

4.3: Cursors Variables

Usage

- Cursor variables are available to every PL/SQL client.
- You can declare a cursor variable in a PL/SQL host environment, and then pass it as a bind variable to PL/SQL.
- Oracle Forms and Oracle Reports, which have a PL/SQL engine, can use cursor variables entirely on the client side.



Copyright © Capgemini 2015. All Rights Reserved 36

Usage of Cursor Variables:

Note:

- Cursor variables are like 'C' pointers, which hold the memory location (address) of some item instead of the item itself.
- So when you are declaring a Cursor Variable you are creating a "pointer", and not an "item".

4.3: Cursors Variables

Cursors and Cursor Variables - Comparison

- To access the processing information stored in an unnamed work area, you can use:
 - an Explicit Cursor, which names the work area or
 - a Cursor Variable, which points to the work area
- However, Cursors and Cursor Variables are not interoperable.
 - a Cursor always refers to the "same query work area".
 - a Cursor Variable can refer to "different work areas".

 Capgemini CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 37

Cursors and Cursor Variables:

- In PL/SQL, a pointer has datatype **REF X**, where **REF** is a short name for **REFERENCE** and X stands for a class of an object. Therefore, a Cursor Variable has datatype REF CURSOR.
- To execute a multi-row query, Oracle opens an unnamed work area that stores processing information.
 - To access the information,
 - you can use an explicit cursor, which names the work area. Or,
 - you can use a cursor variable, which points to the work area.
 - A Cursor always refers to the "same query work area", whereas a Cursor Variable can refer to "different work areas". So, cursors and cursor variables are not interoperable that is, you cannot use one where the other is expected.
- Mainly, Cursor Variables are used "to pass query result sets" between PL/SQL stored subprograms and various clients.
 - Neither PL/SQL nor any of its clients owns a result set.
 - They simply "share a pointer" to the query work area in which the result set is stored.

For example: Oracle Forms application, and Oracle Server can all refer to the same work area.

Cursors and Cursor Variables (contd.):

- A query work area remains accessible as long as any Cursor Variable points to it. Therefore, you can pass the value of a Cursor Variable freely from one scope to another.

For example: If you pass a “host cursor variable” to a PL/SQL block, the work area to which the Cursor Variable points remains accessible after the block completes.

- If you have a PL/SQL engine on the client side, then calls from client to server impose no restrictions.

For example: You can declare a Cursor Variable on the client side, open and fetch from it on the server side, and then continue to fetch from it back on the client side. Also, you can reduce network traffic by having a *PL/SQL* block open (or close) several “host cursor variables” in a single round trip.

Creating Cursor Variables:

- To create cursor variables, two steps are required in the same sequence:
 1. Define a *REF CURSOR* type
 2. Declare cursor variables of that type
- You can define *REF CURSOR* types in any PL/SQL block, subprogram, or package.

4.3: Cursors Variables

Cursor Variables - Example

- Defining REF CURSOR types:

- Syntax:

```
TYPE ref_type_name IS REF CURSOR RETURN return_type;  
DECLARE  
    TYPE DeptCurTyp IS REF CURSOR RETURN  
        department_master%ROWTYPE;
```

- where:

- ref_type_name is a type specifier used in subsequent declarations of cursor variables
 - Return_type must represent a record or a row in a database table.
 - REF CURSOR types are strong (restrictive), or weak (non-restrictive)



Copyright © Capgemini 2015. All Rights Reserved 39

Cursors and Cursor Variables: Defining REF CURSOR types:

- **Example:** In the example shown on the slide , you specify a Return Type that represents a row in the database table dept:

4.3: Cursors Variables

Cursor Variables - Example

```
DECLARE
    TYPE staffCurTyp IS REF CURSOR
        RETURN staff_master%ROWTYPE; -- Strong
        types

        TYPE GenericCurTyp IS REF CURSOR; --
        Weak types
```



Copyright © Capgemini 2015. All Rights Reserved 40

Cursors and Cursor Variables: Defining REF CURSOR types (contd.):

Note:

- A strong REF CURSOR type definition specifies a Return Type.
- However, a weak definition does not specify a Return Type.
- Strong REF CURSOR types are less error prone because the PL/SQL compiler lets you associate a strongly typed Cursor Variable only with type-compatible queries.
- However, weak REF CURSOR types are more flexible because the compiler lets you associate a weakly typed cursor variable with any query.

4.3: Cursors Variables

Cursor Variables - Example

- Declaring Cursor Variables:

- Example 1:

```
DECLARE
  TYPE DeptCurTyp IS REF CURSOR RETURN
    department_master%ROWTYPE;
  dept_cv          DeptCurTyp; -- Declare cursor variable
```

- You cannot declare cursor variables in a package.
 -

- Example 2:

```
TYPE TmpCurTyp IS REF CURSOR RETURN staff_master%ROWTYPE;
tmp_cv TmpCurTyp; -- Declare cursor variable
```



Copyright © Capgemini 2015. All Rights Reserved 41

Cursors and Cursor Variables: Declaring Cursor Variable:

- As shown in examples shown on the slide, in the RETURN clause of a REF CURSOR type definition, you can use %ROWTYPE to specify a record type that represents a row returned by a strongly (not weakly) typed cursor variable.

4.3: Cursors Variables

Cursor Variables - Example

```
DECLARE
  TYPE staffcurtyp is REF CURSOR RETURN
    staff_master%rowtype;
  staff_cv  staffcurtyp; -- declare cursor variable
  staff_cur   staff_master%rowtype;
BEGIN
  open staff_cv for select * from staff_master;
LOOP
  EXIT WHEN staff_cv%notfound;
  FETCH staff_cv into staff_cur;
  INSERT into temp_table VALUES (staff_cv.staff_code,
    staff_cv.staff_name,staff_cv.staff_sal);
END LOOP;
CLOSE staff_cv;
END;
```



Copyright © Capgemini 2015. All Rights Reserved 42

- In the example shown on the slide, using a cursor variable we are fetching data from a table and inserting it in a temp_table.

Summary

- In this lesson, you have learnt:
 - Cursor is a “handle” or “name” for a private SQL area.
 - Implicit cursors are declared for queries that return only one row.
 - Explicit cursors are declared for queries that return more than one row.
 - Like a Cursor, a Cursor Variable points to the current row in the result set of a multi-row query.
 - However, Cursors and Cursor Variables are not interoperable.



Review Question

- Question 1: A “Cursor” is static whereas a “Cursor Variable” is dynamic because it is not tied to a specific query.
 - True / False
- Question 2: %COUNT returns number of rows fetched from the cursor area by using FETCH command.
 - True / False



Review Question

- Question 3: Implicit SQL cursor is opened or closed by the program.
 - True / False
- Question 4: A ___ specifies a Return Type.
- Question 5: PL/SQL provides a shortcut via a ___ Loop, which implicitly handles the cursor processing.



Oracle (PL/SQL)

Lesson 5: Exception Handling and Dynamic SQL

Lesson Objectives

- To understand the following topics:
 - Error Handling
 - Predefined Exceptions
 - Numbered Exceptions
 - User Defined Exceptions
 - Raising Exceptions
 - Control passing to Exception Handlers
 - RAISE_APPLICATION_ERROR



5.1: Error Handling (Exception Handling)

Understanding Exception Handling in PL/SQL

- **Error Handling:**
 - In PL/SQL, a warning or error condition is called an “exception”.
 - Exceptions can be internally defined (by the run-time system) or user defined.
 - Examples of internally defined exceptions:
 - division by zero
 - out of memory
 - Some common internal exceptions have predefined names, namely:
 - ZERO_DIVIDE
 - STORAGE_ERROR

Error Handling:

- A good programming language should provide capabilities of handling errors and recovering from them if possible.
- PL/SQL implements Error Handling via “exceptions” and “exception handlers”.

Types of Errors in PL/SQL

- **Compile Time errors:** They are reported by the PL/SQL compiler, and you have to correct them before recompiling.
- **Run Time errors:** They are reported by the run-time engine. They are handled programmatically by raising an exception, and catching it in the Exception section.

5.1: Error Handling (Exception Handling)

Understanding Exception Handling in PL/SQL

- The other exceptions can be given user-defined names.
- Exceptions can be defined in the declarative part of any PL/SQL block, subprogram, or package. These are user-defined exceptions.



Copyright © Capgemini 2015. All Rights Reserved 4

5.1: Error Handling (Exception Handling)

Declaring Exception

- Exception is an error that is defined by the program.
 - It could be an error with the data, as well.
- There are three types of exceptions in Oracle:
 - Predefined exceptions
 - Numbered exceptions
 - User defined exceptions



Copyright © Capgemini 2015. All Rights Reserved 5

Declaring Exceptions:

- Exceptions are declared in the Declaration section, raised in the Executable section, and handled in the Exception section.

5.2: Declaring Exceptions

Predefined Exception

- Predefined Exceptions correspond to the most common Oracle errors.
 - They are always available to the program. Hence there is no need to declare them.
 - They are automatically raised by ORACLE whenever that particular error condition occurs.
 - Examples: NO_DATA_FOUND,CURSOR_ALREADY_OPEN, PROGRAM_ERROR

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 6

Predefined Exceptions:

- An internal exception is raised implicitly whenever your PL/SQL program violates an Oracle rule or exceeds a system-dependent limit. Every Oracle error has a number, but exceptions must be handled by name. So, PL/SQL predefines some common Oracle errors as exceptions. For example, PL/SQL raises the predefined exception NO_DATA_FOUND if a SELECT INTO statement returns no rows.
- Given below are some Predefined Exceptions:
 - NO_DATA_FOUND
 - This exception is raised when SELECT INTO statement does not return any rows.
 - TOO_MANY_ROWS
 - This exception is raised when SELECT INTO statement returns more than one row.
 - INVALID_CURSOR
 - This exception is raised when an illegal cursor operation is performed such as closing an already closed cursor.
 - VALUE_ERROR
 - This exception is raised when an arithmetic, conversion, truncation, or constraint error occurs in a procedural statement.
 - DUP_VAL_ON_INDEX
 - This exception is raised when the UNIQUE CONSTRAINT is violated.

5.2: Declaring Exceptions

Predefined Exception - Example

- In the following example, the built in exception is handled.

```
DECLARE
    v_staffno  staff_master.staff_code%type;
    v_name      staff_master.staff_name%type;
BEGIN
    SELECT staff_name into v_name FROM staff_master
    WHERE staff_code=&v_staffno;
    dbms_output.put_line(v_name);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        dbms_output.put_line('Not Found');
END;
/
```



Copyright © Capgemini 2015. All Rights Reserved 7

Predefined Exceptions:

In the example shown on the slide, the NO_DATA_FOUND built in exception is handled. It is automatically raised if the SELECT statement does not fetch any value and populate the variable.

5.2: Declaring Exceptions

Numbered Exception

- An exception name can be associated with an ORACLE error.
 - This gives us the ability to trap the error specifically to ORACLE errors
 - This is done with the help of “compiler directives” –
 - PRAGMA EXCEPTION_INIT



Copyright © Capgemini 2015. All Rights Reserved B

Numbered Exception:

The Numbered Exceptions are Oracle errors bound to a user defined exception name.

5.2: Declaring Exceptions

Numbered Exception

■ PRAGMA EXCEPTION_INIT:

- A PRAGMA is a compiler directive that is processed at compile time, not at run time. It is used to name an exception.
- In PL/SQL, the PRAGMA EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number.
 - This arrangement lets you refer to any internal exception(error) by name, and to write a specific handler for it.
- When you see an error stack, or sequence of error messages, the one on top is the one that you can trap and handle.



Copyright © Capgemini 2015. All Rights Reserved 9

5.2: Declaring Exceptions

Numbered Exception (Contd.)

- User defined exceptions can be named with error number between -20000 and -20999.
- The naming is declared in Declaration section.
- It is valid within the PL/SQL blocks only.
- Syntax is:

```
PRAGMA EXCEPTION_INIT(Exception Name,Error_Number);
```



Copyright © Capgemini 2015. All Rights Reserved 10

5.2: Declaring Exceptions

Numbered Exception - Example

- A PL/SQL block to handle Numbered Exceptions

```
DECLARE
    v_bookno number := 10000008;
    child_rec_found EXCEPTION;
    PRAGMA EXCEPTION_INIT (child_rec_found, -2292);
BEGIN
    DELETE from book_master
    WHERE book_code = v_bookno;
EXCEPTION
    WHEN child_rec_found THEN
        INSERT into error_log
        VALUES ('Book entries exist for book:' || v_bookno);
END;
```

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 11

If a user tries to delete record from the parent table wherein child records exist an error is raised by Oracle. We would want to handle this error through the PL/SQL block which is deleting records from a parent table. The example on the slide demonstrates this. In the PL/SQL block we are binding the constraint exception raised by Oracle to user defined exception name.

All oracle errors are negative i.e prefixed with a minus symbol.

In the example we are mapping error-2292 which occurs when referential integrity rule is violated.

5.2: Declaring Exceptions

User-defined Exception

- User-defined Exceptions are:
 - declared in the Declaration section,
 - raised in the Executable section, and
 - handled in the Exception section



Copyright © Capgemini 2015. All Rights Reserved 12

User-Defined Exceptions:

- These exception are entirely user defined based on the application. The programmer is responsible for declaring, raising and handling them.

5.2: Declaring Exceptions

User-defined Exception - Example

- Here is an example of User Defined Exception:

```
DECLARE
    E_Balance_Not_Sufficient EXCEPTION;
    E_Comm_Too_Large EXCEPTION;
    ...
BEGIN
    NULL;
END;
```



Copyright © Capgemini 2015. All Rights Reserved 13

5.3: User Defined Exceptions

Raising Exceptions

- Raising Exceptions:
 - Internal exceptions are raised implicitly by the run-time system, as are user-defined exceptions that are associated with an Oracle error number using EXCEPTION_INIT.
 - Other user-defined exceptions must be raised explicitly by RAISE statements.
 - The syntax is:

```
RAISE Exception_Name;
```

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 14

Raising Exceptions:

When the error associated with an exception occurs, the exception is raised.

This is done through the RAISE command.

5.3: User Defined Exceptions

Raising Exceptions - Example

- An exception is defined and raised as shown below:

```
DECLARE
  ...
  retired_emp EXCEPTION ;
BEGIN
  pl/sql_statements ;
  if error_condition then
    RAISE retired_emp ;
  pl/sql_statements ;
EXCEPTION
  WHEN retired_emp THEN
  pl/sql_statements ;
END ;
```

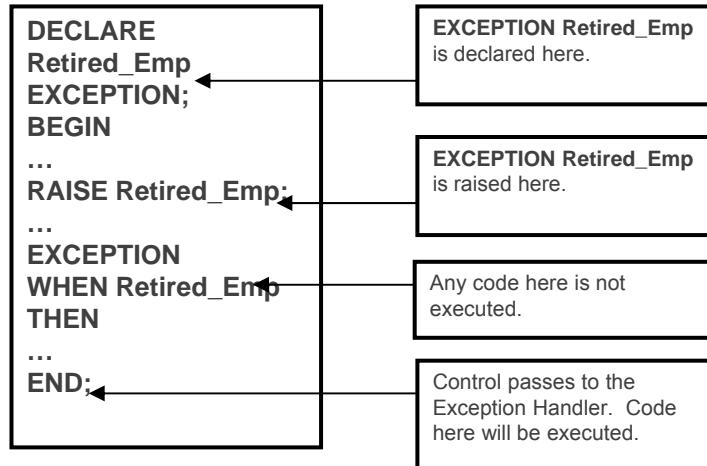


Copyright © Capgemini 2015. All Rights Reserved 15

Control passing to Exception Handler

- Control passing to Exception Handler :
 - When an exception is raised, normal execution of your PL/SQL block or subprogram stops, and control passes to its exception-handling part.
 - To catch the raised exceptions, you write "exception handlers".
 - Each exception handler consists of a WHEN clause, which specifies an exception, followed by a sequence of statements to be executed when that exception is raised.
 - These statements complete execution of the block or subprogram, however, the control does not return to where the exception was raised. In other words, you cannot resume processing where you left off.

Control passing to Exception Handler:



5.3: User Defined Exceptions

User-defined Exception - Example

▪ User Defined Exception Handling:

```
DECLARE
    dup_deptno EXCEPTION;
    v_counter binary_integer;
    v_department number(2) := 50;
BEGIN
    SELECT count(*) into v_counter FROM department_master
    WHERE dept_code=50;
    IF v_counter > 0 THEN
        RAISE dup_deptno ;
    END IF;
    INSERT into department_master
    VALUES (v_department , 'new name');
EXCEPTION
    WHEN dup_deptno THEN
        INSERT into error_log
        VALUES ('Dept: '|| v_department ||' already exists');
END ;
```

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 17

The example on the slide demonstrates user-defined exceptions. It checks for department no value to be inserted in the table. If the value is duplicated it will raise an exception.

5.3: User Defined Exceptions

Others Exception Handler

■ OTHERS Exception Handler:

- The optional OTHERS exception handler, which is always the last handler in a block or subprogram, acts as the handler for all exceptions that are not specifically named in the Exception section.
- A block or subprogram can have only one OTHERS handler.
- To handle a specific case within the OTHERS handler, predefined functions SQLCODE and SQLERRM are used.
 - SQLCODE returns the current error code. And SQLERRM returns the current error message text.
 - The values of SQLCODE and SQLERRM should be assigned to local variables before using it within a SQL statement.

5.3: User Defined Exceptions

Others Exception Handler - Example

```
DECLARE
    v_dummy varchar2(1);
    v_designation number(2) := 109;
BEGIN
    SELECT 'x' into v_dummy FROM designation_master
    WHERE design_code= v_designation;
    INSERT into error_log
    VALUES ('Designation: ' || v_designation || 'already exists');
EXCEPTION
    WHEN no_data_found THEN
        insert into designation_master values (v_designation,'newdesig');
    WHEN OTHERS THEN
        Err_Num = SQLCODE;
        Err_Msg =SUBSTR(SQLERRM, 1, 100);
        INSERT into errors VALUES( err_num, err_msg );
END ;
```

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 19

The example on the slide uses OTHERS Exception handler. If the exception that is raised by the code is not NO_DATA_FOUND, then it will go to the OTHERS exception handler since it will notice that there is no appropriate exception handler defined.

Also observe that the values of SQLCODE and SQLERRM are assigned to variables defined in the block.

5.3: User Defined Exceptions

Raise_Application_Error

- RAISE_APPLICATION_ERROR:
- The procedure RAISE_APPLICATION_ERROR lets you issue user-defined ORA-error messages from stored subprograms.
- In this way, you can report errors to your application and avoid returning unhandled exceptions.
- Syntax:

```
RAISE_APPLICATION_ERROR( Error_Number, Error_Message);
```
- where:
 - Error_Number is a parameter between -20000 and -20999
 - Error_Message is the text associated with this error

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 20

Raise Application Error:

The built-in function RAISE_APPLICATION_ERROR is used to create our own error messages, which can be more descriptive and user friendly than Exception Names.

5.3: User Defined Exceptions

Raise_Application_Error - Example

- Here is an example of Raise Application Error:

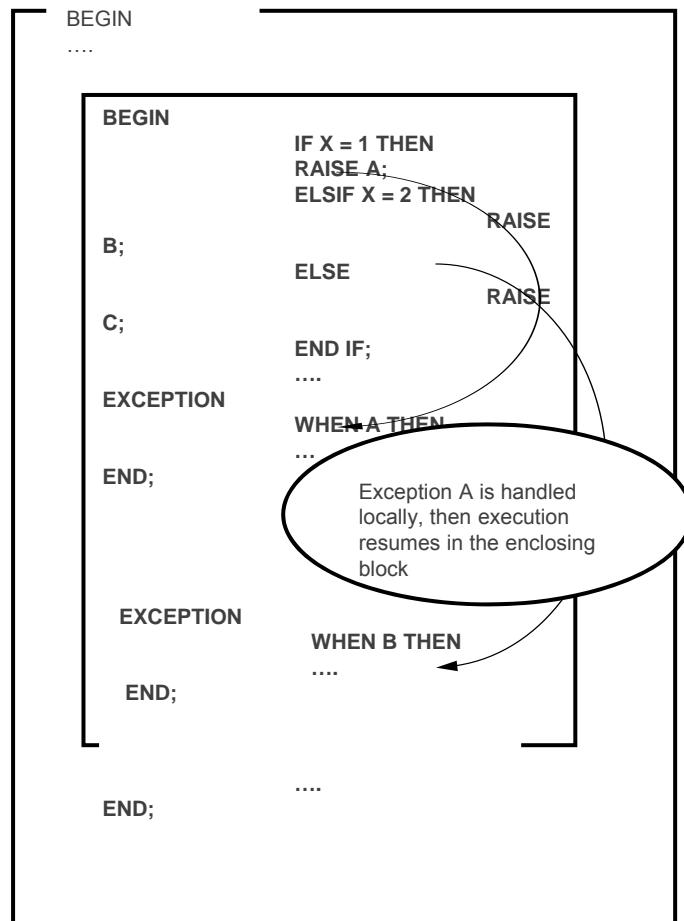
```
DECLARE
    /* VARIABLES */
BEGIN
    .....
    .....
EXCEPTION
    WHEN OTHERS THEN
        -- Will transfer the error to the calling environment
        RAISE_APPLICATION_ERROR( -20999 , 'Contact DBA');
END ;
```



Copyright © Capgemini 2015. All Rights Reserved 21

Propagation of Exceptions:

- When an exception is raised, if PL/SQL cannot find a handler for it in the current block or subprogram, then the exception propagates. That is, the exception reproduces itself in successive enclosing blocks until a handler is found or there are no more blocks to search.



Masking Location of an Error:

- Since the same Exception section is examined for the entire block, it can be difficult to determine, which SQL statement caused the error.
- For example: Consider the following block:

```

SELECT
SELECT
SELECT
EXCEPTION
WHEN NO_DATA_FOUND THEN
--You Don't Know which caused the
NO_DATA_FOUND
END ;

```

- There are two methods to solve this problem:

```

DECLARE
V_Counter NUMBER:= 1;
BEGIN
SELECT .....
V_Counter := 2;
SELECT .....
V_Counter :=3;
SELECT ...
WHEN NO_DATA_FOUND THEN
-- Check values of V_Counter to find out which
SELECT statement
-- caused the exception NO_DATA_FOUND
END ;

```

- The second method is to put the statement in its own block:

```

BEGIN
-- PL/SQL Statements
BEGIN
SELECT .....
EXCEPTION
WHEN NO_DATA_FOUND THEN
--
END;
BEGIN
SELECT .....
EXCEPTION
WHEN NO_DATA_FOUND THEN
--
END;
BEGIN
SELECT .....
EXCEPTION
WHEN NO_DATA_FOUND THEN
--
END;
END ;

```

contd.

Masking Location of an Error (contd.):

```
BEGIN  
-----  
/* PL/SQL statements */  
BEGIN  
SELECT .....  
WHEN NO_DATA_FOUND THEN  
-- Process the error for NO_DATA_FOUND  
END;  
  
/* Some more PL/SQL statements  
This will execute irrespective of when  
NO_DATA_FOUND */  
END;
```

5.4: Dynamic SQL

Introduction

- Dynamic SQL allows an application to run SQL statements whose contents are not known until runtime.
- The statement is built up as a string by the application and is then passed to the server, in a way that is similar to the ADO interface in VB.

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 25

Dynamic SQL:

- Most database applications do a specific job.
- For example:
 - A simple program might prompt the user for an employee number, then update rows in the EMP and DEPT tables.
 - In this case, you know the makeup of the UPDATE statement at pre-compile time. That is, you know which tables might be changed, the constraints defined for each table and column, the columns that might be updated, and the datatype of each column.
- However, some applications must accept (or build), and process a variety of SQL statements at run time.
- For example:
 - A general-purpose report writer must build different SELECT statements for the various reports it generates.
 - In this case, the statement's makeup is unknown until run time. Such statements can, and probably will, change from execution to execution. They are aptly called "Dynamic SQL statements".
- "Dynamic SQL statements" are stored in character strings built by your program at run time. Such strings must contain the text of a valid SQL statement or PL/SQL block. They can also contain placeholders for bind arguments.
 - A placeholder is an undeclared identifier, so its name, to which you must prefix a colon, does not matter.

5.4: Dynamic SQL

Drawbacks of using Dynamic SQL

▪ Disadvantages of Dynamic SQL:

- Generally, Dynamic SQL is slower than Static SQL. Hence it should be used only when absolutely necessary.
- Besides, "syntax checking" and "object validation" cannot be done until runtime, hence code containing large amounts of Dynamic SQL may be littered with mistakes and still compile.



Copyright © Capgemini 2015. All Rights Reserved 20

Disadvantages of Dynamic SQL:

Some dynamic queries require complex coding, the use of special data structures, and more runtime processing.

While you might not notice the added processing time, you might find the coding difficult unless you fully understand dynamic SQL concepts and methods.

5.4: Dynamic SQL

Benefits of using Dynamic SQL

- Advantages of Dynamic SQL:
 - Primarily, Dynamic SQL allows you to perform DDL commands that are not supported directly within PL/SQL.
For example: Creating tables.
 - Dynamic SQL also allows you to access objects that will not exist until runtime.
 - DDL Operations
 - Single Row Queries
 - Dynamic Cursors
 - Native Dynamic SQL versus DBMS_SQL

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 27

Advantages of Dynamic SQL:

- Host programs that accept and process dynamically defined SQL statements are more versatile than plain embedded SQL programs.
- Dynamic SQL statements can be built interactively with input from users having little or no knowledge of SQL.
- **For example:** Your program might simply prompt users for a search condition to be used in the WHERE clause of a SELECT, UPDATE, or DELETE statement. A more complex program might allow users to choose from menus listing SQL operations, table and view names, column names, and so on. Thus, Dynamic SQL lets you write highly flexible applications.

When to use Dynamic SQL:

- In practice, Static SQL will meet nearly all your programming needs. Use Dynamic SQL only if you need its open-ended flexibility. Its use is suggested when one of the following items is unknown at pre-compile time:
 - Text of the SQL statement (commands, clauses, and so on)
 - The number of host variables
 - The datatypes of host variables
 - References to database objects such as columns, indexes, sequences, tables, usernames, and views

5.4: Dynamic SQL

Benefits of using Dynamic SQL (Contd.)

- DDL Operations:

- Commands such as DDL operations, which are not directly supported by PL/SQL, can be performed by using Dynamic SQL:

```
BEGIN  
EXECUTE IMMEDIATE 'TRUNCATE TABLE my_table';  
END; /
```



Copyright © Capgemini 2015. All Rights Reserved 28

5.4: Dynamic SQL

Benefits of using Dynamic SQL (Contd.)

- Single Row Queries

- Given below is a rather simplistic example of a single row query:

```
DECLARE v_sql VARCHAR2(100); v_date DATE; BEGIN  
v_sql := 'SELECT Sysdate FROM dual'; EXECUTE  
IMMEDIATE v_sql INTO v_date; END; /
```



Copyright © Capgemini 2015. All Rights Reserved 29

How Dynamic SQL statements are processed?

- Typically, an application program prompts the user for the text of a SQL statement, and the values of host variables used in the statement. Then Oracle parses the SQL statement. That is, Oracle examines the SQL statement to make sure it follows syntax rules and refers to valid database objects. Parsing also involves checking database access rights, reserving needed resources, and finding the optimal access path.
- Next, Oracle binds the host variables to the SQL statement. That is, Oracle gets the addresses of the host variables so that it can read or write their values.
- Then Oracle executes the SQL statement. That is, Oracle does what the SQL statement requested, such as deleting rows from a table.
- The SQL statement can be executed repeatedly by using new values for the host variables.
- There are many constructs available in Dynamic SQL, which allow a developer to use them from time to time.
- **For Example:** EXECUTE IMMEDIATE

EXECUTE IMMEDIATE

Syntax:

```
EXECUTE IMMEDIATE dynamic_string
[INTO {define_variable[, define_variable]... | record}]
[USING [IN | OUT | IN OUT] bind_argument
 [, [IN | OUT | IN OUT] bind_argument]...];
```

where

- `dynamic_string` is a string expression that represents a SQL statement or PL/SQL block
- `define_variable` is a variable that stores a SELECTed column value
- `record` is a user-defined or %ROWTYPE record that stores a SELECTed row
- `bind_argument` is an expression whose value is passed to the dynamic SQL statement or PL/SQL block.

contd.

How Dynamic SQL statements are processed? (contd.)

- The dynamic_string can contain any valid SQL statement (except multi row select query), or a valid PL/SQL block without the terminator. The string can also contain placeholder for bind variables. However there is a restriction, you cannot use placeholder for defining schema objects (table name etc.)
- The INTO clause used for single row queries, specifies the columns or the record type variable into which the column values are fetched. For each value returned by the query, there must be a type compatible variable defined in the INTO clause.
- You must put all the bind arguments in the USING clause. The default parameter mode is IN. At run time, any bind arguments in the USING clause replace corresponding placeholders in the SQL statement or PL/SQL block. So, every placeholder must be associated with a bind argument in the USING clause. Only numeric, character and string literals are allowed in the USING clause. Boolean variables like TRUE, FALSE and NULLs are not permitted in the USING clause.

5.4: Dynamic SQL

Execute Immediate Statement - Example

- To run a DDL statement in PL/SQL, refer the following example:

```
begin  
execute immediate 'set role all';  
end;
```



Copyright © Capgemini 2015. All Rights Reserved 32

5.4: Dynamic SQL

Execute Immediate Statement - Example

- To retrieve values from a Dynamic statement (INTO clause), refer the following example:

```
DECLARE
  l_cnt varchar2(20);
BEGIN EXECUTE IMMEDIATE 'SELECT count(1) FROM
  staff_master' INTO l_cnt; dbms_output.put_line(l_cnt);
END;
```



Copyright © Capgemini 2015. All Rights Reserved 33

More Examples:

- To dynamically call a routine, refer following example:
 - The bind variables used for parameters of the routine have to be specified along with the parameter type.
 - IN type is the default, others have to be explicitly specified.

```
DECLARE
  l_routin varchar2(100) := 'gen2161.get_rowcnt';
  l_tblnam varchar2(20) := 'emp'; l_cnt number;
  l_status varchar2(200);
BEGIN EXECUTE IMMEDIATE 'BEGIN ' || l_routin ||
  '(2, :3, :4); END;' using l_tblnam, out l_cnt, in out
  l_status;
IF l_status != 'OK' THEN dbms_output.put_line('error');
END IF;
END;
```

To return value into a PL/SQL record type, refer the following example.

- The same option can be used for %rowtype variables, as well.

```
DECLARE
  type empdtlrec is record (empno number(4),
  ename varchar2(20), deptno number(2)); empdtl
  empdtlrec; BEGIN
    EXECUTE IMMEDIATE 'SELECT staff_code,
    staff_name, deptcode ' || 'FROM staff_master
    WHERE staff_code = 100001' into empdtl;
  END;
```

Execute Immediate Statement - Example

- To pass values to a Dynamic statement (USING clause), refer the following example:

```
DECLARE
    depnam varchar2(20) := 'testing';
    l_loc varchar2(10) := 'Dubai';
BEGIN
    EXECUTE IMMEDIATE 'INSERT into dept
VALUES (:1, :2, :3)' using 50, l_depnam,l_loc;
    commit;
END;
```



Copyright © Capgemini 2015. All Rights Reserved 34

More Examples:

- To pass and retrieve values, refer the following example:
 - The INTO clause should precede the USING clause.

```
DECLARE
    l_dept pls_integer := 20;
    l_nam varchar2(20);
    l_loc varchar2(20);
BEGIN
    execute immediate 'select dname, loc from
dept where deptno = :1' into l_nam, l_loc using
l_dept ;
END;
```

Some more examples of EXECUTE IMMEDIATE:

- **Example 1:**

```
DECLARE  
sql_stmt varchar2(100);  
plsql_block varchar2(200);  
my_deptno number(2) := 50;  
my_dname varchar2(15) := 'PERSONNEL';  
my_loc varchar2(15) := 'DALLAS';  
emp_rec emp%ROWTYPE;  
BEGIN
```

```
    :  
    sql_stmt := 'INSERT INTO dept VALUES (:1, :2,  
    :3)';  
    EXECUTE IMMEDIATE sql_stmt  
    USING my_deptno, my_dname, my_loc;
```

- **Example 1a:**

Example to use INSERT. Here 3 place holder columns :1, :2, :3 are used.

```
    :  
    EXECUTE IMMEDIATE  
    'DELETE FROM dept WHERE deptno = :n'  
    USING my_deptno;
```

- **Example 1b:**

Example to use DELETE. Here we are using the SQL statement directly with EXECUTE IMMEDIATE.

```
    :  
    sql_stmt := 'SELECT * FROM emp WHERE  
    empno = :id';  
    EXECUTE IMMEDIATE sql_stmt INTO  
    emp_rec USING 7788;
```

- **Example 1c:**

Example to use INTO clause. In this case entire record is stored in emp_rec for a specific employee code.

contd.

Some more examples of EXECUTE IMMEDIATE (contd.):

- **Example 1d:**

Example to use PL/SQL. Here we are calling a stored procedure.

```
plsql_block := 'BEGIN
emp_stuff.raise_salary(:id,
amt);END;';
EXECUTE IMMEDIATE plsql_block USING
7788, 500;
```

- **Example 1e:**

Example to use DDL.

```
EXECUTE IMMEDIATE
'CREATE TABLE bonus (id
NUMBER, amt NUMBER);'
```

- **Example 1f:**

Example to use Alter Session

```
sql_stmt := 'ALTER SESSION SET
NLS_DATE_FORMAT='MM/DD/YYYY';
EXECUTE IMMEDIATE sql_stmt;
END;'
```

- **Example 2:**

The following procedure accepts two parameters, namely table name and an optional WHERE clause. If the WHERE clause is NULL, then it deletes all the rows from the table, otherwise it deletes the rows which satisfy the criteria.

```
CREATE PROCEDURE Delete_Rows
(table_name IN VARCHAR2,
condition IN varchar2 DEFAULT NULL)
AS
    where_clause varchar2(100) := ''
WHERE '' || condition;
BEGIN
    IF condition IS NULL THEN
        where_clause := NULL;
    END IF;
    EXECUTE IMMEDIATE 'DELETE FROM ' ||
table_name || where_clause;
EXCEPTION
    WHEN OTHERS THEN
        RAISE_APPLICATION_ERROR(-
20001,
        'Error in performing operation');
END;
```

contd.

Some more examples of EXECUTE IMMEDIATE usage (contd.):

- **Example 3:**

Suppose you want to pass NULLs to a Dynamic SQL statement.

Then, you might write the following EXECUTE IMMEDIATE statement:

```
EXECUTE IMMEDIATE 'UPDATE emp SET  
comm = :x' USING NULL;
```

However, this statement fails with a bad expression error because the literal NULL is not allowed in the USING clause. To work around this restriction, simply replace the keyword NULL with an uninitialized variable, as follows :

```
DECLARE  
    a_null CHAR(1); -- set to NULL automatically  
    at run time  
BEGIN  
    EXECUTE IMMEDIATE 'UPDATE emp SET  
    comm = :x' USING a_null;  
END;
```

OPEN-FOR, FETCH, and CLOSE Statements:

- The combination of these three statements allows you to perform multi-row select query. The concept is similar to cursors used earlier.
- There are three steps involved:
 - Open a cursor
 - Fetch rows from the cursor
 - Close the cursor
- Syntax:

```
OPEN {cursor_variable | :host_cursor_variable}
FOR dynamic_string
[USING bind_argument[,bind_argument]...];
```

where:

- Cursor_variable is a cursor of weak type (ref cursor)
- host_cursor_variable is a cursor variable declared in a PL/SQL host environment such as an Pro*c or SQL*J program
- dynamic_string is a string expression that represents a multi-row query
- Syntax:

```
FETCH {cursor_variable | :host_cursor_variable}
INTO {define_variable[, define_variable]... | record};
```

where:

- Cursor_variable and :host_cursor_variable has the same meaning as in OPEN.
- Syntax:

```
CLOSE {cursor_variable |
:host_cursor_variable};
```

where:

- Cursor_variable and :host_cursor_variable has the same meaning as in OPEN.

contd.

OPEN-FOR, FETCH, and CLOSE Statements (contd.)

- Example 1:

```

DECLARE
    TYPE EmpCurTyp IS REF
    CURSOR; --define weak REF CURSOR type
    emp_cv EmpCurTyp; -- declare
    cursor variable
        my_ename VARCHAR2(15);
        my_sal NUMBER := 1000;
BEGIN
    OPEN emp_cv FOR
        -- open cursor
    variable
        'SELECT ename, sal FROM emp
    WHERE sal > :s'
        USING my_sal;
    LOOP
        FETCH
        emp_cv INTO my_ename, my_sal; -- fetch
        next row
        EXIT WHEN
        emp_cv%NOTFOUND; -- exit loop when last
        row
        -- is fetched
        --process row
        ..
    END LOOP;
    CLOSE emp_cv; -- close cursor
variable
END;

```

- Example 2: The following example allows you to fetch rows from the result set of a dynamic multi-row query into a record:

```

DECLARE
    TYPE EmpCurTyp IS REF CURSOR;
    emp_cv EmpCurTyp;
    emp_rec emp%ROWTYPE;
    sql_stmt VARCHAR2(100);
    my_job VARCHAR2(15) := 'CLERK';
BEGIN
    sql_stmt := 'SELECT * FROM emp WHERE
    job = :j';
    OPEN emp_cv FOR sql_stmt USING
    my_job;
    LOOP
        FETCH emp_cv INTO emp_rec;
        EXIT WHEN emp_cv%NOTFOUND;
        --process record
        ..
    END LOOP;
    CLOSE emp_cv;
END;

```

Specifying MODE for parameters:

- You need not specify a parameter mode for input bind arguments (those used, for example, in the WHERE clause) because the mode defaults to IN.
- You can specify the OUT mode for output bind arguments used in the RETURNING clause of an INSERT, UPDATE, or DELETE statement.
- **For example:**

```

DECLARE
  sql_stmt VARCHAR2(100);
  old_loc VARCHAR2(15);
BEGIN
  sql_stmt := 'DELETE FROM dept WHERE
  deptno = 20 RETURNING loc INTO :x';
  EXECUTE IMMEDIATE sql_stmt USING OUT
  old_loc;
  ...
END;

```

- Likewise, when appropriate, you must specify the OUT or IN OUT mode for bind arguments passed as parameters.
- **For example:** Suppose you want to call the following stand-alone procedure:

```

CREATE PROCEDURE Create_Dept (
  deptno INOUT
  number,
  dname IN
  varchar2,
  loc IN varchar2)
AS
BEGIN
  deptno := deptno_seq.NEXTVAL;
  INSERT INTO dept VALUES (deptno, dname, loc);
END;

```

- To call the procedure from a dynamic PL/SQL block, you must specify the IN OUT mode for the bind argument associated with formal parameter deptno, as follows:

```

DECLARE
  plsql_block varchar2(200);
  new_deptno number(2);
  new_dname varchar2(15) := 'ADVERTISING';
  new_loc  varchar2(15) := 'NEW YORK';
BEGIN
  plsql_block := 'BEGIN create_dept(:a, :b, :c)';
  END;';
  EXECUTE IMMEDIATE plsql_block
    USING IN OUT new_deptno, new_dname,
  new_loc;
  IF new_deptno > 90 THEN
  ...
END;

```

Using Duplicate Placeholders:

- Placeholders in a Dynamic SQL statement are associated with bind arguments in the USING clause by position, and not by name. So, if the same placeholder appears two or more times in the SQL statement, each appearance must correspond to a bind argument in the USING clause.
- However, only the unique placeholders in a Dynamic PL/SQL block are associated with bind arguments in the USING clause by position. So, if the same placeholder appears two or more times in a PL/SQL block, all appearances correspond to one bind argument in the USING clause. In the example shown below, the first unique placeholder (x) is associated with the first bind argument (a). Likewise, the second unique placeholder (y) is associated with the second bind argument (b).

```
DECLARE
  a number := 4;
  b number := 7;
BEGIN
  plsql_block := 'BEGIN calc_stats(:x, :x, :y, :x);
END;';
  EXECUTE IMMEDIATE plsql_block USING a,
  b;
  ...
END;
```

Summary

- In this lesson, you have learnt about:
 - Exception Handling
 - User-defined Exceptions
 - Predefined Exceptions
 - Control passing to Exception Handler
 - OTHERS exception handler
 - Association of Exception name to Oracle errors
 - RAISE_APPLICATION_ERROR procedure
 - Dynamic SQL



Summary

Review Question

- Question 1: The procedure ___ lets you issue user-defined ORA-error messages from stored subprograms
- Question 2: The ___ tells the compiler to associate an exception name with an Oracle error number.
- Question 3: ___ returns the current error code. And ___ returns the current error message text.



Oracle (PL/SQL)

Procedures, Functions, and Packages

Lesson Objectives

- To understand the following topics:
 - Subprograms in PL/SQL
 - Anonymous blocks versus Stored Subprograms
 - Procedure
 - Subprogram Parameter modes
 - Functions
 - Packages
 - Package Specification and Package Body
 - Autonomous Transactions



6.1: Subprograms in PL/SQL

Introduction

- A subprogram is a named block of PL/SQL.
- There are two types of subprograms in PL/SQL, namely: Procedures and Functions.
- Each subprogram has:
 - A declarative part
 - An executable part or body, and
 - An exception handling part (which is optional)
- A function is used to perform an action and return a single value.

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 3

Subprograms in PL/SQL:

- The subprograms are compiled and stored in the Oracle database as “stored programs”, and can be invoked whenever required. As the subprograms are stored in a compiled form, when called they only need to be executed. Hence this arrangement saves time needed for compilation.
- When a client executes a procedure or function, the processing is done in the server. This reduces the network traffic.
- Subprograms provide the following advantages:
 - They allow you to write a PL/SQL program that meets our need.
 - They allow you to break the program into manageable modules.
 - They provide reusability and maintainability for the code.

6.1: Subprograms in PL/SQL

Comparison

- Anonymous Blocks & Stored Subprograms Comparison

Anonymous Blocks	Stored Subprograms/Named Blocks
1. Anonymous Blocks do not have names.	1. Stored subprograms are named PL/SQL blocks.
2. They are interactively executed. The block needs to be compiled every time it is run.	2. They are compiled at the time of creation and stored in the database itself. Source code is also stored in the database.
3. Only the user who created the block can use the block.	3. Necessary privileges are required to execute the block.

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 4

6.2: Types of Stored Subprograms

Procedures

- A procedure is used to perform an action.
- It is illegal to constrain datatypes.
- Syntax:

```
CREATE PROCEDURE Proc_Name  
    (Parameter {IN | OUT | IN OUT} datatype := value,...) AS  
        Variable_Declaration ;  
        Cursor_Declaration ;  
        Exception_Declaration ;  
    BEGIN  
        PL/SQL_Statements ;  
    EXCEPTION  
        Exception_Definition ;  
    END Proc_Name ;
```



Copyright © Capgemini 2015. All Rights Reserved 5

Procedures:

- A procedure is a subprogram used to perform a specific action.
- A procedure contains two parts:
 - the specification, and
 - the body
- The procedure specification begins with CREATE and ends with procedure name or parameters list. Procedures that do not take parameters are written without a parenthesis.
- The procedure body starts after the keyword IS or AS and ends with keyword END.

contd.

6.3: Procedures

Subprogram Parameter Modes

IN	OUT	IN OUT
The default	Must be specified	Must be specified
Used to pass values to the procedure.	Used to return values to the caller.	Used to pass initial values to the procedure and return updated values to the caller.
Formal parameter acts like a constant.	Formal parameter acts like an uninitialized variable.	Formal parameter acts like an uninitialized variable.
Formal parameter cannot be assigned a value.	Formal parameter cannot be used in an expression, but should be assigned a value.	Formal parameter should be assigned a value.
Actual parameter can be a constant, literal, initialized variable, or expression.	Actual parameter must be a variable.	Actual parameter must be a variable.
Actual parameter is passed by reference (a pointer to the value is passed in).	Actual parameter is passed by value (a copy of the value is passed out) unless NOCOPY is specified.	Actual parameter is passed by value (a copy of the value is passed in and out) unless NOCOPY is specified.

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 6

Subprogram Parameter Modes:

- You use “parameter modes” to define the behavior of “formal parameters”. The three parameter modes are IN (the default), OUT, and INOUT. The characteristics of the three modes are shown in the slide.
- Any parameter mode can be used with any subprogram.
- Avoid using the OUT and INOUT modes with functions.
- To have a function return multiple values is a poor programming practice. Besides functions should be free from side effects, which change the values of variables that are not local to the subprogram.
- Example1:

```

CREATE PROCEDURE split_name
(
    phrase IN VARCHAR2, first OUT VARCHAR2, last OUT
    VARCHAR2
)
IS
    first := SUBSTR(phrase, 1, INSTR(phrase, '-')-1);
    last := SUBSTR(phrase, INSTR(phrase, '-')+1);
    IF first = 'John' THEN
        DBMS_OUTPUT.PUT_LINE('That is a common first name.');
    END IF;
END;

```

Subprogram Parameter Modes (contd.):**Examples:**

Example 2:

```
SQL > SET SERVEROUTPUT ON
SQL > CREATE OR REPLACE PROCEDURE PROC1 AS
  2  BEGIN
  3    DBMS_OUTPUT.PUT_LINE('Hello from procedure ...');
  4  END;
  5 /
Procedure created.
SQL > EXECUTE PROC1
Hello from procedure ...
PL/SQL procedure successfully created.
```

```
SQL > CREATE OR REPLACE PROCEDURE PROC2
  2  (N1 IN NUMBER, N2 IN NUMBER, TOT OUT NUMBER) IS
  3  BEGIN
  4    TOT := N1 + N2;
  5  END;
  6 /
```

Procedure created.

```
SQL > VARIABLE T NUMBER
SQL > EXEC PROC2(33, 66, :T)
```

PL/SQL procedure successfully completed.

```
SQL > PRINT T
```

T

99

6.3: Procedures

Examples

▪ Example 1:

```
CREATE OR REPLACE PROCEDURE raise_salary
( s_no IN number, raise_sal IN number) IS
  v_cur_salary  number;
  missing_salary exception;
BEGIN
  SELECT staff_sal INTO v_cur_salary FROM staff_master
  WHERE staff_code=s_no;
  IF v_cur_salary IS NULL THEN
    RAISE missing_salary;
  END IF ;
  UPDATE staff_master SET staff_sal = v_cur_salary + raise_sal
  WHERE staff_code = s_no ;
EXCEPTION
  WHEN missing_salary THEN
    INSERT into emp_audit VALUES( sno, 'salary is missing');
END raise_salary;
```

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 8

The procedure example on the slide modifies the salary of staff member. It also handles exceptions appropriately. In addition to the above shown exception you can also handle “NO_DATA_FOUND” exception. The procedure accepts two parameters which is the staff_code and amount that has to be given as raise to the staff member.

6.3: Procedures

Examples

■ Example 2:

```
CREATE OR REPLACE PROCEDURE
    get_details(s_code IN number,
                s_name OUT varchar2,s_sal OUT number ) IS
BEGIN
    SELECT staff_name, staff_sal INTO s_name,s_sal
    FROM staff_master WHERE staff_code=s_code;
EXCEPTION
    WHEN no_data_found THEN
        INSERT into auditstaff
        VALUES( 'No employee with id ' || s_code);
        s_name := null;
        s_sal := null;
END get_details ;
```

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 9

The procedure on the slide accept three parameters, one is IN mode and other two are OUT mode. The procedure retrieves the name and salary of the staff member based on the staff_code passed to the procedure. The S_NAME and S_SAL are the OUT parameters that will return the values to the calling environment

6.3: Procedures

Executing a Procedure

- Executing the Procedure from SQL*PLUS environment,
 - Create a bind variables salary and name SQLPLUS by using VARIABLE command as follows:

```
variable salary number
variable name varchar2(20)
```
 - Execute the procedure with EXECUTE command

```
EXECUTE get_details(100003,:Salary, :Name)
```
 - After execution, use SQL*PLUS PRINT command to view results.

```
print salary
print name
```

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 10

Procedures can be executed through command line as shown on the slide or can be called from other procedures/functions/Anonymous PL/SQL blocks.

On the slide the first snippet declares two variables viz. salary and name. The second snippet calls the procedure and passes the actual parameters. The first is a literal string and the next two parameters are empty variables which will be assigned with values within the procedure.

Calling the procedure from an anonymous PL/SQL block

```
DECLARE
    s_no number(10):=&sno;
    sname varchar2(10);
    sal number(10,2);
BEGIN
    get_details(s_no,sname,sal);
    dbms_output.put_line('Name:'||sname||'Salary'||sal);
END;
```

Parameter default values:

- Like variable declarations, the formal parameters to a procedure or function can have default values.
- If a parameter has default values, it does not have to be passed from the calling environment.
 - If it is passed, actual parameter will be used instead of default.
- Only IN parameters can have default values.

Examples:

Example 1:

```
PROCEDURE Create_Dept( New_Deptno IN NUMBER,  
                      New_Dname IN VARCHAR2 DEFAULT 'TEMP') IS  
BEGIN  
    INSERT INTO department_master  
        VALUES ( New_Deptno, New_Dname, New_Loc)  
    ;  
END ;
```

Example 2:

Now consider the following calls to Create_Dept.

```
BEGIN  
Create_Dept( 50);  
-- Actual call will be Create_Dept ( 50,  
'TEMP', 'TEMP')  
  
Create_Dept ( 50, 'FINANCE');  
-- Actual call will be Create_Dept ( 50,  
'FINANCE' ,TEMP')  
  
Create_Dept( 50, 'FINANCE', 'BOMBAY') ;  
-- Actual call will be Create_Dept(50,  
'FINANCE', 'BOMBAY' )  
  
END;
```

Procedures (contd.):**Using Positional, Named, or Mixed Notation for Subprogram Parameters:**

- When calling a subprogram, you can write the actual parameters by using either Positional notation, Named notation, or Mixed notation.
 - **Positional notation:** You specify the same parameters in the same order as they are declared in the procedure. This notation is compact, but if you specify the parameters (especially literals) in the wrong order, the bug can be hard to detect. You must change your code if the procedure's parameter list changes.
 - **Named notation:** You specify the name of each parameter along with its value. An arrow (=>) serves as the “association operator”. The order of the parameters is not significant.
 - **Mixed notation:** You specify the first parameters with “Positional notation”, and then switch to “Named notation” for the last parameters. You can use this notation to call procedures that have some “required parameters”, followed by some “optional parameters”.
- We have already seen a few examples of calling procedures with Positional notation.
- The example shows calling the Create_Dept procedure with named notations

```
Create_Dept (New_Deptno=> 50,  
New_Dname=>'FINANCE');
```

6.4: Types of Stored Subprograms

Functions

- A function is similar to a procedure.
- A function is used to compute a value.
 - A function accepts one or more parameters, and returns a single value by using a return value.
 - A function can return multiple values by using OUT parameters.
 - A function is used as part of an expression, and can be called as Lvalue = Function_Name(Param1, Param2,)
 - Functions returning a single value for a row can be used with SQL statements.

6.4: Types of Stored Subprograms

Functions

▪ Syntax :

```
CREATE FUNCTION Func_Name(Param datatype :=  
    value,...) RETURN datatype1 AS  
    Variable_Declaration ;  
    Cursor_Declaration ;  
    Exception_Declaration ;  
BEGIN  
    PL/SQL_Statements ;  
    RETURN Variable_Or_Value_Of_Type_Datatype1 ;  
EXCEPTION  
    Exception_Definition ;  
END Func_Name ;
```

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 14

6.5: Functions
Examples

■ Example 1:

```
CREATE FUNCTION crt_dept(dno number,
    dname varchar2) RETURN number AS
BEGIN
    INSERT into department_master
    VALUES (dno,dname);
    return 1;
EXCEPTION
    WHEN others THEN
        return 0;
END crt_dept;
```



Copyright © Capgemini 2015. All Rights Reserved 15

Example 2:

- Function to calculate average salary of a department:
 - Function returns average salary of the department
 - Function returns -1, in case no employees are there in the department.
 - Function returns -2, in case of any other error.

```
CREATE OR REPLACE FUNCTION
get_avg_sal(p_deptno in number) RETURN number as
V_Sal number;
BEGIN
    SELECT Trunc(Avg(staff_sal)) INTO V_Sal
    FROM staff_master
    WHERE deptno=P_Deptno;
    IF v_sal is null THEN
        v_sal := -1 ;
    END IF;
    return v_sal;
EXCEPTION
    WHEN others THEN
        return -2; --signifies any other errors
END get_avg_sal;
```

6.5: Functions

Executing a Function

- Executing functions from SQL*PLUS:
 - Create a bind variable Avg salary in SQLPLUS by using VARIABLE command as follows:

variable flag number
 - Execute the Function with EXECUTE command:

EXECUTE :flag:=crt_dept(60,'Production');
 - After execution, use SQL*PLUS PRINT command to view results.

PRINT flag;

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 15

Functions can also be executed through command line as shown on the slide or can be called from other procedures/functions/Anonymous PL/SQL blocks.

The second snippet calls the function and passes the actual parameters. The variable declared earlier is used for collecting the return value from the function

Calling the function from an anonymous PL/SQL block

```
DECLARE
    avgSalary number;
BEGIN
    avgSalary:= _get_avg_sal(20);
    dbms_output.put_line('The average salary of Dept 20
is'|| avgSalary);
END;
```

```
SELECT get_avg_sal(30) FROM staff_master;
```

Calling function using a Select statement

6.5: Functions

Exceptions handling in Procedures and Functions

- If procedure has no exception handler for any error, the control immediately passes out of the procedure to the calling environment.
- Values of OUT and IN OUT formal parameters are not returned to actual parameters.
- Actual parameters will retain their old values.



Copyright © Capgemini 2015. All Rights Reserved 17

Exceptions raised inside Procedures and Functions:

- If an error occurs inside a procedure, an exception (pre-defined or user-defined) is raised.

6.6: Types of Subprograms

Packages

- A package is a schema object that groups all the logically related PL/SQL types, items, and subprograms.
- Packages usually have two parts, a specification and a body, although sometimes the body is unnecessary.
 - The specification (spec for short) is the interface to your applications. It declares the types, variables, constants, exceptions, cursors, and subprograms available for use.
 - The body fully defines cursors and subprograms, and so implements the spec.
- Each part is separately stored in a Data Dictionary.

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 18

Packages:

- Packages are PL/SQL constructs that allow related objects to be stored together. A Package consists of two parts, namely "Package Specification" and "Package Body". Each of them is stored separately in a "Data Dictionary".
- **Package Specification:** It is used to declare functions and procedures that are part of the package. Package Specification also contains variable and cursor declarations, which are used by the functions and procedures. Any object declared in a Package Specification can be referenced from other PL/SQL blocks. So Packages provide global variables to PL/SQL.
- **Package Body:** It contains the function and procedure definitions, which are declared in the Package Specification. The Package Body is optional. If the Package Specification does not contain any procedures or functions and contains only variable and cursor declarations then the body need not be present.
- All functions and procedures declared in the Package Specification are accessible to all users who have permissions to access the Package. Users cannot access subprograms, which are defined in the Package Body but not declared in the Package Specification. They can only be accessed by the subprograms within the Package Body. This facility is used to hide unwanted or sensitive information from users.
- A Package generally consists of functions and procedures, which are required by a specific application or a particular module of an application.

6.6: Types of Subprograms

Packages

■ Note that:

- Packages variables ~ global variables
- Functions and Procedures ~ accessible to users having access to the package
- Private Subprograms ~ not accessible to users

6.6: Types of Subprograms

Packages

- Syntax of Package Specification:

```
CREATE PACKAGE package_name AS  
    variable_declaration ;  
    cursor_declaration ;  
    FUNCTION func_name(param datatype,...) return datatype1 ;  
    PROCEDURE proc_name(param {in|out|in out}datatype,...);  
END package_name ;
```



Copyright © Capgemini 2015. All Rights Reserved 20

The package specification can contain variables, cursors, procedure and functions. Whatever is specified within the packages are global by default and are accessible to users who have the privileges on the package

6.6: Types of Subprograms

Packages

▪ Syntax of Package Body:

```
CREATE PACKAGE BODY package_name AS
    variable_declaration ;
    cursor_declaration ;
    PROCEDURE proc_name(param {IN|OUT|INOUT} datatype,...) IS
    BEGIN
        pl/sql_statements ;
    END proc_name ;
    FUNCTION func_name(param datatype,...) is
    BEGIN
        pl/sql_statements ;
    END func_name ;
END package_name ;
```

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 21

The package body should contain all the procedures and function declared in the package specification. Any variables and cursors declared within the package body are local to the package body and are accessible only within the package. The package body can contain additional procedures and functions apart from the ones declared in package body. The procedures/functions are local to the package and cannot be accessed by any user outside the package.

6.7: Packages

Example

- Creating Package Specification

```
CREATE OR REPLACE PACKAGE pack1 AS  
  PROCEDURE proc1;  
  FUNCTION fun1 return varchar2;  
END pack1;
```



Copyright © Capgemini 2015. All Rights Reserved 22

6.7: Packages

Example

- Creating Package Body

```
CREATE OR REPLACE PACKAGE BODY pack1 AS
  PROCEDURE proc1 IS
    BEGIN
      dbms_output.put_line('hi a message frm procedure');
    END proc1;
    function fun1 return varchar2 IS
    BEGIN
      return ('hello from fun1');
    END fun1;
  END pack1;
```



Copyright © Capgemini 2015. All Rights Reserved 23

6.7: Packages

Executing a Package

- Executing Procedure from a package:

```
EXEC pack1.proc1
Hi a message frm procedure
```
- Executing Function from a package:

```
SELECT pack1.fun1 FROM dual;

FUN1
-----
hello from fun1
```

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 24

Note:

If the specification of the package declares only types, constants, variables, and exceptions, then the package body is not required there. This type of packages only contains global variables that will be used by subprograms or cursors.

6.7: Packages

Package Instantiation

■ Package Instantiation:

- The packaged procedures and functions have to be prefixed with package names.
- The first time a package is called, it is instantiated.

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 25

Package Instantiation:

- The procedure and function calls are the same as in standalone subprograms.
- The packaged procedures and functions have to be prefixed with package names.
- The first time a package is called, it is instantiated.
 - This means that the package is read from disk into memory, and P-CODE is run.
 - At this point, the memory is allocated for any variables defined in the package.
 - Each session will have its own copy of packaged variables, so there is no problem of two simultaneous sessions accessing the same memory locations.

6.7: Packages

Subprograms and Ref Type Cursors

- You can declare Cursor Variables as the formal parameters of Functions and Procedures.

```
CREATE OR REPLACE PACKAGE staff_data AS  
    TYPE staffcurtyp IS ref cursor return  
        staff_master%rowtype;  
    PROCEDURE open_staff_cur (staff_cur INOUT staffcurtyp);  
END staff_data;
```



Copyright © Capgemini 2015. All Rights Reserved 26

Subprograms and Ref Type Cursors:

- You can declare Cursor Variables as the formal parameters of Functions and Procedures.
- In the following example, you define the REF CURSOR type staffCurTyp, then declare a Cursor Variable of that type as the formal parameter of a procedure:

```
DECLARE  
    TYPE staffCurTyp IS REF CURSOR RETURN  
        staff_master%ROWTYPE;  
    PROCEDURE open_staff_cv (staff_cv IN OUT  
        staffCurTyp) IS
```

- Typically, you open a Cursor Variable by passing it to a stored procedure that declares a Cursor Variable as one of its formal parameters.
- The packaged procedure shown in the slide, for example, opens the cursor variable emp_cur.

6.7: Packages

Subprograms and Ref Type Cursors

- Note: Cursor Variable as the formal parameter should be in IN OUT mode.

```
CREATE OR REPLACE PACKAGE BODY staff_data AS
PROCEDURE open_staff_cur (staff_cur INOUT staffcurtyp) IS
BEGIN
    OPEN staff_cur FOR SELECT * FROM staff_master;
    END open_staff_cur;
END emp_data;
```



Copyright © Capgemini 2015. All Rights Reserved 27

6.7: Packages

Subprograms and Ref Type Cursors

- Execution in SQL*PLUS:
 - Step 1: Declare a bind variable in a PL/SQL host environment of type REFCURSOR.

SQL> VARIABLE cv REFCURSOR

- Step 2: SET AUTOPRINT ON to automatically display the query results.

SQL> set autoprint on

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 28

Subprograms and Ref Type Cursors: Execution in SQL*PLUS:

- When you declare a Cursor Variable as the formal parameter of a subprogram that opens the cursor variable, you must specify the IN OUT mode. That way, the subprogram can pass an open cursor back to the caller.
- To see the value of the Cursor Variable on the SQL prompt, you need to do following:
 - Declare a bind variable in a PL/SQL host environment of type REFCURSOR as shown below. The SQL*Plus datatype REFCURSOR lets you declare Cursor Variables, which you can use to return query results from stored subprograms.
SQL> VARIABLE cv REFCURSOR
 - Use the SQL*Plus command SET AUTOPRINT ON to automatically display the query results.
SQL> set autoprint on
 - Now execute the package with the specified procedure along with the cursor as follows :
SQL> execute emp_data.open_emp_cur(:cv);

6.7: Packages

Subprograms and Ref Type Cursors

- Step 3: Execute the package with the specified procedure along with the cursor as follows:

```
SQL> execute staff_data.open_staff_cur(:cv);
```



Copyright © Capgemini 2015. All Rights Reserved 29

6.7: Packages

Subprograms and Ref Type Cursors

- Passing a Cursor Variable as IN parameter to a stored procedure:
 - Step 1: Create a Package Specification

```
CREATE OR REPLACE PACKAGE staffdata AS
    TYPE cur_type is REF CURSOR;
    TYPE staffcurtyp is REF CURSOR
        return staff%rowtype;
    PROCEDURE ret_data (staff_cur INOUT staffcurtyp,
                        choice in number);
END staffdata;
```



Copyright © Capgemini 2015. All Rights Reserved 30

Subprograms and Ref Type Cursors: Passing a Cursor Variable:

- You can pass a Cursor Variable and an IN parameter to a stored procedure, which will execute the queries with different return types.
- In the example shown in the slide, you are passing the cursor as well as the number variable as choice. Depending on the choice you can write multiple queries, and retrieve the output from the cursor.
- When called, the procedure opens the cursor variable emp_cur for the chosen query.

6.7: Packages

Subprograms and Ref Type Cursors (Contd.)

- Step 2: Create a Package Body:

```
CREATE OR REPLACE PACKAGE BODY staffdata AS
  PROCEDURE ret_data (staff_cur INOUT staffcurtyp,
                      choice IN number) IS
    BEGIN
      IF choice = 1 THEN
        OPEN staff_cur FOR select * FROM staff_master
        WHERE staff_dob IS NOT NULL;
      ELSIF choice = 2 THEN
        OPEN staff_cur FOR SELECT * FROM staff_master
        WHERE staff_sal > 2500;
```



Copyright © Capgemini 2015. All Rights Reserved 31

6.7: Packages

Subprograms and Ref Type Cursors

- Step 2: Create a Package Body (Contd.)

```
ELSIF choice = 3 THEN  
    OPEN staff_cur for SELECT * FROM  
        staff_master WHERE dept_code = 20;  
    END IF;  
END ret_data;  
END empdata;
```



Copyright © Capgemini 2015. All Rights Reserved 32

Step 3: To retrieve the values from the cursor:

- Define a variable in SQL *PLUS environment using variable command.
- Set the autoprint command on the SQL prompt.
- Call the procedure with the package name and the relevant parameters.

```
SQL> variable cur refcursor  
SQL> set autoprint on  
SQL> execute staffdata.ret_data(:cur,1);
```

Subprograms and Ref Type Cursors: Passing a Cursor Variable (contd.):

- In a similar manner, you can pass the Cursor Variable as (: cur) and '2' number for second choice in the EmpData.ret_data procedure. This will give you the output for all the employees who have salary above 2500.
- To see the output of the third cursor, use the same package.procedure name with the ': cur' host variable, and choice value which shows all the employees having department number as 20.
- We can also create a package with the different REF CURSOR TYPES available (that is define the REF CURSOR type in a separate package), and then reference that type in the standalone procedure.
- Example 1: Create a package as shown below:

```
SQL> CREATE OR REPLACE PACKAGE cv_types AS
      TYPE GenericCurTyp IS REF CURSOR;
      TYPE staffCurTyp IS REF CURSOR RETURN
          staff_master%ROWTYPE;
      TYPE deptCurTyp IS REF CURSOR
          RETURN department_master%ROWTYPE;
  END cv_types;
/
Package created.
```

```
SQL> CREATE OR REPLACE PROCEDURE open_pro
  (generic_cv IN OUT
   cv_types.GenericCurTyp,choice IN NUMBER) IS
BEGIN
  IF choice = 1 THEN
    OPEN generic_cv FOR SELECT * FROM staff_master;
  ELSIF choice = 2 THEN
    OPEN generic_cv FOR SELECT * FROM
      department_master;
  ELSIF choice = 3 THEN
    OPEN generic_cv FOR SELECT * FROM item;
  END IF;
END open_pro;
/
Package created.
```

- Example 2: Create a standalone procedure that references the REF CURSOR type GenericCurTyp, which is defined in the package cv_types. Hence create a procedure as shown below:

contd.

Subprograms and Ref Type Cursors: Passing a Cursor Variable (contd.):

- Open_procedure, which has a cursor parameter generic pro is an independent _cv, which refers to type REF_CURSOR defined in the cv_types package. You can pass a Cursor Variable and Selector to a stored procedure that executes queries with different return types (that is what you have done in the Open_pro procedure). When you call this procedure with the Generic_cv cursor along with the Selector value, the generic_cv cursor gets open and it retrieves the values from the different tables.
- To execute this procedure you need to create the variable of type REFCURSOR, and pass that variable in the Open_pro procedure to see the output.
- For example:

```
SQL> execute open_pro(:cv,2);
```

This output is that for the choice number 2, that is the Cursor Variable will show all the rows from the Dept table.

6.7: Packages

Autonomous transactions

- Autonomous transactions are useful for implementing:
 - transaction logging,
 - counters, and
 - other such actions, which needs to be performed independent of whether the calling transaction is committed or rolled-back
- Autonomous transactions:
 - are independent of the parent transaction.
 - do not inherit the characteristic of the parent (calling) transaction.

6.7: Packages

Autonomous transactions

- Note that:

- Any changes made cannot be seen by the calling transaction unless they are committed.
- Rollback of the parent does not rollback the called transaction. There are no limits other than the resource limits on how many Autonomous transactions may be nested.
- Autonomous transactions must be explicitly committed or rolled-back, otherwise an error is generated.

6.7: Packages

Autonomous transactions - Example

- The following example shows how to define an Autonomous block.

```
CREATE PROCEDURE LOG_USAGE ( staff_no IN number,  
                           msg_in IN varchar2)  
IS  
PRAGMA AUTONOMOUS_TRANSACTION;  
contd.
```



Copyright © Capgemini 2015. All Rights Reserved 37

6.7: Packages

Autonomous transactions - Example

```
BEGIN
    INSERT into log1 VALUES (staff_no, msg_in);
    commit;
END LOG_USAGE;
CREATE PROCEDURE chg_emp
IS
BEGIN
    LOG_USAGE(7566,'Changing salary '); -- ←
    UPDATE staff_master
    SET staff_sal = sal + 250
    WHERE staff_code = 100003;
END chg_emp;
```



Copyright © Capgemini 2015. All Rights Reserved 38

Note:

- In the example shown in the slide, we are calling log_usage with the employee number and the appropriate message. Then we are updating the corresponding employee record.
- Irrespective of whether the update is successful or not, the insert in the log_usage procedure is always committed.

Definer's and Invoker's Rights Model:

- In case of stored procedures, functions and packages (stored subprograms), there are always two situations.
 - First situation is where a stored subprogram is created by a user.
 - Second case is when an already created stored subprogram is invoked by a privileged user of the database.
- By default whenever a subprogram is invoked, it is executed with the privileges of the creating user. This mechanism is called "**definer's rights model**".
- In "definer's rights model", if the stored subprogram (based on EMP table) is created by the user Scott, and another user (say TRG1) executes the stored subprogram, then the privileges of Scott (owner) is used in the context. In this case, even if TRG1 does not have any privileges on the table EMP (owned by Scott), he can still execute the stored subprogram and perform DML operations on the table EMP. This is because the subprogram is executed in the "definer's rights model". This model available as the default model.
- After the Oracle 8i release, the "**invoker's rights model**" can be used. In this model, the procedure executes under the privileges of the user executing the subprogram.
- In the "invokers rights model", if the user Scott creates a stored subprogram, and another user (say TRG1) executes the subprogram, then the privileges of TRG1 (the invoker) will be used in the context of the subprogram rather than the owner of subprogram (Scott).
- In this case, if the user Vivek does not have sufficient rights on the table EMP (owned by Scott), then the invocation of the subprogram will result in an appropriate error message to the invoker.
- Example: As user Scott:

```
CREATE PROCEDURE NAME_COUNT
AUTHID CURRENT_USER
IS
BEGIN
    DECLARE
        N NUMBER;
    BEGIN
        SELECT COUNT(*) INTO N FROM
SCOTT.STAFF_MASTER;
        INSERT INTO STAFFCOUNT VALUES
(SYSDATE, N);
    END;
END;
```

- Explanation:

- In line 2, we have defined invoker's rights.
- In line 8, we are referring to the table EMP from Scott's schema. (This is needed, otherwise Oracle will look for EMP table in the invokers schema)
- In line 9, the number of employees is inserted into a table empcount which is present in the current users schema.

Summary

- In this lesson, you have learnt:
 - Subprograms in PL/SQL are named PL/SQL blocks.
 - There are two types of subprograms, namely: Procedures and Functions.
 - Procedure is used to perform an action.
 - Procedures have three subprogram parameter modes, namely: IN, OUT, and INOUT.



Summary

Summary

- Functions are used to compute a value.
 - A function accepts one or more parameters, and returns a single value by using a return value.
 - A function can return multiple values by using OUT parameters.
- Packages are schema objects that groups all the logically related PL/SQL types, items, and subprograms.
 - Packages usually have two parts, a specification and a body,



Summary

Review Question

- Question 1: Anonymous Blocks do not have names.
 - True / False
- Question 2: A function can return multiple values by using OUT parameters
 - True / False
- Question 3: A Package consists of “Package Specification” and “Package Body”, each of them is stored in a Data Dictionary named DBMS_package.



Review Question

- Question 4: An ___ parameter returns a value to the caller of a subprogram.
- Question 5: A procedure contains two parts: ___ and ___.
- Question 6: In ___ notation, the order of the parameters is not significant.



Oracle (PL/SQL)

Lesson 7: Database Triggers

Lesson Objectives

- To understand the following topics:
 - Concept of Database Triggers
 - Types of Triggers
 - Disabling and Dropping Triggers
 - Restriction on Triggers
 - Order of Trigger firing
 - Using :Old and :New values, WHEN clause, Trigger predicates
 - Mutating and Constraining tables



7.1: Database Triggers

Concept of Database Triggers

■ Database Triggers:

- Database Triggers are procedures written in PL/SQL, Java, or C that run (fire) implicitly:
 - Whenever a table or view is modified, or
 - When some user actions or database system actions occur
- They are stored subprograms



Copyright © Capgemini 2015. All Rights Reserved 3

Database Triggers:

- A Trigger defines an action the database should take when some database related event occurs.
 - Anonymous blocks as well as stored subprograms need to be explicitly invoked by the user.
- Database Triggers are PL/SQL blocks, which are implicitly fired by ORACLE RDBMS whenever an event such as INSERT, UPDATE, or DELETE takes place on a table.
- Database triggers are also stored subprograms.

Concept of Database Triggers

- You can write triggers that fire whenever one of the following operations occur:

- User events:
 - DML statements on a particular schema object
 - DDL statements issued within a schema or database
 - user logon or logoff events
- System events:
 - Server errors
 - Database startup
 - Instance shutdown



Copyright © Capgemini 2015. All Rights Reserved 4

Usage of Triggers

- **Triggers can be used for:**

- maintaining complex integrity constraints
- auditing information, that is the Audit trail
- automatically signaling other programs that action needs to take place when changes are made to a table



Copyright © Capgemini 2015. All Rights Reserved 5

Database Triggers (contd.):

Triggers can be used for:

- Maintaining complex integrity constraints. This is not possible through declarative constraints that are enabled at table creation.
- Auditing information in a table by recording the changes and the identify of the person who made them. This is called as an audit trail.
- Automatically signaling other programs that action needs to take place when changes are made to a table.

contd.

Syntax of Triggers

- Syntax:

```
CREATE TRIGGER Trg_Name  
{BEFORE | AFTER} {event} OF Column_Names ON Table_Name  
[FOR EACH ROW]  
[WHEN restriction]  
BEGIN  
    PL/SQL statements;  
END Trg_Name ;
```



Copyright © Capgemini 2015. All Rights Reserved 6

Database Triggers (contd.):

To create a trigger on a table you must be able to alter that table.
The slide shows the syntax for creating a table.

contd.

Database Triggers (contd.):**Parts of a Trigger**

A trigger has three basic parts:

- A triggering event or statement
- A trigger restriction
- A trigger action

Triggering Event or Statement

- A triggering event or statement is the SQL statement, database event, or user event that causes a trigger to fire. A triggering event can be one or more of the following:
 - An INSERT, UPDATE, or DELETE statement on a specific table (or view, in some cases)
 - A CREATE, ALTER, or DROP statement on any schema object
 - A database startup or instance shutdown
 - A specific error message or any error message
 - A user logon or logoff

Trigger Restriction

- A trigger restriction specifies a Boolean expression that must be true for the trigger to fire. The trigger action is not run if the trigger restriction evaluates to false or unknown.

Trigger Action

- A trigger action is the procedure (PL/SQL block, Java program, or C callout) that contains the SQL statements and code to be run when the following events occur:
 - a triggering statement is issued
 - the trigger restriction evaluates to true
- Like stored procedures, a trigger action can:
 - contain SQL, PL/SQL, or Java statements
 - define PL/SQL language constructs such as variables, constants, cursors, exceptions
 - define Java language constructs
 - call stored procedures

Types of Triggers

- **Types of Triggers:**

- Type of Trigger is determined by the triggering event, namely:
 - INSERT
 - UPDATE
 - DELETE
- Triggers can be fired:
 - before or after the operation
 - on row or statement operations
- Trigger can be fired for more than one type of triggering statement

Types of Triggers

Category	Values	Comments
Statement	INSERT, DELETE, UPDATE	Defines which kind of DML statement causes the trigger to fire.
Timing	BEFORE, AFTER	Defines whether the trigger fires before the statement is executed or after the statement is executed.
Level	Row or Statement	<ul style="list-style-type: none">If the trigger is a row-level trigger, it fires once for each row affected by the triggering statement.If the trigger is a statement-level trigger, it fires once, either before or after the statement.A row-level trigger is identified by the FOR EACH ROW clause in the trigger definition.



Copyright © Capgemini 2015. All Rights Reserved 9

Note:

- A trigger can be fired for more than one type of triggering statement. In case of multiple events, OR separates the events.
- From ORACLE 7.1 there is no limit on number of triggers you can write for a table.
 - Earlier you could write maximum of 12 triggers per table, one of each type.

Types of Triggers (contd.)

Row Triggers and Statement Triggers

- When you define a trigger, you can specify the number of times the trigger action has to be run:
 - Once for every row affected by the triggering statement, such as a trigger fired by an UPDATE statement that updates many rows.
 - Once for the triggering statement, no matter how many rows it affects.

Row Triggers

- A Row Trigger is fired each time the table is affected by the triggering statement. For example: If an UPDATE statement updates multiple rows of a table, a row trigger is fired once for each row affected by the UPDATE statement.
- If a triggering statement affects no rows, a row trigger is not run.
- Row triggers are useful if the code in the trigger action depends on data provided by the triggering.

INSTEAD OF Triggers

- INSTEAD OF triggers provide a transparent way of modifying views that cannot be directly modified through DML statements (INSERT, UPDATE, and DELETE). These triggers are called INSTEAD OF triggers because, unlike other types of triggers, Oracle fires the trigger instead of executing the triggering statement.
- You can write normal INSERT, UPDATE, and DELETE statements against the view and the INSTEAD OF trigger is fired to update the underlying tables appropriately. INSTEAD OF triggers are activated for each row of the view that gets modified.

Modify Views

- Modifying views can have ambiguous results:
 - Deleting a row in a view could either mean deleting it from the base table or updating some values so that it is no longer selected by the view.
 - Inserting a row in a view could either mean inserting a new row into the base table or updating an existing row so that it is projected by the view.
 - Updating a column in a view that involves joins might change the semantics of other columns that are not projected by the view.
- Object views present additional problems. For example, a key use of object views is to represent master/detail relationships. This operation inevitably involves joins, but modifying joins is inherently ambiguous.
- As a result of these ambiguities, there are many restrictions on which views are modifiable. An INSTEAD OF trigger can be used on object views as well as relational views that are not otherwise modifiable.

contd.

Types of Triggers (contd.)

Modify Views (contd.)

- A view is inherently modifiable if data can be inserted, updated, or deleted without using INSTEAD OF triggers and if it conforms to the restrictions. Even if the view is inherently modifiable, you might want to perform validations on the values being inserted, updated or deleted. INSTEAD OF triggers can also be used in this case. Here the trigger code performs the validation on the rows being modified and if valid, propagates the changes to the underlying tables, statement or rows that are affected.

Statement Triggers

- A statement trigger is fired once on behalf of the triggering statement, regardless of the number of rows in the table that the triggering statement affects, even if no rows are affected.
For example: If a DELETE statement deletes several rows from a table, a statement-level DELETE trigger is fired only once.
- Statement triggers are useful if the code in the trigger action does not depend on the data provided by the triggering statement or the rows affected.
- For example: Use a statement trigger to:
 - make a complex security check on the current time or user
 - generate a single audit record

BEFORE and AFTER Triggers

- When defining a trigger, you can specify the trigger timing — whether the trigger action has to be run before or after the triggering statement.
- BEFORE and AFTER apply to both statement and row triggers.
- BEFORE and AFTER triggers fired by DML statements can be defined only on tables, not on views. However, triggers on the base tables of a view are fired if an INSERT, UPDATE, or DELETE statement is issued against the view. BEFORE and AFTER triggers fired by DDL statements can be defined only on the database or a schema, and not on particular tables.

Disabling and Dropping Triggers

- To disable a trigger:
- To drop a trigger (by using drop trigger command):

```
ALTER TRIGGER Trigger_Name DISABLE/ENABLE
```

```
DROP TRIGGER Trigger_Name
```



Copyright © Capgemini 2015. All Rights Reserved 12

Note:

An UPDATE or DELETE statement affects multiple rows of a table.

If the trigger is fired once for each affected row, then FOR EACH ROW clause is required. Such a trigger is called a ROW trigger.

If the FOR EACH ROW clause is absent, then the trigger is fired only once for the entire statement. Such triggers are called STATEMENT triggers

Restrictions on Triggers

- **The use of Triggers has the following restrictions:**

- Triggers should not issue transaction control statements (TCL) like COMMIT, SAVEPOINT
- Triggers cannot declare any long or long raw variables
- :new and :old cannot refer to a LONG datatype



Copyright © Capgemini 2015. All Rights Reserved 13

Restrictions on Triggers:

- A Trigger may not issue any transaction control statements (TCL) like COMMIT, SAVEPOINT.
 - Since the triggering statement and the trigger are part of the same transaction, whenever triggering statement is committed or rolled back the work done in the trigger gets committed or rolled back, as well.
- Any procedures or functions that are called by the Trigger cannot have any transaction control statements.
- Trigger body cannot declare any long or long raw variables. Also :new and :old cannot refer to a LONG datatype.
- There are restrictions on the tables that a trigger body can access depending on type of “triggers” and “constraints” on the table.

Order of Trigger Firing

- **Order of Trigger firing is arranged as:**
 - Execute the “before statement level” trigger
 - For each row affected by the triggering statement:
 - Execute the “before row level” trigger
 - Execute the statement
 - Execute the “after row level” trigger
 - Execute the “after statement level” trigger



Copyright © Capgemini 2015. All Rights Reserved 14

Note:

- As each trigger is fired, it will see the changes made by the earlier triggers, as well as the database changes made by the statement.
- The order in which triggers of same type are fired is not defined.
- If the order is important, combine all the operations into one trigger.

Using :Old & :New values in Triggers

Triggering statement	:Old	:New
INSERT	Undefined – all fields are null.	Values that will be inserted when the statement is complete.
UPDATE	Original values for the row before the update.	New values that will be updated when the statement is complete.
DELETE	Original values before the row is deleted.	Undefined – all fields are NULL.

- Note: They are valid only within row level triggers and not in statement level triggers.



Copyright © Capgemini 2015. All Rights Reserved 15

Using :old and :new values in Row Level Triggers:

- Row level trigger fires once per row that is being processed by the triggering statement.
- Inside the trigger, you can access the row that is currently being processed. This is done through keywords :new and :old (they are called as pseudo records). The meaning of the terms is as shown in the slide.

Note:

- The pseudo records are valid only within row level triggers and not in statement level triggers.
 - :old values are not available if the triggering statement is INSERT.
 - :new values are not available if the triggering statement is DELETE.
- Each column is referenced by using the notation :old.Column_Name or :new.Column_Name.
- If a column is not updated by the triggering update statement, then :old and :new values remain the same.

Using WHEN clause

- Use of WHEN clause is valid for row-level triggers only
- Trigger body is executed for rows that meet the specified condition



Copyright © Capgemini 2015. All Rights Reserved 16

Using WHEN Clause:

- The WHEN clause is valid for row-level triggers only. If present, the trigger body will be executed only for those rows that meet the condition specified by the WHEN clause.
- The :new and :old records can be used here without colon.

contd.

Summary

■ In this lesson, you have learnt:

- Database Triggers are procedures written in PL/SQL, Java, or C that run (fire) implicitly:
 - whenever a table or view is modified, or
 - when some user actions or database system actions occur

■ There are three types of triggers:

- Statement based triggers
- Timing based triggers
- Level based triggers



Summary

Summary

- Disabling and Dropping triggers can be done instead of actually removing the triggers
- Order of trigger firing is decided depending on the type of triggers used in the sequence



Summary

Review Question

- **Question 1:** Triggers should not issue Transaction Control Statements (TCL)
 - True / False
- **Question 2:** BEFORE DROP and AFTER DROP triggers are fired when a schema object is dropped
 - True / False
- **Question 3:** The :new and :old records must be used in WHEN clause with a colon
 - True / False



Review Question

- Question 4: A ___ is a table that is currently being modified by a DML statement
- Question 5: A ___ is fired once on behalf of the triggering statement, regardless of the number of rows in the table that the triggering statement affects



Oracle (PL/SQL)

Lesson 8: Locks

Lesson Objectives

- Data Concurrency and Consistency
- Locking in Oracle
- Types of Locks
- DDL & DML Locks



8.1: Data Concurrency and Consistency

Introduction

- Data Concurrency: Ensures users can access data at the same time
- Data Consistency: Ensures each user sees a consistent view of data
 - To ensure data concurrency and consistency means
 - Data must be read and modified in a consistent manner

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 3

Data Concurrency and Consistency

In a single user environment, a user can modify data without concern for other users modifying the same data at the same time. On the other hand, in a multiuser database, various users can update the same data. Concurrency is one of the major concerns in a multiuser environment. When multiple sessions write or read data in a shared environment a database might lose its consistency. Transactions executing simultaneously must produce meaningful and consistent results. Therefore every database provides concurrency control mechanisms.

Most of the databases, the concurrency is managed through various locking mechanisms.

In this topic we see how Oracle ensures Data concurrency and consistency.

Locking Mechanism in Oracle

- Oracle manages control of access to Oracle resources and manages consistency by using Locks
- Locks
 - Used to manage access user defined resources such as tables.
 - Help to maintain transaction consistency in database.
 - Prevents one user from overwriting changes to the database made by another user



Copyright © Capgemini 2015. All Rights Reserved 4

Locking Mechanism in Oracle

Oracle database provides a matured locking mechanism. It follows the rule that reading and writing processes cannot block each other, even if working on the same set of data. Each session receives a read-consistent image of the data. Thus, even if some other process has begun modifying data in the set but did not commit the changes, every subsequent session will be able to read the data just as it was before; once the changes are committed in the first session, every other session is able to see it. The locks are acquired only when the changes are being committed to the database. Oracle automatically selects the least-restrictive lock. User can choose to manually lock a resource (a table, for example). In this case, other users still might be able to access the data, depending on the type of lock deployed. Locks manage access to user defined resources and also maintain transaction consistency in the database. Locks prevents one user from overwriting changes to the database made by another user. If two user processes are executing a series of procedures to update the database with no transaction consistency, there is no guarantee that the data being updated by each user will remain the same for the life of that users transaction. However, locking mechanism provide the ability to perform transaction processing. It allows users to manipulate data freely during the transaction without worry that someone else will change the data before they are done changing it.

Lock Categories

- DML Locks
 - Row Level Locks
 - Table Level Locks
- DDL Locks
 - Exclusive DDL Locks
 - Shared DDL Locks
 - Breakable Parse Locks



Copyright © Capgemini 2015. All Rights Reserved 5

Lock Categories

Locking in Oracle is actually easy, since you really do not have to use explicit locks. Oracle does implicit locking, it is transparent and is efficient in most of the cases. But when you need to understand why a session is blocked, or to have a efficient validation or resolving deadlocks then you require explicit locking to be applied. Locks in Oracle are used for two basic purposes. The first purpose is for Data definition language (DDL) operations which are used for creating and modifying database objects. Locks are acquired for create and alter operations on database objects. The second purpose locks for DML operations which are acquired by user process to make changes to object data. They allow transactions processing to take place. Locks support transaction level read consistency by preventing two transactions from making change to the same data in a table at the same time.

Let us understand DML and DDL locks.

8.2: DML Locks

Types of DML Locks

- DML locks are applied at two levels : Row level and Table Level
- Oracle provides six different types of locks:
 - Exclusive – allows queries on the locked table but prevents any other activity
 - Share – allows concurrent queries but prevents updates to the locked table
 - Share Row Exclusive – allows viewing of whole table but prevents others from locking the table in SHARE mode or updating rows



Copyright © Capgemini 2015. All Rights Reserved

6

The locks in Oracle are acquired on table and so all locks are table locks. However there are two different levels of locking viz. row level and table level. Row level locking is when a table lock allows other table locks to be acquired on a resource while preventing those locks from acquiring certain records or rows within table to make data changes. Table level locking is when other locks are restricted from gaining access to change data in an entire table.

Oracle provide with six different types of locks:

Exclusive : When an exclusive lock is achieved, the lock holder has exclusive access to change the table including contents of the table. Other users can only select data. But no other transaction or DML operation can acquire any type of lock until the exclusive lock is released by the holder.

Share: When one transaction has a share lock on a table, other transactions can also acquire a share, share row or share update lock on the same table. But the other transactions will have to wait until the transaction holding the share lock completes in order to complete their own transactions. If a transaction already holds a share lock on a table then no other transaction can acquire exclusive, row exclusive, or share row exclusive lock on that table.

Share Row Exclusive: A share row exclusive lock held by a transaction allows others to query rows. Other transactions can acquire share-row or share update locks on the table while a transaction holds the share row exclusive lock. Any transaction which tries to perform a DML operation will have to wait until the transaction holding the lock completes.

8.2: DML Locks

Types of DML Locks (contd..)

- Row share – allows concurrent access to the locked table but prevents users from locking the entire table for exclusive access
- Share update – Similar to ROW SHARE, and is included for backward compatibility
- Row exclusive – Similar to ROW SHARE but also prevents locking in SHARE mode. These locks are automatically when updating, inserting or deleting



Copyright © Capgemini 2015. All Rights Reserved

7

Row Share: A row share lock held by a transaction allows others to query any rows or insert new rows. Also users can update or delete rows other than the ones on which lock is being held. As a result two transactions can make changes to different rows in the same table at the same time, using row share, row exclusive, share update and share row exclusive lock. A transaction cannot acquire an exclusive lock on a table if another transaction has already acquired the row share lock on that table.

Share Update: A share update lock is acquired for making changes to data in table rows. When a transaction holds this lock, any other transaction can acquire any other type of lock on a table except for a exclusive lock. This lock is similar in behavior like the Row share lock

Row Exclusive: A row-exclusive lock held by a transaction allows other transactions to query any rows or insert new rows on the table while the row-exclusive lock is being held. In addition, transactions can concurrently process update or delete statements on rows other than those held under the row-exclusive lock in the same table. Therefore, row-exclusive locks allow multiple transactions to obtain simultaneous row-exclusive, share-row, or share-update locks for different rows in the same table. However, while one transaction holds a row-exclusive lock, no other transaction can make changes to rows that the first transaction has changed until the first transaction completes. Additionally, no transaction may obtain an exclusive, share, or share-row-exclusive lock on a table while another transaction holds a row-exclusive lock on that same table.

Acquiring Locks in Oracle

- Syntax

```
LOCK TABLE tablename IN lockmode MODE NOWAIT;
```

- Example

```
LOCK TABLE staff_master IN SHARE UPDATE mode nowait;
```

```
LOCK TABLE student_master IN EXCLUSIVE mode;
```



Copyright © Capgemini 2015. All Rights Reserved 8

Add the notes here.

Deadlock

- Deadlock occurs when two or more sessions are waiting for data locked by each other, resulting in all the sessions being blocked.
- Oracle automatically deals with deadlocks by raising an ORA-00060 exception in one of the sessions.



Copyright © Capgemini 2015. All Rights Reserved 9

A deadlock is the situation where you have two, or more, Oracle "sessions" competing for mutually locked resources. In other words, it is a condition where two or more users are waiting for data locked by each other. Oracle deals with deadlocks automatically raising an exception in one of the sessions.

There are a few reasons why your application may experience deadlocks, most of which are about application design or by poorly implemented locking in application code. However, there are a few situations when, due to certain architectural design decisions, you may experience deadlocks simply due to the internal mechanisms of Oracle itself.

Deadlock

Example

- If session 1 is locking row 1, session 2 locks row 2, then session 1 attempts to lock row 2 which will block since session 2 has the lock on that row, and then session 2 attempts to lock row 1 which will also block since session 1 has the lock on that row, then session 1 is waiting for session 2, and session 2 is waiting on session 1, which of course will never be resolved



Copyright © Capgemini 2015. All Rights Reserved 10

Considering the example given on slide it looks like an infinite loop. This is where Oracle steps in automatically detecting the situation and resolves it by raising an exception for one of the sessions. But what happens to other session. The session continues to wait. The client getting the exception should either commit or rollback to release the locks and only then the operations for both the sessions can go ahead.

8.2: DDL Locks

Types of DDL Locks

- Oracle provides three types of DDL locks
 - Exclusive
 - Shared
 - Breakable Parse



Copyright © Capgemini 2015. All Rights Reserved 11

A DDL Lock protects the definition of a schema object while the object is referenced in a DDL statement. Oracle automatically acquires a DDL lock to prevent other DDL operation from referencing or altering the same object. In case a DDL lock is requested on an object that already has a DDL lock on it, the next lock request will wait. Users cannot explicitly request DDL locks.

Oracle provides three types of DDL locks

Exclusive: This lock prevents other session from obtaining a DDL or DML lock. Most DDL operation require exclusive DDL lock to prevent destructive interference with other DDL operations that might modify or reference the same schema object. For example, a DROP TABLE operation will not be possible while an ALTER TABLE operation is modifying the structure of the same table. A query against the table is not blocked. These locks last for the duration of DDL statement execution.

Shared: This lock prevents destructive interference with conflicting DDL operations, but allows data concurrency for similar DDL operations.

For example, when a CREATE PROCEDURE statement is run, the containing transaction acquires share DDL locks for all referenced tables. You could concurrently create procedures that reference the same tables and acquire shared DDL locks. But no transaction can acquire an exclusive DDL lock on any referenced table. This lock lasts for the duration of DDL statement execution.

Breakable Parse Locks: This lock is held by a SQL statement or PL/SQL program unit for each schema object that it references. Parse locks are acquired so that the associated shared SQL area can be invalidated if a referenced object is altered or dropped. This lock is called as breakable parse lock because it does allow any DDL operation and can be broken to allow conflicting DDL operation. This lock is acquired in the shared pool during the parse phase of a SQL statement.

Summary

- Data Concurrency and Consistency
- Locking in Oracle
- Types of Locks
- DDL & DML Locks



Review Question

■ Question 1: Which of the following is a DML lock?

- Option 1: Share Update
- Option 2: Row update
- Option 3: Row Exclusive

■ Question 2 : DDL locks are acquired automatically and cannot be acquired explicitly.

- True/ False



Oracle (PL/SQL)

Lesson 9: Built-in Packages in Oracle

Lesson Objectives

- To understand the following topics:
 - Testing and Debugging in PL/SQL
 - DBMS_OUTPUT
 - Enabling and Disabling output
 - Writing to the DBMS_OUTPUT Buffer
 - UTL_file
 - Handling LOB (Large Objects)



9.1: Testing and Debugging in PL/SQL

DBMS_OUTPUT package

- PL/SQL has no input/output capability
- However, built-in package DBMS_OUTPUT is provided to generate reports
- The procedure PUT_LINE is also provided that places the contents in the buffer

```
PUT_LINE (VARCHAR2 OR NUMBER OR DATE)
```



Copyright © Capgemini 2015. All Rights Reserved 3

DBMS_OUTPUT package

```
SQL>SET SERVEROUTPUT ON
DECLARE
  V_Variable VARCHAR2(25) :=' Used for'
  || 'Debugging ';
BEGIN
  DBMS_OUTPUT.PUT_LINE(V_Variable);
END;
```



Copyright © Capgemini 2015. All Rights Reserved 4

The code will be written on SQL prompt

9.2: DBMS_OUTPUT

Displaying Output

- DBMS_OUTPUT:

- DBMS_OUTPUT provides a mechanism for displaying information from the PL/SQL program on to your screen (that is your session's output device)
- The DBMS_OUTPUT package is created when the Oracle database is installed
 - The "dbmsoutp.sql" script contains the source code for the specification of this package
 - This script is called by the "catproc.sql" script, which is normally run immediately after database creation



Copyright © Capgemini 2015. All Rights Reserved 5

Note:

The catproc.sql script creates the public synonym DBMS_OUTPUT for the package.

Instance-wise access to this package is provided on installation, so no additional steps should be necessary in order to use DBMS_OUTPUT

9.2: Displaying Output

DBMS_OUTPUT – Program Names

Table: DBMS_OUTPUT programs

Name	Description	Use in SQL?
DISABLE	Disables output from the package; the DBMS_OUTPUT buffer will not be flushed to the screen.	Yes
ENABLE	Enables output from the package.	Yes
GET_LINE	Gets a single line from the buffer.	Yes
GET_LINES	Gets specified number of lines from the buffer and passes them into a PL/SQL table.	Yes
NEW_LINE	Inserts an end-of-line mark in the buffer.	Yes
PUT	Puts information into the buffer.	Yes
PUT_LINE	Puts information into the buffer and appends an end-of-line marker after that data.	Yes



Copyright © Capgemini 2015. All Rights Reserved 6

DBMS_OUTPUT Concepts:

- Each user has a DBMS_OUTPUT buffer of up to 1,000,000 bytes in size. You can write information to this buffer by calling the DBMS_OUTPUT.PUT and DBMS_OUTPUT.PUT_LINE programs.
 - If you are using DBMS_OUTPUT from within SQL*Plus, this information will be automatically displayed when your program terminates.
 - You can (optionally) explicitly retrieve information from the buffer with calls to DBMS_OUTPUT.GET and DBMS_OUTPUT.GET_LINE.
- The DBMS_OUTPUT buffer can be set to a size between 2,000 and 1,000,000 bytes with the DBMS_OUTPUT.ENABLE procedure.
 - If you do not enable the package, no information will be displayed or be retrievable from the buffer.
- The buffer stores three different types of data in their internal representations, namely VARCHAR2, NUMBER, and DATE.
 - These types match the overloading available with the PUT and PUT_LINE procedures.
 - Note that DBMS_OUTPUT does not support Boolean data in either its buffer or its overloading of the PUT procedures

DBMS_OUTPUT - Example

- In this example, the following anonymous PL/SQL block uses DBMS_OUTPUT to display the name and salary of each staff member in department 10:

```
DECLARE
  CURSOR emp_cur IS SELECT staff_name, staff_sal
    FROM staff_master WHERE dept_code = 10
    ORDER BY staff_sal DESC;
BEGIN FOR emp_rec IN emp_cur
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Employee ' ||
      emp_rec.staff_name || ' earns ' ||
      TO_CHAR (emp_rec.staff_sal) || 'rupees');
  END LOOP;
END;
```



Copyright © Capgemini 2015. All Rights Reserved 7

DBMS_OUTPUT Concepts:

The program shown in the slide generates the following output when executed in SQL*Plus:

Employee John earns 32000 rupees

Employee Mohan earns 24000 rupees

DBMS_OUTPUT Exceptions:

- DBMS_OUTPUT does not contain any declared exceptions. Instead, Oracle designed the package to rely on two error numbers in the -20 NNN range (usually reserved for Oracle customers). You may, therefore, encounter one of these two exceptions when using the DBMS_OUTPUT package (no names are associated with these exceptions).
 - The -20000 error number indicates that these package-specific exceptions were raised by a call to RAISE_APPLICATION_ERROR, which is in the DBMS_STANDARD package.
- -20000
 - ORU-10027: buffer overflow, limit of <buf_limit> bytes.
 - If you receive the -10027 error, you should see if you can increase the size of your buffer with another call to DBMS_OUTPUT.ENABLE.
- -20000
 - ORU-10028: line length overflow, limit of 255 bytes per line.
 - If you receive the -10028 error, you should restrict the amount of data you are passing to the buffer in a single call to PUT_LINE, or in a batch of calls to PUT followed by NEW_LINE.
- You may also receive the ORA-06502 error:
 - ORA-06502
 - It is a numeric or value error.
 - If you receive the -06502 error, you have tried to pass more than 255 bytes of data to DBMS_OUTPUT.PUT_LINE. You must break up the line into more than one string.

Drawbacks of DBMS_OUTPUT:

- Before learning all about the DBMS_OUTPUT package, and rushing to use it, you should be aware of several drawbacks with the implementation of this functionality:
 - The "put" procedures that place information in the buffer are overloaded only for strings, dates, and numbers. You cannot request the display of Booleans or any other types of data. You cannot display combinations of data (a string and a number, for instance), without performing the conversions and concatenations yourself.
 - You will see output from this package only after your program completes its execution. You cannot use DBMS_OUTPUT to examine the results of a program while it is running. And if your program terminates with an unhandled exception, you may not see anything at all!
 - If you try to display strings longer than 255 bytes, DBMS_OUTPUT will raise a VALUE_ERROR exception.
 - DBMS_OUTPUT is not a strong choice as a report generator, because it can handle a maximum of only 1,000,000 bytes of data in a session before it raises an exception.

9.2: Displaying Output

Writing to DBMS_OUTPUT buffer

- You can write information to the DBMS_OUTPUT buffer with calls to the PUT, NEW_LINE, and PUT_LINE procedures.



Copyright © Capgemini 2015. All Rights Reserved 9

If you use DBMS_OUTPUT in SQL*Plus, you may find that any leading blanks are automatically truncated. Also, attempts to display blank or NULL lines are completely ignored.

There are workarounds for almost every one of these drawbacks. The solution invariably requires the construction of a package that encapsulates and hides DBMS_OUTPUT.

Writing to DBMS_OUTPUT buffer

- The DBMS_OUTPUT.PUT procedure:
- The PUT procedure puts information into the buffer, but does not append a newline marker into the buffer.
 - Use PUT if you want to place information in the buffer (usually with more than one call to PUT), but not also automatically issue a newline marker.
 - The specification for PUT is overloaded, so that you can pass data in its native format to the package without having to perform conversions.

```
PROCEDURE DBMS_OUTPUT.PUT (A VARCHAR2);
```



Copyright © Capgemini 2013. All Rights Reserved. 10

Writing to DBMS_OUTPUT buffer: DBMS_OUTPUT.PUT procedure:

Note:

- The specification for PUT is overloaded, so that you can pass data in its native format to the package without having to perform conversions.

```
PROCEDURE DBMS_OUTPUT.PUT (A  
VARCHAR2);  
PROCEDURE DBMS_OUTPUT.PUT (A NUMBER);  
PROCEDURE DBMS_OUTPUT.PUT (A DATE);
```

where A is the data being passed.

Example:

- In the following example, three simultaneous calls to PUT, place the employee name, department ID number, and hire date into a single line in the DBMS_OUTPUT buffer:

```
DBMS_OUTPUT.PUT (:employee.lname || ',' ||  
:employee.fname);  
DBMS_OUTPUT.PUT (:employee.department_id);  
DBMS_OUTPUT.PUT (:employee.hiredate);
```

- If you follow these PUT calls with a NEW_LINE call, that information can then be retrieved with a single call to GET_LINE

Writing to DBMS_OUTPUT buffer

- The DBMS_OUTPUT.NEW_LINE procedure:
- The NEW_LINE procedure inserts an end-of-line marker in the buffer.
- Use NEW_LINE after one or more calls to PUT in order to terminate those entries in the buffer with a newline marker.
- Given below is the specification for NEW_LINE:

```
PROCEDURE DBMS_OUTPUT.NEW_LINE;
```



Copyright © Capgemini 2015. All Rights Reserved 11

Writing to DBMS_OUTPUT buffer

- The DBMS_OUTPUT.PUT_LINE procedure:

- The PUT_LINE procedure puts information into the buffer, and then appends a newline marker into the buffer
- The specification for PUT_LINE is overloaded, so that you can pass data in its native format to the package without having to perform conversions:

```
PROCEDURE DBMS_OUTPUT.NEW_LINE(A VARCHAR2);
```



Copyright © Capgemini 2013. All Rights Reserved. 12

Writing to DBMS_OUTPUT buffer: DBMS_OUTPUT.PUT_LINE procedure:

Note:

- The specification for PUT_LINE is overloaded, so that you can pass data in its native format to the package without having to perform conversions:

```
PROCEDURE DBMS_OUTPUT.PUT_LINE (A VARCHAR2);
PROCEDURE DBMS_OUTPUT.PUT_LINE (A NUMBER);
PROCEDURE DBMS_OUTPUT.PUT_LINE (A DATE);
```

where A is the data being passed

- The PUT_LINE procedure is the one most commonly used in SQL*Plus to debug PL/SQL programs.
- When you use PUT_LINE in these situations, you do not need to call GET_LINE to extract the information from the buffer. Instead, SQL*Plus will automatically dump out the DBMS_OUTPUT buffer when your PL/SQL block finishes executing. (You will not see any output until the program ends.)

Writing to DBMS_OUTPUT buffer:**DBMS_OUTPUT.PUT_LINE (contd.):****Example:**

- Suppose that you execute the following three statements in SQL*Plus:

```
SQL> exec DBMS_OUTPUT.PUT ('I am');
SQL> exec DBMS_OUTPUT.PUT (' writing');
SQL> exec DBMS_OUTPUT.PUT ('a');
```

- You will not see anything, because PUT will place the information in the buffer, but will not append the newline marker. Now suppose you issue this next PUT_LINE command, namely:
 - SQL> exec DBMS_OUTPUT.PUT_LINE ('book!');
- Then you will see the following output:
 - I am writing a book!
- All of the information added to the buffer with the calls to PUT, patiently wait to be flushed out with the call to PUT_LINE. This is the behavior you will see when you execute individual calls at the SQL*Plus command prompt to the PUT programs.
- Suppose you place these same commands in a PL/SQL block, namely:

```
BEGIN
    DBMS_OUTPUT.PUT ('I am');
    DBMS_OUTPUT.PUT (' writing ');
    DBMS_OUTPUT.PUT ('a ');
    DBMS_OUTPUT.PUT_LINE ('book');
END;
/
```

- Then the output from this script will be exactly the same as that generated by this single call:
 - SQL> exec DBMS_OUTPUT.PUT_LINE ('I am writing a book!');

9.2: Displaying Output

Retrieving Data from DBMS_OUTPUT buffer

- You can retrieve information from the DBMS_OUTPUT buffer with call to the GET_LINE procedure.
 - The DBMS_OUTPUT.GET_LINE procedure:
 - The GET_LINE procedure retrieves one line of information from the buffer
 - Given below is the specification for the procedure:

```
PROCEDURE DBMS_OUTPUT.GET_LINE (line OUT VARCHAR2,  
status OUT INTEGER);
```



Copyright © Capgemini 2015. All Rights Reserved 14

Retrieving Data from the DBMS_OUTPUT buffer:

- You can use the GET_LINE and GET_LINES procedures to extract information from the DBMS_OUTPUT buffer.
- If you are using DBMS_OUTPUT from within SQL*Plus, however, you will never need to call either of these procedures. Instead, SQL*Plus will automatically extract the information and display it on the screen for you.

The DBMS_OUTPUT.GET_LINE procedure:

- The GET_LINE procedure retrieves one line of information from the buffer.
- Given below is the specification for the procedure:

```
PROCEDURE DBMS_OUTPUT.GET_LINE  
(line OUT VARCHAR2,  
status OUT INTEGER);R
```

- The parameters are summarized as shown below:

Parameter	Description
Line	Retrieved line of text
Status	GET request status

contd.

Retrieving Data from the DBMS_OUTPUT buffer (contd.):**The DBMS_OUTPUT.GET_LINE procedure (contd.)**

- The line can have up to 255 bytes in it, which is not very long. If GET_LINE completes successfully, then status is set to 0. Otherwise, GET_LINE returns a status of 1.
- Note that even though the PUT and PUT_LINE procedures allow you to place information into the buffer in their native representations (dates as dates, numbers and numbers, and so forth), GET_LINE always retrieves the information into a character string. The information returned by GET_LINE is everything in the buffer up to the next newline character. This information might be the data from a single PUT_LINE or from multiple calls to PUT.

• For example:

The following call to GET_LINE extracts the next line of information into a local PL/SQL variable:

```
FUNCTION get_next_line RETURN
VARCHAR2
IS
    return_value VARCHAR2(255);
    get_status INTEGER;
BEGIN
    DBMS_OUTPUT.GET_LINE
    (return_value, get_status);
    IF get_status = 0
    THEN
        RETURN return_value;
    ELSE
        RETURN NULL;
    END IF;
END;
```

UTL-FILE

- **UTL_FILE package:**

- The UTL_FILE package is created when the Oracle database is installed.
- The "utlfile.sql script" (found in the built-in packages source code directory) contains the source code for the specification of this package.
 - This script is called by catproc.sql, which is normally run immediately after database creation
 - The script creates the public synonym UTL_FILE for the package, and grants EXECUTE privilege on the package to public



Copyright © Capgemini 2015. All Rights Reserved 16

UTL_FILE: READING AND WRITING

- UTL_FILE is a package that has been welcomed warmly by PL/SQL developers. It allows PL/SQL programs to both “read from” and “write to” any operating system files that are accessible from the server on which your database instance is running.
 - You can now read ini files and interact with the operating system a little more easily than has been possible in the past.
 - You can directly load data from files into database tables while applying the full power and flexibility of PL/SQL programming.
 - You can directly generate reports from within PL/SQL without worrying about the maximum buffer restrictions of DBMS_OUTPUT.
- The UTL_FILE package is created when the Oracle database is installed. The utlfile.sql script (found in the built-in packages source code directory) contains the source code for the specification of this package. This script is called by catproc.sql, which is normally run immediately after database creation. The script creates the public synonym UTL_FILE for the package and grants EXECUTE privilege on the package to public.

9.3: Filenames

UTL_FILE

Table: UTL_FILE programs

Name	Description	Use in SQL?
FCLOSE	Closes the specified files.	NO
FCLOSE_ALL	Closes all open files.	NO
FFLUSH	Flushes all the data from the UTL_FILE buffer.	NO
FOPEN	Opens the specified file.	NO
GET_LINE	Gets the next line from the file.	NO
IS_OPEN	Returns TRUE if the file is already open.	NO
NEW_LINE	Inserts a newline mark in the file at the end of the current line.	NO
PUT	Puts text into the buffer.	NO
PUT_LINE	Puts a line of text into the file.	NO
PUTF	Puts formatted text into the buffer.	NO



Copyright © Capgemini 2013. All Rights Reserved. 17

File security:

- UTL_FILE lets you read and write files accessible from the server on which your database is running. So theoretically you can use UTL_FILE to write right over your tablespace data files, control files, and so on. That is of course not a very good idea.
- Server security requires the ability to place restrictions on where you can read and write your files.
- UTL_FILE implements this security by limiting access to files that reside in one of the directories specified in the INIT.ORA file for the database instance on which UTL_FILE is running.
- When you call FOPEN to open a file, you must specify both the location and the name of the file, in separate arguments. This file location is then checked against the list of accessible directories.
- Given below is the format of the parameter for file access in the INIT.ORA file:
 - utl_file_dir = <directory>
- Include a parameter for utl_file_dir for each directory you want to make accessible for UTL_FILE operations. For example: The following entries enable four different directories in UNIX:
 - utl_file_dir = /tmp
 - utl_file_dir = /ora_apps/hr/time_reporting
 - utl_file_dir = /ora_apps/hr/time_reporting/log
 - utl_file_dir = /users/test_area
- To bypass server security and allow read/write access to all directories, you can use the special syntax given below:
 - utl_file_dir = *

Specifying file locations:

- The location of the file is an operating system-specific string that specifies the directory or area in which to open the file. The location you provide must have been listed as an accessible directory in the INIT.ORA file for the database instance.
- The INIT.ORA location is a valid directory or area specification, as shown in the following examples:
 - In Windows NT:
'k:\common\debug'
 - In UNIX:
'/usr/od2000/admin'
- Few examples are given below:
 - In Windows NT:

```
file_id := UTL_FILE.FOPEN ('k:\common\debug',
                           'trace.lis', 'R');
```
 - In UNIX:

```
file_id := UTL_FILE.FOPEN ('/usr/od2000/admin',
                           'trace.lis', 'W');
```
- Your location must be an explicit, complete path to the file. You cannot use operating system-specific parameters such as environment variables in UNIX to specify file locations.

UTL_FILE exceptions:

- The package specification of UTL_FILE defines seven exceptions. The cause behind a UTL_FILE exception can often be difficult to understand.
- Given below are the explanations Oracle provides for each of the exceptions:
 - **INVALID_PATH**
The file location or the filename is invalid. Perhaps the directory is not listed as a utl_file_dir parameter in the INIT.ORA file (or doesn't exist as all), or you are trying to read a file and it does not exist.
 - **INVALID_MODE**
The value you provided for the open_mode parameter in UTL_FILE.FOPEN was invalid. It must be "A", "R", or "W".
 - **INVALID_FILEHANDLE**
The file handle you passed to a UTL_FILE program was invalid. You must call UTL_FILE.FOPEN to obtain a valid file handle.
 - **INVALID_OPERATION**
UTL_FILE could not open or operate on the file as requested. For example, if you try to write to a read-only file, you will raise this exception.
 - **READ_ERROR**
The operating system returned an error when you tried to read from the file. (This does not occur very often.)
 - **WRITE_ERROR**
The operating system returned an error when you tried to write to the file. (This does not occur very often.)
 - **INTERNAL_ERROR**
Uh-oh. Something went wrong and the PL/SQL runtime engine couldn't assign blame to any of the previous exceptions. Better call Oracle Support!
- Programs in UTL_FILE may also raise the following standard system exceptions:
 - **NO_DATA_FOUND**
It is raised when you read past the end of the file with UTL_FILE.GET_LINE.
 - **VALUE_ERROR**
It is raised when you try to read or write lines in the file which are too long.
 - **INVALID_MAXLINESIZE**
Oracle 8.0 and above: It is raised when you try to open a file with a maximum linesize outside of the valid range (between 1 through 32767).

9.3: UTL_FILE

Process Flow

- **UTL_FILE process flow:**

- The following sections describe each of the UTL_FILE programs, following the process flow for working with files.
- The flow is described for both writing and reading files.



Copyright © Capgemini 2015. All Rights Reserved 20

Add the notes here.

UTL_FILE: Process Flow

- Write to a file: In order to “write to a file”, you will (in most cases) perform the following steps:
 - Declare a file handle. This handle serves as a pointer to the file for subsequent calls to programs in the UTL_FILE package to manipulate the contents of this file
 - Open the file with a call to FOPEN, which returns a file handle to the file. You can open a file to read, replace, or append text
 - Write data to the file by using the PUT, PUTF, or PUT_LINE procedures
 - Close the file with a call to FCLOSE. This releases resources associated with the file



Copyright © Capgemini 2015. All Rights Reserved 21

Insertion Sort:

In Insertion Sort, if we have an array of “n” elements, then we can say that the first element is sorted, and till now we have read only 1st element.

Read the 2nd element, and find it's position in the array which is already sorted. Till now only 1st element is sorted. So check whether to add it before or after 1st element.

Accordingly, shift all elements on its right by one location on right, and then insert the element at the appropriate location.

Hence this method is appropriately called Insertion Sort.

UTL_FILE: Process Flow

- Read from a file: In order to “read data from a file”, you will (in most cases) perform the following steps:
 - Declare a file handle
 - Declare a VARCHAR2 string buffer that will receive the line of data from the file. You can also read directly from a file into a numeric or date buffer. In this case, the data in the file will be converted implicitly, and so it must be compatible with the datatype of the buffer
 - Open the file using FOPEN in read mode

contd...



Copyright © Capgemini 2015. All Rights Reserved 22

Insertion Sort:

In Insertion Sort, if we have an array of “n” elements, then we can say that the first element is sorted, and till now we have read only 1st element.

Read the 2nd element, and find its position in the array which is already sorted. Till now only 1st element is sorted. So check whether to add it before or after 1st element.

Accordingly, shift all elements on its right by one location on right, and then insert the element at the appropriate location.

Hence this method is appropriately called Insertion Sort.

UTL_FILE: Process Flow

- Use the GET_LINE procedure to read data from the file and into the buffer. To read all the lines from a file, you would execute GET_LINE in a loop
- Close the file with a call to FCLOSE



Copyright © Capgemini 2015. All Rights Reserved 23

Insertion Sort:

In Insertion Sort, if we have an array of “n” elements, then we can say that the first element is sorted, and till now we have read only 1st element.

Read the 2nd element, and find its position in the array which is already sorted. Till now only 1st element is sorted. So check whether to add it before or after 1st element.

Accordingly, shift all elements on its right by one location on right, and then insert the element at the appropriate location.

Hence this method is appropriately called Insertion Sort.

9.3: UTL_FILE

Opening Files

- You can use the FOPEN and IS_OPEN functions when you open files via UTL_FILE
- Note:
 - Using the UTL-FILE package, you can only open a maximum of ten files for each Oracle session



Copyright © Capgemini 2015. All Rights Reserved 24

Insertion Sort:

In Insertion Sort, if we have an array of “n” elements, then we can say that the first element is sorted, and till now we have read only 1st element.

Read the 2nd element, and find it's position in the array which is already sorted. Till now only 1st element is sorted. So check whether to add it before or after 1st element.

Accordingly, shift all elements on its right by one location on right, and then insert the element at the appropriate location.

Hence this method is appropriately called Insertion Sort.

9.3: UTL_FILE

The UTL_FILE.FOPEN Function

- UTL_FILE.FOPEN function:
 - The FOPEN function opens the specified file and returns a file handle that you can then use to manipulate the file.
 - The header for the function is:

```
FUNCTION UTL_FILE.FOPEN (FUNCTION UTL_FILE.FOPEN  
    location IN VARCHAR2, location IN VARCHAR2, filename IN  
    VARCHAR2, filename IN VARCHAR2, open_mode IN  
    VARCHAR2) open_mode IN VARCHAR2, RETURN file_type;  
    max_linesize IN BINARY_INTEGER) RETURN file_type;
```



Copyright © Capgemini 2015. All Rights Reserved 25

UTL_FILE.FOPEN Function:

Parameters for the function shown in the slide are summarized in the following table.

Parameter	Description
location	Location of the file
filename	Name of the file
openmode	Mode in which the file has to be opened
max_linesize	The maximum number of characters per line, including the newline character, for this file. Minimum is 1, maximum is 32767.

contd.

UTL_FILE.FOPEN Function (contd.):

- You can open the file in one of the following three modes:
 - **R mode**
Open the file read-only. If you use this mode, use UTL_FILE's GET_LINE procedure to read from the file.
 - **W mode**
Open the file to read and write in replace mode. When you open in replace mode, all existing lines in the file are removed. If you use this mode, then you can use any of the following UTL_FILE programs to modify the file: PUT, PUT_LINE, NEW_LINE, PUTF, and FFLUSH.
 - **A mode**
Open the file to read and write in append mode. When you open in append mode, all existing lines in the file are kept intact. New lines will be appended after the last line in the file. If you use this mode, then you can use any of the following UTL_FILE programs to modify the file: PUT, PUT_LINE, NEW_LINE, PUTF, and fFFLUSH.
- Example
 - The following example shows how to declare a file handle, and then open a configuration file for that handle in read-only mode:

```
DECLARE
    config_file UTL_FILE.FILE_TYPE;
BEGIN
    config_file := UTL_FILE.FOPEN
    ('/maint/admin', 'config.txt', 'R');
```

9.3: UTL_FILE

The UTL_FILE.IS_OPEN Function

- **UTL_FILE.IS_OPEN function:**

- The IS_OPEN function returns TRUE if the specified handle points to a file that is already open. Otherwise, it returns FALSE.
- The header for the function is:
- where file is the file to be checked

```
FUNCTION UTL_FILE.IS_OPEN (file IN UTL_FILE.FILE_TYPE)
RETURN BOOLEAN;
```



Copyright © Capgemini 2013. All Rights Reserved. 27

UTL_FILE.FOPEN Function:

- Parameters for the function shown in the slide are summarized in the following table.

contd.

9.3: UTL_FILE

Reading from Files

- UTL_FILE provides only one program to retrieve data from a file, namely the GET_LINE procedure



Copyright © Capgemini 2015. All Rights Reserved 28

UTL_FILE.FOPEN Function:

Parameters for the function shown in the slide are summarized in the following table.

contd.

9.3: UTL_FILE

The UTL_FILE.GET_LINE Procedure

- **UTL_FILE.GET_LINE procedure:**

- The GET_LINE procedure reads a line of data from the specified file, if it is open, into the provided line buffer.
- The header for the procedure is:

```
PROCEDURE UTL_FILE.GET_LINE (file IN UTL_FILE.FILE_TYPE,  
buffer OUT VARCHAR2);
```



Copyright © Capgemini 2015. All Rights Reserved 29

UTL_FILE.GET_LINE procedure:

- The GET_LINE procedure reads a line of data from the specified file, if it is open, into the provided line buffer.
Given below is the header for the procedure:

```
PROCEDURE UTL_FILE.GET_LINE  
(file IN UTL_FILE.FILE_TYPE,  
buffer OUT VARCHAR2);
```

- Parameters are summarized in the following table:

Parameter	Description
File	The file handle returned by a call to FOPEN.
Buffer	The buffer into which the line of data is read.

contd.

UTL_FILE.GET_LINE procedure (contd.):

- The variable specified for the buffer parameter must be large enough to hold all the data up to the next carriage return or end-of-file condition in the file. If not, PL/SQL will raise the VALUE_ERROR exception. The line terminator character is not included in the string passed into the buffer.
- For example:
Since GET_LINE reads data only into a string variable, you will have to perform your own conversions to local variables of the appropriate datatype if your file holds numbers or dates. Of course, you can call this procedure and directly read data into string and numeric variables, as well. In this case, PL/SQL will be performing a runtime, implicit conversion for you. In many situations, this is fine.
- It is generally recommended that you avoid implicit conversions and instead perform your own conversion. This approach more clearly documents the steps and dependencies.
- Here is an example:

```
DECLARE
    fileID UTL_FILE.FILE_TYPE;
    strbuffer VARCHAR2(100);
    mynum NUMBER;
BEGIN
    fileID := UTL_FILE.FOPEN ('/tmp', 'numlist.txt',
    'R');
    UTL_FILE.GET_LINE (fileID, strbuffer);
    mynum := TO_NUMBER (strbuffer);
END;
/
```

- When GET_LINE attempts to read past the end of the file, the NO_DATA_FOUND exception is raised. This is the same exception that is raised when you:
 - a) execute an implicit (SELECT INTO) cursor that returns no rows, or
 - b) reference an undefined row of a PL/SQL (nested in PL/SQL8) table.
- If you are performing more than one of these operations in the same PL/SQL block, remember that this same exception can be caused by very different parts of your program.

9.3: UTL_FILE

Writing to Files

- In contrast to the simplicity of reading from a file, UTL_FILE offers a number of different procedures you can use to write to a file:
 - UTL_FILE.PUT procedure
 - Puts a piece of data (string, number, or date) into a file in the current line
 - UTL_FILE.NEW_LINE procedure
 - Puts a newline or line termination character into the file at the current position



Copyright © Capgemini 2013. All Rights Reserved. 31

Writing to Files: UTL_FILE.PUT procedure

- The PUT procedure puts data out to the specified open file.
- Given below is the header for this procedure:

```
PROCEDURE UTL_FILE.PUT
  (file IN UTL_FILE.FILE_TYPE,
   buffer IN VARCHAR2); -----OUT
  replace with IN
```

- Parameters are summarized in the following table.

Parameter	Description
File	The file handle returned by a call to FOPEN.
Buffer	The buffer containing the text to be written to the file; maximum size allowed is 32K for Oracle 8.0.3 and above; for earlier versions, it is 1023 bytes.

- The PUT procedure adds the data to the current line in the opened file, but does not append a line terminator. You must use the NEW_LINE procedure to terminate the current line or use PUT_LINE to write out a complete line with a line termination character.

Writing to Files

- **UTL_FILE.PUT_LINE** procedure
 - Puts a string into a file, followed by a platform-specific line termination character
- **UTL_FILE.PUTF** procedure
 - Puts up to five strings out to the file in a format based on a template string, similar to the printf function in C



Copyright © Capgemini 2015. All Rights Reserved 32

Exceptions:

PUT may raise any of the following exceptions:

UTL_FILE.INVALID_FILEHANDLE
UTL_FILE.INVALID_OPERATION
UTL_FILE.WRITE_ERROR

Note:

Besides the four procedures mentioned in the above slides there is one more procedure called UTL_FILE.FFLUSH procedure.

UTL_FILE.FFLUSH procedure makes sure that all pending data for the specified file is written physically out to a file.
The header for FFLUSH is,

```
PROCEDURE UTL_FILE.FFLUSH (file IN  
UTL_FILE.FILE_TYPE);
```

where: file is the file handle.

One of the four procedures, namely UTL_FILE.PUT_LINE procedure, is discussed in detail in the following slide.

9.3: UTL_FILE

The UTL_FILE.PUT_LINE Procedure

- **UTL_FILE.PUT_LINE procedure:**

- This procedure writes data to a file, and then immediately appends a newline character after the text
- The header for PUT_LINE is:

```
PROCEDURE UTL_FILE.PUT_LINE (file IN UTL_FILE.FILE_TYPE,  
buffer IN VARCHAR2);
```



Copyright © Capgemini 2013. All Rights Reserved. 33

UTL_FILE.PUT_LINE procedure:

- This procedure writes data to a file, and then immediately appends a newline character after the text. Given below is the header for PUT_LINE:

```
PROCEDURE UTL_FILE.PUT_LINE  
(file IN UTL_FILE.FILE_TYPE,  
buffer IN VARCHAR2);
```

- Parameters are summarized in the following table.

Parameter	Description
File	The file handle returned by a call to FOPEN.
Buffer	Text to be written to the file; maximum size allowed is 32K for Oracle 8.0.3 and above; for earlier versions, it is 1023 bytes.

- Before you can call UTL_FILE.PUT_LINE, you must have already opened the file.

```
PROCEDURE emp2file
IS
    fileID UTL_FILE.FILE_TYPE;
BEGIN
    fileID := UTL_FILE.FOPEN ('/tmp', 'emp.dat', 'W');

    /* Quick and dirty construction here! */
    FOR emprec IN (SELECT * FROM emp)
    LOOP
        UTL_FILE.PUT_LINE
            (TO_CHAR (emprec.empno) || ',' ||
             emprec.ename || ',' ||
             ...
             TO_CHAR (emprec.deptno));
    END LOOP;

    UTL_FILE.FCLOSE (fileID);
END;
```

9.3: UTL_FILE

Closing Files

- You can use the FCLOSE and FCLOSE_ALL procedures for closing files



Copyright © Capgemini 2015. All Rights Reserved 35

9.3: UTL_FILE

The UTL_FILE.FCLOSE Procedure

- UTL_FILE.FCLOSE procedure:
 - Use FCLOSE to close an open file.
 - The header for this procedure is:
 - where file is the file to be checked

```
PROCEDURE UTL_FILE.FCLOSE (file IN OUT FILE_TYPE);
```



Copyright © Capgemini 2015. All Rights Reserved 36

UTL_FILE.FCLOSE procedure:

- Use FCLOSE to close an open file. The header for this procedure is:

```
PROCEDURE UTL_FILE.FCLOSE (file IN OUT FILE_TYPE);
```

where file is the file handle.

- Note that the argument to UTL_FILE.FCLOSE is an IN OUT parameter, because the procedure sets the id field of the record to NULL after the file is closed.
- If there is buffered data that has not yet been written to the file when you try to close it, UTL_FILE will raise the WRITE_ERROR exception.

UTL_FILE.FCLOSE_ALL procedure:

- FCLOSE_ALL closes all the opened files. Given below is the header for this procedure:

```
PROCEDURE UTL_FILE.FCLOSE_ALL;
```

- This procedure will come in handy when you have opened a variety of files and want to make sure that none of them are left open when your program terminates.
- In programs in which files have been opened, you should also call FCLOSE_ALL in exception handlers in programs. If there is an abnormal termination of the program, files will then still be closed.

```
EXCEPTION  
  WHEN OTHERS  
  
  THEN  
    UTL_FILE.FCLOSE_ALL;  
    ... other clean up activities ...  
  END;
```

- NOTE: When you close your files with the FCLOSE_ALL procedure, none of your file handles will be marked as closed (the id field, in other words, will still be non-NULL). The result is that any calls to IS_OPEN for those file handles will still return TRUE. You will not, however, be able to perform any read or write operations on those files (unless you reopen them).

Exceptions

- FCLOSE_ALL may raise the exception UTL_FILE.WRITE_ERROR.

9.4: Handling LOB (Large Objects)

DBMS_LOB

Handling LOBs (Large Objects)

- Large Objects (LOBs) are a set of datatypes that are designed to hold large amounts of data.
- LOBs are designed to support Unstructured kind of data.
- In short:
 - LOBs are used to store Large Objects (LOBs).
 - LOBs support random access to data and has maximum size of 4 GB
 - For example: Hospital database



Copyright © Capgemini 2015. All Rights Reserved. 38

Handling LOB (Large Objects):

- Large Objects (LOBs) are a set of datatypes that are designed to hold large amounts of data.
- LOBs are designed to support Unstructured kind of data.

Unstructured Data:

- Unstructured Data cannot be decomposed into Standard Components:
 - Unstructured data cannot be decomposed into standard components. Data about an Employee can be "structured" into a Name (probably a character string), an identification (likely a number), a Salary, and so on. But if you are given a Photo, you find that the data really consists of a long stream of 0s and 1s. These 0s and 1s are used to switch pixels on or off so that you will see the Photo on a display. However, they cannot be broken down into any finer structure in terms of database storage.
- Unstructured Data is Large:
 - Also interesting is the fact that unstructured data such as text, graphic images, still video clips, full motion video, and sound waveforms tend to be large - a typical employee record may be a few hundred bytes, but even small amounts of multimedia data can be thousands of times larger.

Handling LOB (Large Objects) (contd.):**Unstructured Data (contd.):**

- Unstructured Data in System Files needs Accessing from the Database:
Finally, some multimedia data may reside on operating system files, and it is desirable to access them from the database.
- LOB Datatype helps support Internet Applications:
 - With the growth of the internet and content-rich applications, it has become imperative that the database supports a datatype that fulfills the following:
 - datatype should store unstructured data
 - datatype should be optimized for large amounts of such data
 - datatype should provide an uniform way of accessing large unstructured data within the database or outside

9.3: UTL_FILE

Writing to Files

- Two types of LOBs are supported:
- Those stored in the database either in-line in the table or in a separate segment or tablespace, such as BLOB, CLOB, and NCLOB. LOBs in the database are stored inside database tablespaces in a way that optimizes space and provides efficient access. The following SQL datatypes are supported for declaring internal LOBs: BLOB, CLOB, and NCLOB
- Those stored as operating system files, such as BFILEs.



Copyright © Capgemini 2015. All Rights Reserved 40

• Given below is a summary of all the four types of LOBs :

- **BLOB (Binary LOB)**
BLOB stores unstructured binary data up to 4GB in length.
For example: Video or picture information
- **CLOB (Character LOB)**
CLOB stores single-byte character data up to 4GB in length.
For example: Store document
- **NCLOB (National CLOB)**
NCLOB stores a CLOB column that supports multi-byte characters from National Character set defined by Oracle 8 database.
- **BFILE (Binary File)**
BFILE stores read-only binary data as an external file outside the database.
Internal objects store a locator in the Large Object column of a table.
Locator is a pointer that specifies the actual location of LOB stored out-of-line.
The LOB locator for BFILE is the pointer to the location of the binary file stored in the operating system.
Oracle supports data integrity and concurrency for all the LOBs except for BFILEs.

Type of LOBs

- BFILE (Binary File) (contd.)

```
SQL> Create table Leave  
2 (Empno number(4),  
3 S_date date,  
4 E_date date,  
5 snap blob,  
6 msg clob);  
Table created
```



LOB locator:

- The value held in a LOB column or variable is not the actual binary data, but a “locator” or pointer to the physical location of the large object.
- For internal LOBs, since one LOB value can be up to four gigabytes in size, the binary data will be stored “out of line” (i.e., physically separate) from the other column values of a row (unless otherwise specified; see the next paragraph).
 - This allows the physical size of an individual row to be minimized for improved performance (the LOB column contains only a pointer to the large object).
 - Operations involving multiple rows, such as full table scans, can be more efficiently performed.
- A user can specify that the LOB value be stored in the row itself. This is usually done when working with small LOB values. This approach decreases the time needed to obtain the LOB value. However, the LOB data is migrated out of the row when it gets too big.
- For external LOBs, the BFILE value represents a filename and an operating system directory, which is also a pointer to the location of the large object.

BFILE considerations:

- There are some special considerations you should be aware of when you work with BFILEs.

➤ **The DIRECTORY object**

- A BFILE locator consists of a directory alias and a filename. The directory alias is an Oracle8 database object that allows references to operating system directories without hard-coding directory pathnames. This statement creates a directory:

→ CREATE DIRECTORY IMAGES AS
'c:\images';

- To refer to the c:\images directory within SQL, you can use the IMAGES alias, rather than hard-coding the actual directory pathname.
- To create a directory, you need the CREATE DIRECTORY or CREATE ANY DIRECTORY privilege. To reference a directory, you must be granted the READ privilege, as in:

→ GRANT READ ON DIRECTORY IMAGES
TO SCOTT;

➤ **Populating a BFILE locator**

- The Oracle8 built-in function BFILENAME can be used to populate a BFILE locator. BFILENAME is passed a directory alias and filename and returns a locator to the file. In the following block, the BFILE variable corporate_logo is assigned a locator for the file named ourlogo.bmp located in the IMAGES directory:

```

DECLARE
  corporate_logo  BFILE;
BEGIN
  corporate_logo := BFILENAME ( 'IMAGES',
  'ourlogo.bmp' );
END;
The following statements populate the my_book_files
table; each row is associated with a file in the
BOOK_TEXT directory:
INSERT INTO my_book_files ( file_descr, book_file )
  VALUES ( 'Chapter 1', BFILENAME('BOOK_TEXT',
  'chapter01.txt') );
UPDATE my_book_files
  SET book_file = BFILENAME( 'BOOK_TEXT',
  'chapter02rev.txt' )
 WHERE file_descr = 'Chapter 2';

```

- Once a BFILE column or variable is associated with a physical file, read operations on the BFILE can be performed using the DBMS_LOB package.
- Remember that access to physical files via BFILEs is read-only, and that the BFILE value is a pointer. The contents of the file remain outside of the database, but on the same server on which the database resides.

9.4: Handling LOB (Large Objects)

Internal LOB considerations

▪ Initializing and Manipulating LOB values:

- For each LOB column, a “locator value” is stored in the table.
- The locator points to a separate data location, which the database creates to hold the LOB data
- Internal LOB column can have a value, null, or be empty.
- An empty LOB stored in a table is a LOB of zero-length, which has a locator
- You can use EMPTY_BLOB() and EMPTY_CLOB() in INSERT or UPDATE statements to initialize a NULL or non-NUL internal LOB to empty



Copyright © Capgemini 2015. All Rights Reserved 43

Internal LOB considerations

- Setting the LOB to NULL or empty:
 - to set the LOB value to null use the following query:

```
SQL> Insert into Leave values  
(7900, '17-APR-98', '20-APR-98', NULL,'The LC and Amendments  
entry Forms have been completed. All the validations have been  
incorporated and passed for testing.');
```

1 row created.



Copyright © Capgemini 2015. All Rights Reserved 44

Internal LOB considerations

- Setting the LOB to non-NULL:

- Before writing data to an internal LOB, column must be made NON-NULL, since you cannot call the OCI or the PL/SQL DBMS_LOB functions on a NULL LOB

```
SQL> Insert into leave values
2  (7439,'12-APR-98', '17-APR-98', empty_blob(),
3  'The assignments regarding Oracle 8 have
4  been completed. I'll be back on 17th');
1 row created.
```



Copyright © Capgemini 2015. All Rights Reserved 45

Internal LOB considerations: Few more points:

Given below are a few more special considerations for Internal LOBs.

- **Retaining the LOB locator**

- The following statement populates the my_book_text table, which contains CLOB column chapter_text:

```
INSERT INTO my_book_text ( chapter_descr,
chapter_text )
VALUES ( 'Chapter 1', 'It was a dark and stormy
night.' );
```

- Programs within the DBMS_LOB package require a LOB locator to be passed as input. If you want to insert the preceding row and then call a DBMS_LOB program using the row's CLOB value, you must retain the LOB locator created by your INSERT statement.
- You can do this as shown in the following block, which inserts a row, selects the inserted LOB locator, and then calls the DBMS_LOB.GETLENGTH program to get the size of the CLOB chapter_text column. Note that the GETLENGTH program expects a LOB locator.

```
DECLARE
    chapter_loc      CLOB;
    chapter_length   INTEGER;
BEGIN
    INSERT INTO my_book_text ( chapter_descr,
chapter_text )
    VALUES ( 'Chapter 1', 'It was a dark and
stormy night.' );
    SELECT chapter_text
    INTO chapter_loc
    FROM my_book_text
    WHERE chapter_descr = 'Chapter 1';
    chapter_length := DBMS_LOB.GETLENGTH(
chapter_loc );
    DBMS_OUTPUT.PUT_LINE( 'Length of Chapter
1: ' || chapter_length );
END;
/
```

This is the output of the script:
Length of Chapter 1: 31

contd.

Internal LOB considerations: Few more points (contd.):**• The RETURNING clause**

- You can avoid the second trip to the database (i.e. the SELECT of the LOB locator after the INSERT) by using a RETURNING clause in the INSERT statement.
- By using this feature, perform the INSERT operation and the LOB locator value for the new row in a single operation.

```
DECLARE
    chapter_loc      CLOB;
    chapter_length   INTEGER;
BEGIN

    INSERT INTO my_book_text ( chapter_descr,
    chapter_text )
        VALUES ( 'Chapter 1', 'It was a dark and
stormy night.' )
        RETURNING chapter_text INTO chapter_loc;

    chapter_length := DBMS_LOB.GETLENGTH(
    chapter_loc );

    DBMS_OUTPUT.PUT_LINE( 'Length of Chapter
1: ' || chapter_length );

END;
/
```

This is the output of the script:
Length of Chapter 1: 31

- The RETURNING clause can be used in both INSERT and UPDATE statements.

contd.

Internal LOB considerations: Few more points (contd.):**• NULLL LOB locators can be a problem**

- Programs in the DBMS_LOB package expect to be passed a LOB locator that is not NULL.
- For example, the GETLENGTH program raises an exception when passed a LOB locator that is NULL.

```
DECLARE
    chapter_loc      CLOB;
    chapter_length   INTEGER;

BEGIN
    UPDATE my_book_text
        SET chapter_text = NULL
        WHERE chapter_descr = 'Chapter 1'
    RETURNING chapter_text INTO chapter_loc;

    chapter_length := DBMS_LOB.GETLENGTH(
chapter_loc );

    DBMS_OUTPUT.PUT_LINE( 'Length of Chapter 1: ' || chapter_length );

EXCEPTION
    WHEN OTHERS
    THEN
        DBMS_OUTPUT.PUT_LINE('OTHERS Exception' || sqlerrm);

END;
/
```

This is the output of the script:

OTHERS Exception ORA-00600: internal error code,
arguments: ...

- When a BLOB, CLOB, or NCLOB column is set to NULL, both the LOB binary data and its LOB locator are NULL. This NULL LOB locator should not be passed to a program in the DBMS_LOB package.

Internal LOB considerations: Few more points (contd.):

- **NULL versus “empty” LOB locators**
 - Oracle8 provides the built-in functions EMPTY_BLOB and EMPTY_CLOB to set BLOB, CLOB, and NCLOB columns to “empty”.
 - For example:

```
INSERT INTO my_book_text ( chapter_descr, chapter_text )
VALUES ( 'Table of Contents', EMPTY_CLOB() );
```

- The LOB data is set to NULL. However, the associated LOB locator is assigned a valid locator value, which points to the NULL data. This LOB locator can then be passed to DBMS_LOB programs.

```
DECLARE
    chapter_loc    CLOB;
    chapter_length INTEGER;

BEGIN
    INSERT INTO my_book_text (chapter_descr,
    chapter_text)
    VALUES ( 'Table of Contents', EMPTY_CLOB() )
    RETURNING chapter_text INTO chapter_loc;

    chapter_length := DBMS_LOB.GETLENGTH(
    chapter_loc );

    DBMS_OUTPUT.PUT_LINE
    ('Length of Table of Contents: ' || chapter_length);

EXCEPTION
    WHEN OTHERS
    THEN
        DBMS_OUTPUT.PUT_LINE( 'OTHERS Exception ' ||
        sqlerrm);

END;
/
```

This is the output of the script:

Length of Table of Contents: 0

- Note that EMPTY_CLOB can be used to populate both CLOB and NCLOB columns. EMPTY_BLOB and EMPTY_CLOB can be called with or without empty parentheses.
- Note: Do not populate BLOB, CLOB, or NCLOB columns with NULL values. Instead, use the EMPTY_BLOB or EMPTY_CLOB functions, which will populate the columns with a valid LOB locator and set the associated data to NULL.

9.4: Handling LOB (Large Objects)

Accessing External LOBs

- The BFILENAME () function is used to associate a BFILE column with an external file.
 - To create a DIRECTORY object:
 - The first parameter to the BFILENAME () function is the directory alias, and the second parameter is the filename

```
SQL> alter table Leave  
add(b_file bfile);  
Table altered.
```



Copyright © Capgemini 2015. All Rights Reserved 50

Accessing External LOBs - Examples

- Example 1: Create a directory object as shown below:

```
SQL> Create or Replace Directory L_DIR as
  'SUPRIYA_COMP\DRV_SUP_C\SUP';
Directory created.
```

- Example 2: In the following statement, we associate the file TEST.TXT for Empno 7900

```
SQL> UPDATE leave
  SET b_file = bfilename('L_DIR', 'TEST.TXT')
  WHERE EMPNO = 7900;
1 row updated.
```



Copyright © Capgemini 2015. All Rights Reserved 51

Accessing External LOBs - Examples

▪ Note:

- Read operations on the BFILE can be performed by using PL/SQL DBMS_LOB package and OCI.
 - These files are read-only through BFILES.
 - These files cannot be updated or deleted through BFILES



Copyright © Capgemini 2015. All Rights Reserved 52

9.4: Handling LOB (Large Objects)

Updating LOBs

- While updating LOBs:

- Explicitly lock the rows.
 - Use the FOR UPDATE clause in a SELECT statement.

```
SELECT msg FROM Leave WHERE Empno = 7439  
FOR UPDATE;
```

```
UPDATE Leave  
SET msg = 'The assignments regarding Oracle 8 have been completed. You  
can now proceed with Developer V2. I'll be back on 17th.'  
WHERE Empno = 7439;
```

```
1 row updated.
```



Copyright © Capgemini 2015. All Rights Reserved 53

Note:

- To update a column that uses the BFILE datatype, you do not have to lock the rows.
- Using a DBMS_LOB Package:
 - Provides procedures to access LOBs.
 - Allows reading and modifying BLOBs, CLOBs, and NCLOBs, and provides read-only operations on BFILEs.
- All DBMS_LOB routines work based on LOB locators.

9.4: Handling LOB (Large Objects)

DBMS_LOB Package Routines

- Some of the procedures and functions available within DBMS_LOB package are given below:

Procedure or Function	Description
Read	Reads data from LOB value starting at a specific offset.
Substr	Performs SQL Substr function on a LOB value. It returns a part of a LOB value starting at a specific offset.
Instr	Same as SQL Instr function.
GetLength	Performs SQL length function as a LOB value.
Compare	Compares two LOB values.
Write	Writes data to the LOB at specified offset.
Append	Appends the content of source LOB to the destination LOB.
Erase	Erases all or part of LOB.
Trim	Performs SQL Rtrim function on the LOB value.
Copy	Copies all or part of LOB value from one LOB value to other column.



Copyright © Capgemini 2015. All Rights Reserved 54

DBMS LOB Package Routines:

The routines that can modify **BLOB**, **CLOB**, and **NCLOB** values are:

- APPEND() - appends the contents of the source LOB to the destination LOB
- COPY() - copies all or part of the source LOB to the destination LOB
- ERASE() - erases all or part of a LOB
- LOADFROMFILE() - loads BFILE data into an internal LOB
- TRIM() - trims the LOB value to the specified shorter length
- WRITE() - writes data to the LOB from a specified offset

The routines that read or examine **LOB** values are:

- GETLENGTH() - gets the length of the LOB value
- INSTR() - returns the matching position of the nth occurrence of the pattern in the LOB
- READ() - reads data from the LOB starting at the specified offset
- SUBSTR() - returns part of the LOB value starting at the specified offset

The read-only routines specific to **BFILEs** are:

- FILECLOSE() - closes the file
- FILECLOSEALL() - closes all previously opened files
- FILEEXISTS() - checks if the file exists on the server

- FILEGETNAME() - gets the directory alias and file name
- FILEISOPEN() - checks if the file was opened using the input BFILE
- locators
- FILEOPEN() - opens a file

DBMS_LOB Exceptions:

A DBMS_LOB function or procedure can raise any of the named exceptions shown in the table.

Exception	Code in error msg	Meaning
INVALID_ARGVAL	21560	"argument %s is null, invalid, or out of range"
ACCESS_ERROR	22925	Attempt to read/write beyond maximum LOB size on <n>.
NO_DATA_FOUND	1403	EndofLOB indicator for looping read operations.
VALUE_ERROR	6502	Invalid value in parameter.

9.4 Handling LOB (Large Objects)

BFILE Functions and Procedures

Procedure or Function	Description
FileClose	Closes an open BFILE.
FileCloseAll	Closes all the files open in a given session.
FileExists	Finds out whether the file pointed to by the locator actually exists on the server's FileSystem or not.
FileGetName	Determines only the directory alias and the filename.
FileIsOpen	Checks to see if the file is already open.
FileOpen	Opens a BFILE for the read-only access.



Copyright © Capgemini 2015. All Rights Reserved 56

BFILE - Example

- To create a procedure that displays the first 30 characters of the file, refer the following example:

```
CREATE OR REPLACE PROCEDURE b_file(Eno in number) IS  
LOC BFILE;  
    V_FILEEXISTS INTEGER;  
    v_FILEISOPEN INTEGER;  
    NUM NUMBER;  
    OFFSET NUMBER;  
    LEN NUMBER;  
    DIR_ALIAS VARCHAR2(5);  
    NAME VARCHAR2(15);  
    CONTENTS LONG;
```

contd.



Copyright © Capgemini 2015. All Rights Reserved 57

BFILE - Example

```
BEGIN
    SELECT B_FILE INTO LOC FROM LEAVE WHERE EMPNO=Eno;
    -- Check to see if file exists
    V_FILEEXISTS := DBMS_LOB.FILEEXISTS(LOC);
    IF v_FILEEXISTS = 1 THEN
        DBMS_OUTPUT.PUT_LINE('The file exists');
    ELSE
        GoTo E;
    END IF;
    -- Check if file open
    v_FILEISOPEN := DBMS_LOB.FILEISOPEN( LOC );
```

contd.



Copyright © Capgemini 2015. All Rights Reserved 58

BFILE - Example

```
--Determine actions if file is opened or not
IF v_FILEISOPEN = 1 THEN
    DBMS_OUTPUT.PUT_LINE('The file is open');
ELSE
    DBMS_OUTPUT.PUT_LINE('Opening the file');
    DBMS_LOB.FILEOPEN(LOC);
    LEN := DBMS_LOB.GETLENGTH( LOC );
    DBMS_OUTPUT.PUT_LINE('Length of the file : ' ||
    TO_CHAR(LEN));
    NUM := 40; OFFSET := 1;
    DBMS_LOB.READ(LOC, NUM, OFFSET, CONTENTS);
```

contd.



Copyright © Capgemini 2015. All Rights Reserved 59

BFILE - Example

```
DBMS_OUTPUT.PUT_LINE('Contents of the file : '||  
CONTENTS);  
END IF;  
DBMS_LOB.FILEGETNAME(LOC, DIR_ALIAS, NAME);  
DBMS_OUTPUT.PUT_LINE ('Opening '|| dir_alias ||  
name);  
DBMS_LOB.FILECLOSE(LOC); -- Close the BFILE  
<<E>>  
DBMS_OUTPUT.PUT_LINE('The file cannot be found');  
END; /
```



Copyright © Capgemini 2015. All Rights Reserved 60

BFILE - Example

- Test the procedure by executing it at SQL prompt as follows:

```
SQL> execute b_file(7900);
The file exists
Opening the file
Length of the file : 30
Contents of the file: BOSTONS MANAGEMENT CONSULTANTS
Opening L_DIR TEST.TXT
```

- MSG: PL/SQL procedure successfully completed

DBMS_LOB.FILECLOSE() procedure:

- You can call the FILECLOSE() procedure to close a BFILE that has already been opened via the input locator.
- Note that Oracle has only read-only access to BFILEs. This means that BFILEs cannot be written through Oracle.

Syntax:

```
PROCEDURE FILECLOSE (
    file_loc IN OUT BFILE);
```

Parameter:

Parameter Name	Meaning
File_loc	Locator for the BFILE to be closed.

Return values:

None

Pragmas:

None

Example:

```
PROCEDURE Example_5 IS
    fil BFILE;
BEGIN
    SELECT f_lob INTO fil FROM lob_table WHERE
key_value = 99;
    DBMS_LOB.FILEOPEN(fil);
    -- file operations
    DBMS_LOB.FILECLOSE(fil);
EXCEPTION
    WHEN some_exception
    THEN handle_exception;
END;
```

Summary

- In this lesson, you have learnt about:
 - Testing and Debugging in PL/SQL
 - DBMS_OUTPUT
 - Enabling and Disabling output
 - Writing to the DBMS_OUTPUT Buffer
 - UTL_file
 - Handling LOB (Large Objects)



Review Question

- Question 1: The value held in a LOB column or variable is not the actual binary data, but a “locator” or pointer to the physical location of the large object
 - True / False

- Question 2: CLOB stores a column that supports multi-byte characters from National Character set defined by Oracle.
 - True / False



Review Question

- Question 3: If the pointer to the file is already located at the last line of the file, UTL_FILE.GET_LINE does not return data.
 - True / False
- Question 4: The file can be open in one of the following three modes: ___, ___, and ___
- Question 5: The package ___ lets you read and write files accessible from the server on which your database is running.



Oracle (PL/SQL)

Lesson 10: SQL * Plus
Reports

Lesson Objectives

- To understand the following topics:
 - SQL*Plus reporting Commands
 - Generation of SQL Reports with different formats



10.1: SQL *Plus

Overview

- SQL *Plus is an interactive tool for the Oracle RDBMS environment.
- You can use SQL *Plus:
 - to process SQL statements one at a time,
 - to process SQL statements interactively with end users,
 - to use PL/SQL for the procedural processing of SQL statements,
 - to list and print query results,
 - to format query results into reports,
 - to describe the contents of a given table, and
 - to copy data between databases

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

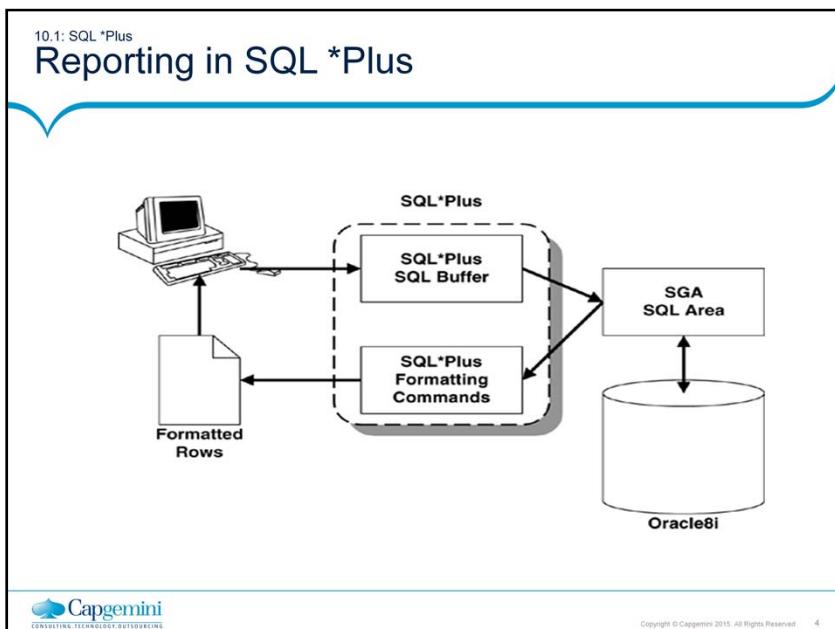
Copyright © Capgemini 2015. All Rights Reserved. 3

SQL *Plus:

- This chapter concentrates on using SQL *Plus to format output into a variety of reports.
- It introduces methods of using SQL*Plus to create dynamic, data-driven SQL *Plus programs, and operating-system specific command language programs.

Brief History of SQL *Plus

- SQL *Plus originated from the beginning of the Oracle RDBMS days as a product called User Friendly Interface (UFI).
- Before version 4 of Oracle RDBMS, UFI was used primarily to administer the Oracle environment.
- UFI later was renamed to SQL*Plus with the UFI product.
- There have been additions to several of the command capabilities, additional ways of starting SQL *Plus, and a changed role for SQL *Plus through the major releases of the Oracle RDBMS kernel.



10.1: SQL *Plus

SQL *Plus Commands

- There are six types of SQL*Plus commands:
 - Commands that initiate the SQL*Plus environment
 - SQL*Plus execute commands
 - SQL*Plus editing commands
 - SQL*Plus formatting commands
 - Miscellaneous commands
 - Access commands for various databases



Copyright © Capgemini 2015. All Rights Reserved 5

10.2: SQL *Plus commands

Commands to initiate SQL *Plus environment

- SQL *Plus is an interactive, ad hoc environment that also can be preprogrammed with the use of SQL*Plus commands, SQL statements, and PL/SQL blocks submitted via a file
- After successfully logging on to SQL*Plus, the user, regardless of the environment he or she is using, receives a SQL *Plus prompt: SQL>
- You can change this prompt message to any text string by changing the SQL *Plus system variable SQLPROMPT



Copyright © Capgemini 2015. All Rights Reserved 6

Commands that initiate the SQL *Plus environment:

- You can enhance the basic SQL *Plus environment for each user or group of users by using a file named LOGIN.SQL. This file should be located in the directory or home environment from which SQL*Plus is initiated.
- Oracle and SQL *Plus is run in a variety of computer environments. The method used to create files and the definition of the home environment vary greatly among types of computer operating systems. Typical contents of this file are various SET commands that alter the SQL *Plus default settings for the particular user.
- The PRODUCT_USER_PROFILE table, owned by SYSTEM, is one way to provide product-level security that enhances the security provided by the SQL GRANT and REVOKE commands. This level of security is used to disable certain SQL and SQL *Plus commands for individual users.
- There are various ways to initiate the SQL *Plus environment, depending on the type of computer platform being used.
- To leave the SQL *Plus environment, simply type EXIT at the SQL> prompt and press RETURN or ENTER key.
- To terminate a SQL *Plus command file, make EXIT the last line of the file.

Character-Mode Environments

- SQL *Plus is a character-based tool that runs in both character-mode and graphical environments. The way in which the SQL *Plus environment is initiated varies greatly between the two types of environments. The following syntax initiates SQL *Plus and prompts the user for a valid username and password at the command prompt:
 - C:\Sqlplus
- After SQL *Plus is started, key in a valid password and press ENTER. SQL*Plus then prompts you for a password. The password is not displayed on-screen.
- After starting SQL*Plus in character mode, the output should look similar to the following:

```
SQL*Plus: Release 9.2.0.1.0 - Production on Tue Jul 8  
21:25:8 2008  
(c) Copyright 19982, 2002 Oracle Corporation. All rights  
reserved.  
Enter username:
```
- The following syntax initiates SQL*Plus but does not prompt for the user ID or password:
 - C:\Sqlplus \nolog
- Using the option _S or _SILENT (_S spelled out) does not display the SQL*Plus version and copyright information:
 - C:\Sqlplus -S userid/password
 - This code is handy when you are initiating reports written in SQL*Plus from a menu system and you want the appearance of a seamless application.
- The following syntax initiates the SQL*Plus environment and connects the user to the remote database identified by the database name:
 - C:\Sqlplus userid/password@database
- This syntax initiates the SQL*Plus environment and executes the SQL*Plus commands and the SQL (or PL/SQL blocks) contained within the file (SQL*Plus command file):
 - C:\sqlplus userid/password @filename
- Always use an operating system-dependent, fully qualified filename when specifying a filename to execute.
- This syntax initiates the SQL*Plus environment and expects the first line of the file to contain a valid user ID and password, in this format:
 - C:\sqlplus @filename
- If the user ID and password are valid, SQL*Plus processes the SQL*Plus commands and the SQL (or PL/SQL blocks) contained in the file.
 - C:\sqlplus userid/password @filename param1 param2
 - ...
- The above command-line parameters are passed to variables inside the SQL*Plus command file and are identified inside this file by &1, &2, and so on.

Graphical-Mode Environments:

- This section discusses the syntax required to initiate the SQL*Plus environment from the Windows graphical environments. When started in graphical mode, SQL*Plus displays a Log On window that requests a valid username, a password, and a connect string, as shown in Figure.



10.2: SQL *Plus commands

Usage of Execute Commands

- You can use the execute commands to:
 - Initiate the processing of SQL statements and PL/SQL blocks,
 - Measure the processing time of SQL or PL/SQL statements,
 - Execute non-Oracle programs,
 - Execute SQL*Forms programs, or
 - Obtain additional help



Copyright © Capgemini 2015. All Rights Reserved 9

Execute Commands

- The following table lists the execute commands:

Command	Description
/	Executes the SQL statement or PL/SQL block currently in the SQL buffer (This is probably the most-used of the SQL *Plus commands).
HELP topic	Provides online assistance with SQL, PL/SQL, or SQL *Plus commands.
HOST	Provides online assistance with SQL, PL/SQL, or SQL *Plus commands.
RUN	Displays and executes the contents of the SQL buffer.
TIMING	Displays the system CPU time with the SQL prompt.

10.2: SQL *Plus commands

Usage of Editing Commands

- The SQL buffer is a work area assigned to the SQL*Plus environment.
- This buffer contains only SQL or PL/SQL syntax.
- You can use the editing commands to load, save, and manipulate the contents of this buffer.



Copyright © Capgemini 2015. All Rights Reserved 11

SQL*Plus Editing Commands (contd.):

Command	Function
A new text or APPEND new text	Appends text to the end of the current line of the SQL buffer
C/target text/new text/ or	Changes the target text to the CHANGE/target text/new text/ new text on the current line in the SQL buffer.
CLEAR BUFFER or CL BUFF	Deletes all lines in the SQL buffer.
DEL	Deletes the current line in the SQL buffer.
DEL y	Deletes line y from the SQL buffer.
DEL y z	Deletes from line y to line z in the SQL buffer.
DEL *	Deletes the current line from the SQL buffer.
DEL LAST	Deletes the last line in the SQL buffer.
EDIT filename	Uses an operating-system_dependent text editor. To edit the SQL buffer with an operating system_dependent text editor, simply leave off the filename.
GET filename	Reads an operating-system_dependent file into the SQL buffer.
I text or INPUT text	Adds the text after the current line in the SQL buffer.
L number or LIST number	Displays the contents of the SQL buffer. When the number syntax is used, LIST will display the line number and make that line the current line in the SQL buffer.
LIST y z	Displays lines y to z from the SQL buffer.
LIST LAST	Lists the last line in the SQL buffer.
SAVE filename	Saves the contents of the SQL buffer to an operating-system_dependent file.
START filename param1 param2 ...	START executes the contents of the SQL*Plus command file named in filename and passes any input parameters to the SQL*Plus command file.

10.2: SQL *Plus commands

Usage of Formatting Commands

- You use the SQL *Plus formatting commands to manipulate the result set from a SQL query
- The formatting commands follow in the subsequent slides



Copyright © Capgemini 2015. All Rights Reserved 13

Formatting Commands

- BREAK ON column_name and options:
 - This command controls the organization of rows returned by the query
 - BREAK can manipulate the appearance of the output by specifying under what conditions a BREAK should occur and what actions should be taken at the BREAK
 - The appearance of the output can be controlled by skipping a line or skipping to the top of the next page and providing totals when used with COMPUTE



Copyright © Capgemini 2015. All Rights Reserved 14

Note:

- Any number of lines can be skipped at a BREAK point.
- BREAK points can be defined at the column level, for multiple columns, on a row, on a page, or on a report.
- Refer the COMPUTE command for BREAK examples.
- Keying the BREAK by itself at the SQL prompt displays the current BREAK settings.

Formatting Commands

- BTITLE print_options and / or text or variable options:
 - BTITLE places text at the bottom of each page.
 - You can use various print options to position text at various locations
 - BTITLE simply centers the text if no print options are specified
 - Print options include BOLD, CENTER, COL, FORMAT, LEFT, RIGHT, SKIP, and TAB
 - BTITLE spelled out by itself, displays the current text setting
 - Other options that you can specify are ON and OFF
 - BTITLE is ON by default



Copyright © Capgemini 2015. All Rights Reserved 15

Formatting Commands

- COLUMN column_name and options:

- COLUMN alters the default display attributes for a given column (column_name) of a SQL query
- You can use a variety of options. However, the more common ones are FORMAT, HEADING, JUSTIFY, NEWLINE, NEW_VALUE, and NOPRINT



Copyright © Capgemini 2015. All Rights Reserved. 16

Formatting Commands:

- **FORMAT:** It is useful for applying editing to numeric fields, date masks to date fields, and specific lengths to variable-length character fields.
- **HEADING:** It overrides the SQL*Plus default heading for the particular column.
- **JUSTIFY:** It overrides the SQL*Plus column alignment to the heading default.
- **NEWLINE:** It prints the column on the beginning of the next line.
- **NEW_VALUE:** It assigns the contents of the column to a SQL*Plus variable (see DEFINE, later in this section).
 - You can then use this value with TTITLE or to store intermediate results for master / detail-type reports.
 - It is useful to store and pass information between two or more separate SQL statements.

10.2: SQL *Plus commands

Usage of Miscellaneous Commands

- Miscellaneous Commands provide a variety of commands that enable you to interact with the user, comment on the code, and enhance coding options.



Copyright © Capgemini 2015. All Rights Reserved. 17

Miscellaneous Commands

▪ ACCEPT:

- ACCEPT receives input from the terminal and places the contents in variable. This variable can already have been defined with the DEFINE command.
- If the PROMPT option is specified, then the text is displayed after skipping a line.
- You can specify the variable attributes of number or char at this stage. The variable is a char if not otherwise defined.

Miscellaneous Commands

- **DEFINE variable:**

- **DEFINE** creates a user-defined variable and assigns it to be of char (character) format
- You can assign this variable to be a default value at this stage



Copyright © Capgemini 2015. All Rights Reserved. 19

Note:

- **DEFINE statements** are handy for assigning a variable name to the input parameters coming into the SQL*Plus command file.
- **For example:**
 `DEFINE SYSTEM_NAME = &1`
- This line creates a character variable `SYSTEM_NAME` and assigns it the text associated with the first input parameter.
- The `DEFINE` statement makes the SQL*Plus command file's code easier to follow.

Miscellaneous Commands (contd.):

- **DESC or DESCRIBE database object:**
DESCRIBE displays the columns associated with a table, view, or synonym.
- **PAUSE text:**
PAUSE prints the contents of text after skipping a line, and then waits for you to press the RETURN (or ENTER) key.
- **PROMPT text:**
PROMPT simply skips a line and prints the contents of text.
- **REM or REMARK:**
SQL*Plus ignores the contents of this line when it is used in SQL*Plus command files. REMARK enables documentation or other comments to be contained in these SQL*Plus command files.
- **SET SQL*Plus System Variable:**
The SET command controls the default settings for the SQL*Plus environment. You can automatically alter these settings for each SQL*Plus session by including them in the LOGIN.SQL file, discussed earlier in this chapter in the "SQL*Plus Commands" section.
Refer Chapter 6 of Oracle's SQL*Plus User's Guide and Reference for a complete listing of the SET options.

Following are some common SET options used for reporting:

- **SET LINESIZE 80:** Controls the width of the output report line.
- **SET PAGESIZE 55:** Controls the number of lines per page.

Following are some common SET options that suppress various SQL*Plus output:

- **SET FEEDBACK OFF:** Suppresses the number of query rows returned.
- **SET VERIFY OFF:** Suppresses the substitution text when using &variables, including command-line variables.
- **SET TERMOUT OFF:** Suppresses all terminal output; this is particularly useful with the SPOOL command.
- **SET ECHO OFF:** Suppresses the display of SQL*Plus commands.
- **SPOOL filename or options:**
 - SPOOL opens, closes, or prints an operating-system_dependent file. Specifying SPOOL filename creates an operating-system-dependent file. Filename can contain the full pathname of the file and extension.
 - If no file extension is given, the file suffix, LST, is appended (filename.LST). Options include OFF and OUT.
 - If OFF is specified, the operating-system_dependent file simply is closed.
 - If OUT is specified, the operating-system_dependent file is closed and sent to the operating-system_dependent printer assigned as the default printer to the user's operating-system environment.

NOTE

If you issue SPOOL filename without issuing a SPOOL OFF or SPOOL OUT, the current operating system_dependent file is closed and the new one, as specified by the SPOOL command, is opened.

- **UNDEFINE variable:**

UNDEFINE removes the previously DEFINEd variable from the SQL*Plus environment.

Access Commands for Various Databases:

- The database-access commands CONNECT, DISCONNECT, and COPY are used to connect to and share data with other Oracle databases. (A discussion of these commands is beyond the scope of this chapter.)

Reporting Techniques:

- This section covers some common SQL*Plus report-formatting features. It also covers techniques for controlling the resulting output. You will see examples of both simple and advanced reporting techniques in this section.
- Listing 4.1 formats the results of a SQL query. It defines a report title and formats, assigns column headings, and applies some control breaks for intermediate and report totaling.
- Listing 4.1. Simple SQL*Plus report code:

```
1: define ASSIGNED_ANALYST = &1
2: set FEEDBACK OFF
3: set VERIFY OFF
4: set TERMOUT OFF
5: set ECHO OFF
6: column APPLICATION_NAME    format a12   heading
`Application'
7: column PROGRAM_NAME        format a12    heading
`Program'
8: column PROGRAM_SIZE       format 9999999 heading
`Program|Size'
9: break on APPLICATION_NAME skip 2
10: break on report skip 2
11: compute sum of PROGRAM_SIZE on
APPLICATION_NAME
12: compute sum of PROGRAM_SIZE on report
13: ttitle `Programs by Application | Assigned to:
&&ASSIGNED_ANALYST'
14: spool ANALYST.OUT
15: select
APPLICATION_NAME,PROGRAM_NAME,nvl(PROGRAM_SIZE,0)
16:   from APPLICATION_PROGRAMS
17:  where ASSIGNED_NAME = `&&ASSIGNED_ANALYST'
18:  order by APPLICATION_NAME,PROGRAM_NAME
19: /
20: spool off
21: exit
```

contd.

Reporting Techniques (contd.):

- Following is the output report from the code in Listing 13.1:
Tue Jul 13

page 1

Programs by Application Assigned to: BILLK		
Program	Application Program	Size
COBOL	CLAIMS	10156
	HOMEOWN	22124
	PREMIUMS	10345
	sum	42625
FORTRAN	ALGEBRA	6892
	MATH1	7210
	SCIENCE1	10240
	sum	24342
	sum	66967

- Listing 4.1 is a simple but common form of SQL*Plus formatting.
- This report passes a command-line parameter (&1 on line 1) and assigns it to the variable name ASSIGNED_ANALYST.
- The ASSIGNED_ANALYST variable is then used in the headings (line 13) and again as part of the SQL query (line 17).
- Lines 2 through 5 suspend all terminal output from the SQL*Plus environment.
- The && is used to denote substitution of an already defined variable.
- This report contains two breaks:
 - one when the column APPLICATION_NAME changes (line 9), and
 - one at the end of the report (line 10)
- Totals are calculated, as well, for each of these breaks (lines 11 and 12).
- The pipe character (|) in the TTITLE command (line 13) moves the following text onto its own line.
- Line 13 opens an operating-system_dependent file named ANALYST.OUT in the current operating-system_dependent directory. The order by clause of the query on line 18 ensures that the breaks occur in an orderly manner.

Note:

- Always order the query output by the breaks expected by the program. The only way to guarantee the order of the rows is to use an ORDER BY clause on the query.

Advanced Reporting Techniques

- Listing 4.2 creates a cross-tabular report with a spreadsheet appearance.
- Listing 4.2. Cross-tabular SQL*Plus report code.

```
1: define RPT_DATE = &1
2: set FEEDBACK OFF
3: set VERIFY OFF
4: set TERMOUT OFF
5: set ECHO OFF
6: column SALES REP      format a12 heading
   `Sales|Person'
7: column NISSAN        format 999999 heading `Nissan'
8: column TOYOTA        format 999999 heading `Toyota'
9: column GM            format 999999 heading `GM'
10: column FORD          format 999999 heading `Ford'
11: column CHRYSLER      format 999999 heading
   `Chrysler'
12: column TOTALS        format 999999 heading `Totals'
13: break on report skip 2
14: compute sum of NISSAN on report
15: compute sum of TOYOTA on report
16: compute sum of GM on report
17: compute sum of FORD on report
18: compute sum of CHRYSLER on report
19: compute sum of TOTALS on report
20: ttitle left `&&IN_DATE' center `Auto Sales' RIGHT `Page: '
   format 999 -
21:      SQL.PNO skip CENTER ` by Sales Person '
22: spool SALES.OUT
23: select SALES REP,
24:       sum(decode(CAR_TYPE,'N',TOTAL_SALES,0))
   NISSAN,
25:       sum(decode(CAR_TYPE,'T',TOTAL_SALES,0))
   TOYOTA,
26:       sum(decode(CAR_TYPE,'G',TOTAL_SALES,0)) GM,
27:       sum(decode(CAR_TYPE,'F',TOTAL_SALES,0)) FORD,
28:       sum(decode(CAR_TYPE,'C',TOTAL_SALES,0))
   CHRYSLER ,
29:       sum(TOTAL_SALES) TOTALS
30: from CAR_SALES
31: where SALES_DATE <= to_date(' &&RPT_DATE')
32: group by SALES REP
33: /
34: spool off
35: exit
```

contd.

Advanced Reporting Techniques (contd.):

- The following code shows the output report from Listing 4.2:

31-AUG-95 Auto Sales

Page: 1

by Sales Person

Sales

Sales Person	Nissan	Toyota	GM	Ford
Chrysler	Totals			

Elizabeth	5500	2500	0	0	4500
12500					

Emily	4000	6000	4400	2000	0
16400					

Thomas	2000	1000	6000	4000	1500
14500					

	-----	-----	-----	-----	-----
	11500	9500	10400	6000	6000

43400

- Listing 4.2 is a cross-tabular SQL*Plus command file. This report passes a command-line parameter (&1 on line 1) and assigns it to the variable name RPT_DATE.
- The RPT_DATE variable is then used in the headings (line 20) and again as part of the SQL query (line 31). Lines 2 through 5 suspend all terminal output from the SQL*Plus environment. The report is created in the operating system_dependent file SALES.OUT.
- Column-formatting commands control the appearance of the columns (lines 6 through 12). The combination of compute commands (lines 14 through 19), the sum statements in the query (lines 24 through 29), and the group by clause in the query (line 32) give the report output the appearance of a cross-tabular report.

Note:

- TTITLE technique in Listing 4.2 (lines 20 and 21) different from that of Listing 4.1 (line 13).

contd.

Advanced Reporting Techniques (contd.):

- Listing 4.3 displays a major break field with the supporting data immediately following.
- Listing 4.3. Master/detail SQL*Plus report code.

```
1:  title `Sales Detail | by Sales Rep'
2:  set HEADINGS OFF
3:  column DUMMY NOPRINT
4:  select 1 DUMMY,SALES REP NO,'Sales Person: '
   || SALES REP
5:  from sales
6:  UNION
7:  select 2 DUMMY,SALES REP NO,'-----'
8:  from sales
9:  UNION
10: select 3 DUMMY,SALES REP NO,
      rpad(CAR MAKE,4) || ` ` ||
11:    to_char(SALE AMT,'$999,999.99')
12:  from sales_detail
13: UNION
14: select 4 DUMMY,SALES REP NO,      -----
15:  from sales
16: UNION
17: select 5 DUMMY,SALES REP NO,'Total: ` ||
18:    to_char(sum(TOTAL SALES),'$999,999.99')
19:  from sales
20: UNION
21: select 6 DUMMY,SALES REP NO,      `
22:  from sales
23: order by 2,1,3
24: /
```

contd.

Advanced Reporting Techniques (contd.):

- The following code shows the output report from Listing 4.3:

Thur Aug 31

page 1

Sales Detail
by Sales Rep

Sales Person: Elizabeth

Chrysler	\$3,000
Chrysler	\$1,500
Nissan	\$2,000
Nissan	\$2,000
Nissan	\$1,500
Toyota	\$2,500

Total: \$12,500

Sales Person: Emily

Ford	\$1,000
Ford	\$1,000
GM	\$2,000
GM	\$2,400
Nissan	\$2,000
Nissan	\$2,000
Toyota	\$1,000
Toyota	\$2,500
Toyota	\$2,500

Total: \$16,400

Sales Person: Thomas

Chrysler	\$1,500
Ford	\$1,000
Ford	\$3,000
GM	\$1,400
GM	\$1,600
GM	\$3,000
Nissan	\$2,000
Toyota	\$1,000

Total: \$16,400

Advanced Reporting Techniques (contd.):

- Listing 4.3 creates a master/detail SQL*Plus report by using the SQL UNION command.
- In this example, there are six distinct, separate types of lines to be printed:
 - the sales person (line 4),
 - a line of dashes under the sales person (line 7),
 - the detail line (line 10),
 - a line of dashes under the detail total (line 14),
 - a total line (line 17), and
 - a blank line (line 21)
- There are six separate queries that have their output merged and sorted together by the SQL JOIN statement (lines 6, 9, 13, 16, 19, and 23).
- When you use JOIN to merge the output of two or more queries, the output result set must have the same number of columns.
- The headings are turned off (line 2) because regular SQL*Plus column headings are not desired for this type of report.
- The first column of each query has an alias column name of DUMMY. This DUMMY column is used to sort the order of the six types of lines (denoted by each of the six queries). The DUMMY column's only role is to maintain the order of the lines within the major sort field (SALES REP NO, in this example). Therefore, the NOPRINT option is specified in line 3.

10.2: SQL *Plus commands

More on Formatting Commands

■ Formatting Columns:

- Through the SQL *Plus COLUMN command, you can change the column headings and reformat the column data in your query results.
- Changing Column Headings:
 - When displaying column headings, you can either use the default heading or you can change it by using the COLUMN command
 - The following sections describe how default headings are derived and how to alter them using the COLUMN command. Refer the COLUMN command for more details



Copyright © Capgemini 2015. All Rights Reserved 28

More on Formatting Commands:

Formatting Columns:

Examples:

- To format the output of a currency value, you can use the following code:

```
COLUMN sal FORMAT $99,999.00 HEADING  
Salary
```

```
COLUMN home_dir NEW_VALUE home_path  
NOPRINT
```

- To assign a value to home_dir you can use the COLUMN command. The first SQL query can reference the home_dir. All other SQL queries can then reference the home_path for the information returned by the first SQL query:

10.1: SQL *Plus
Overview

▪ Default Headings:

- SQL *Plus uses column or expression names as default column headings when displaying query results.
- Column names are often short and cryptic. However, expressions can be hard to understand.
- Changing Default Headings:
 - You can define a more useful column heading with the HEADING clause of the COLUMN command, in the following format:
 - COLUMN column_name HEADING column_heading



Copyright © Capgemini 2015. All Rights Reserved 29

More on Formatting Commands:**COMPUTE function: OF options and ON break options**

- COMPUTE calculates and prints totals for groupings of rows defined by the BREAK command. You can use a variety of standard functions. The most common option is the name of the column in the query on which the total has to be calculated.
- The break option determines where the totals are to be printed and reset, as defined by the BREAK command.

For example:

- The following list produces a report with totals of monthly_sales and commissions when the sales_rep column value changes:

```
BREAK ON sales_rep SKIP 2
BREAK ON REPORT
COMPUTE SUM OF monthly_sales ON sales_rep
COMPUTE SUM OF commissions ON sales_rep
COMPUTE SUM OF monthly_sales ON REPORT
COMPUTE SUM OF commissions ON REPORT
```

- It then skips two lines and produces monthly_sales and commissions totals at the end of the report.

Note:

- The COMPUTE command resets the accumulator fields back to zero after printing.

TTITLE print_options and / or text or variable options

- TTITLE places text at the top of each page.
- You can use various print options that position text at various locations.
- TTITLE centers the text and adds date and page numbers if no print options are specified.
- Print options include BOLD, CENTER, COL, FORMAT, LEFT, RIGHT, SKIP, and TAB.
- TTITLE with no options at all, displays the current text setting.
 - Other options you can specify are ON and OFF.
 - TTITLE is ON by default.

CLEAR and options

- CLEAR resets any of the SQL*Plus formatting commands. You can also use it to clear the screen. The options include BREAKS, BUFFER, COLUMNS, COMPUTES, SCREEN, SQL, and TIMING.

10.2: SQL *Plus commands

Using Commands - Examples

Example 1: Changing a Column Heading

- To produce a report from EMP_DETAILS_VIEW with new headings specified for LAST_NAME, SALARY, and COMMISSION_PCT, key in the following commands:

```
COLUMN LAST_NAME HEADING 'LAST NAME'  
COLUMN SALARY HEADING 'MONTHLY SALARY'  
COLUMN COMMISSION_PCT  
HEADING COMMISSION SELECT LAST_NAME, SALARY,  
COMMISSION_PCT  
FROM EMP_DETAILS_VIEW  
WHERE JOB_ID='SA_MAN';
```



Copyright © Capgemini 2015. All Rights Reserved 31

Output:

LAST NAME	MONTHLY SALARY	COMMISSION
Russell	14000	.4
Partners	13500	.3
Errazuriz	12000	.3

Note: The new headings will remain in effect until you enter different headings, reset each column's format, or exit from SQL *Plus.

Using Commands - Examples

- To change a column heading to two or more words, enclose the new heading in single or double quotation marks when you enter the COLUMN command.
 - To display a column heading on more than one line, use a vertical bar (|) where you want to begin a new line.
- Note: You can use a character other than a vertical bar by changing the setting of the HEADSEP variable of the SET command.

Using Commands - Examples

Example 2: Splitting a Column Heading

- To give the columns SALARY and LAST_NAME the headings as MONTHLY SALARY and LAST NAME respectively, and to split the new headings onto two lines, key in the following commands:

```
COLUMN SALARY HEADING 'MONTHLY|SALARY'  
COLUMN LAST_NAME HEADING 'LAST|NAME'
```



Copyright © Capgemini 2015. All Rights Reserved. 33

Output:

LAST NAME NAME	MONTHLY COMMISSION SALARY	
Russell	14000	.4
Partners	13500	.3
Errazuriz	12000	.3

Using Commands - Examples

Example 3: Setting the Underline Character

- To change the character used to underline headings to an equal sign and rerun the query, key in the following commands:

```
SET UNDERLINE = /
```

LAST NAME	MONTHLY SALARY	COMMISSION
Russell	14000	.4
Partners	13500	.3
Errazuriz	12000	.3



Copyright © Capgemini 2015. All Rights Reserved 34

10.2: SQL *Plus commands

Formatting Number Columns

- When displaying NUMBER columns, you can either accept the SQL *Plus default display width or you can change it by using the COLUMN command.

```
COLUMN column_name CLEAR or exit from SQL*Plus.  
COLUMN SALARY FORMAT $99,990
```

LAST NAME	MONTHLY SALARY	COMMISSION
Russell	\$14,000	.4
Partners	\$13,500	.3
Errazuriz	\$12,000	.3



Copyright © Capgemini 2015. All Rights Reserved. 35

Note:

The above command describe the default display and how you can alter it with the COLUMN command.

10.2: SQL *Plus commands

Formatting Datatypes

- When displaying datatypes, you can either accept the SQL*Plus default display width or you can change it using the COLUMN command.
- The format model will stay in effect until you enter a new one, reset the column's format with the following command:

```
COLUMN column_name CLEAR  
or exit from SQL*Plus.
```



Copyright © Capgemini 2015. All Rights Reserved. 36

10.2: SQL *Plus commands

Formatting Character Column - Example

- To set the width of the column LAST_NAME to four characters and rerun the current query, key in the following command:

```
COLUMN LAST_NAME FORMAT A4 /
```

LAST NAME	MONTHLY SALARY	COMMISSION
Russ	\$14,00	.3
ell		
Part	\$13,500	.4
ners		
Erra	\$12,000	.3
zure		



Copyright © Capgemini 2015. All Rights Reserved 37

10.2: SQL*Plus commands

Listing & Resetting Column Display Attributes

- To list the current display attributes for a given column, use the COLUMN command followed by the column name only, as shown:

```
COLUMN column_name
```

- To list the current display attributes for all columns, key in the COLUMN command with no column names or clauses after it:

```
COLUMN
```



Copyright © Capgemini 2015. All Rights Reserved. 38

Listing & Resetting Column Display Attributes

- To reset the display attributes for a column to their default values, use the CLEAR clause of the COLUMN command as shown:

```
COLUMN column_name CLEAR
```



Copyright © Capgemini 2015. All Rights Reserved. 39

10.1: SQL *Plus commands

Suppressing & Displaying Column Display Attributes

- You can suppress and restore the display attributes you have given a specific column. To suppress a column's display attributes, key in a COLUMN command in the following form:

```
COLUMN column_name OFF
```

- OFF tells SQL *Plus to use the default display attributes for the column. However, it does not remove the attributes you have defined through the COLUMN command



Copyright © Capgemini 2015. All Rights Reserved 40

Suppressing & Displaying Column Display Attributes

- To restore the attributes you defined through COLUMN, use the ON clause:

```
COLUMN column_name ON
```



Copyright © Capgemini 2015. All Rights Reserved 41

10.1: SQL *Plus commands

Suppressing Duplicate Values in Break Columns

- The BREAK command suppresses duplicate values by default in the column or expression you name.
- Thus, to suppress the duplicate values in a column specified in an ORDER BY clause, use the BREAK command in its simplest form as follows:

```
BREAK ON break_column
```



Copyright © Capgemini 2015. All Rights Reserved 42

Suppressing Duplicate Values in Break Columns

- In this example, to suppress the display of duplicate department numbers in the query results shown, key in the following command:

```
BREAK ON DEPARTMENT_ID;
```

- for the following query (which is the current query stored in the buffer):

```
SELECT DEPARTMENT_ID, LAST_NAME, SALARY FROM  
EMP_DETAILS_VIEW WHERE SALARY > 12000 ORDER BY  
DEPARTMENT_ID;
```

Copyright © Capgemini 2015. All Rights Reserved. 43

Output:

DEPARTMENT_ID	LAST_NAME	SALARY
20	Hartstein	13000
80	Russell	14000
	Partner	35000
90	King	12000
	De Haan	50000
	Kochhar	40000

10.1: SQL *Plus commands

Inserting space when break column value changes

- BREAK ON DEPARTMENT_ID SKIP 1

DEPARTMENT_ID	LAST_NAME	SALARY
20	Hartstein	13000
80	Russell	14000
	Partner	35000
90	King	12000
	De Haan	50000
	Kochhar	40000



Copyright © Capgemini 2015. All Rights Reserved 44

Summary

■ In this lesson, you have learnt:

- Using different SQL *Plus Reporting commands
- Generating SQL Reports in different formats



Review Question

- Question 1: ___ command lists the last line in the SQL buffer
- Question 2: ___ command places text at the bottom of each page
- Question 3: SET ___ suppresses the number of query rows returned



Review Question

- Question 4: PAUSE prints the contents of text after skipping a line and then waits for you to press Enter key
 - True / False

- Question 5: SET PAGESIZE controls the width of the output report line
 - True / False



Oracle (PL/SQL)

Lesson 11: SQL * Loader

Lesson Objectives

- To understand the following topics:
 - Outline of SQL*Loader, an Oracle-supplied utility
 - SQL * Loader Environment



11.1: SQL*Loader?

What Is SQL * Loader?

- SQL*Loader is a bulk loader utility used for moving data from external files into the Oracle database.
- SQL*Loader supports various load formats, selective loading, and multi-table loads.



Copyright © Capgemini 2015. All Rights Reserved 3

Note:

SQL*Loader has a powerful data parsing engine that puts little limitation on the format of the data in the datafile. Thus SQL*Loader supports various load formats, selective loading, and multi-table loads.

11.2: SQL*Loader as a Utility

Usage Of SQL * Loader

- SQL*Loader can be used for the following utilities:
 - To load data from multiple datafiles during the same load session.
 - To load data into multiple tables during the same load session.
 - To specify the character set of the data.
 - To selectively load data (you can load records based on the records' values).
 - To manipulate the data before loading it, using SQL functions.
 - To generate unique sequential key values in specified columns.



Copyright © Capgemini 2015. All Rights Reserved 4

11.2: SQL*Loader as a Utility

Usage Of SQL * Loader

- To use the operating system's file system to access the datafiles
- To load data from disk, tape, or named pipe
- To generate sophisticated error reports, which greatly aids troubleshooting
- To load arbitrarily complex object-relational data
- To use secondary datafiles for loading LOBs and collections



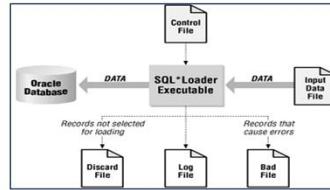
Copyright © Capgemini 2015. All Rights Reserved

5

11.3:SQL*Loader Environment

SQL * Loader Environment

- A typical SQL*Loader session takes a control file as input. The input file controls the behavior of SQL*Loader and one or more datafiles.
- The output of SQL*Loader is an Oracle database (where the data is loaded), a log file, a bad file, and potentially a discard file.



11.3:SQL*Loader Environment

The Control File

- The SQL*Loader Control file is the text file which is a key to any load process.
- The Control file provides the following information to Oracle for data load:
 - Datafile name, location, and format
 - Character sets used in the datafiles
 - Datatypes of fields in those files
 - Information on how each field is delimited
 - Information on the tables and columns to be loaded

 Capgemini
CONSULTING | TECHNOLOGY | OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 7

The Control File:

Some of the items shown in the list on the slide may also be passed to SQL*Loader as command-line parameters.

For example: The name and location of the input file, may be passed on the command line instead of in the Control file. The same goes for the names and locations of the Bad files and the Discard files.

It is also possible for the Control file to contain the actual data to be loaded. This is sometimes done when small amounts of data needs to be distributed to many sites, because it reduces the number of files that need to be passed around (to just one file).

If the data to be loaded is contained in the Control file, then there is no need for a separate Data file.

11.3:SQL*Loader Environment

The Control File

- The SQL*Loader Control file is the text file which is a key to any load process.
- The Control file provides the following information to Oracle for data load:
 - Datafile name, location, and format
 - Character sets used in the datafiles
 - Datatypes of fields in those files
 - Information on how each field is delimited
 - Information on the tables and columns to be loaded

 Capgemini
CONSULTING | TECHNOLOGY | OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 8

The Control File:

Some of the items shown in the list on the slide may also be passed to SQL*Loader as command-line parameters.

For example: The name and location of the input file, may be passed on the command line instead of in the Control file. The same goes for the names and locations of the Bad files and the Discard files.

It is also possible for the Control file to contain the actual data to be loaded. This is sometimes done when small amounts of data needs to be distributed to many sites, because it reduces the number of files that need to be passed around (to just one file).

If the data to be loaded is contained in the Control file, then there is no need for a separate Data file.

11.3:SQL* Loader Environment

The Log File

- The log file is a record of SQL*Loader's activities during a load session.
- The log file contains information as given below:
 - The names of the control file, log file, bad file, discard file, and data file
 - The values of several command-line parameters
 - A detailed breakdown of the fields and datatypes in the data file that was loaded
 - Error messages for records that cause errors
 - Messages indicating when records have been discarded



Copyright © Capgemini 2015. All Rights Reserved 9

Note:

Always review the Log file after a load to ensure that no errors have occurred, or at least that no unexpected errors occurred.

This type of information is written to the Log file. However, the information is not displayed on the terminal screen.

The Log File

- A summary of the load that includes the following:
- The number of logical records read from the data file
- The number of rows rejected because of errors
- The number of rows discarded because of selection criteria
- The time elapsed for the load



Copyright © Capgemini 2015. All Rights Reserved 10

11.3:SQL*Loader Environment

The Bad File & Discard File

- Whenever you insert data into a database, you run the risk of that insert failing because of some type of error.
- Integrity constraint violations undoubtedly represent the most common type of error.
 - However, other problems, such as the lack of free space in a tablespace, can also cause insert operations to fail.
- Whenever SQL*Loader encounters a database error while trying to load a record, it writes that record to a file known as the “Bad file”.



Copyright © Capgemini 2015. All Rights Reserved 11

The Bad File & Discard File

- Discard files are used to hold records that do not meet selection criteria specified in the SQL*Loader control file.
- By default, SQL*Loader will attempt to load all the records contained in the input file.
- Records that do not meet the specified criteria are not loaded, and are instead written to a file known as the “Discard file”.



Copyright © Capgemini 2015. All Rights Reserved 12

Note:

You have the option in your Control file, of specifying selection criteria that a record must meet before it is loaded.

Comparison between Bad Files and Discard Files:

Discard files are optional. You will only get a Discard file if you have specified a Discard file name, and if at least one record is actually discarded during the load.

Bad files are not optional. The only way to avoid having a Bad file generated is to run a load that results in no errors. If even one error occurs, then SQL*Loader will create a Bad file and write the offending input record (or records) to that file.

The format of your Bad files and Discard files will exactly match the format of your input files. This scenario occurs because SQL*Loader writes the exact records, which cause errors, or those that are discarded, to those files.

If you are running a load with multiple input files, you will get a distinct set of bad files and discard files for each input file.

11.4: Invoking SQL*Loader

Methods Of Invoking SQL*Loader

- SQL*Loader can be invoked in one of the following three ways:
 - sqlldr
 - sqlldr keyword=value [keyword=value ...]
 - sqlldr value [value ...]



Copyright © Capgemini 2015. All Rights Reserved 13

Invoking SQL*LOADER:

On Unix systems, the command used to invoke SQL*Loader is sqldr. On Windows systems running Oracle8i, release 8.1 or higher, the command is also sqldr.

Prior to release 8.1, the SQL*Loader command on Windows systems included the first two digits of the Oracle release number. Thus you had sqldr80 (Oracle8, release 8.0), sqldr73 (Oracle7, release 7.3), and so forth.

SQL*Loader can be invoked in one of three ways:

```
sqldr  
sqldr keyword=value [keyword=value ...]  
sqldr value [value ...]
```

Issuing the sqldr command by itself results in a list of valid command-line parameters being displayed. Command-line parameters are usually keyword / value pairs, and may be any combination of the following:

```
USERID={username[/password][@net_service_name]}/  
CONTROL=control_file_name  
LOG=path_file_name  
BAD=path_file_name  
DATA=path_file_name  
DISCARD=path_file_name  
DISCARDMAX=logical_record_count  
SKIP=logical_record_count  
SKIP_INDEX_MAINTENANCE={TRUE | FALSE}  
SKIP_UNUSABLE_INDEXES={TRUE | FALSE}  
LOAD=logical_record_count  
ERRORS=insert_error_count  
ROWS=rows_in_bind_array  
BINDSIZE=bytes_in_bind_array  
SILENT=[(keyword[,keyword...])]  
DIRECT={TRUE | FALSE}  
PARFILE=path_file_name  
PARALLEL={TRUE | FALSE}  
READSIZE=bytes_in_read_buffer  
FILE=database_datafile_name
```

Command-line parameters may be passed by position instead of by keyword. Command-line parameters are discussed in detail in the subsequent slides.

Invoking SQL * Loader

- Given below is a list of Command Line Parameters:
- **USERID = {username[/password] [@net_service_name]}{/}**
 - Specifies the username and password to use when connecting to the database. The `net_service_name` parameter optionally allows you to connect to a remote database. Use a forward-slash character (/) to connect to a local database by using operating system authentication.
- **CONTROL = control_file_name**
 - Specifies the name, which may include the path, of the control file. The default extension is .ctl.
- **LOG = path_file_name**
 - Specifies the name of the Log file to generate for a load session. You may include a path as well. By default, the Log file takes on the name of the Control file, but with a .log extension. The Log file is written to the same directory as the Control file. If you specify a different name, the default extension is still .log.



Copyright © Capgemini 2015. All Rights Reserved 15

Invoking SQL*Loader:

`USERID = {username[/password] [@net_service_name]}{/}`

Note:

On Unix systems, you may want to omit the password and allow SQL*Loader to prompt you for it. If you omit both the username and the password, SQL*Loader will prompt you for both.

On Unix systems you should generally avoid placing a password on the command line, because that password will be displayed whenever other users issue a command, such as `ps -ef`, that displays a list of current processes running on the system. Either let SQL*Loader prompt you for your password, or use operating system authentication. (If you don't know what operating system authentication is, ask your DBA.)

Invoking SQL * Loader

- **BAD = path_file_name**
 - Specifies the name of the Bad file. You may include a path as part of the name. By default, the Bad file takes the name of the Control file, but with a .bad extension. The Bad file is written to the same directory as the Control file.
- **DATA = path_file_name**
 - Specifies the name of the file containing the data to load. You may include a path as part of the name. By default, the name of the Control file is used, but with the .dat extension. If you specify a different name, the default extension is still .dat



Copyright © Capgemini 2015. All Rights Reserved 15

Invoking SQL*Loader (contd.):

LOG = path_file_name

Note: If you use the LOG parameter to specify a name for the log file, it will no longer be written automatically to the directory that contains the control file.

BAD = path_file_name

Note: If you specify a different name, the default extension remains .bad. However, if you use the BAD parameter to specify a bad file name, the default directory becomes your current working directory. If you are loading data from multiple files, then this bad file name only gets associated with the first file being loaded.

Invoking SQL * Loader

- DISCARD = path_file_name
 - Specifies the name of the Discard file. You may include a path as part of the name. By default, the Discard file takes the name of the Control file, but it has a .dis extension. If you specify a different name, the default extension is still .dis.
- DISCARDMAX = logical_record_count
 - Sets an upper limit on the number of logical records that can be discarded before a load will terminate. The limit is actually one less than the value specified for DISCARDMAX.



Copyright © Capgemini 2015. All Rights Reserved 17

Invoking SQL*Loader (contd.):

DATA = path_file_name

Note: If you are loading from multiple files, you can only specify the first file name using this parameter. Place the names of the other files in their respective INFILE clauses in the control file.

DISCARD = path_file_name

Note: If you are loading data from multiple files, then this discard file name only gets associated with the first file being loaded.

Invoking SQL * Loader

- SKIP = logical_record_count
 - Allows you to continue an interrupted load by skipping the specified number of logical records.
- LOAD = logical_record_count
 - Specifies a limit on the number of logical records to load. The default is to load all records. Since LOAD only accepts numeric values, it is not possible to explicitly specify the default behavior.
- ROWS = rows_in_bind_array
 - The precise meaning of this parameter depends on whether you are doing a direct path load or a conventional load.



Copyright © Capgemini 2015. All Rights Reserved 18

Invoking SQL*Loader (contd.):

DATA = path_file_name

Note: If you are loading from multiple files, you can only specify the first file name using this parameter. Place the names of the other files in their respective INFILE clauses in the control file.

DISCARD = path_file_name

Note: If you are loading data from multiple files, then this discard file name only gets associated with the first file being loaded.

Invoking SQL*Loader (contd.):

You can even mix the positional and keyword methods of passing command-line parameters. The rule to be followed while doing this is that all positional parameters must come first. Once you start using keywords, you must continue to do so.

For example:

```
sqlldr system/manager control=profile.ctl log=profile.ctl
```

When you pass parameters positionally, you must not skip any. Also, be sure to get the order right.

You must supply parameter values in the order shown earlier in this section.

Given the fact that you typically will use only a few parameters out of the many that are possible, it's usually easier to pass those parameters as keyword/value pairs than it is to pass them positionally.

Using keyword/value pairs also makes long SQL*Loader commands somewhat self-documenting. An exception to this rule is that you might wish to pass the username and password positionally, since they come first, and then pass in the rest of the parameters by name.

SQL * Loader - Examples

- Example 1:

- sqldr username@server/password control=loader.ctl
 - The sample control file (loader.ctl) will load an external data file containing delimited data:

```
load data
infile 'c:\data\mydata.csv'
into table emp fields terminated by ","
optionally enclosed by "" ( empno, empname, sal, deptno )
```

- The mydata.csv file may look like the sample shown below:

- 10001,"Scott Tiger", 1000, 40
- 10002,"Frank Naude", 500, 20



Copyright © Capgemini 2015. All Rights Reserved 20

SQL * Loader - Examples

■ Example 2:

- Another sample Control file with in-line data formatted as fix length records is discussed in this example.
- The trick is to specify "*" as the name of the data file, and use BEGINDATA to start the data section in the control file:

```
load data infile * replace
  into table departments ( dept position (02:05) char(4), deptname position
(08:27) char(20) )
begindata
COSC COMPUTER SCIENCE
ENGL ENGLISH LITERATURE
MATH MATHEMATICS
POLY POLITICAL SCIENCE
```



Copyright © Capgemini 2015. All Rights Reserved 21

SQL * Loader - Examples

- Example 1:

- Step 1: Create the Relation

- Create table customer in Oracle database Oracle account with the given structure.

Sql>Desc example

Name	Null ?	Type
custid		number
custname		varchar2(12)
age		number



Copyright © Capgemini 2015. All Rights Reserved 22

SQL * Loader Case Examples

- Step 2:

- Create Data File named customer.dat as shown below:
 - 10,"AAA",20
 - 20,"BBB",23
 - 30,"CCC",41
 - 40,"DDD",25



Copyright © Capgemini 2015. All Rights Reserved 23

SQL * Loader Case Examples

- Step 3:

- Create sql loader control file named custcontrol.ctl with the given contents:

```
load data
infile 'd:\customer.dat'
into table customer
fields terminated by ',' optionally enclosed by ""
(custid, custname, age)
```



Copyright © Capgemini 2015. All Rights Reserved 24

SQL * Loader Case Examples

- Step 4:
 - At command prompt, run SQL Loader by entering the following details:

```
C:\sqlldr scott/tiger@oracle9i control=d:\custcontrol.ctl
```

userid/password@Host String



Copyright © Capgemini 2015. All Rights Reserved 25

Summary

- In this lesson, you have learnt:
- SQL*Loader as an Oracle utility
- SQL*Loader environment
 - Control File, Log File, Bad File, and Discard File
 - Invoking SQL*Loader Environment
 - Command Line Parameters



Summary



Copyright © Capgemini 2015. All Rights Reserved 26

Review Question

- Question 1: We can use SQL*Loader to load data into multiple tables during the same load session
 - True / False
- Question 2: A typical SQL*Loader session takes as input a discard file
 - True/False
- Question 3: The SQL*Loader control file is the text file which is a key to any load process.
 - True/False



Review Question

- Question 4: The ___ file is a record of SQL*Loader's activities during a load session.
- Question 5: SQL*Loader Parameter ___ specifies a limit on the number of logical records to load.



Oracle (PL/SQL)

Lesson 12: Oracle Tools

Lesson Objectives

- To understand the following topics:
 - Various tools provided by Oracle standard installation
 - Use of Oracle tools:
 - to maintain your database
 - to create or repair existing database
 - to install or uninstall oracle software, etc



12.1: Oracle Provided Management Tools

Listing Of Tools

Tool	Description
Oracle Universal Installer (OUI)	Java-based graphical user interface used to assist with the installation, upgrade, and removal of Oracle software components.
Oracle Database Configuration Assistant	Java-based graphical user interface that can either interact with OUI or be run independently. Used to create, delete, or modify a database.
Password File Utility	Utility for creating a database password file.
SQL*Plus	Utility used to access data in an Oracle database, alter structures, and perform maintenance.
Oracle Enterprise Manager (OEM)	Graphical user interface used to administer, monitor, tune, and troubleshoot single or multiple databases.

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved. 3

Oracle Provided Management Tools:

Oracle provides many tools to assist both DBAs and developers in accomplishing their jobs in a more effective and efficient manner.

Several tools can be used to help you maintain your databases.

The tools covered by the Oracle exam will be covered in this courseware.

The table shown in the slide provides a list of tools that will be covered, along with a brief description of each.

Key Features of OUI:

Key features of OUI are discussed below:

Unified cross-platform solution:

Java-based Universal Installer offers an installation solution for all Java-enabled platforms allowing for a common “installation flow” and “user experience” independent of the platform.

This provides the ability:

- To have the installation of the same Oracle products

- on different platforms

- To look identical

- To follow the same path

Key Features of OUI (contd.):

- **Complex component and dependency definition:**
 - The new installation engine automatically detects dependencies among components and accordingly performs an installation, depending on the products and the types of the installation the user has selected.
 - It allows more complex installation flow logic to be defined as consistency checks are performed throughout the installation.
- **Web based installation:**
 - The Universal Installer can be used for the following:
 - To point to a URL where a release / staging area was defined
 - To install software remotely, over HTTP
 - A release media, CDROM, or network stage, can be simply placed on a Web server, and the installer can resolve its products installation definition.
 - The Universal Installer allows an installation session that is identical to a session performed locally.
 - The installer's ability to recognize dependent products already existent on the local target becomes more important in this case of remote installation.
 - If product dependencies with correct version numbers are detected on the local target, then the Installer will not re-install them. This potentially reduces the network traffic during installation.
- **Unattended, "silent" installations using Response Files:**
 - Response Files are collections of variable settings that provide values that would have been otherwise asked to the user.
 - For a particular component installation, the Oracle Universal Installer can read these values from a pre-defined Response File.
- **Implicit de-installation:**
 - The de-installation products, installed by using the Universal Installer, are built into the engine itself. The de-install actions are the "undo" of installation actions.
 - At install time, the Installer logs all the actions it performs to special log files.
 - At de-install time, OUI performs the reverse of all these actions.
- **Multiple Oracle homes support:**
 - Oracle Universal Installer maintains an inventory of all the Oracle Homes on target machines, their names, products, and versions of products installed on them.

12.1: Oracle Provided Management Tools

Using DBCA To Create A Database

- Database Creation Assistant (DBCA):
 - The Database Creation Assistant (DBCA) wizard guides you through the database creation process.
 - In this section, we are going to create a database using the DBCA.

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 5

Using DBCA to Create a Database:

Database Configuration Assistant (DBCA) provided by Oracle is a graphical configuration tool that simplifies the manipulation of the Oracle database structures. Its “wizard driven interface” hides the complexities involved in creating, altering, or removing existing databases.

DBCA is automatically launched at the end of an Oracle software installation to complete an Oracle working environment. Alternately, it can be launched standalone to create additional databases or alter existing ones.

DBCA simplifies the creation of a database by providing pre-defined database types. It also allows full custom database definition.

Oracle Corporation recommends that you use the DBCA to create your database. This is because the DBCA preconfigured databases optimize your environment to take advantage of Oracle9i features, such as the “server parameter file” and “automatic undo management”.

The DBCA enables you to define arbitrary tablespaces as well as a part of the “database creation process” itself.

Even if you have datafile requirements that differ from those offered in one of the DBCA templates, use the DBCA and modify the datafiles afterwards.

You can also execute user-specified scripts as part of the “database creation process”.

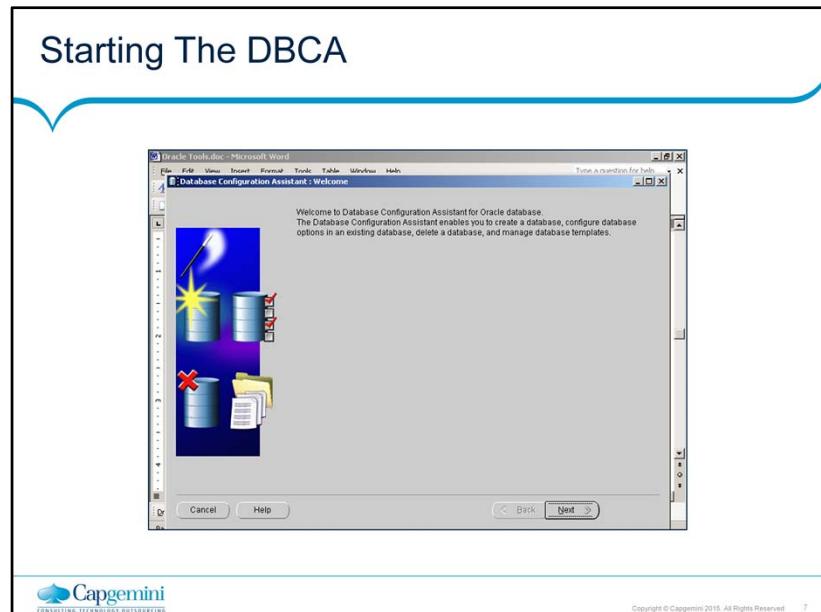
Starting The DBCA

- Starting the DBCA:

- Choose Start □ Programs □ Oracle OraHome92 (or the user-defined Oracle Home location) □ Configuration and Migration Tools □ Database Configuration Assistant.
 - If you are using Linux or UNIX, you can open DBCA from the command prompt with the simple command 'dbca'.
 - However, prior to doing this step, you may have to set up the following environment variables:
 - \$ export ORACLE_HOME=your_installation_directory
 - \$ export PATH=\$PATH:\$ORACLE_HOME/bin
 - \$ dbca



Copyright © Capgemini 2015. All Rights Reserved. 6



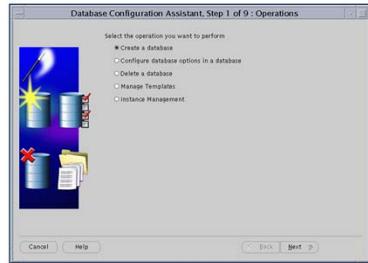
Starting the DBCA:

Once you have selected this option, the Oracle Database Configuration Assistant (DBCA) will start. You can view the DBCA welcome screen.

Simply press the NEXT button, and you will see the first of several screens that will guide you through the database creation. The first screen is shown in the above slide.

Deciding The Kind Of Database

- Deciding the kind of Database to be created:
- Notice that the DBCA can perform several functions:
 - Creation of a database
 - Configuration of database options
 - Removal of a database
 - Management of database templates



Deciding The Kind Of Database

The screenshot shows the Oracle Database Configuration Assistant, Step 3 of 9: Database Templates. The window title is "Database Configuration Assistant, Step 3 of 9 : Database Templates". The main content area displays a list of database templates with the following details:

Select	Template Name	Includes Datafile...
<input type="radio"/>	Data Warehouse	Yes
<input type="radio"/>	General Purpose	Yes
<input type="radio"/>	Transaction Processing	Yes
<input checked="" type="radio"/>	New Database	No

Below the table, there is a "Show Details..." button. At the bottom of the window, there are "Cancel", "Help", "Back", and "Next" buttons.

At the bottom of the slide, there is a Capgemini logo and the text "CONSULTING TECHNOLOGY OUTSOURCING". To the right of the logo, it says "Copyright © Capgemini 2015. All Rights Reserved" and the number "9".

Deciding the Kind of Database to be created (contd.):

Once you select the option, you will see the next screen that asks about the kind of database that is required to be created.

Again, Oracle gives us a number of options as follows:

Creation of a custom database without a template

Creation of a data warehouse, such as, database from a template

Creation of a general purpose, such as, database from a template

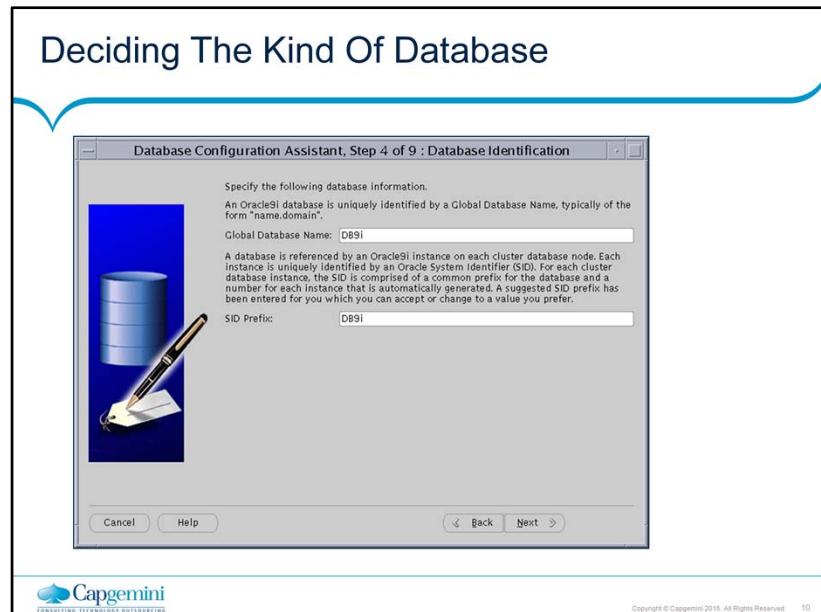
Creation of a transaction processing database from a template

In our case, we will choose to create a general purpose database.

If we select a "data warehouse" or "transaction processing database", we will find that the database creation process is much the same as the one we are about to go through.

The next screen asks for our "global database name" and the "database SID (System Identifier)".

In our example, we call our database DB9i.



Deciding the Kind of Database to be created (contd.):

Note:

After you click Next:

If you have selected the New Database template, then the DBCA displays the Database Features page.

If you have selected one of the other preconfigured database options, then after you click Next the DBCA displays the Initialization Parameters page.

Deciding The Kind Of Database

Database Configuration Assistant, Step 6 of 9 : Database Connection Options

Select the mode in which you want your database to operate by default:

- Create Shared Server Mode

For each client connection the database will allocate a resource dedicated to serving only that client. Use this mode when the number of total client connections is expected to be high, as many clients will be making persistent long-running requests to the database.
- Create Dedicated Server Mode

Several client connections share a database-allocated pool of resources. Use this mode when more than a small number of users need to connect to the database simultaneously, but memory usage will be higher.

Database Configuration Assistant, Step 9 of 9 : Creation Options

Select the following database creation options:

- Create Database
- Save as a Database Template

Name:

Description:

Generate Database Creation Scripts

Destination Directory: /share_e1/nkomar/9i2/admin/DEBS/scripts

Cancel Help Back Next Finish

Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved. 11

Deciding the Kind of Database to be created (contd.):

Note:

After you click Next, the DBCA displays the Creation Options page shown on the slide.

Review the information on the “Summary dialog”, and click OK.

To discontinue database creation, click Cancel.

If you click OK, then the DBCA displays database creation progress indicators.

At this point in the database creation process, you have:

Created an operative database.

Configured the network for the cluster database.

Started the services if you are on a Windows NT or Windows 2000 platform.

Started the listeners and database instances.

Oracle Enterprise Manager

- Oracle Enterprise Manager (OEM) is a set of systems management tools provided by Oracle Corporation for managing the Oracle environment.
- OEM provides tools to monitor the Oracle environment and automate tasks to take database administration a step closer to "Lights Out" management.
- OEM comes with pre-defined jobs like Export, Import, run OS commands, run sql scripts, SQL*Plus commands, etc.
- OEM also gives you the flexibility of scheduling custom jobs written with the TCL language.



Copyright © Capgemini 2015. All Rights Reserved. 12

12.2: Oracle Enterprise Manager

Components Of OEM

- Oracle Enterprise Manager (OEM) has the following components:
 - Management Server (OMS):
 - OMS handles communication with the intelligent agents.
 - The OEM Console connects to the management server to monitor, and configure the Oracle Enterprise.
 - Console:
 - Console is a graphical interface from where one can schedule jobs, events, and monitor the database.
 - Console can be opened from a Windows workstation, Unix XTerm (oemapp command), or Web browser session (oem_webstage).



Copyright © Capgemini 2015. All Rights Reserved 13

Components Of OEM

- Intelligent Agent (OIA):
 - OIA runs on the target database, and takes care of the execution of jobs and events scheduled through the Console.



Copyright © Capgemini 2015. All Rights Reserved 14

12.2: Oracle Enterprise Manager

Characteristics of Console

- For all Oracle Enterprise Manager operations, the Console is the primary interface used.
- It provides menus, toolbars, and the framework to access Oracle tools and utilities in addition to those available through other vendors.
- It displays the Create Object dialog allowing you to create Navigator objects such as jobs, events, database objects, and report definitions.

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved. 15

Note:

Launching the Console standalone allows a single administrator to perform simple database schema, instance, storage, security, and other database tasks by connecting directly to the target database(s).

Launching standalone does not require a middle tier “Management Server” or “Intelligent Agents” on target machines.

Consequently, when you launch the Console standalone, you do not have access to functionality typically available through the Management Server and Intelligent Agent, such as:

- Management of several different target types (e.g. database, web server, application server, applications, etc.)

- Sharing of administrative data among multiple administrators.

- Proactive notification of potential problems

- Automation of repetitive administrative tasks

- Backup and data management tools

- Customization, scheduling, and publishing of reports

- Running the client from within a web browser

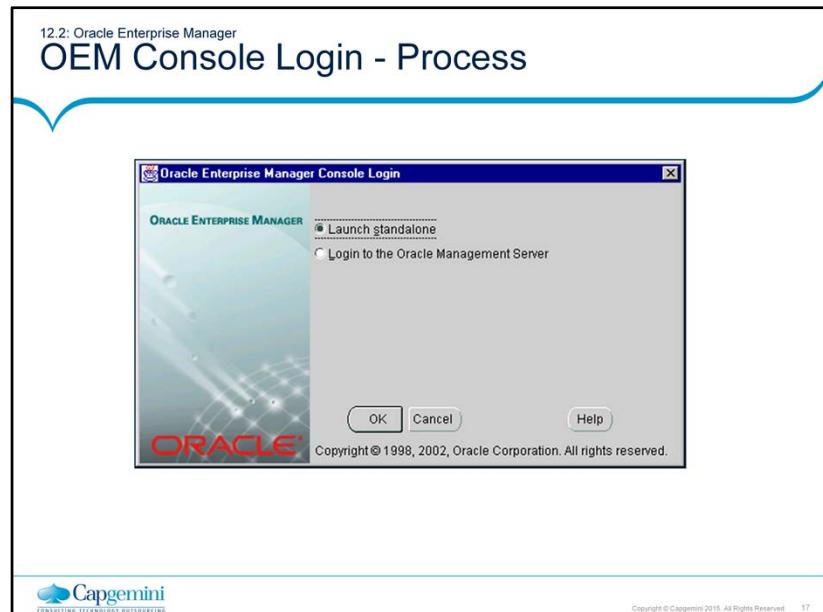
12.2: Oracle Enterprise Manager

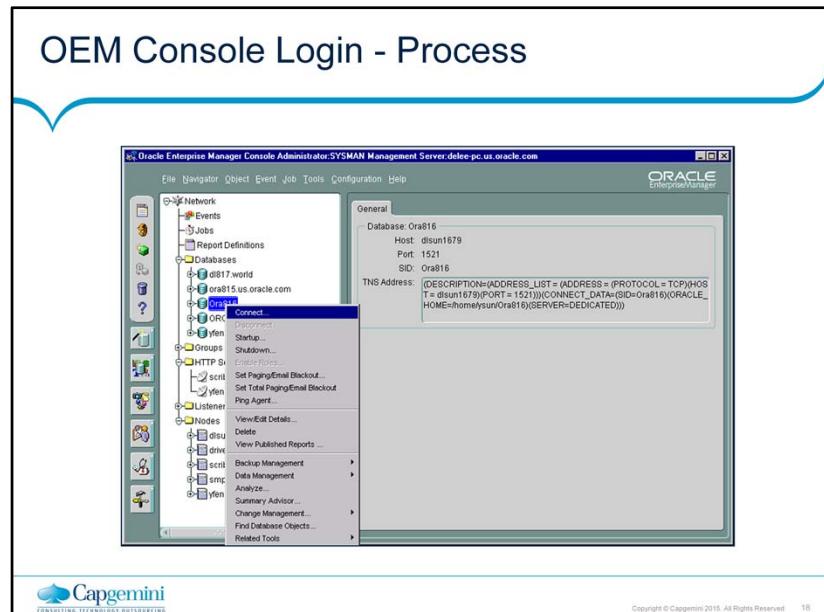
Starting Standalone Console

- On Windows-based platforms, you start the Console from the Windows Start menu.
- You can also start the standalone Console from the command line by using the following command:
 - C:\> oemapp console



Copyright © Capgemini 2015. All Rights Reserved 16





Summary

- In this lesson, you have learnt about:
 - Various Management tools provided in Oracle, namely:
 - Oracle Enterprise Manager (OEM): It is a set of systems management tools provided by Oracle Corporation for managing the Oracle environment.



Review Question

- Question 1: OEM is a set of systems management tools for managing the Oracle environment
 - True / False
- Question 2: Launching OEM standalone does require a middle tier Management Server or Intelligent Agents on target machines.
 - True/False



Oracle PL/SQL Lab Book

Document Revision History

Date	Revision No.	Author	Summary of Changes
05-Feb-2009	0.1D	Rajita Dhumal	Content Creation
09-Feb-2009		CLS team	Review
02-Jun-2011	2.0	Anu Mitra	Integration Refinements
30-Nov-2012	3.0	Hareshkumar Chandiramani	Revamp of Assignments and Conversion to iGATE format.

Table of Contents

<i>Document Revision History</i>	2
<i>Table of Contents</i>	3
<i>Getting Started</i>	4
<i>Overview</i>	4
<i>Setup Checklist for Oracle 9i</i>	4
<i>Instructions</i>	4
<i>Learning More (Bibliography if applicable)</i>	4
<i>Lab 1. Introduction to Data Dictionary</i>	5
<i>Lab 2. Introduction to PL/SQL and Cursors</i>	6
<i>Lab 3. Exception Handling and Dynamic SQL</i>	8
<i>Lab 4. Database Programming</i>	10
<i>Lab 5. Case Study 1</i>	14
<i>Lab 6. Case Study 2</i>	16
<i>Lab 7. Handling Files, DBMS_LOB</i>	18
<i>Lab 8. SQL *Plus Reports</i>	19
<i>Lab 9. SQL Loader</i>	23
<i>Appendices</i>	28
<i>Appendix A: Oracle Standards</i>	28
<i>Appendix B: Coding Best Practices</i>	29
<i>Appendix C: Table of Figures</i>	30
<i>Appendix D: Table of Examples</i>	31

Getting Started

Overview

This lab book is a guided tour for learning Oracle 9i. It comprises 'To Do' assignments. Follow the steps provided and work out the 'To Do' assignments.

Setup Checklist for Oracle 9i

Here is what is expected on your machine in order for the lab to work.

Minimum System Requirements

- Intel Pentium 90 or higher (P166 recommended)
- Microsoft Windows 95, 98, or NT 4.0, 2k, XP.
- Memory: 32MB of RAM (64MB or more recommended)

Please ensure that the following is done:

- Oracle Client is installed on every machine
- Connectivity to Oracle Server

Instructions

- For all coding standards refer Appendix A. All lab assignments should refer coding standards.
- Create a directory by your name in drive <drive>. In this directory, create a subdirectory Oracle 9i_assgn. For each lab exercise create a directory as lab <lab number>.

Learning More (Bibliography if applicable)

- Oracle10g - SQL - Student Guide - Volume 1 by Oracle Press
- Oracle10g - SQL - Student Guide - Volume 2 by Oracle Press
- Oracle10g database administration fundamentals volume 1 by Oracle Press
- Oracle10g Complete Reference by Oracle Press
- Oracle10g SQL with an Introduction to PL/SQL by Lannes L. Morris-Murphy

Lab 1. Introduction to Data Dictionary

Goals	Getting the details from various Data Dictionary Objects
Time	45 min

- 1.1: Get the details of all the database objects and their types created by the current user.
- 1.2 Get the details of all the table names owned by current user
- 1.3 Get the details of table names and corresponding column names
- 1.4 Get the details of column names and corresponding constraint names
- 1.5: Get the details of the constraints and corresponding table name.
- 1.6: Get the details of all the View names and corresponding Text of the same.
- 1.7: Get the details of all the Sequence names and their last numbers reached so far.
- 1.8: Get the details of all the Synonym names and their parent object names.
- 1.9: Get the list of all the Index names

Lab 2. Introduction to PL/SQL and Cursors

Goals	The following set of exercises are designed to implement the following <ul style="list-style-type: none"> • PL/SQL variables and data types • Create, Compile and Run anonymous PL/SQL blocks • Usage of Cursors
Time	1hr 30 min

2.1

Identify the problems(if any) in the below declarations:

```
DECLARE
V_Sample1 NUMBER(2);
V_Sample2 CONSTANT NUMBER(2) ;
V_Sample3 NUMBER(2) NOT NULL ;
V_Sample4 NUMBER(2) := 50;
V_Sample5 NUMBER(2) DEFAULT 25;
```

Example 1: Declaration Block

2.2

The following PL/SQL block is incomplete.

Modify the block to achieve requirements as stated in the comments in the block.

```
DECLARE --outer block
var_num1 NUMBER := 5;
BEGIN

DECLARE --inner block
var_num1 NUMBER := 10;
BEGIN
DBMS_OUTPUT.PUT_LINE('Value for var_num1:' ||var_num1);
--Can outer block variable (var_num1) be printed here.If Yes,Print the same.
END;
--Can inner block variable(var_num1) be printed here.If Yes,Print the same.
END;
```

Example 2: PL/SQL block

2.3. Write a PL/SQL block to retrieve all staff (code, name, salary) under specific department number and display the result. (Note: The Department_Code will be accepted from user. Cursor to be used.)

2.4. Write a PL/SQL block to increase the salary by 30 % or 5000 whichever minimum for a given Department_Code.

2.5. Write a PL/SQL block to generate the following report for a given Department code

Student_Code	Sudent_Name	Subject1	Subject2	Subject3	Total	Percentage
Grade						

Note: Display suitable error message if wrong department code has entered and if there is no student in the given department.

For Grade:

Student should pass in each subject individually (pass marks 60).

Percent \geq 80 then grade= A

Percent \geq 70 and $<$ 80 then grade= B

Percent \geq 60 and $<$ 70 then grade= C

Else D

2.6. Write a PL/SQL block to retrieve the details of the staff belonging to a particular department. Department code should be passed as a parameter to the cursor.

Lab 3. Exception Handling and Dynamic SQL

Goals	Implementing Exception Handling ,Analyzing and Debugging
Time	1 hr

3.1: Modify the programs created in Lab2 to implement Exception Handling

3.2 The following PL/SQL block attempts to calculate bonus of staff for a given MGR_CODE. Bonus is to be considered as twice of salary. Though Exception Handling has been implemented but block is unable to handle the same.

Debug and verify the current behavior to trace the problem.

```

DECLARE
  V_BONUS V_SAL%TYPE;
  V_SAL STAFF_MASTER.STAFF_SAL%TYPE;

BEGIN
  SELECT STAFF_SAL INTO V_SAL
  FROM STAFF_MASTER
  WHERE MGR_CODE=100006;

  V_BONUS:=2*V_SAL;
  DBMS_OUTPUT.PUT_LINE('STAFF SALARY IS ' || V_SAL);
  DBMS_OUTPUT.PUT_LINE('STAFF BONUS IS ' || V_BONUS);

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('GIVEN CODE IS NOT VALID.ENTER VALID CODE');
END;
  
```

Example 3: PL/SQL block

3.3 Rewrite the above block to achieve the requirement.

3.4

Predict the output of the following block ? What corrections would be needed to make it more efficient?

```
BEGIN
    DECLARE
        fname emp.ename%TYPE;
    BEGIN
        SELECT ename INTO fname
        FROM emp
        WHERE 1=2;

        DBMS_OUTPUT.PUT_LINE('This statement will print');
    EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('Some inner block error');
    END;

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No data found in fname');

    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Some outer block error');
    END;
```

Example 4: PL/SQL Block with Exception Handling

3.5 Debug the above block to trace the flow of control.

Additionally one can make appropriate changes in Select statement defined in the block to check the flow.

3.6: Write a PL/SQL program to check for the commission for an employee no 7369. If no commission exists, then display the error message. Use Exceptions.

3.7: Write a PL/SQL block to drop any user defined table.

Lab 4. Database Programming

Goals	The following set of exercises are designed to implement the following <ul style="list-style-type: none"> • Implement business logic using Database Programming like Procedures and Functions • Implement validations in Procedures and Functions • Working with Packages • Performance Tuning
Time	3.5 Hrs

Note: Procedures and functions should handle validations, pre-defined oracle server and user defined exceptions wherever applicable. Also use cursors wherever applicable.

4.1 Write a PL/SQL block to find the maximum salary of the staff in the given department.
 Note: Department code should be passed as parameter to the cursor.

4.2. Write a function to compute age. The function should accept a date and return age in years.

4.3. Write a procedure that accept staff code and update staff name to Upper case. If the staff name is null raise a user defined exception.

4.4 Write a procedure to find the manager of a staff. Procedure should return the following – Staff_Code, Staff_Name, Dept_Code and Manager Name.

4.5. Write a function to compute the following. Function should take Staff_Code and return the cost to company.

DA = 15% Salary, HRA= 20% of Salary, TA= 8% of Salary.

Special Allowance will be decided based on the service in the company.

< 1 Year	Nil
=> 1 Year < 2 Year	10% of Salary
=> 2 Year < 4 Year	20% of Salary
> 4 Year	30% of Salary

4.6. Write a procedure that displays the following information of all staff

Staff_Name	Department Name	Designation	Salary	Status
------------	-----------------	-------------	--------	--------

Note: - Status will be (Greater, Lesser or Equal) respective to average salary of their own department. Display an error message Staff_Master table is empty if there is no matching record.

4.7. Write a procedure that accept Staff_Code and update the salary and store the old salary details in Staff_Master_Back (Staff_Master_Back has the same structure without any constraint) table.

Exp < 2 then no Update
Exp > 2 and < 5 then 20% of salary
Exp > 5 then 25% of salary

4.8. Create a procedure that accepts the book code as parameter from the user. Display the details of the students/staff that have borrowed that book and has not returned the same. The following details should be displayed

Student/Staff Code Student/Staff Name Issue Date Designation Expected Ret_Date

4.9. Write a package which will contain a procedure and a function.

Function: This function will return years of experience for a staff. This function will take the hiredate of the staff as an input parameter. The output will be rounded to the nearest year (1.4 year will be considered as 1 year and 1.5 year will be considered as 2 year).

Procedure: Capture the value returned by the above function to calculate the additional allowance for the staff based on the experience.

Additional Allowance = Year of experience x 3000

Calculate the additional allowance and store Staff_Code, Date of Joining, and Experience in years and additional allowance in Staff_Allowance table.

4.10. Write a procedure to insert details into Book_Transaction table. Procedure should accept the book code and staff/student code. Date of issue is current date and the expected return date should be 10 days from the current date. If the expected return date falls on Saturday or Sunday, then it should be the next working day.

4.11: Write a function named 'get_total_records', to pass the table name as a parameter, and get back the number of records that are contained in the table. Test your function with multiple tables.

4.12

Tune the following Oracle Procedure enabling to gain better performance.

Objective: The Procedure should update the salary of an employee and at the same time retrieve the employee's name and new salary into PL/SQL variables.

```

CREATE OR REPLACE PROCEDURE update_salary (emp_id NUMBER) IS
  v_name  VARCHAR2(15);
  v_newsal NUMBER;
BEGIN
  UPDATE emp_copy SET sal = sal * 1.1
  WHERE empno = emp_id;

  SELECT ename, sal INTO v_name, v_newsal
  FROM emp_copy
  WHERE empno = emp_id;

  DBMS_OUTPUT.PUT_LINE('Emp Name:' || v_name);
  DBMS_OUTPUT.PUT_LINE('Ename:' || v_newsal);
END;
  
```

Example 5: Oracle Procedure

4.13

The following procedure attempts to delete data from table passed as parameter. This procedure has compilation errors. Identify and correct the problem.

```

CREATE or REPLACE PROCEDURE gettable(table_name in varchar2) AS
BEGIN
  DELETE FROM table_name;
END;
  
```

Example 6: Oracle Procedure

4.14

Write a procedure which prints the following report using procedure:

The procedure should take deptno as user input and appropriately print the emp details.

Also display :

Number of Employees, Total Salary, Maximum Salary, Average Salary

Note: The block should achieve the same without using Aggregate Functions.

Sample output for deptno 10 is shown below:

```
Employee Name : CLARK
Employee Job : MANAGER
Employee Salary : 2450
Employee Comission :
*****
Employee Name : KING
Employee Job : PRESIDENT
Employee Salary : 5000
Employee Comission :
*****
Employee Name : MILLER
Employee Job : CLERK
Employee Salary : 1300
Employee Comission :
*****
Number of Employees : 3
Total Salary : 8750
Maximum Salary : 5000
Average Salary : 2916.67
```

Figure 1 :Report

4.15: Write a query to view the list of all procedures ,functions and packages from the Data Dictionary.

Lab 5. Case Study 1

Goals	Implementation of Procedures/Functions ,Packages with Testing and Review
Time	2.5hrs

Consider the following tables for the case study.

Customer_Masters

Name	Null?	Type
Cust_Id	Not Null	Number(6)
Cust_Name	Not Null	Varchar2(20)
Address		Varchar2(50)
Date_of_acc_creation		Date
Customer_Type		Char(3)

Note: Customer type can be either IND or NRI

Account_Masters Table

Name	Null?	Type
Account_Number	Not Null	Number(6)
Cust_ID		Number(6)
Account_Type		Char(3)
Ledger_Balance		Number(10)

Note: Account type can be either Savings (SAV) or Salary (SAL) account.
For savings account minimum amount should be 5000.

Transaction_Masters

Name	Null?	Type
Transaction_Id	Not Null	Number(6)
Account_Number		Number(6)
Date_of_Transaction		Date
From_Account_Number	Not Null	Number(6)
To_Account_Number	Not Null	Number(6)
Amount	Not Null	Number(10)
Transaction_Type	Not Null	Char(2)

Note: Transaction type can be either Credit (CR) or Debit (DB).

Procedure and function should be written inside a package.
All validations should be taken care.

5.1 Create appropriate Test Cases for the case study followed up by Self/Peer to Peer Review and close any defects for the same.

5.2 Write a procedure to accept customer name, address, and customer type and account type. Insert the details into the respective tables.

5.3. Write a procedure to accept customer id, amount and the account number to which the customer requires to transfer money. Following validations need to be done

- Customer id should be valid
- From account number should belong to that customer
- To account number cannot be null but can be an account which need not exist in account masters (some other account)
- Adequate balance needs to be available for debit

5.4 Ensure all the Test cases defined are executed. Have appropriate Self/Peer to Peer Code Review and close any defects for the same.

Lab 6. Case Study 2

Goals	Implementation of Procedures/Functions ,Packages with Testing and Review
Time	2.5hrs

Consider the following table (myEmp) structure for the case study

EmpNo	Ename	City	Designation	Salary
-------	-------	------	-------------	--------

The following procedure accepts Task number and based on the same performs an appropriate task.

```

PROCEDURE run_task (task_number_in IN INTEGER)
IS
BEGIN
  IF task_number_in = 1
  THEN
    add_emp;
    --should add new emps in myEmp.
    --EmpNo should be inserted through Sequence.
    --All other data to be taken as parameters.Default location is Mumbai.
  END IF;
  IF task_number_in = 2
  THEN
    raise_sal;
    --should modify salary of an existing emp.
    --should take new salary and empno as input parameters
    --Should handle exception in case empno not found
    --upper limit of rasing salary is 30%. should raise exception appropriately
  END IF;
  IF task_number_in = 3
  THEN
    remove_emp;
    --should remove an existing emp
    --should take empno as parameter
  END IF;
END;
  
```

```
--Handle exception if empno not available  
END IF;  
END run_task;
```

Example 7: Sample Oracle Procedure

However ,it has been observed the method adopted in above procedure is inefficient.

6.1

Create appropriate Test Cases for the case study followed up by Self/Peer to Peer Review and close any defects for the same.

6.2

Recreate the procedure (run_task) which is more efficient in performing the same.

6.3

Also, create relevant procedures (add_emp , raise_sal ,remove_emp)
with relevant logic (read comments)to verify the same.

6.4 Extend the above implementation using Packages

6.5) Ensure all the Test cases defined are executed. Have appropriate Self/Peer to Peer Code Review and close any defects for the same.

Lab 7. Handling Files, DBMS_LOB

Goals	Working with UTL_FILE and subprograms of this package
Time	2 hr 30 mins

7.1: The following PL/SQL block creates file “TestFile.txt” with appropriate contents. Enhance the block by reading the contents back from the file and displaying it at SQL prompt.

```

Declare
  TextHandler Utl_File.File_type;
  WriteMessage Varchar2(400);
  ReadMessage Varchar2(400);
Begin
  TextHandler:=Utl_File.Fopen('d:\Sample','TestFile.txt','W');
  WriteMessage:='FOPEN is a Function, which returns the value of type
  File_Type \n UTL_file.PUT_LINE is a procedure in UTL_FILE, which write a line
  to a file, Specific line terminator will be appended '\n';

  Utl_file.Putf(TextHandler,writeMessage);
  Utl_File.Fflush(TextHandler);
  Utl_File.Fclose(TextHandler);

End;
/

```

Example 8: Block using File Handling operations

7.2 Extend the implementation in the above block by incorporating Exception Handling.

7.3 We need to maintain the above block in database permanently.What can be done for the same ? Rewrite the above to achieve the same.

7.4: Write a PL/SQL block to create a file “EmpDeptDetails.Txt” .The contents in the file map to columns from both the tables Emp and Dept.
 (Empno,Ename,Job, Sal, Dname, Loc)
 Also read the contents back from the file and display it on prompt.

Lab 8. SQL*Plus Reports

Note: Demos are provided for additional reference.

Goals	Use SQL*Plus Reports feature and come up with reports in specified formats.
Time	1 hr

8.1: Using Multiple Spacing Techniques

Suppose you have more than one column in your ORDER BY clause and wish to insert space when each column's value changes. Each BREAK command you enter replaces the previous one.

Now consider a scenario where you want to do either of the following:

- to use different spacing techniques in one report, or
- to insert space after the value changes in more than one ordered column

Then you must specify "multiple columns" and "actions" in a single BREAK command.

Step 1: Combine the Spacing Techniques.

```
SELECT DEPARTMENT_ID, JOB_ID, LAST_NAME, SALARY FROM
EMP_DETAILS_VIEW
WHERE SALARY>12000
ORDER BY DEPARTMENT_ID, JOB_ID;
```

Example 9: Sample Code

Now, to skip a page when the value of DEPARTMENT_ID changes, and to skip one line when the value of JOB_ID changes, key in the following command:

```
BREAK ON DEPARTMENT_ID SKIP PAGE ON JOB_ID SKIP 1
```

Example 10: Sample Code

To show that SKIP PAGE has taken effect, create a TTITLE with a page number:

```
TTITLE COL 35 FORMAT 9 'Page:' SQL.PNO
```

Example 101: Sample Code

Page: 1			
DEPARTMENT_ID	JOB_ID	LAST_NAME	SALARY
20	MK_MAN	Hartstein	13000
Page: 2			
DEPARTMENT_ID	JOB_ID	LAST_NAME	SALARY
80	SA_MAN	Russell	14000
	Partners		13500
Page: 3			
DEPARTMENT_ID	JOB_ID	LAST_NAME	SALARY
90	AD_PRES	King	24000
	AD_VP	Kochhar	17000
	De Haan		17000

6 rows selected.

Figure 2: Report

Step 2: Produce a report that does the following when the value of JOB_ID changes:

- prints duplicate job values,
- prints the average of SALARY, and
- inserts one blank line

Additionally the report should do the following when the value of DEPT_ID changes:

- prints the sum of SALARY, and
- inserts another blank line

The details should be displayed for all departments respective to jobs. (**To Do**)

DEPT_ID	JOB_ID	ENAME	SALARY
50	SH_CLERK	Taylor	3200
	SH_CLERK	Fleaur	3100
.			
.			
.	SH_CLERK	Gates	2900
*****			-----
	Avg:		3000
DEPT_ID	JOB_ID	ENAME	SALARY
50	SALESMAN	Perkins	2500
	SALESMAN	Bell	4000
.			
.			
.	SALESMAN	Grant	2600
*****			-----
	Avg:		3215
DEPT_ID	JOB_ID	ENAME	SALARY
*****			-----
sum			64300
-			
-			
-			
-			
-			

25 rows selected.

Figure 3: Report

8.2: Computing and Printing Subtotals

Step 1: Generate SQL Report in the following format.

SALES DEPARTMENT PERSONNEL REPORT		
PERFECT WIDGETS		
		01-JAN-2008 PAGE: 1
DEPARTMENT_ID	LAST_NAME	SALARY
-----	-----	-----
20	Hartstein	13000
80	Russell	14000
80	Partners	13500
90	King	24000
90	Kochhar	17000
90	De Haan	17000

		98500
COMPANY CONFIDENTIAL		
6 rows selected.		

Figure 4: SQL Report

Step 2: Generate SQL Report in the following format.

Dept No.	Job Name	No. of Employees	Average Salary/Job
---	-----	-----	-----
10	SALES	4	\$13,000.00
	CLEARK	2	\$10,666.00
20	MANAGER	3	\$14,000.00
	SALES	6	\$11,000.00
30	CLERK	10	\$13,500.00
	MANAGER	3	\$15,000.00
	SALES	4	\$10,000.00
40	PRESIDENT	1	424,000.00
		-----	-----
Grand Total of Sal:			\$98,560.00
No. Of Employees:			100

Figure 5: SQL Report

Lab 9. SQL Loader

Note: Demos are provided for additional reference.

Goals	<ul style="list-style-type: none">• Use the SQL Loader utility and upload the given files under specified conditions.
Time	1 hr

9.1: Load Variable-Length Data

Step 1: Crosscheck for the relation. If it does not exist, create the Dept table.

Step 2: Create and save the control file named as 'deptcontrol.ctl' in the specified location. Control file should contain following details:

```
12,RESEARCH,"SARATOGA"  
10,"ACCOUNTING",CLEVELAND  
11,"ART",SALEM  
13,FINANCE,"BOSTON"  
21,"SALES",PHILA.  
22,"SALES",ROCHESTER  
42,"INT'L","SAN FRAN
```

Example 112: Control File

Step 3: Execute the SQL Loader command at the command prompt and verify the updated Relation by issuing the command given below:

```
SQL>select * from dept;
```

Example 13: Sample Code

9.2: Use the SQL Loader utility. Upload the data file having Fixed-Format Fields, as shown below, into Emp table.

Data file

- Given below are a few sample data lines from the file ulcase2.dat.
- Blank fields are automatically set to null.

7782 CLARK	MANAGER	7839	2572.50	10
7839 KING	PRESIDENT		5500.00	10
7934 MILLER	CLERK	7782	920.00	10
7566 JONES	MANAGER	7839	3123.75	20
7499 ALLEN	SALESMAN	7698	1600.00	300.00
7654 MARTIN	SALESMAN	7698	1312.50	1400.00
7658 CHAN	ANALYST	7566	3450.00	20
7654 MARTIN	SALESMAN	7698	1312.50	1400.00

Example 124: Sample Code

9.3: Load combined physical records

Control file:

The control file is ulcase4.ctl:

```

LOAD DATA
INFILE 'ulcase4.dat'
1) DISCARDFILE 'ulcase4.dsc'
2) DISCARDMAX 999
3) REPLACE
4) CONTINUEIF THIS (1) = '*'
INTO TABLE emp
(empno      POSITION(1:4)      INTEGER EXTERNAL,
ename       POSITION(6:15)     CHAR,
job        POSITION(17:25)    CHAR,
mgr        POSITION(27:30)    INTEGER EXTERNAL,
sal         POSITION(32:39)   DECIMAL EXTERNAL,
comm       POSITION(41:48)    DECIMAL EXTERNAL,
deptno     POSITION(50:51)    INTEGER EXTERNAL,
hiredate   POSITION(52:60)    INTEGER EXTERNAL)
  
```

Example 135: Control file

Datafile

A sample datafile used for this case, ulcase4.dat, is shown below. Asterisks are in the first position. Further, a newline character, though not visible, is in position 20. Note that clerk's commission is -10, and SQL*Loader loads the value, converting it to a negative number.

*7782 CLARK						
MANAGER	7839	2572.50	-10	25	12-NOV-85	
*7839 KING						
PRESIDENT		5500.00	25	05-APR-83		
*7934 MILLER						
CLERK	7782	920.00	25	08-MAY-80		
*7566 JONES						
MANAGER	7839	3123.75	25	17-JUL-85		
*7499 ALLEN						
SALESMAN	7698	1600.00	300.00	25	3-JUN-84	
*7654 MARTIN						
SALESMAN	7698	1312.50	1400.00	25	21-DEC-85	
*7658 CHAN						
ANALYST	7566	3450.00	25	16-FEB-84		
*CHEN						
ANALYST	7566	3450.00	25	16-FEB-84		
*7658 CHIN						
ANALYST	7566	3450.00	25	16-FEB-84		

Example 146: Datafile

9.4: Load data into multiple tables.

The control file is ulcase5.ctl.

```
-- Loads EMP records from first 23 characters
-- Creates and loads PROJ records for each PROJNO listed
-- for each employee
LOAD DATA
INFILE 'ulcase5.dat'
BADFILE 'ulcase5.bad'
DISCARDFILE 'ulcase5.dsc'
1) REPLACE
2) INTO TABLE emp
(empno POSITION(1:4) INTEGER EXTERNAL,
ename POSITION(6:15) CHAR,
deptno POSITION(17:18) CHAR,
mgr POSITION(20:23) INTEGER EXTERNAL)
2) INTO TABLE proj
-- PROJ has two columns, both not null: EMPNO and PROJNO
3) WHEN projno != ' '
```

```

(empno POSITION(1:4) INTEGER EXTERNAL,
3) projno POSITION(25:27) INTEGER EXTERNAL) -- 1st proj
2) INTO TABLE proj
4) WHEN projno != ' '
  (empno POSITION(1:4) INTEGER EXTERNAL,
4) projno POSITION(29:31 INTEGER EXTERNAL) -- 2nd proj

2) INTO TABLE proj
5) WHEN projno != ' '
  (empno POSITION(1:4) INTEGER EXTERNAL,
5) projno POSITION(33:35) INTEGER EXTERNAL) -- 3rd proj

```

Example 157: Control File

Notes:

1. REPLACE specifies that if there is data in the tables to be loaded (emp and proj), SQL*loader should delete the data before loading new rows.
2. Multiple INTO TABLE clauses load two tables, emp and proj. The same set of records is processed three times, using different combinations of columns each time to load table proj.
3. WHEN loads only rows with nonblank project numbers. When projno is defined as columns 25...27, rows are inserted into proj only if there is a value in those columns.
4. When projno is defined as columns 29...31, rows are inserted into proj only if there is a value in those columns.
5. When projno is defined as columns 33...35, rows are inserted into proj only if there is a value in those columns.

Datafile:

1234 BAKER	10 9999 101 102	103
1234 JOKER	10 9999 777 888	999
2664 YOUNG	20 2893 425 abc	102
5321 OTOOLE	10 9999 321 55	40
2134 FARMER	20 4555 236 456	
2414 LITTLE	20 5634 236 456	40
6542 LEE	10 4532 102 321	14
2849 EDDS	xx 4555 294	40
4532 PERKINS	10 9999	40
1244 HUNT	11 3452 665 133	456
123 DOOLITTLE	12 9940	132

1453 MACDONALD 25 5532

200

Example 18: Datafile

9.5: Use SQL Loader utility to load specific data from the given flat file into the table. (To Do)

The given flat file contains data in the form of DEPTNO, DEPTNAME, and LOCATION.

Using SQL Loader utility, load only DEPTNO and LOCATION from the .dat file into the department table. Do not overwrite the source data present in department table.

```
12, Research, "Saratoga"
10, "Accounting", Cleveland
11, "Art", Salem
13, Finance, Boston
21, Sales, Phila
22, Sales, Rochester
42, "Int'l", "San Fran"
```

Example19: Flat file

Appendices

Appendix A: Oracle Standards

Key points to keep in mind:

1. Write comments in your stored Procedures, Functions and SQL batches generously, whenever something is not very obvious. This helps other programmers to clearly understand your code. Do not worry about the length of the comments, as it will not impact the performance.
2. Prefix the table names with owner names, as this improves readability, and avoids any unnecessary confusion.

Some more Oracle standards:

To be shared by Faculty in class

Appendix B: Coding Best Practices

1. Avoid Dynamic SQL statements as much as possible. Dynamic SQL tends to be slower than Static SQL.
2. Perform all your referential integrity checks and data validations by using constraints (foreign key and check constraints). These constraints are faster than triggers. So use triggers only for auditing, custom tasks, and validations that cannot be performed by using these constraints.
3. Do not call functions repeatedly within your stored procedures, triggers, functions, and batches. For example: You might need the length of a string variable in many places of your procedure. However do not call the LENGTH function whenever it is needed. Instead call the LENGTH function once, and store the result in a variable, for later use.

Appendix C: Table of Figures

Figure 1: Report.....	20
Figure 2: Report.....	200
Figure 3: Report.....	211
Figure 4: SQL Report	222
Figure 5: SQL Report	223

Appendix D: Table of Examples

Example 1: Declaration Block.....	6
Example 2: PL/SQL block.....	6
Example 3: PL/SQL block.....	8
Example 4: PL/SQL Block with Exception Handling.....	9
Example 5: Oracle Procedure	12
Example 6: Oracle Procedure	12
Example 7: Sample Oracle Procedure	17
Example 8: Block using File Handling operations	18
Example 9: Sample Code.....	19
Example 11: Sample Code.....	19
Example 12: Control File	23
Example 14: Sample Code.....	24
Example 15: Control file	24
Example 16: Datafile	25
Example 17: Control File	26