Contents

Recap of the previous Lecture (if needed)

Chapter 1

RDBMS Programming Part1: Basic

1.1 Course Layout

Date: December 7, 2015

- **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.

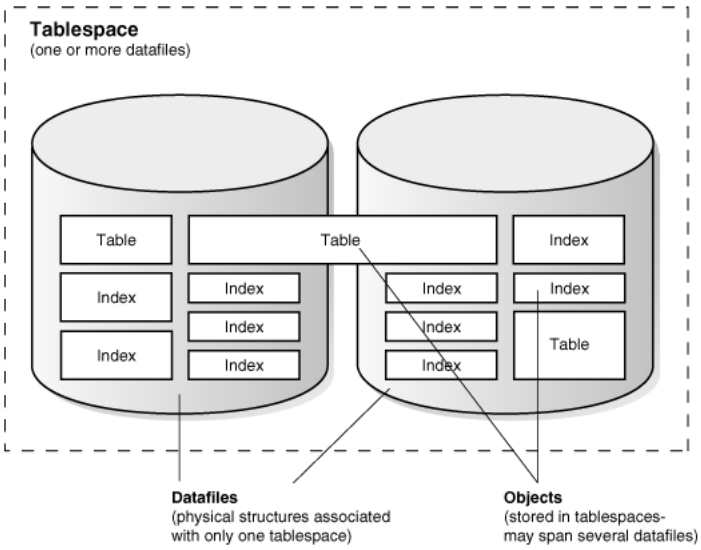

Figure 2.1: Tablespace, Datafiles and Objects

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.

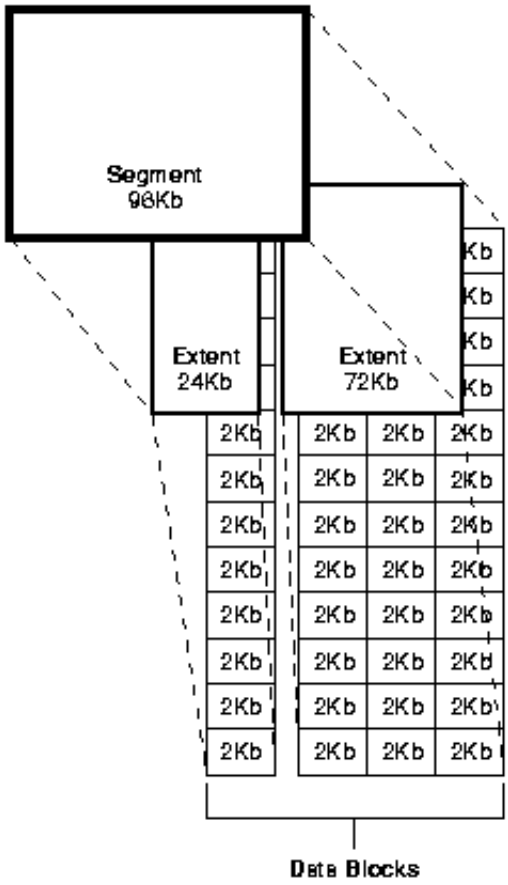

Figure 2.2: Oracle Storage Management

- **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:

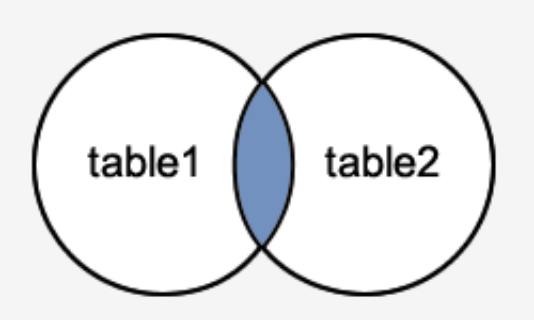

Figure 3.1: Inner Join

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    ...
```

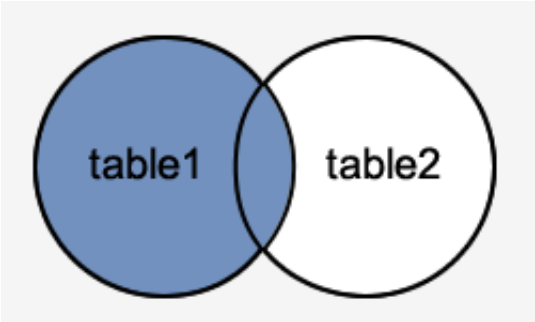

Figure 3.2: Left Outer Join

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.
```

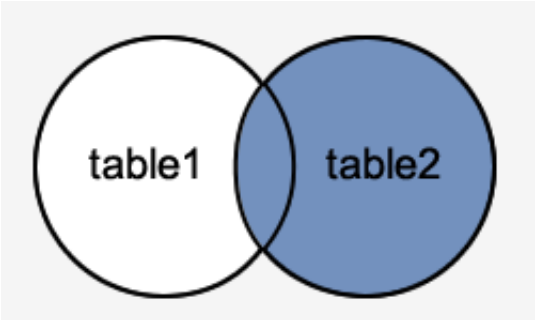

Figure 3.3: Right Outer Join

(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;
```

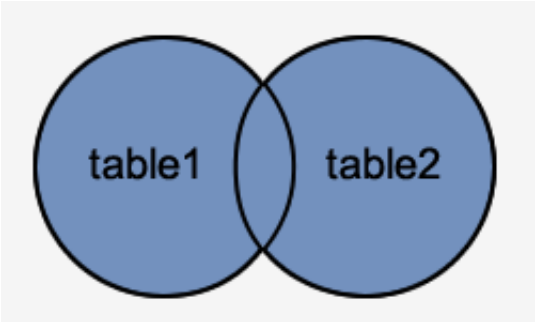

Figure 3.4: Full Outer Join

(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.

```
Employee
EmpID  EmpName
13  Jason
8  Alex
3  Ram
17  Babu
25  Johnson

Location
EmpID  EmpLoc
13  San Jose
8  Los Angeles
3  Pune, India
17  Chennai, India
39  Bangalore, India
```

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 following program illustrates the memory allocation differences between the CHAR and VARCHAR2 datatypes:

   ```
   DECLARE
   c CHAR(32767) := ' ';
   v VARCHAR2(32767) := ' ';
   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: You can size a VARCHAR2 datatype *up to 32,767 bytes* in length.You can store character strings larger than 4,000 bytes in CLOB or LONG columns.

Lets look at the example below:

```
 var1 VARCHAR2(100); -- Explicitly 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

(Source http://www.tutorialspoint.com/plsql/plsql_cursors.htm)

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;
```

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