Contents

Recap of the previous Lecture (if needed)

Chapter 1

RDBMS Programming Part1: Basic

1.1 Course Layout

Date: January 11, 2016

- **Week 1: Class 1 (Lecture 01)**

  Relational Database Programming: Introduction. Its role in S/W development. Relational Database Basic Constructs: Table, Keys, Views, Cardinality.

  Introduction to SQL. Difference between SQL and C/Java.

- **Week 1: Class 2 (Lecture 02)**

  Introduction to SQL (Con.) Basic concepts of Data Definition Language (DDL) and Data Manipulation Language (DML). DDL explained: How to "create table " with specification of Tablespace, user-defined constraints such as primary key and foreign key.

  Loading data from external sources (using SQL Loader).

- **Week 2: Class 1 (Lecture 03)**

  Basic Parts of Speech of SQL: *select, from, where, order by.* Usage of the operators such as: *greater than, less than, equal to , not equal to, and, or, Like, NULL, NOT NULL.*

  Sub-queries in *where* clause.

- **Week 2: Class 2 (Lecture 04)** Redundancy and Functional composition in Database. Concept of Joins: Natural joins. View: its usage and restrictions.

- **Week 3: Class 1 (Lecture 05)** Use built-in functions in SQL statements. Highlights a number of built-in functions: *CONCANT, INITCAP,INSTR, LOWER, UPPER, LENGTH, L/R PAD, L/R TRIM, SUBSTR, REPLACE,COUNT.*

  Regular Expression (O).

- **Week 3: Class 2 (Lecture 06)** SQL aggregation: group by and having clause. Outer joins (two versions). Inner joins (two versions). DECODE operator.

- **Week 4: Class 1 (Lecture 07)**

  ROLLUP and CUBE operator.

  SQL with Date and Time.

  Introduction to PL/SQL. Characters & Lexical Units. Blocks, variable types, variable scope.

- **Week4: Class 2 (Lecture 08)**

  PL/SQL Control Structures: *IF, CASE, LOOP, FOR, WHILE*

- **Week 5: Class 1 (Lecture 09)** Functions and Procedures: Basic Architecture. Subroutine calling: Positional Notation, Named Notation, Mixed Notation.

- **Week 5: Class 2 (Lecture 10)** Functions and Procedures: Further details. Explore all options in both Functions and Procedures. Comprehensive review of Functions and Procedures with real-life examples.Choice of Back-end verses Front-end functions.

- **Week 6: Class 1 (Lecture 11)** Introduction to Cursor: Identify the scenarios where usa of cursor is redundant and mandatory. Implicit cursor & Explicit cursor. 4 steps of Explicit cursor. Cursor FOR loop (minimized form).

- **Week 6: Class 2 (Lecture 12)**

  Records: explicit record, nested record, record as object, record as return type.

- **Week 7: Class 1 (Lecture 13)**

  Exception Handling: using (a) built-in functions and (b) user-defined exceptions.

- **Week 7: Class 2 (Lecture 14)** Transaction Management Part 1.

- **Week 8: Class 1 (Lecture 15)** Transaction Management Part 2.

- **Week 8: Class 2 (Lecture 16)** Oracle Collection Part 1: Collection Types (Varrays, Nested Tables, Associative Arrays).

- **Week 9: Class 1 (Lecture 17)**

  Oracle Collection Part 2: Collection Set Operators.

- **Week 9: Class 2 (Lecture 18)**

  Oracle Collection Part 3: Collection API.

- **Week 10: Class 1 (Lecture 19)** Large Objects Part 1: CLOB, BLOLB, Securefiles, BFILES.

- **Week 10: Class 2 (Lecture 20)** Large Objects Part 2: DBMSLOB Package.

- **Week 11: Class 1 (Lecture 21)** PL/SQL Package Part 1: Package Architecture.Package Specification. Package Body.

- **Week 11: Class 2 (Lecture 22)** PL/SQL Package Part 2: Grants and Synonyms. Managing Packages in the Database Catalog.
- **Week 12: Class 1 (Lecture 23)** Database Triggers Part 1. Use of triggers in integrity Management of BI.
- **Week 12: Class 2 (Lecture 24)** Database Triggers Part 2. Triggers and Procedures. Types of Triggers. Controlling Triggers.
- **Week 13: Class 1 (Lecture 25)** Introduction to Dynamic SQL.Dynamic SQL Architecture.Dynamic Statements. Dynamic Statements with inputs. DBMSSQL Package.
- **Week 13: Class 2 (Lecture 26)** Introduction to Object in Database Programming.
- **Week 14: Class 1 (Lecture 27)**

  Database Administration: Introduction Part 1: Types of Oracle Database Users.Tasks of a Database Administrator.
- **Week 14: Class 2 (Lecture 28)** Database Administration: Introduction Part 2: Managing Users and Securing the Database.Monitoring Database Operations. Backup Process in Oracle Database.
- **Week 15: Class 1 (Lecture 29)** Database performance Tuning Part 1: Primary and Secondary Index. Simple and complex index. Reverse Key Index. Use of ROWID.
- **Week 15: Class 2 (Lecture 30)** Database performance Tuning 2: Views, clusters, sequence. PL/SQL Security.

  Brief Introduction to other Relational Databases such as : MySQL, PostGRE, MS SQL Server (O).

Chapter 2

Lecture (Lec 1 & 2) for Week 1

2.1 Basic Constructs of RDBMS

- **Schema** A schema is a collection of database objects. A schema is owned by a database user and has the same name as that user. Schema objects are the logical structures that directly refer to the database's data. Schema objects include structures like *tables, views, and indexes.*

- **Table** Tables are the basic unit of data storage in an Oracle database. Database tables hold all user-accessible data. Each table has *columns and rows.* A table that has an employee database, for example, can have a column called employee number, and each row in that column is an employee's number.

- **Indexes** Indexes are optional structures associated with tables. Indexes can be created to increase the performance of data retrieval.

  Indexes are created on one or more columns of a table. After it is created, an index is automatically maintained and used by Oracle. Changes to table data (such as adding new rows, updating rows, or deleting rows) are automatically incorporated into all relevant indexes with complete transparency to the users.

- **Views** Views are customized presentations of data in one or more tables or other views. A view can also be considered a stored query. Views do not actually contain data. Rather, they derive their data from the tables on which they are based, referred to as the base tables of the views.

  Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table. They also hide data complexity and store complex queries.

- **Clusters** Clusters are groups of one or more tables physically stored together because they share common columns and are often used together. Because related rows are physically stored together, disk access time improves.

  Like indexes, clusters do not affect application design. Whether a table is part of a cluster is transparent to users and to applications. Data stored in a clustered table is accessed by SQL in the same way as data stored in a nonclustered table.

- **Synonyms** A synonym is an alias for any table, view, materialized view, sequence, procedure, function, package, type, Java class schema object, user-defined object type, or another synonym. Because a synonym is simply an alias, it requires no storage other than its definition in the data dictionary.

- **Data Dictionary** One of the most important parts of an Oracle database is its data dictionary, which is a read-only set of tables that provides information about the database. A data dictionary contains:

  - The definitions of all schema objects in the database (tables, views, indexes, clusters, synonyms, sequences, procedures, functions, packages, triggers, and so on)
  - How much space has been allocated for, and is currently used by, the schema objects
  - Integrity constraint information
  - Privileges and roles each user has been granted
  - Auditing information, such as who has accessed or updated various schema objects

- **DDL and DML** SQL statements are divided into two major categories: data definition language (DDL) and data manipulation language (DML).

  DDL: DDL statements are used to build and modify the structure of your tables and other objects in the database.

  DDL Example: CREATE TABLE , ALTER TABLE.

  DML:DML statements are used to work with the data in tables. When you are connected to most multi-user databases (whether in a client program or by a connection from a Web page script), you are in effect working with a private copy of your tables that cant be seen by anyone else until you are finished (or tell the system that you are finished).

  DML Example: The insert statement. (INSERT INTO ¡table name¿ VALUES (¡value 1¿, ... ¡value n¿);)

  The update statement. UPDATE ¡table name¿ SET ¡attribute¿ = ¡expression¿ WHERE ¡condition¿;

- **Cardinality** In data modelling terms, cardinality is how one table relates to another.

  - 1-1 (one row in table A relates to one row in tableB)
  - 1-Many (one row in table A relates to many rows in tableB)
  - Many-Many (Many rows in table A relate to many rows in tableB)

2.1.1 Tablespace, Datafiles and Objects

(source: https://docs.oracle.com/cd/A57673_01/DOC/server/doc/SCN73/ch4.htm)

The following block-diagram shows the relationship.

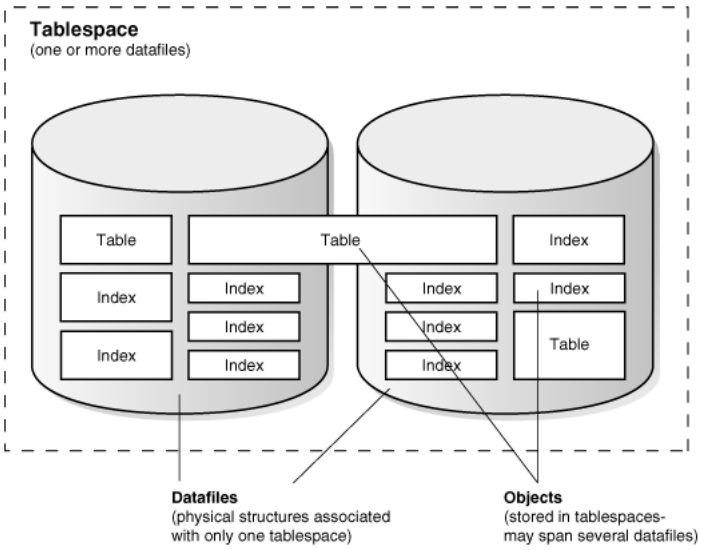

Figure 2.1: Tablespace, Datafiles and Objects

Although databases, tablespaces, datafiles, and segments are closely related, they have important differences:

- **Databases and tablespaces.** An Oracle database is comprised of one or more logical storage units called tablespaces. The database also has a lot more (background process). The database's data is collectively stored in the database's tablespaces.
- **Tablespaces and datafiles.** Each tablespace in an Oracle database is comprised of one or more operating system files called datafiles. A tablespace's datafiles physically store the associated database data on disk.
- **Databases and datafiles.**A database's data is collectively stored in the datafiles that constitute each tablespace of the database. For example, the simplest Oracle database would have one tablespace and one datafile. A more complicated database might have three tablespaces, each comprised of two datafiles (for a total of six datafiles).
- **Tablespace.** Tablespaces are the bridge between certain physical and logical components of the Oracle database. Tablespaces are where you store Oracle database objects such as tables, indexes and rollback segments.

  [A Rollback Segment is a database object containing before-images of data written to the database. Rollback segments are used to: i) Undo changes when a transaction is rolled back ii) Recover the database to a consistent state in case of failures]

  A database is divided into one or more logical storage units called tablespaces. A database administrator can use tablespaces to do the following:

- control disk space allocation for database data
- assign specific space quotas for database users
- control availability of data by taking individual tablespaces online or offline

2.1.1.1 Basic Operations on Tablespace

Basics of Space Management: Introduction to Data Blocks, Extents, and Segments.

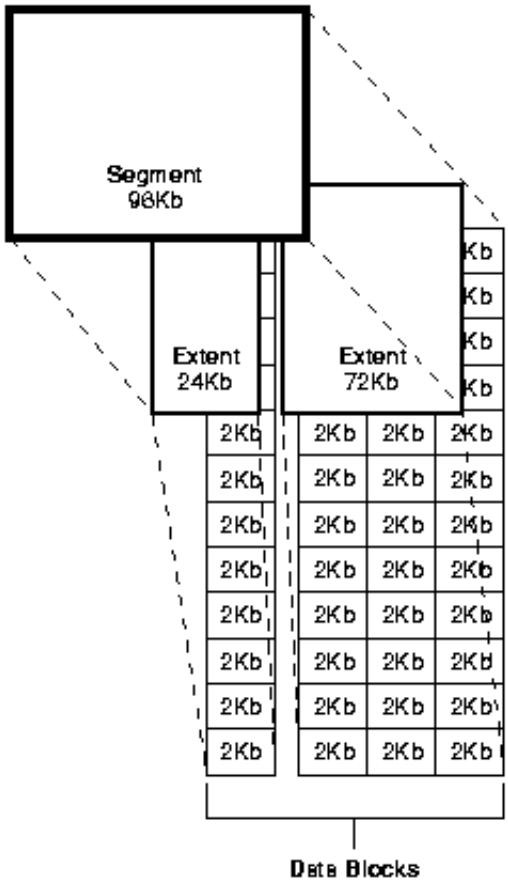

Figure 2.2: Oracle Storage Management

- **Data blocks.**At the finest level of granularity, Oracle stores data in data blocks (also called logical blocks, Oracle blocks, or pages). One data block corresponds to a specific number of bytes of physical database space on disk.
- **Extent.** The next level of logical database space is an extent. An extent is a specific number of contiguous data blocks allocated for storing a specific type of information.
- **Segment.** The level of logical database storage above an extent is called a segment. A segment is a set of extents, each of which has been allocated for a specific data structure and all of which are stored in the same tablespace. For example, each table's data is stored in its own data segment, while each index's data is stored in

its own index segment. If the table or index is partitioned, each partition is stored in its own segment.

*Oracle allocates space for segments in units of one **extent**. When the existing extents of a segment are full, Oracle allocates another extent for that segment.*

Step 1: Create a Tablespace first.

```
CREATE TABLESPACE mytspace
DATAFILE '/u02/oracle/data/lmtbsb01.dbf' SIZE 50M
EXTENT MANAGEMENT LOCAL AUTOALLOCATE;
```

AUTOALLOCATE causes the tablespace to be system managed with a minimum extent size of 64K.

Step 2 (a): Create an user and assign that user to a specific tablespace.

```
CREATE USER sidney
  IDENTIFIED BY test123
  DEFAULT TABLESPACE mytspace
```

Specify the default tablespace for objects that the user creates. If you omit this clause, then the user's objects are stored in the database default tablespace.

If no default tablespace has been specified for the database, then the user's objects are stored in the SYSTEM tablespace.

There are a number of other parameters which we are now ignoring as not part of our study objective.

Step 2 (b): Create a specific table and assign a tablespace with it (this will overrule previous).

```
create table intel
(dt varchar2(20),
tm varchar2(28),
ep number,
moteid number,
temperature number(12,5),
humidity number(20,8),
light number(12,6),
voltage number(10,6)
) tablespace NEW_TBSPACE;
```

Step 3: How to get information about free available space for a tablespace. Use `DBA_FREE_SPACE` data-dictionary.

```
SELECT TABLESPACE_NAME,SUM(BYTES)/1024/1024/1024 "FREE SPACE(GB)"
FROM DBA_FREE_SPACE GROUP BY TABLESPACE_NAME;
```

2.1.1.2 More Operations on Tablespace

Will be covered at the Admin and Tuning Sections W13 to W15

- The SYSTEM Tablespace
- Allocating More Space for a Database
- Online and Offline Tablespaces
- Read-Only Tablespaces

Chapter 3

Lecture (Lec 3 & 4) for Week 2

3.1 Basic Parts of Speech of SQL

Before you start: SQL*Plus tells you how many rows it found in One table after a valid query on it. (Notice the 14 rows selected notation at the bottom of the display.) This is called *feedback*.

It can be controlled in the following way:

```
set feedback off

set feedback 25

show feedback
```

3.1.1 Select, from, where, and order by

Run an SQL query similar to the following:

```
select Feature, Section, Page
from NEWSPAPER
where Section = 'F'
order by Page desc, Feature;
```

Default is asc in order by clause.

Logical Operators

```
Equal, Greater Than, Less Than, Not Equal
Page= 6 Page is equal to 6.
Page> 6 Page is greater than 6.
Page>= 6 Page is greater than or equal to 6.
```

```
Page< 6 Page is less than 6.
Page<= 6 Page is less than or equal to 6.
Page!= 6 Page is not equal to 6.
Page^= 6 Page is not equal to 6.
Page<> 6 Page is not equal to 6.
```

LIKE

```
Feature LIKE 'Mo%' Feature begins with the letters Mo.

Feature LIKE '_ _ I%' Feature has an I in the third position.

Feature LIKE '%o%o%' Feature has two os in it.
```

LIKE performs pattern matching. An underline character (_) represents exactly one character. A percent sign (%) represents any number of characters, including zero characters.

NULL and NOT NULL

IS NULL essentially instructs Oracle to identify columns in which the *data is missing.*

Tests Against a List of Values.

For Numbers

```
Page IN (1,2,3) => Page is in the list (1,2,3)

Page NOT IN (1,2,3) => Page is not in the list (1,2,3)

Page BETWEEN 6 AND 10 => Page is equal to 6, 10, or anything in between

Page NOT BETWEEN 6 AND 10 => Page is below 6 or above 10
```

For Strings

```
Section IN ('A','C','F') => Section is in the list ('A','C','F')

Section NOT IN ('A','C','F'))=>  Section is not in the list ('A','C','F')

Section BETWEEN 'B' AND 'D' =>Section is equal to 'B', 'D', or anything
in between (alphabetically)

Section NOT BETWEEN 'B' AND 'D' => Section is below 'B' or above 'D' (alphabetically)
```

AND , OR Similar.

3.1.2 Sub-queries.

Example in a real-life scenario:

A subquery answers multiple-part questions. For example, to determine who works in Taylor's department, you can first use a subquery to determine the department in which Taylor works. You can then answer the original question with the parent SELECT statement.

A subquery in the WHERE clause of a SELECT statement is also called *a nested subquery.*

A subquery in the FROM clause of a SELECT statement is also called *an inline view.*

(a) Subquery in WHERE clause: Example

The following statement *returns data about employees whose salaries exceed their department average.* The following statement assigns an alias to employees, the table containing the salary information, and then uses the alias in a correlated subquery:

```
SELECT department_id, last_name, salary
  FROM employees x
  WHERE salary > (SELECT AVG(salary)
     FROM employees
     WHERE x.department_id = department_id)
  ORDER BY department_id;
```

(b) Subquery in FROM clause : Example

A subquery can also be found in the FROM clause. These are called inline views.

Display the top five earner names and salaries from the EMPLOYEES table:

```
SELECT ROWNUM as RANK, last_name, salary
FROM (SELECT last_name, salary
 FROM employees
 ORDER BY salary DESC)
 WHERE ROWNUM <= 5;
```

3.2 Joins,UNION, INTERSECT, and MINUS

3.2.1 Joins

3.2.1.1 Inner Join or Natural Join

(a) New format using *natural* keyword

You can use the natural keyword to indicate that a join should be performed based on all columns that have the same name in the two tables being joined. For example, what titles in `BOOK_ORDER` match those already in BOOKSHELF?

```
 select Title
from BOOK_ORDER natural join BOOKSHELF;
```

The natural join returned the results as if you had typed in the following:

```
 select BO.Title
from BOOK_ORDER BO, BOOKSHELF
where BO.Title = BOOKSHELF.Title
and BO.Publisher = BOOKSHELF.Publisher
and BO.CategoryName = BOOKSHELF.CategoryName;
```

Note: The main difference is that INNER JOIN is used in real life; NATURAL JOIN is only used in textbooks.

INNER JOIN Note that they support the on and using clauses, so you can specify your join criteria as shown in the following listing:

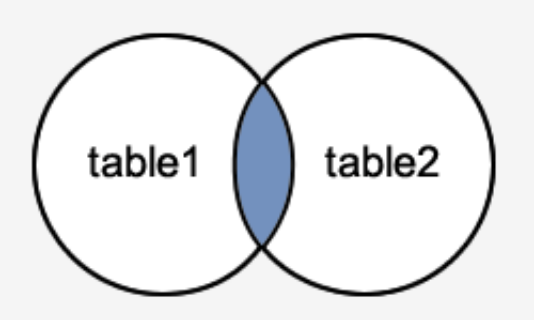

Figure 3.1: Inner Join

New Syntax:

```
 SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
INNER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

Old Syntax:

```
 SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers, orders
WHERE suppliers.supplier_id = orders.supplier_id;
```

(b) Classical format

```
 SELECT ...
FROM   dataset_one d1
,      dataset_two d2
WHERE  d1.column(s) = d2.column(s)
AND    ...
```

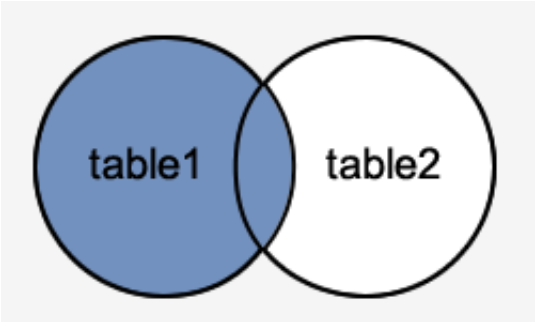

Figure 3.2: Left Outer Join

3.2.1.2 Outer Join

```
 The New syntax for the Oracle LEFT OUTER JOIN is:

 SELECT columns
FROM table1
LEFT [OUTER] JOIN table2
ON table1.column = table2.column;

 Example:

 SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
LEFT OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;


Example Old:

 SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers, orders
WHERE suppliers.supplier_id = orders.supplier_id(+);


Right Outer:
New:

SELECT orders.order_id, orders.order_date, suppliers.supplier_name
FROM suppliers
RIGHT OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;

Right Outer:
Old:
```

```
SELECT orders.order_id, orders.order_date, suppliers.supplier_name
FROM suppliers, orders
WHERE suppliers.supplier_id(+) = orders.supplier_id;
```

```
Full Outer :

SELECT columns
FROM table1
FULL [OUTER] JOIN table2
ON table1.column = table2.column;

New:

SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
FULL OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;

Old:
As a final note, it is worth mentioning that the FULL OUTER JOIN example above could
written in the old syntax without using a UNION query.
```

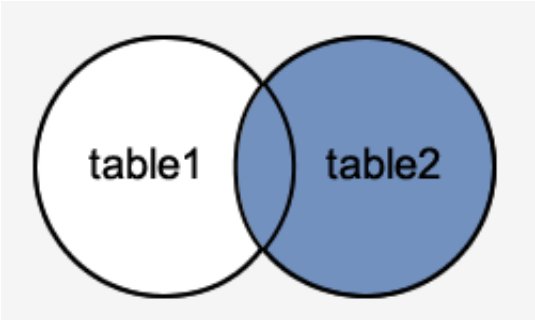

Figure 3.3: Right Outer Join

(a) Pre-Oracle9i Syntax for Outer Joins

```
 select B.Title, MAX(BC.ReturnedDate - BC.CheckoutDate)
"Most Days Out"
from BOOKSHELF_CHECKOUT BC, BOOKSHELF B
where BC.Title (+) = B.Title
group by B.Title;
```

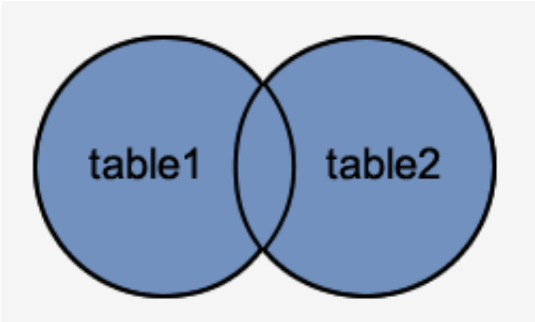

Figure 3.4: Full Outer Join

(b) Modern Syntax for Outer Joins

```
select B.Title, MAX(BC.ReturnedDate - BC.CheckoutDate)
"Most Days Out"
from BOOKSHELF_CHECKOUT BC right outer join BOOKSHELF B
on BC.Title = B.Title
group by B.Title;
```

Left and Right Outer: So, what is the difference between the right and left outer joins? The difference is simple in a left outer join, all of the rows from the left table will be displayed, regardless of whether there are any matching columns in the right table. In a right outer join, all of the rows from the right table will be displayed, regardless of whether there are any matching columns in the left table. Hopefully the example that we gave above help clarified this as well.

Chapter 4

Lecture (Lec 5 & 6) for Week 3

4.1 Built-in Functions

1. **LOWER and UPPER:** select City, LOWER(City), LOWER('City') from WEATHER;
2. **Concatenation** ||

```
select City||Country from LOCATION;

select CONCAT(City, Country) from LOCATION;

  2 are equivalent
```

3. **RPAD and LPAD:**

   The syntax:

```
LPAD( string1, paddedlength, [ pad_string ] )
```

   Parameters or Arguments:

   string1 is the string to pad characters to (the left-hand side).

   paddedlength is the number of characters to return. If the paddedlength is smaller than the original string, the LPAD function will *truncate the string to the size of paddedlength.*

   padstring is optional. This is the string that will be padded to the left-hand side of string1. If this parameter is *omitted*, the LPAD function will *pad spaces* to the left-side of string1.

   Example:

```
   LPAD('tech', 7);
Result: '   tech'

LPAD('tech', 2);
Result: 'te'

LPAD('tech', 8, '0');
Result: '0000tech'

LPAD('tech on the net', 15, 'z');
Result: 'tech on the net'

LPAD('tech on the net', 16, 'z');
Result: 'ztech on the net'
```

4. **LTRIM, RTRIM, and TRIM:** They trim off (removes) all specified characters from the left and right ends of strings.

   The syntax:

```
LTRIM( string1, [ trimstring ] )
```

   Parameters or Arguments:

   string1 is the string to trim the characters from the left-hand side.

   trimstring is the string that will be removed from the left-hand side of string1. If this parameter is *omitted*, the LTRIM function will remove *all leading spaces* from string1.

   Example:

```
  LTRIM('   tech')
Result: 'tech'

LTRIM('   tech', ' ')
Result: 'tech'

LTRIM('000123', '0')
Result: '123'

LTRIM('123123Tech', '123')
Result: 'Tech'
```

```
LTRIM('123123Tech123', '123')
Result: 'Tech123'

LTRIM('xyxzyyyTech', 'xyz')
Result: 'Tech'

LTRIM('6372Tech', '0123456789')
Result: 'Tech'
```

Last Example: In this example, every number combination from 0 to 9 has been listed in the trimstring parameter. By doing this, it does not matter the order that the numbers appear in string1, all leading numbers will be removed by the LTRIM function.

Combining: LTRIM and RTRIM

```
  (RTRIM('*#Hello*','*')

  LTRIM('*#Hello*','*#')

  Now combine them:

  LTRIM(RTRIM('*#Hello*','*'),'*#')
```

Using the TRIM Function: The preceding example showed how to combine two functionsa useful skill when dealing with string manipulation. *If you are trimming the exact same data from both the beginning and the end of the string, you can use the TRIM function in place of an LTRIM/RTRIM combination.*

```
  TRIM('   tech   ')
Result: 'tech'

Note: If you do not specify trim_character, the default value is
a blank space.


TRIM(' '  FROM  '   tech   ')
Result: 'tech'
```

```
TRIM(LEADING '0' FROM '000123')
Result: '123'

Note: If you specify LEADING, Oracle removes any leading characters equal
to trim_character.

TRIM(TRAILING '1' FROM 'Tech1')
Result: 'Tech'

Note: If you specify TRAILING, Oracle removes any trailing characters equal
to trim_character.


TRIM(BOTH '1' FROM '123Tech111')
Result: '23Tech'

Note: If you specify BOTH or none of the three, Oracle removes
leading and trailing characters equal to trim_character.
```

5. **LENGTH** LENGTH tells you how long a string ishow many characters it has in it, including letters, spaces, and anything else.
6. **SUBSTR:**

   The syntax:

   ```
   SUBSTR( string, start_position, [ length ] )
   ```

   Parameters or Arguments:

   string is the source string.

   startposition is the position for extraction. The first position in the string is always 1.

   length is optional. It is the number of characters to extract. If this parameter is omitted, the SUBSTR function will return the entire string.

   Note: If startposition is a positive number, then the SUBSTR function starts from the beginning of the string.

   If startposition is a negative number, then the SUBSTR function starts from the end of the string and counts backwards.

   Example:

```
  SUBSTR('This is a test', 6, 2)
Result: 'is'

SUBSTR('This is a test', 6)
Result: 'is a test'

SUBSTR('TechOnTheNet', 1, 4)
Result: 'Tech'

SUBSTR('TechOnTheNet', -3, 3)
Result: 'Net'

SUBSTR('TechOnTheNet', -6, 3)
Result: 'The'

SUBSTR('TechOnTheNet', -8, 2)
Result: 'On'
```

7. **INSTR:**

Syntax:

```
INSTR( string, substring [, start_position [,  nth_appearance ] ] )
```

Parameters or Arguments:

string is the string to search.

substring is the substring to search for in string.

startposition is the position in string where the search will start. This argument is optional. If omitted, it defaults to 1. The first position in the string is 1. If the startposition is *negative*, the INSTR function counts back startposition number of characters from the end of string and then searches towards the beginning of string.

nthappearance is the nth appearance of substring. This is optional. If omitted, it defaults to 1.

Example:

```
  INSTR('Tech on the net', 'e')
Result: 2   (the first occurrence of 'e')

INSTR('Tech on the net', 'e', 1, 1)
Result: 2   (the first occurrence of 'e')
```

```
INSTR('Tech on the net', 'e', 1, 2)
Result: 11  (the second occurrence of 'e')

INSTR('Tech on the net', 'e', 1, 3)
Result: 14  (the third occurrence of 'e')
```

8. **ASCII and CHR:**

   Example:

```
select CHR(70)||CHR(83)||CHR(79)||CHR(85)||CHR(71) R
as ChrValues
from DUAL;

OUTPUT:

R
-----
FSOUG
```

   The ASCII function performs the reverse operationbut if you pass it a string, only the first character of the string will be acted upon:

```
 select ASCII('FSOUG') from DUAL;
ASCII('FSOUG')
--------------
70
```

4.2 Regular Expression

Mainly used for advanced searching. Oracle 10g introduced support support for regular expressions in SQL and PL/SQL.

Example 1 : REGEXP_SUBSTR

```
 DROP TABLE t1;
CREATE TABLE t1 (
  data VARCHAR2(50)
);

INSERT INTO t1 VALUES ('FALL 2014');
INSERT INTO t1 VALUES ('2014 CODE-B');
INSERT INTO t1 VALUES ('CODE-A 2014 CODE-D');
```

```
INSERT INTO t1 VALUES ('ADSHLHSALK');
INSERT INTO t1 VALUES ('FALL 2004');
COMMIT;

SELECT * FROM t1;

DATA
--------------------------------------------------------------------------------
FALL 2014
2014 CODE-B
CODE-A 2014 CODE-D
ADSHLHSALK
FALL 2004

5 rows selected.

SQL>
```

Objective: If we needed to return rows containing a specific year we could use the LIKE operator (WHERE data LIKE '%2014%'), but how do we return rows using a comparison (¡, ¡=, ¿, ¿=, ¡¿)?

One Solution is: to pull out the 4 figure year and convert it to a number, so we don't accidentally do an ASCII comparison.

That is pretty easy using regular expressions.

Solution using Regular Expression: We can identify digits using the `"\d"` or `"[0-9]"` operators. We want a group of four of them, which is represented by the `"{4}"` operator. So our regular expression will be `"\d{4}"` or "[0-9]4". The `REGEXP_SUBSTR` function returns the string matching the regular expression, so that can be used to extract the text of interest. We then just need to convert it to a number and perform our comparison.

Solution Query:

```
 SELECT *
FROM   t1
WHERE  TO_NUMBER(REGEXP_SUBSTR(data, '\d{4}')) >= 2014;

DATA
----------------------------------------------------------
FALL 2014
2014 CODE-B
CODE-A 2014 CODE-D

3 rows selected.

SQL>
```

4.3 GROUP BY, GROUP BY HAVING. ROLLUP, CUBE and DECODE

1. **GROUP BY:** An aggregate function takes multiple rows of data returned by a query and aggregates them into a single result row.

2. **GROUP BY HAVING:** The SQL HAVING Clause is used in combination with the GROUP BY Clause to restrict the groups of returned rows to only those whose the condition is TRUE.

   Example:

```
SELECT department, SUM(sales) AS "Total sales"
FROM order_details
GROUP BY department
HAVING SUM(sales) > 1000;
```

3. **ROLLUP:** ROLLUP enables a SELECT statement to *calculate multiple levels of subtotals across a specified group of dimensions.* It also calculates a grand total. ROLLUP is a simple extension to the GROUP BY clause, so its syntax is extremely easy to use. The ROLLUP extension is highly efficient, adding minimal overhead to a query.

   Syntax:

```
SELECT ... GROUP BY
  ROLLUP(grouping_column_reference_list)
```

   Details: ROLLUP will create subtotals at *n+1* levels, where n is the number of grouping columns. For instance, if a query specifies ROLLUP on grouping columns of Time, Region, and Department (n=3), the result set will include rows at four aggregation levels.

   Example:

```
select Dept,Desig,count(*)Total
from emp
group by rollup(Dept,Desig)
order by Dept,Desig;
```

   Sample Output:

```
   DEPT DESIG          TOTAL
---------- --------------------  ----------
10 Asst Manager  2
10 Manager 3
10 5
20 Asst Manager  3
20 Manager 2
20 5
30 Asst Manager  1
30 Manager 2
30 3
13
```

You should learn other possible options on ROLLUP.

When to Use ROLLUP?

(a) It is very helpful for subtotaling along a hierarchical dimension such as time or geography. For instance, a query could specify a ROLLUP of year/month/day or country/state/city.

(b) It simplifies and speeds the population and maintenance of summary tables. Data warehouse administrators may want to make extensive use of it. Note that population of summary tables is even faster if the ROLLUP query executes in parallel.

4. **CUBE:** Why CUBE? It has a strong relationship with Data Warehousing. Lets have a look on it.(Slides).

   CUBE enables a SELECT statement to calculate subtotals for all possible combinations of a group of dimensions. It also calculates a grand total. This is the set of information typically needed for all cross-tabular reports, so CUBE can calculate a cross-tabular report with a single SELECT statement. Like ROLLUP, CUBE is a simple extension to the GROUP BY clause.

   Syntax:

```
SELECT ...  GROUP BY
  CUBE (grouping_column_reference_list)
```

Example:

```
 select Dept,Desig,count(*)Total
from emp
```

```
group by cube(Dept,Desig)
order by Dept,Desig;
```

Sample Output:

```
 DEPT  DESIG      TOTAL
---------- -------------------- ----------
10 Asst Manager   2
10 Manager  3
10  5
20 Asst Manager   3
20 Manager  2
20  5
30 Asst Manager   1
30 Manager  2
30  3
   Asst Manager   6
   Manager  7

      DEPT DESIG      TOTAL
---------- -------------------- ----------
13
```

When to Use CUBE?

(a) Use CUBE in any situation requiring cross-tabular reports.

(b) CUBE is especially valuable in queries that use columns from multiple dimensions rather than columns representing different levels of a single dimension. For instance, a commonly requested cross-tabulation might need subtotals for all the combinations of month/state/product.

4.4 Date Operations

DATE is an Oracle datatype, just as VARCHAR2 and NUMBER are, and it has its own unique properties. The DATE datatype is stored in a special internal Oracle format that includes not just the month, day, and year, but also the hour, minute, and second. TIMESTAMP datatype stores fractional seconds.

```
 select SysDate from DUAL;

 SYSDATE
---------
```

```
28-FEB-08


select Systimestamp from dual;
SYSTIMESTAMP
---------------------------------------------------------------------------
27-JAN-15 11:29:35.765787 +06:00
```

Fundamental Operations: (Chapter 10 for details)

1. **TO_DATE and TO_CHAR Formatting**

   TO_CHAR

```
  select max(hire_date)MX,
   TO_CHAR(max(hire_date),'DD-MM-YYYY') DateConv
    from employees;

    Sample Output:

    MX    DATECONV
------------------ ----------
21-APR-08    21-04-2008
```

   Month Part:

```
 Month Formats:
Format  Result
Month    August
Mon      Aug
```

   Day Part:

   The day of the month is produced by the DD in the format. A suffix of th on DD tells Oracle to use ordinal suffixes, such as TH, RD, and ND with the number. In this instance, the suffixes are also case sensitive, but their case is set by the DD, not the th:

```
 Format Result
DDth or DDTH 11TH
Ddth or DdTH 11Th
Ddth or ddTH 11th
```

For part of one hour: `'HH:MI:SS'`

2. **Adding and subtracting Month** You can fast forward or go back in term of number of Months. Use `ADD_MONTHS`

   Example:

```
  select hire_date DT, ADD_MONTHS(hire_date,47)Date_After47Moth
from employees
where department_id=50;
```

3. **Difference Between Two Dates** Date1 - Date1 gives the result in number of days.

   Example:

```
  select max(hire_date)MX
    from employees;


select min(hire_date)MN
    from employees;

select (max(hire_date)-min(hire_date))/365 Year
    from employees;
```

4. **Date inside where clause**

```
  --bETWEEN CLAUSE--

select hire_date
from employees
where hire_date between '01-JAN-06' AND '01-FEB-09';


--MORE SECURE WAY TO IT--

select hire_date
from employees
where hire_date between TO_DATE('01-JAN-06','DD-MON-YY') AND TO_DATE('01-FEB-09'
```

EXTRACT Function: You can use the EXTRACT function in place of the TO_CHAR function when you are selecting portions of date valuessuch as just the month or day from a date. The EXTRACT functions syntax is:

```
 EXTRACT
( { { YEAR
| MONTH
| DAY
| HOUR
| MINUTE
| SECOND
}
| { TIMEZONE_HOUR
| TIMEZONE_MINUTE
}
| { TIMEZONE_REGION
| TIMEZONE_ABBR
}
}
FROM { datetime_value_expression | interval_value_expression }
)
```

Example:

```
 select First_Name, EXTRACT( Month from hire_date) M
from employees;
```

Chapter 5

Lecture (Lec 6 & 7) for Week 4

Introduction to PL SQL.

A few points about PL SQL:

- PL/SQL lets you write code once and deploy it in the database nearest the data. PL/SQL can simplify application development, optimize execution, and improve resource utilization in the database.
- The language is a case-insensitive programming language, like SQL.
- **History:** PL/SQL was developed by modeling concepts of structured programming, static data typing, modularity, exception management, and parallel (concurrent) processing found in the *Ada programming language*. The Ada programming language, developed for the United States Department of Defense, *was designed to support military real-time and safety-critical embedded systems, such as those in airplanes and missiles.* The Ada programming language *borrowed significant syntax from the Pascal programming language*, including the assignment and comparison operators and the single-quote delimiters.

5.0.1 Block

PL/SQL is a blocked programming language. Program units can be *named or unnamed* blocks. Unnamed blocks are known as *anonymous* blocks and are labeled so throughout the book. The PL/SQL coding style differs from that of the C, C++, and Java programming languages. For example, curly braces do not delimit blocks in PL/SQL.

Anonymous-block programs are effective in some situations. You typically use anonymous blocks when building scripts to seed data or perform one-time processing activities. They are also effective when you want to nest activity in another PL/SQL blocks execution section. The basic anonymous-block structure must contain an execution section. You can also put optional declaration and exception sections in anonymous blocks. The following illustrates an anonymous-block prototype:

```
 [DECLARE]
declaration_statements
BEGIN
```

```
execution_statements
[EXCEPTION]
exception_handling_statements
END;
/
```

Lets write out first block that will do nothing:

```
 BEGIN
NULL;
END;
/
```

Hello World:

```
 SET SERVEROUTPUT ON SIZE 1000000
BEGIN
dbms_output.put_line('Hello World.');
END;
/
```

scanf:

```
 --scanf--

DECLARE
my_var VARCHAR2(30);
BEGIN
my_var := '&i';
dbms_output.put_line('Hello '|| my_var );
END;
```

5.0.2 Variables, Assignments, and Operators

Delimiters: Lexical delimiters are symbols or symbol sets.

For complete set see table 3-1. Here are some important symbols frequently used in pl sql programming.

- **:= Assignment**

  The assignment operator is a colon immediately followed by an equal symbol. It is the only assignment operator in the language. You assign a right operand to a left operand, like a := b + c;

- **. Association**

  The component selector is a period, and it glues references together, for example, a schema and a table, a package and a function, or an object and a member method. Component selectors are also used to link cursors and cursor attributes (columns). The following are some prototype examples:

```
schema_name.table_name
package_name.function_name
object_name.member_method_name
cursor_name.cursor_attribute
object_name.nested_object_name.object_attribute
These are referenced in subsequent chapters throughout this book.
```

- **@ Association.** The remote access indicator lets you access a remote database through database links.
- **= Comparison.**
- **¡¿ != Comparison** Not equal to.
- **– Delimiter** In-line comment.
- **/* Delimiter */** Multi-line comment.
- **" Delimiter.**

```
 quoted identifier delimiter is a double quote. It lets you access tables create
in case-sensitive fashion from the database catalog. This is required when you ha
created database catalog objects in case-sensitive fashion. You can do this from
Oracle 10g forward.
For example, you create a case-sensitive table or column by using quoted identif
delimiters:
CREATE TABLE "Demo"
("Demo_ID" NUMBER
, demo_value VARCHAR2(10));
You insert a row by using the following quote-delimited syntax:
INSERT INTO "Demo1" VALUES
(1,'One Line ONLY.');#
```

5.1 Data-types

See Figure 3.5 for details.

5.1.1 Scalar Datatypes

Scalar datatypes use the following prototype inside the declaration block of your programs:

```
variable_name datatype [NOT NULL] [:= literal_value];
```

1. **Boolean.** The BOOLEAN datatype has three possible values: TRUE, FALSE, and NULL.

   ```
   var1 BOOLEAN; -- Implicitly assigned a null value.
   var2 BOOLEAN NOT NULL := TRUE; -- Explicitly assigned a TRUE value.
   var3 BOOLEAN NOT NULL := FALSE; -- Explicitly assigned a FALSE value.
   ```

   There is little need to subtype a BOOLEAN datatype, but you can do it. The subtyping syntax is:

   ```
   SUBTYPE booked IS BOOLEAN;
   ```

   This creates a subtype BOOKED that is an unconstrained BOOLEAN datatype. You may find this useful when you need a second name for a BOOLEAN datatype, but generally subtyping a Boolean is not very useful.

2. **Characters and Strings.**

   The VARCHAR2 datatype stores variable-length character strings. When you create a table with a VARCHAR2 column, you specify a maximum string length (in bytes or characters) between 1 and 4000 bytes for the VARCHAR2 column. For each row, Oracle Database stores each value in the column as a variable-length field unless a value exceeds the column's maximum length, in which case Oracle Database returns an error. Using VARCHAR2 and VARCHAR saves on space used by the table.

   Difference between Char and Varchar2: The following program illustrates the memory allocation differences between the CHAR and VARCHAR2 datatypes:

   ```
   DECLARE
   c CHAR(32767) := 'hello';
   v VARCHAR2(32767) := 'hello';
   BEGIN
   dbms_output.put_line('c is ['||LENGTH(c)||']');
   dbms_output.put_line('v is ['||LENGTH(v)||']');
   v := v || ' ';
   dbms_output.put_line('v is ['||LENGTH(v)||']'); END;
   /
   ```

Output:

```
c is [32767]
v is [1]
v is [2]
```

The output shows that a CHAR variable sets the *allocated memory size when defined*. The allocated memory can exceed what is required to manage the value in the variable. The output also shows that the VARCHAR2 variable *dynamically allocates* only the required memory to host its value.

More on VARCHAR2: Globalization support allows the use of various character sets for the character datatypes. Globalization support lets you process *single-byte* and *multibyte* character data and convert between character sets. Client sessions can use client character sets that are different from the database character set.

The length semantics of character datatypes can be measured in *bytes* or *characters*.

- **Byte semantics** treat strings as a sequence of bytes. This is the default for character datatypes.
- **Character semantics** treat strings as a sequence of characters. A character is technically a codepoint of the database character set.

For *single byte character* sets, columns defined in character semantics are basically the same as those defined in byte semantics. *Character semantics are useful for defining varying-width multibyte strings*; it reduces the complexity when defining the actual length requirements for data storage. *For example*, in a Unicode database (UTF8), you must define a VARCHAR2 column that can store up to five Chinese characters together with five English characters. In byte semantics, this would require (5*3 bytes) + (1*5 bytes) = 20 bytes; in character semantics, the column would require 10 characters.

Lets look at the example below:

```
 var1 VARCHAR2(100); -- Implicitly sized at 100 byte.
var2 VARCHAR2(100 BYTE); -- Explicitly sized at 100 byte.
var3 VARCHAR2(100 CHAR); -- Explicitly sized at 100 character.
```

When you use character space allocation, the maximum size changes, depending on the character set of your database. Some character sets use *two or three bytes to store characters*. You divide 32,767 by the number of bytes required per character, which means the maximum for a VARCHAR2 is 16,383 for a two-byte character set and 10,922 for a three-byte character set.

3. **Date:** Already discussed. See interval and TIMESTAMP.

4. **NUMBER.** It can store numbers in the range of 1.0E-130 (1 times 10 raised to the negative 130th power) to 1.0E126 (1 times 10 raised to the 126th power).

   The following is the prototype for declaring a fixed-point NUMBER datatype:

   ```
   NUMBER[(precision, [scale])] [NOT NULL]
   ```

5. **Large Objects (LOBs)** (Details will be covered later.) Large objects (LOBs) provide you with four datatypes: BFILE, BLOB, CLOB, and NCLOB. The BFILE is a datatype that points to an external file, which limits its maximum size to 4 gigabytes. The BLOB, CLOB and NCLOB are internally managed types, and their maximum size is 8 to 128 terabytes, depending on the`db_block_size` parameter value.

5.1.2 Composite Datatypes

There are two composite generalized datatypes: records and collections. (Will be covered next).

Just have look at the following example for Record.

```
 DECLARE
TYPE demo_record_type IS RECORD
( id NUMBER DEFAULT 1
, value VARCHAR2(10) := 'One');
demo DEMO_RECORD_TYPE;
BEGIN
dbms_output.put_line('['||demo.id||']['||demo.value||']');
END;
/
```

Chapter 6

Lecture (Lec 8 & 9) for Week 5

6.1 Control Structure

6.1.1 Conditional statements

1. **IF Statements.** IF statements evaluate a condition. The condition can be any comparison expression, or set of comparison expressions that evaluates to a logical true or false.

   Example:

```
DECLARE
X NUMBER;
BEGIN
X:=10;
IF (X = 0) THEN
 dbms_output.put_line('The value of x is 0 ');
ELSIF(X between 1 and 10) THEN
dbms_output.put_line('The value of x is between 1 and 10 ');
ELSE
dbms_output.put_line('The value of x is greater than 10 ');
END IF;
END;
```

   Example in Real Scenario:

```
  DECLARE
CGPA NUMBER;
X NUMBER :=034403;
BEGIN

SELECT MAX(CGPA) INTO CGPA
```

```
FROM STUDENTS
WHERE ID=X;

IF (CGPA >3.78) THEN
 dbms_output.put_line('Brilliant');
ELSIF(CGPA between 3.5 and 3.78) THEN
dbms_output.put_line('Mid Level');
ELSE
dbms_output.put_line('Poor');
END IF;
END;
```

2. **Simple CASE Statements.** The simple CASE statement sets a selector that is any PL/SQL datatype except a BLOB, BFILE, or composite type.

   Example:

```
  DECLARE
selector NUMBER := 1;
BEGIN
CASE selector
WHEN 0 THEN
dbms_output.put_line('Case 0!');
WHEN 1 THEN
dbms_output.put_line('Case 1!');
ELSE
dbms_output.put_line('No match!');
END CASE;
END;
/
```

3. **Searched CASE Statements**

   Lets look at the example: (this is one step advancement in SQL).

```
  SELECT name, ID,
(CASE
  WHEN salary < 1000 THEN 'Low'
  WHEN salary BETWEEN 1000 AND 3000 THEN 'Medium'
  WHEN salary > 3000 THEN 'High'
  ELSE 'N/A'
      END) salary
    FROM emp
```

```
    ORDER BY name;
```

6.1.2 LOOP

1. **LOOP and EXIT Statements** Simple loops are explicit block structures. A simple loop starts and ends with the LOOP reserved word. An EXIT statement or an EXIT WHEN statement is required to break the loop.

   Example: LOOP EXIT:

```
  DECLARE
   x number := 10;
BEGIN
   LOOP
      dbms_output.put_line(x);
      x := x + 10;
      IF x > 50 THEN
         exit;
      END IF;
   END LOOP;
   -- after exit, control resumes here
   dbms_output.put_line('After Exit x is: ' || x);
END;
/

SQL> /
10
20
30
40
50
After Exit x is: 60

PL/SQL procedure successfully completed.
```

   Example: LOOP EXIT WHEN

```
  DECLARE
   x number := 10;
BEGIN
   LOOP
      dbms_output.put_line(x);
```

```
      x := x + 10;
      exit WHEN x > 50;
   END LOOP;
   -- after exit, control resumes here
   dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

2. **FOR Loop.** A FOR LOOP is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

   Syntax:

```
  FOR counter IN initial_value .. final_value LOOP
   sequence_of_statements;
END LOOP;
```

   Few Points:

   - After the body of the for loop executes, the value of the counter variable is increased or decreased.
   - The `initial_value` and `final_value` of the loop variable or counter can be literals, variables, or expressions but must evaluate to numbers. Otherwise, `PL/SQL` raises the predefined exception `VALUE_ERROR`.
   - The `initial_value` need not to be 1; however, the loop counter increment (or decrement) must be 1.
   - `PL/SQL` allows determine the loop range dynamically at run time.

   Example:

```
 DECLARE
   a number(2);
BEGIN
   FOR a in 10 .. 20 LOOP
       dbms_output.put_line('value of a: ' || a);
  END LOOP;
END;
/

OUTPUT:
```

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20

PL/SQL procedure successfully completed.
```

Reverse FOR LOOP: By default, iteration proceeds from the initial value to the final value, generally upward from the lower bound to the higher bound. You can reverse this order by using the REVERSE keyword. In such case, iteration proceeds the other way. After each iteration, the loop counter is decremented.

However, you must write the range bounds *in ascending (not descending) order.*

Example:

```
  DECLARE
   a number(2) ;
BEGIN
   FOR a IN REVERSE 10 .. 20 LOOP
      dbms_output.put_line('value of a: ' || a);
   END LOOP;
END;
/

OUTPUT:

value of a: 20
value of a: 19
value of a: 18
value of a: 17
value of a: 16
value of a: 15
value of a: 14
value of a: 13
value of a: 12
```

```
value of a: 11
value of a: 10

PL/SQL procedure successfully completed.
```

3. **WHILE Loop.** A WHILE LOOP statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

   Syntax:

```
 WHILE condition LOOP
   sequence_of_statements
END LOOP;
```

   Example:

```
 DECLARE
   a number(2) := 10;
BEGIN
   WHILE a < 20 LOOP
      dbms_output.put_line('value of a: ' || a);
      a := a + 1;
   END LOOP;
END;
/

OUTPUT:

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

PL/SQL procedure successfully completed.
```

   Example:

```
   DECLARE
   a number(2) := 10;
BEGIN
   WHILE a < 20 LOOP
      dbms_output.put_line('value of a: ' || a);
      a := a + 1;
   END LOOP;
END;
/
```

4. **Cursor.** Will be covered after function and procedure.

Chapter 7

Lecture (Lec 10 & 11 for Week 6

7.1 Functions and Procedures

Motivation for Functions and Procedures: **Modular Code.** Modularization is the process by which you break up large blocks of code into smaller pieces (modules) that can be called by other modules. Modularization of code is analogous to normalization of data, with many of the same benefits and a few additional advantages. With modularization, your code becomes:

- **More reusable.** By breaking up a large program or entire application into individual components that plug-and-play together, you will usually find that many modules are used by more than one other program in your current application. Designed properly, these utility programs could even be of use in other applications!
- **More manageable.** Which would you rather debug: a 1,000-line program or five individual 200-line programs that call each other as needed? Our minds work better when we can focus on smaller tasks. You can also test and debug on a per-program scale (called unit testing) before individual modules are combined for a more complicated integration test.
- **More readable.** Modules have names, and names describe behavior. The more you move or hide your code behind a programmatic interface, the easier it is to read and understand what that program is doing. Modularization helps you focus on the big picture rather than on the individual executable statements. You might even end up with that most elusive kind of software: self-documenting code.
- **More reliable.** The code you produce will have fewer errors. The errors you do find will be easier to fix because they will be isolated within a module. In addition, your code will be easier to maintain because there is less of it and it is more readable.

Forms of Modularization: PL/SQL offers the following structures that modularize your code in different ways:

- **Procedure.** A program that performs one or more actions and is called as an executable PL/SQL statement. You can pass information into and out of a procedure through its parameter list.

- **Function.** A program that returns data through its RETURN clause, and is used just like a PL/SQL expression. You can pass information into a function through its parameter list.
- **Database trigger.** A set of commands that *are triggered to execute (e.g., log in, modify a row in a table, execute a DDL statement) when an event occurs* in the database.
- **Package.** A named collection of procedures, functions, types, and variables. A package is not really a module (its more of a meta-module), but it is so closely related that.

7.1.1 Procedures and Function

Procedure:A procedure is a module that performs one or more actions. Because a procedure call is a standalone executable statement in PL/SQL, a PL/SQL block could consist of nothing more than a single call to a procedure. Procedures are key building blocks of modular code, allowing you to both consolidate and reuse your program logic.

Syntax:

```
   [CREATE [OR REPLACE]]
PROCEDURE procedure_name[(parameter[, parameter]...)]
   [AUTHID {DEFINER | CURRENT_USER}] {IS | AS}
   [PRAGMA AUTONOMOUS_TRANSACTION;]
   [local declarations]
BEGIN
   executable statements
[EXCEPTION
   exception handlers]
END [name];
```

Function: A function is a module that returns data through its RETURN clause, rather than in an OUT or IN OUT argument (We will see what it means). Unlike a procedure call, which is a standalone executable statement, a call to a function can exist only as part of an executable statement, such as an element in an expression or the value assigned as the default in a declaration of a variable.

Syntax

```
   [CREATE [OR REPLACE]]
FUNCTION function_name[(parameter[, parameter]...)]
RETURN RETURN_TYPE
   [AUTHID {DEFINER | CURRENT_USER}] {IS | AS}
   [PRAGMA AUTONOMOUS_TRANSACTION;]
   [local declarations]
BEGIN
```

```
   executable statements
[EXCEPTION
   exception handlers]

   RETURN STATEMENT;
END [name];
```

Common in Procedure and Function:

- Has a name.
- Can take parameters, and can return values.
- Is stored in the data dictionary.
- Can be called by many users.

Note: The term *stored procedure* is sometimes used generically for both stored procedures and stored functions. The only difference between procedures and functions is that functions always return a single value to the caller, while procedures do not return a value to the caller.

7.1.1.1 Parameters for Procedures and Functions

Parameter modes define the behavior of formal parameters. The three parameter modes, **IN (the default), OUT, and IN OUT**, can be used with any subprogram. However, avoid using the OUT and IN OUT modes with functions.

- **IN:** The value of the actual parameter is passed into the procedure when the procedure is invoked. Inside the procedure, the formal parameter acts like a PL/SQL constantit is considered *read-only* and cannot be changed.
- **OUT:** Any value the actual parameter has when the procedure is called is ignored. Inside the procedure, *the formal parameter acts like an uninitialized PL/SQL variable and thus has a value of NULL.* It can be read from and written to.
- **INOUT:** This mode is a combination of IN and OUT.

Formal Parameters: When you declare a parameter, however, you must leave out the constraining part of the declaration.

Datatype indicator. 1. Using the `%TYPE Attribute` 2. Using the `%ROWTYPE Attribute`

```
   DECLARE
   emprec    employees_temp%ROWTYPE;
   id emp.id%TYPE;
BEGIN
   emprec.empid := NULL;  -- this works, null constraint is not inherited
```

```
-- emprec.empid := 10000002; -- invalid, number precision too large
   emprec.deptid := 50; -- this works, check constraint is not inherited
-- the default value is not inherited in the following
   DBMS_OUTPUT.PUT_LINE('emprec.deptname: ' || emprec.deptname);
END;
/
```

Example: Procedure with IN and OUT parameter

```
   CREATE OR REPLACE PROCEDURE PROC1(ID IN NUMBER, SALARY OUT NUMBER)
AS BEGIN
SELECT MAX(SALARY) INTO SALARY
FROM EMP
WHERE ID = ID;
END;
/
---now call it from annonymous block--
DECLARE
amount NUMBER;
BEGIN
PROC1(101, amount);
dbms_output.put_line(amount);
END;
```

Parameter Positions:

- **Positional Notation:** You use positional notation to call the function as follows:

```
  BEGIN
dbms_output.put_line(add_three_numbers(3,4,5));
END;
```

- **Named Notation:** You call the function using named notation by:

```
  BEGIN
dbms_output.put_line(add_three_numbers(c => 4,b => 5,c => 3));
END;
```

- **Mixed Notation:** You call the function by a mix of both positional and named notation by:

```
  BEGIN
dbms_output.put_line(add_three_numbers(3,c => 4,b => 5));
END;
```

Note: There is a restriction on mixed notation. All positional notation actual parameters must occur first and in the same order as they are defined by the function signature.

Chapter 8

Lecture (Lec 11 & 12 for Week 7

8.1 Cursor

In response to any DML statement the database creates a memory area, known as *context area*, for processing an SQL statement, which contains all information needed for processing the statement, for example, number of rows processed, etc.

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the *active set*.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

- Implicit cursors.
- Explicit cursors.

8.1.1 Implicit Cursor

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor, which always has the attributes like `%FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT`.

Attribute	Description
%FOUND	Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
%NOTFOUND	The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
%ISOPEN	Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
%ROWCOUNT	Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Example: Implicit Cursor

```
 DECLARE
   total_rows number(2);
BEGIN
   UPDATE emp
   SET salary = salary + 500;
   IF sql%notfound THEN
      dbms_output.put_line('no customers selected');
   ELSIF sql%found THEN
      total_rows := sql%rowcount;
      dbms_output.put_line( total_rows || ' customers selected ');
   END IF;
END;
/
```

8.1.2 Explicit Cursors

Explicit cursors are programmer defined cursors for gaining more control over the context area. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns one or more rows.

```
CURSOR cursor_name IS select_statement;
```

4 Steps for Cursors:

1. Declaring the cursor for initializing in the memory
2. Opening the cursor for allocating memory
3. Fetching the cursor for retrieving data
4. Closing the cursor to release allocated memory

Similar to typical file operation.

Example:

```
DECLARE
   c_id customers.id%type;
   c_name customers.name%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
      FETCH c_customers into c_id, c_name, c_addr;
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
   CLOSE c_customers;
END;
/
```

CURSOR FOR Loop: The cursor FOR loop is an elegant and natural extension of the numeric FOR loop in PL/SQL. With a numeric FOR loop, the body of the loop executes once for every integer value between the low and high values specified in the range. With a cursor FOR loop, the body of the loop is executed for each row returned by the query.

Syntax

```
  FOR record_index in cursor_name
LOOP
   {...statements...}
END LOOP;
```

Example

```
 CREATE OR REPLACE Function TotalIncome
   ( name_in IN varchar2 )
   RETURN varchar2
IS
   total_val number(6);

   cursor c1 is
     SELECT monthly_income
     FROM employees
     WHERE name = name_in;

BEGIN

   total_val := 0;

   FOR employee_rec in c1
   LOOP
      total_val := total_val + employee_rec.monthly_income;
   END LOOP;

   RETURN total_val;

END;
```

8.1.2.1 Variables in Explicit Cursor Queries

An explicit cursor query can reference any variable in its scope. When you open an explicit cursor, PL/SQL evaluates any variables in the query and uses those values when identifying the result set. Changing the values of the variables later does not change the result set.

In the following Example, the explicit cursor query references the variable factor. When the cursor opens, factor has the value 2. Therefore, `sal_multiple` is always 2 times sal, despite that factor is incremented after every fetch.

```
 DECLARE
  sal           employees.salary%TYPE;
  sal_multiple  employees.salary%TYPE;
  factor        INTEGER := 2;

  CURSOR c1 IS
    SELECT salary, salary*factor FROM employees
    WHERE job_id LIKE 'AD_%';

BEGIN
```

```
  OPEN c1;  -- PL/SQL evaluates factor

  LOOP
    FETCH c1 INTO sal, sal_multiple;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
    factor := factor + 1;  -- Does not affect sal_multiple
  END LOOP;

  CLOSE c1;
END;
/


Result:

factor = 2
sal          = 4451
sal_multiple = 8902
factor = 3
sal          = 26460
sal_multiple = 52920
factor = 4
sal          = 18742.5
sal_multiple = 37485
factor = 5
sal          = 18742.5
sal_multiple = 37485
```

8.1.2.2 Explicit Cursors that Accept Parameters

You can create an explicit cursor that has formal parameters, and then pass different actual parameters to the cursor each time you open it. In the cursor query, you can use a formal cursor parameter anywhere that you can use a constant. Outside the cursor query, you cannot reference formal cursor parameters.

Following Example creates an explicit cursor whose two formal parameters represent a job and its maximum salary. When opened with a specified job and maximum salary, the cursor query selects the employees with that job who are overpaid (for each such employee, the query selects the first and last name and amount overpaid). Next, the example creates a procedure that prints the cursor query result set. Finally, the example opens the cursor with one set of actual parameters, prints the result set, closes the cursor, opens the cursor with different actual parameters, prints the result set, and closes the

cursor.

```
 DECLARE
  CURSOR c (job VARCHAR2, max_sal NUMBER) IS
    SELECT last_name, first_name, (salary - max_sal) overpayment
    FROM employees
    WHERE job_id = job
    AND salary > max_sal
    ORDER BY salary;

  PROCEDURE print_overpaid IS
    last_name_   employees.last_name%TYPE;
    first_name_  employees.first_name%TYPE;
    overpayment_       employees.salary%TYPE;
  BEGIN
    LOOP
      FETCH c INTO last_name_, first_name_, overpayment_;
      EXIT WHEN c%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(last_name_ || ', ' || first_name_ ||
        ' (by ' || overpayment_ || ')');
    END LOOP;
  END print_overpaid;

BEGIN
  DBMS_OUTPUT.PUT_LINE('----------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Stock Clerks:');
  DBMS_OUTPUT.PUT_LINE('----------------------');
  OPEN c('ST_CLERK', 2500);
  print_overpaid;
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE('------------------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Sales Representatives:');
  DBMS_OUTPUT.PUT_LINE('------------------------------');
  OPEN c('SA_REP', 10000);
  print_overpaid;
  CLOSE c;
END;
/
```

Result:

```
----------------------
Overpaid Stock Clerks:
----------------------
Davies, Curtis (by 15.3)
Nayer, Julia (by 177.08)
Stiles, Stephen (by 177.08)
Bissot, Laura (by 338.87)
Mallin, Jason (by 338.87)
Rajs, Trenna (by 662.43)
Ladwig, Renske (by 824.21)
------------------------------
Overpaid Sales Representatives:
------------------------------
Fox, Tayler (by 80)
Tucker, Peter (by 500)
King, Janette (by 500)
Bloom, Harrison (by 500)
Vishney, Clara (by 1025)
Abel, Ellen (by 1550)
Ozer, Lisa (by 2075)

PL/SQL procedure successfully completed.
```

Examples: End of the day in bank. Annual salary increment for each employee of an organization.

Chapter 9

Lecture (Week 8 & 9 (opt)

9.1 Records

A record provides the means of defining a programming structure. It is similar to structure in C.

9.1.1 Defining Record

3 ways to do it.

1. Using the %ROWTYPE attribute.
2. Using explicit definition in the declaration section of a PL/SQL program.
3. Last, a record type may be defined as a database structure or object type.[will be covered later]

9.1.2 Defining Implicity with the %ROWTYPE Attribute

The %ROWTYPE attribute may be applied to or reference a PL/SQL cursor, table, object, or view in the database. It inherits the definition of a row for any of those objects. Within the row all fields are implicitly defined as the column data types of the table being referenced.

Example:

```
 DECLARE
-- Define a variable with an implicit record type.
individual individuals%ROWTYPE;
BEGIN
-- Initialize the field values for the record.
individual.individual_id := 1;
individual.first_name := 'John';
```

```
individual.middle_initial := 'D';
individual.last_name := 'Rockefeller';
-- Insert into the table.
INSERT
INTO individuals
VALUES
(individual.individual_id
,individual.first_name
,individual.middle_initial
,individual.last_name);
-- Commit the work.
COMMIT;
END;
/
```

9.1.3 Defining Record Types Explicitly as PL/SQL Structures

You may build a record explicitly by defining a record type in the declaration section of a PL/SQL program. You may use explicit variable typing or the attribute to define variables. Both styles are used in the example. The following program shows how to define a record type.

Example:

```
 DECLARE
-- Define a record type.
TYPE individual_record IS RECORD
(individual_id INTEGER
,first_name VARCHAR2(30 CHAR)
,middle_initial individuals.middle_initial%TYPE
,last_name VARCHAR2(30 CHAR));

-- Define a variable of the record type.
individual INDIVIDUAL_RECORD;

BEGIN
-- Initialize the field values for the record.
individual.individual_id := 2;
individual.first_name := 'John';
individual.middle_initial := 'P';
individual.last_name := 'Morgan';
-- Insert into the table.
INSERT
INTO individuals
VALUES
```

```
(individual.individual_id
,individual.first_name
,individual.middle_initial
,individual.last_name);
-- Commit the work.
COMMIT;
END;
/
```

Nested Records: Records can be nested. When you nest record types, you increase the complexity of the syntax to access record type elements. *Adding a layer to the dot notation enables this.*

Example:

```
 DECLARE
-- Define a record type.
TYPE individual_record IS RECORD
(individual_id INTEGER
,first_name VARCHAR2(30 CHAR)
,middle_initial VARCHAR2(1 CHAR)
,last_name VARCHAR2(30 CHAR));

-- Define a record type.
TYPE address_record IS RECORD
(address_id INTEGER
,individual_id INTEGER
,street_address1 VARCHAR2(30 CHAR)
,street_address2 VARCHAR2(30 CHAR)
,street_address3 VARCHAR2(30 CHAR)
,city VARCHAR2(20 CHAR)
,state VARCHAR2(20 CHAR)
,postal_code VARCHAR2(20 CHAR)
,country_code VARCHAR2(10 CHAR));

-- Define a record type of two user-defined record types.
TYPE individual_address_record IS RECORD
(individual INDIVIDUAL_RECORD
,address ADDRESS_RECORD);
-- Define a user-defined compound record type.
individual_address INDIVIDUAL_ADDRESS_RECORD;
BEGIN
-- Initialize the field values for the record.
individual_address.individual.individual_id := 3;
```

```
individual_address.individual.first_name := 'Ulysses';
individual_address.individual.middle_initial := 'S';
individual_address.individual.last_name := 'Grant';
-- Initialize the field values for the record.
individual_address.address.address_id := 1;
individual_address.address.individual_id := 3;
individual_address.address.street_address1 :=
'Riverside Park';
individual_address.address.street_address2 := '';
```

Chapter 10

Lecture for Week 10,11

10.1 Triggers

Before getting into triggers it is important to discuss few points about one topic:*views*. For information about views in details See: Orale 11g Complete Reference,Chapter 17

Views: Views are known as logical tables. They represent the data of one of more tables. A view derives its data from the tables on which it is based. These tables are called base tables. Views can be based on actual tables or another view also.

Views are very powerful and handy since they can be treated just like any other table but do not occupy the space of a table.

Creating Views: Suppose we have EMP and DEPT table. To see the empno, ename, sal, deptno, department name and location we have to give a join query like this.

```
select e.empno,e.ename,e.sal,e.deptno,d.dname,d.loc

    from emp e, dept d where e.deptno=d.deptno;
```

Update a View: If a view is based on a single underlying table, you can insert, update, or delete rows in the view. This will actually insert, update, or delete rows in the underlying table.

There are restrictions on your ability to do this, although the restrictions are quite sensible:

- You cannot insert if the underlying table has any NOT NULL columns that dont appear in the view.
- You cannot insert or update if any one of the views columns referenced in the insert or update contains functions or calculations.
- You cannot insert, update, or delete if the view contains group by, distinct, or a reference to the pseudo-column RowNum.

Stability of a View.
If the base table is dropped then the view becomes invalid.

```
create view RAIN_VIEW as
select City, Precipitation
from TROUBLE;

View created.

drop table TROUBLE;
select * from RAIN_VIEW;

*
ERROR at line 1:
ORA-04063: view "PRACTICE.RAIN_VIEW" has errors


create or replace view RAIN_VIEW as
select * from TROUBLE;

alter table TROUBLE add (Warning
VARCHAR2(20)
);
Table altered.


Despite the change to the views base table, the view is still valid,
but the Warning column is missing.
```

Creating a Read-Only View. Use the *with read only* clause.

```
create or replace view RAIN_READ_ONLY as
select * from TROUBLE
with read only;
```

Introduction to Database Triggers: Database triggers are specialized stored programs. As such, they are defined by very similar DDL rules. Likewise, triggers can call SQL statements and PL/SQL functions and procedures. You can choose to implement triggers in PL/SQL or Java.

Database triggers differ from stored functions and procedures because you cant call them directly. Database triggers are fired when a triggering event occurs in the database.

10.1.1 Data Definition Language triggers

These triggers fire when you create, change, or remove objects in a database schema. They are useful to control or monitor DDL statements.

table and trigger share the same name as they use two separate namespaces.

10.1.2 Events that work with DDL triggers

(For details See Table 10-1 of Book Oracle 11g PL/SQL Programming)
ALTER,CREATE, DROP, GRANT, RENAME, REVOKE.

10.1.3 Event Attribute Functions

These are system-defined event attribute functions. For instance:

```
ORA_DICT_OBJ_NAME
ORA_DICT_OBJ_OWNER
ORA_DICT_OBJ_TYPE
ORA_CLIENT_IP_ADDRESS
ORA_DATABASE_NAME
ORA_IS_ALTER_COLUMN
ORA_LOGIN_USER
SPACE_ERROR_INFO
ORA_SYSEVENT
```

How to use them: Just like any other function.
Example:

```
DECLARE
password VARCHAR2(60);
BEGIN
IF ora_dict_obj_type = 'USER' THEN
password := ora_des_encrypted_password;
END IF;
END;
```

```
DECLARE
ip_address VARCHAR2(11);
BEGIN
IF ora_sysevent = 'LOGON' THEN
ip_address := ora_client_ip_address;
```

```
END IF;
END;
```

10.1.4 Building DDL Triggers

Syntax:

```
 CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF} ddl_event ON {DATABASE | SCHEMA}
[WHEN (logical_expression)]
[DECLARE]
declaration_statements;
BEGIN
execution_statements;
END [trigger_name];
/
```

Few Points:

- **Schema trigger**: The keyword schema decides its type. A SCHEMA trigger is created on a schema and fires whenever the user who owns it is the current user and initiates the triggering event.

  Suppose that both user1 and user2 own schema triggers, and user1 invokes a DR unit owned by user2. Inside the DR unit, user2 is the current user. Therefore, if the DR unit initiates the triggering event of a schema trigger that user2 owns, then that trigger fires. However, if the DR unit initiates the triggering event of a schema trigger that user1 owns, then that trigger does not fire. [present a suitable example in this regard.]

- **DATABASE Triggers:** Use keyword DATABASE for it. A DATABASE trigger is created on the database and fires whenever any database user initiates the triggering event.

Example Suppose we want to log 3 types of DDL (CREATE,ALTER AND DELETE) for a particular user (schema).

Note: Event Attribute Functions have been used to get the necessary information.

```
 create or replace trigger ddl_trigger
    before create or alter or drop on SCHEMA
    begin
    dbms_output.put_line('Who did it?  '||ora_dict_obj_owner);
        dbms_output.put_line('What was the Operation?  ' ||ora_sysevent);
        dbms_output.put_line('On what?  ' ||ora_dict_obj_name);
        dbms_output.put_line('On type of object it was?  ' ||ora_dict_obj_type);
    end;
```

10.1.5 Data Manipulation Language Triggers

DML triggers can fire before or after INSERT, UPDATE, and DELETE statements.

Row-level and statement-level trigger: DML triggers can be statement- or row-level activities. Statement-level triggers fire and perform a statement or set of statements once no matter how many rows are affected by the DML event. Row-level triggers fire and perform a statement or set of statements for each row changed by a DML statement.

DML triggers Syntax:

```
 CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER}
{INSERT | UPDATE | UPDATE OF column1 [, column2 [, column(n+1)]] | DELETE}
ON table_name
[FOR EACH ROW]
[WHEN (logical_expression)]
[DECLARE]
[PRAGMA AUTONOMOUS_TRANSACTION;]
declaration_statements;
BEGIN
execution_statements;
END [trigger_name];
```

Note: The clause *[FOR EACH ROW]* specifies that it is row-level trigger.

Example:

```
 CREATE OR REPLACE TRIGGER demo_trigger_types1
  BEFORE DELETE OR INSERT OR UPDATE ON employees
  --FOR EACH ROW

--DECLARE

BEGIN

    dbms_output.put_line('Row Level Trigger Fires each time. ');

END;
```

```
 CREATE OR REPLACE TRIGGER demo_trigger_types1
  BEFORE DELETE OR INSERT OR UPDATE ON employees
  --FOR EACH ROW
```

```
--DECLARE

BEGIN

    dbms_output.put_line('Row Level Trigger Fires . ');

END;
```

10.1.6 OLD and NEW Pseudorecords

When a row-level trigger fires, the PL/SQL runtime system creates and populates the two pseudorecords OLD and NEW. They are called pseudorecords because they have some, but not all, of the properties of records.

For the row that the trigger is processing:

- For an INSERT trigger, OLD contains no values, and NEW contains the new values.
- For an UPDATE trigger, OLD contains the old values, and NEW contains the new values.
- For a DELETE trigger, OLD contains the old values, and NEW contains no values.

Example:

```
  Example 2: Need to discuss OLD and NEW

CREATE OR REPLACE TRIGGER demo_old_new
  BEFORE UPDATE ON employees
  FOR EACH ROW
WHEN (NEW.EMPLOYEE_ID > 0)
DECLARE
    sal_diff number;
BEGIN
    sal_diff  := :NEW.SALARY  - :OLD.SALARY;
    dbms_output.put('Old salary: ' || :OLD.SALARY);
    dbms_output.put('  New salary: ' || :NEW.SALARY);
    dbms_output.put_line('  Difference ' || sal_diff);
END;
```

```
--Another example-

CREATE OR REPLACE TRIGGER EVAL_CHANGE_TRIGGER
  AFTER INSERT OR UPDATE OR DELETE
  ON EVALUATIONS
DECLARE
  log_action  EVALUATIONS_LOG.action%TYPE;
BEGIN
  IF INSERTING THEN
    log_action := 'Insert';
  ELSIF UPDATING THEN
    log_action := 'Update';
  ELSIF DELETING THEN
    log_action := 'Delete';
  ELSE
    DBMS_OUTPUT.PUT_LINE('This code is not reachable.');
  END IF;

  INSERT INTO EVALUATIONS_LOG (log_date, action)
    VALUES (SYSDATE, log_action);
END;
```

Side Note: Create Sequence. Mainly used to handle an autonumber field.

```
Syntax:
CREATE SEQUENCE sequence_name
MINVALUE value
MAXVALUE value
START WITH value
INCREMENT BY value
CACHE value;


--Example--
CREATE SEQUENCE supplier_seq
MINVALUE 1
MAXVALUE 999999
START WITH 1
INCREMENT BY 1
CACHE 20;
```

```
  ---How to use it--
  supplier_seq.NEXTVAL;
```

New and Old Example:

```
  CREATE OR REPLACE
TRIGGER NEW_EVALUATION_TRIGGER
BEFORE INSERT ON EVALUATIONS
FOR EACH ROW
BEGIN
  :NEW.evaluation_id := evaluations_sequence.NEXTVAL
END;
```

Another example from Book: Use Regular Expression.

The following example demonstrates a trigger that replaces a whitespace in a last name with a dash for hyphenated names:

```
 CREATE OR REPLACE TRIGGER TRG_REG
BEFORE INSERT ON EMP
FOR EACH ROW
WHEN (REGEXP_LIKE(NEW.name,' '))
BEGIN
  :NEW.name := REGEXP_REPLACE(:NEW.Name,' ','-',1,1);
END TRG_REG;
```

NOTE: See the format of NEW Pseudorecord inside hte WHEN clause. There is no : before it.

10.1.7 Trigger Enhancements (11g)

Resource Extracted from: http://oracle-base.com/articles/11g/trigger-enhancements-11gr1.p

10.1.7.1 Order of Firing Trigger

Oracle allows more than one trigger to be created *for the same timing point*, but it has *never guaranteed the execution order of those triggers*. The *Oracle 11g* trigger syntax now includes the *FOLLOWS* clause to guarantee execution order for triggers defined with the same timing point. The following example creates a table with two triggers for the same timing point.

```
  CREATE TABLE trigger_follows_test (
  id          NUMBER,
  description VARCHAR2(50)
);

CREATE OR REPLACE TRIGGER trigger_follows_test_trg_1
BEFORE INSERT ON trigger_follows_test
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.put_line('TRIGGER_FOLLOWS_TEST_TRG_1 - Executed');
END;
/

CREATE OR REPLACE TRIGGER trigger_follows_test_trg_2
BEFORE INSERT ON trigger_follows_test
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.put_line('TRIGGER_FOLLOWS_TEST_TRG_2 - Executed');
END;
/
```

If we insert into the test table, there is no guarantee of the execution order.

```
  SQL> SET SERVEROUTPUT ON
SQL> INSERT INTO trigger_follows_test VALUES (1, 'ONE');
TRIGGER_FOLLOWS_TEST_TRG_1 - Executed
TRIGGER_FOLLOWS_TEST_TRG_2 - Executed

1 row created.

SQL>
```

Now our objective is:
We can specify that the `TRIGGER_FOLLOWS_TEST_TRG_2` **trigger should be executed before the** `TRIGGER_FOLLOWS_TEST_TRG_1` trigger by recreating the `TRIGGER_FOLLOWS_TEST_TRG_1` trigger using the FOLLOWS clause.

```
  CREATE OR REPLACE TRIGGER trigger_follows_test_trg_1
BEFORE INSERT ON trigger_follows_test
FOR EACH ROW
FOLLOWS trigger_follows_test_trg_2
```

```
BEGIN
  DBMS_OUTPUT.put_line('TRIGGER_FOLLOWS_TEST_TRG_1 - Executed');
END;
/
```

Now the TRIGGER_FOLLOWS_TEST_TRG_1 trigger always follows the TRIGGER_FOLLOWS_TEST_TRG_2 trigger.

```
  SQL> SET SERVEROUTPUT ON
SQL> INSERT INTO trigger_follows_test VALUES (2, 'TWO');
TRIGGER_FOLLOWS_TEST_TRG_2 - Executed
TRIGGER_FOLLOWS_TEST_TRG_1 - Executed

1 row created.

SQL>
```

10.1.7.2 Compound Triggers

A compound trigger allows code for one or more timing points for a specific object to be combined into a single trigger. The individual timing points can share a single global declaration section, whose state is maintained for the lifetime of the statement. Once a statement ends, due to successful completion or an error, the trigger state is cleaned up. In previous releases this type of functionality was only possible by defining multiple triggers whose code and global variables were defined in a separate package, as shown in the Mutating Table Exceptions article, but the compound trigger allows for a much tidier solution.

The triggering actions are defined in the same way as any other DML trigger, with the addition of the COMPOUND TRIGGER clause. The main body of the trigger is made up of an optional global declaration section and one or more timing point sections, each of which may contain a local declaration section whose state is not maintained.

See the example in the web given at the beginning of the section.

10.1.7.3 Enable and Disable Triggers

It has been possible to enable and disable triggers for some time using the ALTER TRIGGER and ALTER TABLE commands.

```
ALTER TRIGGER <trigger-name> DISABLE;

ALTER TRIGGER <trigger-name> ENABLE;
```

```
ALTER TABLE <table-name> DISABLE ALL TRIGGERS;

ALTER TABLE <table-name> ENABLE ALL TRIGGERS;
```

Chapter 11

Lecture (Lec 19 & 20) for Week 12

11.1 Collections

See book Pl SQL 11g Page 212. Topic: varrays.
Topics are:

1. Defining and Using Varrays as PLSQL Program Constructs 2. Defining Varrays in Database Tables : Page 219

Chapter 12

Lecture (Lec 21 & 22) for Week 13

12.1 Performance Tuning

This is a very vast topic in itself. In this course details are out of the scope. Only basic concepts and hand-on skills will be covered. Text in many places are verbatim copy from text book by Sudarsan and Korth.

Index and Hashing: Motivation. Many queries reference only a small proportion of the records in a file. For ex- ample, a query like Find all instructors in the Physics department or Find the total number of credits earned by the student with ID 22201 references only a fraction of the student records. It is inefficient for the system to read every tuple in the instructor relation to check if the dept name value is Physics. Likewise, it is inefficient to read the entire student relation just to find the one tuple for the ID 32556,. Ideally, the system should be able to locate these records directly. To allow these forms of access, we design additional structures that we associate with files.

12.1.1 Basic Concepts

An index for a file in a database system works in much the same way as the index in this textbook. If we want to learn about a particular topic (specified by a word or a phrase) in this textbook, we can search for the topic in the index at the back of the book, find the pages where it occurs, and then read the pages to find the information for which we are looking. The words in the index are in sorted order, making it easy to find the word we want. Moreover, the index is much smaller than the book, further reducing the effort needed. Database-system indices play the same role as book indices in libraries. For example, to retrieve a student record given an ID, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the appropriate student record. Keeping a sorted list of students ID would not work well on very large databases with thousands of students, since the index would itself be very big; further, even though keeping the index sorted reduces the search time, finding a student can still be rather time-consuming. Instead, more sophisticated indexing techniques may be used. We shall discuss several of these techniques in this chapter.

There are two basic kinds of indices:

- **Ordered indices.** Based on a sorted ordering of the values.
- **Hash indices.** Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function, called a hash function.

12.1.2 Bitmap Indices

A bitmap is simply an array of bits. In its simplest form, a bitmap index on the attribute A of relation r consists of one bitmap for each value that A can take. Each bitmap has as many bits as the number of records in the relation. The ith bit of the bitmap for value v j is set to 1 if the record numbered i has the value v j for attribute A. All other bits of the bitmap are set to 0.

In our example, there is one bitmap for the value m and one for f. The ith bit of the bitmap for m is set to 1 if the gender value of the record numbered i is m. All other bits of the bitmap for m are set to 0. Similarly, the bitmap for f has the value 1 for bits corresponding to records with the value f for the gender attribute; all other bits have the value 0. Figure 12.1 shows an example of bitmap indices on a relation instructor info.

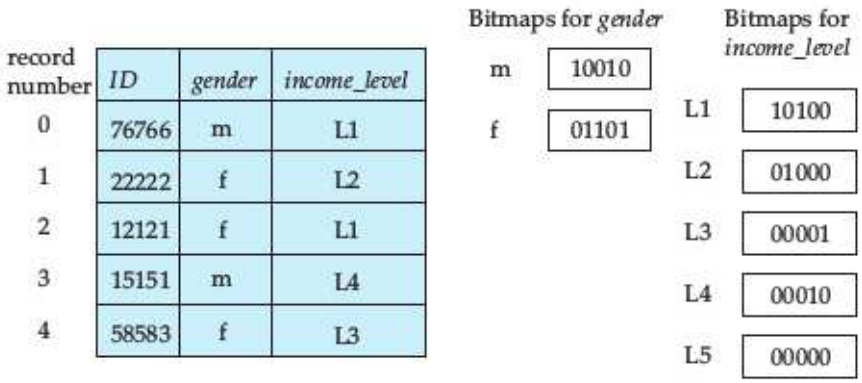

Figure 12.1: Bitmap Index:Concept

When to use it: We now consider when bitmaps are useful. The simplest way of retrieving all records with value m (or value f) would be to simply read all records of the relation and select those records with value m (or f, respectively). The bitmap index doesnt really help to speed up such a selection. While it would allow us to read only those records for a specific gender, it is likely that every disk block for the file would have to be read anyway.

In fact, bitmap indices are useful for selections mainly when there are selec- tions on multiple keys. Suppose we create a bitmap index on attribute income level, which we described earlier, in addition to the bitmap index on gender.

Consider now a query that selects women with income in the range $10,000 to 19,999$. This query can be expressed as :

```
 select *
from r
where gender = f and income level = L2;
```

To evaluate this selection, we fetch the bitmaps for gender value f and the bitmap for income level value L2, and perform an intersection (logical-and) of the two bitmaps. In other words, we compute a new bitmap where bit i has value 1 if the ith bit of the two bitmaps are both 1, and has a value 0 otherwise. In the example in Figure 11.35, the intersection of the bitmap for gender = f (01101) and the bitmap for income level = L2 (01000) gives the bitmap 01000.

Another important use of bitmaps is to count the number of tuples satisfying a given selection. Such queries are important for data analysis. For instance, if we wish to find out how many women have an income level L2, we compute the intersection of the two bitmaps and then count the number of bits that are 1 in the intersection bitmap. We can thus get the desired result from the bitmap index, without even accessing the relation.

Bitmap Index in Data Warehouses: Bitmap indexes are widely used in data warehousing environments. The environments typically have large amounts of data and ad hoc queries, but a low level of concurrent DML transactions. For such applications, bitmap indexing provides:

- Reduced response time for large classes of ad hoc queries.
- Reduced storage requirements compared to other indexing techniques.

12.1.3 ROWID in Oracle

For details see the web:
https://docs.oracle.com/cd/E11882_01/server.112/e40540/logical.htm#CNCPT89139

Oracle Database uses a rowid to uniquely identify a row. Internally, the rowid is a structure that holds information that the database needs to access a row. A rowid is not physically stored in the database, but is inferred from the file and block on which the data is stored.

An *extended rowid* includes a data object number. This rowid type uses a base 64 encoding of the physical address for each row. The encoding characters are `A-Z, a-z, 0-9, +, and /`.

```
Example 12-1 ROWID Pseudocolumn1

SQL> SELECT ROWID FROM employees WHERE employee_id = 100;

ROWID
------------------
AAAPecAAFAAAABSAAA
```

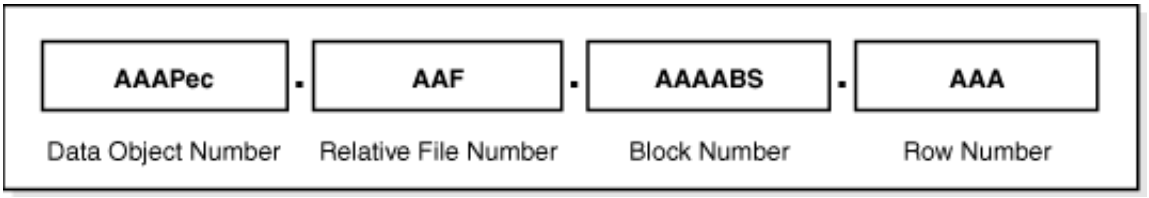

Figure 12.2: ROWID Format

ROWID Format:

An extended rowid is displayed in a four-piece format, OOOOOOFFFBBBBBBRRR, with the format divided into the following components:

- **OOOOOO**
  The data object number identifies the segment (data object AAAPec in Figure 12.2). A data object number is assigned to every database segment. Schema objects in the same segment, such as a table cluster, have the same data object number.
- **FFF**
  The tablespace-relative data file number identifies the data file that contains the row (file AAF in Figure 12.2).
- **BBBBBB**
  The data block number identifies the block that contains the row (block AAAABS in Figure 12.2). Block numbers are relative to their data file, not their tablespace. Thus, two rows with identical block numbers could reside in different data files of the same tablespace.
- **RRR**
  The row number identifies the row in the block (row AAA in Example 12-1).The row number identifies the row in the block (row AAA in Figure 12.2).

12.1.4 Creating an Index

You create an index via the create index command. When you designate a primary key or a unique column during table creation or maintenance, Oracle will automatically create a unique index to support that constraint.

12.1.4.1 Guidelines for Application-Specific Indexes

You can create indexes on columns to speed up queries. Indexes provide faster access to data for operations that return a small portion of a table's rows.

In general, create an index on a column in any of the following situations:

- The column is queried frequently.

- A referential constraint exists on the column.
- A UNIQUE key constraint exists on the column.

You can create an index on any column; however, if the column is not used in any of these situations, creating an index on the column does not increase performance and the index takes up resources unnecessarily.

SQL CREATE INDEX Syntax:
Creates an index on a table. Duplicate values are allowed:

```
CREATE INDEX index_name
ON table_name (column_name)
```

SQL CREATE UNIQUE INDEX Syntax:
Creates an index on a table. Duplicate values are NOT allowed:

```
CREATE UNIQUE INDEX index_name
ON table_name (column_name)
```

Examples:

```
 CREATE INDEX PIndex
ON Persons (LastName)

or

CREATE INDEX PIndex
ON Persons (LastName, FirstName)

bitmap:

CREATE BITMAP INDEX emp_bitmap_idx
ON EMP(Gender);
```

AN On-hand practice:

```
 SET AUTOTRACE ON;

 SELECT FIRST_NAME
 FROM EMPLOYEES
 WHERE DEPARTMENT_ID=50;

 CREATE INDEX IND_EMP
 ON EMPLOYEES (DEPARTMENT_ID);
```

```
SELECT FIRST_NAME
FROM EMPLOYEES
WHERE DEPARTMENT_ID=50;
```

Build Function Based Index: If the where clause uses any function then basic index will not be used, so no performance is gained.

```
DROP INDEX first_name_idx;
CREATE INDEX first_name_idx ON user_data (UPPER(first_name));
```

12.1.4.2 Index-Organized Table

An index-organized table keeps its data sorted according to the primary key column values for the table. An index-organized table stores its data as if the entire table was stored in an index. Indexes serve two main purposes:

- **To enforce uniqueness**. When a PRIMARY KEY or UNIQUE constraint is created, Oracle creates an index to enforce the uniqueness of the indexed columns.
- **To improve performance**. When a query can use an index, the performance of the query may dramatically improve.

Why Not Normal Index? An index-organized table allows you to store its entire data in an index. A normal index only stores the indexed columns; an index-organized table stores all its columns in the index.

To create an index-organized table, use the organization index clause of the create table

```
create table TROUBLE_IOT (
City VARCHAR2(13),
SampleDate DATE,
Noon NUMBER(4,1),
Midnight NUMBER(4,1),
Precipitation NUMBER,
constraint TROUBLE_IOT_PK PRIMARY KEY (City, SampleDate))
organization index;
```

To create TROUBLE as an index-organized table, *you must create a PRIMARY KEY constraint* on it.

In general, an index-organized table is most effective when the primary key constitutes a large part of the tables columns.

12.1.5 Clusters & Sequence

Clusters. Clustering is a method of storing tables that are intimately related and often joined together into the same area on disk.

For example, instead of the BOOKSHELF table being in one section of the disk and the BOOKSHELF_AUTHOR table being somewhere else, their rows could be interleaved together in a single area, called a cluster. The *cluster key* is the column or columns by which the tables are usually joined in a query (for example, Title for the BOOKSHELF and BOOKSHELF_AUTHOR tables). To cluster tables, you must own the tables you are going to cluster together.

Step by step Example:

1. Create the cluster first.

```
create cluster BOOKandAUTHOR (Col1 VARCHAR2(100));
Cluster created.
```

   This creates a cluster (a space is set aside, as it would be for a table) with nothing in it. The use of Col1 for the cluster key is irrelevant; youll never use it again. *However, its definition should match the primary key of the table to be added.*

2. Next, tables are created to be included in this cluster:

```
  create table BOOKSHELF
(Title VARCHAR2(100) primary key,
Publisher VARCHAR2(20),
CategoryName VARCHAR2(20),
Rating VARCHAR2(2),
constraint CATFK foreign key (CategoryName)
references CATEGORY(CategoryName)
)
cluster BOOKandAUTHOR (Title);
```

3. Prior to inserting rows into BOOKSHELF, you must create a cluster index:

```
    create index BOOKandAUTHORndx
on cluster BOOKandAUTHOR;
```

4. Now a second table is added to the cluster:

```
  create table BOOKSHELF_AUTHOR
(Title VARCHAR2(100),
AuthorName VARCHAR2(50),
constraint TitleFK Foreign key (Title) references BOOKSHELF(Title),
constraint AuthorNameFK Foreign key (AuthorName)
references AUTHOR(AuthorName)
)
cluster BOOKandAUTHOR (Title);
```

When these two tables are clustered, each unique title is actually stored only once, in the cluster key. To each title are attached the columns from both of these tables.

The data from both of these tables is actually stored in a single location, almost as if the cluster were a big table containing data drawn from both of the tables that make it up.

Sequence: Already shown.

```
create sequence CustomerID increment by 1 start with 1000;
```

Then CustemerID.NextVal is the calling form.

12.1.6 Partitioning

As the number of rows in your tables grows, the management and performance impacts will increase. Backups will take longer, recoveries will take longer, and queries that span an entire table will take longer. You can mitigate the administrative and performance issues for large tables by separating the rows of a single table into multiple parts. Dividing a tables data in this manner is called partitioning the table; the table that is partitioned is called a partitioned table, and the parts are called partitions.

Partitioning is useful for very large tables. By splitting a large tables rows across multiple smaller partitions, you accomplish several important goals:

- The performance of queries against the tables may improve because Oracle may have to search only one partition (one part of the table) instead of the entire table to resolve a query.
- The table may be easier to manage. Because the partitioned tables data is stored in multiple parts, it may be easier to load and delete data in the partitions than in the large table.
- Backup and recovery operations may perform better. Because the partitions are smaller than the partitioned table, you may have more options for backing up and recovering the partitions than you would have for a single large table.

Do practice the examples of *Range Partition* and *Hash Partition*.

Chapter 13

Lecture (Lec 21 & 22) for Week 13

13.1 SQL Loader

It is used to upload data from a flat file to Oracle tables.

Steps Involved:

1. **Input File:** This is the input text file that contains the data that needs to be loaded into an oracle table. Each and every records needs to be in a separate line, and the column values should be delimited by some common delimiter character. For some of the examples mentioned below, well use the following *employee.txt* file to upload the data to the employee table.

```
$ gedit employee.txt

100,Thomas,Sales,5000
200,Jason,Technology,5500
300,Mayla,Technology,7000
400,Nisha,Marketing,9500
500,Randy,Technology,6000
501,Ritu,Accounting,5400
```

2. **Control File:** This contains the instructions to the sqlldr utility. This tells sqlldr the location of the input file, the format of the input file, and other optional meta data information required by the sqlldr to upload the data into oracle tables.

```
$ gedit example1.ctl

load data
 infile '/home/yourname/employee.txt'
[APPEND] into table employee
 fields terminated by ","
 ( id, name, dept, salary )
```

The above control file indicates the followings:

- infile Indicates the location of the input data file
- into table Indicates the table name where this data should be inserted (APPEND will add next time)
- fields terminated by Indicates the delimiter that is used in the input file to separate the fields (common delimiter is WHITESPACE (space))
- (id, name, dept, salary) Lists the name of the column names in the table into which the data should be uploaded

3. **Invoke SQL Loader:** To invoke SQL loader you need commands like:

```
sqlldr usename/password@database control=/home/myname/example1.ctl
readsize=1000000 log=/home/myname/log.dat;
```

4. **Examine What Happened:** If everything is valid then the data will be loaded successfully. You can see the log about it. It should look like:

```
$ cat sqlldr-add-new.log

Control File:   /home/yourame/example1.ctl
Data File:      /home/yourname/employee.txt

Table EMPLOYEE:
  5 Rows successfully loaded.
  0 Rows not loaded due to data errors.
  0 Rows not loaded because all WHEN clauses were failed.
  0 Rows not loaded because all fields were null.

Elapsed time was:     00:00:00.04
CPU time was:         00:00:00.00
```

Chapter 14

Lab Exercises

14.1 Lab for W1 & W2

Done.

14.2 Lab for W3

1. Create one tablespace *myspace* and one user. And assign the tablespace to the new user. Create a number of objects (4 tables: T1, T2, T3 and T4) and assign one table's data (T4) to a different tablespace *myspace2*.

2. Remember the following definitions:
   A subquery in the WHERE clause of a SELECT statement is also called *a nested subquery*.

   A subquery in the FROM clause of a SELECT statement is also called *an inline view*.

   Now create a number of related tables at your own choice and show the functionality of *a nested subquery* and *an inline view*.

3. Show with suitable example the difference between Left and Right Outer joins. You must create tables with relationship among them and insert relevant records to demonstrate it. Also show how natural join works here.

14.3 Lab for W4 FOR GROUP A

Consider the following scenario:

Bangladesh Government wants to create its national database for the various purposes. The most essential information Government is interested to maintain are (but not limited to) such as:
Citizen Name, Date of Birth, Permanent Address, Present Address, Occupation, Physical Height, Photograph, Scanned copy of Signature, Monthly Salary, Blood Group.

Address should be clustered in a way that anytime for a given area all information are available very precisely. Occupation should also be organized containing 2 layers, for instance: Layer 1 Teaching and Layer 2: School Teaching/College Teaching.

Parts of Task A:

1. First design an ER-D for the above scenario and then write DDL statements to create them. Your DDL should maintain the space requirements of each tables considering the above scenario.

2. Insert some data in all relevant tables created.

3. Answer the followings:

   - Find all people name, DOB who are in Teaching profession. Then narrow down to only University Teaching.
   - Find out top 5 professions according to salary.
   - Generate a report for people who are are born either in Dhaka or Khulna with the following information: Name, DOB, Monthly Salary (if available)

Task B:

1. Show with appropriate example the use of the following built-in functions:

CONCANT, INITCAP, INSTR, LOWER, UPPER, LENGTH, L/R PAD, L/R TRIM, SUBSTR, COUNT

2. Present an example to explain the working principal of REGEXP_SUBSTR.

14.4 Lab for W4 FOR GROUP B

Consider the following scenario:

Suppose you are given the task to automation of a large bank. The bank has more than 1 hundred branch located at different parts of the country. Customers primarily come to any branch of the bank and can open an account. The account could be either be *Saving* or *Current*. Only difference between them is that of calculation of interests after a certain time frame. The number of customer in that bank is at least several hundred thousand. Bank wants to keep information of its Customers at least the followings:

Citizen Name, Date of Birth, Permanent Address, Present Address, Occupation, Physical Height, Photograph, Scanned copy of Signature, Monthly Salary, Blood Group.

Address should be clustered in a way that anytime for a given area all information are available very precisely. Occupation should also be organized containing 2 layers, for instance: Layer 1 Teaching and Layer 2: School Teaching/College Teaching.

When a customer *deposits* or *withdraws* money a *transaction* is made in the database containing its ID, date and time, amount and type of operation. It is also important that it is tracked for each transaction who is the operator. (So you need to organize employees information of the Bank).

Parts of Task A:

- Design the ER-D and then write DDL statements to create them. Your DDL should maintain the space requirements of each tables considering the above scenario.
- Insert some data in all relevant tables created.
- Answer the followings:

    1. Find the total number of customers at each branch where each branch has at least 100 customers, show them in ascending order.
    2. Find all customer name, DOB who are in Teaching profession. Then narrow down to only University Teaching.
    3. Find out the customers name, his home district and DOB who deposited a total of at least 20000 BDT upto now.

Task B:

1. Show with appropriate example the use of the following built-in functions:

CONCANT, INITCAP, INSTR, LOWER, UPPER, LENGTH, L/R PAD, L/R TRIM, SUBSTR, COUNT

2. Present an example to explain the working principal of DECODE and REGEXP_SUBSTR.

14.5 Lab for W4 FOR GROUP C

Consider the following scenario:

Suppose you are given the task for automation of the result processing system of a very large university where number of department is more than 50, and in each department there are at least 3 programs and number of students (in total) is at least 30000 for one year.

Each year students are *admitted* for a *specific program.* Each program has a fixed number of semesters. In each semester a student is *registered* against a number of courses. Normally one course is *conducted* by one teacher. Grading policy is similar to that of IUT.

A course has its code, title, credit as attribute.

Beside academic activities, university wants to keep students basic information such as :

Student Name, Date of Birth, Permanent Address, Present Address, Physical Height, Photograph, Scanned copy of Signature, Department, Program, Blood Group.

Address should be clustered in a way that anytime for a given area all information are available very precisely.

University also keeps information about extra-curricular activities of each student as well as anti-disciplinary activities.

Parts of Task A:

- Design the ER-D and then write DDL statements to create them. Your DDL should maintain the space requirements of each tables considering the above scenario.
- Insert some data in all relevant tables created.
- Answer the followings:

    1. Find the total number of students for each department and each program of that department.
    2. Find out list of courses (code and title) a student (his name, DOB) has taken for a specific semester. Also it gives their corresponding letter grades for each course. If any letter grade is not found then show 'NOT FOUND'.
    3. For a given program and semester find out total number of grade distribution (i.e. total number got A+ for instance).
    4. Find out top 3 students according to extra-curricular activities. Also list worst 3 students according to anti-disciplinary activities.

Task B:

1. Show with appropriate example the use of the following built-in functions:

CONCANT, INITCAP, INSTR, LOWER, UPPER, LENGTH, L/R PAD, L/R TRIM, SUBSTR, COUNT

2. Present an example to explain the working principal of DECODE and REGEXP_SUBSTR.

14.6 Lab for W5 FOR GROUP C

Refer to the previous assignment for week 4. Complete the pending parts.

Additional Module:
Add the following module in your design:

- The university is interested to keep records of its ex-students information (i.e. Alumni). Expand your design so that the university can get some statistical idea about job status of its former students both home and abroad.

Additional Queries:

1. Recall your previous task: *For a given program and semester find out total number of grade distribution (i.e. total number got A+ for instance).* Now you need to write query to show the grade distribution according to department and program. Also it should show total number of different grades without considering department and program (i.e. total A+/A/A-/B holder of this university).

14.7 Lab for W6 FOR GROUP A

Demonstration of UI of their projects and identification of subprograms.

14.8 Lab for W6 FOR GROUP B

Demonstration of UI of their projects and identification of subprograms.

14.9 Lab for W6 FOR GROUP C

Demonstration of UI of their projects and identification of subprograms.

14.10 Lab for W7 FOR GROUP A

Consider the following entities:

1. `Citizen < ID,Name,DOB >`

2. `Salary < CID, Dt, Amount >`

3. `AirportsLog < Dt, Tp (type) , PName (name of desti) >`

Now write a function (or more) to satisfy the following requirements:
Input: Citizen ID.
Output: Status
Status Domain:`<CIP, VIP, ORDINARY>`
Algorithm for Status evaluation:
It is CIP if: 1) His average salary is above 100000 for the last 5 years. and 2)He has made a total of 10 departures in that period.

It is VIP if: 1) His average salary is between 50000 and 100000 for the last 5 years. and 2)He has made a total of 5 departures in that period.

Otherwise set his status ORDINARY.
NOTE: Make the solution as modular as possible.

14.11 Lab for W7 FOR GROUP B

Consider the following entities:

```
1. Acc_type < ID,Name,IRate,GraceP >
 GraceP: 1= each 3 months, 2= each 6 months

2. Customers < ID, Name, DOB, Address >

3. Accounts < AccNo, CID, Acc_type (FK), OpenDt, Current_Balance,LastDtIgiven >

4. Transactions < Dt, AccNo (FK), Tp (type: deposit or withdraw),Amount >
```

Task A: Now write a function (or more) to satisfy the following requirements:
Input: Customer ID.
Output: Status
Status Domain:<CIP, VIP, ORDINARY>
Algorithm for Status evaluation:

If his current balance is more than 100000 then his status is VIP.
If his current balance is between 40000 and 100000 then his status is IMPORTANT.
Otherwise ORDINARY.

Task B: Write a subprogram to update a customer's current based on:
If today is between 1 and 3 months comparing with LastDtIgiven then calculate his interest based on the corresponding table (i.e. Acc_type).

If today is between 3 and 6 months comparing with LastDtIgiven then calculate his interest based on the corresponding table (i.e. Acc_type)

14.12 Lab for W7 FOR GROUP C

Consider the following entities:

```
1. Acc_type < ID,Name,IRate,GraceP >
 GraceP: 1= each 3 months, 2= each 6 months

2. Customers < ID, Name, DOB, Address >

3. Defaulters <CID (FK), Dt (of entry),Tp (Major or minor)>

3. Accounts < AccNo, CID, Acc_type (FK), OpenDt, Current_Balance,LastDtIgiven >

4. Transactions < Dt, AccNo (FK), Tp (type: deposit or withdraw),Amount >
```

Task A: Now write a function (or more) to satisfy the following requirements:
Input: Customer ID.
Output: Status
Status Domain:<CIP, VIP, ORDINARY>
Algorithm for Status evaluation:

1. **VIP:** If his current balance is more than 200000 and he he has no allegation at all (major or minor) then his status is VIP.
2. **IMPORTANT:** If his current balance is between 40000 and 200000 and his has no major allegation (i.e. Defaulters Table) then his status is IMPORTANT.
3. **ORDINARY:** If his current balance is between 10000 and 40000 and his has only one major allegation (i.e. Defaulters Table) then his status is ORDINARY.
4. **CRIMINAL:** If (his current balance is irrelevant) his has 3 or more major allegations and 5 or more minor allegations (i.e. Defaulters Table) then his status is CRIMINAL.

Task B: Write a subprogram to update a customer's current balance based on:
If today is between 1 and 3 months comparing with LastDtIgiven then calculate his interest based on the corresponding table (i.e. Acc_type).

If today is between 3 and 6 months comparing with LastDtIgiven then calculate his interest based on the corresponding table (i.e. Acc_type)

14.13 Lab for W8 FOR GROUP A

(After Mid-Semester Exam.)
Overall review of the previous lessons. Give specific problem involving cursors from the mid-semester exam. Show them how to create a one-to-many UI (i.e. Master-Detail in Oracle technology).

14.14 Lab for W8 FOR GROUP B

(After Mid-Semester Exam.)
(Identical to A)

Overall review of the previous lessons. Give specific problem involving cursors from the mid-semester exam. Show them how to create a one-to-many UI (i.e. Master-Detail in Oracle technology).

14.15 Lab for W8 FOR GROUP C

(After Mid-Semester Exam.)
(Identical to A)

Overall review of the previous lessons. Give specific problem involving cursors from the mid-semester exam. Show them how to create a one-to-many UI (i.e. Master-Detail in Oracle technology).

14.16 Lab for W9 FOR GROUP A

Consider the following scenario:
A bank issues loan to its customers. There are three loan schemes: i) S-A ii)S-B and iii)S-C. They are defined as follows:

Scheme	No. of Installment	Service Charge for remaining loan	Eligibility
S-A	30	5%	Total Transaction in the last 12 months$\geq$ 2000000
S-B	20	10%	Total Transaction in the last 12 months$\geq$ 1000000
S-C	15	15%	Total Transaction in the last 12 months$\geq$ 500000

Your tasks are as follows:

1. Design the table definitions and issue the required DDLs. Additional assumptions are welcome in design phase.
2. Write a function to assign a customer to a specific category of loans as mentioned.
3. Once a customer is assigned to a specific loan scheme, write a procedure to schedule his loan. Assume each loan must be paid after 3 months interval.

14.17 Lab for W10 FOR GROUP A

Consider the main problem for your group as mentioned in 14.3. In this lab you are asked to generate the ID of each citizen. ID generation Algorithm is given below:

ID Formate: *DVYYYYDDMM.XXXXXXX*
Algorithm: DV is the division code of the citizen was born. YYYYDDMM is the numeric representation of his Date of Birth. And XXXXXXX is the serial no (in increasing order).

Your task is to write a function to generate the ID as described. It takes two input parameters: DOB and DV code and returns the ID.

14.18 Lab for W10 FOR GROUP B

Consider the main problem for your group as mentioned in 14.4. In this lab you are asked to generate the Account No in a specified format given below:

ACC No Format: *TTBBBYYYYMMDD.XXXXXX*
where T=account type
B=Branch Code
X=Serial No
YMD=Year Month Day of DOB of the citizen.

Your tasks are:

- Create DDLs as needed to fulfill the requirements.
- Write a function that takes 3 IN parameters such as Branch Code,Acc Type, Customer ID to fetch its DOB. And it returns a number as the new Account No.

14.19 Lab for W10 FOR GROUP C

Consider the main problem for your group as mentioned in 14.5. In this lab you are asked to generate the Student ID in a specified format given below:

Student ID format: *YYDPXX*
where, Y=Year of enrollment.
D=Dept Code
P=Program Code
X=Serial No.

Your tasks are:

- Create DDLs as needed to fulfill the requirements.
- Write a function that takes 3 IN parameters such as Date of enrollment,Department Code and Program Code. And it returns a number as the new Student ID.

14.20 Lab for W11 FOR GROUP A

Refer to problem in 14.3 for Country Database Management. Now we concentrate on Government Employees. Salary of an employee is defined as :

Total Monthly Salary=Basic + Housing. 40% + Transport. 10%

There are 3 different scales defined by Government:

S1, S2 and S3 each has only one value: Basic amount.
Also add designation to emp.
Tasks A: Design a monthly salary transaction (format is : *TPYYYYDDMMHH-MISS*). Note partial salary (all basic, housing and transport) must be calculated if one joins in the middle of the month.
Part B:

Now you need to design the Tax Module in connection with bank account. Tax offices maintain the following rules for tax collections:

1. If Total Salary Paid (TSP) at the end of a year is $\leq$100000 then no tax is applied. (It should consult your salary monthly transaction for it.)
2. If TSP is between 100000 and 40000 then 5% tax is applied.
3. If TSP is between 400000 and 100000 then 10% tax is applied.
4. Else 20% tax is applied.

14.21 Lab for W11 FOR GROUP B

Refer to problem in 14.4. Here you need to work on 3 types of accounts:

1. Single Account: One person one account.
2. Joint Account: More than one persons having one account.
3. Business Account: (opt) Account will be opened for a specific organization but will be operated by one or more persons of that organization.

Part A:

Your tasks are as follows:

- You need to design ER for the daily transactions with the following two restrictions:
    1. Max amount per transaction $\leq$100000.
    2. Max amount per month against an account number $\leq$500000.
- The format of Daily Transaction is *TPYYYYDDMMHHMISS*, date implies date of opening account. TP is the type of account.

Part B:

Now you need to design the Tax Module in connection with bank account. Tax offices maintain the following rules for tax collections:

1. If Total Balance (TB) at the end of a year is $\leq$100000 then no tax is applied. (It should consult your daily bank transaction considering both credit and debit.)
2. If TB is between 100000 and 40000 then 5% tax is applied.
3. If TB is between 400000 and 100000 then 10% tax is applied.
4. Else 20% tax is applied.

14.22 Lab for W11 FOR GROUP C

Refer to problem in 14.5. Account section will be added here for the salary management of its employees. University maintains 3 different scales as follows:

S1, S2 and S3

1. S1: In S1 total salary is calculated as:
   Total Monthly Salary=Basic + Housing. 10% of Basic + Transport. 15% of Basic

2. S2: In S2 total salary is calculated as:
   Total Monthly Salary=Basic + Housing. 20% of Basic + Transport. 10% of Basic

3. S3: In S3 total salary is calculated as:
   Total Monthly Salary=Basic + Housing. 30% of Basic + Transport. 5% of Basic

Task A:

- Create a function to calculate the salary of any employee. *Note* partial salary (all basic, housing and transport) must be calculated if one joins in the middle of the month.
- Design a monthly salary transaction (format is : *YYYYDDMMHHMISS.NNN*) withing the row-level trigger.

Part B:

Now you need to design the Tax Module in connection with bank account. Tax offices maintain the following rules for tax collections:

1. If Total Salary Paid (TSP) at the end of a year is $\leq$100000 then no tax is applied.
2. If TSP is between 100000 and 400000 then 5% tax is applied.
3. If TSP is between 400000 and 100000 then 10% tax is applied.
4. Else 20% tax is applied.

You are required to create necessary functions to calculate the total tax payable according to the above rules.

14.23 Lab for W13 FOR GROUP A

Consider 2 flat data files: i) inputdata1.txt and ii)inputdata2.txt

The contents of i) inputdata1.txt is like this:

```
101#a#JAN-12-1987
201#b#JAN-22-1985
301#c#JAN-20-1988
```

Where the first one is the ID, 2nd one is the Name and last one is the Join Date.

The contents of i) inputdata2.txt is like this:

```
401#g#JAN-12-1987
201#b#JAN-22-1990
701#h#JAN-20-1988
```

Where the first one is the ID, 2nd one is the Name and last one is the Join Date.

Tasks:

1. Load these data into oracle table using SQL Loader
2. Note that ID 201 is common in both files. The common is determined if the ID and name is identical. In this case it should take only the entry with latest date of joining (here `201 b JAN-22-1990` is the valid entry). You need to accomplish this check for each data being finally loaded. (Nothing with SQL loader, write one procedure to do it).

14.24 Lab for W13 FOR GROUP B

Consider 2 flat data files: i) inputdata1.txt and ii)inputdata2.txt

The contents of i) inputdata1.txt is like this:

```
101  a  JAN-12-1987
201  b  JAN-22-1985
301  c  JAN-20-1988
```

Where the first one is the ID, 2nd one is the Name and last one is the Join Date.

The contents of i) inputdata2.txt is like this:

```
401  g  JAN-12-1987
201  b  JAN-22-1990
701  h  JAN-20-1988
```

Where the first one is the ID, 2nd one is the Name and last one is the Join Date.

Tasks:

1. Load these data into oracle table using SQL Loader
2. Note that ID 201 is common in both files. The common is determined if the ID and name is identical. In this case it should take only the entry with latest date of joining (here `201 b JAN-22-1990` is the valid entry). You need to accomplish this check for each data being finally loaded. (Nothing with SQL loader, write one procedure to do it).

   Additionally, you must store the deleted records with time of deletion (use row-level trigger)

14.25 Lab for W13 FOR GROUP C

Consider 2 flat data files: i) inputdata1.txt and ii)inputdata2.txt

The contents of i) inputdata1.txt is like this:

```
101  a  JAN-12-1987
201  b  JAN-22-1985
301  c  JAN-20-1988
```

Where the first one is the ID, 2nd one is the Name and last one is the Join Date.

The contents of i) inputdata2.txt is like this:

```
401  g  JAN-12-1987
201  b  JAN-22-1990
701  h  JAN-20-1988
```

Where the first one is the ID, 2nd one is the Name and last one is the Join Date.

Tasks:

1. Load these data into oracle table using SQL Loader
2. Note that ID 201 is common in both files. The common is determined if the ID and name is identical. In this case it should take only the entry with latest date of joining (here `201 b JAN-22-1990` is the valid entry). You need to accomplish this check for each data being finally loaded. (Nothing with SQL loader, write one procedure or SQL statement to do it).

   Additionally, you must store the deleted records with time of deletion (use row-level trigger)
3. Write SQL statement to select all data from a table and send them to a specific file (use spool). *Your are advised to search on net to get the solution since it is not included in Lecture Note.*