Unit I: Introduction to RDBMS

➢ 1.1 Introduction to popular rdbms product and their features

A relational DBMS (RDBMS) stores data in a tabular form where a column represents a property and each row in a table represents a record.

RDBMSs allow Create, Read, Update, and Delete (CRUD) operations. Structured Query Language (SQL) is the language that is used to query, update, and delete data in relational database management systems (RDBMS). SQL is a standard query language. SQL language's queries are also known as SQL commands or SQL statements.

Here is the list of most popular RDBMSs are:

Oracle
MySQL
SQL Server
PostgreSQL
IBM DB2
Microsoft Access

1. Oracle

Oracle developed by Oracle Corporation is the most popular relational database system (RDBMS). Not only Oracle is a RDBMS, but also provides functionality for Cloud, Document Store, Graph DBMS, Key-value storage, BLOG, and PDF Storages. Recently. Oracle just announced autonomous feature that allows database to be intelligent and self-managed.

The current version of Oracle Database is 18c.

Oracle database is a relational database (RDBMS). Relational databases store data in a tabular form of rows and columns. Column of a database table represents the attributes of an entity and rows of a table stores records. An RDBMS that implements object-oriented features such as user-defined types, inheritance, and polymorphism is called an object-relational database management system (ORDBMS). Oracle Database has extended the relational model to an object-relational model, making it possible to store complex business models in a relational database.

One characteristic of an RDBMS is the independence of physical data storage from logical data structures.

In Oracle Database, a database schema is a collection of logical data structures, or schema objects. A database user owns a database schema, which has the same name as the user name.

Schema objects are user-created structures that directly refer to the data in the database. The database supports many types of schema objects, the most important of which are tables and indexes.

A schema object is one type of database object. Some database objects, such as profiles and roles, do not reside in schemas.

2. MySQL

MySQL is world's most popular database that is open source and free. MySQL was acquired by Oracle as a part of Sun Microsystems acquisition in 2009.
In MySQL, the SQL part of "MySQL" stands for "Structured Query Language". SQL is the most common standardized language used to access databases. Depending on your programming environment, you might enter SQL directly (for example, to generate reports), embed SQL statements into code written in another language, or use a language-specific API that hides the SQL syntax.

Key properties of MySQL:

MySQL is a database management system.

MySQL databases are relational.

MySQL software is Open Source.

The MySQL Database Server is very fast, reliable, scalable, and easy to use.

MySQL Server works in client/server or embedded systems.

3. SQL Server

SQL Server database developed by Microsoft is one of the most popular databases in the world. Initially launched in 1989 and written in C, C++, SQL

Server is now widely used among major companies. SQL Server is also a part of Microsoft's Azure cloud as Azure SQL Server. The current version of SQL Server is SQL Server 2019.

Similar to Oracle and MySQL, SQL Server is also a relational database management system (RDBMS).

Some popular SQL Server editions are:

Azure SQL Database is the cloud-based version of Microsoft SQL Server, presented as a platform as a service offering on Microsoft Azure.

Compact (SQL CE), the compact edition is an embedded database engine. Unlike the other editions of SQL Server, the SQL CE engine is based on SQL Mobile (initially designed for use with hand-held devices) and does not share the same binaries. Due to its small size (1 MB DLL footprint), it has a markedly reduced feature set compared to the other editions. For example, it supports a subset of the standard data types, does not support stored procedures or Views or multiple-statement batches (among other limitations). It is limited to 4 GB maximum database size and cannot be run as a Windows service, Compact Edition must be hosted by the application using it. The 3.5 version includes support for ADO.NET Synchronization Services. SQL CE does not support ODBC connectivity, unlike SQL Server proper.

SQL Server Enterprise Edition is the main database most companies buy that comes with every product feature.

SQL Server Developer Edition includes the same features as SQL Server Enterprise Edition, but is limited to one license only that is used for development purposes by software developers.

4. PostgreSQL

PostgreSQL is a powerful, open source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads. The origins of PostgreSQL date back to 1986 as part of the POSTGRES project at the University of California at Berkeley and has more than 30 years of active development on the core platform. The current version of PostgreSQL is 11.4 that was released on 20 June 2019. PostgreSQL is written in C language and managed by the PostgreSQL Global Development Group.

PostgreSQL is known for its architecture, reliability, data integrity, robust feature set, extensibility, and the dedication of the open source community behind the software to consistently deliver performant and innovative solutions.

5. IBM DB2

IBM Db2 database is a relational database that delivers advanced data management and analytics capabilities for your transactional and warehousing workloads. This operational database is designed to deliver high performance, actionable insights, data availability, and reliability, and it is supported across Linux, Unix, and Windows operating systems.

The Db2 database software includes advanced features such as in-memory technology, advanced management and development tools, storage optimization, workload management, actionable compression, and continuous data availability.

With version 11.5, Db2 includes added AI functionality so that businesses can embrace forward-looking data science and AI technologies to retain competitive differentiation. Now you can leverage one platform that is both powered by AI and built for AI to optimize performance and availability while empowering your data scientists to find deeper insights.

Db2 Database can now be purchased as part of the IBM Hybrid Data Management Platform. With a single license, you can change the deployment mix of products at any time, and only get charged for what you use.

Db2 is now powered by AI. It allows you to use machine learning to tune workloads and optimize queries for significantly faster querying.

Db2 is fast to deploy and you can get up and running on your laptop within minutes, and start building applications quickly with a full featured database.

Db2 supports multiple data types and languages including NoSQL, pureXML, Graph and JSON. You can build in Java, .Net, Ruby, Python, Perl and more.

Db2 database is designed for high performance with in-memory technology, storage optimization, and workload management.

Db2 provides advanced protection with the help of built-in native encryption with enterprise key management.

6. Microsoft Access

Believe it or not. MS Access still remains one of the top 10 databases for local data storages. Access is usually not used in remote or centralized storage. It is used for local small databases.

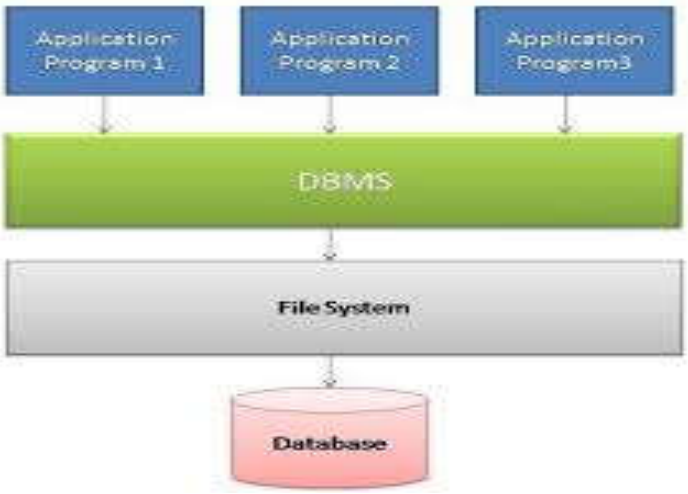
➢ 1.2Difference between DBMS and RDBMS

Parameter	DBMS	RDBMS
Storage	DBMS stores data as a file.	Data is stored in the form of tables.
Database structure	DBMS system, stores data in either a navigational or hierarchical form.	RDBMS uses a tabular structure where the headers are the column names, and the rows contain corresponding values
Number of Users	DBMS supports single user only.	It supports multiple users.
ACID	In a regular database, the data may not be stored following the ACID model. This can develop inconsistencies in the database.	Relational databases are harder to construct, but they are consistent and well structured. They obey ACID (Atomicity, Consistency, Isolation, Durability).
Type of program	It is the program for managing the databases on the computer networks and the system hard disks.	It is the database systems which are used for maintaining the relationships among the tables.
Hardware and software needs.	Low software and hardware needs.	Higher hardware and software need.
Integrity constraints	DBMS does not support the integrity constants. The integrity constants are not imposed at the file level.	RDBMS supports the integrity constraints at the schema level. Values beyond a defined range cannot be stored into the particular RDMS column.
Normalization	DBMS does not support Normalization	RDBMS can be Normalized.
Distributed Databases	DBMS does not support distributed database.	RBMS offers support for distributed databases.
Ideally suited for	DBMS system mainly deals with small quantity of data.	RDMS is designed to handle a large amount of data.
Dr. E.F. Codd	Dbms satisfy less than seven of Dr.	Dbms satisfy 8 to 10 Dr. E.F.

Parameter	DBMS	RDBMS
Rules	E.F. Codd Rules	Codd Rules
Client Server	DBMS does not support client-server architecture	RDBMS supports client-server architecture.
Data Fetching	Data fetching is slower for the complex and large amount of data.	Data fetching is rapid because of its relational approach.
Data Redundancy	Data redundancy is common in this model.	Keys and indexes do not allow Data redundancy.
Data Relationship	No relationship between data	Data is stored in the form of tables which are related to each other with the help of foreign keys.
Security	There is no security.	Multiple levels of security. Log files are created at OS, Command, and object level.
Data Access	Data elements need to access individually.	Data can be easily accessed using SQL query. Multiple data elements can be accessed at the same time.
Examples	Examples of DBMS are a file system, XML, Windows Registry, etc.	Example of RDBMS is MySQL, Oracle, SQL Server, etc.

➢ 1.3 Relationship Between DBMS and Application Program

There is very close relationship between database management system and applications program. The application program provides the user-interface to send requests to database management system and to receive processed results from database management system. The database management system processed the requests and returns the results to the application program and also controls and manages the database.
In most of the database management systems, application programs are used for interfacing with database. The user communicates with the data system through the user-interface part of the application program. The application program is developed according to the requirements of the users. The application programmer must have the complete knowledge about database management system and databases used in it. Then he can easily develop the application program. The application program provides an easy and user-friendly user-interface to access the database.


Application Program 1
Application Program 2
Application Program3
DBMS
File System
Database

➢

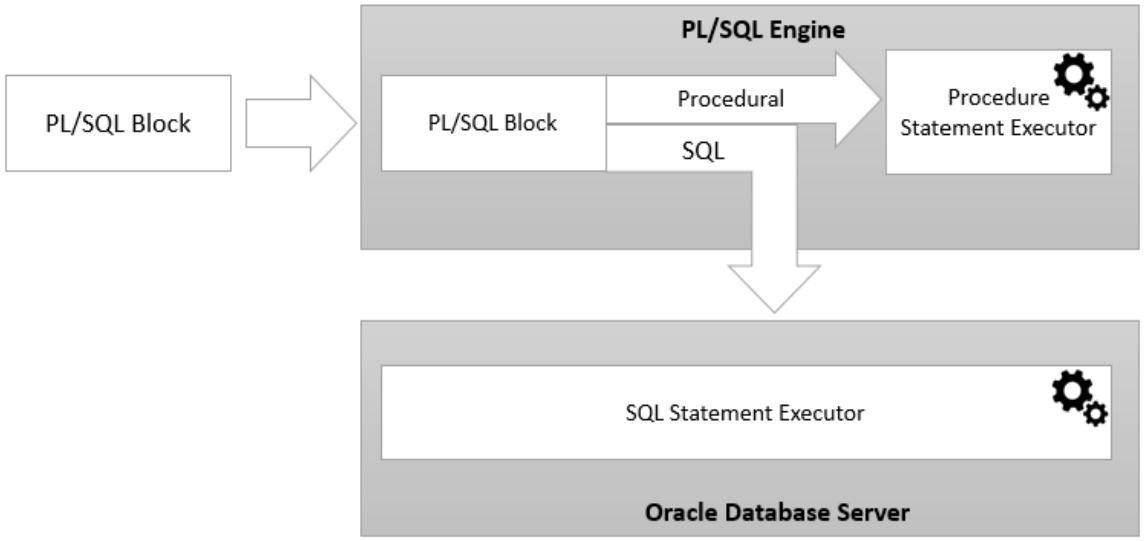
Unit II:PL-SQL

2.1 Overview of PLSQL

PL/SQL is a block structured language. The programs of PL/SQL are logical blocks that can contain any number of nested sub-blocks. Pl/SQL stands for "Procedural Language extension of SQL" that is used in Oracle. PL/SQL is integrated with Oracle database (since version 7). The functionalities of PL/SQL usually extended after each release of Oracle database. Although PL/SQL is closely integrated with SQL language, yet it adds some programming constraints that are not available in SQL.

PL/SQL architecture

The following picture illustrates the PL/SQL architecture:



PL/SQL engine is in charge of compiling PL/SQL code into byte-code and executes the executable code. The PL/SQL engine can only be installed in an Oracle Database server or an application development tool such as Oracle Forms.

Once you submit a PL/SQL block to the Oracle Database server, the PL/SQL engine collaborates with the SQL engine to compile and execute the code. PL/SQL engine runs the procedural elements while the SQL engine processes the SQL statements.

Now you should have a basic understanding of PL/SQL programming language and its architecture. Let's create the first working PL/SQL anonymous block.

2.2 Data Types

PL/SQL divides the scalar data types into four families:

- Number
- Boolean
- Character

- Datetime

A scalar data type may have subtypes. A subtype is a data type that is a subset of another data type, which is its base type. A subtype further defines a base type by restricting the value or size of the base data type.

Note that PL/SQL scalar data types include SQL data types and its own data type such as Boolean.

Numeric data types

The numeric data types represent real numbers, integers, and floating-point numbers. They are stored as `NUMBER`, IEEE floating-point storage types (`BINARY_FLOAT` and `BINARY_DOUBLE`).
The data types `NUMBER`, `BINARY_FLOAT`, and `BINARY_DOUBLE` are SQL data types.

Boolean data type

The `BOOLEAN` datatype has three data values: TRUE, FALSE, and NULL. Boolean values are typically used in control flow structure such as IF-THEN, CASE, and loop statements like LOOP, FOR LOOP, and WHILE LOOP.
SQL does not have the `BOOLEAN` data type, therefore, you cannot:

- Assign a `BOOLEAN` value to a table column.
- Select the value from a table column into a `BOOLEAN` variable.
- Use a `BOOLEAN` value in a SQL function.
- Use a `BOOLEAN` expression in a SQL statement.
- Use a `BOOLEAN` value in the `DBMS_OUTPUT.PUTLINE` and `DBMS_OUTPUT.PUT` subprograms.

Character data types

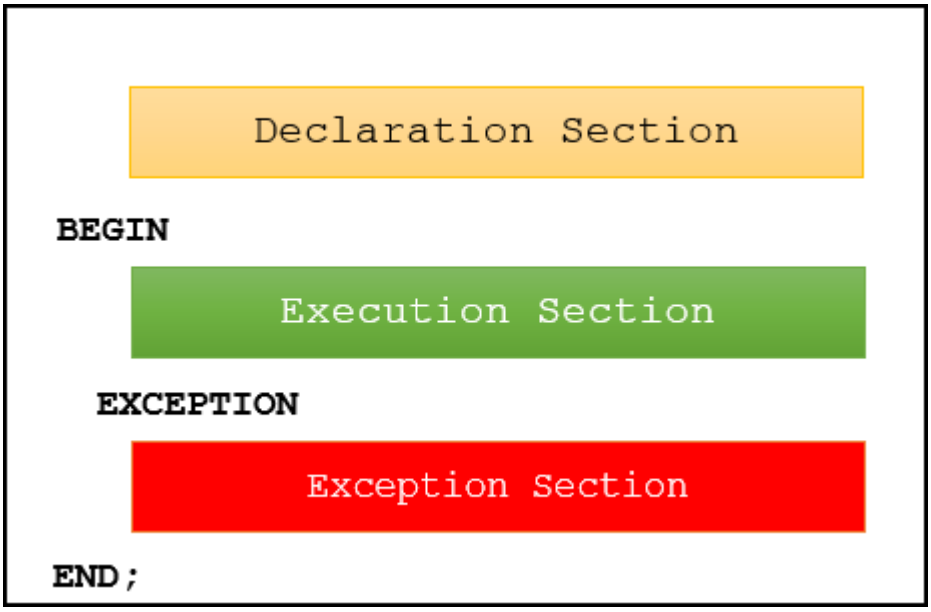
The character data types represent alphanumeric text. PL/SQL uses the SQL character data types such as `CHAR`, `VARCHAR2`, `LONG`, `RAW`, `LONG RAW`, `ROWID`, and `UROWID`.

- `CHAR(n)` is a fixed-length character type whose length is from 1 to 32,767 bytes.
- `VARCHAR2(n)` is varying length character data from 1 to 32,767 bytes.

Datetime data types

The datetime data types represent dates, timestamp with or without time zone and intervals. PL/SQL datetime data types are `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, `INTERVAL YEAR TO MONTH`, and `INTERVAL DAY TO SECOND`.

2.3 PLSQL Block
The following picture illustrates the structure of a PL/SQL block:



1) Declaration section

A PL/SQL block has a declaration section where you declare variables, allocate memory for cursors, and define data types.

2) Executable section

A PL/SQL block has an executable section. An executable section starts with the keyword `BEGIN` and ends with the keyword `END`. The executable section must have a least one executable statement, even if it is the `NULL` statement which does nothing.

3) Exception-handling section

A PL/SQL block has an exception-handling section that starts with the keyword `EXCEPTION`. The exception-handling section is where you catch and handle exceptions raised by the code in the execution section.

Note a block itself is an executable statement, therefore you can nest a block within other blocks.

PL/SQL anonymous block example

The following example shows a simple PL/SQL anonymous block with one executable section.

```
1 BEGIN
2    DBMS_OUTPUT.put_line ('Hello World!');
3 END;
```

The executable section calls the `DMBS_OUTPUT.PUT_LINE` procedure to display the `"Hello World"` message on the screen.

Execute a PL/SQL anonymous block using SQL*Plus

Once you have the code of an anonymous block, you can execute it using SQL*Plus, which is a command-line interface for executing SQL statement and PL/SQL blocks provided by Oracle Database.

The following picture illustrates how to execute a PL/SQL block using SQL*Plus:

```
SQL> SET SERVEROUTPUT ON;
SQL> BEGIN
  2     DBMS_OUTPUT.PUT_LINE('Hello World');
  3  END;
  4  /
Hello World

PL/SQL procedure successfully completed.
```

First, connect to the Oracle Database server using a username and password.Second, turn on the server output using the `SET SERVEROUTPUT ON` command so that the `DBMS_OUTPUT.PUT_LINE` procedure will display text on the screen.

Third, type the code of the block and enter a forward slash (`/`) to instruct SQL*Plus to execute the block. Once you type the forward-slash (/), SQL*Plus will execute the block and display the `Hello World` message on the screen

2.3.1 % type, % rowtype

Previous Next

Using %TYPE and %ROWTYPE Attributes to Declare Identical Datatypes

As part of the declaration for each PL/SQL variable, you declare its datatype. Usually, this datatype is one of the types shared between PL/SQL and SQL, such as `NUMBER` or `VARCHAR2`. For easier code maintenance that interacts with the database, you can also use the special qualifiers `%TYPE` and `%ROWTYPE` to declare variables that hold table columns or table rows.

This section contains the following topics:

- Using the %TYPE Attribute to Declare Variables
- Using the %ROWTYPE Attribute to Declare Variables

Using the %TYPE Attribute to Declare Variables

The `%TYPE` attribute provides the datatype of a variable or table column. This is particularly useful when declaring variables that will hold values of a table column. For example, suppose you want to declare variables as the same datatype as the `employee_id` and `last_name` columns in `employees` table. To declare variables

named `empid` and `emplname` that have the same datatype as the table columns, use dot notation and the `%TYPE` attribute. See Example: Using %TYPE With Table Columns in PL/SQL.

Using %TYPE With Table Columns in PL/SQL

```
DECLARE -- declare variables using %TYPE attribute
   empid    employees.employee_id%TYPE;  -- employee_id
datatype is NUMBER(6)
   emplname employees.last_name%TYPE;  -- last_name datatype
is VARCHAR2(25)
BEGIN
   empid    := 100301;  -- this is OK because it fits in
NUMBER(6)
--   empid  := 3018907;  -- this is too large and will cause
an overflow
   emplname := 'Patel'; --  this is OK because it fits in
VARCHAR2(25)
   DBMS_OUTPUT.PUT_LINE('Employee ID: ' || empid);  -- display
data
   DBMS_OUTPUT.PUT_LINE('Employee name: ' || emplname); --
display data
END;
/
```

Using the %ROWTYPE Attribute to Declare Variables

For easier maintenance of code that interacts with the database, you can use the `%ROWTYPE` attribute to declare a variable that represents a row in a table. A PL/SQL record is the datatype that stores the same information as a row in a table.

In PL/SQL, records are used to group data. A record consists of a number of related fields in which data values can be stored. The record can store an entire row of data selected from the table or fetched from a cursor or cursor variable. For information about records, see "Using Record Types".

Columns in a row and corresponding fields in a record have the same names and datatypes. In Example: Using %ROWTYPE with a PL/SQL Record, you declare a record named `emp_rec`. Its fields have the same names and datatypes as the columns in the `employees` table. You use dot notation to reference fields, such as `emp_rec.last_name`.

In Example: Using %ROWTYPE with a PL/SQL Record, the `SELECT` statement is used to store row information from the `employees` table into the `emp_rec` record. When you run the `SELECT INTO` statement, the value in the `first_name` column of the `employees` table is assigned to the `first_name` field of `emp_rec`; the value in the `last_name` column is assigned to the `last_name` field of `emp_rec`; and so on.

Using %ROWTYPE with a PL/SQL Record

```
DECLARE -- declare variables
-- declare record variable that represents a row fetched from
the employees table
   emp_rec employees%ROWTYPE; -- declare variable with
%ROWTYPE attribute
BEGIN
  SELECT * INTO emp_rec FROM EMPLOYEES WHERE employee_id =
120; -- retrieve record
  DBMS_OUTPUT.PUT_LINE('Employee name: ' || emp_rec.first_name
|| ' '
                       || emp_rec.last_name); -- display
END;
/
```

2.3.2 Operators, Functions, comparison, numeric, char,**date**

Precedence

Precedence is the order in which Oracle evaluates different operators in the same expression. When evaluating an expression containing multiple operators, Oracle evaluates operators with higher precedence before evaluating those with lower precedence. Oracle evaluates operators with equal precedence from left to right within an expression.

Table 3-1 lists the levels of precedence among SQL operators from high to low. Operators listed on the same line have the same precedence.

Table 3-1 SQL Operator Precedence

Operator	Operation
+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	comparison
NOT	exponentiation, logical negation
AND	conjunction
OR	disjunction

Arithmetic Operators

You can use an arithmetic operator in an expression to negate, add, subtract, multiply, and divide numeric values. The result of the operation is also a numeric value. Some of these operators are also used in date arithmetic. Table 3-2 lists arithmetic operators.

Table 3-2 Arithmetic Operators

Operator	Purpose	Example
+ -	Denotes a positive or negative expression. These are unary operators.	`SELECT * FROM orders WHERE qtysold = -1; SELECT * FROM emp WHERE -sal < 0;`
* /	Multiplies, divides. These are binary operators.	`UPDATE emp SET sal = sal * 1.1;`
+ -	Adds, subtracts. These are binary operators.	`SELECT sal + comm FROM emp WHERE SYSDATE - hiredate > 365;`

Concatenation Operator

The concatenation operator manipulates character strings. Table 3-3 describes the concatenation operator.

Table 3-3 Concatenation Operator

Operator	Purpose	Example
||	Concatenates character strings.	`SELECT 'Name is ' || ename FROM emp;`

Comparison Operators

Comparison operators compare one expression with another. The result of such a comparison can be TRUE, FALSE, or UNKNOWN. For information on conditions, see "Conditions". Table 3-4 lists comparison operators.

Table 3-4 Comparison Operators

Operator	Purpose	Example
=	Equality test.	`SELECT * FROM emp WHERE sal = 1500;`
!= ^= <> ¬=	Inequality test. Some forms of the inequality operator may be unavailable on some platforms.	`SELECT * FROM emp WHERE sal != 1500;`
> <	"Greater than" and "less than" tests.	`SELECT * FROM emp WHERE sal > 1500; SELECT * FROM emp WHERE sal < 1500;`

>=
 	
<=
	"Greater than or equal to" and "less than or equal to" tests.	SELECT * FROM emp
		  WHERE sal >= 1500;
		SELECT * FROM emp
		  WHERE sal <= 1500;

IN	"Equal to any member of" test. Equivalent to "= ANY".	SELECT * FROM emp
		  WHERE job IN
		
		('CLERK','ANALYST');
		SELECT * FROM emp
		  WHERE sal IN
		  (SELECT sal FROM
		emp
		  WHERE deptno =
		30);

NOT IN	Equivalent to "!=ALL". Evaluates to FALSE if any member of the set is NULL.	SELECT * FROM emp
		  WHERE sal NOT IN
		  (SELECT sal FROM
		emp
		  WHERE deptno =
		30);
		SELECT * FROM emp
		  WHERE job NOT IN
		  ('CLERK',
		ANALYST');

ANY
SOME
	Compares a value to each value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=.
	
	Evaluates to FALSE if the query returns no rows.
		SELECT * FROM emp
		  WHERE sal = ANY
		  (SELECT sal FROM
		emp
		  WHERE deptno =
		30);

ALL	Compares a value to every value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=.
	
	Evaluates to TRUE if the query returns no rows.
		SELECT * FROM emp
		  WHERE sal >=
		  ALL (1400, 3000);

[NOT]
BETWEEN
x AND y
	[Not] greater than or equal to *x* and less than or equal to *y*.	SELECT * FROM emp
		  WHERE sal
		  BETWEEN 2000 AND
		3000;

EXISTS	TRUE if a subquery returns at least one row.	SELECT ename, deptno
		  FROM dept
		  WHERE EXISTS
		  (SELECT * FROM emp
		  WHERE
		dept.deptno
		  = emp.deptno);

x [NOT] LIKE y	TRUE if *x* does [not] match the pattern *y*. Within *y*, the character "%" matches any string of zero or more	See "LIKE Operator".

[ESCAPE 'z']	characters except null. The character "_" matches any single character. Any character, excepting percent (%) and underbar (_) may follow ESCAPE; a wildcard character is treated as a literal if preceded by the character designated as the escape character.	SELECT * FROM tab1 WHERE col1 LIKE 'A_C/%E%' ESCAPE '/';
IS [NOT] NULL	Tests for nulls. This is the only operator that you should use to test for nulls. See "Nulls".	SELECT ename, deptno FROM emp WHERE comm IS NULL;

Logical Operators

A logical operator combines the results of two component conditions to produce a single result based on them or to invert the result of a single condition. Table 3-5 lists logical operators.

Table 3-5 Logical Operators

Operator	Function	Example
NOT	Returns TRUE if the following condition is FALSE. Returns FALSE if it is TRUE. If it is UNKNOWN, it remains UNKNOWN.	SELECT * FROM emp WHERE NOT (job IS NULL); SELECT * FROM emp WHERE NOT (sal BETWEEN 1000 AND 2000);
AND	Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE. Otherwise returns UNKNOWN.	SELECT * FROM emp WHERE job = 'CLERK' AND deptno = 10;
OR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE. Otherwise returns UNKNOWN.	SELECT * FROM emp WHERE job = 'CLERK' OR deptno = 10;

Number Functions

Number functions accept numeric input and return numeric values. This section lists the SQL number functions. Most of these functions return values that are accurate to 38 decimal digits. The transcendental functions COS, COSH, EXP, LN, LOG, SIN, SINH, SQRT, TAN, and TANH are accurate to 36 decimal digits. The

transcendental functions ACOS, ASIN, ATAN, and ATAN2 are accurate to 30 decimal digits.

ABS

Purpose Returns the absolute value of *n*.

Example

```
SELECT ABS(-15) "Absolute" FROM DUAL;

  Absolute
----------
        15
```

CEIL

Purpose Returns smallest integer greater than or equal to *n*.

Example

```
SELECT CEIL(15.7) "Ceiling" FROM DUAL;

   Ceiling
----------
        16
```

EXP

Purpose Returns e raised to the *n*th power; e = 2.71828183 ...

Example

```
SELECT EXP(4) "e to the 4th power" FROM DUAL;

e to the 4th power
------------------
          54.59815
```

FLOOR

Purpose Returns largest integer equal to or less than *n*.

Example

```
SELECT FLOOR(15.7) "Floor" FROM DUAL;

     Floor
----------
        15
```

MOD

Syntax `MOD(m,n)`

Purpose Returns remainder of *m* divided by *n*. Returns *m* if *n* is 0.

Example

```
SELECT MOD(11,4) "Modulus" FROM DUAL;
```

```
   Modulus
----------
         3
```

This function behaves differently from the classical mathematical modulus function when *m* is negative. The classical modulus can be expressed using the MOD function with this formula:

```
m - n * FLOOR(m/n)
```

The following statement illustrates the difference between the MOD function and the classical modulus:

```
SELECT m, n, MOD(m, n),
m - n * FLOOR(m/n) "Classical Modulus"
  FROM test_mod_table;

         M          N   MOD(M,N) Classical Modulus
---------- ---------- ---------- -----------------
        11          4          3                 3
        11         -4          3                -1
       -11          4         -3                 1
       -11         -4         -3                -3
```

POWER

Purpose

Returns *m* raised to the *n*th power. The base *m* and the exponent *n* can be any numbers, but if *m* is negative, *n* must be an integer.

Example

```
SELECT POWER(3,2) "Raised" FROM DUAL;

    Raised
----------
         9
```

ROUND

Syntax

```
ROUND(n[,m])
```

Purpose

Returns *n* rounded to *m* places right of the decimal point; if *m* is omitted, to 0 places. *m* can be negative to round off digits left of the decimal point. *m* must be an integer.

Example 1

```
SELECT ROUND(15.193,1) "Round" FROM DUAL;

     Round
----------
      15.2
```

Example 2

```
SELECT CHR(16705 USING NCHAR_CS)
FROM DUAL;

C
-
A
```

CONCAT

Syntax `CONCAT(char1, char2)`

Purpose Returns *char1* concatenated with *char2*. This function is equivalent to the concatenation operator (||). For information on this operator, see "Concatenation Operator".

Example This example uses nesting to concatenate three character strings:

```
SELECT CONCAT( CONCAT(ename, ' is a '), job) "Job"
FROM emp
WHERE empno = 7900;

Job
-----------------
JAMES is a CLERK
```

SUBSTR

Syntax `SUBSTR(char, m [,n])`

Purpose Returns a portion of *char*, beginning at character *m*, *n* characters long. If *m* is 0, it is treated as 1. If *m* is positive, Oracle counts from the beginning of *char* to find the first character. If *m* is negative, Oracle counts backwards from the end of *char*. If *n* is omitted, Oracle returns all characters to the end of *char*. If n is less than 1, a null is returned.

Floating-point numbers passed as arguments to *substr* are automatically converted to integers.

Example 1

```
SELECT SUBSTR('ABCDEFG',3.1,4) "Subs"
     FROM DUAL;

Subs
----
CDEF
```

UPPER

Syntax `UPPER(char)`

Purpose Returns *char*, with all letters uppercase. The return value has the same

datatype as the argument *char*.

Example
```
SELECT UPPER('Large') "Uppercase"
     FROM DUAL;

Upper
-----
LARGE
```

Character Functions Returning Number Values

This section lists character functions that return number values.

ASCII

Syntax
```
ASCII(char)
```

Purpose

Returns the decimal representation in the database character set of the first character of *char*. If your database character set is 7-bit ASCII, this function returns an ASCII value. If your database character set is EBCDIC Code Page 500, this function returns an EBCDIC value. Note that there is no similar EBCDIC character function.

Example
```
SELECT ASCII('Q')
     FROM DUAL;

ASCII('Q')
----------
        81
```

Date Functions

Date functions operate on values of the DATE datatype. All date functions return a value of DATE datatype, except the MONTHS_BETWEEN function, which returns a number.

ADD_MONTHS

Syntax
```
ADD_MONTHS(d,n)
```

Purpose

Returns the date *d* plus *n* months. The argument *n* can be any integer. If *d* is the last day of the month or if the resulting month has fewer days than the day component of *d*, then the result is the last day of the resulting month. Otherwise, the result has the same day component as *d*.

Example
```
SELECT TO_CHAR(
     ADD_MONTHS(hiredate,1),
     'DD-MON-YYYY') "Next month"
     FROM emp
     WHERE ename = 'SMITH';
```

```
Next Month
-----------
17-JAN-1981
```

MONTHS_BETWEEN

Syntax `MONTHS_BETWEEN(d1, d2)`

Purpose Returns number of months between dates *d1* and *d2*. If *d1* is later than *d2*, result is positive; if earlier, negative. If *d1* and *d2* are either the same days of the month or both last days of months, the result is always an integer; otherwise Oracle calculates the fractional portion of the result based on a 31-day month and considers the difference in time components of *d1* and *d2*.

Example
```
SELECT MONTHS_BETWEEN
   (TO_DATE('02-02-1995','MM-DD-YYYY'),
    TO_DATE('01-01-1995','MM-DD-YYYY') ) "Months"
    FROM DUAL;

    Months
----------
1.03225806
```

SYSDATE

Syntax `SYSDATE`

Purpose Returns the current date and time. Requires no arguments. In distributed SQL statements, this function returns the date and time on your local database. You cannot use this function in the condition of a CHECK constraint.

Example
```
SELECT TO_CHAR
     (SYSDATE, 'MM-DD-YYYY HH24:MI:SS')"NOW"
      FROM DUAL;
NOW
-------------------
10-29-1993 20:27:11
```

2.3.3 Control Statement

The `IF` statement has the three forms:

- `IF THEN`
- `IF THEN ELSE`
- `IF THEN ELSIF`

PL/SQL IF THEN statement

The following illustrates the structure of the `IF THEN` statement:

```
1 IF condition THEN
2     statements;
3 END IF;
```

The condition is a Boolean expression that always evaluates to TRUE, FALSE, or NULL.

If the condition evaluates to TRUE, the `statements` after the `THEN` execute. Otherwise, the `IF` statement does nothing.

PL/SQL IF THEN statement example

In the following example, the statements between `THEN` and `END IF` execute because the sales revenue is greater than 100,000.

```
1 DECLARE n_sales NUMBER := 2000000;
2 BEGIN
3   IF n_sales > 100000 THEN
4     DBMS_OUTPUT.PUT_LINE( 'Sales revenue is greater than 100K ' );
5   END IF;
6 END;
```

PL/SQL IF THEN ELSE statement

The `IF THEN ELSE` statement has the following structure:

```
1 IF condition THEN
2    statements;
3 ELSE
4    else_statements;
5 END IF;
```

If the condition evaluates to TRUE, then the statements between `THEN` and `ELSE` execute. In case the condition evaluates to FALSE or NULL, the `else_statements` between `ELSE` and `END IF` executes.

IF THEN ELSE statement example

The following example sets the sales commission to 10% if the sales revenue is greater than 200,000. Otherwise, the sales commission is set to 5%.

```
1   DECLARE
2    n_sales NUMBER := 300000;
3    n_commission NUMBER( 10, 2 ) := 0;
4   BEGIN
5    IF n_sales > 200000 THEN
6      n_commission := n_sales * 0.1;
7    ELSE
8      n_commission := n_sales * 0.05;
9    END IF;
10 END;
```

PL/SQL IF THEN ELSIF statement

The following illustrates the structure of the `IF THEN ELSIF` statement:

```
1   IF condition_1 THEN
2    statements_1
3   ELSIF condition_2 THEN
4    statements_2
5   [ ELSIF condition_3 THEN
6      statements_3
7   ]
8   ...
9   [ ELSE
10    else_statements
11 ]
12 END IF;
```

In this structure, the condition between `IF` and `THEN`, which is the first condition, is always evaluated. Each other condition between `ELSEIF` and `THEN` is evaluated only if the preceding condition is FALSE. For example, the `condition_2` is evaluated only if

the `condition_1` is false, the `condition_3` is evaluated only if the `condition_2` is false, and so on.
If a condition is true, other subsequent conditions are not evaluated. If no condition is true, the `else_statements` between the `ELSE` and `ENDIF` execute. In case you skip the the `ELSE` clause and no condition is TRUE, then the `IF THEN ELSIF` does nothing
IF THEN ELSIF statement example

The following example uses the `IF THEN ELSIF` statement to set the sales commission based on the sales revenue.

```
1  DECLARE
2    n_sales NUMBER := 300000;
3    n_commission NUMBER( 10, 2 ) := 0;
4  BEGIN
5    IF n_sales > 200000 THEN
6      n_commission := n_sales * 0.1;
7    ELSIF n_sales <= 200000 AND n_sales > 100000 THEN
8      n_commission := n_sales * 0.05;
9    ELSIF n_sales <= 100000 AND n_sales > 50000 THEN
10     n_commission := n_sales * 0.03;
11   ELSE
12     n_commission := n_sales * 0.02;
13   END IF;
14 END;
```

Nested IF statement

You can nest an `IF` statement within another `IF` statement as shown below:

```
1 IF condition_1 THEN
2    IF condition_2 THEN
3       nested_if_statements;
4    END IF;
5 ELSE
6    else_statements;
7 END IF;
```

However, if you have too many levels of nesting, the code will be hard to read and maintain, so you should avoid nesting the `IF` statements.

What is CASE Statement?

A CASE statement is similar to IF-THEN-ELSIF statement that selects one alternative based on the condition from the available options.

- CASE statement uses "selector" rather than a Boolean expression to choose the sequence.
- The value of the expression in the CASE statement will be treated as a selector.
- The expression could be of any type (arithmetic, variables, etc.)
- Each alternative is assigned with a certain pre-defined value (selector), and the alternative with selector value that matches the conditional expression value will get executed.
- Unlike IF-THEN-ELSIF, the CASE statement can also be used in SQL statements.

- ELSE block in CASE statement holds the sequence that needs to be executed when none of the alternatives got selected.

Syntax:

```
CASE (expression)
 WHEN <valuel> THEN action_blockl;
 WHEN <value2> THEN action_block2;
 WHEN <value3> THEN action_block3;
 ELSE action_block_default;
END CASE;
```

- In the above syntax, the expression will return a value that could be of any type (variable, number, etc.).
- Each 'WHEN' clause is treated as an alternatives which have <value> and <action_block>.
- The 'WHEN' clause which matches the value as that of the expression will be selected, and the corresponding <action_block> will be executed.
- 'ELSE' block is optional which hold the <action_block_default> that needs to be executed when none of the alternatives match the expression value.
- The 'END' marks the end of the CASE statement, and it is a mandatory part of the CASE.

Example 1: Arithmetic Calculation using Case

In this example, we are going to do arithmetic calculation between two numbers 55 and 5.

```
DECLARE
a NUMBER :=55;
b NUMBER :=5;
arth_operation VARCHAR2(20) :='MULTIPLY’;
BEGIN
dbms_output.put_line(‘Program started.' );
CASE (arth_operation)
WHEN ‘ADD’ THEN dbms_output.put_line(‘Addition of the numbers are: '|| a+b );
WHEN ‘SUBTRACT' THEN dbms_output.put_line(‘Subtraction of the numbers are: '||a-b
);
WHEN ‘MULTIPLY' THEN dbms_output.put_line(‘Multiplication of the numbers are: '||
a*b
);
WHEN ‘DIVIDE' THEN dbms_output.put_line(‘Division of the numbers are:'|| a/b);
ELSE dbms_output.put_line(‘No operation action defined. Invalid operation');
END CASE;
```

```
dbms_output.put_line(‘Program completed.' );
END;
/
```

Types of Loop in PL/SQL

PL/SQL provides following three types of loops

- Basic loop statement
- For loop statement
- While loop statement

Basic Loop Statement

This loop statement is the simplest loop structure in PL/SQL. The execution block starts with keyword 'LOOP' and ends with the keyword 'END LOOP'.

The exit condition should be given inside this execution block so that control exit from the loop.

It needs EXIT keyword to be given explicitly in the execution part to exit from the loop.




```
 LOOP
<execution block starts>
<EXIT condition based on developer criteria>
<execution_block_ends>
END LOOP;
```

Syntax Explanation:

- In the above syntax, key word 'LOOP' marks the beginning of the loop and 'END LOOP' marks the end of the loop.
- The execution block contains all the code that needs to be executed including the EXIT condition.
- The execution part can contain any execution statement.

Note: Basic loop statement with no EXIT keyword will be an INFINITE-LOOP that will never stop.

Example 1: In this example, we are going to print number from 1 to 5 using basic loop statement. For that, we will execute the following code.

```
1.  DECLARE
2.  a NUMBER :=1;
3.  BEGIN
4.  dbms_output.put_line('Program started.' );
5.  LOOP
6.  dbms_output.put_line(a);
7.  a:=a+1;
8.  EXIT WHEN a>5;
9.  END LOOP;
10. dbms_output.put_line('Program completed.' );
11. END;
12. /
```

Basic Loop

Output:

```
Program started.
1
2
3
4
5
Program completed.
```

What is For Loop?

"FOR LOOP" statement is best suitable when you want to execute a code for a known number of times rather than based on some other conditions.

In this loop, the lower limit and the higher limit will be specified and as long as the loop variable is in between this range, the loop will be executed.

The loop variable is self-incremental, so no explicit increment operation is needed in this loop. The loop variable need not to be declared, as it is declared implicitly.

```
FOR <loop_variable> in <lower_limit> .. <higher_limit>
LOOP
<execution block starts>
.
.
.
<execution_block_ends>
 END LOOP;
```

Syntax Explanation:

- In the above syntax, keyword 'FOR' marks beginning of the loop and 'END LOOP' marks the end of the loop.
- Loop variable is evaluated every time before executing the execution part.
- The execution block contains all the code that needs to be executed. The execution part can contain any execution statement.
- The loop_variable is declared implicitly during the execution of the entire loop, and the scope of this loop_variable will be only inside this loop.
- If the loop variable came out of the range, then control will exit from the loop.
- The loop can be made to work in the reverse order by adding the keyword 'REVERSE' before lower_limit.

Example 1: In this example, we are going to print number from 1 to 5 using FOR loop statement. For that, we will execute the following code.

```
1.  BEGIN
2.  dbms_output.put_line('Program started.' );
3.  FOR a IN 1 .. 5
4.  LOOP
5.  dbms_output.put_line(a);
6.  END LOOP;
7.  dbms_output.put_line('Program completed.' );
8.  END;
9.  /
```

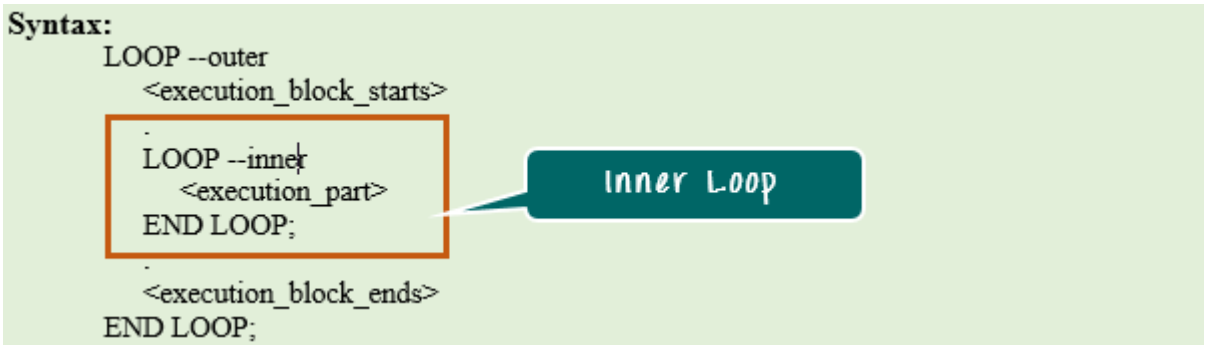
Range Specification

FOR Loop

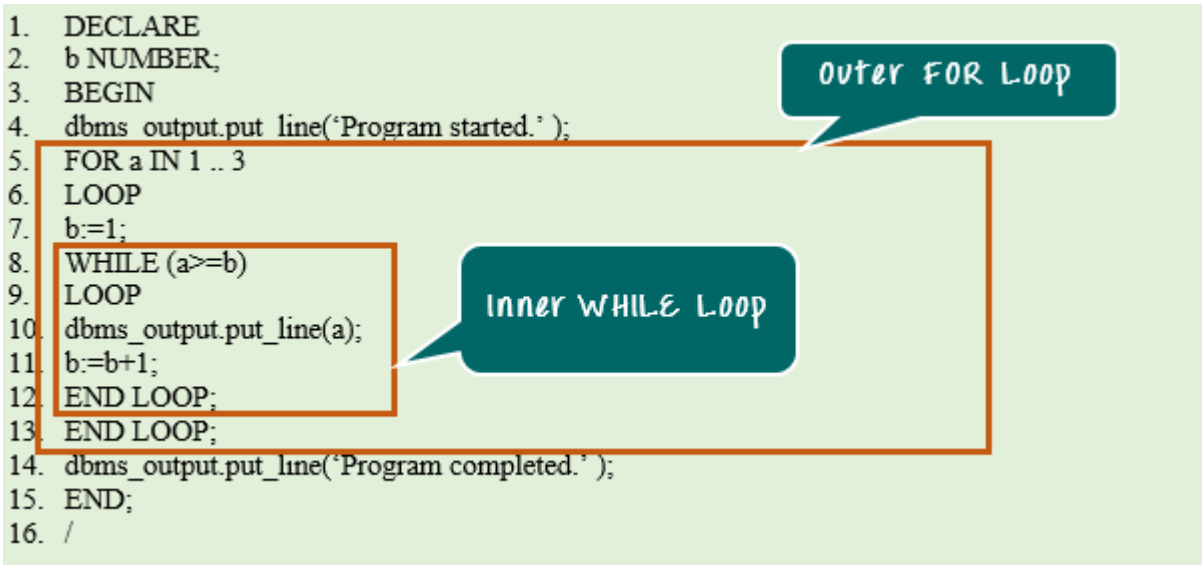
Output:

```
Program started.
1
2
3
4
5
Program completed.
```

The loop statements can also be nested. The outer and inner loop can be of different types. In the nested loop, for every one iteration value of the outer loop, the inner loop will be executed fully.



Example 1: In this example, we are going to print number from 1 to 3 using FOR loop statement. Each number will be printed as many times as its value. For that, we will execute the following code.



What is While Loop?

WHILE loop statement works similar to the Basic loop statement except **the EXIT condition is at the very beginning of the loop**.

It works like an entry-check loop in which execution block will not even be executed once if the condition is not satisfied, as the exit condition is checking before execution part. It does not require keyword 'EXIT' explicitly to exit from the loop since it is validating the condition implicitly each time of the loop.

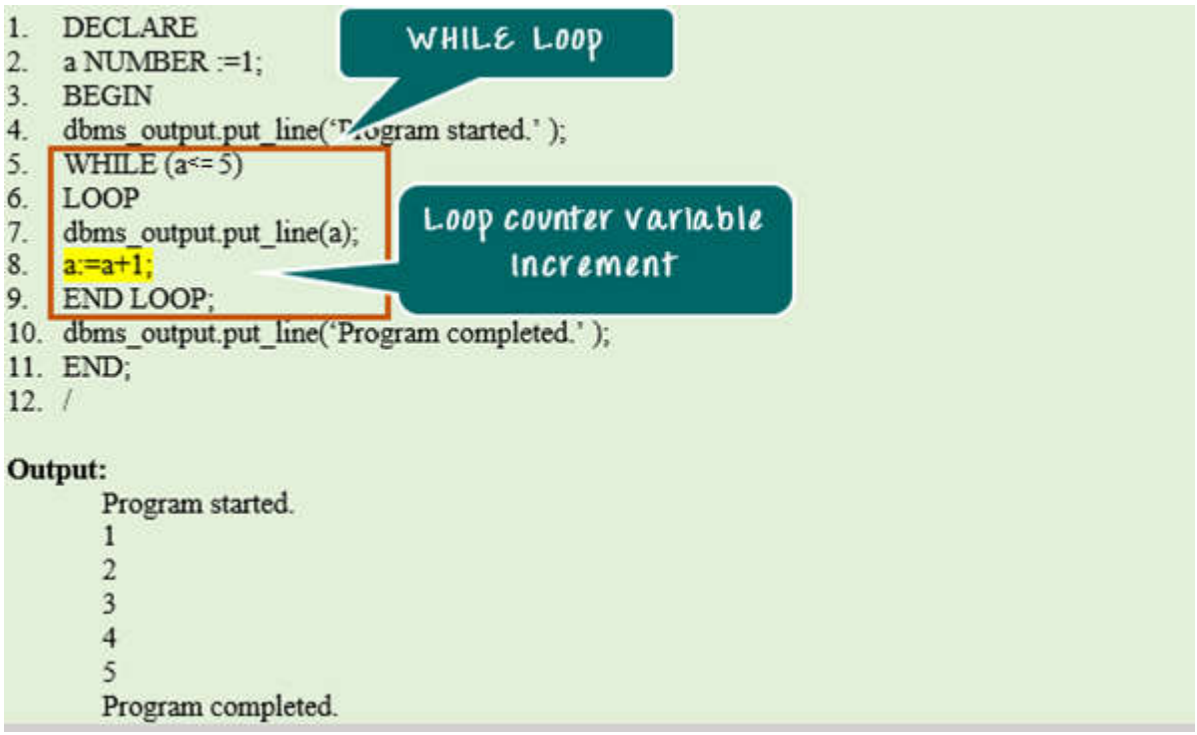
```
WHILE <EXIT condition>
 LOOP
<execution block starts>
.
.
.
```

```
<execution_block_ends>
 END LOOP;
```

Syntax Explanation:

- In the above syntax, keyword 'WHILE' marks beginning of the loop and 'END LOOP' marks the end of the loop.
- EXIT condition is evaluated each time before the execution part is starting executing.
- The execution block contains all the code that needs to be executed.
- The execution part can contain any execution statement.

Example 1: In this example, we are going to print number from 1 to 4 using WHILE loop statement. For that, we will execute the following code.




```
1.  DECLARE
2.  a NUMBER :=1;
3.  BEGIN
4.  dbms_output.put_line('Program started.' );
5.  WHILE (a<=5)
6.  LOOP
7.  dbms_output.put_line(a);
8.  a:=a+1;
9.  END LOOP;
10. dbms_output.put_line('Program completed.' );
11. END;
12. /
```

Output:

```
Program started.
1
2
3
4
5
Program completed.
```

What is Exception Handling in PL/SQL?

An exception occurs when the PL/SQL engine encounters an instruction which it cannot execute due to an error that occurs at run-time. These errors will not be captured at the time of compilation and hence these needed to handle only at the run-time.

For example, if PL/SQL engine receives an instruction to divide any number by '0', then the PL/SQL engine will throw it as an exception. The exception is only raised at the run-time by the PL/SQL engine.

Exceptions will stop the program from executing further, so to avoid such condition, they need to be captured and handled separately. This process is called as Exception-Handling, in which the programmer handles the exception that can occur at the run time.

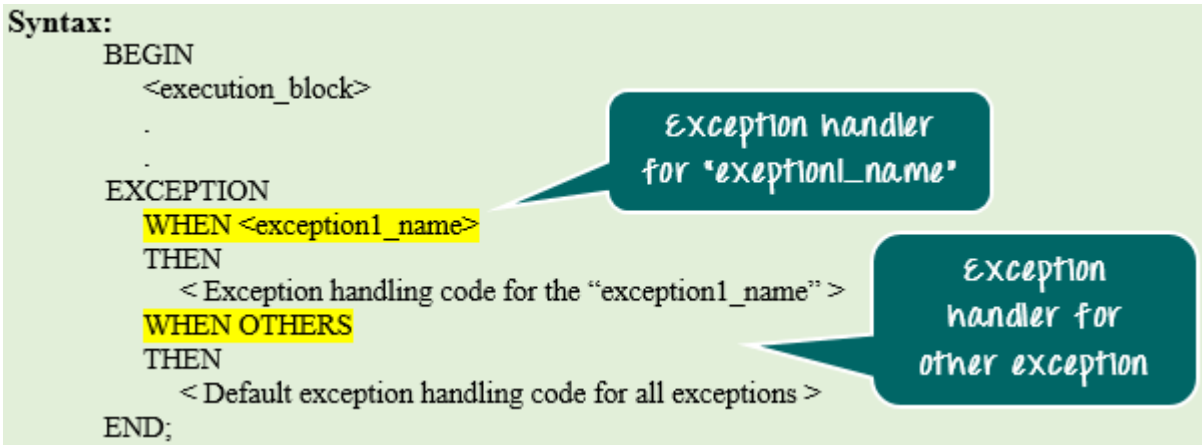
In this tutorial, you will learn the following topics-

- Exception-Handling Syntax
- Types of Exception
- Predefined Exceptions
- User-defined Exception
- PL/SQL Raise Exception
- Important points to note in Exception

Exception-Handling Syntax

Exceptions are handled at the block, level, i.e., once if any exception occurs in any block then the control will come out of execution part of that block. The exception will then be handled at the exception handling part of that block. After handling the exception, it is not possible to resend control back to the execution section of that block.

The below syntax explains how to catch and handle the exception.




```
BEGIN
<execution block>
.
.
EXCEPTION
WHEN <exception1_name>
THEN
  <Exception handling code for the "exception 1 _name'' >
WHEN OTHERS
```

```
THEN
  <Default exception handling code for all exceptions >
END;
```

Syntax Explanation:

- In the above syntax, the exception-handling block contains series of WHEN condition to handle the exception.
- Each WHEN condition is followed by the exception name which is expected to be raised at the run time.
- When any exception is raised at runtime, then the PL/SQL engine will look in the exception handling part for that particular exception. It will start from the first 'WHEN' clause and, sequentially it will search.
- If it found the exception handling for the exception which has been raised, then it will execute that particular handling code part.
- If none of the 'WHEN' clause is present for the exception which has been raised, then PL/SQL engine will execute the 'WHEN OTHERS' part (if present). This is common for all the exception.
- After executing the exception, part control will go out of the current block.
- Only one exception part can be executed for a block at run-time. After executing it, the controller will skip the remaining exception handling part and will go out of the current block.

Note: WHEN OTHERS should always be at the last position of the sequence. The exception handling part present after WHEN OTHERS will never get executed as the control will exit from the block after executing the WHEN OTHERS.

Types of Exception

There are two types of Exceptions in PI/SQL.

1. Predefined Exceptions
2. User-defined Exception

Predefined Exceptions

Oracle has predefined some common exception. These exceptions have a unique exception name and error number. These exceptions are already defined in the 'STANDARD' package in Oracle. In code, we can directly use these predefined exception name to handle them.

Below are the few predefined exceptions

Exception	Error Code	Exception Reason
CURSOR_ALREADY_OPEN	ORA-06511	Trying to open a cursor which is already opened
INVALID_CURSOR	ORA-01001	Illegal cursor operations like closing an unopened cursor
INVALID_NUMBER	ORA-01722	Conversion of character to a number failed due to invalid number character
NO_DATA_FOUND	ORA-01403	When 'SELECT' statement that contains INTO clause fetches no rows.
TOO_MANY_ROWS	ORA-01422	When a 'SELECT' statement with INTO clause returns more than one row
VALUE_ERROR	ORA-06502	Arithmetic or size constraint error (eg: assigning a value to a variable that is larger than the variable size)
ZERO_DIVIDE	ORA-01476	Dividing a number by '0'

User-defined Exception

In Oracle, other than the above-predefined exceptions, the programmer can create their own exception and handle them. They can be created at a subprogram level in the declaration part. These exceptions are visible only in that subprogram. The exception that is defined in the package specification is public exception, and it is visible wherever the package is accessible. <

Syntax: At subprogram level

```
DECLARE
<exception_name> EXCEPTION;
BEGIN
<Execution block>
EXCEPTION
WHEN <exception_name> THEN
<Handler>
END;
```

- In the above syntax, the variable 'exception_name' is defined as 'EXCEPTION' type.
- This can be used as in a similar way as a predefined exception.

Syntax: At Package Specification level

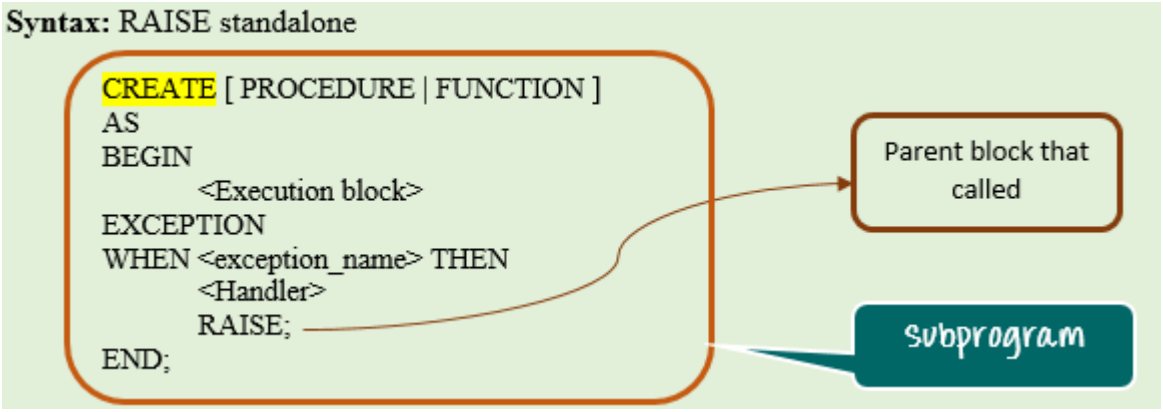
```
CREATE PACKAGE <package_name>
 IS
<exception_name> EXCEPTION;
.
.
END <package_name>;
```

- In the above syntax, the variable 'exception_name' is defined as 'EXCEPTION' type in the package specification of <package_name>.
- This can be used in the database wherever package 'package_name' can be called.

PL/SQL Raise Exception

All the predefined exceptions are raised implicitly whenever the error occurs. But the user-defined exceptions needs to be raised explicitly. This can be achieved using the keyword 'RAISE'. This can be used in any of the ways mentioned below.

If 'RAISE' is used separately in the program, then it will propagate the already raised exception to the parent block. Only in exception block can be used as shown below.



Syntax Explanation:

- In the above syntax, the keyword RAISE is used in the exception handling block.
- Whenever program encounters exception "exception_name", the exception is handled and will be completed normally
- But the keyword 'RAISE' in the exception handling part will propagate this particular exception to the parent program.

2.5 Functions , Procedures

What is Procedure in PL/SQL?

A Procedure is a subprogram unit that consists of a group of PL/SQL statements. Each procedure in Oracle has its own unique name by which it can be referred. This subprogram unit is stored as a database object. Below are the characteristics of this subprogram unit.

Note: Subprogram is nothing but a procedure, and it needs to be created manually as per the requirement. Once created they will be stored as database objects.

- Procedures are standalone blocks of a program that can be stored in the database.
- Call to these procedures can be made by referring to their name, to execute the PL/SQL statements.
- It is mainly used to execute a process in PL/SQL.
- It can have nested blocks, or it can be defined and nested inside the other blocks or packages.
- It contains declaration part (optional), execution part, exception handling part (optional).
- The values can be passed into the procedure or fetched from the procedure through parameters.

- These parameters should be included in the calling statement.
- Procedure can have a RETURN statement to return the control to the calling block, but it cannot return any values through the RETURN statement.
- Procedures cannot be called directly from SELECT statements. They can be called from another block or through EXEC keyword.

Syntax:

```
CREATE OR REPLACE PROCEDURE
<procedure_name>
        (
        <parameterl IN/OUT <datatype>
        ..
        .
        )
[ IS | AS ]
        <declaration_part>
BEGIN
        <execution part>
EXCEPTION
        <exception handling part>
END;
```

- CREATE PROCEDURE instructs the compiler to create new procedure. Keyword 'OR REPLACE' instructs the compile to replace the existing procedure (if any) with the current one.
- Procedure name should be unique.
- Keyword 'IS' will be used, when the procedure is nested into some other blocks. If the procedure is standalone then 'AS' will be used. Other than this coding standard, both have the same meaning.

Example1: Creating Procedure and calling it using EXEC

In this example, we are going to create a procedure that takes the name as input and prints the welcome message as output. We are going to use EXEC command to call procedure.

```
CREATE OR REPLACE PROCEDURE welcome_msg (p_name IN VARCHAR2)
IS
BEGIN
dbms_output.put_line (‘Welcome '|| p_name);
END;
/
EXEC welcome_msg (‘Guru99’);
```

What is Function?

Functions is a standalone PL/SQL subprogram. Like PL/SQL procedure, functions have a unique name by which it can be referred. These are stored as PL/SQL database objects. Below are some of the characteristics of functions.

- Functions are a standalone block that is mainly used for calculation purpose.
- Function use RETURN keyword to return the value, and the datatype of this is defined at the time of creation.
- A Function should either return a value or raise the exception, i.e. return is mandatory in functions.
- Function with no DML statements can be directly called in SELECT query whereas the function with DML operation can only be called from other PL/SQL blocks.
- It can have nested blocks, or it can be defined and nested inside the other blocks or packages.
- It contains declaration part (optional), execution part, exception handling part (optional).
- The values can be passed into the function or fetched from the procedure through the parameters.
- These parameters should be included in the calling statement.
- Function can also return the value through OUT parameters other than using RETURN.
- Since it will always return the value, in calling statement it always accompanies with assignment operator to populate the variables.

Syntax:

```
CREATE OR REPLACE FUNCTION
<procedure_name>
        (
        <parameter1 IN/OUT <datatype>
        ..
        .
        )
        RETURN <datatype>
[ IS | AS ]
        <declaration_part>
BEGIN
        <execution part>
EXCEPTION
        <exception handling part>
END;
```

Syntax

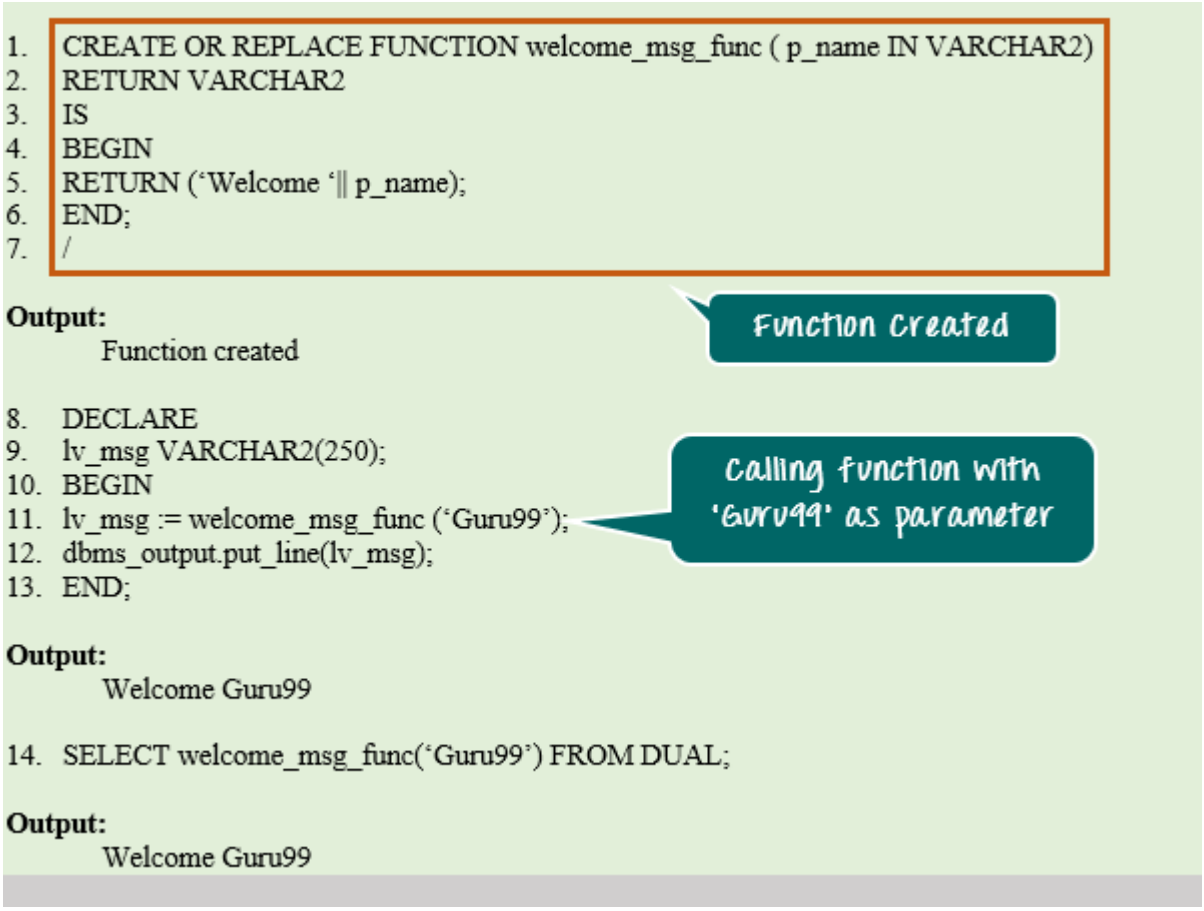
```
CREATE OR REPLACE FUNCTION
```

```
<procedure_name>
(
<parameterl IN/OUT <datatype>
)
RETURN <datatype>
[ IS | AS ]
<declaration_part>
BEGIN
<execution part>
EXCEPTION
<exception handling part>
END;
```

- CREATE FUNCTION instructs the compiler to create a new function. Keyword 'OR REPLACE' instructs the compiler to replace the existing function (if any) with the current one.
- The Function name should be unique.
- RETURN datatype should be mentioned.
- Keyword 'IS' will be used, when the procedure is nested into some other blocks. If the procedure is standalone then 'AS' will be used. Other than this coding standard, both have the same meaning.

Example1: Creating Function and calling it using Anonymous Block

In this program, we are going to create a function that takes the name as input and returns the welcome message as output. We are going to use anonymous block and select statement to call the function.



What is CURSOR in PL/SQL?

A Cursor is a pointer to this context area. Oracle creates context area for processing an SQL statement which contains all information about the statement.

PL/SQL allows the programmer to control the context area through the cursor. A cursor holds the rows returned by the SQL statement. The set of rows the cursor holds is referred as active set. These cursors can also be named so that they can be referred from another place of the code.

In this tutorial you will learn-

- Implicit Cursor
- Explicit Cursor
- Parameterised cursor
- Cursor Attributes
- FOR Loop Cursor statement

The cursor is of two types.

- Implicit Cursor
- Explicit Cursor

Implicit Cursor

Whenever any DML operations occur in the database, an implicit cursor is created that holds the rows affected, in that particular operation. These cursors cannot be named and, hence they cannot be controlled or referred from another place of the code. We can refer only to the most recent cursor through the cursor attributes.

Explicit Cursor

Programmers are allowed to create named context area to execute their DML operations to get more control over it. The explicit cursor should be defined in the declaration section of the PL/SQL block, and it is created for the 'SELECT' statement that needs to be used in the code.

Below are steps that involved in working with explicit cursors.

- **Declaring the cursor**

  Declaring the cursor simply means to create one named context area for the 'SELECT' statement that is defined in the declaration part. The name of this context area is same as the cursor name.

- **Opening Cursor**

  Opening the cursor will instruct the PL/SQL to allocate the memory for this cursor. It will make the cursor ready to fetch the records.

- **Fetching Data from the Cursor**

  In this process, the 'SELECT' statement is executed and the rows fetched is stored in the allocated memory. These are now called as active sets. Fetching data from the cursor is a record-level activity that means we can access the data in a record-by-record way.

  Each fetch statement will fetch one active set and holds the information of that particular record. This statement is same as 'SELECT' statement that fetches the record and assigns to the variable in the 'INTO' clause, but it will not throw any exceptions.

- **Closing the Cursor**

Once all the record is fetched now, we need to close the cursor so that the memory allocated to this context area will be released.

Syntax:

```
DECLARE
CURSOR <cursor_name> IS <SELECT statement^>
<cursor_variable declaration>
BEGIN
OPEN <cursor_name>;
FETCH <cursor_name> INTO <cursor_variable>;
.
.
CLOSE <cursor_name>;
END;
```

- In the above syntax, the declaration part contains the declaration of the cursor and the cursor variable in which the fetched data will be assigned.
- The cursor is created for the 'SELECT' statement that is given in the cursor declaration.
- In execution part, the declared cursor is opened, fetched and closed.

Parameterised Cursor

An explicit cursor may accept a list of parameters. Each time you open the cursor, you can pass different arguments to the cursor, which results in different result sets.

The following shows the syntax of a declaring a cursor with parameters:

```
1 CURSOR cursor_name (parameter_list)
2 IS
3 cursor_query;
```

In the cursor query, each parameter in the parameter list can be used anywhere which a constant is used. The cursor parameters cannot be referenced outside of the cursor query.

To open a cursor with parameters, you use the following syntax:

```
1 OPEN cursor_name (value_list);
```

In this syntax, you passed arguments corresponding to the parameters of the cursor.

Cursors with parameters are also known as parameterized cursors.

PL/SQL cursor with parameters example

The following example illustrates how to use a cursor with parameters:

```
1  DECLARE
2     r_product products%rowtype;
3     CURSOR c_product (low_price NUMBER, high_price NUMBER)
4     IS
5        SELECT *
6        FROM products
7        WHERE list_price BETWEEN low_price AND high_price;
8  BEGIN
9     -- show mass products
10    dbms_output.put_line('Mass products: ');
11    OPEN c_product(50,100);
12    LOOP
13       FETCH c_product INTO r_product;
14       EXIT WHEN c_product%notfound;
15       dbms_output.put_line(r_product.product_name || ': ' ||r_product.list_price);
16    END LOOP;
17    CLOSE c_product;
18
19    -- show luxury products
20    dbms_output.put_line('Luxury products: ');
21    OPEN c_product(800,1000);
22    LOOP
23       FETCH c_product INTO r_product;
24       EXIT WHEN c_product%notfound;
25       dbms_output.put_line(r_product.product_name || ': ' ||r_product.list_price);
26    END LOOP;
27    CLOSE c_product;
28
29 END;
30 /
```

Cursor Attributes

Both Implicit cursor and the explicit cursor has certain attributes that can be accessed. These attributes give more information about the cursor operations. Below are the different cursor attributes and their usage.

Cursor Attribute	Description
%FOUND	It returns the Boolean result 'TRUE' if the most recent fetch operation fetched a record successfully, else it will return FALSE.
%NOTFOUND	This works oppositely to %FOUND it will return 'TRUE' if the most recent fetch operation could not able to fetch any record.

%ISOPEN	It returns Boolean result 'TRUE' if the given cursor is already opened, else it returns 'FALSE'
%ROWCOUNT	It returns the numerical value. It gives the actual count of records that got affected by the DML activity.

Example 1: In this example, we are going to see how to declare, open, fetch and close the explicit cursor.

We will project all the employee's name from emp table using a cursor. We will also use cursor attribute to set the loop to fetch all the record from the cursor.

```
1.  DECLARE
2.  CURSOR guru99_det IS SELECT emp_name FROM emp;
3.  lv_emp_name emp.emp_name%type;

4.  BEGIN
5.  OPEN guru99_det;

6.  LOOP
7.  FETCH guru99_det INTO lv_emp_name;
8.  IF guru99_det%NOTFOUND
9.  THEN
10. EXIT;
11. END IF;
12. Dbms_output.put_line ('Employee Fetched: ' || lv_emp_name);
13. END LOOP;
14. Dbms_output.put_line ('Total rows fetched is '||guru99_det%ROWCOUNT);
15. CLOSE guru99_det;
16. END;
17. /
```

Cursor declaration

Opening cursor

Loop to Fetch Cursor Data

Closing cursor

Output:

```
Employee Fetched: BBB
Employee Fetched: XXX
Employee Fetched: YYY
Total rows fetched is 3
```

What is Trigger in PL/SQL?

TRIGGERS are stored programs that are fired by Oracle engine automatically when DML Statements like insert, update, delete are executed on the table or some events occur. The code to be excecuted in

case of a trigger can be defined as per the requirement. You can choose the event upon which the trigger needs to be fired and the timing of the execution. The purpose of trigger is to maintain the integrity of information on the database.

In this tutorial, you will learn-

- Benefits of Triggers
- Types of Triggers in Oracle
- How to Create Trigger
- :NEW and :OLD Clause
- INSTEAD OF Trigger
- Compound Trigger

Benefits of Triggers

Following are the benefits of triggers.

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Types of Triggers in Oracle

Triggers can be classified based on the following parameters.

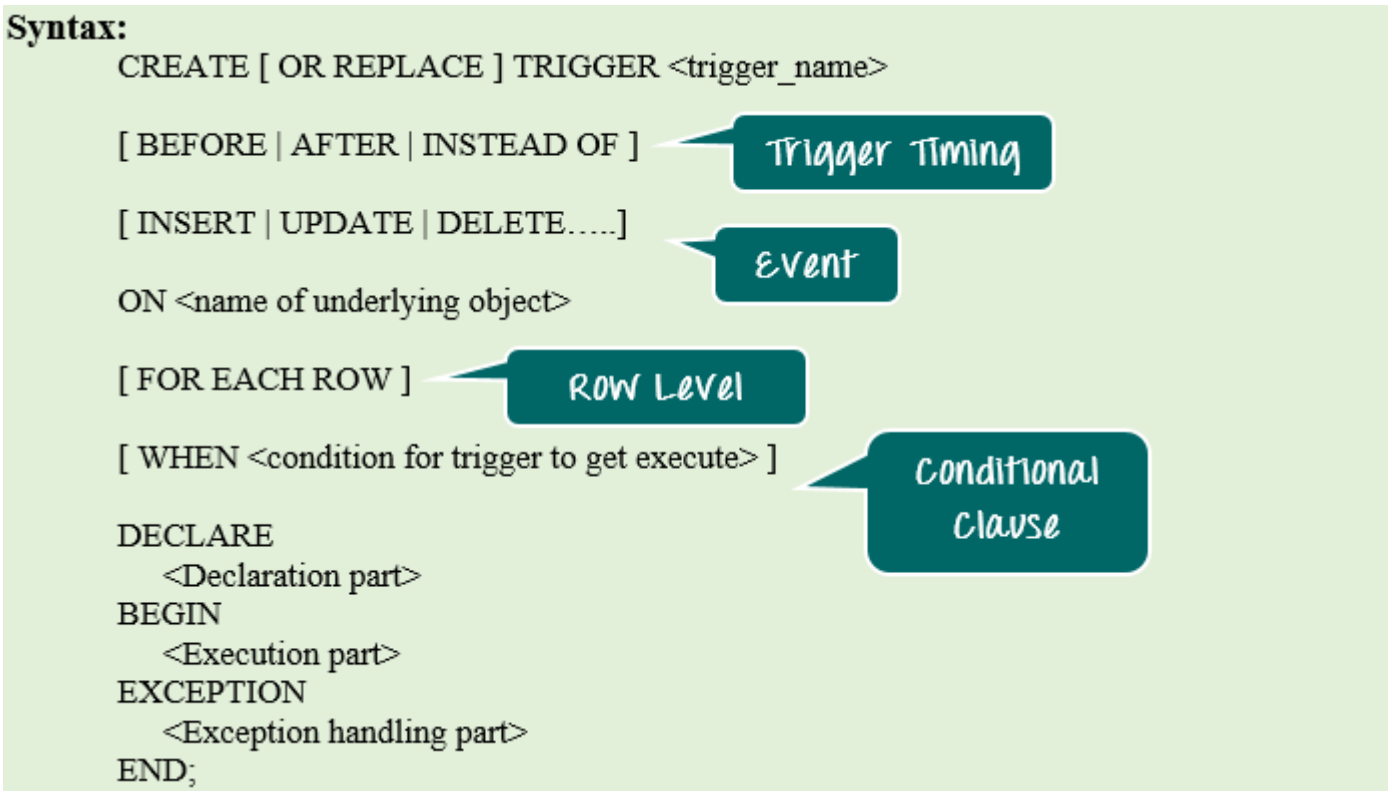
- Classification based on the **timing**
  - BEFORE Trigger: It fires before the specified event has occurred.
  - AFTER Trigger: It fires after the specified event has occurred.
  - INSTEAD OF Trigger: A special type. You will learn more about the further topics. (only for DML)
- Classification based on the **level**
  - STATEMENT level Trigger: It fires one time for the specified event statement.
  - ROW level Trigger: It fires for each record that got affected in the specified event. (only for DML)
- Classification based on the **Event**
  - DML Trigger: It fires when the DML event is specified (INSERT/UPDATE/DELETE)

- DDL Trigger: It fires when the DDL event is specified (CREATE/ALTER)
- DATABASE Trigger: It fires when the database event is specified (LOGON/LOGOFF/STARTUP/SHUTDOWN)

So each trigger is the combination of above parameters.

How to Create Trigger

Below is the syntax for creating a trigger.




```
CREATE [ OR REPLACE ] TRIGGER <trigger_name>

[BEFORE | AFTER | INSTEAD OF ]

[INSERT | UPDATE | DELETE......]

ON<name of underlying object>

[FOR EACH ROW]

[WHEN<condition for trigger to get execute> ]

DECLARE
```

```
<Declaration part>
BEGIN
<Execution part>
EXCEPTION
<Exception handling part>
END;
```

Syntax Explanation:

- The above syntax shows the different optional statements that are present in trigger creation.
- BEFORE/ AFTER will specify the event timings.
- INSERT/UPDATE/LOGON/CREATE/etc. will specify the event for which the trigger needs to be fired.
- ON clause will specify on which object the above-mentioned event is valid. For example, this will be the table name on which the DML event may occur in the case of DML Trigger.
- Command "FOR EACH ROW" will specify the ROW level trigger.
- WHEN clause will specify the additional condition in which the trigger needs to fire.
- The declaration part, execution part, exception handling part is same as that of the other PL/SQL blocks. Declaration part and exception handling part are optional.

:NEW and :OLD Clause

In a row level trigger, the trigger fires for each related row. And sometimes it is required to know the value before and after the DML statement.

Oracle has provided two clauses in the RECORD-level trigger to hold these values. We can use these clauses to refer to the old and new values inside the trigger body.

- :NEW – It holds a new value for the columns of the base table/view during the trigger execution
- :OLD – It holds old value of the columns of the base table/view during the trigger execution

This clause should be used based on the DML event. Below table will specify which clause is valid for which DML statement (INSERT/UPDATE/DELETE).

	INSERT	UPDATE	DELETE

:NEW	VALID	VALID	INVALID. There is no new value in delete case.
:OLD	INVALID. There is no old value in insert case	VALID	VALID

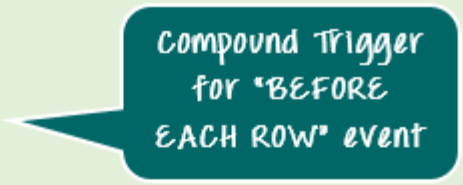
INSTEAD OF Trigger

"INSTEAD OF trigger" is the special type of trigger. It is used only in DML triggers. It is used when any DML event is going to occur on the complex view.

Consider an example in which a view is made from 3 base tables. When any DML event is issued over this view, that will become invalid because the data is taken from 3 different tables. So in this INSTEAD OF trigger is used. The INSTEAD OF trigger is used to modify the base tables directly instead of modifying the view for the given event.

Example 1: In this example, we are going to create a trigger to auto-populate the salary column with the default value 5000.

```
1.  CREATE TRIGGER emp_trig
2.  FOR INSERT
3.  ON emp
4.  COMPOUND TRIGGER
5.  BEFORE EACH ROW IS
6.  BEGIN
7.  :new.salary:=5000;
8.  END BEFORE EACH ROW;
9.  END emp_trig;
10. /
```



Output:

Trigger created.

```
11. BEGIN
12. INSERT INTO EMP VALUES (1004, 'CCC', 15000, 'AAA',30);
13. COMMIT;
14. END;
15. /
```

Output:

PL/SQL procedure successfully completed

```
16. SELECT * FROM emp WHERE emp_no=1004;
```

Output:

EMP_NAME	EMP_NO	SALARY	MANAGER	DEPT_NO
CCC	1004	5000	AAA	30

```
CREATE TRIGGER emp_trig
FOR INSERT
ON emp
COMPOUND TRIGGER
BEFORE EACH ROW IS
BEGIN
:new.salary:=5000;
END BEFORE EACH ROW;
END emp_trig;
/
BEGIN
INSERT INTO EMP VALUES(1004,'CCC',15000,'AAA',30);
COMMIT;
END;
/
```

What is Package in Oracle?

PL/SQL package is a logical grouping of a related subprogram (procedure/function) into a single element. A Package is compiled and stored as a database object that can be used later.

In this tutorial, you will learn-

- Components of Packages
- Package Specification
- Package Body
- Referring Package Elements
- Create Package in PL/SQL
- Forward Declarations
- Cursors Usage in Package
- Overloading
- Dependency in Packages
- Package Information
- UTL FILE – An Overview

Components of Packages

PL/SQL package has two components.

- Package Specification
- Package Body

Package Specification

Package specification consists of a declaration of all the public variables, cursors, objects, procedures, functions, and exception.

Below are few characteristics of the Package specification.

- The elements which are all declared in the specification can be accessed from outside of the package. Such elements are known as a public element.
- The package specification is a standalone element that means it can exist alone without package body.
- Whenever a package has referred an instance of the package is created for that particular session.
- After the instance is created for a session, all the package elements that are initiated in that instance are valid until the end of the session.

Syntax

```
CREATE [OR REPLACE] PACKAGE <package_name>
IS
<sub_program and public element declaration>
.
.
```

```
END <package name>
```

The above syntax shows the creation of package specification.

Package Body

It consists of the definition of all the elements that are present in the package specification. It can also have a definition of elements that are not declared in the specification, these elements are called private elements and can be called only from inside the package.

Below are characteristics of a package body.

- It should contain definitions for all the subprograms/cursors that have been declared in the specification.
- It can also have more subprograms or other elements that are not declared in specification. These are called private elements.
- It is a dependable object, and it depends on package specification.
- The state of the package body becomes 'Invalid' whenever the specification is compiled. Therefore, it needs to be recompiled each time after the compilation of specification.
- The private elements should be defined first before they are used in the package body.
- The first part of the package is the global declaration part. This includes variables, cursors and private elements (forward declaration) that is visible to the entire package.
- The last part of the package is Package initialization part that executes one time whenever a package is referred first time in the session.

Syntax:

```
CREATE [OR REPLACE] PACKAGE BODY <package_name>
IS
<global_declaration part>
<Private element definition>
<sub_program and public element definition>
.
<Package Initialization>
END <package_name>
```

- The above syntax shows the creation of package body.

Now we are going to see how to refer package elements in the program.

Referring Package Elements

Once the elements are declared and defined in the package, we need to refer the elements to use them.

All the public elements of the package can be referred by calling the package name followed by the element name separated by period i.e. '<package_name>.<element_name>'.

The public variable of the package can also be used in the same way to assign and fetch values from them i.e. '<package_name>.<variable_name>'.

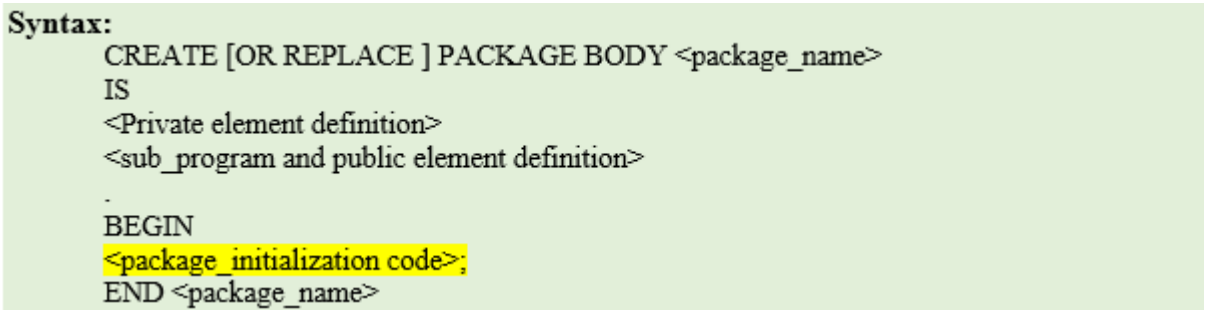
Create Package in PL/SQL

In PL/SQL whenever a package is referred/called in a session a new instance will be created for that package.

Oracle provides a facility to initialize package elements or to perform any activity at the time of this instance creation through 'Package Initialization'.

This is nothing but an execution block that is written in the package body after defining all the package elements. This block will be executed whenever a package is referred for the first time in the session.

Syntax




```
CREATE [OR REPLACE] PACKAGE BODY <package_name>
IS
<Private element definition>
<sub_program and public element definition>
.
BEGINE
<Package Initialization>
END <package_name>
```

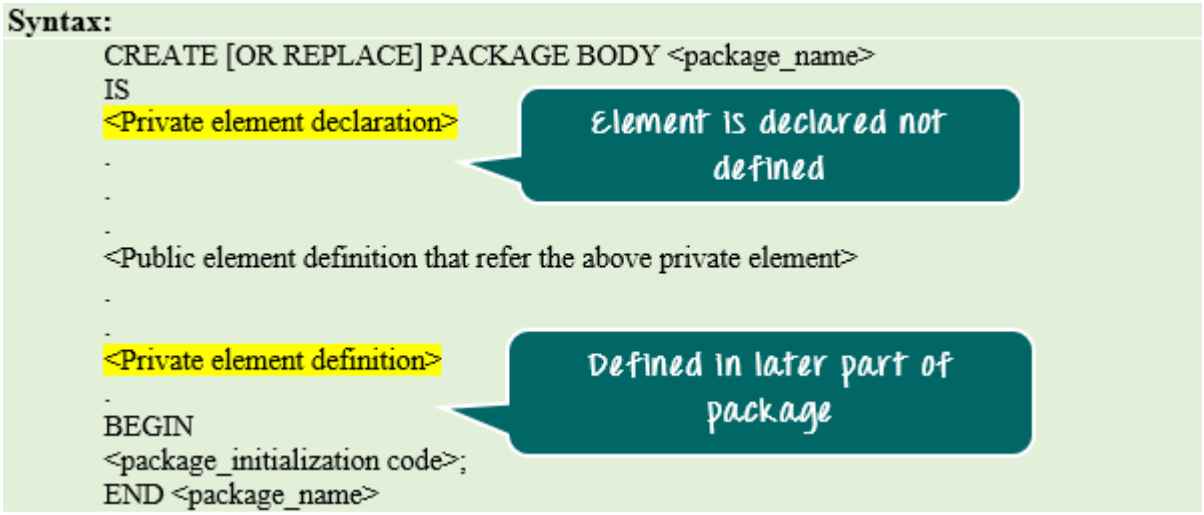
- The above syntax shows the definition of package initialization in the package body.

Forward Declarations

Forward declaration/reference in the package is nothing but declaring the private elements separately and defining it in the later part of the package body.

Private elements can be referred only if it is already declared in the package body. For this reason, forward declaration is used. But it is rather unusual to use because in most of the time private elements are declared and defined in the first part of the package body.

Forward declaration is an option provided by Oracle, it is not mandatory and using and not using is up to programmer's requirement.



Unit III:Transaction Management

3.1 Transaction concept

! A *transaction* is a *unit* of program execution that accesses and possibly updates various data items.
! A transaction must see a consistent database.
! During transaction execution the database may be inconsistent.
! When the transaction is committed, the database must be consistent.
! Two main issues to deal with:
! Failures of various kinds, such as hardware failures and system crashes
! Concurrent execution of multiple transactions

ACID Properties

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  ! That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Example of Fund Transfer

- Transaction to transfer $50 from account *A* to account *B*:

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

- Consistency requirement – the sum of *A* and *B* is unchanged by the execution of the transaction.
- Atomicity requirement — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

Example of Fund Transfer

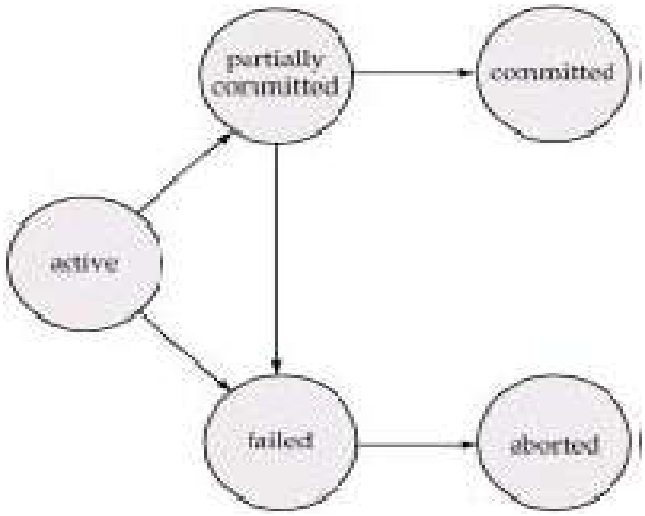
- Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.
- Isolation requirement — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).

Can be ensured trivially by running transactions ***serially***, that is one after the other. However, executing multiple transactions concurrently has significant benefits, as we will see.

Transaction State

- **Active,** the initial state; the transaction stays in this state while it is executing
- **Partially committed,** after the final statement has been executed.
- **Failed,** after the discovery that normal execution can no longer proceed.
- **Aborted,** after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - ! restart the transaction – only if no internal logical error
  - ! kill the transaction
- **Committed,** after *successful completion*.

Transaction State



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - ! **increased processor and disk utilization**, leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk
  - ! **reduced average response time** for transactions: short transactions need not wait behind long ones.
- *Concurrency control schemes* – mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
  - ! Will study in Chapter 14, after studying notion of correctness of concurrent executions.

Schedules

- *Schedules* – sequences that indicate the chronological order in which instructions of concurrent transactions are executed
  - ! a schedule for a set of transactions must consist of all instructions of those transactions

! must preserve the order in which the instructions appear in each individual transaction.

Example Schedules

Let T_1 transfer $50 from A to B, and T_2 transfer 10% of the balance from A to B. The following is a serial schedule (Schedule 1 in the text), in which T_1 is followed by T_2.

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Example Schedule

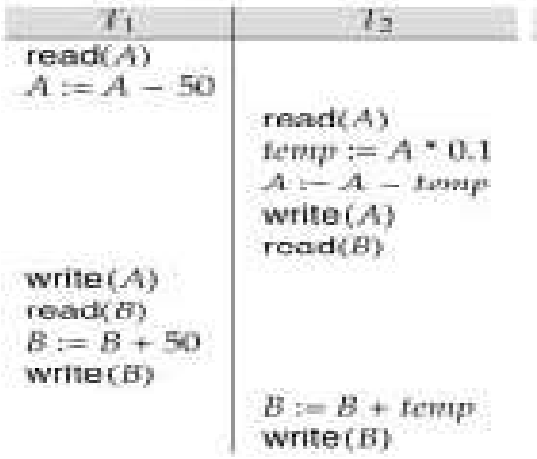
Let T_1 and T_2 be the transactions defined previously. The following schedule (Schedule 3 in the text) is not a serial schedule, but it is *equivalent* to Schedule 1.

In both Schedule 1 and 3, the sum A + B is preserved.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Example Schedules

The following concurrent schedule (Schedule 4 in the text) does not preserve the value of the the sum $A + B$.



Serializability

- Basic Assumption – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. conflict serializability
2. view serializability

- We ignore operations other than **read** and **write** instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only **read** and **write** instructions.

Conflict Serializability

- Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.

1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
3. I_i = **write**(Q), I_j = **read**(Q). They conflict
4. I_i = **write**(Q), I_j = **write**(Q). They conflict

- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them. If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule
- Example of a schedule that is not conflict serializable:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial

schedule < T_4, T_3 >.

Conflict Serializability

- Schedule 3 below can be transformed into Schedule 1, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

View Serializability

- Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met:

1. For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i must, in schedule S', also read the initial value of Q.
2. For each data item Q if transaction T_i executes **read**(Q) in schedule S, and that value was produced by transaction T_j (if any), then transaction T_i must in schedule S' also read the value of Q that was produced by transaction T_j.
3. For each data item Q, the transaction (if any) that performs the final **write**(Q) operation in schedule S must perform the final **write**(Q) operation in schedule S'.

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serializability

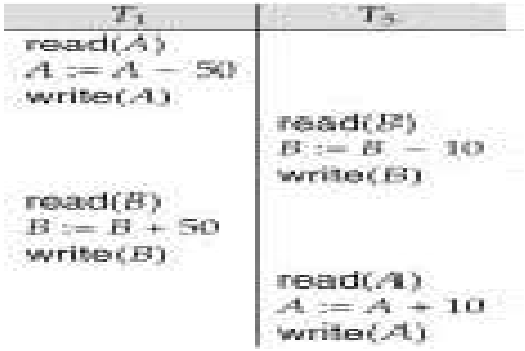
- A schedule S is **view serializable** it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Schedule 9 (from text) — a schedule which is view-serializable but *not* conflict serializable.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- Every view serializable schedule that is not conflict serializable has **blind writes.**

Other Notions of Serializability

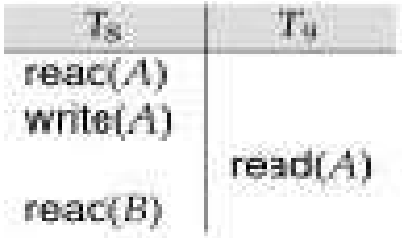
- Schedule 8 (from text) given below produces same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent or view equivalent to it.



- Determining such equivalence requires analysis of operations other than read and write.

Recoverability

- **Recoverable schedule** — if a transaction T_j reads a data items previously written by a transaction T_i , the commit operation of T_i appears before the commit operation of T_j.
- The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read

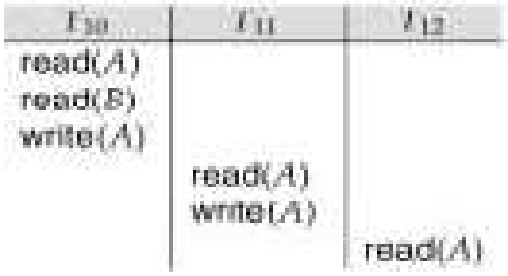


- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence database must ensure that schedules are recoverable.

Need to address the effect of transaction failures on concurrently running transactions.

Recoverability

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)



If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- Can lead to the undoing of a significant amount of work

Recoverability

- **Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable

- It is desirable to restrict the schedules to those that are cascadeless

Implementation of Isolation

- Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency..
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:

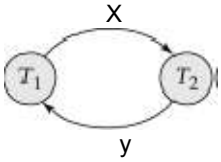
! **Commit work** commits current transaction and begins a new one.
! **Rollback work** causes current transaction to abort.

- Levels of consistency specified by SQL-92:

! **Serializable** — default
! **Repeatable read**
! **Read committed**
! **Read uncommitted**

Testing for Serializability

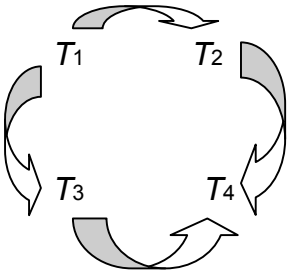
- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example 1**



Example Schedule (Schedule A)

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				

Precedence Graph for Schedule A



Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n_2 time, where n is the number of vertices in the graph. (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph. This is a linear order consistent with the partial order of the graph.

For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$.

Test for View Serializability

- The precedence graph test for conflict serializability must be modified to apply to a test for view serializability.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems. Thus existence of an efficient algorithm is unlikely.

However practical algorithms that just check some *sufficient conditions* for view serializability can still be used.

Unit IV: Concurrency Control & Recovery System

Concurrency control is the procedure in DBMS for managing simultaneous operations without conflicting with each another. Concurrent access is quite easy if all users are just reading data. There is no way they can interfere with one another. Though for any practical database, would have a mix of reading and WRITE operations and hence the concurrency is a challenge.

Potential problems of Concurrency

Here, are some issues which you will likely to face while using the Concurrency Control method:

- **Lost Updates** occur when multiple transactions select the same row and update the row based on the value selected
- Uncommitted dependency issues occur when the second transaction selects a row which is updated by another transaction (**dirty read**)
- **Non-Repeatable Read** occurs when a second transaction is trying to access the same row several times and reads different data each time.
- **Incorrect Summary issue** occurs when one transaction takes summary over the value of all the instances of a repeated data-item, and second transaction update few instances of that specific data-item. In that situation, the resulting summary does not reflect a correct result.

4.1 Lock Based Protocol

A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks help synchronize access to the database items by concurrent transactions.

All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.

Binary Locks: A Binary lock on a data item can either locked or unlocked states.

Shared/exclusive: This type of locking mechanism separates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is called an exclusive lock.

1. Shared Lock (S):

A shared lock is also called a Read-only lock. With the shared lock, the data item can be shared between transactions. This is because you will never have permission to update data on the data item.

For example, consider a case where two transactions are reading the account balance of a person. The database will let them read by placing a shared lock. However, if another transaction wants to update that account's balance, shared lock prevent it until the reading process is over.

2. Exclusive Lock (X):

With the Exclusive Lock, a data item can be read as well as written. This is exclusive and can't be held concurrently on the same data item. X-lock is requested using lock-x instruction. Transactions may unlock the data item after finishing the 'write' operation.

For example, when a transaction needs to update the account balance of a person. You can allows this transaction by placing X lock on it. Therefore, when the second transaction wants to read or write, exclusive lock prevent this operation.

4.1.2 Granting of Locks
Lock Compatibility Matrix –

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.
- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive(X) on the item no other transaction may hold any lock on the item.

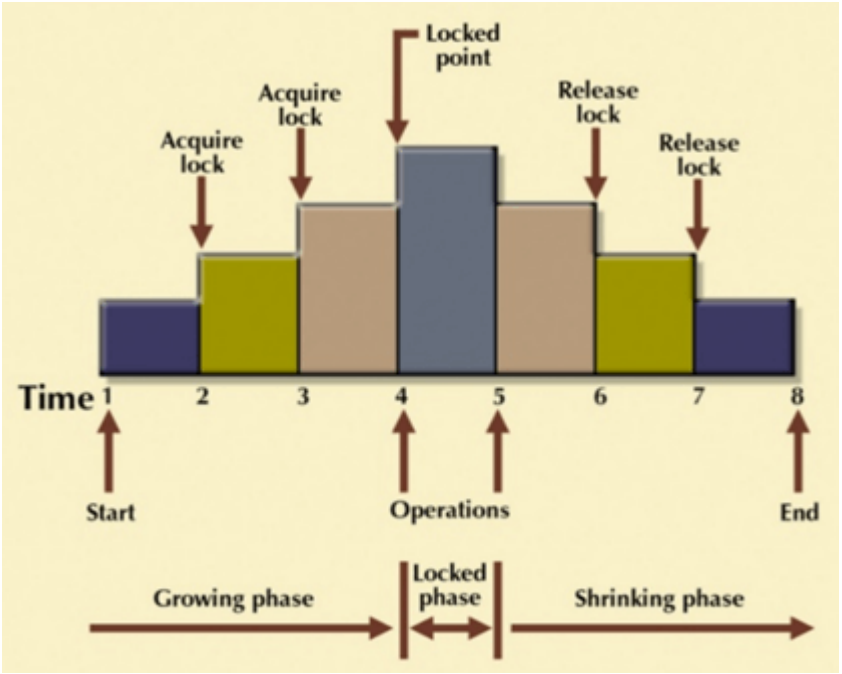
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. Then the lock is granted.

4.1.3 Two Phase Locking Protocol

Two-Phase locking protocol which is also known as a 2PL protocol. It is also called P2L. In this type of locking protocol, the transaction should acquire a lock after it releases one of its locks.

This locking protocol divides the execution phase of a transaction into three different parts.

- In the first phase, when the transaction begins to execute, it requires permission for the locks it needs.
- The second part is where the transaction obtains all the locks. When a transaction releases its first lock, the third phase starts.
- In this third phase, the transaction cannot demand any new locks. Instead, it only releases the acquired locks.



The Two-Phase Locking protocol allows each transaction to make a lock or unlock request in two steps:

- **Growing Phase**: In this phase transaction may obtain locks but may not release any locks.
- **Shrinking Phase**: In this phase, a transaction may release locks but not obtain any new lock

It is true that the 2PL protocol offers serializability. However, it does not ensure that deadlocks do not happen.

4.2 Timestamp Based Protocol

The timestamp-based algorithm uses a timestamp to serialize the execution of concurrent transactions. This protocol ensures that every conflicting read and write operations are executed in timestamp order. The protocol uses the **System Time or Logical Count as** a Timestamp.

The older transaction is always given priority in this method. It uses system time to determine the time stamp of the transaction. This is the most commonly used concurrency protocol.

Lock-based protocols help you to manage the order between the conflicting transactions when they will execute. Timestamp-based protocols manage conflicts as soon as an operation is created.

Example:

```
Suppose there are there transactions T1, T2, and T3.
T1 has entered the system at time 0010
T2 has entered the system at 0020
T3 has entered the system at 0030
Priority will be given to transaction T1, then transaction T2 and lastly
Transaction T3.
```

Timestamp Ordering Protocol –

The main idea for this protocol is to order the transactions based on their Timestamps. A schedule in which the transactions participate is then serializable and the only *equivalent serial schedule permitted* has the transactions in the order of their Timestamp Values. Stating simply, the schedule is equivalent to the particular *Serial Order* corresponding to the *order of the Transaction timestamps*. Algorithm must ensure that, for each items accessed by *Conflicting Operations* in the schedule, the order in which the item is accessed does not violate the ordering. To ensure this, use two Timestamp Values relating to each database item **X**.

- **W_TS(X)** is the largest timestamp of any transaction that executed **write(X)** successfully.
- **R_TS(X)** is the largest timestamp of any transaction that executed **read(X)** successfully.

Basic Timestamp Ordering –

Every transaction is issued a timestamp based on when it enters the system. Suppose, if an old transaction T_i has timestamp $TS(T_i)$, a new transaction T_j is assigned timestamp $TS(T_j)$ such that **$TS(T_i) < TS(T_j)$**.The protocol manages concurrent execution such that the timestamps determine the serializability order.

The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. Whenever some Transaction *T* tries to issue a R_item(X) or a W_item(X), the Basic TO algorithm compares the timestamp of *T* with **R_TS(X) & W_TS(X)** to ensure that the Timestamp order is not violated. This describe the Basic TO protocol in following two cases.

1. Whenever a Transaction *T* issues a **W_item(X)** operation, check the following conditions:

    - If ***R_TS(X) > TS(T)*** or if ***W_TS(X) > TS(T)***, then abort and rollback T and reject the operation. else,
    - Execute W_item(X) operation of T and set W_TS(X) to TS(T).
2. Whenever a Transaction *T* issues a **R_item(X)** operation, check the following conditions:

    - If ***W_TS(X) > TS(T)***, then abort and reject T and reject the operation, else
    - If W_TS(X) <= TS(T), then execute the R_item(X) operation of T and set R_TS(X) to the larger of TS(T) and current R_TS(X).

Strict Timestamp Ordering –
A variation of Basic TO is called **Strict TO** ensures that the schedules are both Strict and Conflict Serializable. In this variation, a Transaction T that issues a R_item(X) or W_item(X) such that TS(T) > W_TS(X) has its read or write operation delayed until the Transaction **T**' that wrote the values of X has committed or aborted.

4.2.3 Thomas's Write Rule

Thomas Write Rule does not enforce *Conflict Serializablity* but rejects fewer Write Operations by modifying the check Operations for W_item(X)

1. If **R_TS(X) > TS(T)**, then abort and rollback T and reject the operation.
2. If **W_TS(X) > TS(T)**, then don't execute the Write Operation and continue processing. This is a case of *Outdated or Obsolete Writes*. Remember, outdated writes are ignored in Thomas Write Rule but a Transaction following Basic TO protocol will abort such a Transaction.
3. If neither the condition in 1 or 2 occurs, then and only then execute the W_item(X) operation of T and set W_TS(X) to TS(T)

4.3 Validation Based Protocol

Validation Based Protocol in DBMS

Validation Based Protocol is also called Optimistic Concurrency Control Technique. This protocol is used in DBMS (Database Management System) for avoiding concurrency in transactions. It is called optimistic because of the assumption it makes, i.e. very less interference occurs therefore there is no need for checking while the transaction is executed.
In this technique, no checking is done while the transaction is been executed. Until the transaction end is reached updates in the transaction are not applied directly to the database. All updates are applied to local copies of data items kept for transaction. At the end of transaction execution, while execution of transaction, a **validation phase** checks whether any of transaction updates violate serializability.

If there is no violation of serializability the transaction is committed and the database is updated; or else, the transaction is updated and then restarted.
Optimistic Concurrency Control is a three phase protocol. The three phases for validation based protocol:

1. **Read Phase:**
   Values of committed data items from the database can be read by a transaction. Updates are only applied to local data versions.
2. **Validation Phase:**
   Checking is performed to make sure that there is no violation of serializability when the transaction updates are applied to database.
3. **Write Phase:**
   On the success of validation phase, the transaction updates are applied to the database, otherwise, the updates are discarded and the transaction is slowed down.

The idea behind optimistic concurrency is to do all the checks at once; hence transaction execution proceeds with a minimum of overhead until the validation phase is reached. If there is not much interference among transactions most of them will have successful validation, otherwise, results will be discarded and restarted later. These circumstances are not much favorable for optimization techniques, since, the assumption of less interference is not satisfied.

4.4 Deadlock Handling

Deadlock **occurs when each transaction T in a schedule of *two or more* transaction waiting for some item locked by some other transaction T' in the set. Thus, both end up in a deadlock situation, waiting for the other to release the lock on the item.**

4.4.1 Deadlock Prevention

Deadlock Prevention Schemes based on Timestamp…

As discussed, Timestamps are unique identifiers assigned to each transaction. They are based on the order in which Transactions are started. Say if $\mathbf{T_1}$ starts before $\mathbf{T_2}$ then $\mathbf{TS(T_1)}$ will be less then(<) $\mathbf{TS(T_2)}$.
There are two schemes to prevent deadlock called *wound-wait* and *wait-die*. Say there are two transactions $\mathbf{T_i}$ and $\mathbf{T_j}$, now say $\mathbf{T_i}$ tries to lock an item *X* but item *X* is already locked by some $\mathbf{T_j}$, now in such a conflicting situation the two schemes which prevents deadlock. We'll use this context shortly.

- **Wait_Die :** An older transaction is allowed to wait for a younger transaction, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted.
  From the context above, if $\mathbf{TS(T_i) < TS(T_j)}$, then ($T_i$ older than T_j) T_i is allowed to wait; otherwise *abort* T_i (T_i younger than T_j) *and restart it later with the same timestamp.*
- **Wound_Wait :** It is just the opposite of the Wait_Die technique. Here, a younger transaction is allowed to wait for an older one, whereas if an older transaction requests an item held by the younger transaction, we preempt the younger transaction by aborting it.
  From the context above, if $\mathbf{TS(T_i) < TS(T_j)}$, then ($T_i$ older than T_j) T_j is aborted (i.e., $T_i$wounds T_j) and restarts it later with the same Timestamp; *otherwise (T_i younger than T_j) T_i is allowed to wait.*

Starvation: **One problem that may occur when we use locking is starvation which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair, giving priority to some transactions over others.**

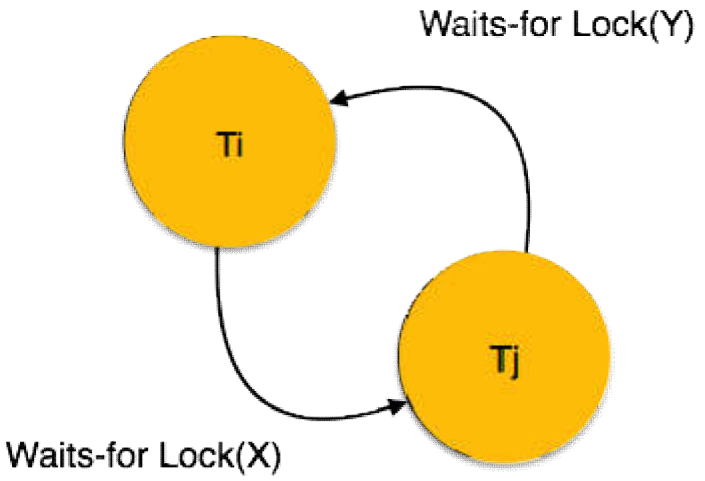
Deadlock Avoidance

Aborting a transaction is not always a practical approach. Instead, deadlock avoidance mechanisms can be used to detect any deadlock situation in advance. Methods like "wait-for graph" are available but they are suitable for only those systems where transactions are lightweight having fewer instances of resource. In a bulky system, deadlock prevention techniques may work well.

Wait-for Graph

This is a simple method available to track if any deadlock situation may arise. For each transaction entering into the system, a node is created. When a transaction T_i requests for a lock on an item, say X, which is held by some other transaction T_j, a directed edge is created from T_i to T_j. If T_j releases item X, the edge between them is dropped and T_i locks the data item.

The system maintains this wait-for graph for every transaction waiting for some data items held by others. The system keeps checking if there's any cycle in the graph.



Here, we can use any of the two following approaches −

- First, do not allow any request for an item, which is already locked by another transaction. This is not always feasible and may cause starvation, where a transaction indefinitely waits for a data item and can never acquire it.
- The second option is to roll back one of the transactions. It is not always feasible to roll back the younger transaction, as it may be important than the older one. With the help of some relative algorithm, a transaction is chosen, which is to be aborted. This transaction is known as the **victim** and the process is known as **victim selection**

Deadlock Detection

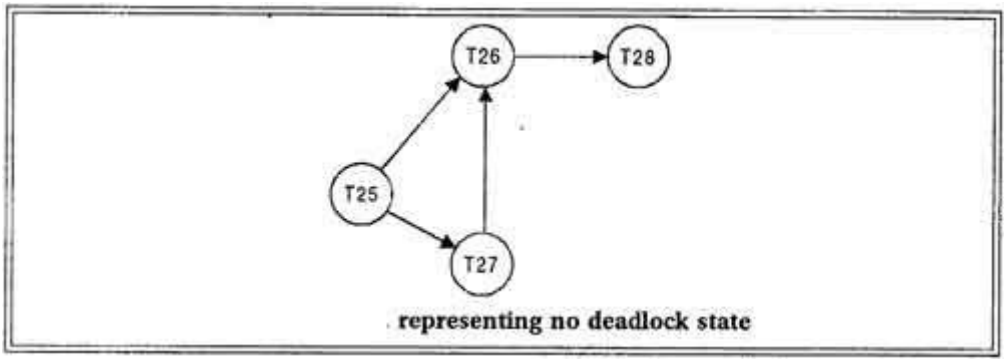
A simple way to detect a state of deadlock is with the help of wait-for graph. This graph is constructed and maintained by the system. One node is created in the wait-for graph for each transaction that is currently executing. Whenever a transaction Ti is waiting to lock an item X that is currently locked by a transaction Tj, a directed edge (Ti->Tj).is created in the wait-for graph. When Tj releases the lock(s) on the items that Ti was waiting for, the directed edge is dropped from the wait-for graph. We have a state of deadlock if and only if the wait-for graph has a cycle. Then each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

To illustrate these concepts, consider the following wait-for graph in figure. Here:

Transaction T25 is waiting for transactions T26 and T27.

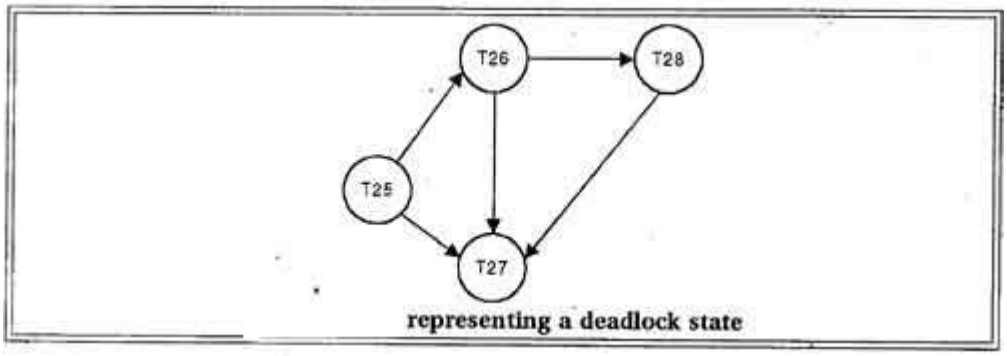
Transactions T27 is waiting for transaction T26.

Transaction T26 is waiting for transaction T28.



This wait-for graph has no cycle, so there is no deadlock state.

Suppose now that transaction T28 is requesting an item held by T27. Then the edge T28 ----------->T27 is added to the wait -for graph, resulting in a new system state as shown in figure.



This time the graph contains the cycle.

T26------>T28------->T27----------->T26

It means that transactions T26, T27 and T28 are all deadlocked.

Invoking the deadlock detection algorithm

The invoking of deadlock detection algorithm depends on two factors:

• How often does a deadlock occur?

• How many transactions will be affected by the deadlock?

If deadlocks occur frequently, then the detection algorithm should be invoked more frequently than usual. Data items allocated to deadlocked transactions will be unavailable to other transactions until the deadlock can be broken. In the worst case, we would invoke the detection algorithm every time a request for allocation could not be granted immediately.

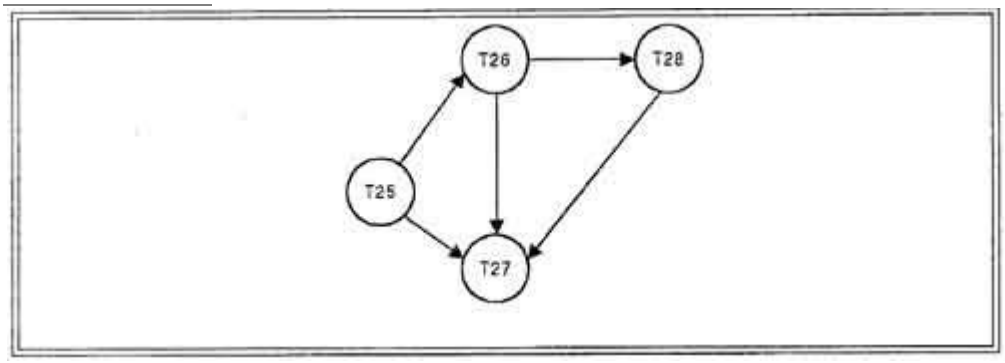
Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Choosing which transaction to abort is known as Victim Selection.

Choice of Deadlock victim

In below wait-for graph transactions T26, T28 and T27 are deadlocked. In order to remove deadlock one of the transaction out of these three transactions must be roll backed.

We should roll back those transactions that will incur the minimum cost. When a deadlock is detected, the choice of which transaction to abort can be made using following criteria:



• The transaction which have the fewest locks

• The transaction that has done the least work

• The transaction that is farthest from completion

Rollback

Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back. The simplest solution is a total rollback; Abort the transaction and then restart it. However it is more effective to roll back the transaction only as far as necessary to break the deadlock. But this method requires the system to maintain additional information about the state of all the running system.

5.1 Failure Classification

To see where the problem has occurred, we generalize a failure into various categories, as follows −

Transaction failure

A transaction has to abort when it fails to execute or when it reaches a point from where it can't go any further. This is called transaction failure where only a few transactions or processes are hurt.

Reasons for a transaction failure could be −

- **Logical errors** − Where a transaction cannot complete because it has some code error or any internal error condition.
- **System errors** − Where the database system itself terminates an active transaction because the DBMS is not able to execute it, or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability, the system aborts an active transaction.

System Crash

There are problems − external to the system − that may cause the system to stop abruptly and cause the system to crash. For example, interruptions in power supply may cause the failure of underlying hardware or software failure.

Examples may include operating system errors.

Disk Failure

In early days of technology evolution, it was a common problem where hard-disk drives or storage drives used to fail frequently.

Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or a part of disk storage.

Storage Structure

We have already described the storage system. In brief, the storage structure can be divided into two categories −

- **Volatile storage** − As the name suggests, a volatile storage cannot survive system crashes. Volatile storage devices are placed very close to the CPU; normally they are embedded onto the chipset itself. For example, main memory and cache memory are examples of volatile storage. They are fast but can store only a small amount of information.
- **Non-volatile storage** − These memories are made to survive system crashes. They are huge in data storage capacity, but slower in accessibility. Examples may include hard-disks, magnetic tapes, flash memory, and non-volatile (battery backed up) RAM.

Recovery and Atomicity

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of

various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

When a DBMS recovers from a crash, it should maintain the following −

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −

- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

Log-based Recovery

Log is a sequence of records, which maintains the records of actions performed by a transaction. It is important that the logs are written prior to the actual modification and stored on a stable storage media, which is failsafe.

Log-based recovery works as follows −

- The log file is kept on a stable storage media.
- When a transaction enters the system and starts execution, it writes a log about it.

<T_n, Start>

- When the transaction modifies an item X, it write logs as follows −

<T_n, X, V_1, V_2>

It reads T_n has changed the value of X, from V_1 to V_2.

- When the transaction finishes, it logs −

<T_n, commit>

The database can be modified using two approaches −

- **Deferred database modification** − All logs are written on to the stable storage and the database is updated when a transaction commits.
- **Immediate database modification** − Each log follows an actual database modification. That is, the database is modified immediately after every operation.

Recovery with Concurrent Transactions

When more than one transaction are being executed in parallel, the logs are interleaved. At the time of recovery, it would become hard for the recovery system

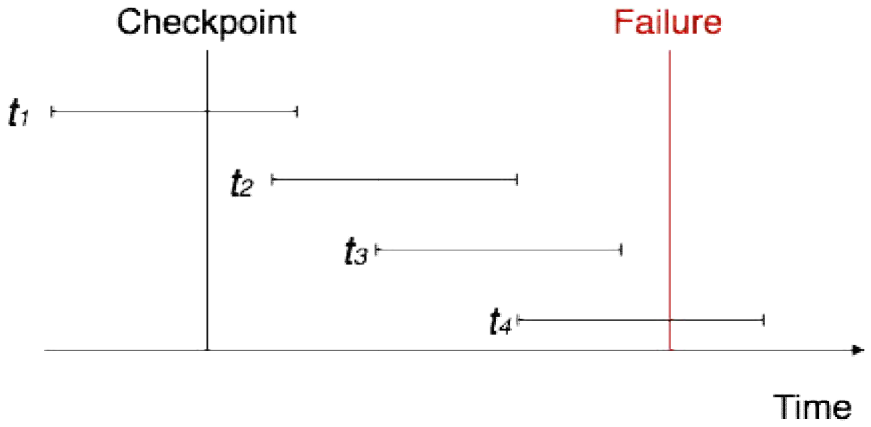
to backtrack all logs, and then start recovering. To ease this situation, most modern DBMS use the concept of 'checkpoints'.

Checkpoint

Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. As time passes, the log file may grow too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

Restart Recovery

When a system with concurrent transactions crashes and recovers, it behaves in the following manner −



- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an undo-list and a redo-list.
- If the recovery system sees a log with <T_n, Start> and <T_n, Commit> or just <T_n, Commit>, it puts the transaction in the redo-list.
- If the recovery system sees a log with <T_n, Start> but no commit or abort log found, it puts the transaction in undo-list.

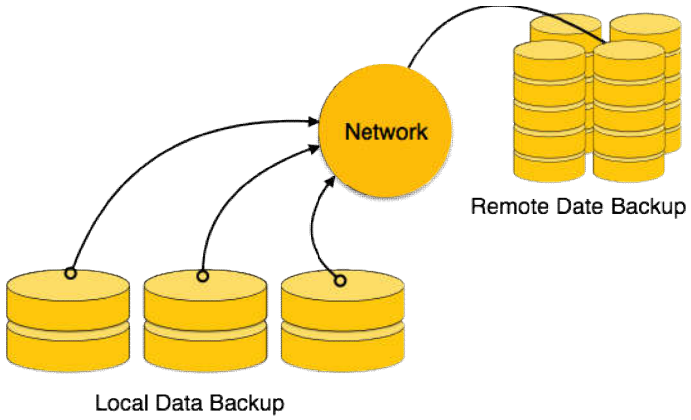
All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

5.4.1 Transaction Rollback

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

Remote Backup

Remote backup provides a sense of security in case the primary location where the database is located gets destroyed. Remote backup can be offline or real-time or online. In case it is offline, it is maintained manually.



Online backup systems are more real-time and lifesavers for database administrators and investors. An online backup system is a mechanism where every bit of the real-time data is backed up simultaneously at two distant places. One of them is directly connected to the system and the other one is kept at a remote place as backup.

As soon as the primary database storage fails, the backup system senses the failure and switches the user system to the remote storage. Sometimes this is so instant that the users can’t even realize a failure.